VNIVERSITAS BALEARICA

Universitat
de les Illes Balears

DOCTORAL THESIS
2017

ADDING FAULT TOLERANCE TO A FLEXIBLE REAL-TIME ETHERNET NETWORK FOR EMBEDDED SYSTEMS

David Gessner

Universitat
de les Illes Balears

DOCTORAL THESIS
2017

Doctoral Programme of Information and Communications Technology

ADDING FAULT TOLERANCE TO A FLEXIBLE REAL-TIME ETHERNET NETWORK FOR EMBEDDED SYSTEMS

David Gessner

Thesis Supervisor: Dr. Julián Proenza
Thesis Supervisor: Dr. Manuel Barranco
Thesis Tutor: Dr. Julián Proenza

Doctor by the Universitat de les Illes Balears

Statement of Authorship

This thesis has been submitted to the *Escola de Doctorat*, *Universitat de les Illes Balears*, in fulfilment of the requirements for the degree of *Doctor en Tecnologías de la Información y las Comunicaciones*. I hereby declare that, except where specific reference is made to the work of others, the content of this dissertation is entirely my own work, describes my own research and has not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university.

David Gessner

Palma de Mallorca, September 2017

Supervisors' Agreement

Julián Proenza, Ph.D. in Computer Science and *Profesor Titular de Universidad* at the *Department of Mathematics and Computer Science*, *Universitat de les Illes Balears* and Manuel Barranco, Ph.D. in Computer Science and *Profesor Contratado Doctor* of the same department

DECLARE

that the thesis titled *Adding fault tolerance to a flexible real-time Ethernet network for embedded systems*, presented by David Gessner to obtain the degree of *Doctor en Tecnologías de la Información y las Comunicaciones*, has been completed under our supervision and meets the requirements to opt for an International Doctorate.

For all intents and purposes, we hereby sign this document.

Dr. Julián Proenza Arenas

Dr. Manuel Alejandro Barranco González

Palma de Mallorca, September 2017

Abstract

Distributed embedded systems consist of a set of computing nodes that cooperate to achieve a common goal within a larger physical system. In modern societies they are ubiquitous. Among other places we find them in aircraft, cars, trains, factories, chemical plants, power plants, human-made satellites, space probes, household appliances, and medical equipment.

Wherever we find a distributed embedded system, and whatever it does, its nodes must collaborate to achieve a common goal. For example, the nodes in a car collaborate to ensure a safe and comfortable journey and the nodes in a factory collaborate to manufacture a given product. Collaboration calls for a means of communication: a communication subsystem or network that allows nodes to exchange messages among themselves.

These messages must be exchanged on time. Distributed embedded systems interact with physical environments that progress inexorably and do not wait. This means that the network must support *real-time* communication; that is, messages must be exchanged before specified instants of time, called deadlines.

For this task, Ethernet has traditionally been unsuitable. It could not provide real-time guarantees. But this has changed thanks to the development of new Ethernet-based protocols. Among them, the ones based on the Flexible Time-Triggered (FTT) communication paradigm stand out because of their flexibility: they can guarantee the timely delivery of messages even if the real-time parameters of messages change at runtime. For instance, deadlines, transmission times, periods, and minimum interarrival times of messages are all allowed to change at runtime.

Despite their advantages, current Ethernet versions of FTT can only meet real-time requirements in the absence of faults. But this is unacceptable in many of the domains where distributed embedded systems are deployed, such as critical subsystems in avionics or cars. Faults do occur and they have to be dealt with for the system to be reliable.

In this dissertation we deal with such faults. We show that FTT, using Ethernet

as the underlying technology, can be made fault tolerant, and that this can be done without sacrificing the real-time guarantees or flexibility that the FTT paradigm provides. Why is this important? Because it opens up the possibility of using high-bandwidth and inexpensive Ethernet in future distributed embedded systems that must simultaneously be fault-tolerant, flexible, and real-time.

As to how we show that FTT over Ethernet can be made fault tolerant while preserving its flexibility and hard real-time guarantees, we present a novel communication subsystem that proves the point. In particular, we design such a communication subsystem and show its feasibility by means of a series of proof-of-concept prototypes.

The novel communication subsystem, called *Flexible Time-Triggered Replicated Star for Ethernet* (FTTRS), is based on *Hard Real-Time Ethernet Switching* (HaRTES), which is a switched Ethernet implementation of the FTT paradigm. Specifically, FTTRS addresses the vulnerabilities of HaRTES: it eliminates the single points of failure of HaRTES by replicating critical components, it ensures the replica determinism of these replicated components, it restricts the failure semantics of the components as necessary, and it makes message exchanges capable of tolerating transient channel faults, all the while maintaining the flexibility and real-time guarantees of the FTT paradigm.

Abstract in Spanish

Un sistema empotrado distribuido consiste en un conjunto de nodos computacionales que cooperan para alcanzar un objetivo común dentro de un sistema físico más grande. En las sociedades modernas podemos encontrar estos sistemas por todos lados: en aviones, coches y trenes; en fábricas, plantas químicas y centrales eléctricas; en satélites artificiales y sondas espaciales; en aparatos domésticos y en equipos médicos; y en muchos otros sistemas.

Sea donde sea que los encontremos, e independientemente de lo que hagan, los nodos de un sistema empotrado distribuido deben colaborar para alcanzar un objetivo común. Así por ejemplo los nodos de un coche colaboran para asegurar un viaje seguro y agradable y los nodos en una fábrica colaboran para fabricar un producto. Como la colaboración requiere de comunicación, los sistemas empotrados distribuidos necesitan un subsistema de comunicación o red que permita el intercambio de mensajes entre nodos.

Estos mensajes deben ser intercambiados a tiempo ya que los sistemas empotrados distribuidos interactúan con entornos físicos que avanzan de forma inexorable y no esperan. Eso significa que la red debe dar soporte a la comunicación en tiempo real; es decir, los mensajes deben ser intercambiados antes de instantes de tiempo preestablecidos, llamados plazos temporales.

Para este tipo de comunicación, Ethernet ha sido tradicionalmente inadecuado ya que no podía proporcionar garantías de tiempo real. Pero esto ha cambiado gracias a nuevos protocols basados en Ethernet. Entre ellos destacan por su flexibilidad los que se basan en el paradigma de comunicación FTT (*flexible time-triggered*). La propiedad clave de FTT es que puede garantizar la entrega a tiempo de mensajes incluso si los parámetros de tiempo real cambian durante la ejecución del sistema. Mientras se ejecuta el sistema, tanto los plazos temporales como los tiempos de transmisión, los periodos y los tiempos mínimos entre llegadas de los mensajes pueden cambiar.

A pesar de sus ventajas, las versiones de FTT actuales para Ethernet sólo sa-

tisfacen los requisitos de tiempo real si no hay fallos ni averías de componentes. Desafortunadamente esto hace que sean inadecuadas para muchas aplicaciones en las cuales se usan sistemas empotrados distribuidos, como por ejemplo los subsistemas críticos de aviones y automóviles. Al fin y al cabo, en la realidad, fallos sí que ocurren y hay que tratarlos para que el sistema sea fiable.

En esta tesis demostramos que FTT basado en Ethernet puede ser modificado para ser tolerante a fallos y que esto se puede hacer sin sacrificar las garantías de tiempo real o la flexibilidad que el paradigma FTT proporciona. Esto abre la posibilidad de usar Ethernet, con su bajo coste y alto ancho de banda, en futuros sistemas empotrados distribuidos que deben ser tolerantes a fallos, flexibles y a la vez de tiempo real.

Establecemos la veracidad de que FTT sobre Ethernet puede hacerse tolerante a fallos (preservando su flexibilidad y garantías de tiempo real) mediante un novedoso subsistema de comunicación que lo demuestra. En particular, diseñamos tal subsistema de comunicación y demostramos su factibilidad mediante una serie de prototipos de prueba de concepto.

El nuevo subsistema de comunicación, denominado *Flexible Time-Triggered Replicated Star for Ethernet* (FTTRS), se basa en *Hard Real-Time Ethernet Switching* (HaRTES), que es una implementación en Ethernet conmutado del paradigma FTT. En concreto, FTTRS trata las vulnerabilidades de HaRTES: elimina los puntos únicos de avería de HaRTES mediante la replicación de componentes críticos, garantiza el determinismo de réplica de estos componentes replicados, restringe la semántica de averías de los componentes según sea necesario, y hace que los intercambios de mensaje sean capaces de tolerar fallos transitorios de canal, todo ello manteniendo la flexibilidad y las garantías de tiempo real del paradigma FTT.

Abstract in Catalan

Un sistema encastat distribuït consisteix en un conjunt de nodes computacionals que cooperen per aconseguir un objectiu comú dins d'un sistema físic més gran. En les societats modernes podem trobar aquests sistemes per tots costats: en avions, cotxes i trens; en fàbriques, plantes químiques i centrals elèctriques; en satèl·lits artificials i sondes espacials; en aparells domèstics i en equips mèdics; i en molts altres sistemes.

Sigui on sigui que els trobem, i independentment del que facin, els nodes d'un sistema encastat distribuït han de col·laborar per aconseguir un objectiu comú. Així per exemple els nodes d'un cotxe col·laboren per assegurar un viatge segur i agradable i els nodes en una fàbrica col·laboren per fabricar un producte. Com la col·laboració requereix de comunicació, els sistemes encastats distribuïts necessiten un subsistema de comunicació o xarxa que permeti l'intercanvi de missatges entre nodes.

Aquests missatges han de ser intercanviats a temps ja que els sistemes encastats distribuïts interactuen amb entorns físics que avancen de forma inexorable i no esperen. Això significa que la xarxa ha de donar suport a la comunicació en temps real; és a dir, els missatges han de ser intercanviats abans d'instants de temps preestablerts, anomenats terminis temporals.

Per a aquest tipus de comunicació, Ethernet ha estat tradicionalment inadequat ja que no podia proporcionar garanties de temps real. Però això ha canviat gràcies a nous protocols basats en Ethernet. Entre ells destaquen per la seva flexibilitat els que es basen en el paradigma de comunicació FTT (flexible time-triggered). La propietat clau de FTT és que pot garantir el lliurament a temps de missatges fins i tot si els paràmetres de temps real canvien durant l'execució del sistema. Mentre s'executa el sistema, tant els terminis temporals com els temps de transmissió, els períodes i els temps mínims entre arribades dels missatges poden canviar.

Malgrat els seus avantatges, les versions de FTT actuals per Ethernet només satisfan els requisits de temps real si no hi ha fallades ni avaries de components.

Desafortunadament això fa que siguin inadequades per a moltes aplicacions en les quals s'usen sistemes encastats distribuïts, com per exemple els subsistemes crítics d'avions i automòbils. Al cap i a la fi, en la realitat, fallades sí que ocorren i cal tractar-los perquè el sistema sigui fiable. En aquesta tesi anem a veure que FTT basat en Ethernet pot ser modificat per ser tolerant a fallades i que això es pot fer sense sacrificar les garanties de temps real o la flexibilitat que el paradigma FTT proporciona. Això obre la possibilitat d'usar Ethernet, amb el seu baix cost i alt ample de banda, en futurs sistemes encastats distribuïts que han de ser tolerants a fallades, flexibles i alhora de temps real.

Establim la veracitat de que FTT sobre Ethernet pot fer-se tolerant a fallades (preservant la seva flexibilitat i garanties de temps real) mitjançant un nou subsistema de comunicació que ho demostra. En particular, en aquesta tesi dissenyam tal subsistema de comunicació i demostram la seva factibilitat mitjançant una sèrie de prototips de prova de concepte.

El nou subsistema de comunicació, denominat *Flexible Time-Triggered Replicated Star for Ethernet* (FTTRS), es basa en *Hard Real-Time Ethernet Switching* (HaRTES), que és una implementació en Ethernet commutat del paradigma FTT. En concret, FTTRS tracta les vulnerabilitats d'HaRTES: elimina els punts únics d'avaria d'HaRTES mitjançant la replicació de components crítics, garanteix el determinisme de rèplica d'aquests components replicats, restringeix la semàntica d'avaries dels components segons sigui necessari, i fa que els intercanvis de missatge siguin capaços de tolerar fallades transitòries de canal, mantenint la flexibilitat i les garanties en temps real del paradigma FTT.

Für Veet.

La science, mon garçon, est faite d'erreurs, mais d'erreurs qu'il est bon de commettre, car elles mènent peu à peu à la vérité.
[Science, my boy, is made up of mistakes, but they are useful mistakes to make, because they lead little by little to the truth.]

Jules Verne, *Voyage au centre de la terre*

Preface

This dissertation is ultimately a result of my father's passion for computers. Because of him I grew up around computers—as a toddler I was the only one among my schoolmates who had one at home. He unwittingly put me on the path towards studying computer engineering.

During my final year in computer engineering, I took a course in operating system design with Manuel Barranco. Near the end of that course Manuel asked if I would like to do my final year project with him, and Julián Proenza, as my supervisors. Manuel at the time was working on his own PhD thesis and the idea for my final year project was to implement a part of what he was designing: a software driver for ReCANcentrate, a fault-tolerant network architecture for Controller Area Network. I agreed, not because the topic of fault-tolerant networks fascinated me—it does now, but back then I knew too little about it—but because I looked forward to doing my project with Manuel and Julián. I had studied under both and knew them as exceptional teachers.

After I graduated in computer engineering, I did a master in communication and information technologies. My master thesis was also supervised by Manuel and Julián, and thus I continued working on fault-tolerant systems.

After my master's I worked at the University of the Balearic Islands (UIB) as a technician for the CANbids project, a research project that Julián was in charge of. I thus became an official member of his research team.

When my contract ended and the CANbids project was over, I got involved in our team's next project—this time as a PhD student. The project was titled *Fault Tolerance for Flexible Time-Triggered Communication* (FT4FTT) and its purpose was to show the viability of building a highly reliable distributed embedded system that can satisfy changing real-time requirements. Building such a distributed embedded system involved the design of two main subsystems: a fault-tolerant communication subsystem and a node replication mechanism. Sinisa Derasevic, another PhD student in our team, was put in charge of designing the node replication and I was put in charge of designing the communication subsystem. This communication subsystem is the subject of the present dissertation.

Publications Arising from This Dissertation

The present work is about adding fault tolerance to one of the Ethernet incarnations of the Flexible Time-Triggered (FTT) communication paradigm. The result is a fault-tolerant, flexible, and real-time communication subsystem for distributed embedded systems. This subsystem uses a replicated star topology and hence I have named it *Flexible Time-Triggered Replicated Star for Ethernet*, or FTTRS for short. My work on FTTRS resulted in a series of publications, some of which I am the primary author of and others of which I am a co-author. The following is a walk through these publications.

Key Publications of Which I Am a Primary Author

The architecture I propose for FTTRS (Chapter 8, Section 8.3) was first presented in the publication below:

- David Gessner, Julián Proenza, Manuel Barranco, and Luís Almeida. "Towards a Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2013)*. IEEE, 2013.

The mechanism I designed to enforce replica determinism for the FTTRS masters (Chapter 8, Section 8.7.4) was first published here:

- David Gessner, Julián Proenza, and Manuel Barranco. "A Proposal for Master Replica Control in the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 10th IEEE International Workshop on Factory Communication Systems (WFCS 2014)*. IEEE, 2014.

The initial proposal on how to prevent two-faced behaviors of slaves (Chapter 8, Section 8.6.2) and how slaves should manage the trigger message redundancy in FTTRS (Section 8.7.2) can be found in the following paper, which won the *10th IEEE Workshop on Factory Communication Systems Best Work in Progress Paper Award*:

- David Gessner, Julián Proenza, and Manuel Barranco. "A Proposal for Managing the Redundancy Provided by the Flexible Time-Triggered Replicated

Star for Ethernet". In: *Proceedings of the 10th IEEE International Workshop on Factory Communication Systems (WFCS 2014).* IEEE, 2014.

The experimental validation of FTTRS (Chapter 9) has resulted in two additional publications of which I am the primary author.

The experimental assessment of the slave elementary cycle synchronization mechanism (Chapter 9, Section 9.3) was first published in

- David Gessner, Inés Álvarez, Alberto Ballesteros, Manuel Barranco, and Julián Proenza. "Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014).* IEEE, 2014.

The experimental assessment of the fault tolerance mechanisms of FTTRS (Chapter 9, Section 9.5) was presented in

- David Gessner, Alberto Ballesteros, Andreu Adrover, and Julián Proenza. "Experimental Evaluation of Network Component Crashes and Trigger Message Omissions in the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 11th IEEE World Conference on Factory Communication Systems (WFCS 2015).* IEEE, 2015.

Related Publications of Which I Am a Primary Author

Although this dissertation is about making the Ethernet incarnation of FTT fault tolerant, my earliest work was not based on Ethernet, but on FTT-CAN, the Controller Area Network implementation of FTT. For FTT-CAN a fault-tolerant design already existed, but none that leveraged the advantages of star topologies. Exploring how to combine star topologies with FTT-CAN resulted in the following paper, which won the *17th IEEE ETFA Best Work in Progress Paper Award in Emerging Technologies*:

- David Gessner, Manuel Barranco, Julián Proenza, and Michael Short. "A First Qualitative Evaluation of Star Replication Schemes for FTT-CAN". in: *Proceedings of the 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2012).* IEEE, 2012.

There are two further publications related to FTTRS that describe results I obtained, but that are excluded from this dissertation as they do not contribute towards proving my thesis.

The first is about an approach of using graphs and continuous-time Markov chains that I began developing to evaluate the reliability of Ethernet-based FTT networks. It was first described in the following publication:

- David Gessner, Paulo Portugal, Julián Proenza, and Manuel Barranco. "Towards a Reliability Analysis of the Design Space for the Communication Subsystem of FT4FTT". in: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014).* IEEE, 2014.

The second is a preliminary layer architecture for FTTRS, which was presented in

- David Gessner, Ignasi Furió, and Julián Proenza. "Towards a Layered Architecture for the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2015).* IEEE, 2015.

Related Publications Due to Others

There are three publications that describe results that were not obtained by me and where my involvement was minor, but which spawned from my research.

First, there is a publication about the mechanism to synchronize elementary cycles among master replicas (Chapter 8, Section 8.7.4). I conceived the general idea of using semi-active replication for this. Later, Alberto Ballesteros worked out the details and performed an experimental validation (Chapter 9, Section 9.4). The details of the publication are the following:

- Alberto Ballesteros, Julián Proenza, David Gessner, Guillermo Rodriguez-Navas, and Thilo Sauter. "Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014).* IEEE, 2014.

Second, the FTTRS design makes use of port guardians that Ballesteros originally designed for HaRTES (Chapter 8, Section 8.6.2). They were first presented in

- Alberto Ballesteros, David Gessner, Julián Proenza, Manuel Barranco, and Paulo Pedreiras. "Towards Preventing Error Propagation in a Real-Time Ethernet Switch". In: *Proceedings of the 18th IEEE International Conference* *on Emerging Technologies and Factory Automation (ETFA 2013).* IEEE, 2013.

Third, Ballesteros developed a prototype of the FT4FTT architecture, which uses FTTRS as the underlying communication subsystem and thus shows that FTTRS can be used in a real control application (Chapter 9, Section 9.6). The details are described in the following publication, which won the *12th IEEE World Conference on Factory Communication Systems Best Work in Progress Paper Award*:

- Alberto Ballesteros, Sinisa Derasevic, David Gessner, Francisca Font, Inés Álvarez, Manuel Barranco, and Julián Proenza. "First Implementation and Test of a Node Replication Scheme on Top of the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016).* IEEE. 2016

Finally, there is another publication on FT4FTT, which also uses FTTRS as the underlying communication subsystem, but in which I did not participate. Even so, this publication further supports the claim that FTTRS works correctly and that higher network layers can be put on top of it (Section 9.6). The publication is the following:

- Alberto Ballesteros, Sinisa Derasevic, Manuel Barranco, and Julián Proenza. "First Implementation and Test of Reintegration Mechanisms for Node Replicas in the FT4FTT Architecture". In: *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016).* IEEE. 2016

Tools Used

I wrote this dissertation using the LaTeX typesetting system. For the figures I used PGF/TikZ, a graphics package for TeX and LaTeX. For the figures showing plots or graphs of functions I used the LaTeX package pgfplots, which also generates TikZ code.

I did some preliminary simulations to guide the design of FTTRS using the Python programming language and the process-based discrete-event simulation framework SimPy.

I dictated almost all the text of this dissertation using the Dragon NaturallySpeaking speech-to-text software, running within a virtual Windows machine on an Ubuntu GNU/Linux host.

To keep track of all the different versions of the writing and code that I produced, as well as to have a reliable backup of all my work, I used the distributed version control system git.

Finally, I used several tools to push myself to work harder. First, to track the amount of time I spent working, as well as to increase my focus, I used the pomodoro technique. To force myself each week to work a minimum number of pomodoros—even if I didn't want to—I used Beeminder, a goal tracking and commitment device app. Lastly, I also used percentile feedback for tracking my productivity, which is based on continuously comparing one's current productivity with past productivity, thereby motivating one to do better than in the past.

Writing Style

Academese versus Classic Style

I made an effort to write in *classic style*,[1] which Steven Pinker recommends for academic writers in his book *The Sense of Style*.[2] In the book Pinker applies lessons from cognitive science and modern linguistics to the process of writing and introduces the style as being based on simulating two experiences: "showing the reader something in the world, and engaging her in conversation". The style is not about any particular choice of words, but about the roles writers mentally assign to themselves, their readers, and the subject under discussion. Neither is it new: it dates back centuries and, according to Thomas and Turner,[3] has been perfected in the seventeenth century by French writers such as Descartes and Pascal.

Following Pinker's advice I have also deliberately tried to avoid excessive academese to make my writing more readable. The following are examples, drawn from Pinker's book, that contrast academese with the style he advocates:

[1] Francis-Noël Thomas and Mark Turner. *Clear and simple as the truth: Writing classic prose*. Princeton University Press, 2011.

[2] Steven Pinker. *The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century*. Penguin Books, 2014.

[3] Thomas and Turner, *Clear and simple as the truth*.

In recent years, an increasing number of psychologists and linguists have turned their attention to the problem of child language acquisition. In this article, recent research on this process will be reviewed.	All children acquire the ability to speak a language without explicit lessons. How do they accomplish this feat?
Comprehension checks were used as exclusion criteria.	We excluded people who failed to understand the instructions.
This chapter discusses the factors that cause names to rise and fall in popularity.	What makes a name rise and fall in popularity?
The first topic to be discussed is proper names.	Let us begin with proper names.

Recommendations by the IEEE[4] do not disagree with Pinker:

> Write clear, simple sentences in the form of noun-verb-object. [...] Avoid unnecessary phrases such as "Obviously," or "As previously mentioned." [...] Be cautious about using technical jargon that may not be understood by an international audience outside of your immediate subspecialty. [...] Avoid abbreviations if possible. [...] Use simple, common words: "start" instead of "initiate." "Use" instead of "utilize." [...] Try to avoid "lazy" verbs such as demonstrate, exhibit, present, observe, occur, report, and show.

Further writing advice I followed stems from a 22-page booklet published by *The Chronicle of Higher Education*, titled *Why Academics Stink at Writing—and How to Fix It*.[5]

[4] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Authorship Series: How to Write for Technical Periodicals & Conferences*. URL: https://www.ieee.org/publications_standards/publications/authors/author_guide_interactive.pdf (visited on 2016-10-14), pp. 17–18.

[5] Steven Pinker et al. *Why Academics Stink at Writing—and How to Fix It.* The Chronicle of Higher Education, Sept. 2014.

On the Use of First-Person Pronouns

I will generally use the pronoun “we” to refer to you—the reader—and myself—the writer. This is in line with classic style, where our mental model is a conversation. Occasionally I will also use the first-person singular pronoun “I”, even beyond the preface. Some academics consider this inappropriate in scientific writing, but it is a matter of style and useful in a dissertation, which, after all, must be single authored.

Indeed, the use of “I” is allowed or even encouraged by several academic style guides from diverse research areas. For instance, the *The Publication Manual of the APA*[6] has been encouraging the use of personal pronouns since 1974:

> *We* means two or more authors or experimenters, including yourself.
> Use *I* when that is what you mean.

Similarly, the *AIP Style Manual*[7] has the following to say on the use of first-person pronouns:

> The old taboo against using the first person in formal prose has long been deplored by the best authorities and ignored by some of the best writers. “We” may be used naturally by two or more authors in referring to themselves; “we” may also be used to refer to a single author and the author’s associates. A single author should also use “we” in the common construction that politely includes the reader: “We have already seen. . . .” But never use “we” as a mere substitute of “I,” as in, for example, “In our opinion. . . ,” which attempts modesty and achieves the reverse; either write “my” or resort to a genuinely impersonal construction.

How to write and publish a scientific paper,[8] originally published in 1979, and now in its eighth edition, also advocates the first-person:

> Because of this idea [that it is somehow impolite to use first-person pronouns], the scientist commonly uses verbose (and imprecise) statements such as “It was found that” in preference to the short, unambiguous “I found.”

[6] American Psychological Association. *The Publication Manual of the American Psychological Association*. 2nd. American Psychological Association, 1974, p. 26.

[7] American Institute of Physics. *AIP Style Manual*. 1990, p. 14.

[8] Barbara Gastel and Robert A Day. *How to write and publish a scientific paper*. 8th. ABC-CLIO, 2016, p. 202.

> Young scientists should renounce the false modesty of their predecessors. Do not be afraid to name the agent of the action in a sentence, even when it is “I” or “we”.

Science, the peer-reviewed academic journal of the American Association for the Advancement of Science (AAAS), in print since 1880, and which has a 2015 impact factor of 34.661, writes

> [u]se first person, not third; do not use first person plural when singular is appropriate.[9]

Several other examples from other areas are given in Sword’s book *Stylish Academic Writing*.[10]

Furthermore, it is simply not true that scientists generally avoid the use of “I”. Sword, in her book, analyzed the writing styles of 1000 academic articles from across the social sciences, humanities, and sciences (including computer science). For the computer science articles in particular, she found that the use of “I” and “we” is prevalent (82 percent).[11] For her entire data set, she concluded the following:

> Overall, I could identify no particularly strong correlation between pronoun usage and the number of authors per article; that is, single-authored articles are neither more nor less likely than multiple-authored articles to contain first-person pronouns. Nor did I find a single discipline in which first-person pronouns are either universally required or universally banned.

Shelton[12] came to a similar conclusion for papers published in IEEE journals, using a smaller sample:

> For further understanding of IEEE advice on use of first person, it is instructive to examine the editorial practice for IEEE journals and magazines. Inspection of the November 2013 issue of the journal Computer (IEEE Computer Society, 2013a), revealed use of the first person (I or we) in 14 of 18 articles. Similarly, 10 of 12 articles in the July-August 2015 edition of the IEEE magazine Security & Privacy (IEEE Computer Society, 2015) were written in the first person.

[9] American Association for the Advancement of Science. *Articles for* Science.

[10] Helen Sword. *Stylish Academic Writing*. Harvard University Press, 2012, p. 39.

[11] Ibid., p. 18.

[12] D. Cragin Shelton. “Writing in the First Person for Academic and Research Publication”. In: *Proceedings of the EDSIG Conference*. 2015.

Indeed, IEEE's style guide[13] explicitly allows the use of "I":

> If you wish, you may write in the first person singular or plural and use the active voice ("I observed that ..." or "We observed that ..." instead of "It was observed that ...").

The IEEE guide for authors[14] even endorses it:

> Write in the first person ("I," "we") to make it clear who has done the work and the writing.

Using "I" is not only used in journal articles, but also in dissertations in our very own field of research—dependability. For instance, Elizabeth Latronico, who did her PhD under Philip Koopman, uses "I" generously in her dissertation.[15]

Finally, the first-person singular has been used in computer science from the very beginning. Alan Turing—the father of computer science—used it in his landmark 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem",[16] which he published as a still unknown master's student at King's College, Cambridge.

On the Use of Literary Devices

I will use some literary devices, such as metaphors, epigraphs, analogies, alliteration, and allusions. Sword asked over 70 academics from various disciplines to describe what constitutes stylish academic writing in their respective fields.[17] She then analyzed these papers and concluded that such devices are used by many of the papers that are praised for their readability.

Using such devices in academic writing is not so uncommon as some may think. Computer science in particular is full of metaphors and analogies, but we are often so used to them that we do not recognize them as such. Examples include the following:

[13] Institute of Electrical and Electronics Engineers (IEEE). *Preparation of Papers for IEEE transactions and journals*. URL: https://www.ieee.org/documents/transactions_journals.pdf (visited on 2016-10-14), p. 2.

[14] IEEE, *IEEE Authorship Series: How to Write for Technical Periodicals & Conferences*, p. 17.

[15] Elizabeth Ann Latronico. "Reliability Validation of Group Membership Services for X-by-Wire Protocols". PhD thesis. Electrical and Computer Engineering Department, Carnegie Mellon University, May 2005.

[16] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 42 (1936), pp. 230–265.

[17] Sword, *Stylish Academic Writing*.

- The word *machine* to describe mathematical abstractions such as Turing machines, finite state machines, Moore machines, and Mealy machines.
- The words *key*, *virus*, *worm*, *Trojan*, *zombie*, and many more used in security.
- The words *packet*, *link*, *hub*, *bus*, *web*, *page*, and others used in networking, including the word *network* itself.
- The words *stack*, *list*, *tree*, *queue*, and *heap* that describe data structures.
- The verb *to bootstrap*, or *to boot*, which comes from the expression "to pull oneself up by one's bootstraps".
- The words *desktop*, *file*, *folder*, *mouse*, *control panel*, and *recycle bin*.
- The expressions *babbling idiot*, *master*, and *slave*.

Other examples of colorful language include Radia Perlman's algorhyme,[18] a poem that introduced the spanning tree protocol; and articles with such titles as "Two ways to bake your pizza—translating parameterised types into Java",[19] "The toilet paper problem",[20] "Et tu, XML? The downfall of the relational empire",[21] "Well-typed programs can't be blamed",[22] and two notable examples by the Turing-award-winning distributed systems pioneer Leslie Lamport: "The Byzantine Generals Problem"[23] and "The Part-Time Parliament".[24]

On the Citation Style

In computer science, engineering, and math the most common citation style in journals seems to be a numeric style, where references are cited using numbers enclosed in brackets, like this [42]. This makes citations compact, which is perfect

[18] Radia Perlman. "An Algorithm for Distributed Computation of a Spanning tree in an Extended LAN". in: *ACM SIGCOMM Computer Communication Review*. Vol. 15. 4. ACM. 1985, pp. 44–53.

[19] Martin Odersky, Enno Runne, and Philip Wadler. "Two ways to bake your pizza—translating parameterised types into Java". In: *Generic Programming*. Springer, 2000, pp. 114–132.

[20] Donald E Knuth. "The toilet paper problem". In: *The American Mathematical Monthly* 91.8 (1984), pp. 465–470.

[21] Philip Wadler. "Et tu, XML? The downfall of the relational empire". In: *Proceedings of the 27th Very Large Data Bases Conference (VLDB 2001)*. 2001, p. 15.

[22] Philip Wadler and Robert Bruce Findler. "Well-typed programs can't be blamed". In: *European Symposium on Programming*. Springer. 2009, pp. 1–16.

[23] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[24] Leslie Lamport. "The Part-Time Parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

for journal articles, where space has traditionally been at a premium. It is, however, not particularly reader-friendly. When one comes across such a citation (e.g., [42]), one has to scan through the list of references to figure out what reference is being cited. Moreover, readers may have to look up the same citation over and over. After all, most of us have a hard time remembering nondescript numbers (e.g., what reference was number 42 again?). These disadvantages are particularly salient in long documents, such as dissertations, that cite hundreds of references.

Another style often used in longer documents, such as dissertations, is the alphabetical style, where references are cited using letters and numbers created from author names and publication dates, like this [APF97] or this [Bar10]. This style is still compact, although less than the numeric style, and has the advantage of creating somewhat more memorable references. Nevertheless, the references are far from self descriptive and still require the reader to frequently scan the bibliography (e.g., what reference was APF97 again? Am I confusing it with APF98 or with AFP97?).

Yet another style is the author-year style, where references are cited by listing their authors and publication year within parentheses, like this (Smith et al., 2010). This creates references that are easier to remember than numbers or a mishmash of letters and numbers. The disadvantage, however, is that it can create long parenthetical blocks of text that interrupt the reader's flow when several references are cited. Consider the following example:

> Because some authors support the issue at hand (Smith et al., 2010; Smith et al., 2012; Baker et al. 1999; Williams 1995; Williams 1997; Williams 1998; Jones et al. 2015; Jones et al. 2017; Johnson et al. 1982; Lee et al. 2001), while others argue against it (Robinson 2005; Wilson 2004; Miller et al. 1999; Anderson et al. 2014), it is clear that there is so far no consensus.

Since for this dissertation I had no page limit, the compactness of the numeric and alphabetical citation styles had no significant advantage. I have thus chosen a verbose footnote-based citation style (illustrated by the footnotes of the previous section), which is more convenient for the reader. With this style, figuring out what reference is being cited simply involves a quick glance at the bottom of the page, where the reader will find a footnote listing the authors and titles of the works being cited, making it unnecessary to consult the bibliography. Moreover, this style, contrary to the author-year citation style, avoids inserting long blocks of parenthetical text in the middle of the main text.

Acknowledgments

I feel lucky to have had Julián Proenza and Manuel Barranco as my supervisors. Without them I would not have written this dissertation, nor any other, I am sure. They believed in me and welcomed me as a member of their research team early on, giving me the chance to participate in all aspects of research over the years, including peer review, writing of grant proposals and research papers, collaborating with researchers internationally, and participating in conferences all around Europe. In all this—and more—they have been incredibly kind, generous, and well-meaning. I learned a lot and they provided invaluable support and advise.

I want to thank Alberto Ortiz for having helped me navigate the bureaucracy involved in completing a doctoral program at the UIB.

I am indebted to the external reviewers who, as part of the process of completing a doctoral program at the UIB, have taken the time to read and assess this dissertation, and who have provided valuable feedback.

Special thanks go to Alberto Ballesteros, Inés Álvarez, and Sinisa Derasevic. Besides my supervisors, they were the people I most closely worked with. We were all members of the *systems* team of the *Systems, Robotics, and Vision group* (SRV) at the UIB. Without them my struggle towards completing this dissertation would have been terribly lonely. With them, I felt we were all in it together, helping each other.

Thanks also go to Xisco, Emilio, Francesc, Pep Lluís, Joan Pau, Volker, Stephan, and Miquel, all of whom have shared the SRV lab with me at various points in time. We were working on different projects, but when I asked for help, they always provided it generously.

I had hoped to learn from Guillermo Rodríguez-Navas, but he left the UIB shortly after I started my thesis. Still, whenever I saw him, he offered help. I know he would have helped me even more, despite being in another country, had I only asked.

Other people have helped me with my research at various stages and in various ways, including Ignasi Furió, Andreu Adrover, Biel Cardona, Michael Short, and Thilo Sauter.

I did most of my research at the University of the Balearic Islands. But not all. Some of it I did at the *Faculdade de Engenharia da Universidade do Porto*, in Portugal, where I stayed for four months in 2014. For this I have to thank Luís Almeida, who invited me and who was very welcoming. He helped me better understand FTT. Not only did he explain many of the details himself, but he also put me in touch with Paulo Pedreiras and Ricardo Marau, who I have to thank for further cementing my understanding of FTT.

Another person from Porto I shall not forget is Paulo Portugal. He showed interest in one of the ideas I was pursuing at the time and provided valuable feedback that resulted in one of the papers that I am most proud of. (Unfortunately, for practical reasons, this work did not end up in this dissertation.)

I am grateful for my mother, my brother, Fabiola, her family, and my father, who was always proud of me and to whom I dedicate this dissertation.

Finally, I want to thank you, the reader, for your interest. Dissertations are a lot of work, but almost no one reads them. They are still worth the effort—among many other things, they teach the author intellectual humility—but if I can count you as a reader, so much the better.

Funding

The research that resulted in this dissertation was supported by the Spanish *Ministerio de Economía y Competitividad* and the *Fondo Europeo de Desarrollo Regional* through grant BES-2012-052040 (MINECO) and through the FT4FTT project,[25] grant DPI2011-22992 (MINECO/FEDER, UE). It was also partially supported by the Spanish *Agencia Estatal de Investigación* and the *Fondo Europeo de Desarrollo Regional* through the DFT4FTT project,[26] grant TEC2015-70313-R (AEI/FEDER, UE). My research stay at the *Faculdade de Engenharia da Universidade do Porto*, Portugal, in 2014, during which I developed part of this dissertation, was financed by the Spanish *Ministerio de Economía y Competitividad* through grant EEBB-I-14-08994.

[25] Systems, Robotics, and Vision group (UIB). *Webpage of the FT4FTT project*. 2016. URL: http://srv.uib.es/ft4ftt/ (visited on 2016-11-12).

[26] Systems, Robotics, and Vision group (UIB). *Webpage of the DFT4FTT project*. 2017. URL: http://srv.uib.es/dft4ftt/ (visited on 2017-02-27).

Brief Contents

Detailed Contents

List of Figures

List of Tables

List of Acronyms

AFDX Avionics Full-Duplex Ethernet.

ARINC Aeronautical Radio Incorporated.

ART Asynchronous Requirements Table.

AVB Audio Video Bridging.

CAN Controller Area Network.

COTS Commercial Off-The Shelf.

CRC Cyclic Redundancy Check.

CSMA/BA Carrier Sense Multiple Access with Bitwise Arbitration.

CSMA/CD Carrier Sense Multiple Access with Collision Detection.

DES Distributed Embedded System.

DFT4FTT Dynamic Fault-Tolerance for Flexible Time-Triggered Communication.

EC Elementary Cycle.

EFMA Ethernet in the First Mile Alliance.

FCS Frame Check Sequence.

FT4FTT Fault-Tolerance for Flexible Time-Triggered Communication.

FTT-CAN Flexible Time-Triggered Controller Area Network.

FTT-SE Flexible Time-Triggered Switched Ethernet.

FTTRS Flexible Time-Triggered Replicated Star for Ethernet.

FTT Flexible Time-Triggered.

HSR High-availability Seamless Redundancy.

HaRTES/PG Hard Real-Time Ethernet Switching with Port Guardians.

HaRTES Hard Real-Time Ethernet Switching.

IEC International Electrotechnical Commission.

IEEE Institute of Electrical and Electronics Engineers.

ISO International Organization for Standardization.

LAN Local Area Network.

MAC Media Access Control.

MAN Metropolitan Area Network.

NRDB Node Requirements Database.

OSI Open Systems Interconnection.

PARC Palo Alto Research Center.

PRP Parallel Redundancy Protocol.

SCSR System Configuration and Status Record.

SRDB System Requirements Database.

SRT Synchronous Requirements Table.

STP Spanning Tree Protocol.

TM Trigger Message.

TSN Time-Sensitive Networking.

Chapter 1

Introduction

> And Mike [a computer] took on endless new jobs. In May 2075, besides controlling robot traffic and catapult and giving ballistic advice and/or control for manned ships, Mike controlled phone system for all Luna, same for Luna-Terra voice & video, handled air, water, temperature, humidity, and sewage for Luna City, Novy Leningrad, and several smaller warrens (not Hong Kong in Luna), did accounting and payrolls for Luna Authority, and, by lease, same for many firms and banks.
>
> Mannie in Robert A. Heinlein's *The Moon Is a Harsh Mistress*, 1966

In science fiction the future has often been envisioned as a utopian one in which the quality of life of citizens is improved through technology. Common themes include ubiquitous self-driving vehicles; robots at home, on the street, in space, and just about everywhere; and smart cities (on Earth or the Moon!) that take care of themselves and their citizens by means of widespread sensors and the computerized management of their assets, which may include transportation systems, power plants, water supply networks, waste management, and so forth. Science fiction also often imagines us as a spacefaring civilization involved in such endeavors as asteroid mining, space colonization, and extensive space exploration.

To make such a future a reality, we still have to make progress in several research areas. One of them is the area of distributed embedded systems. Without advances in this area, our visions of the future will not be realized anytime soon. After all, the way to implement many of the innovations from science fiction is by means of distributed embedded systems, and in particular, distributed embedded systems that are highly reliable, hard real-time, and adaptive; that is, distributed embedded systems that are unlikely to fail prematurely, that are guaranteed to respond in time, and that can change on their own to deal with new situations.

Since distributed embedded systems rely on a network, this network must also be highly reliable and it must provide the proper support for applications that are hard real-time and adaptive.

In this dissertation we will see that Ethernet—which nowadays can be found in virtually every office and modern household, and increasingly in vehicles, factories, and other industrial applications—can be used to build such a network. Why does this matter? Because it may make the world of tomorrow—with its smart cities, ubiquitous robots and self driving vehicles, advanced space probes, and so forth—a reality sooner rather than later by leveraging Ethernet's attributes: its low cost due to economies of scale, the widespread expertise for it, its continuously increasing bandwidth, and, in general, its promising future due to all the investments that are being poured into it by both the industry and academia.

1.1 A Little Bit of Background

What are highly reliable, hard real-time, and adaptive distributed embedded systems? Let us first define embedded systems.

An **embedded system** is a small computer (think of a single microchip or just a handful of microchips, not of a laptop or desktop computer) that does a narrow set of tasks—such as control or monitoring tasks—within a larger physical system. An example might be a small computer controlling the heating, ventilation, and air-conditioning of a room.

What, then, is a distributed embedded system? A **distributed embedded system** is an embedded system comprised of not one, but several computers. These computers—called computing nodes or simply **nodes**—cooperate to perform a common function, doing so by exchanging messages through a network (see Figure 1.1). A modern passenger airplane is an example: it has several nodes that are distributed among the wings, cockpit, tail, turbine engines, landing gear, etc., and they all cooperate to perform the common function of transporting passengers. For this they

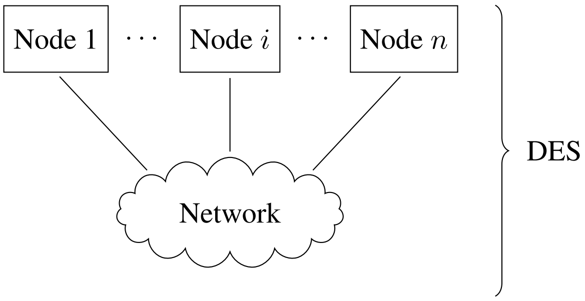


Figure 1.1: A distributed embedded system (DES). It is comprised of a set of nodes interconnected by a network.

collect information from sensors, process this information, exchange some of this information, and interact with the physical environment through actuators based on that information.

What does it mean for a system to be highly reliable? A system is **highly reliable** if it can perform its function continuously for long periods of time, without failing due to faults. This is especially important for systems whose failure may be disastrous, such as passenger-carrying vehicles or life support systems.

What does it mean for a system to be hard real-time? A system is **real-time** if its correctness not only depends on *what* results it produces, but also on *when* it produces those results. A real-time system thus needs to satisfy temporal constraints. The most common temporal constraint are **deadlines**, which are specified instants of time before which certain tasks should be completed. A system is **hard real-time** if not completing a task before its deadline can cause a system failure.[1] An example might be a car with a computer-controlled airbag. If the airbag inflates too late, the airbag control has failed.

Finally, what does it mean for a system to be adaptive? A system is **adaptive** if it can change on its own to deal with unpredictable circumstances. For instance, an adaptive smart city might automatically reroute traffic or reallocate resources in the event of some disaster occurring in one part of the city. Or a smart grid may need to continuously redistribute energy as needed as we rely further and further on

[1] Some authors, such as Kopetz (Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science & Business Media, 2011), use an overlapping, but different definition of hard real-time system. They say that a system is hard real-time if missing a deadline may have severe consequences, such as damages to property or life. This alternate definition implies the first, but the converse is not true: missing a deadline may lead to a system failure without that failure causing catastrophes. I prefer the expression safety-critical real-time system for the alternate definition.

renewable energy sources, which are as unpredictable as a sudden gust of wind.

So what is a highly reliable, hard real-time, and adaptive distributed embedded system? Putting everything together, it is a set of computers embedded within a physical system

- that cooperate to perform a common function, where the function, or the circumstances under which this function needs to be performed, may change unpredictably;
- that perform the function by executing time-critical tasks, which, if delayed too much, lead to a service failure; and
- that perform the function without being interrupted prematurely due to faults.

Do the distributed embedded systems of the future need to be highly reliable, hard real-time, and adaptive? Not all, but certainly many. Let us go through a few examples.

Think of our spacefaring future. Because of the high cost of space exploration, space agencies have already strived for decades to avoid equipment failures. Avoiding such failures will become even more important once we humans start colonizing space. Spacecraft will thus have to become more reliable than ever. And even with an increase in reliability, many future space adventures will still have to be carried out by unmanned probes—not only because of space's hostility towards human life, but also because of the long mission times. Space probes may thus have to face unpredictable environments on their own, especially once we get more ambitious and start exploring areas of space where communication is permanently obstructed, such as the hypothesized subsurface oceans of Europa. Future space probes should therefore be adaptive. Finally, they also have to be hard real-time. For example, if during landing a probe releases its parachute at the wrong time, it will crash into the ground, which is clearly a failure.[2]

Or think of self driving cars. For the safety of their passengers, these need to be highly reliable. Since they are driving through a physical environment, they need to be hard real-time to avoid obstacles, to activate the brakes in time to avoid collisions, and to accelerate to clear intersections before other vehicles can crash into them from the side. They also need to be adaptive to operate anywhere and at anytime: driving in a traffic clogged city center is different from driving off-road or

[2] Indeed, this is precisely what seems to have happened recently to the European Space Agency's Schiaparelli Lander, which on 19 October 2016 crashed into the surface of Mars (Jonathan Amos. *Schiaparelli Mars probe's parachute 'jettisoned too early'*. 2016. URL: http://www.bbc.com/news/science-environment-37715202 [visited on 2016-11-14]).

on a highway; driving when the roads are dry is different from driving when they are slippery due to rain, mud, or ice; driving with a trailer attached is different from driving without one; and driving a damaged vehicle back to safety after something broke down in the backwoods is different from driving an unblemished vehicle.

Or think of mobile rescue robots that are sent into a disaster area. A malfunction of such a robot is unacceptable if its mission is to rescue people or prevent further disasters. It thus needs to be highly reliable. Since it is moving through the physical environment and interacting with people and objects, it needs to be real-time, and even hard real-time: the robot may get stuck because it did not clear an object in time or it may hurt people or cause further damages through untimely movements or actions—all of which are failures of the robot. Further, a disaster area is highly unpredictable and we cannot foresee all circumstances in which the robot may find itself. It therefore needs to be adaptive.

Maybe these applications seem far-fetched, but they are not: space exploration has relied on unmanned probes for several decades now; several organizations are actively working on self driving cars,[3] with policymakers being advised to take this upcoming revolution into account;[4] Tesla's commercial cars are already equipped with partial automation,[5] although with some controversy about its reliability[6]; and during the 2011 nuclear disaster at the Fukushima Daiichi Nuclear Power Plants, mobile rescue robots were deployed.[7]

This last example is particularly illustrative for our purposes. Although rescue robots were deployed in Fukushima, the deployment did not occur immediately because the robots lacked sufficient reliability and adaptivity. Specifically, the "hardware reliability, communication systems, and basic sensors [of the robots] were considered inadequate" and thus valuable time was lost because the robots first had to be "retrofitted for the disaster response missions".[8]

[3] Examples include Audi, Bosch, Ford, General Motors, Google, Mercedes-Benz, Nissan, Oxford University, Peugeot, Renault, Tesla Motors, Toyota, and Volvo.

[4] James M. Anderson et al. *Autonomous Vehicle Technology: a Guide for Policymakers*. Rand Corporation, 2014.

[5] Using SAE's classification of driving automation, Tesla's Autopilot provides level 2 autonomous driving (Society of Automotive Engineers (SAE). *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*. SAE J3016 [SAE]. 2016).

[6] There have been several fatal crashes in 2016 with Tesla's autopilot engaged.

[7] Keiji Nagatani et al. "Emergency Response to the Nuclear Accident at the Fukushima Daiichi Nuclear Power Plants Using Mobile Rescue Robots". In: *Journal of Field Robotics* 30.1 (2013), pp. 44–63.

[8] Ibid.

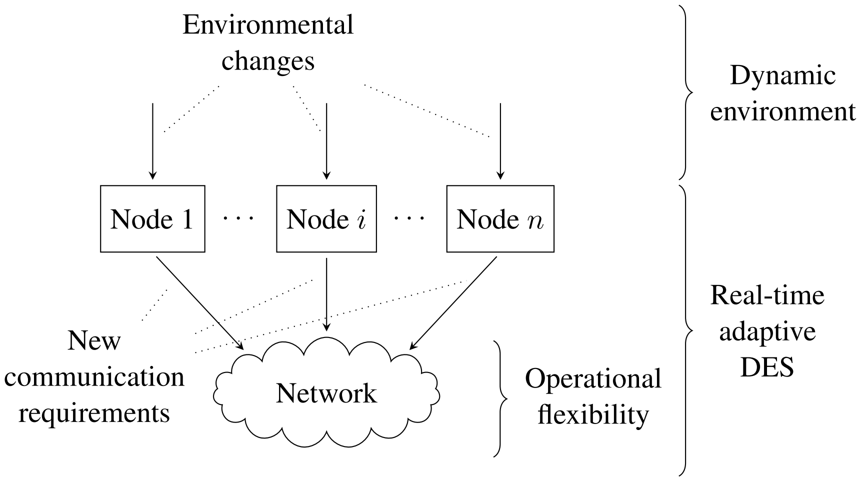


Figure 1.2: A real-time adaptive distributed embedded system (DES). It operates within a dynamic environment to which it adapts by recognizing environmental changes and modifying its own behavior. The behavior changes may impose new communication requirements on the network, which the latter can only meet if it has the necessary operational flexibility.

1.2 Problem Statement

A real-time adaptive distributed embedded system is designed to operate within a **dynamic environment**: an environment where environmental changes may occur. For this the nodes must detect the environmental changes and then modify their behavior to deal with the changes. This may force new real-time communication requirements upon the network: the network may have to allow the nodes to exchange new types of messages, messages that must be exchanged more frequently, messages that carry larger payloads, and so forth—all without compromising real-time guarantees. To meet these new requirements, the network must be **flexible**, that is, susceptible of modification. In fact, if the distributed embedded system is to be truly adaptive, the network must not only be flexible, but it must provide **operational flexibility**, meaning that it must be modifiable at runtime. Moreover, it must be possible for the modifications to be triggered by the nodes or the network itself—the distributed embedded system would hardly be adaptive if a person or other external agent had to be called in to perform the modifications.

Figure 1.2 illustrates the role of a network with operational flexibility within a real-time adaptive distributed embedded system. The top half of the figure corresponds to the environment and the bottom half to the real-time adaptive distributed embedded system (DES). The distributed embedded system is comprised of a network and several nodes, of which at least a few are equipped with adequate sensors. The nodes detect environmental changes and, to deal with them, change their behavior. This forces new communication requirements upon the network. The network may thus have to be modified to meet the new requirements. And since these modifications must occur at runtime, without external intervention, the only way for the whole system to be adaptive is if the network provides operational flexibility.

Providing operational flexibility is easy if the network has significant overcapacity or the new requirements are soft. Having significant overcapacity, however, is generally not the case in distributed embedded systems, which tend to have restrictions on cost, size, weight, and power consumption. And for the distributed embedded systems we are aiming at, which must be highly reliable and hard real-time, the requirements are anything but soft: messages *must* be exchanged reliably and they *must* be exchanged on time. Otherwise, we may have a system failure.

In this case, when there is no significant overcapacity and the requirements are hard, providing operational flexibility becomes difficult. In fact, *there is currently no Ethernet-based communication subsystem available that provides operational flexibility with hard real-time and reliability guarantees*.

Is this true? Let us see.

Ethernet has been used as the underlying network technology for distributed embedded systems for some time now. For instance, in airplanes Ethernet is used in the form of the Avionics Full-Duplex Switched Ethernet (AFDX) protocol and for industrial applications we can find a variety of Ethernet-based protocols, including Fieldbus Foundation High-Speed Ethernet, EtherNet/IP, Process Field Net (PROFINET), EtherCAT, SERCOS III, Media Redundancy Protocol (MRP), Parallel Redundancy Protocol (PRP), High-availability Seamless Redundancy (HSR), Ethernet PowerLink; and TTEthernet. All these protocols support real-time applications, some of them support hard real-time applications, and some of them are also highly reliable. But none also provide the necessary operational flexibility: either they cannot be modified at runtime at all or, if they can, they cannot be modified while ensuring that the nodes, as the network is being modified, can continue to reliably exchange messages without missing deadlines.

The only Ethernet-based protocol that may turn out to have the necessary operational flexibility, while being hard real-time and highly reliable, is Time-Sensitive Networking (TSN), but, as of this writing, it is still actively being developed and not yet available. Moreover, from what has been revealed so far, it seems that TSN is targeted for highly reliable, hard real-time distributed embedded systems, but not distributed embedded systems that are also adaptive.

What *is* lacking, thus, is an Ethernet-based communication subsystem suitable for distributed embedded systems that are highly reliable *and* hard real-time *and* at the same time adaptive.

1.3 Aim of the Study

In this dissertation we will tackle part of the problem of making Ethernet suitable for highly reliable, hard real-time, and adaptive distributed embedded systems. Specifically, we will design an Ethernet-based communication subsystem that is fault tolerant, that guarantees that messages are exchanged on time, and that allows nodes to change the real-time parameters of messages. For instance, the communication subsystem will allow nodes to change the frequency with which certain messages are transmitted, what deadlines the messages have to meet, or the volume of data that the messages convey. Moreover—and this will be our main focus—the communication subsystem will ensure that all messages will continue to meet their deadlines even if faults affect the communication.

Designing this communication subsystem situates our work in the intersection of five research areas: distributed embedded systems, hard real-time systems, Ethernet, flexible systems, and highly reliable systems. Figure 1.3 illustrates this by means of a Venn diagram.

Designing such a communication subsystem from the ground up is ambitious. Luckily, we can build on previous work. Specifically, we can build on the Flexible Time-Triggered (FTT) communication paradigm,[9] which has been applied to Ethernet.[10] This paradigm addresses two of the main issues we face: it supports applications with hard real-time communication requirements and it allows changing these requirements while the system is in operation. For this the paradigm relies on a master/slave approach that roughly works as follows: a privileged node, called the **FTT master**, controls the communication by telling the other nodes, the **FTT slaves**, what messages they can transmit, and when they can do so, so that

[9] Paulo Pedreiras and Luís Almeida. "The Flexible Time-Triggered (FTT) Paradigm: an Approach to QoS Management in Distributed Real-Time Systems". In: *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2001, p. 9.

[10] Paulo Pedreiras, Luís Almeida, and P. Gai. "The FTT-Ethernet Protocol: Merging Flexibility, Timeliness and Efficiency". In: *Proceedings of the 14th Euromicro Conference on Real-Time Systems* (2002), pp. 134–142; Ricardo Marau, Luís Almeida, and Paulo Pedreiras. "Enhancing Real-Time Communication over COTS Ethernet Switches". In: *Proceedings of the 6th IEEE International Workshop on Factory Communication Systems (WFCS 2006)*. Vol. 6. IEEE. 2006, pp. 295–302; R. Santos et al. "A Synthesizable Ethernet Switch with Enhanced Real-Time Features". In: *Proceedings of the 35th Annual Conference of the IEEE Industrial Electronics Society (IECON 2009)*. IEEE. 2009, pp. 2817–2824.

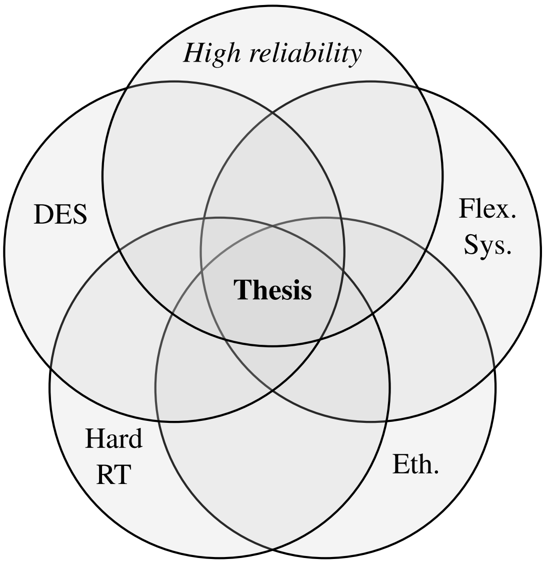


Figure 1.3: Venn diagram showing the research context of the thesis. The thesis lies in the intersection of distributed embedded systems (DES), hard real-time systems, Ethernet, flexible systems, and highly reliable systems.

all real-time requirements are met. The operational flexibility comes from the fact that FTT slaves can request changes to the communication requirements by sending appropriate messages to the FTT master.

Hard real-time communication is supported by FTT by means of periodic and sporadic real-time messages. **Periodic messages** are messages that become available periodically and which are suitable for nodes to regularly exchange the value of sensors or the results of computations. **Sporadic messages**, on the other hand, are messages that are not exchanged regularly, but for which there is a **minimum interarrival time**, a minimum amount of time between when two successive messages can become available. Sporadic messages are used by nodes to exchange non-regular data, such as data from alarms or other unpredictable events.

As shown in Figure 1.4, the fact that FTT supports applications with hard real-time requirements and allows changing these requirements at runtime situates the Ethernet versions of FTT in almost the same research context as our work. What is missing is high reliability. We will address this.

High reliability can be achieved by building a system such that it is unlikely to suffer faults or by building it to operate despite faults. In other words, high reliability can be achieved through fault prevention or fault tolerance. Fault prevention in-

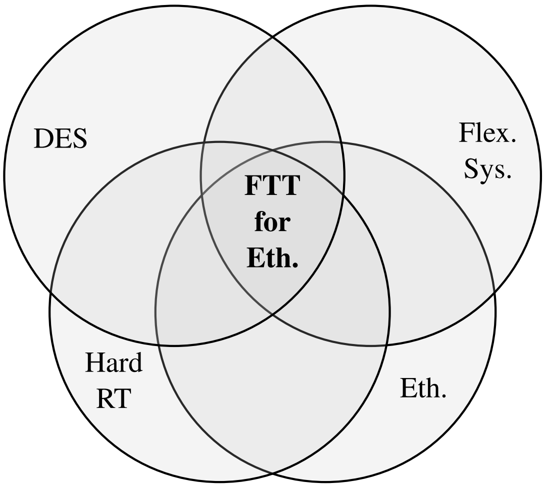


Figure 1.4: Venn diagram showing the research context of FTT for Ethernet.

volves employing high-quality components and manufacturing processes, carefully implementing the system, performing extensive testing, using simple components instead of complex ones, shielding components against electromagnetic interferences, and so forth. Yet, at the end faults are unavoidable. They cannot be prevented indefinitely. In fact, with fault avoidance alone the probability of faults may still be too high for many critical applications. Thus, to maximize the reliability of a system we should not rely solely on fault prevention. We should also use fault-tolerance techniques. Our focus will be on fault tolerance and we will leave fault prevention mostly out of the scope of our work.

Our aim, then, will be to show that the FTT paradigm, used on Ethernet, can be made to tolerate faults, where the faults may be permanent, transient, or both at the same time. Moreover, we will show that the fault tolerance can be added without sacrificing the timely exchange of periodic and sporadic messages, and without sacrificing the operational flexibility of the FTT paradigm. In other words, our aim is to prove the following thesis:

> *We can add tolerance to coincident permanent and transient faults to the FTT communication paradigm, using Ethernet as the underlying technology, while maintaining the paradigm's main features: support for the timely exchange of both periodic and sporadic real-time messages, and support for updating the real-time parameters of these messages at runtime.*

1.4 What This Dissertation Does Not Address

Our work will deal with aspects of fault tolerance in the context of communication networks. To demonstrate the truthfulness of our thesis, we will design a fault-tolerant network architecture, based on FTT and switched Ethernet; and we will show its feasibility by presenting proof-of-concept prototypes, which we will then subject to fault injection experiments. To accomplish these tasks in the amount of time that is reasonable for a PhD thesis, we have to narrow down the scope of our work. Let us, then, point out a few things we are not going to address.

- We will not provide flexible real-time communication ourselves, but improve the reliability of an existing solution, i.e., FTT, while preserving its flexibility.
- We want our communication subsystem to be flexible enough to adapt to changing real-time requirements. Adapting to other requirement changes (power consumption, performance, and so forth) is out of the scope.
- As a starting point for our own solution we will not consider multi-hop FTT topologies, i.e., FTT topologies that involve more than one hub or switch.
- We will focus on fault tolerance, not fault prevention, to make our solution highly reliable.
- A distributed embedded system has two key parts: a communication subsystem and a set of nodes. We will focus on making the communication subsystem fault tolerant. Making nodes fault tolerant is out of the scope.
- We focus on Ethernet. We do not consider other network technologies. The only exception will be a brief discussion of FTT-CAN, which is an implementation of FTT for Controller Area Network (see Chapter 7).
- We do not consider deliberate faults. That is, we do not deal with security.
- We do not consider design faults. We will therefore not use design diversity or techniques such as N-version programming.
- We do not analyze the monetary (financial) cost of implementing, deploying, or operating a solution based on our design.
- We do not analyze the performance (e.g., bandwidth loss due to information redundancy) of our solution. We assume that we have enough bandwidth available for the message retransmission schemes we are going to design.

1.5 Significance of the Study

What is the potential impact of proving our thesis?

First, it will extend the applicability of Ethernet. Ethernet has been used in many domains. It started out as a network technology for non-real-time applications (e.g., office environments). Then it spread to real-time applications (e.g., industrial applications). It further spread to hard real-time and highly reliable applications (e.g., avionics). Proving our thesis will show that Ethernet can also spread to applications that have hard real-time requirements, that need to be able to adapt to changes in these requirements, and that at the same time must tolerate faults. The thesis will thus show that hard real-time, flexibility, and reliability are not mutually exclusive for Ethernet.

Second, in a broader sense, it will help demolish the belief that flexibility is necessarily at odds with high reliability and hard real-time requirements. The traditional argument was that flexibility compromises the reliability of a system and risks meeting hard deadlines. However, for certain applications adding flexibility is the *only* way of achieving high reliability and meeting hard deadlines. After all, if the environment in which the system is operating is dynamic and unpredictable, then we can only guarantee the continuous operation of the system by endowing it with flexibility. Without it, the system may simply be incapable of continuing to provide its service and fail.

Third, it will extend the applicability of the FTT paradigm. Proving our thesis shows that FTT can be made fault-tolerant on Ethernet and that it can be made to not only tolerate faults, but to *seamlessly* tolerate them. This is important for applications that must provide their service continuously, without any failover time whatsoever.

1.6 Overview of the Study

So what lies ahead of us? How will we show that the FTT paradigm can be made to tolerate faults on Ethernet while still allowing hard real-time communication requirements to change dynamically? Basically we will design, analyze, and show the feasibility of a network architecture that proves the point. For this we will have to embark on a research journey.

First of all, we will have to agree on a series of basic concepts and vocabulary from dependability (Chapter 2). This will allow us to properly talk about the aspects relevant to the design of fault tolerant systems. Then, to prepare ourselves for designing our own fault-tolerant communication subsystem, we will recall the basics of designing fault-tolerant systems (Chapter 3). In particular, we will also review the problem of replica determinism (Chapter 4), which appears as soon as we replicate components to provide a common service so that the failure of some of them can be tolerated. Then we will discuss some aspects of real-time communication (Chapter 5); after all, even though we will rely on FTT to provide real-time guarantees, we will need some concepts from the area of real-time communication to properly describe how FTT works. Finally, since our goal is to design a communication subsystem based on Ethernet, we will review this network technology (Chapter 6).

Thus equipped, we will survey the FTT landscape (Chapter 7). We will inspect existing FTT-based network architectures, with an eye to their reliability limitations. This will allow us to pick the one that will best serve us as a starting point for our own solution.

With the most promising FTT-based network architecture in hands, we will begin to prove our thesis: we will design a new Ethernet-based solution that adds fault-tolerance mechanisms, while preserving FTT's flexibility as much as possible (Chapter 8).

Next, to complete the proof of our thesis, we will see that our design is not just an idea, but can in fact be built. For this we will take a look at several proof-of-concept prototypes and fault injection experiments that several students carried out to show that our fault-tolerance mechanisms work in practice (Chapter 9).

Finally, we will conclude our journey by reflecting on what we have learned, on what overall themes can be identified, and on what is important; and we will discuss what future work might be worth pursuing next (Chapter 10).

Part I

Background

Chapter 2

Dependability: Basic Concepts and Terminology

> The Ramans, it seemed, had brought the art of triple-redundancy to a high degree of perfection. This was demonstrated in the airlock system, the stairways at the Hub, the artificial suns. And where it really mattered, they had even taken the next step. New York appeared to be an example of triple-triple redundancy.
>
> Arthur C. Clarke, *Rendezvous with Rama*

Although what the Ramans did might suggest otherwise, making a system highly reliable through fault tolerance is not simply a matter of stacking redundancy on top of redundancy. If we are not careful, adding redundancy to a system might do more harm than good and *decrease* a system's reliability. After all, the more components a system has, the more components can fail and the higher the risk for components to interact in unintended ways. It is only if we take great care in putting the components together that we can ensure an increase in reliability. For this reason, if we want to build our own highly-reliable fault-tolerant communication subsystem, we need to recall what prevents systems from being highly reliable and how the corresponding obstacles can be overcome.

This requires familiarity with a series of concepts and terms that allow us to think about errors, faults, failures, their relationships, their classification, the means to prevent them, and so forth. We will find these concepts and terms within the more general area of dependability, which encompasses reliability and fault tolerant systems.

Discussing dependability will thus be our first objective. We will begin by defining dependability (Section 2.1). Then we will introduce the basic terminology that lets us talk about systems and what they do (Section 2.2). Next, we will define what it means for a system to do what it is intended to do (Section 2.3). At this point we will be able to discuss the concept of dependability in more detail: what its attributes are, what threatens dependability, and thus reliability, and how dependability, and thus reliability, can be achieved (Section 2.4). Finally, we will survey the different types of faults, errors, and failures that may threaten the dependability, and thus reliability, of a system (Section 2.5).

2.1 What Is Dependability?

The concepts and terms we need for our upcoming tasks are not only of interest to us, who are trying to increase the reliability of systems, but also to our colleagues dealing with availability, safety, maintainability, and other related areas. In the past this has led to some bickering. For instance, one quarrel point was whether one discipline (e.g., availability) is a special case of another (e.g., reliability), or vice versa.[1] To put an end to this, and promote more valuable interactions, key figures from our overlapping research communities stepped in. They strived to reach a minimum consensus on terms and concepts we all can share and to put all our disciplines under a new umbrella term: **dependability**.

These key figures defined dependability in various ways over the years. For example, in 1985, based on an earlier definition by Carter,[2] Laprie defined the term as follows:

> Computer system dependability is the quality of the delivered service such that reliance can justifiably be placed on this service.[3]

[1] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology*. Springer Verlag, 1992, p. 6.

[2] William C. Carter. "A Time for Reflection". In: *Proceedings of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS-8)*. 1982, p. 41.

[3] Jean-Claude Laprie. "Dependable Computing and Fault-Tolerance". In: *Digest of Papers FTCS-15* (1985), pp. 2–11.

Later, both in a chapter of a 1991 research report edited by Powell and in his 1992 book, Laprie revised the definition of dependability to read as follows:

> [Dependability is] the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers.[4]

And a decade later, in a technical report, Avižienis, Laprie, and Randell once again revised the definition:

> [Dependability is] the ability to deliver service that can justifiably be trusted.[5]

These definitions present dependability as a property that is neither measurable nor quantifiable. If this is a problem, a more recent definition can be used:

> [Dependability is] the ability of a system to avoid service failures that are more frequent or more severe than is acceptable.[6]

With this last definition, if we can agree on what we consider too frequent and too severe, dependability can be expressed as a probability over time.

Regardless of the specific phrasing, the definitions of dependability all try to capture the same idea. A few examples might help to further clarify the concept.

Consider a flight control system for a passenger airplane, the power grid, and a bank's accounting system. These are all systems where faults and errors can be detrimental to our lives or livelihoods. Yet, our main concern is different in each of them. For a flight control system we must be paranoid about its reliability (its ability to deliver a correct service continuously); for a power grid its availability (its ability to deliver a service when required) is our utmost concern; and for a bank's accounting system we worry most about its integrity (whether it has been improperly altered). Despite their differences, these systems have something in common: they are all examples of systems that must be **dependable** : "able to be trusted to do or provide what is needed" and "able to be depended on".[7] Or, in line

[4] David Powell, ed. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer, 1991; Laprie, *Dependability: Basic Concepts and Terminology*.

[5] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. *Fundamental Concepts of Dependability*. University of Newcastle upon Tyne, Computing Science Newcastle upon Tyne, UK, 2001.

[6] Algirdas Avižienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[7] Merriam-Webster.com. *Dependable*. 2016. URL: http://www.merriam-webster.com/dictionary/dependable (visited on 2016-05-18).

with the definitions we just saw for dependability, they are systems that must be trustworthy enough for us to justifiably place reliance on the service they deliver, that must be able to deliver a service that can justifiably be trusted, and that must be able to avoid service failures that are more frequent or severe than is acceptable.

At the end, dependability lies in the eyes of the beholder: what we consider justifiable, trustworthy, and acceptable, others might consider indefensible, dubious, and below par. Yet, it is a useful term and one that has been adopted widely.[8]

2.2 A System and Its Environment

In a moment we will dissect the notion of dependability into its attributes, threats, and means (Section 2.4). This will clarify how reliability and fault tolerance fit into dependability and how dependability, or reliability, is achieved. Before we do so, however, we first need to define a series of concepts, which are summarized in Figure 2.1. (The concepts we will use are essentially the ones from the paper by Avižienis et al.[9] I have, however, chosen to describe some of them more formally in an attempt to reduce ambiguities.)

A **system** is "an entity that interacts with other entities".[10] These other entities are themselves systems. What exactly these entities are is irrelevant for our forthcoming discussion of dependability. They may be devices, hardware components, pieces of software, humans, or, in general, any object that is part of the physical world. We can think of them as black boxes.

The entities, or other systems, that surround a given system constitute its **environment**.[11] For instance, if our system is a self-driving car, then its environment will include the passengers, the road on which it is traveling, other vehicles, pedestrians, the troposphere with its weather phenomena, and a variety of sources of electromagnetic signals or interferences such as radio stations, GPS satellites, and road-side power lines.

A system is separated from its environment by a shared frontier called the **system boundary**.[12] For a real system the boundary may be a bit fuzzy: where exactly does one system begin and another end? Even so, for our purposes we can assume

[8] As of March 2016, according to Google Scholar, the 1985 paper by Laprie has 744 citations, his 1992 book has 1102 citations, the 2001 technical report by Avižienis, Laprie, and Randell has 951 citations, and the 2004 paper by Avižienis et al. has an impressive 3967 citations.

[9] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing".

[10] Ibid., Sec. 2.1.

[11] Ibid., Sec. 2.1.

[12] Ibid., Sec. 2.1.

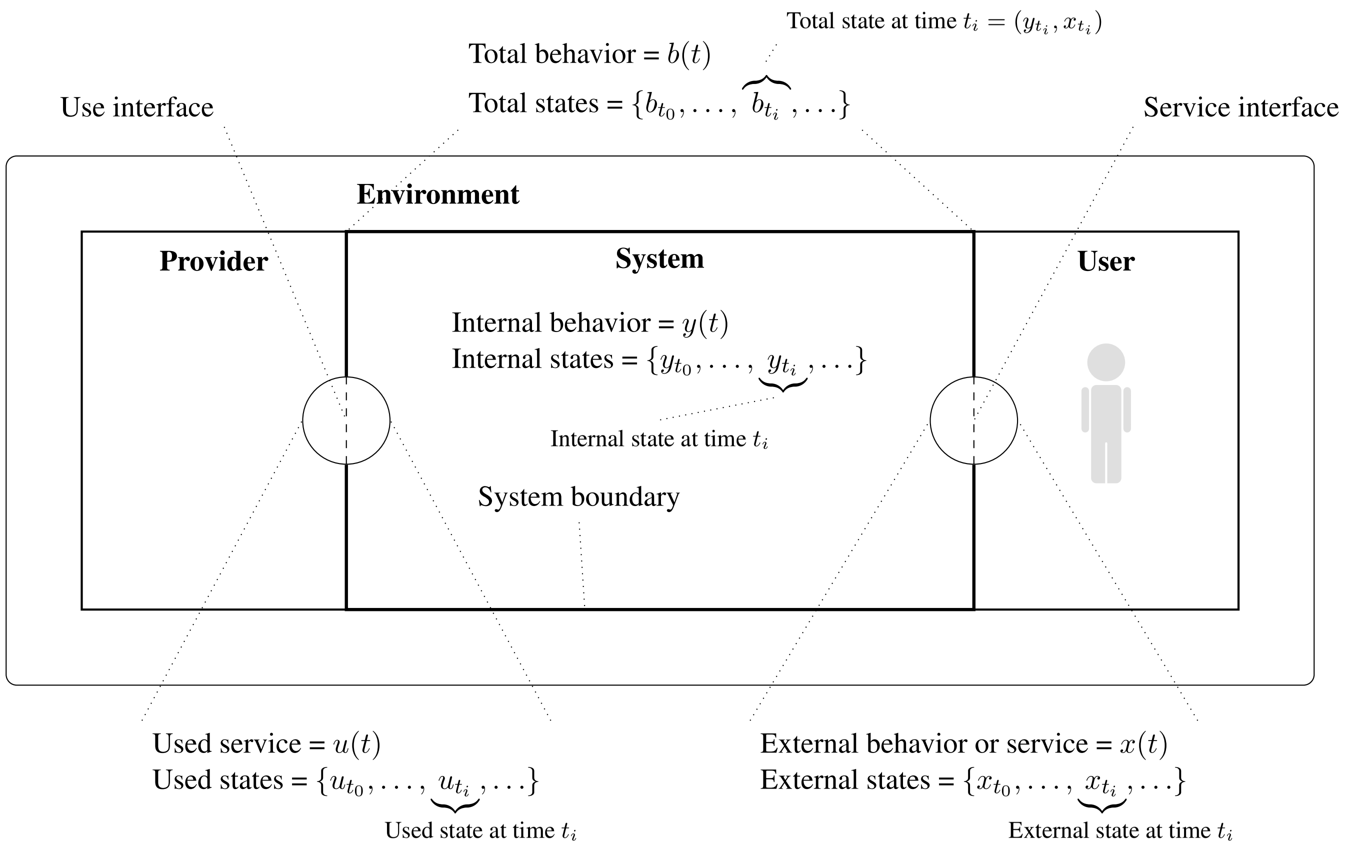

Total state at time $t_i = (y_{t_i}, x_{t_i})$
Total behavior = $b(t)$
Total states = $\{b_{t_0}, \ldots, \overbrace{b_{t_i}}, \ldots\}$
Use interface
Service interface
Environment
Provider
System
User
Internal behavior = $y(t)$
Internal states = $\{y_{t_0}, \ldots, \underbrace{y_{t_i}}, \ldots\}$
Internal state at time t_i
System boundary
Used service = $u(t)$
Used states = $\{u_{t_0}, \ldots, \underbrace{u_{t_i}}, \ldots\}$
Used state at time t_i
External behavior or service = $x(t)$
External states = $\{x_{t_0}, \ldots, \underbrace{x_{t_i}}, \ldots\}$
External state at time t_i

Figure 2.1: A system, its environment, and how it interacts with it. The system uses the provider's service through a use interface. This leads to internal state changes, which are characterized by the internal behavior. The external behavior of our system, as perceived by the user, is the service of our system. The internal behavior, together with the external behavior, is the total behavior of the system. The system boundary separates the system from its environment.

that a well-defined system boundary exists.

The **function** of a system is, quite simply, "what the system is intended to do".[13] The function of a vehicle is to transport passengers, merchandise, or other loads and cargo; the function of a surgical robot is to assist a surgeon in the operating room; and the function of an HVAC (heating, ventilation, and air-conditioning) system is to keep the humidity, air quality, and temperature of a room, building, or vehicle within the bounds established by the user.

The **user** of a system P is another system U for which P performs its function.[14] With respect to the user U, the system P is a **provider**.[15] The user might be a human being, or not. A given system's user is always part of that system's environment, and so are any providers of that system. Thus, U is part of P's environment, and vice versa.

Figure 2.1 shows a system, separated from its environment by a system boundary. Elements of the environment are a provider to the left and a user to the right of the system. Although the figure shows only one provider and one user, a system may have multiple providers and users. Moreover, a given system might be a provider with respect to some systems and a user with respect to others.

Which system is a provider, which is a user, and which is just a bystander depends on the lens through which we look at the whole. With a magnifying lens we may see one transistor being the provider of a service to another transistor, with several other transistors being part of the environment. If we zoom out, the whole set of transistors that we categorized as provider, user, and environment, may all constitute a single logic gate. This logic gate, in turn, may be the provider of another logic gate, the user of other logic gates, and there may be additional logic gates that are part of the environment. If we zoom out once again, the whole set of logic gates may constitute an integrated circuit and, thus, a provider or user of other systems. In this way, we can zoom out more and more, from transistors, to logic gates, to integrated circuits, to computers, to computer networks, and to a whole internetwork like the Internet.

We can also take the opposite approach and start with one system and keep zooming in. In that case we may go from the Internet, to one computer network, to one computer, to one integrated circuit, to one logic gate, and to one transistor. If we do this, we are step-by-step breaking up one system into further subsystems or **components**.[16]

[13] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 2.1.

[14] Ibid., Sec. 2.1.

[15] Ibid., Sec. 2.1.

[16] Ibid., Sec. 2.1.

The **total behavior**[17] of a system is "what the system does to implement its function and is described by a sequence of states".[18] Avižienis et al. call each such state a **total state**.[19]

We can speak with less ambiguity if we formalize Avižienis et al.'s notion of (total) behavior as a function of time. Let T denote the **mission time** of the system, i.e., the time during which the system should implement its function, and let B denote the set of possible total states of the system. Given T and B, the total behavior is a function $b\colon T \to B$ such that $b(t)$ returns the total state in which the system is at time t.

Sometimes it is more convenient to think of the total behavior as an indexed family,[20] i.e., a set of total states indexed by time. In that case we may write $b[T] = \{b_t \mid t \in T\}$, where b_t is an alternative notation for $b(t)$ that emphasizes that the behavior can be viewed as a set of states. We may call $b[T]$ a **family of total states** of the system. If we think of time as being discrete, as opposed to continuous, then T can be represented by a subset of the natural numbers and then $b[T]$ is, in agreement with Avižienis et al.,[21] a sequence of total states $\langle b_1, b_2, b_3, \ldots \rangle$.

Avižienis et al. partition each total state into two parts: the part that cannot be perceived by any user, which they call **internal state**, and the part perceivable by some user, which they call **external state**.[22] Using our formalism, each total state b_t is therefore a tuple $(y_t, x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(n)})$, where y_t denotes the internal state of the system at time t, where n denotes the number of users of the system, and where each $x_t^{(i)}$ is the external state of the system at time t perceivable by the ith user. Hence, if Y denotes the set of possible internal states and $X^{(i)}$ the set of possible external states perceivable by a user i, then the total behavior is more precisely a tuple-valued function $b\colon T \to Y \times X^{(1)} \times X^{(2)} \times \cdots X^{(n)}$.

Although Avižienis et al. do not further subdivide the notion of total behavior, I think it is more consistent to do so. Thus, just like Avižienis et al. subdivide total state into internal and external state, let us also subdivide the total behavior into an **internal behavior** and **external behavior**. We can decompose the total behavior into an internal behavior $y\colon T \to Y$ and n external behaviors $x_i\colon T \to X^{(i)}$, one for each user i of the system. In agreement with Avižienis et al.,[23]

[17] Ibid., Sec. 2.1. Avižienis et al. do not use the qualifier "total" and simply call it "behavior".

[18] Ibid., Sec. 2.1.

[19] Ibid., Sec. 2.1. Avižienis et al. do use the qualifier "total" when talking about such states.

[20] An indexed family is a generalization of a sequence where the index set does not need to be the set of natural numbers.

[21] Ibid., Sec. 2.1.

[22] Ibid., Sec. 2.1.

[23] In ibid., Sec. 2.1, they define the service delivered by a system as "its behavior as it is perceived

each external behavior is then a **service** delivered by the system. We then have that $y(t)$ is the internal state of the system at time t and $x^{(i)}(t)$ is the external state at time t perceived by user i. In other words, y and each $x^{(i)}$ are such that $b(t) = (y(t), x^{(1)}(t), x^{(2)}(t), \ldots, x^{(n)}(t))$.

In Figure 2.1 on page 21 the system's total behavior $b(t)$ is shown at the top. The family of total states is $\{b_{t_0}, \ldots, b_{t_i}, \ldots\}$. Each total state b_{t_i} is partitioned into one internal state y_{t_i} and one external state x_{t_i}. The internal behavior of the system is illustrated in the middle of the figure, within the box labeled *system*, and the external behavior is shown at the bottom right.

As with the total behavior, we can also interpret the internal and external behaviors as indexed families. The **family of internal states** of the system is the indexed family $y[T] = \{y_t \mid t \in T\}$ and the **family of external states** delivered to user i is the indexed family $x^{(i)}[T] = \{x^{(i)}(t) \mid t \in T\}$. Again, these indexed families are sequences of states if time is discrete.

For brevity, we may simply say "the internal states", "the external states", and "the total states", instead of "the family of internal states", "the family of external states", and "the family of total states".

As shown in Figure 2.1, the point on the system boundary through which our system delivers a service to one of its users is a **service interface**.[24] In other words, the service interface is a point of the system boundary where a user can perceive external states. If our system has a provider whose service it uses, then the point on the system boundary through which our system perceives that provider's service is a **use interface**.[25] Hence, whether a point of the system boundary is a service interface or use interface depends on the frame of reference: the same point is a service interface from the provider's perspective and a use interface from the user's perspective.

The service delivered by a provider to our system may be called a **used service**[26] and we can call the corresponding family of external states of the provider a **family of used states**. Hence, a used service is also a function from the mission time to a set of external states, but this time the external states belong to the provider. In Figure 2.1 the system's used service is shown at the bottom left and corresponds to the use interface provided by the system's provider.

For a concrete example, let us consider a digital wrist watch. From our point

by its user(s)".

[24] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 2.1.

[25] Ibid., Sec. 2.1.

[26] Avižienis et al. do not define the term "used service".

of view, we are the users of the watch and the watch itself is a service provider. The service interface is the LCD display of the watch. The external behavior or service is a function that tells us what is displayed on the LCD at any given moment. The external state at a given moment is the specific string of symbols and numbers shown on the display at that moment. The family of external states is the set of all successively displayed strings of symbols and numbers, indexed by time. The internal behavior is a function that tells us the state at a given point in time of the quartz oscillator, battery, and integrated circuits that are part of the watch. The states of the oscillator, battery, and integrated circuits are all internal states. The total state of the watch at a given moment includes the string of symbols and numbers shown on the display, as well as the state of its integrated circuits, battery, and oscillator. Finally, the total behavior of the watch is a function telling us the total state of the watch for each moment.

For an example that is more relevant to us, consider two computers interconnected by a communication link. If the communication is two-way, we may think of the situation as follows. Each computer is both a provider and user of the other, and the service interface is the communication link. The service of each computer would be a function that tells us what message it transmits to the other for each point in time. The internal behavior would be a function that tells us for each moment in time the states of the CPU, memory, hard disk, communication controller, and so forth. Alternatively, if we want to consider the link as a system itself, then we can consider the link as both a user and provider of each computer, and each computer as both a user and provider of the link. If the link has two wires, one for transmission and one for reception, then the computer's service interface might be the point where the transmission wire plugs into the computer. Similarly, the computer's use interface might be the point where the reception wire plugs into the computer. If each transmitted or received message is a sequence of bits, and each bit is encoded as a certain voltage level, then we may want to consider the service of the computer to be a function that tells us the voltage level applied by the computer to the transmission wire at any given moment. The link's service, or the computer's used service, would be a function telling us the voltage level conducted by the link to the reception wire at any given point in time.

As we can see from this last example, the concepts we have introduced are abstract and not necessarily have a unique correspondence to reality. Sometimes we may prefer to think of external states as messages, and at other times it may be more adequate to think of them as voltage levels. The link may be a service interface at a higher layer of abstraction, or it may be a system itself at a lower layer of abstraction.

Let us now dig deeper into one of these concepts. Specifically, since reliability

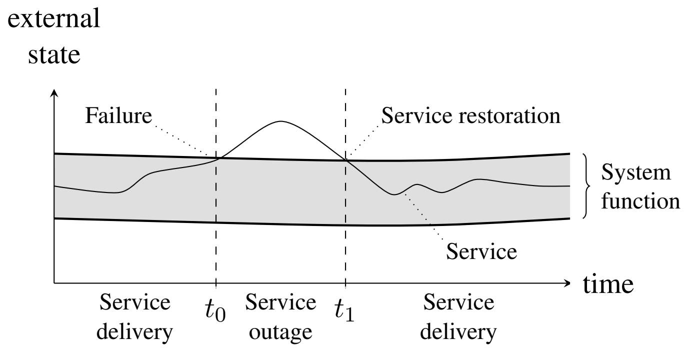


Figure 2.2: A yellow brick road through the service space. As long as the system's service remains within the yellow brick road, i.e., complies with the function, no failure occurs.

is concerned with ensuring that a system delivers a correct service for sufficiently long periods of time, we must clarify what we mean by "correct service". Then we will return to our discussion of dependability.

2.3 Delivering a Correct Service

To make it easier to talk about what it means to deliver a correct service, let us visualize a system's service by plotting it on a two-dimensional space such as in Figure 2.2. To ensure that we can always say whether one instant of time comes before another, we must assume that T, the mission time, is a **totally ordered set**, i.e., a set for which a binary relation "$\leq$" exists that is antisymmetric (if $t \leq t'$ and $t' \leq t$, then $t = t'$), transitive (if $t \leq t'$ and $t' \leq t''$, then $t \leq t''$), and total (for any $t, t' \in T$, either $t \leq t'$ or $t' \leq t$). The relation "$\leq$" is said to be a **total order** on T. Moreover, let us assume that time is continuous. Also, let us assume that the external states are also continuous and totally ordered, and that the external states corresponding to a correct service are contiguous. (See Appendix A on how a correct service can be defined when these assumptions are relaxed.)

One dimension of the two-dimensional space is the set of external states of the system. For instance, if the system is a life support system for a space shuttle, then the set of external states could be the possible oxygen levels within the ship's cabin. The other dimension is time. Let us call such a two-dimensional space a two-dimensional **service space**. Again, Figure 2.2 shows an example. Time is drawn on the horizontal axis and the external states are drawn on the vertical axis.

According to Avižienis et al., a system delivers a **correct service** when it implements its function, that is, when the system does what it is intended to do.[27] Since a system does what it is intended to do when its external states match the external states dictated by the system's function, we can imagine a system's function as a yellow brick road through the system's service space. Figure 2.2 shows such a road as a shaded region. As long as the system's service stays within the road, the system delivers a correct service.

A period of time where the system does what it is intended to do is called a **service delivery**.[28] In Figure 2.2 a service delivery occurs until time t_0 and after time t_1. Sometimes, however, a system may not do what it is intended to do. This happens as soon as the system's service transitions from correct service to incorrect service. This transition is called a **service failure**, or simply failure.[29] In our analogy, a failure occurs when the system's service drives off the road. In Figure 2.2 this occurs at time t_0.

A period of time where the system does not deliver a correct service is called a **service outage**.[30] In terms of the yellow brick road, it is a period of time during which the service is driving off-road. In Figure 2.2 a service outage occurs in the interval of time $[t_0, t_1]$.

For convenience, let us use the adjective **faulty** to say that a system is suffering a service outage; that way we can refer to such a system by talking about a faulty system, instead of having to use long-winded expressions such as "a system that is suffering a service outage" or "a system that has suffered a failure".

A **service restoration** is the transition from incorrect service to correct service.[31] In other words, the event of returning to the road is a service restoration. In Figure 2.2 a service restoration occurs at time t_1.

A system's **functional specification** is a description of its function.[32] In terms of the yellow brick road, we can imagine a functional specification as a region between road markings. Ideally, the road markings should coincide precisely with the borders of the road. Unfortunately, this is not always the case. Figure 2.3 shows an example. The system's service dutifully stays within the road markings, but, sadly, some inept

[27] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 2.2.

[28] Ibid., Sec. 3.1.

[29] Ibid., Sec. 2.2.

[30] Ibid., Sec. 2.2.

[31] Ibid., Sec. 2.2.

[32] Ibid., Sec. 2.1.

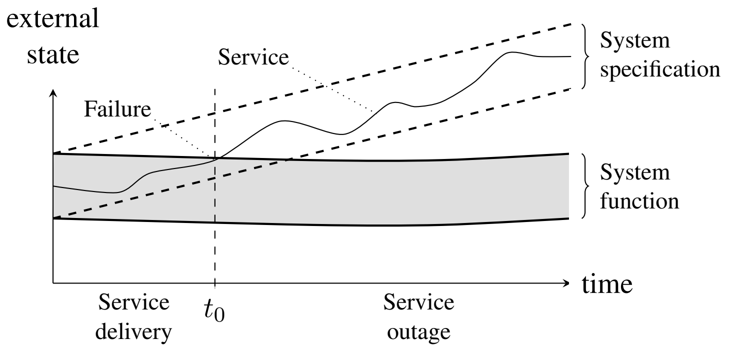


Figure 2.3: A system's service let astray by an incorrect functional specification.

system designer has led the service astray with an incorrect specification. We can conclude from this that a system can only deliver a correct service if the system's functional specification is correct.

Up to now we have been talking about the service of a system in the singular. A system, however, may have to deliver an additional service to comply with its function. For instance, apart from ensuring adequate oxygen levels, a life support system must also ensure that the temperature of the space shuttle's cabin is within a range that humans can live in. In such a case we can imagine a *three-dimensional* service space. As before, one dimension would be time and another dimension would be the oxygen level within the cabin. However, we would now have a third dimension, which would be the temperature. Considering this extra dimension, we can now visualize the function of the system as a yellow brick *tunnel* through the three-dimensional service space. The height of the tunnel might represent the range of bearable temperatures and the width the range of survivable oxygen levels. Given such a tunnel, the system will be delivering a correct service as long as the composition of the two services, the **composite service** (oxygen level and temperature), remains within it; whereas a service failure occurs when the composite service veers off and exits the tunnel through a wall (we have to imagine the walls to be permeable so that the composite service can drive right through them). A service outage is a period of time during which the composite service is roaming outside the tunnel. A service restoration occurs when the composite service reenters the tunnel.

A system may even have to deliver more than two services. For instance, a life support system not only has to control oxygen levels and temperature, but also humidity, carbon dioxide levels, air quality, pressure, and so forth. In that case we have to imagine a multidimensional service hyperspace, a yellow brick hypertunnel, and the composite service trying to pilot through that hypertunnel without flying off a hyperwall.

Despite a yellow brick road (or tunnel or hypertunnel) not being an accurate way of describing a system's function in all cases, I will mention this metaphor again from time to time so that we can get a visual sense of some of the concepts we still have to cover. (Again, see Appendix A on how a correct service can be defined more generally.)

We have now covered what it means for a system to deliver a correct service. The next natural question for us is how to maximize the probability that a system delivers a correct service. For this we need to return to the concept of dependability: we have to see how reliability and fault tolerance fit into dependability, see what threatens dependability, and thus reliability, and how dependability, and hence reliability, can be achieved.

2.4 Dependability Attributes, Threats, and Means

Laprie not only defined the concept of dependability, but also broke it down into attributes and then specified threats that can interfere with it and means that promote it.[33] Later, Avižienis et al. slightly revised all this in their 2004 paper.[34] The result is summarized in the **dependability tree** shown in Figure 2.4. We will now examine this tree, beginning with the attributes.

2.4.1 Attributes of Dependability

As we can see in Figure 2.4, reliability is considered an attribute of dependability, together with availability, safety, integrity, and maintainability. Let us take a closer look at these attributes.

- **Availability**: according to Avižienis et al., the availability of a system is its "readiness for correct service".[35] This is frequently expressed as a function of time: the availability $A(t)$ of a system is the probability that it can deliver a correct service at time t. Availability captures how likely it is for us to get a correct service from a system whenever we want. Following up on our

[33] Laprie, "Dependable Computing and Fault-Tolerance"; Laprie, *Dependability: Basic Concepts and Terminology*.

[34] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing".

[35] Ibid., Sec. 2.3.

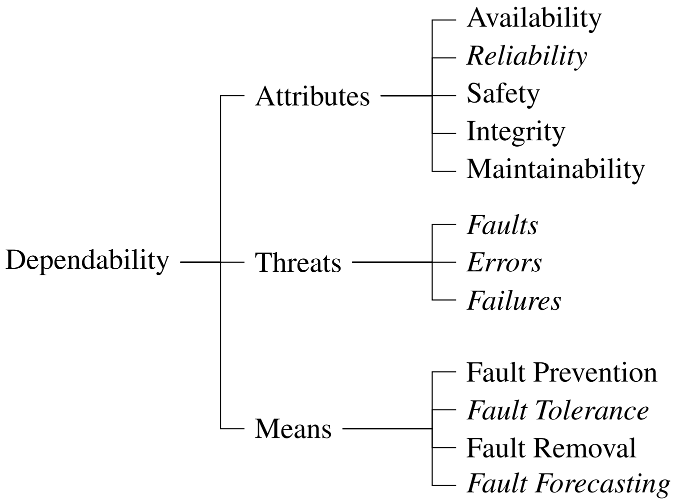


Figure 2.4: The dependability tree. It shows how the fundamental concepts of dependability relate to each other. The concepts shown in *italics* are the ones we are chiefly concerned with. (The figure is based on Avižienis et al.'s *dependability and security tree* in "Basic Concepts and Taxonomy of Dependable and Secure Computing". I removed the term security from the original figure and added the italics.)

analogy, a system's availability indicates how likely it is for the system's service to be within the yellow brick hypertunnel at any given point in time.

- **Reliability**: the reliability of a system is its ability to deliver "continuity of correct service".[36] This can also be expressed as a function of time, making it amenable to mathematical treatment: the reliability $R(t)$ of a system is the probability that it provides a correct service in the time interval $[0, t]$. Reliability captures how likely it is for us to get continuous (uninterrupted) correct service from a system. In terms of the yellow brick hypertunnel, the reliability of a system tells us how likely it is for a system's service to remain within the tunnel for a given interval of time.

- **Safety**: the safety of a system is the "absence of catastrophic consequences on the user(s) and the environment" of the system.[37] Once again, we can also express this as a function of time: the safety $S(t)$ of a system is the probability that it will not suffer any failures in the time interval $[0, t]$ that are catastrophic. Safety thus captures how likely it is that the system will not cause catastrophes, whether the service is within the yellow brick hypertunnel or not.

- **Integrity**: the integrity of a system is the "absence of improper system alterations".[38] Integrity includes both the absence of deliberate tampering with the system as well as the absence of non-deliberate tampering. Moreover, the entity doing the tampering may be any system (human or not) that is part of the environment or part of the system itself.

- **Maintainability**: the maintainability of a system is its "ability to undergo modifications and repairs".[39] Maintainability captures, among other things, how easy it is for a repairperson to put a system's service back on track after it has veered off. Moreover, if the modifications are functional, we can think of maintainability including the ability to change the path and shape of the yellow brick hypertunnel.

(Integrity and maintainability are less commonly expressed as functions of time.)

Our focus will be on reliability. After all, we are on a quest to build a communication subsystem that can provide *continuous* service—and which can do so under

[36] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 2.3.

[37] Ibid., Sec. 2.3.

[38] Ibid., Sec. 2.3.

[39] Ibid., Sec. 2.3.

faults and when facing changes in real-time requirements. Thus, reliability is what we have to target.

Incidentally, reliability is one of the more demanding attributes. It is arguably more demanding than availability and safety. Why? Because a system with high reliability naturally presents high availability and safety, whereas the opposite is not necessarily true. Indeed, when it comes to availability and safety, failures are acceptable as long as they are repaired quickly and do not provoke catastrophes, respectively. For high reliability, in contrast, failures during the mission time are not acceptable at all.

Let us now proceed with the examination of the tree (Figure 2.4, page 30) and move onto the next branch: the threats to dependability.

2.4.2 Threats to Dependability and Reliability

As we can see in the second branch of the dependability tree (Figure 2.4), the threats to dependability, and thus reliability, are faults, errors, and failures. They are the impediments that threaten the correct service of a system, and hence its dependability and reliability.

We already know that a failure is the transition from correct service to incorrect service. But what are errors and faults?

According to Avižienis et al., an **error** is "the part of a system's total state that may lead to a failure".[40] Hence, if the family of total states of a system is $\{b_{t_0}, \ldots, b_{t_1}, \ldots\}$, then an error is a state b'_{t_i} that may lead to a failure. Coming back to our analogy, it is a state of the system that puts the service on a collision course with a wall of the yellow brick hypertunnel. Examples of errors include a wrong computation result, an incorrect value for a state variable, a message with a wrong destination address, or an actuator such as a valve or hydraulic arm being in an improper position.

What makes a state that should be b_{t_i} become an erroneous state b'_{t_i}? This is where faults come in. A **fault** is "the adjudged or hypothesized cause of an error".[41] Hence, we can think of faults as destructive forces acting upon a system's design, manufacturing, or operation that can lead to internal or external states that are harmful to the system's service. Examples of faults include software bugs, ambiguous text in the system's functional specification, broken wires, short circuits, rubbed-off cable insulation, loose connectors, and electromagnetic interference.

[40] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 3.4.

[41] Ibid., Sec. 2.2.

There exists a causal relationship between faults, errors, and failures: a fault may cause an error, one error may cause additional errors, and one of these errors may cause a failure. This is illustrated in Figure 2.5.

The figure shows a system with two components A and B, and a user that receives the system's service through the service interface of component B. Moreover, component B itself receives a service from component A through a corresponding service interface. Now, assume that the system starts out with the user receiving a correct service and that there are no errors. Everything is working fine for a while, but then a failure occurs and an incorrect service is delivered to the user. What happened?

It all starts with a fault. At the beginning this fault is not causing any error and is said to be **dormant** (not causing any error).[42] An example of a dormant fault might be a line of code that has not been executed so far, but that contains a bug. Then the fault triggers an error. This occurs because it has been **activated**.[43] Continuing with our example, the dormant fault that is the line of buggy code is activated when that line is executed. As a result we have a first undesirable alteration of the system's total state, i.e., an error. In Figure 2.5 this first error is shown within component A and is labeled Error_0.

Now the inevitable demise of the system's service may unfold. If the system has no fault tolerance, then, like a deadly virus having injected its genetic material into a cell, leading to the first infected cell, which then spawns more viruses leading to more and more infected cells, spreading from organ to organ, until killing the host organism, a fault causes a first error, which then **propagates** by spawning more and more errors, spreading from one component to another, until ultimately reaching the service interface of the system's user, where it terminates the system's correct service and thus causes a failure. In the figure we see this as Error_0 propagating, causing Error_1, which in turn propagates to cause Error_2, which in turn propagates to cause Error_3, which then propagates to cause Error_4, which then finally affects the service delivered by the system to the user. And here, Error_4 causes a failure of the system by throwing the system's service out of the yellow brick hypertunnel.

The causal chain of events that starts with a fault, continues with errors, and ends with a failure is called a **chain of threats**.[44] We can think of a chain of threats as a row of dominoes, where the first domino that is knocked over is a fault, the last domino is a failure, and the ones in between are errors.

[42] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 3.5.

[43] Ibid., Sec. 3.5.

[44] Ibid., Sec. 3.5.

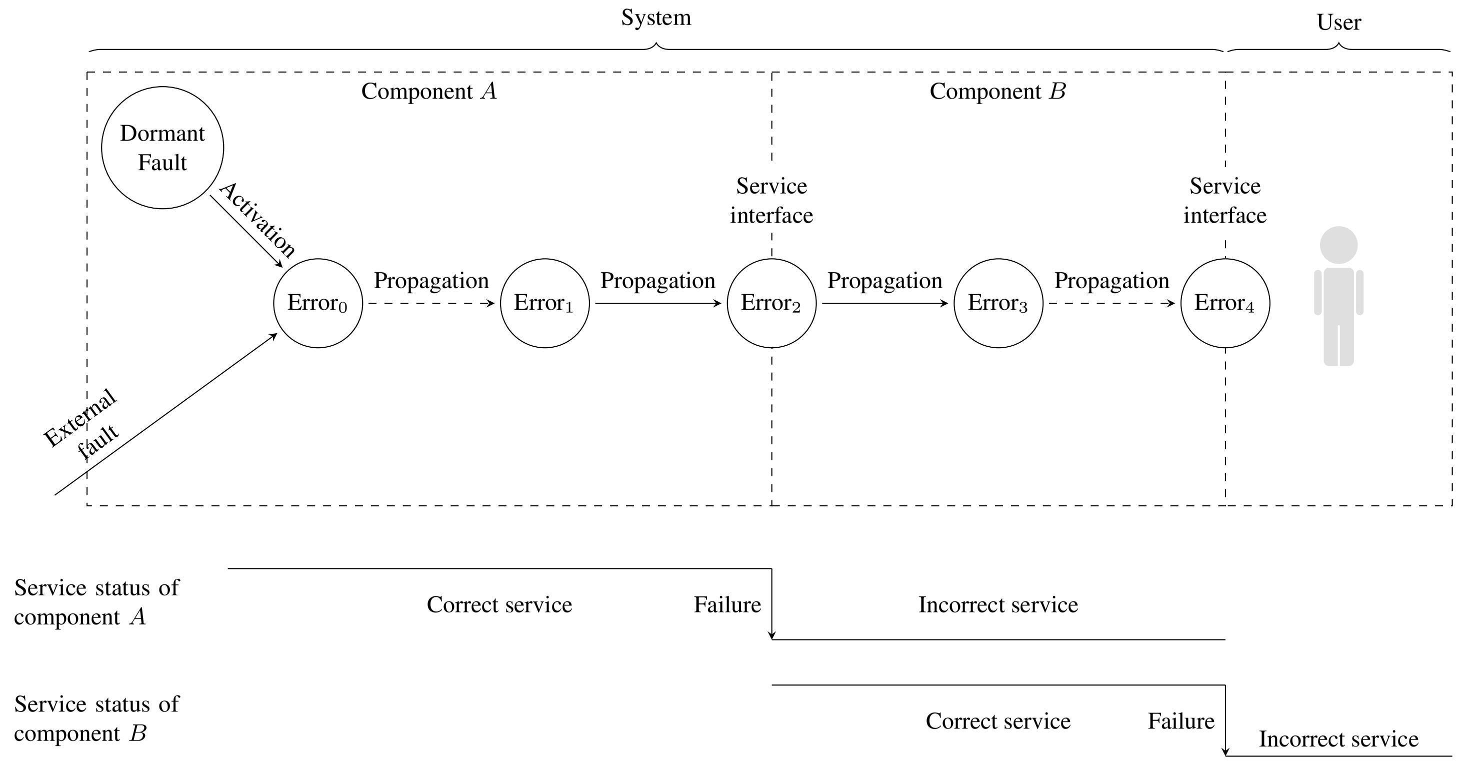


Figure 2.5: Error propagation. (The figure is, with only minor modifications, a reproduction of the error propagation figure by Avižienis et al. in “Basic Concepts and Taxonomy of Dependable and Secure Computing”.)

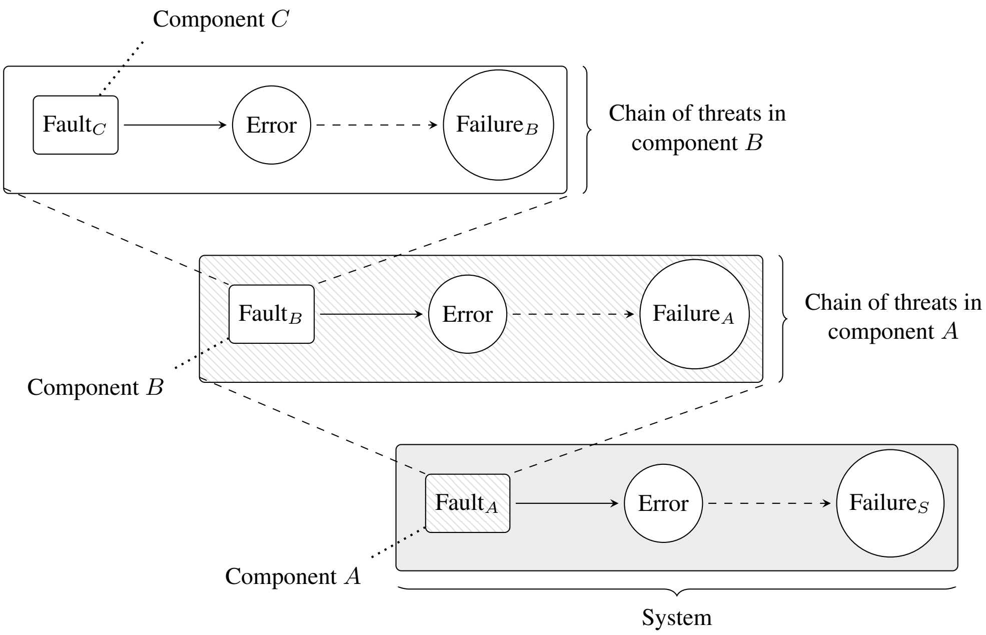


Figure 2.6: A concatenation of threat chains. The chain of threats within one component is concatenated with the chain of threats of the next outer component.

Although we have defined failure as a transition from correct service to incorrect service, in light of the just described error propagation, we can also interpret failures as errors that have propagated so far that they have reached the system's service interface. Thus, if Error$_2$ in Figure 2.5 disrupts the service of component A, then we can think of it as a failure of component A. Similarly, if Error$_4$ disrupts the service of component B, and thus the service of the system, we can think of it as a failure of both component B and of the whole system. Moreover, since an error can be the cause of another error, and a failure of a provider can be the cause of an error in the corresponding user, we have that by definition errors and failures can also be faults. The distinction between fault, error, and failure can be blurry.

Figure 2.6 further illustrates this. It shows a system at the bottom with a chain of threats inside it. This chain ends with the failure of the system (circle labeled Failure$_S$) and starts with the failure of a component A, which is a fault from the system's point of view (box labeled Fault$_A$). And how did Fault$_A$ come about? It started with the failure of a component B nested within A. This is illustrated in the second chain of threats, which unravels within component A and is shown in the center of the figure. As we can see, component A's chain begins with the failure of B, which is a fault from A's point of view (box labeled Fault$_B$). Component B, in turn, fails due to another chain of threats, which is shown at the top and unfolds within B. This chain begins with a failure of a component C, which is located within B, and whose failure is a fault from B's point of view (box labeled Fault$_C$). We could now dive deeper and deeper into further and further components, but let us stop here. The crux of the matter is this: at each level of the recursion, a failure within the inner chain is a fault within the next outer chain. Thus, chains of threats are concatenated and what is a fault, error, or failure is a matter of perspective. The last domino in one chain of threats is the first domino in the next outer chain of threats. Moreover, whether one domino is the first or the last depends on where we trace the boundary between components.

Now onto the last branch of the dependability tree!

2.4.3 The Means to Achieve Dependability and Reliability

The threats to dependability—faults, errors, and failures—are the events that we, as system designers, have to face to build a highly reliable system. Fortunately, we are not helpless against this menace and have several means at our disposal to protect our system. We find these **means** in the third branch of the dependability tree (Figure 2.4, page 30). In line with the quantifiable definition of dependability (page 19), these means are methods and techniques that make it possible for a system to avoid service failures that are more frequent or more severe than is acceptable. Laprie[45] and Avižienis et al.[46] have identified the following ones:

- **Fault prevention**: fault prevention are "means to prevent the occurrence or introduction of faults".[47] It refers to methods aimed at nipping the chain of threats in the bud by preventing faults from occurring in the first place. If we think again of the chain of threats as a row of dominoes, then we may imagine fault prevention as preventing the insertion of wobbly dominoes.

- **Fault tolerance**: fault tolerance are "means to avoid service failures in the presence of faults".[48] It refers to methods aimed at cutting the chain of threats short, so that, despite the occurrence of faults, the last event of the chain—the failure—does not occur. If we think of the chain of threats as a

[45] Laprie, *Dependability: Basic Concepts and Terminology.*

[46] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing".

[47] Ibid., Sec. 2.4.

[48] Ibid., Sec. 2.4.

row of dominoes, then fault tolerance consists in techniques that prevent the last domino from falling despite an earlier domino having been knocked over.

- **Fault removal**: fault removal are "means to reduce the number and severity of faults".[49] If we think of the chain of threats as a row of dominoes, then fault removal may consist in identifying wobbly dominoes and removing them or replacing them with sturdier ones.

- **Fault forecasting**: fault forecasting are "means to estimate the present number, the future incidence, and the likely consequences of faults".[50] We can distinguish between qualitative fault forecasting and quantitive fault forecasting. Qualitative fault forecasting includes identifying in which way components have to fail, or in what combinations, to cause a system failure. Quantitive fault forecasting aims at establishing the extent to which the dependability attributes are satisfied. For instance, quantitive fault forecasting includes reliability analyses, which consist in establishing the probability that the system will not suffer a failure in a given time interval.

2.5 Know Thy Enemy: A Taxonomy of Faults, Errors, and Failures

The chain of threats is what stands in the way of a correct service. It is what can whack a system's service out of the yellow brick hypertunnel and cause a failure. Although we have discussed how to deal with this enemy in the abstract (Section 2.4.3), to actually design a highly reliable system we have to put aside these generalities and be specific. What exactly are the threats that we will have to deal with? What are the faults we are going to face? What sort of errors can we expect? In which ways do we foresee system and component failures to manifest? To answer these questions we will have to be clear about the space of possibilities: what faults, errors, and failures exist? Afterwards we will be in a position to decide which specific faults, errors, and failures out of the space of possibilities to take on.

2.5.1 Classes of Faults

Avižienis et al. have identified eight basic criteria to classify faults. These are summarized in Figure 2.7. Let us go through each of them.

[49] Ibid., Sec. 2.4.

[50] Ibid., Sec. 2.4.

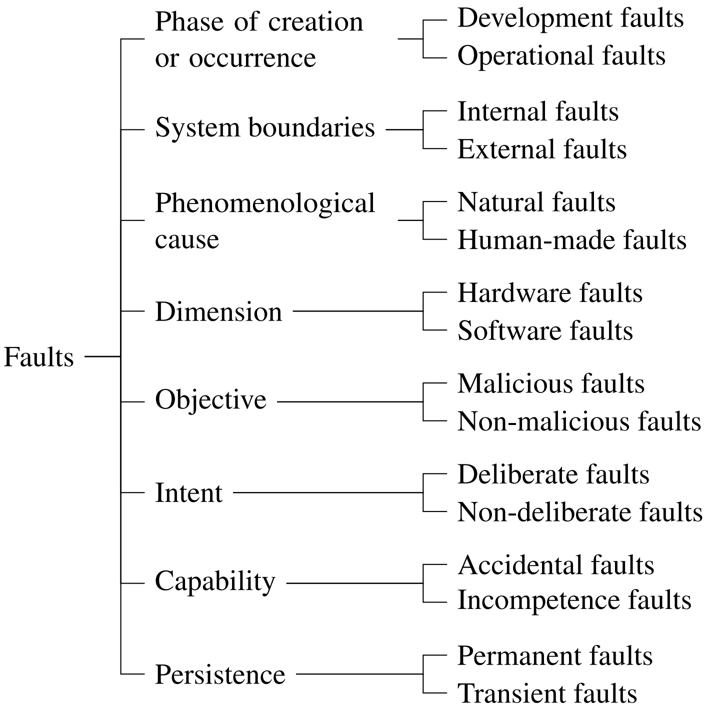


Figure 2.7: Avižienis et al.'s fault taxonomy. (The figure is based on the figure of elementary fault classes by Avižienis et al. in "Basic Concepts and Taxonomy of Dependable and Secure Computing". It differs in that it does not spell out the definition of each fault.)

Classification According to Phase of Creation or Occurrence

The lifecycle of a system consists of a development phase and a use phase. The development phase involves all activities that lead to the system being ready to deliver its service for the first time, from the conception of an initial idea, to a specification, to a design, to the manufacturing, and to the final deployment. The use phase involves everything that happens after the system has been deployed and consists of alternating periods of service delivery, service outage, and service shutdown (an intentional and authorized interruption of the service).

Faults can be introduced into a system during either phase. Faults introduced during the development phase are called **development faults**. Faults introduced during the use phase are called **operational faults**.

Examples of development faults include typos in the specification of a system, manufacturing defects, software bugs, and misconfigurations during deployment. Examples of operational faults include the physical deterioration of components, electromagnetic interferences, and providing incorrect input to the system.

Classification According to System Boundaries

As we saw earlier (Section 2.2), a system is separated from its environment by a common frontier called the system boundary. Based on this boundary, faults can be classified according to whether they originate within it or outside of it. Faults originating within the system boundary are called **internal faults**. Faults originating outside of it are called **external faults**. What exactly is an internal or external fault therefore depends on where we trace the system boundary.

The physical deterioration of a component would be an example of an internal fault. The failure of a cooling system that is part of the environment, and whose purpose is to prevent overheating of the system, would be an example of an external fault.

Classification According to the Phenomenological Cause

Another way to classify faults (which perhaps is more interesting to lawyers than to us) is whether they can be attributed to people or whether they are due to natural phenomena. Using this criterion, we distinguish between **human-made faults**, which are those for which we can blame a person, and **natural faults**, which are those for which we will have to blame natural phenomena.

Classification According to Dimension

The dimension of a fault refers to whether it affects hardware or software. In the first case we talk about **hardware faults**; in the second, we talk about **software faults**.

Examples of hardware faults include the deterioration of physical parts, loose connectors, broken wires, and manufacturing defects.

Examples of software faults include typos in source code, incorrectly implemented functions, and a variety of flaws that can cause software aging,[51] i.e., problems that accrue progressively over time due to, for instance, not releasing operating system resources or not performing hard drive defragmentation.

Classification According to Objective

Human-made faults, which we saw a moment ago, can be classified according to their objective. In that case we distinguish between malicious and non-malicious faults. **Malicious faults** are those that are introduced with the objective to cause harm to the system or its environment. **Non-malicious faults**, unsurprisingly, are those introduced without the intent to cause harm.

Examples of malicious faults include Trojan horses, trapdoors, viruses, worms, zombies, and wiretapping.

Examples of non-malicious faults include any honest mistake when designing, deploying, or using a system.

Classification According to Intent

Another way of classifying human-made faults is according to their intent. Here we distinguish between deliberate faults and non-deliberate faults. To decide whether a fault is deliberate or non-deliberate, we basically have to ask the person that just introduced the fault "did you do that on purpose?". If the answer is yes, we have a **deliberate fault**; otherwise, we have a **non-deliberate fault**.

An example is when a designer purposely chooses not to add any electromagnetic shielding to reduce the weight or cost of a system. Depending on the electromagnetic harshness of the environment where the system needs to operate, this may be a fault.

[51] Michael Grottke, Rivalino Matias, and Kishor S. Trivedi. "The Fundamentals of Software Aging". In: *IEEE Proceedings of Workshop on Software Aging and Rejuvenation, in conjunction with ISSRE. Seattle, WA*. 2008.

If it is, it was a deliberate one.

An example of a non-deliberate fault is an unnoticed typo on a command line or clumsily spilling a coffee over a system component.

Classification According to Capability

Avižienis et al. also classify human-made faults according to the capability (or competence) of the person introducing the fault. Using this criterion we distinguish between accidental faults and incompetence faults. **Accidental faults** are those introduced inadvertently, presumably due to a lack of attention and not due to a lack of skills; whereas **incompetence faults** are those introduced due to a lack of skills.

Classification According to Persistence

Finally, Avižienis et al. further classify faults according to their persistence: faults can either be **permanent faults**, meaning that once present, they do not disappear again without external interventions such as repairs; or they can be **transient faults**, meaning that they disappear after some time.

An example of a permanent fault is a deteriorated component. An example of a transient fault is an electromagnetic interference.

Combined Fault Classes

The preceding eight classification criteria determine the elementary fault classes. If these criteria could be combined in arbitrary ways, then there would be $2^8 = 256$ combined fault classes. But not all combinations make sense. For instance, objective, intent, and capability only make sense for human-made faults, and not for natural faults. So which combinations do make sense? Avižienis et al. have identified 31 combined fault classes.[52] These are summarized in Figure 2.8.

To the left of the figure we find the elementary fault classes. A grid of vertical and horizontal lines is used to indicate the combinations that Avižienis et al. have identified. Each of the 31 vertical lines of the grid corresponds to one combined fault class. For instance, the first combined fault class, labeled by the number 1, is a permanent, accidental, non-deliberate, non-malicious, software, human-made,

[52] Avižienis et al., “Basic Concepts and Taxonomy of Dependable and Secure Computing”, Sec. 3.2.1.

internal, and development fault. The boxes at the bottom of the figure point out representative examples of the corresponding combined fault classes.

When the time comes to decide what faults to consider when designing our communication subsystem (Section 8.2), we will use this figure as a reference.

So much for faults. Next within the chain of threats come errors.

2.5.2 Classes of Errors

We saw that an error is a system state that may lead to a failure (Section 2.4.2). But what classes of errors are there? It depends on the criteria we use for classification. Let us go through the ones that Avižienis et al.[53] have identified.

Detected Versus Latent Errors

"An error is **detected** if its presence is indicated by an error message or error signal".[54] On the other hand, an error is **latent** if it is "present but not detected".[55]

Multiple Related Errors Versus Single Errors

We know that errors are caused by faults since, by definition, faults are the adjudged or hypothesized cause of an error. Sometimes, however, a single fault may cause not one error in one component, but multiple errors simultaneously in several components. This is analogous to a single domino falling over and simultaneously knocking over two or more dominoes that stand next to each other side-by-side instead of facing each other in a sequence. The simultaneously knocked over dominoes, or more precisely, the simultaneously caused errors, are called **multiple related errors**[56]—they are related because they are caused by the same single fault. An example of multiple related errors are those caused by an electromagnetic interference that affects several components.

In contrast, **single errors**[57] are those that affect one component only. Their cause can be traced back to separate faults affecting each a separate component. They are dominoes that have been knocked over individually by separate faults.

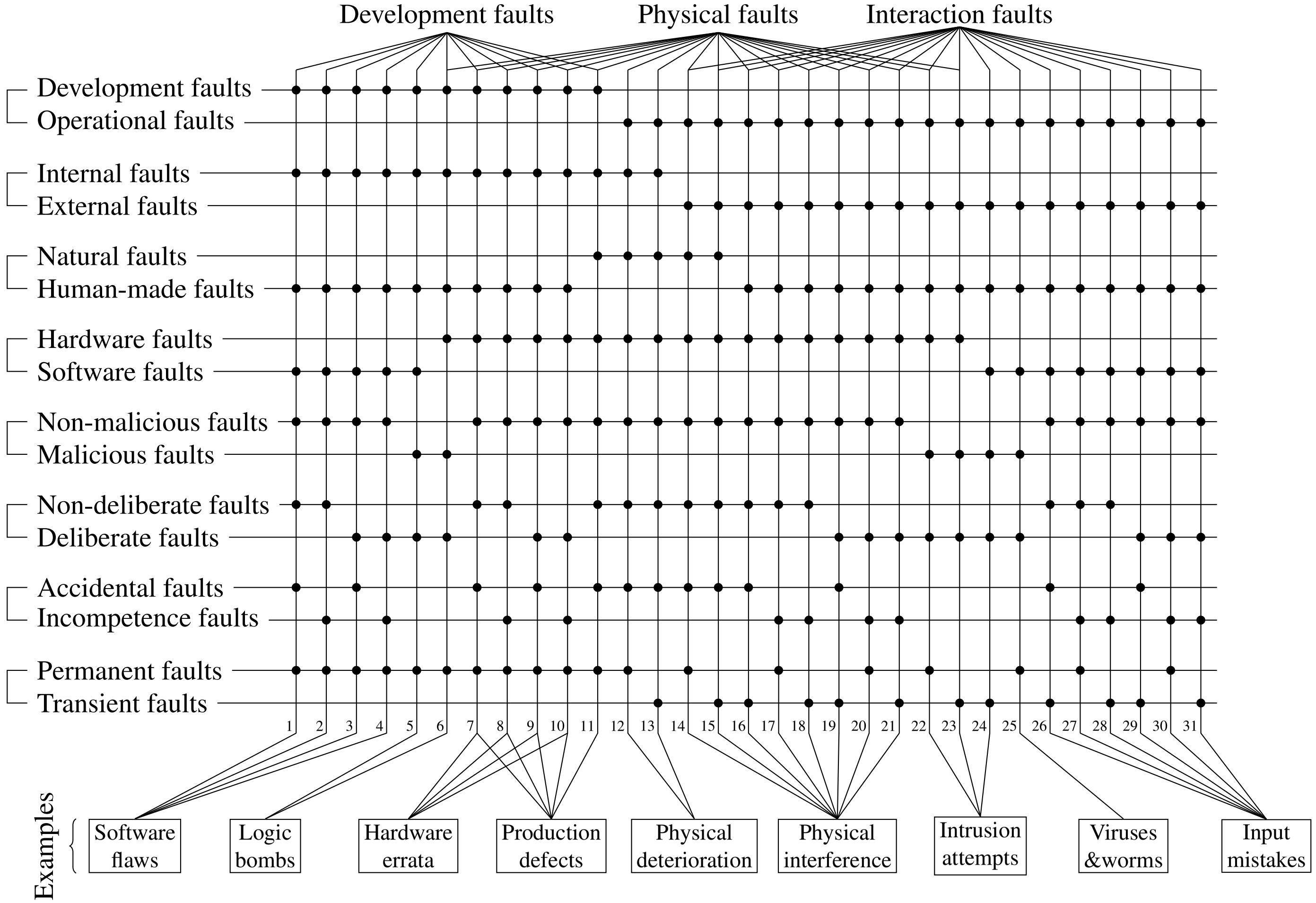
[53] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 3.4.

[54] Ibid., Sec. 3.4.

[55] Ibid., Sec. 3.4.

[56] Ibid., Sec. 3.4.

[57] Ibid., Sec. 3.4.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Development faults	●	●	●	●	●	●	●	●	●	●	●																				
Operational faults												●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Internal faults	●	●	●	●	●	●	●	●	●	●	●	●	●																		
External faults														●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Natural faults											●	●	●	●	●																
Human-made faults	●	●	●	●	●	●	●	●	●	●						●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Hardware faults						●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●								
Software faults	●	●	●	●	●																			●	●	●	●	●	●	●	●
Non-malicious faults	●	●	●	●			●	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●	●	●	●	●
Malicious faults					●	●																●	●	●	●						
Non-deliberate faults	●	●					●	●			●	●	●	●	●	●	●	●								●	●	●			
Deliberate faults			●	●	●	●			●	●									●	●	●	●	●	●	●				●	●	●
Accidental faults	●		●				●		●		●	●	●	●	●	●			●							●			●		
Incompetence faults		●		●				●		●							●	●		●	●						●	●		●	●
Permanent faults	●	●	●	●	●	●	●	●	●	●	●	●		●			●			●		●			●		●			●	
Transient faults													●		●	●		●	●		●		●	●		●		●	●		●

Figure 2.8: Classes of combined faults. (I created the figure based on the figure of combined fault classes in Avižienis et al.'s "Basic Concepts and Taxonomy of Dependable and Secure Computing".)

Classification of Errors According to the Failures They Cause

Another way to classify errors is according to the type of failure they ultimately cause. For this we need to know what types of failures there are. So let us move on and find out.

2.5.3 Failure Modes

Service failure modes, or simply **failure modes**, are the different ways in which a failure can manifest.[58] They are a description of a system's service during a service outage, or the effects on the system's environment of this service during a service outage. Another way to put it is this: since a system's service is its external *behavior*, a failure mode is a description of a system's *misbehavior* during a service outage.

Instead of the term failure mode, we could just as well use the terms "failure classes", "failure types", or "kinds of failure", but "failure modes" is the term that seems to have stuck with dependability researchers in recent years. So what failure modes are there?

So far we have been following the footsteps of Laprie and Avižienis et al. by using their terminology. If we were to continue on this path, we would now classify failure modes according to different viewpoints: the failure domain, which distinguishes between content and timing failures; the detectability of failures, which distinguishes between signaled and unsignaled failures; the consistency of failures, which distinguishes between consistent and inconsistent failures; and the consequences of failures, which ranks failures according to their severity, from minor failures, all the way to catastrophic failures. For us, however, another classification scheme will be more useful.

This alternate classification scheme, with slight variations, has been used by several dependability researchers over the years—for instance, by Barborak, Dahbura, and Malek,[59] and by Poledna.[60] It applies to message passing systems, which we will discuss in a moment (Section 2.5.3), and its key feature is that it puts failure modes into an inclusion hierarchy (a.k.a. nested hierarchy): a hierarchy where one level includes all subsequent levels and is included in all previous levels, like

[58] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Sec. 3.3.1.

[59] Michael Barborak, Anton Dahbura, and Miroslaw Malek. "The Consensus Problem in Fault Tolerant Computing". In: *ACM Computing Surveys (CSur)* 25.2 (1993), pp. 171–220.

[60] Stefan Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism.* Kluwer Academic Publishers, 1996.

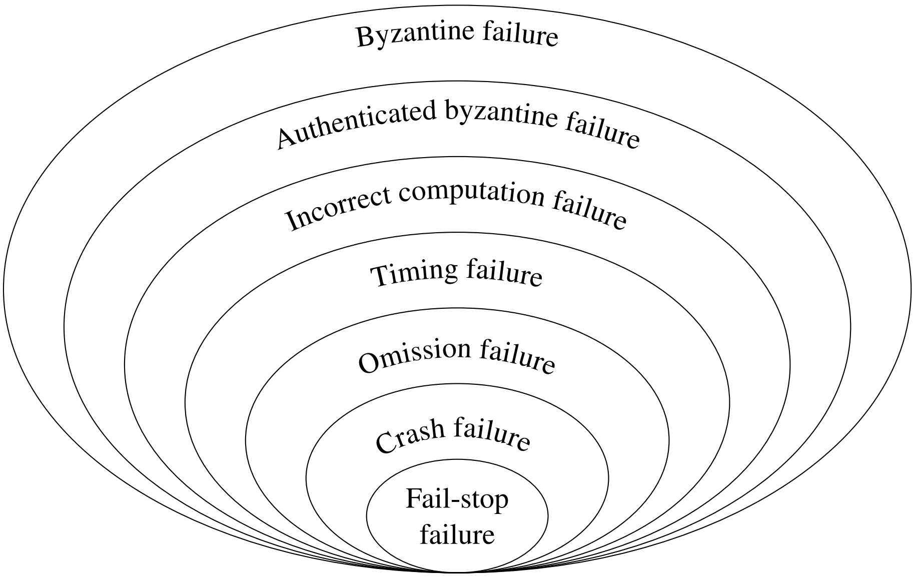


Figure 2.9: The inclusion hierarchy of failure modes. The hierarchy is ordered by how easy it is to deal with a given failure mode, fail-stop failures being the easiest and byzantine failures being the hardest to deal with.

Russian matryoshka dolls. Figure 2.9 shows a Venn diagram of this hierarchy of failure modes.

The hierarchy is ordered by how easy it is to deal with the corresponding failure mode: it is easier to face the innermost level (fail-stop failure) than the outermost level (byzantine failure). Specifically, if all the components of our system fail exhibiting a byzantine failure mode, then it will be an arduous undertaking to make the system highly reliable. After all, since the hierarchy is an inclusion hierarchy, a fail-stop failure is also a crash failure, which in turn is also an omission failure, which in turn is also a timing failure, which in turn is also an incorrect computation failure, which in turn is also an authenticated byzantine failure, which, finally, is also a byzantine failure. Thus, when we are facing a component exhibiting a byzantine failure mode, we are essentially facing a component that is failing in any arbitrary way possible. If, in contrast, all of the components of our system can only fail exhibiting a fail-stop failure mode, then we will have a much easier time ensuring high reliability. This is why the inclusion hierarchy is so useful to designers of highly reliable systems. It tells us what failure modes are especially worrisome and, when we have a choice, what failure mode we should strive for when designing a system or component.

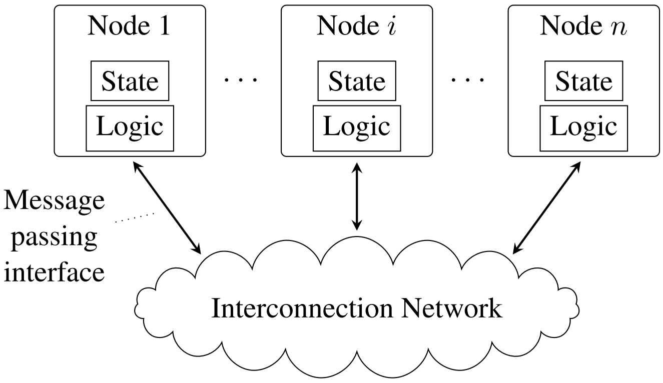


Figure 2.10: Message passing system.

Before we now take a closer look at the failure modes that make up this hierarchy, let us remind ourselves of what a message passing system is.

Message Passing Systems

Up to this point we discussed dependability in general. As we saw in Figure 2.1 (page 21), we thought of systems as black boxes with internal and external states. The external states could be perceived by other systems—the users—at points called service interfaces or use interfaces, depending on whether we took the view of the provider or the user. The successive external states that a user perceived at a given service interface was a service it used. From the provider's perspective, the successive external states were the service it delivered. We kept the discussion generic and did not specify the nature of the systems nor of their states—they could have been voltage levels, sequences of bits, messages, or even the position or orientation of mechanical components such as valves and gears. Now we will get more specific and adopt a model that is especially appropriate for distributed embedded systems and, thus, for the research presented in this dissertation.

Figure 2.10 illustrates what we should have in mind now, namely, a message passing system.[61] Under this model, a distributed embedded system is comprised of a set of nodes, a network, and interfaces for message passing. (Each node may also have actuators, sensors, and other systems connected to it, but I have omitted them from the figure as they are out of the scope of this dissertation.)

[61] A description of message passing systems can be found in Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. Vol. 42. John Wiley & Sons, 2005, Sec. 5.1, p. 103. To be more generic, I have changed the terminology from processor, memory, and links to state, logic, and message passing interfaces, respectively.

The message passing interfaces allow the nodes to exchange messages through the network. How the network and these interfaces are implemented is irrelevant for now (the network may be a simple bus or something more complex, and the interfaces will in most cases comprise at least a cable, a transceiver, and a network controller). Each node contains as components some logic and state (if the nodes are microcontrollers, then most of the logic will be software executing on a processor and most of the state will be the information stored in registers and memories). Since a node is a system, it has an internal behavior, external behavior, and use and service interfaces through which it interacts with other systems. In particular, the internal behavior of a node is given by the successive states of its components, such as processors, memory modules, and so forth. Its external behavior or service towards other nodes is given by the sequence of messages it transmits through its interface. At the same time, the service a node uses from other nodes is a sequence of messages it receives from the network. The message passing interfaces constitute both use and service interfaces: they allow a node to both receive and transmit messages.

Having narrowed down the context to message passing systems, we can now discuss the different failure modes of the failure mode inclusion hierarchy.

Byzantine Failures

The **byzantine failure mode** is the most general one, and hence also the hardest to deal with. It puts no restrictions on how the node's service may deviate from correct service. Like a vicious foe that flouts any rules, a node that has failed with this failure mode may behave with whimsical arbitrariness, including mischievous behaviors. In particular, byzantine behaviors include the following:

- A node pretends to be another node by, for instance, forging messages with an incorrect source address. We will call such behavior an **impersonation**. Others call it masquerading.

- A node sends one message to one recipient and another non-equivalent message to a different recipient even though it should have transmitted equivalent messages to both recipients. We will call such behavior a **two-faced behavior**. Others call it inconsistent behavior.

- The system may exhibit any of the other failure modes of the inclusion hierarchy (Figure 2.9).

Like others have done before us, e.g., Proenza[62] and Barranco,[63] and in agreement with the paper by Lamport, Shostak, and Pease[64] that coined the term “byzantine”, we will assume that the only incorrect behaviors that are exclusively byzantine are impersonations and two-faced behaviors.

By the way, if a message passing system only has two nodes, then impersonations and two-faced behaviors are generally not possible.

The byzantine failure mode is also known as arbitrary, fail-uncontrolled, or malicious failure mode.[65]

Authenticated Byzantine Failures

The **authenticated byzantine failure mode** is equivalent to the byzantine failure mode except that it excludes impersonations. It is called authenticated byzantine failure mode because it is usually enforced by adding a mechanism that allows a node to verify the authenticity of the messages from other nodes. Thus, a node cannot pretend to be another node.

The authenticated byzantine failure mode is also known as authentication detectable byzantine failure mode.[66]

Incorrect Computation Failures

The **incorrect computation failure mode** occurs when a node’s service deviates from correct service in the value domain, time domain, or both, but without these deviations causing any impersonations or two-faced behaviors.

A deviation in the value domain occurs when a node sends a message with incorrect contents. A deviation in the time domain can be a timing deviation or an omission. A **timing deviation** occurs when a node sends a message too soon or too

[62] Julián Proenza. “RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance”. PhD thesis. Universitat de les Illes Balears, 2007.

[63] Manuel Barranco. “Improving Error Containment and Reliability of Communication Subsystems Based on Controller Area Network (CAN) by Means of Adequate Star Topologies”. PhD thesis. Universitat de les Illes Balears, 2010.

[64] Lamport, Shostak, and Pease, “The Byzantine Generals Problem”.

[65] A note of caution: the word “malicious” is used in two different senses. When talking about faults, it refers to human-made faults introduced with the goal of causing harm. When talking about failures, malicious is synonymous with byzantine.

[66] Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, Actually, Poledna calls them “*authentification* detectable byzantine failures”, but this seems to be a Germanism—the German word for authentication is “Authentifizierung”.

late. An **omission** occurs when a node delays the transmission of a message forever.

Timing Failures

The **timing failure mode** occurs when a system delivers a correct service in the value domain, but not in the time domain. That is, a node sends messages with correct contents, but at the wrong time. It is thus identical to the incorrect computation failure mode except that it excludes deviations in the value domain that constitute a failure.

A particular timing failure in distributed embedded systems is the **babbling-idiot** failure mode, which consists in a node transmitting messages nonstop, one after the other. This failure mode is especially problematic when transmissions from one node can prevent transmissions from other nodes because the communication resources are monopolized.

Timing failures are sometimes called performance failures.[67]

Omission Failures

The **omission failure mode** occurs when a node delays a message forever.

Crash Failures

The **crash failure mode** occurs when a node permanently omits the transmission of any message. Crash failures are also called halt failures, silent failures, or simply silence.

Fail-Stop Failures

The **fail-stop failure mode** occurs when a node suffers a crash and, in addition, other correct nodes can detect the crash. Moreover, it is usually assumed that other nodes have access to the last correct service state of the crashed node because the state is stored in some stable storage.[68]

Fail-stop failures are also called stopping failures or signaled failures.

[67] Ibid., p. 25.

[68] Ibid., p. 25.

2.6 Discussion

We picked up several terms and concepts from dependability that will serve us well on our quest to design, analyze, and build an Ethernet-based communication subsystem for highly reliable, hard real-time, and flexible distributed embedded systems, and thus prove our thesis. We discussed an abstract way of thinking about what a system is, what its environment is, what its function is, and how it interacts with other systems. We saw that a system perceives through use interfaces the external behavior of providers and exposes through service interfaces its own external behavior to users. We determined that these external behaviors constitute the service of systems. We remarked on what it means for such a service to be correct. We identified the threats—faults, errors, and failures—that may prevent a service from being correct. We noted that these threats form a causal chain and that this chain is what ultimately ends a system's correct service: a fault causes an error, which causes more and more errors through error propagation, until one or more of these reach the system's service interface, where they impact the service, leading the service to no longer comply with the system's function. We recognized several means to protect a system from such threats: we can prevent faults from occurring in the first place (fault prevention), we can ensure that the system continues to provide a correct service despite faults (fault tolerance), we can periodically intervene in the development or use of the system to exorcise any faults we can identify (fault removal), and we can estimate how successful all this will be in ensuring that our system will not fail prematurely (fault forecasting). We established how we can classify the threats into different classes of faults and errors, and into different failure modes. And we saw that these failure modes can be put into a hierarchy, which not only orders them by inclusion, but also by how malicious or benign they are. Hence, we singled out the failure modes to fear the most and thus to avoid when possible.

Next, we will dive deeper into one of the means for making a system reliable: we will discuss the design of fault-tolerant systems.

Chapter 3

Design of Fault-Tolerant Systems

> Extinction is the rule, survival is the exception.
>
> Carl Sagan

Having discussed dependability (Chapter 2), we now have a vocabulary and conceptual framework that allows us to think about what prevents systems from being highly reliable and what to do about it. As we know, what prevents reliability are the threats to dependability—faults, errors, and failures—and what we can do about it is to employ the means for dependability—fault prevention, fault tolerance, fault removal, and fault forecasting. Next on our agenda is a discussion of the basics of designing a fault-tolerant system.

We will begin by justifying why our focus in the remainder of our journey will be on fault tolerance and fault forecasting, and why we will neglect any further discussion of fault prevention and fault removal (Section 3.1). Then we will survey the major fault-tolerance techniques we can use for making a system fault-tolerant (Section 3.2). Since all these techniques are ultimately based on redundancy, we will examine the four main types of redundancy at our disposal: temporal, spatial, information, and software redundancy (Section 3.3). Afterwards, we will discuss the importance of basing the design of a fault-tolerant system on the right fault and failure-mode assumptions (Section 3.4). We will also discuss how to turn our assumptions into reality to minimize the probability that our system fails because it encounters component failures that violate our assumptions (Section 3.5). We will define the important concept of single point of failure (Section 3.6). And we will discuss the importance of error containment (Section 3.7). Finally, we will conclude with a discussion of how fault-tolerance and real-time requirements may interact (Section 3.8).

3.1 Our Weapons of Choice: Fault Tolerance and Fault Forecasting

To make a system reliable we have the four means of dependability at our disposal (Section 2.4.3 on page 36). But which of the four should we choose for our Ethernet-based communication subsystem? Since the choices are not mutually exclusive, we should use all of them. And this is what we will do. Nevertheless, for the following reasons we will only further discuss fault tolerance and fault forecasting.

Fault prevention can take place during the development and use of a system. During use, it may entail such things as not allowing untrained personnel to interact with the system and not misusing the system for a purpose it has not been designed for. Since our concern is only the development of a communication subsystem, and not its use, we will only deal with fault prevention during the development.

During the development we can prevent faults in several ways: we can choose high-quality hardware components for our system, we can follow sound design principles, we can study for long hours to ensure that we are as competent as possible, we can use the best manufacturing processes only, and so forth. All of this is important to maximize the reliability of a system, but not worth elaborating on to prove our thesis (page 10). Hence, we will not discuss fault prevention any further.

As to fault removal, like fault prevention, it can be performed during the development of a system and during its use. During use fault removal involves maintenance operations such as repairs. But again, the use phase is not our concern. Fault removal during development does concern us, but the design we will follow is already free of any faults I have been able to identify.[1]

Regarding fault forecasting, it is certainly something we will rely on: we will identify the reliability limitations of current FTT solutions (Chapter 7), which is an exercise in qualitative fault forecasting. For now, though, let us postpone this topic until later.

Finally, fault tolerance is the main subject of our thesis. We will talk more about it next and then later apply it when designing our Ethernet-based communication subsystem (Chapter 8).

[1] As is usual in the presentation of scientific results, I will be presenting an idealized research trajectory and not bore you with discussing where I took wrong turns, had setbacks, or otherwise floundered.

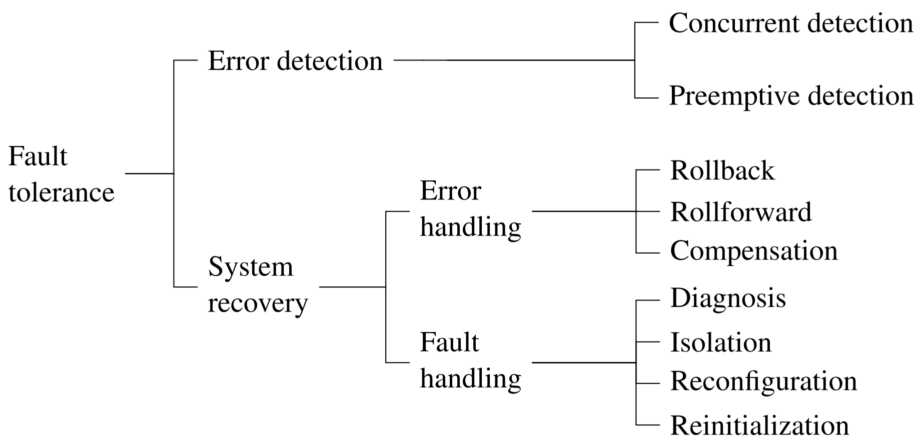


Figure 3.1: Fault tolerance techniques. (The figure is based on the figure of fault tolerance techniques by Avižienis et al. in "Basic Concepts and Taxonomy of Dependable and Secure Computing". It differs in that I have removed the full definition of each fault-tolerance technique to reduce the size of the figure. The definitions can be found in Section 3.2)

3.2 Fault-Tolerance Techniques

As we know, the goal of fault tolerance is to avoid failures even when faults are present. For this several techniques can be used. Figure 3.1 shows how Avižienis et al.[2] have classified these techniques into two main categories: error detection and system recovery. Both are usually used in conjunction to prevent a running system from failing.

Error detection refers to techniques whose purpose is to identify the presence of errors.[3] We can distinguish between two subcategories: **concurrent detection**, which takes place during normal service delivery, and **preemptive detection**, which takes place while normal service delivery is suspended. An example of concurrent detection are cyclic redundancy checks (CRCs) in messages transmitted through a network. An example of preemptive detection are checks of the consistency of a filesystem during the booting of an operating system.

System recovery (also called error recovery) is the transformation of "a system state that contains one or more errors and (possibly) faults into a state without

[2] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing".

[3] Ibid., Fig. 16.

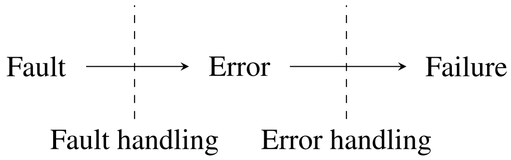


Figure 3.2: Points in the chain of threats where fault handling and error handling intervene to prevent failures.

detected errors and without faults that can be activated again".[4] System recovery consists of error handling and fault handling.

Error handling (also called error processing) is the process of eliminating errors from the system.[5] Figure 3.2 shows schematically where within the chain of threats error handling intervenes to prevent failures. There are three techniques a system can use for error handling. The first is **rollback** (also called backward recovery), which consists in bringing the system back to a correct state that was previously saved. One example might be restoring corrupted files from a backup. The second is **rollforward** (also called forward recovery) and it consists in replacing the erroneous state with a new state. An example of this might be overwriting an erroneous sensor value stored in memory with a more recent value. The third technique is **error compensation**, which consists in having enough redundancy to conceal an error from the service delivered to a user. Concealing an error so that a provided service is still correct is called **fault masking** (or error masking) and " [...] results from the systematic usage of [error] compensation".[6] An example of how to achieve fault masking through error compensation is by having several identical components produce a result each and then applying a majority vote on these results. As long as only a minority of the results are erroneous, the voted result will compensate these errors. Error compensation has an inherent limit on the number of permanent faults in distinct components that can be masked: as more permanent faults in more and more components are masked, less redundancy remains available. Such degradation of redundancy, and hence fault masking, is known as **redundancy attrition**.[7]

Fault handling (also called fault treatment) "prevents faults from being activated again".[8] Figure 3.2 also shows where within the chain of threats fault handling

[4] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", Fig. 16.

[5] Ibid., Fig. 16.

[6] Ibid., Sec. 5.2.1.

[7] Proenza, "RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance", p. 20.

[8] Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing".

intervenes to prevent failures. It involves four steps. First, **fault diagnosis**, which consists in the identification and recording of the location and type of faults that have caused errors. Second, **isolation**, which consists in the "physical or logical exclusion of the faulty components" so that they no longer participate in service delivery. In other words, isolation makes identified faults dormant (since dormant faults are also called passive faults, fault isolation is also known as fault passivation). Isolation helps provide **error containment**, which means that it helps prevent errors from propagating from faulty components to other components.[9] The third step in fault handling is **reconfiguration**. It consists in discarding faulty components and using spare ones instead, or, if no spares are available, in reassigning the service of the faulty components to non-faulty ones that are already providing some other service. The last step is **reinitialization**. It consists in checking, updating, and recording the new configuration that resulted from a reconfiguration and in updating corresponding internal states such as system tables and records. Fault handling is thus very similar to what a repair person would do: diagnose what is causing errors, isolate the culprit, reconfigure the system so that it can continue delivering a correct service, and reinitialize any necessary components. The key difference is that fault handling does not involve a repair person, nor any other external agent, but is performed by the system itself.

What technique shall we use for our fault-tolerant, hard real-time, flexible Ethernet-based communication subsystem? Mostly concurrent error detection and error compensation. But we are getting ahead of ourselves. We will have a more well-founded basis for choosing among the techniques once we have further discussed fault tolerance (this and the next chapter) and we have analyzed the reliability limitations of the current FTT solutions (Chapter 7).

3.3 Types of Redundancy

Fault tolerance is based on redundancy: it is based on the use of more components and more internal or external states than would be necessary if we could guarantee that the system would never encounter a fault. This is true irrespective of the fault tolerance technique used.

What types of redundancy are there? The ones typically used in digital fault-tolerant systems are temporal redundancy, spatial redundancy, information redundancy, and software redundancy.

[9] Barry W. Johnson. *Design and Analysis of Fault Tolerant Systems*. Ed. by Harold S. Stone. Addison-Wesley Publishing Company, Inc., 1989.

Temporal redundancy,[10] also known as time redundancy,[11] consists in having a provider deliver an external state once to a user and then, some time later, having it deliver additional external states that would be unnecessary in the absence of faults. For networked computers, temporal redundancy typically means doing the same computation more than once or sending the same message more than once. Temporal redundancy can be used for error compensation: if one external state is erroneous, it may be compensated by a subsequent identical external state. It may also be used for error detection: if one external state and a subsequent one that are supposed to be identical are not, then there must have been an error.

Spatial redundancy[12] consists in a system having more physical components than would be necessary if no faults could occur. Spatial redundancy is also known as hardware redundancy, structural redundancy, space redundancy,[13] or physical redundancy. An example of spatial redundancy is **hardware replication**,[14] where we have two or more providers, called **replicas**, that can deliver the same service to the same user, such as when we have a computer with two hard disks that store the exact same information (this is known as disk mirroring or RAID 1, where RAID stands for redundant array of independent disks). Another example of spatial redundancy is adding components whose sole purpose is the detection of errors. Spatial redundancy can thus be used for several of the fault-tolerance techniques: it can be used for error detection by, for instance, comparing the services of two identical replicas (which is called dual modular redundancy) or by having a component that checks that the external states of a provider remain within a set of permissible values; it can be used for error compensation by having several replicas provide the same service so that even if some of them fail, there are others that can continue providing the service; and it is essential for the reconfiguration phase of fault handling, when the faulty components are affected by permanent faults.

[10] K. Arun and H. Nitin. “Understanding Fault Tolerance and Reliability”. In: *Computer* (1997), p. 46; Laura L. Pullum. *Software Fault Tolerance Techniques and Implementation.* Artech House, 2001, p. 21; George A Reis et al. “SWIFT: Software Implemented Fault Tolerance”. In: *Proceedings of the 3rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2005).* IEEE Computer Society. 2005, pp. 243–254, Sec. 3.1.

[11] Algirdas Avizienis and Jean-Claude Laprie, eds. *Dependable Computing for Critical Applications*. Vol. 4. Springer Science & Business Media, 2012, p. 304.

[12] Arun and Nitin, “Understanding Fault Tolerance and Reliability”, p. 46; Reis et al., “SWIFT: Software Implemented Fault Tolerance”, Sec. 3.1.

[13] Avizienis and Laprie, *Dependable Computing for Critical Applications*, p. 304.

[14] Some authors, such as Poledna, (Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*) equate spatial redundancy with the term “replication”. I, however, will use the term “replication”, without the qualifier “hardware”, to also apply in the context of redundancy that is not spatial so that we may talk about the replication of computations, processes, software, messages, and so forth. Moreover, I will consider hardware replication to be only one particular type of spatial redundancy.

Software redundancy[15] applies to systems that execute software and refers to the addition of extra software, beyond what is needed to deliver a service in the absence of faults. An example of software redundancy is when we have a computer that includes software to check the integrity of databases or file systems.

Information redundancy,[16] also known as data redundancy means having more information stored or exchanged than would be necessary in the absence of faults. For digital systems, information redundancy typically involves the addition of redundant bits to data. This may be done for error detection, as is the case with parity bits and cyclic redundancy checks (CRCs), or for error correction, as is the case with Hamming codes. Information redundancy is sometimes considered a special case of software redundancy.[17]

When we use redundancy, the redundant parts—especially in the case of hardware and software replication—can be implemented following the same design or following diverse designs. In the latter case we talk about **design diversity**, which allows to tolerate design faults as well.

3.4 Assumptions, Assumptions, Assumptions

When designing a fault-tolerant system, it is essential to define our assumptions.[18] What types of faults can the components of our system suffer? With what rate do they fail? And if they do fail, how do they fail? With what failure mode? The answers we presuppose for these questions are our assumptions and they dictate, to a large extent, our design.

The types of faults we want to address dictate the type of redundancy our system must have. If we want to address development faults, we need design diversity. If we only address operational faults, replicating identical hardware may be enough. If we address permanent faults, we need spatial redundancy. If we only address transient faults, temporal redundancy may suffice.

The rate with which components fail dictates the amount of redundancy we need to make our system reliable. If we expect components to fail once per day, we may need more redundancy than if we expect them to fail once per year.

[15] Pullum, *Software Fault Tolerance Techniques and Implementation*, p. 18.

[16] Arun and Nitin, “Understanding Fault Tolerance and Reliability”, p. 46; Pullum, *Software Fault Tolerance Techniques and Implementation*, p. 19.

[17] Pullum, *Software Fault Tolerance Techniques and Implementation*, p. 19.

[18] David Powell. “Failure Mode Assumptions and Assumption Coverage”. In: *Predictably Dependable Computing Systems*. Springer, 1995, pp. 123–140.

The expected failure modes dictate both the design and amount of redundancy we need. For instance, if a particular type of component can only fail with a silent failure mode, then we need $k + 1$ replicas of that component to tolerate the failure of k such components. If, however, the components can fail with a byzantine failure mode, then we need $3k + 1$ components to tolerate the failure of k such components.[19] In general, the deeper within the inclusion hierarchy (Figure 2.9 on page 45) a failure mode is, the easier it is to deal with.

The assumptions we make about faults constitute a fault model. A **fault model** therefore is a description of the types of faults we assume to act upon a system and of the rate with which we assume particular faults to disturb the system.

The assumptions we make about failure modes constitute a failure model. A **failure model** therefore describes in what way—with what failure mode—we expect the different components of a system to fail.

Getting our assumptions right is crucial. If our fault and failure models are too optimistic, we will design a system that will run into faults, errors, and failures it cannot deal with. And that will be the end of our supposedly fault-tolerant system.

Determining whether our failure model is realistic can be quantified using the concept of failure mode assumption coverage.[20] The **failure mode assumption coverage** for a component is the probability that during a service outage the component's service will indeed misbehave with the failure mode we assumed for it. A high assumption coverage means that the given component is likely to fail with the assumed failure mode; whereas a small assumption coverage means that the component is likely to fail with a different mode than the one we assumed.

Assumption coverage is an important measure. Just imagine we design and build a system that can tolerate the failure of any component that exhibits a failure mode M; we then proudly present our system at a dependability conference; and then a fellow researcher points out that for our components the probability of failing with failure mode M is only 0.05. This would mean that we built a system designed to tolerate only a minority of the component failures that will actually occur!

Abashed, we might return home to design another system for next year's conference. But this time we make sure to maximize the failure mode assumption coverage. Hence, we assume a byzantine failure mode for every component. Since a byzantine failure mode corresponds to all possible failures, we now have an assumption coverage of unity for every component.

Now all that remains to be done is to design our system such that it can tolerate a

[19] Lamport, Shostak, and Pease, "The Byzantine Generals Problem".

[20] Powell, "Failure Mode Assumptions and Assumption Coverage".

byzantine failure of any component. This, unfortunately, is no easy task.

As we begin to labor, we soon realize that assuming the worst possible failure model requires so much redundancy that it makes our system extremely costly. But it is even worse, as we are adding more and more redundancy, and more and more complex fault tolerance mechanisms, experiments with prototypes soon show that our system is becoming less and less reliable. What is going on?

Although assuming the worst possible fault model and failure model can ensure that our assumptions are not violated during system operation, it has a significant downside besides cost. If we assume the worst, we must use more redundancy; but if we use more redundancy, we are also increasing the possible sources of failure.[21] In particular, since no system is perfect, with increased redundancy we increase the probability of **uncovered failures** occurring, which are component failures that, despite all our efforts and best intentions, our system will not be able to tolerate.[22] Basically, there is always at least one constellation of component misbehaviors that brings our system down; and as we add more and more redundancy, the system becomes more and more complex and such constellations become more and more likely.

We thus face a dilemma. If we are too optimistic with our assumptions, our system will fail prematurely because it will not have enough redundancy or lack the proper fault tolerance mechanisms. But if we are too pessimistic, our system may also fail prematurely because additional redundancy increases the probability of uncovered component failures occurring.

The goal then is not to maximize the assumption coverage, which is trivial (just assume byzantine failures), but to choose a failure model whose assumption coverage is sufficiently high, but not so high that we need so much redundancy that the probability of uncovered failures gets out of hand.

We thus have to strike the right balance between optimism and pessimism for our failure models. But how can we tell if we have have struck the right balance? Are we considering all unavoidable failure modes? Or are we fretting about modes that we can safely ignore? Should we be as paranoid as assuming that this critical component is likely to fail with a byzantine failure mode? Or are we erring in the other direction and assuming that a component will fail in a benevolent way, when in fact it is much more likely to turn malicious upon a failure? Unfortunately it is quite hard to answer these questions. To create accurate failure models we may have to rely on experience, on statistical analyses of the failure modes of components,

[21] Ibid.

[22] Suprasad V. Amari et al. “Imperfect Coverage Models: Status and Trends”. In: *Handbook of Performability Engineering*. Springer, 2008, pp. 321–348.

and case studies showing how other systems have failed in the past.

Fortunately, for the failure model, there is another way out of the dilemma: do not assume the failure model, but turn it into reality. This can be achieved by restricting the failure semantics of the system's components. Let us see what this means.

3.5 Turning Assumptions into Reality: Restricting the Failure Semantics

The **failure semantics** of a system are the ways in which it can misbehave during a service outage.[23] Failure semantics are usually expressed in terms of a failure mode. Thus, we say that a system has M failure semantics if the failure mode a system exhibits during a service outage is M.[24] For instance, if we refer back to the failure mode inclusion hierarchy (Figure 2.9, page 45), a system has incorrect computation failure semantics if the probability that it misbehaves with a byzantine failure mode or authenticated byzantine failure mode is negligible.

As we hinted at a moment ago, we can turn a failure model into reality by restricting the failure semantics of a system and its components.

Restricting failure semantics means adding some mechanisms to a system to enforce particular external behaviors. If we again visualize a system's function as a yellow brick road through a service space, then we can think about the difference between unrestricted failure semantics and restricted failure semantics as follows. A system has **unrestricted failure semantics**—which is equivalent to byzantine failure semantics—if there are no bounds or limits on where outside the road the system's service may roam during a service outage. This is illustrated at the top of Figure 3.3. At time t_0 the system suffers a failure and from then onwards wanders aimlessly through the **failure subspace**, the part of the service space lying outside the system function. In contrast, a system has **restricted failure semantics** if during a service outage its misbehavior is restricted—the external states are erroneous, but not arbitrary. This is illustrated at the bottom of Figure 3.3. Again, the system suffers a failure at time t_0, which, as usual, we can think of as a departure from the road. But now the service is not allowed to freely roam through the failure subspace. Instead, the service is immediately catapulted into a fenced region of the failure subspace. This fenced region—the restricted failure mode—limits the external states that the system can deliver during a service outage. In other words,

[23] Flaviu Cristian. "Understanding Fault-Tolerant Distributed Systems". In: *Communications of the ACM* 34.2 (1991), pp. 56–78.

[24] Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, p. 26.

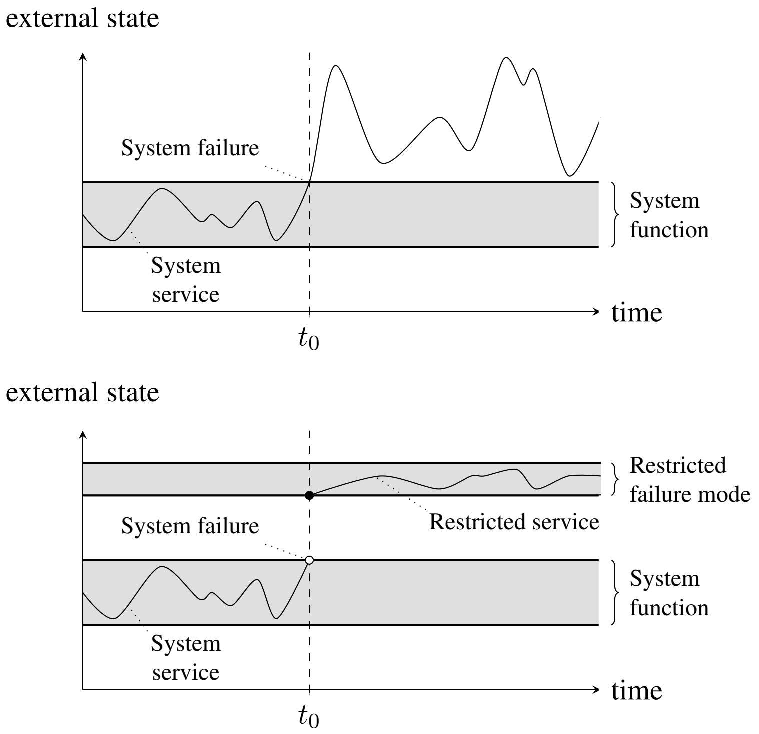


Figure 3.3: Example of restricting the failure mode of a system. The top of the figure shows the service corresponding to a system with byzantine failure semantics: when the system fails, there is no restriction on the external states it may deliver. The bottom of the figure shows the service after its failure semantics has been restricted: the system can no longer behave arbitrarily after it has suffered a failure, but it is restricted to a particular failure mode (which in general does not need to be a connected region).

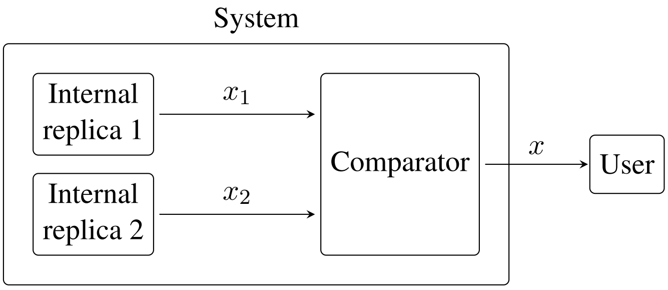


Figure 3.4: Restricting a system's failure semantics through internal duplication with comparison. As long as the internal replicas produce an equivalent service, the comparator forwards either service through the system's service interface; but as soon as the comparator detects that the services diverge, it provides a service belonging to a restricted failure mode.

the restricted failure mode confines the service to a particular misbehavior. For instance, if the external states of the restricted failure mode correspond to silence, then the system would have silent failure semantics and we would say that the system is **fail silent**.

So what mechanisms can we use to catapult a system's service into a restricted failure mode upon a failure? There are two that will be of particular interest to us: internal duplication with comparison is one, guardians are the other.

3.5.1 Internal Duplication with Comparison

Figure 3.4 shows schematically the architecture of a system whose failure semantics is restricted using internal duplication with comparison. The system has three components: two components that implement the system's function, which are the internal replicas 1 and 2 on the left, and one component that restricts the failure semantics, which is the comparator. The service of the first replica is labeled x_1, the service of the second replica is labeled x_2, and the service of the comparator—and of the system as a whole—is labeled x. Internal duplication with comparison then works as follows. As long as the services of the replicas are equivalent, the comparator simply forwards either service. But as soon as they are no longer equivalent, the comparator delivers a service corresponding to a restricted failure mode.

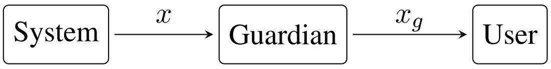


Figure 3.5: Restricting a system's failure semantics by means of a guardian. The guardian provides the same external states as the system unless it detects that the system's service is incorrect. When that happens, it produces a service corresponding to a restricted failure mode.

Symbolically we can express the service provided by the comparator as follows:

$$x(t+\Delta t) = \begin{cases} x_1(t), & \text{if } x_1(t) \equiv x_2(t), \\ s \in X_M, & \text{otherwise,} \end{cases}$$

where Δt is the comparator's processing delay, $\equiv$ denotes equivalence, and X_M denotes the set of external states belonging to a restricted failure mode M. In particular, if X_M corresponds to a set of states that a user would interpret as silence, then internal duplication with comparison makes a system fail silent. Well, at least it does so if the comparator itself is fail silent. Fortunately, a comparator can often be implemented as a device that is sufficiently simple to be fail silent.

While implementing a comparator that is fail silent may not be too hard, ensuring that the two internal replicas deliver an equivalent service in the absence of faults is more challenging. But it can be accomplished. For instance, if the replicas are CPUs executing some program, we can make them behave identically in the absence of faults by driving them with a single shared oscillator and otherwise equip them with identical hardware that is preloaded with identical software. As a result, we obtain two CPUs operating in what is called lockstep execution (Proenza[25] lists a few example CPUs that allow lockstep execution).

Another difficulty lies in how to ensure that if one of the replicas fails, it does so independently of the other. If complete independence is not possible, then we must at least ensure that if one replica fails and begins to deliver a family of erroneous states $\{x'_{t_0}, \ldots, x'_{t_1}, \ldots\}$, the other replica does not do the same at the same time. Otherwise, the comparator will determine the services of the replicas to still be equivalent and it will not restrict the failure semantics.

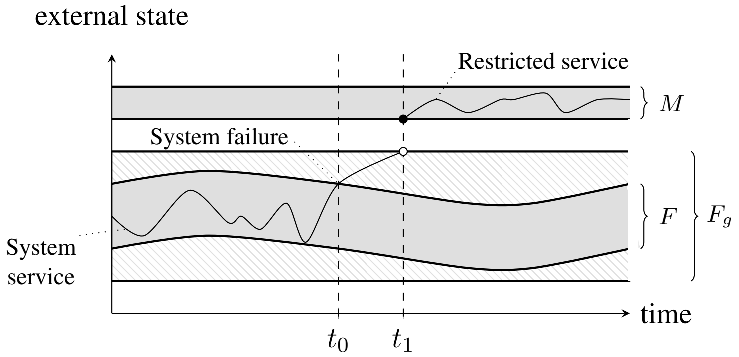


Figure 3.6: Visualizing the service restricted by a guardian in terms of a yellow brick road. At time t_0 the system suffers a failure, but the guardian does not detect this until time t_1, when the service deviates from F_g, which is the external behavior that the guardian considers correct. At that point the guardian restricts the service to the failure mode M.

3.5.2 Guardians

A guardian is a system component that monitors the service of another system and restricts the service by filtering out undesired misbehaviors. We can think of a guardian as a failure mode filter or, as Bauer, Kopetz, and Steiner[26] put it, as a failure mode converter: it converts an incorrect service belonging to one failure mode into an incorrect service corresponding to a more benign failure mode.

Figure 3.5 shows how a guardian is put between a system and its user to restrict the system's failure semantics. It monitors a system's service x, allowing x to reach the user only if it is correct or corresponds to a restricted failure mode.

Figure 3.6 illustrates how a guardian restricts a system's service to lie within a given region of the service space. The system's function F is a road through the service space. The system's function according to the guardian, i.e., F_g, is a wider road through the service space that includes F. The guardian's view is a wider road because a guardian does not know with absolute precision what the function of the system is. In Figure 3.6, a failure occurs at t_0, but the guardian does not consider the

[25] Proenza, "RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance", p. 84.

[26] Günther Bauer, Hermann Kopetz, and Wilfried Steiner. "Byzantine Fault Containment in TTP/C". in: *Proceedings of the 2002 International Workshop on Real-Time LANs in the Internet Age (RTLIA 2002)* (2002), p. 9.

system's service incorrect until time t_1, when it exits the road given by F_g. So it is only at time t_1 that the guardian restricts the service of the system to a failure mode M. As a result, the system's service is restricted to the region given by $F_g \cup M$.

The main difficulty with guardians is equipping them with a sufficiently accurate model of the system's function. Ideally, this model, i.e., the region F_g, should equal the system's actual function F. In practice, however, this is rarely possible without making the guardian itself as complex as the system it is monitoring—and once it is that complex, we have to start worrying about the failure modes of the guardian. Hence, F_g is usually a proper superset of F. In terms of the yellow brick road, F_g is a road that overlaps with F, but that is wider and that may include disjoint regions of the failure subspace. Note that if F_g is not a superset of F, then the guardian may wrongly restrict a correct service. Specifically, it would restrict any correct service that lies in the region $F \setminus F_g$.

For a concrete example, consider the following. Assume the external states of the system monitored by the guardian are messages transmitted through a communication link. The guardian is put in the middle of this link and starts monitoring the system's messages. Assume the messages contain a source address, a payload, and a destination address. If we wanted F_g to equal F, the guardian would have to know for every point in time what are valid source addresses, payloads, and destination addresses. But to know this, it may have to keep track of the history of exchanged messages, implement a state machine, be able to compute payloads itself, and so forth. Putting all this complexity into the guardian would make it about as unreliable as the supervised subsystem, which would seriously affect the reliability of the whole system. Hence, making $F_g = F$ is not feasible. However, we can make the guardian just monitor the source address of the messages the system sends. That way $F_g \supset F$, but the guardian can at least prevent impersonations by the system, without itself becoming too complex.

3.6 Single Points of Failure

Many systems have components whose failure may not disrupt the service of the whole system and that therefore are not critical for the system's service. Many systems, however, also have components whose failure inevitably leads to a whole system failure. Such components are called **single points of failure**: they are essential for the system's service and their failure is not tolerated.

Although single points of failure can be problematic, they do not have to be: a single point of failure may be an extremely reliable component. Unfortunately, however, digital systems tend to be unreliable. Hence, for highly reliable digital systems, single points of failure are rarely acceptable. The solution is replication, of which we will see more once we discuss replica determinism (Chapter 4).

During our analysis of the current FTT solutions (Chapter 7) one of our paramount tasks will be to identify if and what single points of failure each of these solutions has. In particular, we will have to identify what components, upon a failure, disrupt the communication service provided to the nodes. Then (Chapter 8), we will have to come up with a way of using replication to avoid any single points of failure in our own solution.

3.7 The Importance of Error Containment

We saw earlier (Section 3.3) that hardware replication is one type of spatial redundancy we can use to implement fault tolerance. In fact, if we want to tolerate the permanent failure of a hardware component that is a single point of failure, we *must* use hardware replication—temporal, information, and software redundancy are worthless in that case.

In the context of hardware replication, a set of replicated components, or replicas, is called a **replica group**. Hardware replication works best if the components of a replica group **fail independently**: if one replica fails, this does not increase the probability of another failing.

In terms of the chain of threats, for two replicas to fail independently, errors must not propagate from one to the other. Moreover, they must not be vulnerable to **common mode faults** , i.e., single faults that can cause the failure of multiple components. One example of the latter are **spatial proximity faults**, i.e., faults that affect multiple components due to how close they are to each other, as when some large object smashes into a system and destroys a whole set of components.

If we want to maximize the reliability achieved through replication, we must thus reduce the probability of errors propagating between replicas and of common mode faults occurring. Preventing error propagation is done by adding **error-containment mechanisms** to a system—mechanisms that prevent the propagation of errors between components. Common mode faults are typically prevented by minimizing the number of shared resources (e.g., shared power supplies) and by distributing the components within the system such that they are not too close to each other.

Since preventing the propagation of all errors is not always possible, error-containment mechanisms should focus, first of all, on preventing the propagation of the most destructive and probable errors. This is another reason why it is important to have accurate fault and failure models: if our models are inaccurate, we may misplace our efforts on preventing the propagation of errors that are unlikely or benign when compared to others.

Besides being important for hardware replication, error containment is also important whenever a system has components that are not single points of failure, i.e., components that are not critical for the system to function and that may thus fail without causing a global failure. In that case, to increase the system's reliability, we want to maximize the probability that if the non-critical component fails, this does not lead to a global failure. Again, we can accomplish this by using error containment.

So error containment is clearly important. But how do we contain errors? We have two approaches.

One approach is to restrict the failure semantics of the components and then prepare the rest of the system to deal with the restricted semantics.[27] This second step is crucial: it does us no good if all components are restricted to the fail-stop failure mode (the most benign failure mode), but the system is not prepared to deal with components exhibiting that failure mode. Now, how to prepare the system to deal with the restricted failure mode of its components depends on the particularities of the system. It depends on the number of components, their function, their failure semantics, how they are put together to form the system, the fault-tolerance techniques that we rely on, the system's function and its real-time and dependability requirements, and so forth. Hence, there is no general approach for preparing a system to deal with the restricted failure semantics of its components.

The other approach for containing errors, which we may use in combination with the previous one, is to make the system's components themselves fault tolerant. Then, if an error occurs within a component, the component itself tolerates it, without the component's service towards other components of the system being affected. In other words, errors within a component remain internal and never manifest externally. With this approach, contrary to the previous one, the rest of the system can remain oblivious of the errors that have occurred within a particular fault-tolerant component.

[27] Proenza, "RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance", p. 228.

3.8 When Fault Tolerance Meets Real-Time

We have several fault tolerance techniques to choose from: rollback, rollforward, error compensation, and fault handling. How do we decide which is most appropriate for us? Since we are designers of a system that must satisfy real-time requirements, one important consideration is how long it takes the system to tolerate a fault: if our system takes too long, the fault may lead to a service failure. But what is "too long"? There are two metrics we can use to decide what it means: grace time and failure recovery time.

The **grace time**[28] is the amount of time that a user can tolerate a service outage of a provider without itself suffering a failure. For instance, if the provider is a network and the user is a node on that network whose service is to broadcast an alarm at most 100 ms after its sensors have detected a fire, then if a single network outage lasts less than 100 ms, the node should be able to transmit its alarm on time when a fire occurs. Hence, the node's grace time is 100 ms.

The **failure recovery time**, or simply **recovery time**,[29] is the time it takes a provider to detect a service failure plus the time it takes the provider to restore correct service through the application of fault tolerance techniques such as rollback, rollforward, and fault handling.

To better understand these two concepts and how they relate to each other, let us consider two examples: one where the failure recovery time of a provider is larger than the grace time of the user and one where it is shorter.

Figure 3.7 on the next page shows two service spaces: one at the top corresponding to a service provider and one at the bottom corresponding to a user of this provider. The function of the provider and user are shown as shaded regions and, as we did earlier (Section 2.3), we can think of each such region as a yellow brick road on which the corresponding service should remain. As we can see, however, at time t_0 the provider exits its road, which corresponds to a service failure of the provider. At time t_3 the provider's fault tolerance mechanisms have restored the provider's service, but by this time the user has already suffered a failure at time t_2. The reason for the provider's temporal service outage causing a failure in the user is that the

[28] Minh Huynh, Stuart Goose, and Prasant Mohapatra. "Resilience technologies in Ethernet". In: *Computer Networks* 54.1 (2010), Sec. 5.4; Hubert Kirrmann, Mats Hansson, and Peter Muri. "IEC 62439 PRP: Bumpless Recovery for Highly Available, Hard Real-Time Industrial Networks". In: *Proceedings of the 12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2007)*. IEEE. 2007, Sec. 1.

[29] Lukasz Wisniewski et al. "A Survey of Ethernet Redundancy Methods for Real-Time Ethernet Networks and its Possible Improvements". In: *Fieldbuses and Networks in Industrial and Embedded Systems*. Vol. 8. 1. 2009, pp. 163–170, Sec. 2.

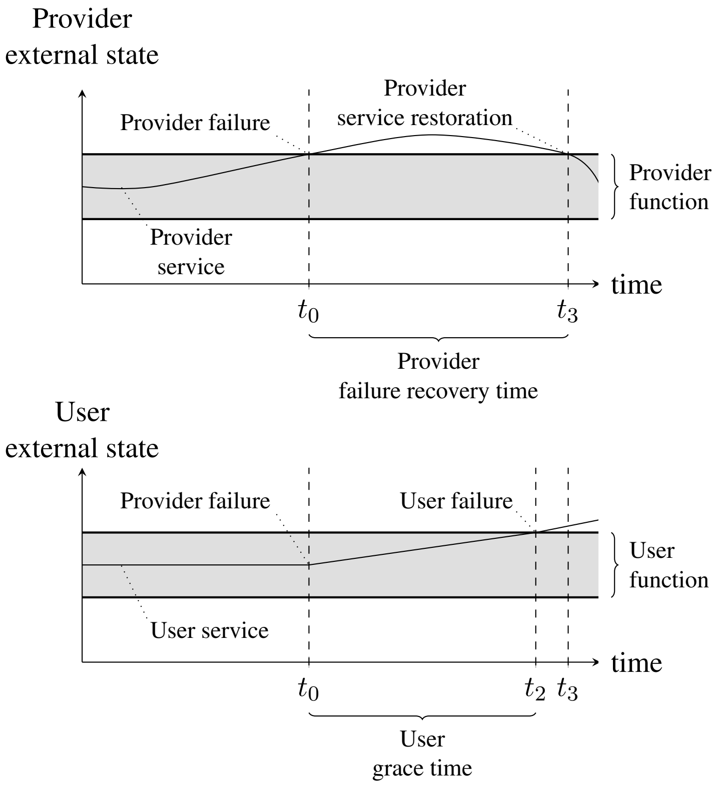


Figure 3.7: Illustration of how the failure recovery time of a provider exceeding the grace time of a user leads to the user's failure.

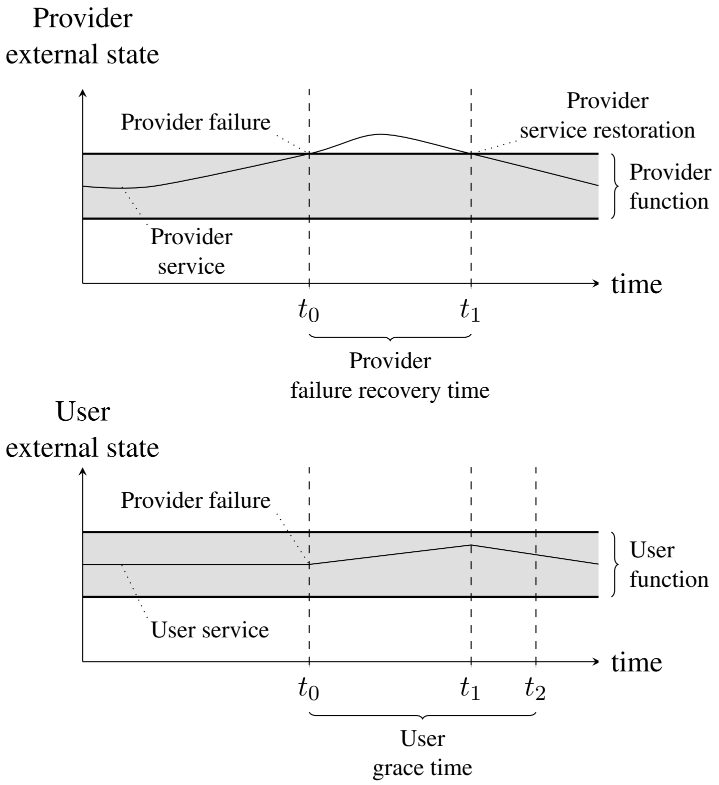


Figure 3.8: Illustration of how a provider's failure recovery time being shorter than the user's grace time avoids the user's failure.

provider's failure recovery time, $t_3 - t_0$, is larger than the user's grace time, $t_2 - t_0$.

Now for the second example. Figure 3.8 shows again two service spaces: one for a service provider at the top and one for a user of that provider's service at the bottom. Again, the provider suffers a failure at time t_0. This time, however, the user does not suffer a failure because the provider's service is restored at time t_1, which is before the end of the user's grace time at time t_2.

If the failure recovery time of a provider is zero, we say that it provides **seamless fault tolerance** or that it has **seamless redundancy**.[30]

Seamless fault tolerance can be ensured if we use error compensation—at least

[30] H. Kirrmann et al. "Seamless and Low-Cost Redundancy for Substation Automation Systems (High Availability Seamless Redundancy, HSR)". in: *Power and Energy Society General Meeting, 2011 IEEE*. July 2011, pp. 1–7, Sec. 2.2.

if done properly and before the redundancy has attrited. This is so because error compensation results in fault masking and hence in zero failure recovery times. Thus, error compensation is the fault tolerance technique of choice if we have users with the shortest of grace times. Rollback, rollforward, and fault handling, in contrast, are, in general, not guaranteed to deliver seamless fault tolerance. As we will see (Chapter 8), we will rely on error compensation above all other fault tolerance techniques to make our Ethernet-based communication subsystem tolerate faults seamlessly.

3.9 Discussion

We went through the main points to consider when designing a fault-tolerant system. Things to keep in mind include the following.

We need a fault model and failure model during the design, but getting these models right is difficult: if we are too optimistic, we will neglect designing the system to tolerate faults it is likely to encounter; but if we are too pessimistic, we may need so much redundancy that the system begins to become less reliable because of uncovered failures. Luckily, we can circumvent this problem to some extent by choosing a failure model that is not too pessimistic and then turning that failure model into reality by restricting the failure semantics of the system's components.

The failure of components that are single points of failure, unless they are highly reliable, needs to be tolerated. For this we can use hardware replication. However, replication will only be useful if under most circumstances the failure of one replica does not lead to the failure of all other replicas. Hence, error containment is extremely important to benefit from replication.

Finally, we saw that we cannot allow a real-time system to take too much time to recover from failures. Otherwise, any user of that system may itself fail. What constitutes "too much time" is given by the user's grace time and the system's recovery time.

Next, we will delve deeper into hardware replication. In particular, we will review how to ensure that replicas in a distributed system can correctly proceed with their service when any one of them fails.

Chapter 4

Replica Determinism

A man with a watch knows what time it is. A man with two watches is never sure.

Segal's law

As we saw in the introduction (Chapter 1), distributed embedded systems are embedded systems comprised of several nodes that cooperate through a network to perform a common function. For this they collect data from sensors, process that data into usable information, exchange that information, make decisions based on that information, and interact with the physical environment through actuators according to that information. There is information and more information and it needs to be consistent for the nodes to properly cooperate. Unfortunately, however, as soon as this information is available at more than one node, there is the potential for disagreement. And just like a man with inconsistent watches may get confused, a distributed embedded system with inconsistent information may get confused—and fail! And the problem is exacerbated if we replicate nodes: replicas not only need to duplicate hardware, but need to duplicate much of their internal state, increasing the amount of information that needs to be consistent among nodes. In distributed systems we therefore need to take special care in ensuring that the replicated nodes do not disagree. How to achieve this is what we will discuss next. This will prepare us for replicating the FTT master, a node that is critical in any FTT-based communication system and that constitutes a single point of failure in current Ethernet-based versions of FTT (in Chapter 7 we will further discuss the FTT master).

We will begin by discussing the main communication models used in message

Table 4.1: Summary of service access approaches.

Communication	Initiated by	Addressing
Client/server	Service user	Unicast
Producer/consumer	Service provider	Broadcast
Publisher/subscriber	Service provider	Multicast

passing systems, namely client/server, producer/consumer, and publisher/subscriber communication (Section 4.1). Then, drawing from Poledna's 1996 book, which assumes client/server communication, we will discuss replica determinism: the problem of ensuring that non-faulty replicas behave consistently towards their users so that they can replace each other in case of failure. We will define what replica determinism is (Section 4.2), what prevents it (Section 4.3), and how to ensure it (Section 4.4).

4.1 Communication Models for Message Passing Systems

As we know (Figure 2.10 on page 46), a message passing system is comprised of a set of nodes, a network, and message passing interfaces. In a message passing system, nodes do not directly interact with each other, but only through the network. Nevertheless, at times it will be convenient to think otherwise. In that case we may consider the network together with all message passing interfaces to be a single service interface and use interface shared by all nodes. Nodes then interact directly with each other using this shared interface. If we adopt this view now, we can distinguish three basic approaches with which nodes access each other's service, which are summarized in Table 4.1.

In **client/server communication** nodes can play the role of clients or servers. **Clients** are nodes that explicitly request a particular service from other nodes, called **servers**, which respond to the requests by providing their service. Since in a message passing system a service is a sequence of messages, we can also restate client/server communication as follows: clients are nodes that send messages to explicitly request a particular sequence of messages from other nodes, called servers, which respond to the request by transmitting the requested messages. The messages that a client sends are called **request messages**. The messages that the server responds with are called **response messages**. Client/server communication is based on **unicast** communication, i.e., one-to-one communication between nodes.

In **producer/consumer communication** nodes are classified as producers and consumers. **Producers** are nodes that broadcast messages without any node first requesting those messages: they provide a service to all other nodes, whether these want it or not. The other nodes, the **consumers**, receive the unsolicited messages and then decide whether to discard them or whether to act upon them in some way. Producer/consumer communication is based on **broadcast** communication, i.e., one-to-all communication.

In **publisher/subscriber communication** nodes are classified as publishers and subscribers. Each **publisher** is a node that initiates its service by sending messages to a subset of all other nodes, which are said to be **subscribers**. This approach is close to the producer/consumer approach, except that it is based on **multicast** communication, i.e., one-to-many communication. That is, instead of producers transmitting messages to all nodes, we now have publishers transmitting messages to subsets of nodes.

Some versions of FTT rely mostly on a producer/consumer model and others on a publisher/subscriber model (see Chapter 7). Moreover, all versions essentially use a client/server model when FTT slaves request the FTT master to update the current real-time parameters. Specifically, when a slave wants the real-time parameters to change, it acts as a client that sends an update request to the master, which in turn acts as a server by sending a response. The main difference with respect to a proper client/server approach is that the response may be broadcast to all slaves.

Client/server communication is the context under which Poledna discusses replica determinism.[1] Since the operational flexibility of FTT is essentially enabled by client/server communication between the slaves and the master, what Poledna has to say about replica determinism in the context of client/server communication will be useful to us when we replicate the FTT master.

4.2 What Is Replica Determinism?

A **replica group** is a set of replicas: a set of replicated components that work together to provide a single correct service even if some of them fail. For instance, a replica group might comprise several replicated nodes that perform the same computations and then vote on the results to obtain error compensation. The voted results then constitute the single correct service delivered by the group. Ideally, a user of the group should see the whole group as a single provider, but one that is more reliable than any single replica.

(The replicas of a group could, in principle, be any type of component—hardware

[1] Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism.*

or software. In this chapter, however, and following Poledna's lead, we will be referring only to the replication of nodes.)

For the replicas of a group to be able to provide a common service, and continue to provide it despite the failure of any of them, they need to be *replica deterministic*. What does this mean? Well, what exactly it means depends on the common service that the replica group provides, on how the replicas collaborate to provide that common service, and on how the corresponding users use that service. For that reason, Poledna adopted a generic and deliberately vague definition of replica determinism. According to him, a replica group exhibits replica determinism if "correct servers show *correspondence* of server outputs and/or service state changes under the assumption that all servers within a group start in the same initial state, executing *corresponding* service requests within a *given time interval"*.[2]

In his definition, Poledna uses the words "server" and "requests". This reveals that he studied the problem of replica determinism under the assumption of client/server communication. Since FTT only follows a client/server approach for some aspects of the communication, let us use a more general wording and define replica determinism as follows.

A replica group exhibits **replica determinism**, or is **replica deterministic**, if correct replicas show *correspondence* in their outputs and/or service state changes under the assumption that all replicas within a group start in the same initial state and process *corresponding* inputs within a *given time interval*.

With this definition, a replica does not necessarily need to be a server.

The emphasis in our definition, as in Poledna's original, is to highlight the intentional vagueness of what is meant by "correspondence", "corresponding", and "given time interval". For instance, correspondence in outputs may mean that replicated nodes must transmit identical messages. But it may also mean that they must only transmit messages that a user would consider equivalent. For some replication techniques, such as when some of the replicas are backups, it may even mean that only one replica transmits messages, while all others remain silent. Similarly, corresponding inputs may mean that all replicas must receive identical messages, but it may also mean that the messages just need to be equivalent, or, perhaps, some replicas may not even have to receive some messages at all. "Given time interval" is left undefined as well: maybe the inputs need to be processed by all replicas within the same second, or maybe they need to be processed within a millisecond or microsecond; or perhaps synchronization does not matter much at all and it suffices for all nodes to process their inputs within the same minute or even

[2] Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, p. 33. Emphasis in the original.

hour. Again, what “correspondence”, “corresponding”, and “given time interval” means depends on the particularities of the system and its service.

Given its vagueness, the concept of replica determinism must be made concrete for a particular replica group. This is done by defining a **correspondency requirement** in value and time for the replicas:[3] we, as system designers, must define when we consider that in our system replicas have “correspondence in their outputs and/or service state changes”, when we consider that the replicas are “processing corresponding inputs”, and what we consider to be appropriate time intervals for these tasks.

Once we have clarified what exactly replica determinism means for a particular replica group, we must make sure that the group starts out being replica deterministic, and, then, remains so. Otherwise, the group will not be able to tolerate the failure of any one of its members. So let us see what, in general, may prevent a group of replicated nodes from being replica deterministic.

4.3 What Prevents Replica Determinism?

Poledna identified several possible causes for replicas to behave nondeterministically. His initial list does not claim to be exhaustive, but illustrative. Some of these causes include the following:[4]

- Inconsistent inputs: since the output of a node is usually a function of its inputs, if two replicas receive inconsistent inputs, they are likely to produce inconsistent outputs, i.e., outputs that may not satisfy our correspondency requirement.
- Inconsistent order: if the order of inputs matters, then even if two replicas receive the same set of inputs, their outputs may violate our correspondency requirement if they are processing the inputs in a different order.
- Message transmission delays: the nodes in a distributed system exchange messages through a network and these messages may reach some nodes sooner than others. This may lead to a loss of replica determinism if a replica’s output is a function of when it receives a particular message.
- Nondeterministic program constructs: as we saw (Figure 2.10 on page 46), nodes have an internal state and logic. This means that replica nondeter-

[3] Ibid., p. 33.

[4] Ibid., pp. 35–41.

minism can arise from replicated nodes if these use nondeterministic logic. For instance, it may arise if the software executed by the replicas uses statements that are nondeterministic, such as the `select` statement in the Ada programming language.[5]

- Local information: if a replica produces outputs based on information that only it knows, its output is likely to not correspond to the output of the other replicas.

Since this list is not exhaustive, simply avoiding each of the above causes will not guarantee that replicas will be deterministic. We could extend the list with a few more examples,[6] but this would still not ensure that we are taking into account all possible sources of replica nondeterminism. What we have to do is to consider the primary—or ultimate—causes of replica nondeterminism. This is precisely what Poledna did.

Poledna referred to these primary causes as **atomic sources of replica nondeterminism** and identified the following ones:[7]

- The real world abstraction limitation.
- Impossibility of exact agreement.
- Intention and missing coordination.

Let us discuss each of them.

The real world abstraction limitation. Without getting into the debate of whether reality is fundamentally analog or digital, the computing nodes we are considering are digital and many of the real-world quantities they measure and manipulate (temperature, pressure, voltage levels, etc.) are at a non-quantum scale and decidedly analog. Thus, as Poledna pointed out,[8] replicated nodes have to quantize analog values using finite sets of discrete numbers and this is inevitably a source of replica nondeterminism: tiny differences in the measured values, induced, e.g., by noise or the limited precision of sensors, may lead different nodes to map what should

[5] Proenza, "RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance", p. 57.

[6] Poledna has a few more, such as inconsistent membership information, dynamic scheduling decisions, and timeouts. See Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, pp. 35–41.

[7] Ibid., p. 45.

[8] Ibid., Sec. 4.3.1.

be equivalent measurements to different discrete numbers. Moreover, as Poledna indicated as well,[9] replica nondeterminism may also stem from nodes differing

- in how they store the same analog quantity as 0's and 1's (e.g., they may differ in their floating-point number representation),
- in how they round and truncate results of arithmetic operations, and
- in which order they evaluate arithmetic expressions (remember that floating-point arithmetic is not necessarily associative[10]).

Impossibility of exact agreement. Poledna further pointed out that the inevitable replica nondeterminism due to the quantization of analog quantities cannot be totally eliminated by any systematic procedure.[11] That is, once tiny deviations in measurements have let replicas to disagree on the value of a real-world quantity, there is no general way for them to resolve their disagreement such that they subsequently agree exactly all the time. According to Poledna we only have two options: either we demonstrate that disagreement does not significantly affect the service of the system (e.g., because replicas are so trivial that they do not have any inputs nor changing internal states) or we settle on having replicas agree only some of the time by having them regularly exchange information. What "some of the time" means and what exactly it means for replicas to agree depends on the application. For instance, one application may require replicas to agree exactly on the value of a floating-point number every few seconds, while for another it may suffice for the replicas to agree on the order of magnitude of a value every few hours. The important point is that replicas must reach agreement whenever they are about to make a decision that may lead them to provide diverging services to their users.

Intention and missing coordination. Finally, Poledna also identified as an atomic source of replica nondeterminism failing to properly design a replica. Specifically, a system designer may intentionally introduce nondeterminism in the replicas or fail to add mechanisms to coordinate nondeterministic actions among replicas.[12] For instance, an example of intentional insertion of replica nondeterminism is a designer deciding to use true random number generators. An example of missing coordination is a designer implementing replicas using nondeterministic program

[9] Ibid., pp. 48–49.

[10] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". In: *ACM Computing Surveys (CSUR)* 23.1 (1991).

[11] Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, Sec. 4.3.2.

[12] Ibid., Sec. 4.3.3.

constructs (see page 77) and then forgetting to properly coordinate the replicas. Fortunately, failing to properly design the replicas can be avoided: we just have to ensure that we do not intentionally introduce nondeterminism nor fail to coordinate replicas that can take nondeterministic actions.

Poledna concluded that whenever replicas fail to be replica deterministic, this can always be traced back to one of the above three atomic sources of replica nondeterminism. Thus, according to Poledna, the real world abstraction limitation, the impossibility of exact agreement, and the intentional introduction of nondeterminism and the failure to properly coordinate replicas constitute an exhaustive list of all ultimate causes for replica nondeterminism. (Unfortunately, however, he did not, and could not, provide a formal proof. This is so because, as he put it, "there is no exact and formal definition of replica determinism" and hence "there is no possibility to define an induction rule or to give a [proof by] contradiction".[13])

4.4 How To Enforce Replica Determinism

Enforcing replica determinism means making sure that the replicas of a group are replica deterministic. Techniques that help us with this task are called **replica control techniques**.[14]

Internal replica control techniques help us enforce replica determinism among replicas by preventing them from being internally nondeterministic. Replicas are **internally nondeterministic**, or have **internal nondeterminism**, if they may lose replica determinism even if all replicas receive the exact same inputs at the exact same time. Essentially, a replica is internally nondeterministic if there are causes within the replica itself—within its boundary—that may prevent it from being replica deterministic with respect to its replica group. That is, a replica is internally nondeterministic if it is implemented with components that exhibit nondeterministic behavior. Internal nondeterminism can always be traced back to Poledna's third atomic source: intention and missing coordination.

Internal replica control techniques generally consist in avoiding the causes of internal nondeterminism, e.g., avoiding nondeterministic program constructs and exclusively using information that is inherently global. Internal replica control may, however, also involve active steps such as adding mechanisms that allow replicas to coordinate nondeterministic decisions.

External replica control techniques are those that help ensure that replicas

[13] Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, p. 45.

[14] Ibid., p. 61.

process corresponding inputs. For example, and returning to the initial list of causes of replica nondeterminism that we saw a moment ago (Section 4.3), external replica control may fight against inconsistent inputs, inconsistent order, and message transmission delays. More generally, external replica control targets Poledna's first two atomic sources of replica nondeterminism: the real world abstraction limitation and the impossibility of exact agreement.

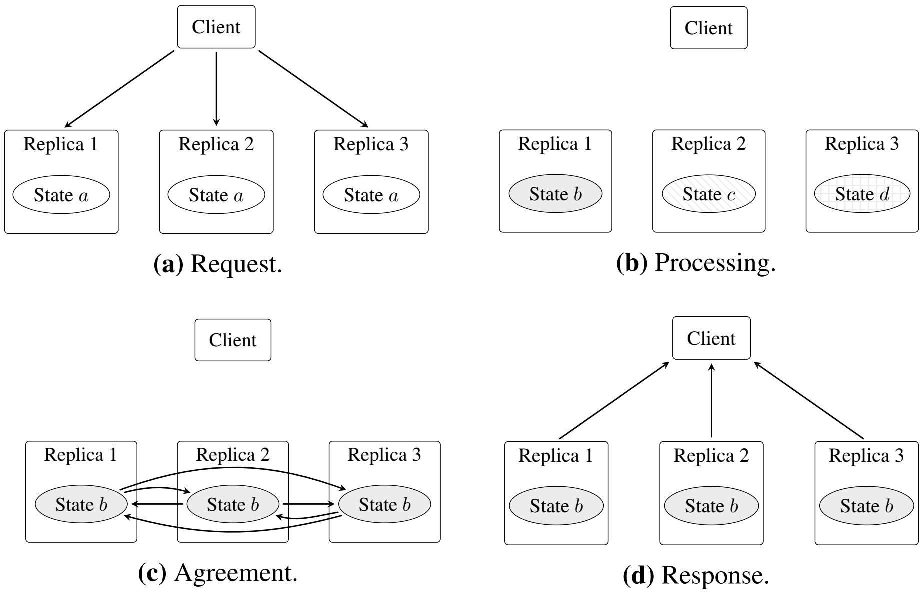
Examples of external replica control techniques are communication protocols that provide reliable broadcast or total-order broadcast, which is also called atomic broadcast. (Reliable broadcast essentially means that whenever a correct node sends a message to all other nodes, the correct ones all eventually deliver the message to the locally running application; total-order broadcast means that in addition all correct nodes also deliver the set of broadcast messages in the same order.[15])

Since for our fault-tolerant communication subsystem we will replicate FTT masters and these update the real-time communication parameters following an approach similar to the client/server approach, it will be instructive to review the four basic external replica control techniques that we can use for client/server communication: active replication, passive replication, semi-active replication, and semi-passive replication. In the following discussion we should keep in mind that replica determinism is a property that only applies to non-faulty replicas.

4.4.1 Active Replication

Active replication consists in having all replicas of a group provide the same service concurrently, with no replica being singled out as a privileged one with respect to the others.[16] A client sends the same request to all replicas and each replica then processes the request on its own and sends a response back.

To ensure that the replicas deliver a corresponding service to the client, and hence are replica deterministic, we have two main options for active replication. One is based on reconciling the internal states of the replicas and the other is based on the use of a total-order broadcast protocol.



(a) Request.

(b) Processing.

(c) Agreement.

(d) Response.

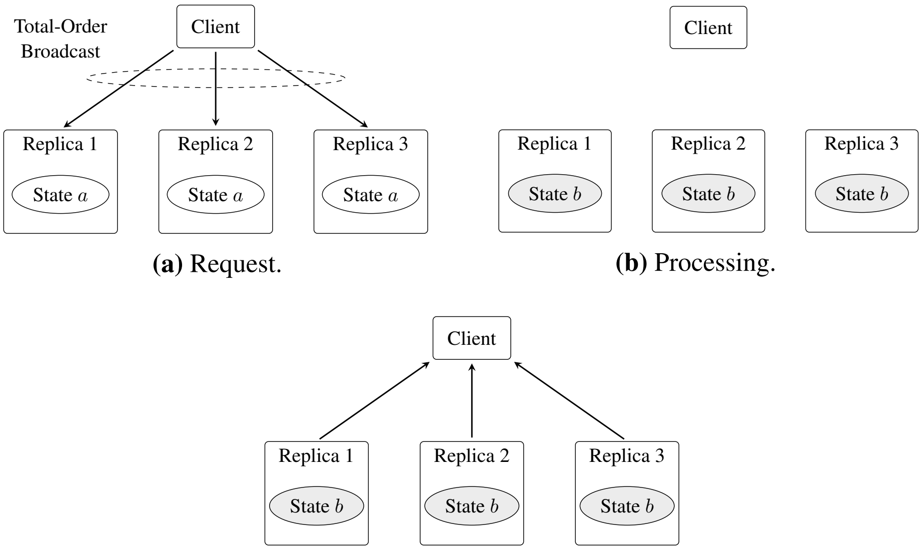
Figure 4.1: Active replication with state reconciliation.

Active Replication Using State Reconciliation

In Figure 4.1 we can see how active replication using state reconciliation works. Each replica has an internal state, which initially is consistent among the replica group. Now, when a client requests the service provided by the group, it sends equivalent request messages to each of the replicas, as shown in subfigure (a). Due to errors in the network, some replicas may receive a corrupted request or no request at all, while others receive a correct request. Hence, when each replica subsequently processes the request it received, the group may reach inconsistent internal states, as we can see in subfigure (b). Moreover, even if there are no network errors, correct replicas may reach inconsistent internal states if the processing of the requests involves nondeterministic actions. Either way, after the processing the replicas must reconcile their states to ensure that they send corresponding responses back. This reconciliation is accomplished through the execution of an **agreement protocol**: a network protocol whose goal is to ensure that the replicas agree on the same internal state. This is illustrated in subfigure (c). Finally, once an agreement is reached, each replica sends a response back to the client, as subfigure (d) shows.

[15] For the formal definitions see Xavier Défago, André Schiper, and Péter Urbán. "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey". In: *ACM Computing Surveys* 36 (2003), Sec. 2.3.

[16] Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, p. 106.



(a) Request. **(b)** Processing. **(c)** Response.

Figure 4.2: Active replication with total-order broadcast and deterministic replicas.

For this last step to obey replica determinism, the response must be computed in a deterministic way. Furthermore, we must provide some mechanism to ensure that network errors affecting the response messages do not compromise the replica determinism. One solution is to have the replicas add an error detection code to the response messages so that the client can discard corrupted ones. If replicas have at most omission failure semantics, this allows the replica group to mask the failure of any individual replica, making this technique adequate when the failure of replicas should be tolerated seamlessly. If replicas can fail in more malicious ways, this needs to be taken into account by the agreement protocol and the protocol used to send responses back. The client may also have to take some action to discard faulty responses. For instance, it may apply some voting algorithm to the responses.

Using active replication with state reconciliation yields a tight synchronization among all correct replicas: towards the client they act in unison.

Active Replication Using Total-Order Broadcast

In active replication with total-order broadcast, the replicas start out with the same internal state, are internally deterministic, and relevant state changes are only triggered by requests. This means that a replica's responses only depend on the replica's initial state and the sequence of requests it has processed and responded to previously. Hence, if we can ensure that all replicas receive the exact same inputs in the same order, the replica group will be replica deterministic. This is the idea behind active replication using total-order broadcast.

Figure 4.2 illustrates how it works. The client sends the requests using a total-order broadcast primitive, as shown in subfigure (a). Sending requests using **total-order broadcast** essentially means that all non-faulty replicas will deliver for internal processing every request sent by a non-faulty client, and do this delivery in the same order.[17] Next, each replica processes the request it received from the client. We can see this in subfigure (b). Since the replicas are internally deterministic and the request was sent using a total-order broadcast protocol, all replicas reach the same internal state. Thus, contrary to the previous approach, it is not necessary to execute an agreement protocol among the replicas. Instead, the replicas can simply compute a response message and send it back to the client, as illustrated in subfigure (c).

As with the previous approach, the response must be sent with a protocol or other mechanism that does not compromise the replica determinism. If the messages are corrupted during transmission, this needs to be detectable. In that case, if the replicas have at most omission failure semantics, all the client has to do is to accept any one of the responses. If the replicas can suffer more malicious failures, then, as in active replication with state reconciliation, the responses must be sent back with an appropriate protocol and the client may have to take some actions to discard incorrect responses, such as applying some voting algorithm to the responses.

Finally, in active replication with total-order broadcast, the synchronization among the replicas is also tight and allows zero failure recovery times, i.e., seamless fault tolerance.

Active replication using total-order broadcast is also known as the state machine approach.[18]

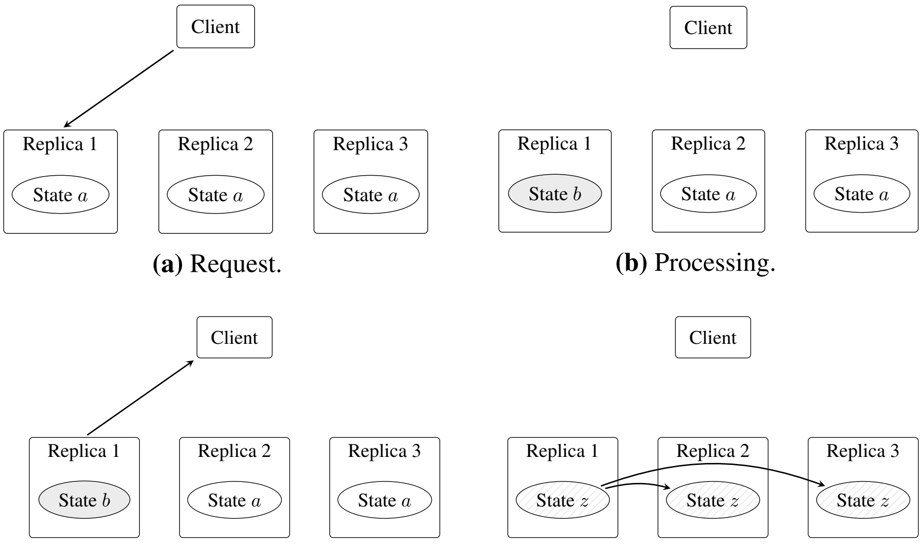
4.4.2 Passive Replication

Passive replication consists in having only one replica of a group provide the service, while the others act as backups.[19] This is illustrated in Figure 4.3. When a client requests a service, it does so from a privileged replica called the **primary**

[17] A survey of several algorithms that implement total-order broadcast can be found in Défago, Schiper, and Urbán, “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey”.

[18] Fred B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[19] Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing*.



(a) Request.

(b) Processing.

(c) Response.

(d) Checkpointing.

Figure 4.3: Passive replication.

replica, which is the leftmost replica in subfigure (a). The other replicas, called **backup replicas** or **standby replicas**, do not receive the request from the client. The primary replica then processes the request and updates any required internal state, as shown in subfigure (b). Next, the primary replica computes a response and sends it back to the client. This is illustrated in subfigure (c). Finally, the primary also updates the internal state of the backups by sending them appropriate messages, as shown in subfigure (d). This update process is called checkpointing and ensures that the backups' internal state remains up-to-date.[20] Checkpointing can be done systematically whenever the primary sends a response or it can be done periodically.[21] Systematic checkpointing has high overhead during normal operation, but allows a short failure recovery time. With a low checkpointing frequency, periodic checkpointing can have less overhead during normal operation, but this then increases the failure recovery time.

Contrary to active replication with total-order broadcast, passive replication does not require the processing of requests to be deterministic. Any nondeterministic decisions are taken by the primary and any resulting state is at some point transferred from the primary to the backups.

The main disadvantage of passive replication is that the primary replica may

[20] Ibid., p. 111.

[21] Ibid., p. 112.

constitute a single point of failure: it may fail in a way such that it propagates incorrect states to the backups during the checkpointing phase, thus leading to a failure of the whole replica group. Therefore, in passive replication the replicas typically cannot have byzantine failure semantics as in active replication, but must be fail silent, i.e., have crash failure semantics.

Another disadvantage of passive replication is that its failure recovery time can be quite long. When the primary replica fails, this needs to be detected by the client and the backups. Then, the client together with all surviving backups need to reach an agreement on which replicas remain non-faulty within the group and which replica should become the new primary. For reaching such an agreement it also helps if the replicas are fail silent. Moreover, the new primary's internal state may be out of date and some update process may need to be initiated—for instance, the new primary may have to communicate to the client the last request it knows about and the client may have to retransmit any lost requests. Hence, passive replication is generally inadequate for seamless fault tolerance.

Passive replication is also known as the primary/backup approach.[22]

4.4.3 Semi-Active Replication

Semi-active replication is similar to active replication in that all replicas of a group process client requests and send responses back.[23] The difference lies in that one of the replicas plays a privileged role with respect to the others. Specifically, there is one replica, typically called the **leader**, that makes all nondeterministic decisions and ensures consensus by communicating these decisions to the others, which are called **followers**.[24]

The approach is illustrated in Figure 4.4. As in active replication, the client sends a request to all replicas—subfigure (a)—and all replicas process the request on their own—subfigure (b). Since the request was not sent using a total-order broadcast and the replicas are not necessarily internally deterministic, the internal states after the processing may be different among the replicas. Hence, to reach a consensus the leader communicates its nondeterministic decisions to the followers by sending appropriate messages. This is shown in subfigure (c). Finally, depending on how the technique is implemented, either only the leader sends a response back, or all replicas—leader and followers—as shown in subfigure (d).

[22] Navin Budhiraja et al. "The Primary-Backup Approach". In: *Distributed systems* 2 (1993), pp. 199–216.

[23] Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, p. 116.

[24] Ibid., p. 116.

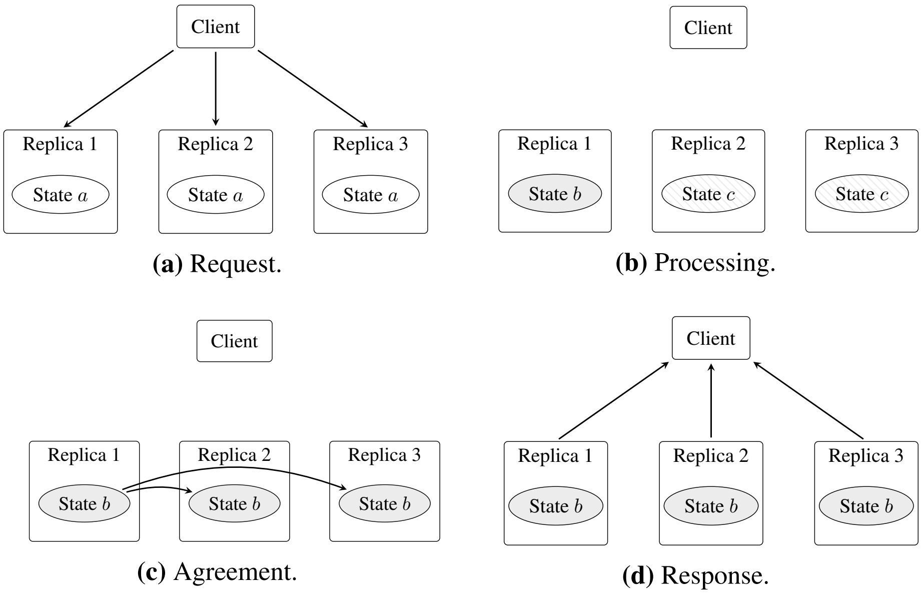
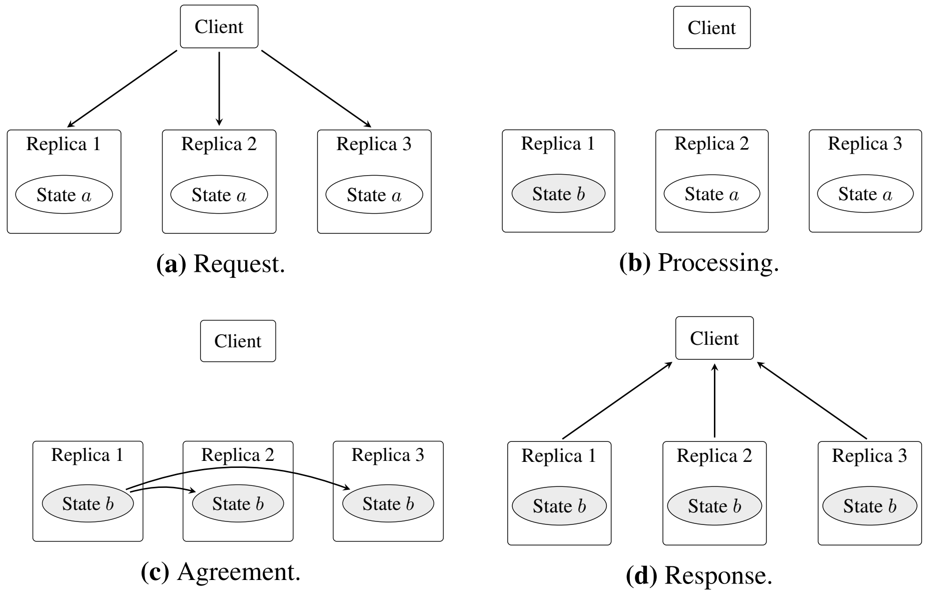


Figure 4.4: Semi-active replication.

The main advantage of semi-active replication over active replication is that it allows nondeterministic replicas without requiring a costly agreement protocol among all replicas.

The main disadvantage is that the leader, like the primary in passive replication, may be a single point of failure: if it fails such that it propagates an incorrect state during the agreement phase, the whole replica group may become faulty. For this reason, in semi-active replication the replicas generally have to be fail silent—although under certain conditions, the failure semantics can be relaxed.[25] Moreover, similar to passive replication, when the leader fails, the replica group needs to initiate a potentially time-intensive election process to decide upon a new leader. However, it may take less time than in passive replication since the client does not need to be involved in the election process—it sends its requests to all replicas anyway. Furthermore, the synchronization among the replicas is tighter than in passive replication: the internal states of the followers only become out-of-date if nondeterministic decisions have to be taken by the leader and consensus is reached before each response by having the leader only transfer the nondeterministic decisions, instead of the whole internal state as in passive replication. Hence, during a recovery it generally takes less time to bring the new leader up-to-date.

[25] David Powell. "Distributed Fault Tolerance: Lessons from Delta-4". In: *IEEE Micro* 14.1 (1994), pp. 36–47.



(a) Request.

(b) Processing.

(c) Agreement.

(d) Response.

Figure 4.5: Semi-passive replication.

4.4.4 Semi-Passive Replication

In **semi-passive replication**, like in passive replication, but distinct from active and semi-active replication, only one replica processes the client requests. As in passive replication, this replica is called the primary. The client, however, does not know which replica is the primary one. Hence, it sends any request to all replicas, as shown in Figure 4.5 (a). Since the processing is done only by the primary (replica 1 in the figure), the processing of the requests does not need to be deterministic. After the processing—subfigure (b)—the primary communicates its internal state to the backup replicas—subfigure (c). Finally, the whole group sends a response back to the client (d).

To avoid long failure recovery times when the primary fails, Défago, Schiper, and Sergent,[26] who introduced the semi-passive replication approach, chose to avoid a costly election protocol. Instead, who is the primary at any given point in time is determined using the rotating coordinator paradigm:[27] the role of primary is rotated

[26] Xavier Défago, André Schiper, and Nicole Sergent. "Semi-Passive Replication". In: *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS 1998)*. 96. 1998.

[27] Péter Urbán et al. "Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pp. 4–17.

Table 4.2: Typical characteristics of the different replication techniques. See Section 4.4.5 for details.

Replication technique	Failure recovery time	Internal non-determinism	Failure semantics	Synch.
Active (reconciliation)	Seamless	Allowed	Byzantine	Tight
Active (total order)	Seamless	Forbidden	Byzantine	Tight
Passive	Long	Allowed	Fail silent	Loose
Semi-active	Short	Allowed	Fail silent	Medium
Semi-passive	Short	Allowed	Fail silent	Medium

among the replicas and when the current primary fails, the one to take over is the replica that is next in the preestablished rotation schedule.

As in the other replication techniques where one replica has a privileged role, in semi-passive replication the replicas generally also need to be fail silent to avoid a faulty primary to propagate incorrect states to the backups. Nevertheless, it is also possible to have semi-passive replication in the presence of byzantine failures.[28]

4.4.5 A Brief Comparison of the Replication Techniques

Table 4.2 summarizes the main characteristics of the five replication techniques we have seen. The first column indicates the replication technique. The second column indicates how long the failure recovery time typically is. The third column indicates whether the replicas are allowed to have internal nondeterminism. The fourth column indicates what failure semantics are typically acceptable for the replicas. The last column indicates how tight the synchronization among the replicas is: the tighter the synchronization, the more up-to-date the internal states of all replicas are at any given point in time.

The table shows typical values and should not be taken as indisputable. For particular designs and implementations the attributes may be different. For instance, passive replication can have tight synchronization if systematic checkpointing is used.

[28] HariGovind V. Ramasamy, Adnan Agbaria, and William H. Sanders. *Semi-Passive Replication in the Presence of Byzantine Faults*. Tech. rep. 2004.

4.5 Discussion

In a distributed system the replicas are typically nodes that are physically separated and that, besides the network, do not share any physical resources (power supplies, clocks, processors, etc.). Hence, it is generally easier to provide error containment among the nodes of a distributed system than among the components of a centralized system (e.g., among the components of a single computer). In that sense, distributed systems have an advantage. Unfortunately, however, the advantage of not sharing physical resources also makes it harder to ensure that the replicated nodes are replica deterministic. Why? Mainly because of three disadvantages of distributed systems: the communication among nodes through a network is unreliable when compared with the communication among components within a computer; message transmission delays are non-negligible; and there is inexact global state because the events occur faster than the communication among nodes.[29] In essence, ensuring replica determinism in distributed systems is more difficult because it is harder to guarantee that replicas process corresponding inputs within a given time interval—the inputs are mostly messages and these may get lost, be delayed, arrive out of order, become corrupted, and so forth.

Despite the additional difficulties of achieving replica determinism in distributed systems, it can be achieved. For this, we first of all need to identify what exactly constitutes replica determinism in our particular system. We do this by defining correspondency requirements for the inputs and outputs of the replicas. Then we use appropriate replica control techniques to enforce replica determinism.

In a system where the nodes access each other's service using a client/server approach, we have five basic replication techniques at our disposal that provide replica control. Later (Chapter 8), we will base our approach to replicate FTT masters on some of these techniques.

Next, we will cover the basics of real-time communication so that when we discuss how FTT works (Chapter 7) we will have an agreed-upon vocabulary.

[29] Paulo Veríssimo. *Comunicação e Computação em Sistemas Distribuídos - Lição de Síntese*. URL: http://www.navigators.di.fc.ul.pt/docs/abstracts/ccsd-aula.html (visited on 2016-12-10), pp. 2–3; Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, p. 23; Proenza, "RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance", p. 52.

Chapter 5

Real-Time Communication

> It's weird how I am constantly surprised by the passage of time when it's literally the most predictable thing in the Universe.
>
> Randall Munroe, *xkcd 1477: Star Wars*

As stated in the introduction (Chapter 1), our aim is to show that the FTT paradigm used on Ethernet can be made to tolerate transient and permanent faults; and that this can be done without sacrificing the support for real-time communication and the operational flexibility towards changing real-time requirements. This means that although our focus will be on reliability and fault tolerance, we will still have to deal with some aspects of real-time communication. In particular, we will have to ensure that the fault-tolerance mechanisms we design (Chapter 8) do not compromise FTT's real-time guarantees. Moreover, when we discuss the FTT paradigm and its different implementations (Chapter 7), we will need some common background on real-time communication. The goal now is to introduce this background.

We will begin with a discussion of the main parameters of real-time messages (Section 5.1). Then we will discuss the three main types of real-time messages: periodic, aperiodic, and sporadic messages (Section 5.2). Next, we will discuss the basics of real-time traffic scheduling (Section 5.3), which is one of the main tasks performed by an FTT master. Finally, we will discuss the two classic communication paradigms that FTT incorporates: event- and time-triggered communication (Section 5.4).

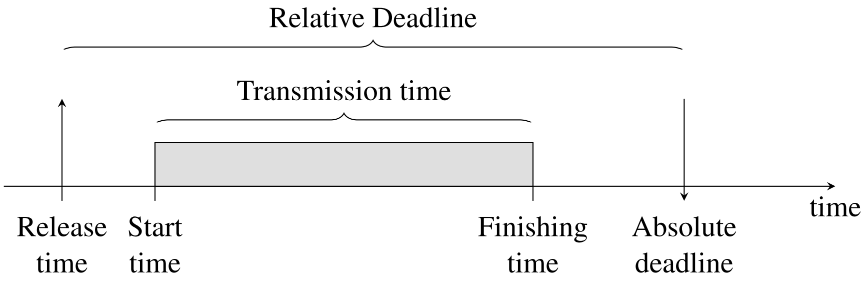


Figure 5.1: Parameters of a real-time message. (Based on a figure by Buttazzo in *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, p. 27, which illustrates the typical parameters of a real-time task.)

5.1 Parameters of a Real-Time Message

A **real-time message** is a message that should be exchanged on time, that should not be delayed arbitrarily. A **non-real-time message**, in contrast, is a message for which no timing constraints exist.

Real-time messages are characterized by temporal parameters. The main ones are summarized in Figure 5.1 and are the following:

- **Release time**: this is the time at which the message becomes available for transmission. For instance, if the message conveys the result of some computation carried out by a node, then the release time might be the time at which the computed result is ready to be transmitted by the node. Release time is sometimes also called request time or activation time (or arrival time, but I find this confusing for real-time messages as it may be mistaken for the time at which the message arrives at its final destination).
- **Transmission time**: this is the time it takes to transmit the message. For instance, if the network operates at 100 Mbps and the message has a length of 512 bits, then the transmission time is 5.12 µs.
- **Message transmission start time**: this is the instant of time at which the transmission of the message begins. It does usually not coincide with the release time because usually there are at least some queuing and network access delays that postpone the start of a transmission.
- **Message transmission finishing time**: this is the instant of time at which the transmission of the message completes.

- **Absolute deadline**: this is the instant of time before which the message must have been transmitted in its entirety.
- **Relative deadline**: this is the amount of time after the release time that is available for the message to be transmitted in its entirety.

(For now we can assume that all intervals and instants of time are measured according to absolute Newtonian time; that is, "time is considered an external and continuous dimension that is perceived equally everywhere".[1] In practice, however, nodes do not have access to a source of absolute time. Instead, each node has its own clock that needs to be synchronized, in time or frequency, with the clocks of all other nodes.[2])

A message **meets** its deadline if its finishing time is less than or equal to the absolute deadline; whereas a message **misses** its deadline if its finishing time is greater than the absolute deadline. What happens if a message misses its deadline depends on the deadline's **criticality**, a measure of how important it is that the deadline is not missed. This leads to the classification of deadlines, whether absolute or relative, into three distinct classes: hard deadlines, firm deadlines, and soft deadlines.

Hard deadlines are those that, if missed, may cause a system failure.[3] **Firm deadlines** are those that, if missed, imply that the corresponding overdue messages are useless to the recipient or any other node. **Soft deadlines** are those that, if missed, imply that the corresponding overdue messages are still somewhat useful.

According to this classification, the messages themselves are also said to be hard, firm, or soft. That is, a **hard real-time message** is one whose deadlines are hard, a **firm real-time message** is one whose deadlines are firm, and a **soft real-time message** is one whose deadlines are soft. Messages that do not have deadlines are called non-real-time messages, which agrees with the earlier definition on page 92.

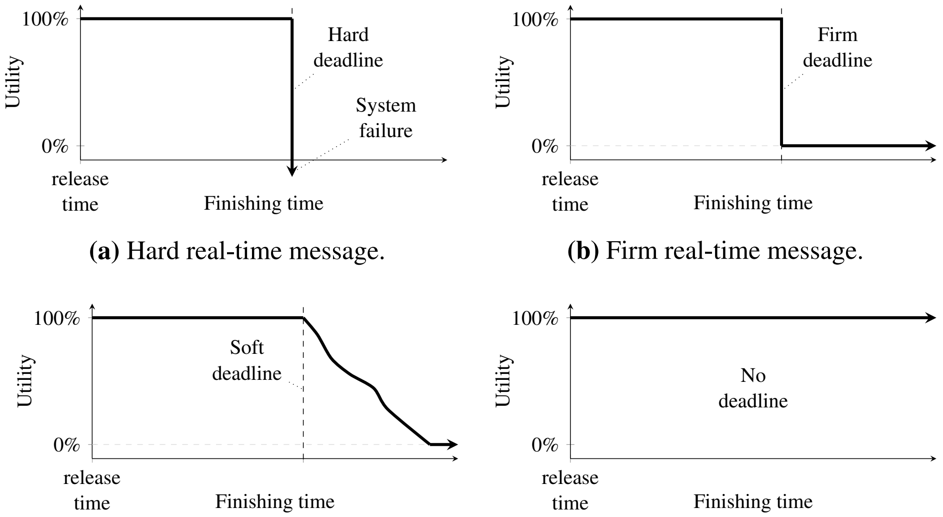
Since we distinguish firm from soft real-time messages according to their usefulness, but "usefulness" is a bit vague, we can make the classification more precise by using **time/utility functions** (originally called value functions[4]): functions that

[1] Guillermo Rodrıguez-Navas. "Design and Formal Verification of a Fault-Tolerant Clock Synchronization Subsystem for the Controller Area Network". PhD thesis. Universitat de les Illes Balears, 2010, p. 9.

[2] Ibid., Ch. 2.

[3] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. 3rd. Pearson Education, 2001, p. 435. Other authors understand hard deadlines as those that, if missed, cause severe consequences. See also the footnote on page 3.

[4] E. Douglas Jensen, C. Douglas Locke, and Hideyuki Tokuda. "A Time-Driven Scheduling Model for Real-Time Operating Systems." In: *Proceedings of the IEEE Real-Time Systems Symposium (RTSS*



(a) Hard real-time message.

(b) Firm real-time message.

(c) Soft real-time message.

(d) Non-real-time message.

Figure 5.2: Utility of the different types of real-time messages as a function of the finishing time and deadline.

quantify the utility of a message as a function of its finishing time. Figure 5.2 illustrates this. Regardless of the type of message, if it finishes anywhere between the release time and the deadline, its utility is 100%. Now, what happens after the deadline varies according to the type of message. If it is a hard real-time message, as shown in subfigure (a), then missing the deadline may cause a system failure, which may correspond to a negative utility, meaning that the system is now causing harm instead of good (in particular, if the system is a safety-critical one, the negative utility might exceed by orders of magnitude any positive utility a timely message could ever have). If the message is firm real-time (b), then its utility becomes zero after the deadline, but without necessarily causing a system failure. If it is soft real-time (c), then its utility decreases, according to some function, but without ever becoming negative. If the message is non-real-time (d), then its utility is a constant function, always being 100% regardless of the message's finishing time.

With some generality, we can say that hard deadlines should never be missed, firm deadlines may be missed occasionally, and soft deadlines may be missed frequently—although what is an occasional and a frequent miss cannot be generalized and depends on the particular system and its application.

Finally, we can also classify systems themselves as hard, firm, soft, or non-real-

1985). Vol. 85. 1985, pp. 112–122.

time. Hence, and in accordance with the definition of a hard real-time system we saw at the very beginning (Chapter 1), a **hard real-time system** is one that has to meet at least one hard deadline. A **firm real-time system** is one that has at least one firm deadline, but no hard deadlines. A **soft real-time system** is one that has at least one soft deadline, but no hard nor firm deadlines. And a **non-real-time system** is one that does not have any deadlines.

5.2 Periodic, Aperiodic, and Sporadic Messages

In the context of real-time communication it is customary to use the term "message" to refer not only to an individual message, but to a whole sequence—or **stream**[5]—of related messages, e.g., messages that carry the value of some particular sensor. For instance, if at time t_i a message m_i is released that carries the value of a sensor S and at time t_{i+1} a message m_{i+1} is released that carries the next value of that same sensor S, then we may use the term "message" to refer to the sequence of messages $m = \langle m_1, m_2, \ldots, m_i, m_{i+1}, \ldots \rangle$. We may then say that the *message* m is released at times $t_1, t_2, \ldots, t_i, t_{i+1}, \ldots$, meaning that m_1 is released at time t_1, that m_2 is released at time t_2, and so forth.

Using this convention we classify messages (i.e., sequences of messages) not only according to their criticality (hard, firm, soft, or non-real-time), but also according to their release pattern. This leads to the classification of messages into periodic, aperiodic, and sporadic.[6]

Periodic messages are messages that have a periodic release time, meaning that the time between successive release times is constant: if m_i is released at time t_i, then m_{i+1} is released at time $t_{i+1} = t_i + \tau$, where τ is a constant, called the **period** of message m. Hence, if we know the first release time of a periodic message, we know all following release times as well. For this reason periodic messages are usually parameterized by their period τ and their first release time t_1, which then is called the message's **offset** or phase.

Aperiodic messages are messages whose release time is not periodic. They may be released at any time and we generally do not know their release times in advance. In other words, if m_i is released at time t_i and m_{i+1} is released at time t_{i+1}, then the length of the time interval $[t_i, t_{i+1}]$ is a non-negative random variable.

Finally, **sporadic messages** are messages with a **minimum interarrival time**:

[5] Stream is the term used in FTT.

[6] Roman Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms*. 1st ed. Vol. 22. Real-Time Systems Series. Springer US, 2005, page 120.

they can be released at any time, except that there exists a minimum time interval between successive releases. In other words, if m_i is released at time t_i and m_{i+1} is released at time t_{i+1}, then the length of the time interval $[t_i, t_{i+1}]$ is a random variable that takes values in the range $[d, \infty)$, where $d > 0$ is the minimum interarrival time of m. One obvious consequence of having a minimum interarrival time is that the maximum rate with which sporadic messages can be released is restricted: if the minimum interarrival time of m is d, then m cannot be released with a frequency higher than $1/d$.

(Some authors consider sporadic messages to be a particular type of aperiodic messages, i.e., a subset of aperiodic messages. I prefer to keep them disjoint as otherwise the word aperiodic is ambiguous: it may either mean that the message has no minimum interarrival time or that it has a minimum interarrival time. So when I say "aperiodic message", I mean a message that does not have a period nor a known minimum interarrival time.)

5.3 Real-Time Traffic Scheduling

Scheduling refers to the assignment of work over time to resources that complete that work, usually with the goal of minimizing or maximizing some metric, such as response time, throughput, makespan, idle time, or deadline misses. The classic example in computing is process scheduling, which is done by an operating system and consists in determining which process (instance of a program) should be assigned next to the central processing unit (CPU) for execution. But the scheduled work does not have to be a set of computer processes. For instance, the work could just as well be the printing and binding of books in a print shop,[7] telephone calls to be answered by call-center employees, or messages to be transmitted through the network of a distributed system. The latter is what is of interest to us and is called traffic scheduling. Specifically, if the traffic consists of messages that must be exchanged before deadlines, then it is called real-time traffic scheduling. **Real-time traffic scheduling** is therefore the action of determining which nodes should transmit what messages at what time to guarantee that all messages will reach their destination before their deadlines.

[7] Bookbinding was the problem that gave rise to scheduling theory as its own discipline: it motivated Johnson to write his 1954 seminal paper, "Optimal Two and Three Stage Production Schedules with Setup Times Included", which first showed that scheduling problems can be solved algorithmically and that for some such problems optimal schedules exist. See R. A. Dudek, S. S. Panwalkar, and M. L. Smith. "The Lessons of Flowshop Scheduling Research". In: *Operations Research* 40.1 (1992), pp. 7–13; Chris N. Potts and Vitaly A. Strusevich. "Fifty Years of Scheduling: a Survey of Milestones". In: *Journal of the Operational Research Society* 60.1 (2009).

A **traffic scheduler** is the entity doing the scheduling of messages (it could be a person, but will typically be a computer process). To decide what message should be transmitted by whom at what time, the scheduler needs to have some criteria to prioritize the transmission of the different messages. The main temporal ones are the release time, transmission time, and relative or absolute deadline of each message to be scheduled. Besides these, however, a scheduler may also take into account further criteria. For instance, it may take into account the topology of the network; whether multiple messages can be transmitted simultaneously; the time it takes a message to traverse the network; whether messages can be preempted, i.e., aborted to make room for a higher priority message; whether messages are unicast, broadcast, or multicast; the probability with which messages may be lost due to network errors; or user-assigned priorities that allow us to indicate that some messages are intrinsically more important than others.

Whatever the particular criteria that prioritizes transmissions, a (deadline-based[8]) real-time traffic scheduler relies on an admission control. The **admission control** is a process whose job is to determine if a given set of messages can be exchanged on the network without any messages reaching their destination after their deadline. If this is possible, we say that the message set is **schedulable** or **feasible**. Otherwise, the message set is said to be **unschedulable** or **infeasible**. To determine if a set of messages is schedulable or not, the admission control executes a **schedulability test**, also known as **feasibility test**, which is an algorithm that returns true if the messages are guaranteed to be schedulable and returns false otherwise.

If the messages are schedulable, then the job of the real-time traffic scheduler is to come up with a **feasible schedule**: an assignment of transmission start times (or transmission request times[9]) to the messages that ensures that each finishing time is less than the corresponding absolute deadline.

5.3.1 Classification of Scheduling Algorithms

Scheduling algorithms can be classified in various ways, among them static versus dynamic, and offline versus online.[10]

[8] There are also schedulers based on time/utility functions.

[9] In some networks with a shared communication channel, such as Controller Area Network, the transmission start time is decided by the network's underlying arbitration algorithm. A feasible schedule might then tell the nodes when to request transmissions, leaving the exact transmission start time up to the network. Of course, the scheduler should still ensure that all messages reach their destinations on time, but it may not need to predict when exactly each message will start its transmission.

[10] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. Vol. 24. Springer Science & Business Media, 2011, pp. 35-36.

In **static scheduling** the scheduler decides the schedule based on fixed parameters, that is, parameters that do not change at runtime.[11] For instance, for periodic messages the offset and period are typically fixed.

In **dynamic scheduling**, on the other hand, the scheduler decides the schedule taking into account dynamic parameters as well, that is, parameters that may change at runtime or that cannot be known beforehand.[12] The release time of aperiodic and sporadic messages are examples of such parameters, as are relative deadlines of such messages.

In **offline scheduling**, or pre-runtime scheduling, the schedule with which messages are to be transmitted is generated for all messages in advance, before the very first release time.[13] Offline scheduling necessarily requires the scheduling to be static. Nevertheless, one could have several static schedules, allowing one to tell the scheduler to use one or another at runtime, thereby allowing some limited operational flexibility.

In **online scheduling** , or runtime scheduling, in contrast, the scheduler decides the schedule at runtime, while messages are released.[14] Online scheduling can be dynamic or static, depending on whether messages have dynamic parameters or not.

For flexible and hard real-time distributed embedded systems the scheduling should be dynamic and online. In FTT, as we will see (Chapter 7), it is.

Let us now change the subject to another aspect of real-time communication that is important to understand FTT, namely, event- and time-triggered communication, two classic communication paradigms that are both used in FTT.

5.4 The Classics: Event- and Time-Triggered Communication

FTT is a **communication paradigm**: a way of coordinating message transmissions in a network. Before we examine how it works (Chapter 7), we should take a look at the two classic communication paradigms on which it is based: event-triggered communication and time-triggered communication.[15]

In **event-triggered communication** the communication activities are triggered

[11] Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, p. 36.

[12] Ibid., p. 36.

[13] Ibid., p. 36.

[14] Ibid., p. 36.

[15] Roman Obermaisser. *Time-Triggered Communication*. CRC Press, 2011.

by the occurrence of events in the components of nodes or in systems they use.[16] For instance, a node may request the transmission of a message whenever a temperature sensor it uses reads a value above a certain threshold or when its processor has finished executing a certain task. In **time-triggered communication**, in contrast, communication activities are initiated according to a schedule dictated by a **global clock**, that is, a clock shared by all nodes.[17] The difference is like that between calling our colleagues and friends for lunch at the canteen when our stomachs growl versus calling them when the clock has struck lunchtime—one is event triggered, the other is time triggered.

Both event-triggered and time-triggered communication have advantages and disadvantages.

Event-triggered communication allows more flexibility and can be, on average, more efficient: if a node suddenly has to deal with unforeseen events, it simply goes ahead and transmits corresponding messages, and if no event has occurred, no message is transmitted unnecessarily. Event-triggered communication can thus respond much more rapidly to asynchronous events: if an alarm needs to be dispatched, this can be done immediately—no need to wait for the clock to strike the next message transmission time. The superior flexibility of event-triggered communication, however, also makes it less predictable. The schedule with which messages are transmitted is not minutely planned. Instead, the schedule unfolds dynamically at runtime as events occur in different nodes. This can lead to worst-case scenarios where messages are triggered in all nodes simultaneously—and under such scenarios the system can only meet all deadlines if it has significant overcapacity (e.g., network bandwidth) with respect to average-case scenarios.

In time-triggered communication, on the other hand, the schedule is preplanned. This makes it less flexible, but easier for nodes to coordinate their actions and to meet deadlines: the schedule has typically been subjected to a schedulability analysis to guarantee that messages will not violate deadlines. Another difference is that time-triggered communication, on average, wastes more bandwidth than event-triggered communication: in the time-triggered approach, if a message transmission has been scheduled, but the corresponding node has no need to transmit a message, the allocated bandwidth is lost and not recovered. But on the bright side, when all messages need to be transmitted, the time-triggered approach is more resource efficient: we can tell the nodes to transmit their messages in an orderly fashion that still guarantees that all deadlines will be met, thus avoiding an onrush of simultaneously transmitted messages. Also, since it is more predictable, it tends to allow a distributed embedded system to control physical systems while keeping

[16] Ibid., p. 1.

[17] Ibid., p. 1.

the control jitter low.[18] (Control jitter is the difference between the maximum and minimum delay between successive control actions. If it is low, the control is smooth; if it is high, the control is jerky.) Finally, by being more predictable, time-triggered communication also makes it easier to detect errors: if a message is scheduled, but not transmitted, this immediately indicates an error.

Event- and time-triggered communication also has a connection to the different types of real-time messages we saw earlier (Section 5.2). In event-triggered communication messages are typically aperiodic or sporadic; whereas in time-triggered communication they are typically periodic. Moreover, event-triggered communication generally implies dynamic (and thus online) scheduling; while time-triggered communication typically implies static scheduling (which may occur online or offline). Although this is not always the case. In FTT dynamic planning-based scheduling is used to take into account periodic messages whose parameters may change if they pass a schedulability test.

5.5 Discussion

We have completed a basic overview of real-time communication. We discussed the main parameters of real-time messages: release time, transmission time, start time, finishing time, absolute deadline, and relative deadline. We classified deadlines into hard, firm, and soft depending on whether missing them leads to a system failure, makes a message useless, or implies that the message still has some value. Next we classified messages into periodic, aperiodic, and sporadic depending on whether their release time is periodic, is not periodic, or there is a minimum interarrival time between releases. We then moved on to discuss real-time traffic scheduling, highlighting what it means for a message set to be schedulable and distinguishing between static versus dynamic scheduling and offline versus online scheduling. Finally, we reviewed the two classic communication paradigms used in real-time communication: event- and time-triggered communication.

We will now move on to discuss Ethernet in more detail. After all, it is the underlying technology on which we will base our fault-tolerant communication subsystem and we should therefore examine it in more detail.

[18] Amos Albert. “Comparison of Event Triggered and Time Triggered Concepts with Regard to Distributed Control Systems”. In: *Embedded World* 2004 (2004), pp. 235–252.

Chapter 6

Ethernet

Ethernet always wins.

Andy Bechtolsheim

Since we want to build a fault-tolerant communication subsystem based on Ethernet, it will be helpful to review this protocol.

To establish some context, we will begin with a quick overview of the original Ethernet (Section 6.1). Afterwards, we will review why the best-effort delivery provided by traditional Ethernet protocols is generally not good enough for distributed embedded systems (Section 6.2). Then we will discuss why even guaranteed real-time delivery may not be good enough (Section 6.3). Finally, to prepare ourselves for the analysis of the different FTT versions (Chapter 7), some of which use shared Ethernet and some of which use switched Ethernet, we will go over the main differences between shared and switched Ethernet (Section 6.4).

6.1 A Little Bit of History

When we meet someone for the first time we ask “where are you from?” and “what do you do?”. We ask these questions because their answers give us hints about a person’s life, allowing us to quickly infer key characteristics about them. In a similar spirit, let us answer where Ethernet is from (Section 6.1.1) and where it is employed (Section 6.1.2) to reveal some of its key characteristics.

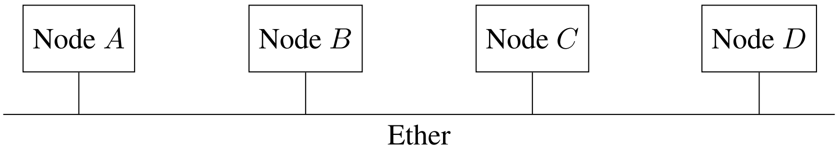


Figure 6.1: Topology of the first Ethernet. Nodes were tapped into a single coaxial cable, called the Ether.

6.1.1 The Origin of Ethernet

The term “Ethernet” is nowadays a broad term that covers many network technologies, protocols, and standards. But originally it referred to a single protocol, invented in 1973 by Robert Metcalfe and David R. Boggs at the legendary Xerox Palo Alto Research Center (PARC).[1] The aim of this protocol was to interconnect personal computers by means of a single shared cable, called the Ether, into which the computers were tapped in as shown in Figure 6.1.

The original Ethernet was based on ALOHAnet, a wireless computer network developed at the University of Hawaii in the late 1960s.[2] ALOHAnet had introduced a then novel communication scheme that worked as follows.[3] When a computer wants to transmit a message through a shared medium (the air in case of ALOHAnet), it just goes ahead and transmits. Then it waits for an acknowledgment message. If it receives one, it knows that its transmission succeeded. Otherwise, it assumes that a **message collision** occurred, i.e., that its transmission was corrupted by a coinciding transmission from another computer. Upon an assumed collision, each transmitting computer waits for a random amount of time—called a **random backoff time**—before retransmitting. If the computers choose different backoff times, and no new transmissions interfere, the retransmissions will succeed. Otherwise, collisions occur again. These subsequent collisions are resolved in the same manner: through retransmissions after random backoff times, whose maximum increases after each collision. Metcalfe adapted this idea to allow several computers to transmit through another shared medium—a coaxial cable. Thus, Ethernet was born.

One of the innovations that Metcalfe introduced was the idea of **carrier sensing** before transmitting. Instead of just going ahead and transmitting a message, as

[1] Charles Severance. “Bob Metcalfe: Ethernet at Forty”. In: *Computer* 46.5 (2013), pp. 6–9.

[2] Norman Abramson. “Development of the ALOHANET”. in: *IEEE Transactions on Information Theory* 31.2 (1985), pp. 119–123.

[3] Norman Abramson. “The ALOHA System: Another Alternative for Computer Communications”. In: *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*. ACM. 1970, pp. 281–285.

in ALOHAnet, in Ethernet a node would first check if another node is already transmitting. Only if not, would it go ahead with its own transmission. This reduced the probability of collisions, making communication more efficient.

Another innovation was **collision detection**, which lets nodes abort garbled transmissions. In ALOHAnet a transmitting computer could not detect when another computer was transmitting simultaneously and causing collisions. Why? Because the transmitters were radio nodes that used the air as a shared medium (all nodes used the same frequencies) and, at the time, a radio antenna could not listen for incoming signals while transmitting over the same frequencies.[4] Thus, in ALOHAnet collisions were not detected, but only *assumed* if after a long timeout an acknowledgment message had not been received. Moreover, since an acknowledgment can only be received after a full transmission, garbled transmissions were not aborted. Ethernet, in contrast, used a coaxial cable as the shared medium. This medium allowed the detection of incoming signals while transmitting, and thus the *detection* of collisions. A node could therefore detect and abort a garbled transmission, freeing up bandwidth by not continuing with the transmission of useless bits.

Besides these two innovations, Metcalfe and his colleagues added two more features:[5] they added a source and destination address to each message (ALOHAnet had only one address per message) and they added a cyclic redundancy checksum (CRC) to each message.

(Further details about the historical Ethernet can be found in Metcalfe and Boggs's 1976 paper.[6])

Soon after the invention of Ethernet at Xerox PARC, other companies became interested in using the new protocol. To ensure interoperability, Metcalfe decided that the best way forward was to develop a standard.

The first Ethernet standard was jointly published by Digital Equipment Corporation (DEC), Intel, and Xerox in 1980.[7] This became known as the DIX Ethernet standard, where DIX stands for DEC, Intel, and Xerox. The same year the Institute

[4] In fact, it is only recently that wireless transceivers that support full-duplex communication (simultaneous sending and receiving) over the same frequency have become possible using a technique for self-interference cancellation called antenna cancellation (Jung Il Choi et al. "Achieving Single Channel, Full Duplex Wireless Communication". In: *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*. MobiCom '10. Chicago, Illinois, USA: ACM, 2010, pp. 1–12). Before that, the hundreds of thousands of times stronger outgoing signal of an antenna would simply drown out any incoming signal.

[5] Severance, "Bob Metcalfe: Ethernet at Forty".

[6] Robert M. Metcalfe and David R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks". In: *Commun. ACM* 19.7 (July 1976), pp. 395–404.

[7] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation. *The Ethernet, a Local Area Network, Data Link Layer and Physical Layer Specification*. Sept. 1980.

of Electrical and Electronics Engineers (IEEE) formed the IEEE 802 LAN/MAN standard committee to begin working on a set of standards for Local Area Networks (LANs) and Metropolitan Area Networks (MANs). Among this set of standards, the committee published the 802.3 standard of Ethernet, first as a draft, in 1983, and then as an approved standard, in 1985. The standard put Metcalfe's two main innovations—carrier sense and collision detection—right in the title: *IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*.[8] This standard, or more precisely, the revised version of this standard,[9] is what most people today probably mean by the term "Ethernet". I say "probably" because, as we will see in a moment, there are many modified versions of Ethernet and, depending on the context, a person might be referring to any one of those.

6.1.2 Ethernet's Unstoppable Career

Since its invention, Ethernet has been unstoppable, acquiring new jobs in an increasing number of domains.

Ethernet's first job was to interconnect computers in local area networks (LANs). For a time, Ethernet was not the only contender, but at the end of the 1980s through the mid-1990s it demolished its competition in the so-called LAN wars. It accomplished this by continuously reinventing itself through further standardization efforts in the years following the first DIX and IEEE 802.3 standards. Some of the milestones involved repeatedly increasing Ethernet's transmission speed and range, going from a physical line topology to a physical tree and mesh topology by first introducing hubs and then switches (see Section 6.4 for more on hubs and switches), changing the underlying medium from coaxial to twisted-pair and fiber optic cables, adding wired half-duplex and then full-duplex communication (also introduced in Section 6.4), and introducing wireless versions of Ethernet.[10]

Not content with being the sole survivor of the LAN wars—and thus the *de facto* choice for wiring any office, household, or campus—Ethernet soon looked for additional jobs. The following are a few highlights.

Around the year 2000, Ethernet began to be considered for the first mile—the

[8] Institute of Electrical and Electronics Engineers (IEEE). *Standards for Local Area Networks: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. 802.3-1985 (ANSI/IEEE). 1985.

[9] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Ethernet—Section three*. 802.3-2015 (IEEE). 2015.

[10] Charles E. Spurgeon. *Ethernet: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2000.

portion of the Internet that connects an Internet service provider's central office to a business, residential home, or other end-user location.[11] This interest resulted in the formation of the Ethernet in the First Mile Alliance (EFMA) in 2001, an alliance of several companies (including Cisco, Ericsson, Intel, Texas Instruments, and Infineon) to promote the use of Ethernet in the first mile.[12] As a result, in the year 2004, the IEEE 802.3 standard was amended.[13] The amendment, nicknamed Ethernet in the First Mile (EFM), added additional specifications for Ethernet to be used on "subscriber access networks", another word for "first mile".

Ethernet did not stop at the first mile. During the 2000s Internet service providers were also increasingly interested in using Ethernet for their wide area networks (WANs).[14] So another alliance of companies was formed, also in 2001, called the Metro Ethernet Forum, whose stated mission was to accelerate the worldwide adoption of Carrier Ethernet, i.e., Ethernet for WANs.[15] And again, the IEEE standards had to be amended, resulting in, among other standards, the IEEE Provider Bridges and the Provider Backbone Bridges standards.[16]

As if having displaced other technologies in local area networks and within Internet service providers was not enough of an achievement yet, Ethernet looked for still more job opportunities and in the 2000s also became the main networking technology in data centers. Once again, IEEE standards had to be amended. Thus, the IEEE Data Center Bridging Task Group of the IEEE 802.1 Working Group began developing several amendments in the mid-2000s, among them the IEEE 802.1Qau Congestion Notification standard,[17] the IEEE 802.1Qbb Priority-based

[11] Howard Frazier and Gerry Pesavento. "Ethernet Takes on the First Mile". In: *IT Professional* 3.4 (2001), pp. 17–22.

[12] Michael Beck. *Ethernet in the First Mile: The IEEE 802.3ah EFM Standard (Communications Engineering Series)*. McGraw-Hill Education, 2005.

[13] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Information technology–Local and Metropolitan Area Networks — Part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment: Media Access Control Parameters, Physical Layers, and Management Parameters for Subscriber Access Networks*. 802.3ah-2004 (IEEE). 2004.

[14] Abdul Kasim et al. *Delivering Carrier Ethernet: Extending Ethernet Beyond the LAN*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2008.

[15] Metro Ethernet Forum. *Website of the Metro Ethernet Forum*. 2014. URL: https://www.mef.net (visited on 2016-05-15).

[16] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 4: Provider Bridges*. 802.1ah-2005 (IEEE). 2005, (the first draft was published in 2002); Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 7: Provider Backbone Bridges*. 802.1ah-2008 (IEEE). 2008, (the first draft was published in 2005).

[17] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 10: Congestion Notification*.

Flow Control standard,[18] and the IEEE 802.1Qaz Enhanced Transmission Selection standard.[19]

Finally, and most interesting to us, Ethernet is also competing with other technologies in the domain of distributed embedded systems. In this domain fieldbuses have traditionally been at work. But this is changing: Ethernet is already being used in cars[20] and aircrafts,[21] it is vying for supremacy in factories and industrial plants,[22] and it may play a big role in the Internet of Things[23] (and thus Industry 4.0). Yet, at least for now, fieldbuses are still retaining much of their stronghold.

One reason for Ethernet having failed to conquer the domain of distributed embedded systems as swiftly as others is that it has traditionally been a protocol that only provides **best-effort message delivery**. As the name suggests, best-effort delivery means that Ethernet tries its best to deliver messages to their intended recipients. This, however, also means that Ethernet—at least traditionally—could not guarantee that messages will be delivered in time, or at all. Many distributed embedded systems, however, do require such guarantees.

6.2 Best-Effort Is Not Always Good Enough

Most distributed embedded systems are real-time systems. They interact with a physical environment that progresses inexorably: liquids gush out of valves, gases build up within containers, chemical reactions unfold, gravity pulls relentlessly. It is therefore the nodes of the distributed embedded system—with their sensors and actuators—that need to keep up with the environment and certainly not the other way around. The environment will not wait for a slow embedded system.

802.1Qau-2010 (IEEE). 2010, (the first draft was published in 2006).

[18] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 17: Priority-based Flow Control*. 802.1Qbb-2011 (IEEE). 2011, (the first draft was published in 2008).

[19] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 18: Enhanced Transmission Selection*. 802.1Qaz-2011 (IEEE). 2011, (the first draft was published in 2008).

[20] Kirsten Matheus and Thomas Königseder. *Automotive Ethernet*. Cambridge University Press, 2014.

[21] Aeronautical Radio, Incorporated (ARINC). *Aircraft Data Network — Part 7, Avionics Full Duplex Switched Ethernet Network (AFDX)*. ARINC 664P7-1 (ARINC). 2009.

[22] Peter Danielis et al. "Survey on Real-Time Communication via Ethernet in Industrial Automation Environments". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014.

[23] Uday Mudoi. *Ethernet Evolves Again To Meet The Internet Of Things*. 2014. URL: http://electronicdesign.com/communications/ethernet-evolves-again-meet-internet-things (visited on 2017-01-05).

To keep up with the environment, the nodes of the distributed embedded system must exchange messages and they must do so before the message-exchange deadlines imposed by the environment pass. This means that for most distributed embedded systems best-effort delivery is not good enough. Yet, the original Ethernet protocol, and most subsequent Ethernet-based protocols, only provide best-effort delivery. Luckily, Ethernet can be improved to provide delivery that is better than best-effort.

If we assume that links, switches, or other network elements of an Ethernet network do not suffer permanent failures, and we only care about messages being delivered eventually, i.e., at some point in time, then we can put a transport layer protocol[24] such as the transmission control protocol (TCP) on top of Ethernet. This protocol will then ask the underlying Ethernet to resend any messages that have been lost or corrupted due to transient faults in the components of the communication subsystem. Thus, we obtain *guaranteed* delivery in the presence of transient faults.

If we now also assume that messages may not reach the destination because of permanent faults, such as link or switch crashes, then a transport layer protocol is no longer enough for guaranteed delivery. Ethernet will have to be able to provide alternative communication paths from a source to a destination. Fortunately, there are Ethernet-based solutions that do precisely that: when the current communication paths are disrupted due to permanent faults, they provide alternative paths. Examples are the Spanning Tree Protocol (STP),[25] the Rapid Spanning Tree Protocol (RSTP),[26] Transparent Interconnection of Lots of Links (TRILL),[27] Shortest Path Bridging (SPB),[28] and the Media Redundancy Protocol (MRP).[29] All of them allow an Ethernet network to recover upon the crash of links or intermediate nodes such as switches. They can therefore guarantee that transport layer messages will eventually reach their recipients even if communication paths are disrupted.

If we are concerned about timely delivery, and not just eventual delivery, then guaranteed delivery is not sufficient. We need guaranteed and *real-time* delivery. In

[24] International Organization for Standardization (ISO). *Information technology — Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO/IEC 7498-1:1994. Geneva, 1994.

[25] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local Area Network MAC (Media Access Control) Bridges*. 802.1D-1998 (IEEE). 1998.

[26] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local Area Network MAC (Media Access Control) Bridges*. 802.1D-2004 (IEEE). 2004.

[27] R. Perlman et al. *Routing Bridges (RBridges): Base Protocol Specification*. 6325 (RFC). 2011.

[28] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks—Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 20: Shortest Path Bridging*. 802.1aq-2012 (IEEE). 2012.

[29] International Electrotechnical Commission (IEC). *Industrial communication networks—high availability automation networks—part 2: media redundancy protocol (MRP)*. IEC 62439-2 (IEC).

that case the retransmissions of typical transport layer protocols are problematic. Typical transport layer protocols are only concerned with eventual delivery and do not take into account any deadlines—they simply resend lost messages over and over, regardless of the latency this introduces. And as to tolerating permanent faults, the spanning tree protocols, and the other ones we just saw, also pose problems if we want real-time delivery in addition to guaranteed delivery. Why? Because these protocols have non-zero failure recovery times and thus require some time to reestablish communication after link or switch failures. Thus, if the deadlines are short, the network recovery time may be too large. For very short deadlines we even need Ethernet-based solutions that provide seamless fault tolerance. Luckily, there are such solutions. Some examples include Ethernet PowerLink[30] when used with a ring topology or with two physically independent networks; TTEthernet,[31] which can provide up to three independent communication channels; High-availability Seamless Redundancy (HSR),[32] which daisy chains Ethernet nodes into a ring topology in which frames are duplicated and then transmitted in opposite directions along the ring; the Parallel Redundancy Protocol (PRP),[33] which uses two independent Ethernet networks simultaneously; Avionics Full-Duplex Ethernet (AFDX),[34] which also relies on two independent networks for redundancy; and the upcoming Time-Sensitive Networking (TSN) set of standards,[35] which can provide seamless redundancy through multipathing, i.e., the reservation of multiple disjoint network paths from a source to a destination. With such protocols we can prevent messages from reaching their destination too late due to excessive network recovery times, even if deadlines are short.

Excessive network recovery times and deadline-agnostic transport layers, however, are not the only impediments for the timely delivery of messages in Ethernet.

[30] Ethernet POWERLINK Standardisation Group. *Ethernet POWERLINK Communication Profile Specification — Version 1.3.0.* EPSG Draft Standard 301 (EPSG). 2016.

[31] Society of Automotive Engineers (SAE). *Time-Triggered Ethernet.* SAE AS 6802 (SAE). 2011.

[32] International Electrotechnical Commission (IEC). *Industrial communication networks—high availability automation networks—part 3: parallel redundancy protocol (PRP) and high-availability seamless redundancy (HSR).* IEC 62439-3 (IEC).

[33] Ibid.

[34] ARINC, *AFDX.*

[35] Institute of Electrical and Electronics Engineers (IEEE). *Timing and Synchronization: Enhancements and Performance Improvements — Draft 0.7.* 802.1ASbt (IEEE). 2014; Institute of Electrical and Electronics Engineers (IEEE). *Enhancements for Scheduled Traffic — Draft 2.1.* 802.1Qbv (IEEE). 2014; Institute of Electrical and Electronics Engineers (IEEE). *Frame Preemption — Draft 1.1.* 802.1Qbu (IEEE). 2014; Institute of Electrical and Electronics Engineers (IEEE). *Path Control and Reservation — Draft 1.1.* 802.1Qca (IEEE). 2014; Institute of Electrical and Electronics Engineers (IEEE). *Frame Replication and Elimination for Reliability — Draft 0.5.* 802.1CB (IEEE). 2014; Institute of Electrical and Electronics Engineers (IEEE). *Stream Reservation Protocol (SRP) Enhancements and Performance Improvements — Draft 0.2.* 802.1Qcc (IEEE). 2014.

Another impediment is the nondeterminism of most Ethernet-based protocols. This is particularly true for protocols based on the original CSMA/CD randomized retransmission algorithm: their deliberate random delays are not particularly conducive to meeting deadlines. After all, random delays make it hard to bound the response time of messages and thus provide real-time guarantees. However, even full-duplex switched Ethernet, which does not use CSMA/CD (see Section 6.4), may delay messages unpredictably—not because of collisions and random backoff times, but because of queuing at switches and end nodes. Indeed, switched Ethernet (without additional enhancements such as the congestion notification standard that is part of data center bridging) is even said to be a **lossy protocol**, meaning that if the rate with which frames arrive at a switch is greater than the rate with which they leave the switch, then frames may be lost due to queue overflows. Even so, there are some Ethernet-based protocols that have overcome these problems and are suitable for real-time applications. Some older such protocols are described in a book chapter by Almeida and Pedreiras.[36] There are also some more recent protocols.

One example is the AFDX protocol we mentioned before. It is based on switched Ethernet and rate-constrained transmission. The latter means that a transmitting node cannot transmit as frequently as it wants, but has to wait a certain amount of time, called the bandwidth allocation gap, between successive transmissions to a given set of nodes. By thus constraining the rate with which messages are transmitted, AFDX can prevent excessive queuing in switches and provide hard real-time guarantees.

Another example are the set of TSN standards, which we also just mentioned. To provide hard real-time guarantees, TSN builds on the set of Audio Video Bridging (AVB) standards,[37] which were designed for soft real-time applications. Unfortunately, however, many details on the operation of TSN are not yet publicly available because, as I am writing this, it is still under development.

[36] Luís Almeida and Paulo Pedreiras. "Approaches to Enforce Real-Time Behavior in Ethernet". In: *The Industrial Communication Technology Handbook*. Ed. by Richard Zurawski. CRC Press, 2005. Chap. 20.

[37] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*. 802.1AS-2011 (IEEE). 2011; Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area networks–Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP)*. 802.1Qat-2010 (IEEE). 2010; Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*. 802.1Qav-2009 (IEEE). 2009; Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area networks–Audio Video Bridging (AVB) Systems*. 802.1BA-2011 (IEEE). 2011.

TTEthernet[38] also has support for hard real-time communication. It achieves this by means of time-triggered messages that are transmitted according to a schedule calculated offline (Section 5.3.1) and special Ethernet switches that prioritize the transmission of these messages over event-triggered messages.

Finally, another notable example—the most notable for us—are the Ethernet versions of FTT. They can also provide hard real-time guarantees. We will cover them in our survey (Chapter 7).

By using Ethernet-based protocols that provide adequate fault tolerance and that have been designed with real-time applications in mind, such as AFDX, TTEthernet, and foreseeably TSN, we can obtain **guaranteed real-time message delivery**: messages are delivered in time even if the network suffers partial failures. Is this good enough? For many applications it surely is, but not for the ones we envisioned (Chapter 1).

6.3 Guaranteed Real-Time Delivery May Not Be Good Enough Either

We know by now that for most distributed embedded systems best-effort delivery is not good enough. They interact with physical environments, which progress on their own terms and will not wait for the nodes of any distributed embedded system. In particular, such environments will not wait for the nodes to finish exchanging messages and they will not condone the loss of service-critical messages. Distributed embedded systems thus generally require guaranteed real-time delivery, and not just best-effort delivery. For some distributed embedded systems, however, even guaranteed real-time message delivery is not good enough. We saw some of these systems at the very beginning (Chapter 1). I am referring to the self-driving cars, mobile rescue robots, and smart cities of the future. For the communication subsystems of these systems, guaranteed and real-time message delivery is a prerequisite, but not the only prerequisite. If we think back, we will remember that to be adaptive these systems will have to be flexible and that this implies that their underlying communication subsystems will need to be flexible as well. At the very least, their communication subsystems will need to be sufficiently flexible to accommodate changing real-time requirements at runtime. Yet, it is precisely here that real-time Ethernet protocols currently fail us.

AFDX can provide seamless fault tolerance and meet hard deadlines, but it can only do so when the real-time requirements do not change. This is so because the

[38] SAE, *Time-Triggered Ethernet*.

bandwidth allocation gap, the parameter that limits the rate with which messages are transmitted, is calculated (e.g., using network calculus[39]) before an AFDX network is put into operation. Once the network is operating, the bandwidth allocation gap is fixed. And since the bandwidth allocation gap has been calculated assuming certain real-time parameters, these parameters will also have to remain fixed.

TTEthernet, similar to AFDX, provides seamless fault tolerance and can meet hard deadlines. However, just like AFDX, it also lacks flexibility: the schedule for the time-triggered messages is calculated offline, before the network is put into operation, and cannot be changed later on without halting and reconfiguring the network.

There is also TSN. It, however, is still under development and we cannot yet say whether it would be suitable for distributed embedded systems that need to be hard real-time, highly reliable, and flexible.

Finally, the FTT versions of Ethernet can meet hard real-time deadlines and are flexible, but they are not fault tolerant. Thus, they cannot guarantee the delivery of messages in the presence of faults.

Thus, neither of these Ethernet-based protocols, nor any other that I am aware of, can currently be used for distributed embedded systems that need to be flexible in their real-time behavior, while simultaneously being highly reliable and capable of meeting hard deadlines. We already discussed that at the very beginning (Chapter 1) and it is precisely why we embarked on this research journey.

Our next stop will be the land of FTT, whose landscape we will survey. But before we depart, we should review the key differences between shared and switched Ethernet.

6.4 Shared versus Switched Ethernet

In our upcoming survey of FTT (Chapter 7) we will see that some versions of FTT build on shared Ethernet and others on switched Ethernet. To design our own fault tolerant version of FTT (Chapter 8) we will have to decide which option is more appropriate for us as a starting point. We must therefore have a clear understanding of how shared Ethernet differs from switched Ethernet.

[39] Network calculus is somewhat similar to queuing theory, but focuses on worst-case metrics (e.g., worst-case response times) instead of averages (e.g., average response times).

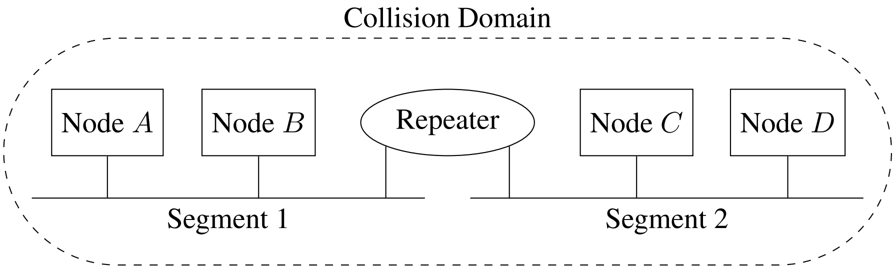


Figure 6.2: Joining Ethernet segments by means of a repeater yields a single collision domain.

6.4.1 Shared Ethernet

Shared Ethernet refers to versions of Ethernet where any two simultaneously transmitted messages collide. Does this mean that there are versions of Ethernet where messages can be transmitted simultaneously without collisions? Yes. Those are the switched Ethernet versions of Ethernet. Before we get to those, let us talk a bit more about shared Ethernet.

The original version of Ethernet relied on a single shared medium: one coaxial cable. In this original version, if we wanted to interconnect more computers, we could face tricky situations. What if we lacked a cable long enough to reach all the computers? Or what if we had started interconnecting one set of computers with one cable, and another set of computers with another, and now wanted to interconnect the two sets? In such cases we might want to join different cables to create a single Ethernet. This was precisely what Ethernet repeaters were invented for.

An **Ethernet repeater** is a device that joins two Ethernet cables and repeats on one cable any bits it sees on the other cable. Figure 6.2 shows an example. A cable together with any nodes tapped into it is called an **Ethernet segment**. The figure shows two such segments: one comprised of nodes A and B, and the other comprised of nodes C and D. With this setup, the nodes from one segment can exchange messages among themselves, while also being able to send and receive messages from the other segment. The repeater interconnects the segments at the physical layer of the OSI reference model.[40] Thus, if there is a collision on one segment, the repeater forwards the garbled bits to the other segment. As a result, the whole network comprised of the two segments acts like a single cable spanning all nodes. By connecting further repeaters in series, with Ethernet segments in between,

[40] ISO, *OSI – Basic Reference Model.*

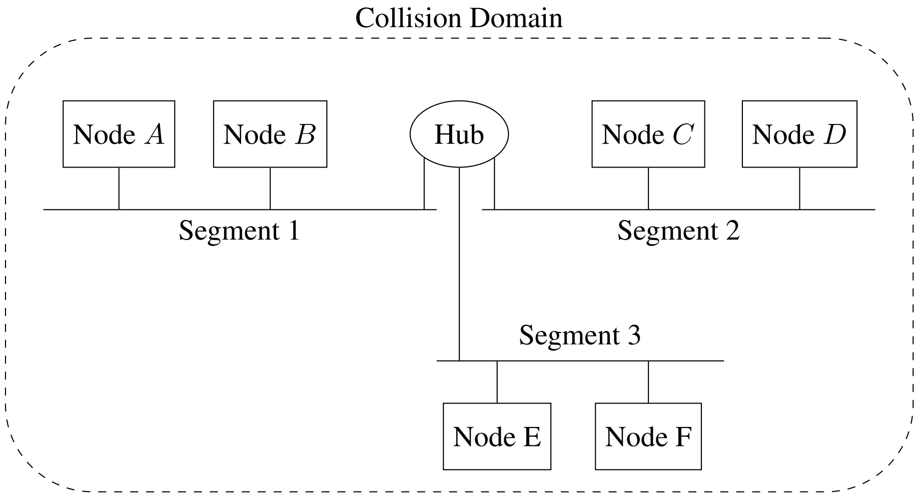


Figure 6.3: Joining Ethernet segments by means of a hub enables tree topologies, but the network still constitutes a single collision domain.

up to 1024 nodes can be interconnected.[41]

Regardless of the number of segments, if they are interconnected by repeaters, there is still only a single shared communication medium where collisions can occur. For this reason, Ethernet segments interconnected by repeaters are said to constitute a single **collision domain**. Referring once again to Figure 6.2, the two segments joined by the repeater constitute a single collision domain: messages transmitted simultaneously by nodes A and B not only collide with each other, but can also collide with messages transmitted by nodes C or D.

Using repeaters limited the topology of Ethernet networks to bus-based topologies. Bus topologies, however, were inconvenient for interconnecting computers in a building; especially since buildings were already wired using a star topology (star topologies were the topology of choice for the telephone system[42]). For this reason multiport repeaters, called repeater hubs, or simply **hubs**, became popular.

Hubs allow a network to branch out in different directions. An example is shown in Figure 6.3. It shows an Ethernet network comprised of three Ethernet segments interconnected by a hub. Since a hub, like a dual-port repeater, ensures that the physical-layer signals originating in one segment are repeated on the other segments, any set of Ethernet segments interconnected by hubs still constitutes a

[41] Spurgeon, *Ethernet: The Definitive Guide*, p. 60.

[42] Radia Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Pearson Education, 2000, Sec. 5.1.

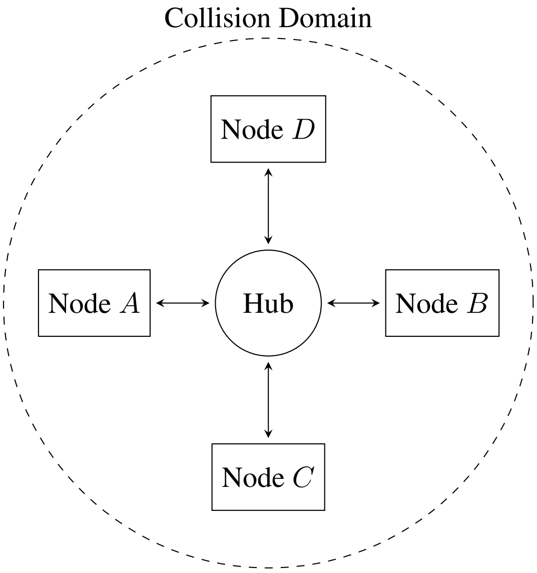


Figure 6.4: Using a hub and placing only one node in each segment yields a pure star topology.

single collision domain. Thus, the whole network shown in Figure 6.3 constitutes one collision domain. By putting only one node on each segment, a pure star topology is obtained. An example of this is shown in Figure 6.4.

With hubs it became much easier to interconnect the computers within a building. Ethernet segments could now be interconnected using multiple hubs to form more complex topologies and an Ethernet could now easily follow a corridor and spread out into different offices or cover the crisscrossing aisles of a cubicle farm. The only restriction was that the topology had to be a cascaded star or tree, i.e., a topology without loops. Otherwise, an Ethernet message could circle back and collide with itself (like a light cycle from the movie Tron driving in a circle and colliding with its own solid neon-light jetwall).

6.4.2 Switched Ethernet

Switched Ethernet refers to versions of Ethernet where there is no longer a single collision domain. Instead, there are either several collision domains, or none at all. To understand how this is achieved, and what the advantages are, another brief review of the history is in order.

Although shared Ethernet was widely adopted for local area networks, in the

1980s it was increasingly criticized for behaving poorly under high traffic loads. These criticisms were mainly in the form of theoretical studies.[43] As a response, David R. Boggs, the coinventor of Ethernet, together with others, retorted that in practice Ethernet behaved quite well for typical office applications.[44] Meanwhile, another blow against shared Ethernet was that it could only cover distances of up to 1.5 kilometers, while other network technologies such as IBM's token ring could cover much larger distances.[45] The reason for this was that in shared Ethernet the propagation delay between any pair of nodes had to be small enough for collisions to be detected. If two nodes were separated more than 1.5 kilometers, they might transmit a message at the same time and complete the full transmission before the nodes could detect a collision.

With its presumed poor performance and limited range, Ethernet's reputation was tarnished. Then, Mark F. Kempf, an engineer at Digital Equipment Corporation (DEC), invented the Ethernet bridge.[46]

An **Ethernet bridge** is a device that, like a repeater, can be used to join Ethernet segments. The key difference is that a bridge breaks up the single collision domain. Figure 6.5 illustrates this. We again have two Ethernet segments, one with nodes A and B, and another with nodes C and D. This time, however, they are interconnected by a bridge. Unlike a repeater, a bridge no longer blindly copies bits from one segment to the other. Instead, a bridge copies messages only if necessary. Specifically, it buffers messages, inspects them, and only copies them from one segment to the other if their destination address matches a node on the second segment or if the destination address is multicast or broadcast. Hence, if node A in

[43] Werner Bux. "Local-Area Subnetworks: a Performance Comparison". In: *IEEE Transactions on Communications* 29.10 (1981), pp. 1465–1473; Edward J. Coyle and Bede Liu. "Finite Population CSMA/CD Networks". In: *IEEE Transactions on Communications* 31.11 (1983), pp. 1247–1251; Edward J. Coyle and Bede Liu. "A Matrix Representation of CSMA/CD Networks". In: *IEEE Transactions on Communications* 33.1 (1985), pp. 53–64; Timothy A. Gonsalves and Fouad A. Tobagi. "On the Performance Effects of Station Locations and Access Protocol Parameters in Ethernet Networks". In: *IEEE Transactions on Communications* 36.4 (1988), pp. 441–449; Hideaki Takagi and Leonard Kleinrock. "Throughput analysis for persistent CSMA systems". In: *IEEE Transactions on Communications* 33.7 (1985), pp. 627–638; Shuji Tasaka. "Dynamic behavior of a CSMA-CD system with a finite population of buffered users". In: *IEEE Transactions on Communications* 34.6 (1986), pp. 576–586; Fouad A. Tobagi and V. Bruce Hunt. "Performance analysis of carrier sense multiple access with collision detection". In: *Computer Networks (1976)* 4.5 (1980), pp. 245–259.

[44] D. R. Boggs, J. C. Mogul, and C. A. Kent. "Measured Capacity of an Ethernet: Myths and Reality". In: *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM '88. Stanford, California, USA: ACM, 1988, pp. 222–234.

[45] George Varghese. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. The Morgan Kaufmann Series in Networking. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[46] Mark F. Kempf. *Bridge Circuit for Interconnecting Networks*. US Patent 4,597,078. June 1986.

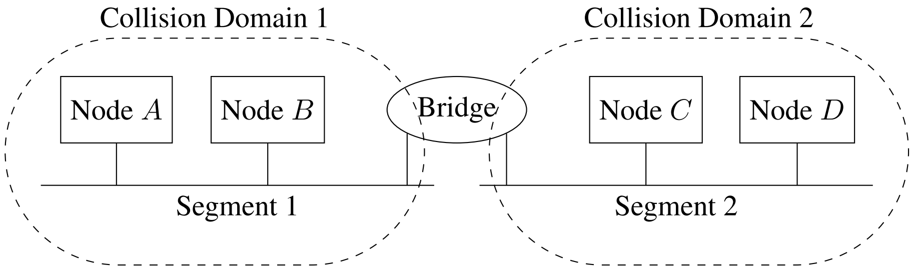


Figure 6.5: Joining Ethernet segments by means of a bridge divides the network into separate collision domains.

Figure 6.5 sends a message to node B, the bridge buffers the message, determines that it is not destined for segment 2, and simply drops (discards) it. Only if node A sends a message with a destination address matching one of the nodes on segment 2, does the bridge forward the message. Similarly, only messages originating in segment 2 that are destined to a node in segment 1 are forwarded by the bridge. All others are dropped. Moreover, if there is a collision on one segment, the bridge does not forward garbled messages to the other segment. As a result, messages can now only collide within a segment, and not between segments, resulting in two separate collision domains.

What is the advantage of a separate collision domain for each segment? Well, it solves the two problems for which shared Ethernet had been criticized.

First, it increases the available bandwidth. As an example, let us refer again to Figure 6.5. Assume a fraction p of all traffic is local to each segment. That is, of all the traffic originating in a given segment, a fraction p is not forwarded by the bridge to the other segment. In that case, the fraction p can be sent simultaneously on both segments. Hence, the overall bandwidth of the two Ethernet segments is increased to $(1 + p)$ times the bandwidth of a single segment.[47]

Second, it expands the range of an Ethernet network. With a bridge, the limitation of allowing at most 1.5 kilometers of distance between any pair of nodes applies to each segment separately.[48] Thus, the Ethernet of Figure 6.5 can span up to 3 kilometers, 1.5 kilometers for each segment. This is in stark contrast to the Ethernet of Figure 6.2, which despite having two segments, can only span 1.5 kilometers.

Besides these two advantages, there is another advantage that is more pertinent to us: having separate collision domains can increase the dependability of an

[47] Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*, p. 224.

[48] Ibid.

Ethernet. As an example, imagine that due to some fault node A of Figure 6.2 begins transmitting messages non-stop to node B, without any sense or logic, and without backing off upon collisions. If a repeater is used, as is the case in Figure 6.2, this disrupts communication on the whole Ethernet. If, by contrast, a bridge is used as in Figure 6.5, this only disrupts communication on segment 1. Nodes C and D can still communicate with each other.

Let us now focus our attention on discussing the topologies that switched Ethernet enables. That way, when we design our own Ethernet-based communication subsystem (Chapter 8)—which I can already anticipate will be based on switched Ethernet—we will be in a good position to decide on a network topology to use.

As we saw a moment ago (Section 6.4.1), shared Ethernet could not have network topologies with loops because otherwise a message could collide with itself. With bridges we no longer have the problem of self-colliding messages because they break up the network into separate collision domains. Yet, loops are still problematic. The reason for this, however, is different now. The problem with loops in switched Ethernet is that a message may circulate forever, going round and round without ever stopping.

To solve the problem of circulating messages, Radia Perlman, a routing architect who also worked at DEC, found a solution: she invented the spanning tree protocol.[49] In a nutshell, the spanning tree protocol prevents loops in an Ethernet network by having bridges block some of their ports in such a way that the physical topology of the network is pruned into a loop-free topology, i.e., a tree topology, that spans all of the network's segments. This provides a path between any pair of nodes, but without allowing loops in which messages could circulate.

To enable star-based topologies, Kempf considered in his patent[50] bridges with more than two ports. However, it was not Kempf, nor his employer DEC, who sold the first multiport Ethernet bridge, but a company called Kalpana.[51] Since this company called its product EtherSwitch, multiport bridges became known as **switches**. A switch, thus, is the same as a bridge, but it is commonly understood to be one with more than two ports.[52]

As we saw, switches (or bridges) can break up a collision domain into several

[49] Perlman, "An Algorithm for Distributed Computation of a Spanning tree in an Extended LAN", The paper is a curious one. In it, Perlman introduces her spanning tree algorithm in the form of a poem, which she calls an "algorhyme"—a rhyming algorithm.

[50] Kempf, *Bridge Circuit for Interconnecting Networks*.

[51] Robert J. Kohlhepp. "The 10 Most Important Products of the Decade — Number Five: Kalpana EtherSwitch". In: *Network Computing* (2000).

[52] The IEEE standards, however, continue to use the term bridge, whether the device has two ports or more.

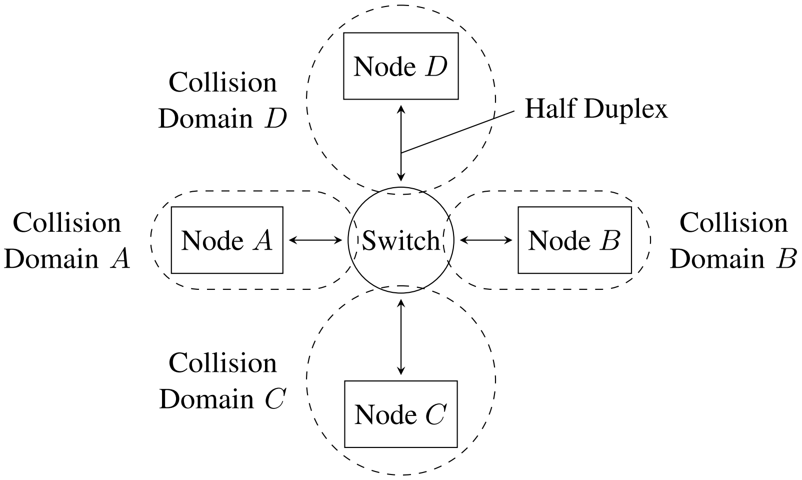


Figure 6.6: Using a microsegmented half-duplex switched-Ethernet topology yields a separate collision domain for each node.

separate collision domains. We also saw that this improves the bandwidth, range, and dependability of Ethernet. Switches, however, can even be used to eliminate collision domains altogether, making even more bandwidth available to the nodes and potentially making the network even more dependable. This is achieved by means of so-called microsegmented topologies.

A **microsegmented topology** is an Ethernet topology in which each node is connected to a switch by a separate cable. That is, nodes no longer share any segments. Instead, segments are point-to-point connections between one node and the switch. Another way to think of this is as follows: a microsegmented topology is one where the number of nodes per segment is reduced until there is only one node per segment.

Figure 6.6 shows an example of a microsegmented topology—or more precisely, because it only has one switch, it shows an example of a **microsegmented star topology**. The example has four nodes, labeled A, B, C, and D, each located on its own segment, and thus within its own collision domain. With this setup, messages from a given node can only collide with messages forwarded to it by the switch. For instance, in collision domain A, messages can only collide if node A transmits a message at the same time as the switch forwards a message to it. Is this an improvement?

When it comes to bandwidth, it certainly is. There are now many more combinations of node pairs that can communicate with each other without causing collisions that disrupt the communication between other pairs of nodes. As an example, just like in Figure 6.5, nodes A and B can exchange messages that do not collide with messages exchanged between nodes C and D. However, it is now also possible for A and C to exchange messages that do not collide with messages exchanged between B and D; or nodes B and C could exchange messages that do not collide with messages exchanged between A and D.

When it comes to dependability, a microsegmented topology may also be an improvement. For instance, if a segment suffers a failure, this may only affect the node located on that segment. Also, having only one node per segment makes it easier to locate a faulty node. On the other hand, there is also a significant downside to a microsegmented topology with a single switch: all communication now depends on the switch. This was not the case with a non-microsegmented topology, as in Figure 6.5, where some nodes may continue to exchange messages even if the switch (or bridge) fails. Then again, a microsegmented topology makes it easier to use the switch for error containment since it can, in principle, disconnect faulty nodes to prevent them from interfering with the communication between other nodes. Moreover, as we will see (Chapter 8), the problem of a switch constituting a single point of failure can be solved using switch redundancy.

When it comes to the range, a microsegmented topology can cover nodes spread out over a larger area, but the maximum distance between a pair of nodes is not necessarily increased.

By the way, when we are talking about microsegmented topologies, it is customary to use the term *link* instead of segment. So, from now on, let us follow this convention and talk about links instead of segments.

The links we have seen so far, on which collisions can occur, as in Figure 6.6, are called **half-duplex** links: they do not allow the traversal of messages in both directions simultaneously, but only in one direction at a time. However, since 1993, when Kalpana introduced the first commercial full-duplex Ethernet switch,[53] switches can also have **full-duplex** links: links that do allow the traversal of messages in both directions simultaneously, like a two-way highway on which cars drive in opposite directions. In fact, nowadays it is hard to find a switch that does not support full-duplex links since full-duplex operation for Ethernet was already standardized in 1997[54] and switch vendors try to comply with standards.[55]

[53] Jeffrey D. Baher. “Switch Pioneer Kalpana Now Offers Bidirectional Ethernet”. In: *PC Magazine* November 9 Issue (Nov. 1993).

[54] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks — Full Duplex Operation*. 802.3x-1997 (IEEE). 1997.

[55] Although full-duplex operation happens to be precisely one area where vendors often do not comply with IEEE standards. It is not that they do not sell full-duplex switches, but that they sell switches that *only* support full duplex, even though to be truly compliant with standards their switches

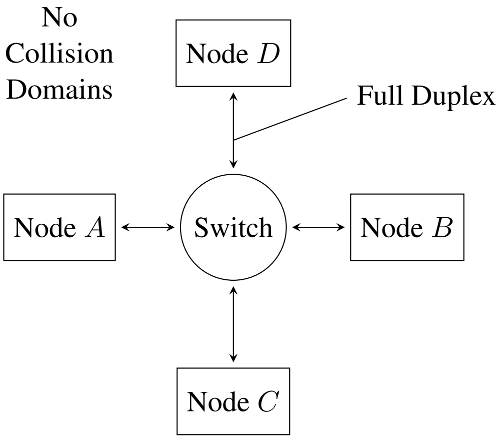


Figure 6.7: Using a microsegmented full-duplex switched-Ethernet topology eliminates all collision domains.

When links operate in full-duplex mode, only microsegmented topologies can be used. This, however, does not mean that the topology needs to be a simplex star; the topology may involve several interconnected switches.

Furthermore, when full-duplex links are used, there is no longer any possibility for collisions. In fact, with full-duplex links the classical CSMA/CD algorithm is not used and there are no collision domains at all. This is illustrated in Figure 6.7. Now a node can transmit its messages to the switch while, at the same time, the switch transmits messages in the other direction. This has several advantages.

The main advantage has to do with performance. Nodes and switches no longer have to waste any time retransmitting messages that were garbled because of collisions. Moreover, a node and a switch no longer have to take turns to transmit messages to each other through a link. Instead, they can transmit simultaneously to each other at full speed.

Another advantage of full-duplex links is that they can be much longer than half-duplex links. While in half-duplex mode the length of links (or segments) was limited to ensure that two simultaneously transmitting nodes would always detect a collision, in full-duplex mode the only limiting factor is signal attenuation.

Full-duplex links can also have dependability advantages. For instance, if communication in one direction is disrupted because of some permanent fault, communication may still be possible in the other direction. Then again, full-duplex links

should support both full- and half-duplex communication. This is particularly true for gigabit Ethernet switches.

can introduce a new source of problems, namely, **duplex mismatch**—a condition where the two endpoints of a link are using the link in different duplex modes (one uses half and the other full duplex).

6.5 Discussion

Ethernet is a prime example of a *disruptive technology*:[56] it starts out in one application domain; becomes better and better—whether in cost, performance, reliability, or some other attribute—until it is good enough for new domains; and then it spills over into these new domains, overwhelming the well-established solutions, ultimately displacing them. Several domains have already succumbed to this fate. One domain, however, has remained impervious for some time: the domain of distributed embedded systems. Historically this was due to Ethernet having nondeterministic transmission delays, lossy transmission (messages can be lost due to overflowing queues), and excessive failure recovery times. But these issues have been resolved. It may, thus, only be a matter of time for Ethernet to become the networking technology of choice in this domain as well. Betting on Ethernet thus seems wise. Yet, when it comes to distributed embedded systems that must be flexible in addition to highly reliable and hard real-time, Ethernet is still not good enough. But we are working towards changing that.

Our next stop is a survey of the current Ethernet-based FTT solutions. So let us go ahead and see which of these solutions is most promising to serve as the basis for our own communication subsystem.

[56] Joseph L. Bower and Clayton M. Christensen. *Disruptive Technologies: Catching the Wave*. Harvard Business Review, 1995.

Chapter 7

Current FTT Solutions

> If you can't criticise, you can't optimise.
>
> Harry Potter, in Eliezer Yudkowsky's *Harry Potter and the Methods of Rationality*

We set out to design a communication subsystem that brings us closer to future distributed embedded systems: a communication subsystem that can tolerate network faults, support hard real-time communication, and has the necessary operational flexibility to accommodate changing real-time requirements. We saw that Ethernet is a good choice to serve as the underlying network technology due to its low cost, high bandwidth, widespread expertise for it, and overall phenomenal track record. We also prepared ourselves by discussing several things: we introduced some vocabulary and concepts from dependability that allow us to think about reliability and fault tolerance (Chapter 2); we discussed how to design a fault-tolerant system (Chapter 3); we discussed replica determinism (Chapter 4); we discussed the basics of real-time communication (Chapter 5); and we discussed Ethernet (Chapter 6). We did, however, not talk much about how to make a network flexible towards changing real-time requirements. This is so because we will sidestep this problem. Instead of making Ethernet flexible ourselves, we will choose one of the existing FTT solutions as the basis for our work. But which one? This is the question we will tackle next.

We will begin with an overview of the FTT paradigm to see how it works in general (Section 7.1). Then, we will overview the current FTT solutions (Section 7.2), with the goal of describing their main differences and what their reliability

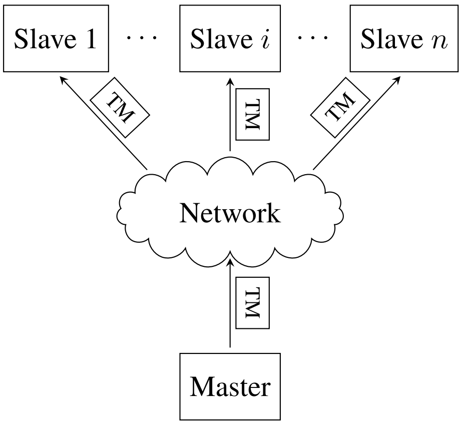


Figure 7.1: FTT uses a master/multi-slave approach to control access to the network. The master periodically broadcasts a single polling message, called trigger message (TM), to all slaves.

limitations are. Finally, we will discuss which FTT version we will base our work on (Section 7.3).

7.1 Flexible Time-Triggered Communication

The **flexible time-triggered communication paradigm**,[1] or FTT for short, is a way of coordinating message transmissions in a network. It is a way of deciding what nodes should transmit what types of message at what time so that all message exchanges are accomplished in a timely manner, before any deadlines are missed. Moreover—and this is the main reason we are interested in it—it enables these message exchanges to be *flexible*. Here flexible means that the parameters of the real-time messages—their periods, deadlines, transmission times, and so forth—are not static, but allowed to change at runtime if the changes result in a schedulable message set.

Figure 7.1 shows the general architecture of an FTT-based distributed embedded system. Several nodes are interconnected by means of a network. One of these is the **FTT master**. All others are **FTT slaves**. The master controls the communication by periodically broadcasting a special message, called **trigger message** (TM). This message periodically polls the slaves for message transmissions: it periodically tells

[1] Pedreiras and Almeida, "The Flexible Time-Triggered (FTT) Paradigm: an Approach to QoS Management in Distributed Real-Time Systems".

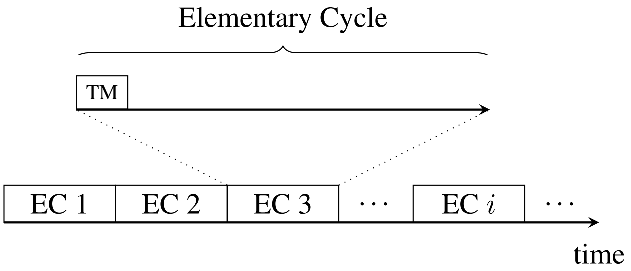


Figure 7.2: In FTT the communication time is structured into elementary cycles. Each elementary cycle begins with the broadcast of a trigger message.

the slaves *when* they are allowed to transmit and *what* they are allowed to transmit. Assuming no faults, it works as follows.

As illustrated in Figure 7.2, the trigger message structures the communication time into slots of fixed duration called **elementary cycles** (EC). Each time the trigger message is broadcast, a new elementary cycle begins. These cycles are numbered and are the granularity with which periods, offsets, minimum interarrival times, and relative deadlines are specified for real-time messages. Thus, a periodic message—or more precisely a stream of periodic messages (Section 5.2)—might be specified as having a period of 5 elementary cycles, a relative deadline of 2 elementary cycles, and an offset of 3 elementary cycles. The master, using such parameters, together with others such as the message transmission time, computes in each elementary cycle the schedule for the next one. That is, given the set of real-time message streams from which messages can be released, the master decides from which streams messages shall be transmitted in the next elementary cycle.[2] In the next elementary cycle this schedule is then conveyed in the trigger message. All slaves receive the trigger message, extract the schedule, and process it to find out what messages they are supposed to transmit in the just started elementary cycle. Thus, by transmitting the trigger message, the master essentially tells the slaves "hereby elementary cycle i begins and in this elementary cycle slave s_1 shall transmit the set of messages M_1, slave s_2 shall transmit the set of messages M_2, slave s_3 shall transmit the set of messages M_3", and so forth for all n slaves. The set of messages $M_i = M_1 \cup M_2 \cup M_3 \cup \cdots \cup M_n$ is the set of messages that the master scheduled during elementary cycle $i - 1$ for transmission in elementary cycle i.

[2] If the master does not have enough computing power to calculate the schedule each elementary cycle, then it may use more than one, but calculate the schedule for several elementary cycles into the future to compensate. This solution has been used as an option in FTT-CAN (Luís Almeida, Paulo Pedreiras, and José A. Fonseca. "The FTT-CAN Protocol: Why and How". In: *IEEE Transactions on Industrial Electronics* 49.6 [2002], pp. 1189–1201).

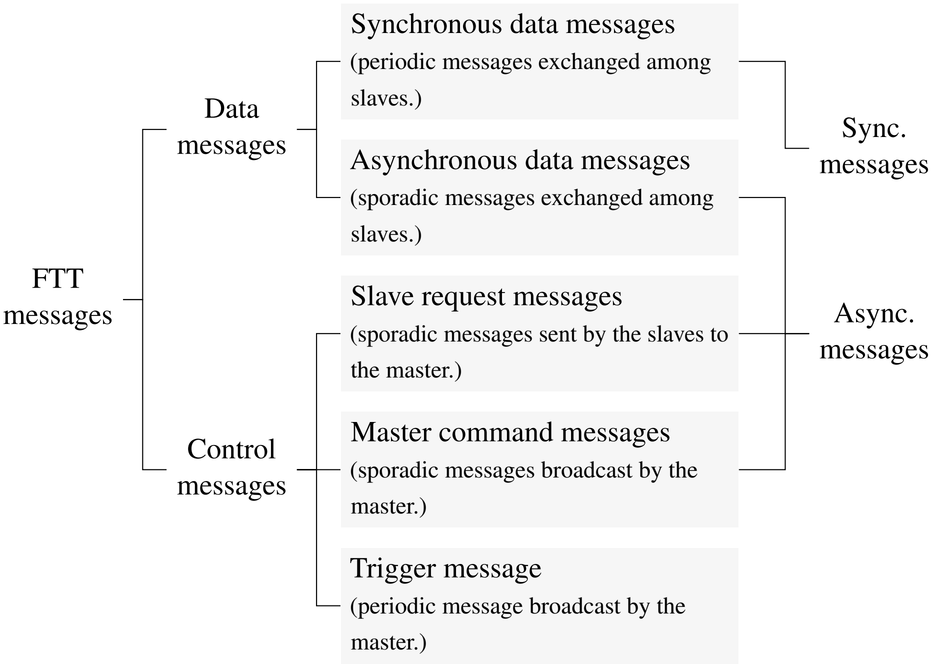

FTT messages
Data messages
Control messages
Synchronous data messages
(periodic messages exchanged among slaves.)
Asynchronous data messages
(sporadic messages exchanged among slaves.)
Slave request messages
(sporadic messages sent by the slaves to the master.)
Master command messages
(sporadic messages broadcast by the master.)
Trigger message
(periodic message broadcast by the master.)
Sync. messages
Async. messages

Figure 7.3: Common message types in FTT. Particular versions of FTT may have additional message types.

In FTT, messages can be classified according to two criteria. This is shown in Figure 7.3, which summarizes the types of messages common to all versions of FTT.

According to their purpose, FTT messages can be classified as data messages or control messages.[3] **Data messages** are used to exchange application data among slaves. **Control messages** are used to manage the communication and always involve the master—either as the recipient or the transmitter.

According to their temporal behavior, messages—other than the trigger message—can be classified as synchronous or asynchronous (right hand side of Figure 7.3). **Synchronous messages** are released, i.e., become ready for transmission, periodically, with the period expressed as an integer multiple of the elementary cycle length. For instance, a synchronous message might be released every three elementary cycles. **Asynchronous messages** are released sporadically, with the minimum interarrival time also expressed as an integer multiple of the elementary cycle length. For instance, an asynchronous message might be released at most every two elementary cycles. Both synchronous and asynchronous messages have real-time parameters such as deadlines and transmission times.

Depending on whether they are synchronous or asynchronous, data messages are classified as **synchronous data messages** or **asynchronous data messages**. All real-time messages that slaves exchange among themselves are either of one or the other type. Non-real-time messages may be treated as asynchronous messages with infinite deadlines or as a category on their own, depending on the version of FTT (more on this in Section 7.2).

Regarding control messages, there are three types shared by all versions of FTT. These are slave request messages, master command messages, and the trigger message. **Slave request messages** and **master command messages** are sporadic and considered asynchronous. Their purpose is mostly to allow the slaves and the master to negotiate changes in the real-time parameters. The trigger message, although periodic, is not considered synchronous.

The schedule that the master computes elementary cycle by elementary cycle is based on the contents of a local database, called the **system requirements database** (SRDB). Its contents are summarized in Figure 7.4. The database has two tables: the **synchronous requirements table** (SRT) for synchronous messages and the **asynchronous requirements table** (ART) for asynchronous messages. Moreover,

[3] Paulo Pedreiras. "Supporting Flexible Real-Time Communication on Distributed Systems". PhD thesis. University of Aveiro, 2003, pp. 122, 125–126; Alberto Ballesteros and Julián Proenza. *A Description of the FTT-SE Protocol.* Tech. rep. A-06-2013. Universitat de les Illes Balears, Dec. 2013, p. 4.

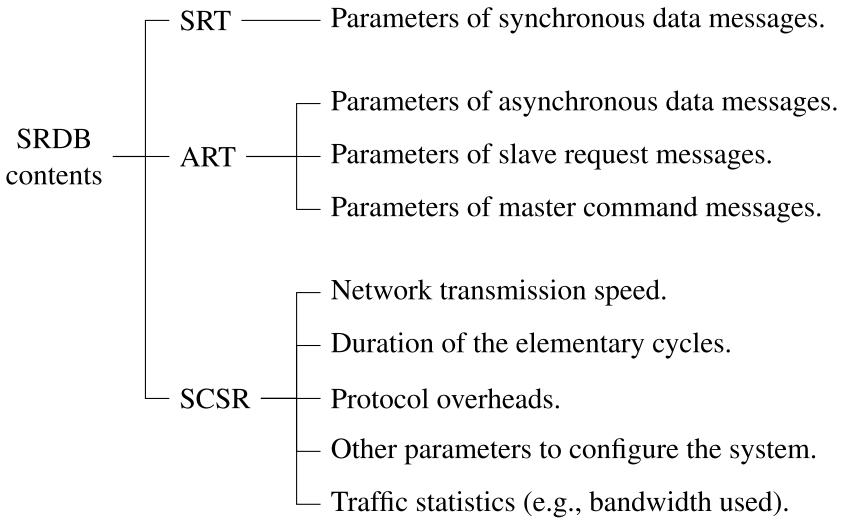

SRDB contents
SRT
Parameters of synchronous data messages.
ART
Parameters of asynchronous data messages.
Parameters of slave request messages.
Parameters of master command messages.
SCSR
Network transmission speed.
Duration of the elementary cycles.
Protocol overheads.
Other parameters to configure the system.
Traffic statistics (e.g., bandwidth used).

Figure 7.4: Basic contents of the system requirements database. The database is stored by the master. Particular versions of FTT may contain additional information in the database.

the SRDB also stores the **system configuration and status record** (SCSR), which specifies the network's transmission speed, the duration of the elementary cycles, overheads due to the underlying network technology (CAN or Ethernet), other system configuration parameters, and traffic statistics (e.g., amount of bandwidth used by synchronous messages versus asynchronous messages).[4]

The contents of the synchronous requirements table can be expressed as a set of tuples

$$\text{SRT} = \{\mathrm{msg}_i \mid \mathrm{msg}_i = (C_i, D_i, T_i, O_i, P_i), i \in [1, N_{\text{S}}]\}.$$

Here N_S is the number of current synchronous message streams. In each tuple, C_i is a transmission time, D_i is a relative deadline, T_i is a period, O_i is an offset, and P_i is an intrinsic message priority (a priority that informs the master about how valuable the message is to the application executed by the slaves). Each tuple msg_i specifies the parameters of one of the N_S synchronous message streams and is identified by an integer called a **synchronous stream identifier**.

The contents of the asynchronous requirements tables can be expressed similarly.

[4] Pedreiras, "Supporting Flexible Real-Time Communication on Distributed Systems", p. 79; Joaquim Ferreira. "Fault-Tolerance in Flexible Real-Time Communication Systems". PhD thesis. Universidade de Aveiro, 2005, p. 59.

If N_A is the number of current asynchronous message streams, then

$$\text{ART} = \{\text{msg}_i \mid \text{msg}_i = (C_i, D_i, I_i, P_i), i \in [1, N_\text{A}]\},$$

where, as before, C_i is a transmission time, D_i is a relative deadline, and P_i is an intrinsic message priority. The tuples differ from the previous ones in that they have a minimum interarrival time I_i, instead of a period and offset. As before, each tuple is identified by an integer, which this time is called an **asynchronous stream identifier**.

The above tables specify the minimum parameters that a master stores. Particular versions of FTT may store additional information.

The flexibility in FTT comes from the fact that the contents of the SRDB may change at runtime and, if they do, the master takes these changes into account for the scheduling of future elementary cycles.

What may cause the SRDB to change its contents? The answer is **update requests**, which can originate from an application running on top of the master node or from slaves. In the latter case, they reach the master in the form of slave request messages.

The master subjects each update request to an **admission control**, meaning that it executes a schedulability test to see if after the update the set of message streams would still be schedulable. If so, and only then, the requested change is incorporated into the SRDB. When that happens, the slaves need to be informed. This is done by the master broadcasting command messages. Each slave receives these and updates its **node requirements database** (NRDB), which is the counterpart to the SRDB in a slave.[5] The NRDB tells a slave, among other things, if it is the one that is supposed to transmit a message polled by the trigger message.[6]

So much for FTT in general. Let us now dive into the particularities of each version.

7.2 Versions of FTT

FTT can be used in different ways on top of different network technologies. This has led to several versions, among them three for Ethernet and one for Controller

[5] Ballesteros and Proenza, *A Description of the FTT-SE Protocol*, p. 7; Pedreiras, "Supporting Flexible Real-Time Communication on Distributed Systems", p. 82.

[6] Ricardo Marau. "Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management". PhD thesis. Universidade de Aveiro, 2009, p. 65.

Area Network[7] (CAN). We will begin with the three versions for Ethernet: FTT-Ethernet, FTT Switched Ethernet (FTT-SE), and HaRTES. Then we will take a look at FTT-CAN, the version for Controller Area Network. As you read, you might want to refer to Table 7.1 on the next page. It summarizes the main differences among the different versions.

7.2.1 FTT-Ethernet

FTT-Ethernet[8] was the first implementation of FTT on top of Ethernet. It was specifically designed to work on top of shared Ethernet,[9] meaning that it assumes a single collision domain and only one node can transmit at a time. Since message collisions in the underlying shared Ethernet would involve the CSMA/CD algorithm—and this algorithm hinders the meeting of real-time deadlines—FTT-Ethernet avoids message collisions altogether. How? First, it puts FTT's master/multi-slave communication on top of CSMA/CD. A node is therefore no longer allowed to transmit whenever it wants, but only within the elementary cycles in which it is polled. This, however, does not prevent collisions within an elementary cycle. After all, the master is likely to poll several slaves to transmit within the same elementary cycle. The solution is that trigger messages not only specify which slaves should transmit what messages in a given elementary cycle, but also when *within* an elementary cycle each transmission should start. For this the trigger messages indicate for each scheduled message an offset from the beginning of the corresponding elementary cycle, chosen such that no messages collide.

At the application layer, FTT-Ethernet uses producer/consumer cooperation among slaves. An Ethernet frame that carries an FTT message is not sent to a particular slave, but to all slaves. These then extract the message from the frame,[10] inspect it at the application layer, and check whether it is relevant to them. Specifically, they check whether the FTT message has an identifier for which they are a consumer according to their local NRDB.

[7] Robert Bosch GmbH. *CAN Specification Version 2.0.* Tech. rep. Robert Bosch GmbH, 1991; International Organization for Standardization (ISO). *Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling*. ISO 11898-1:2015 (ISO). 2015.

[8] Paulo Pedreiras et al. "FTT-Ethernet: A Flexible Real-Time Communication Protocol that Supports Dynamic QoS Management on Ethernet-Based Systems". In: *IEEE Transactions on Industrial Informatics* 1.3 (Aug. 2005), pp. 162–172.

[9] It can also work on top of switched Ethernet, but there it is inefficient when compared with the other versions of FTT specifically designed for switched Ethernet. We will only consider FTT-Ethernet on top of shared Ethernet.

[10] I distinguish between messages and frames. Messages are the FTT protocol data units and frames the Ethernet protocol data units.

Table 7.1: Main differences between FTT versions.

	FTT-Ethernet	FTT-SE	HaRTES	FTT-CAN
Underlying tech.	Shared Ethernet	Switched Ethernet	Switched Ethernet	Controller Area Network
MAC layer	M/S[a] over CSMA/CD	M/S over full duplex	M/S over full duplex	M/S over CSMA/BA
Application layer	Producer/consumer	Publisher/subscriber	Publisher/subscriber	Producer/consumer
Synchronous traffic	Blind Polling	Blind Polling	Blind Polling	Blind Polling
Asynhronous traffic	Blind polling	Signaled Polling	Shaped by switch	Confined CSMA/BA[b]
Master location	Node	Node	Switch	Node[k]
Topology	Bus or star	Star or tree[i]	Star	Bus[g]
Shared channel	Yes[c]	No[d]	No[d]	Yes[c]
Collision domain	Single	None[d]	None[d]	—
Non-FTT nodes	No	No	Yes	No
Standard hardware	Yes	Yes	No[e]	No[f]
SRDB tables	SRT, ART, NRT[j], SCSR	SRT, ART, SCSR	SRT, ART, SCSR	SRT, ART, NRT, SCSR
Single points of failure	Master, bus/hub, TM, master link[h]	Master, master link, switch, TM	Switch, TM	None
Fault-tolerance mechanisms	Native Ethernet error detection	Native Ethernet error detection	Native Ethernet error detection, policing of correct timing of real-time FTT messages	Native CAN error detection, master and bus replication, bus guardians, internally duplicated and compared masters, detection of permanent bus errors, scheduling retransmissions

[a] M/S: master/multi-slave.

[b] Confined to an asynchronous window within each elementary cycle.

[c] Shared broadcast medium.

[d] Uses microsegmented star topology with full-duplex links.

[e] Requires FTT-enabled switch.

[f] Fault-tolerant versions require custom bus guardians and internally duplicated and compared masters; masters may require an EC scheduling coprocessor.

[g] Use of star topologies seems feasible, but has not been demonstrated. See David Gessner et al. "A First Qualitative Evaluation of Star Replication Schemes for FTT-CAN". in: *Proceedings of the 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2012)*. IEEE, 2012.

[h] If hub is used.

[i] FTT-SE has also been extended to tree topologies that use multiple switches. See Mohammad Ashjaei et al. "Dynamic Reconfiguration in Multi-Hop Switched Ethernet Networks". In: *ACM SIGBED Review* 11.3 (2014).

[j] Non-real-time table. Stores parameters for non-real-time messages.

[k] If the fault-tolerant version of FTT-CAN is used.

Slave 1
Slave 2
Slave 3
Master

Figure 7.5: FTT-Ethernet architecture when using an Ethernet bus.

To handle non-real-time traffic, a master in FTT-Ethernet not only stores a synchronous and an asynchronous requirements table, but also a **non-real-time table** (NRT).[11] Based on the parameters in this table, the non-real-time messages are polled by the master if, after polling the real-time asynchronous messages, there is still time available within the elementary cycle.[12]

The polling in FTT-Ethernet is **blind polling**:[13] all messages—whether periodic, aperiodic, or sporadic, and whether real-time or not—are polled by the trigger message when they *may* have been released, i.e., when they *might* be ready for transmission. This is natural for synchronous messages, which are periodic and released with predictable regularity. But it is inefficient for asynchronous and non-real-time messages, which are sporadic and aperiodic, respectively. For instance, an asynchronous message may have a *minimum* interarrival time of 5 elementary cycles, but an actual interarrival time may be much longer, perhaps hundreds of elementary cycles. Nevertheless, the master has no knowledge about this: it is oblivious to the actual release times. Thus, it regularly polls asynchronous and non-real-time messages just in case, resulting in wasted bandwidth.

Regarding its topology, since FTT-Ethernet builds on shared Ethernet, we have two basic options: use an Ethernet bus, as illustrated in Figure 7.5, or use an Ethernet hub, as illustrated in Figure 7.6. If a hub is used, the link that connects a slave to the hub is called a **slave link**. The one that connects the master is called a **master link**.

(More complex bus-based or hub-based topologies are also possible, but I will disregard them since, as I said in Section 1.4 on page 11, I am not considering multi-hop topologies, i.e., topologies with more than one switch or hub.)

Since FTT-Ethernet uses shared Ethernet, all nodes share a single broadcast channel: only one message can be transmitted successfully at a time and it is not possible for disjoint subsets of nodes to exchange messages simultaneously. As a consequence, all nodes have (in the absence of faults) the same view of what

[11] Pedreiras et al., "FTT-Ethernet: A Flexible Real-Time Communication Protocol that Supports Dynamic QoS Management on Ethernet-Based Systems", section III.C.

[12] Ibid., Sec. III.B.

[13] Marau, "Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management", p. 50.

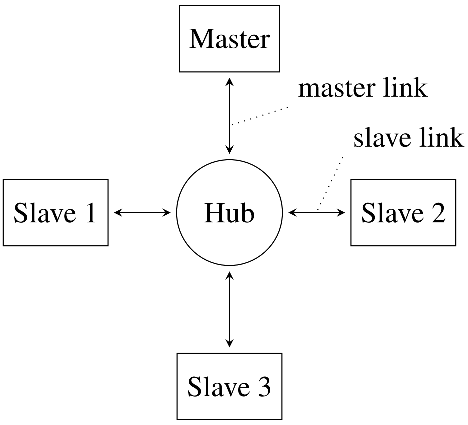


Figure 7.6: FTT-Ethernet architecture when using an Ethernet hub.

happens within each elementary cycle. This is not true for the switched Ethernet versions of FTT, i.e., FTT-SE and HaRTES. There each node may see a different set of messages passing by within the same elementary cycle.

Reliability Limitations of FTT-Ethernet

Scrutinizing FTT-Ethernet for its reliability limitations, we note that it has several single points of failure. If a bus is used, the single points of failure are the master and the bus. If a hub is used, the single points of failure are the master, the hub, and the master link.

The messages sent by the master, i.e., the trigger messages and master command messages, are also a problem for achieving high reliability in FTT-Ethernet. After all, these messages must reach all correct slaves for FTT to work. Yet, even if no component is affected by permanent faults, transient faults could prevent them from reaching their destination.

FTT-Ethernet is also extremely vulnerable to timing failures. At worst, a node becoming a babbling idiot may keep transmitting Ethernet frames, disregarding the polling of the trigger message and thus preventing other nodes from transmitting their scheduled frames on time. This is especially problematic considering the *channel capture effect* of CSMA/CD, a phenomenon where under high load one node can monopolize the shared medium.[14] But even if a node suffers a more benign

[14] K. K. Ramakrishnan and Henry Yang. “The Ethernet Capture Effect: Analysis and Solution”. In: *Proceedings of the 19th IEEE Conference on Local Computer Networks (LCN 1994)*. IEEE. 1994, pp. 228–240.

timing failure, the timely delivery of messages may still be hampered on the whole network: a node that fails to transmit with the correct offset within an elementary cycle may unravel a whole cascade of message collisions, causing multiple message exchanges to miss their deadlines. Messages may even collide with the trigger message, postponing its successful transmission.

If FTT-Ethernet is used with a bus topology, the bus is not only a single point of failure, but one that is especially vulnerable. A shortened or ruptured cable anywhere can cause a failure of the whole communication. In contrast, if a hub is used, a cable's failure may only prevent one node from communicating. After all, a hub, contrary to a bus, is an active device that can, at least in principle, contain some errors at its ports. Moreover, the potential for making a bus reliable through replication is lower than in a star topology. A replicated bus is liable to spatial proximity faults because "all redundant buses necessarily come into close proximity at each node".[15] Thus, if something smashes into a node, it is likely to sever all buses. In replicated star topologies, on the other hand, if a node is destroyed because of an impact, it may only affect it and its links.

The main features of FTT-Ethernet are summarized in the second column of Table 7.1 on page 131.

7.2.2 FTT Switched Ethernet

FTT Switched Ethernet,[16] or FTT-SE for short, is the second incarnation of the FTT paradigm on Ethernet. As its name suggests, it is not based on shared Ethernet, but on switched Ethernet. Specifically, an FTT-SE network uses a plain commercial off-the-shelf Ethernet switch in a microsegmented full-duplex topology, as shown in Figure 7.7.[17] This means that several message exchanges can occur

[15] John Rushby. "Bus Architectures for Safety-Critical Embedded Systems". In: *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT 2001).* (Tahoe City, CA, USA). Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Lecture Notes in Computer Science. Springer. 2001, pp. 306–323, p. 312.

[16] Marau, Almeida, and Pedreiras, "Enhancing Real-Time Communication over COTS Ethernet Switches"; Marau, "Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management".

[17] We will only consider FTT-SE with a single switch. There is, however, also research into scaling it to multi-hop networks. Relevant papers include Farahnaz Yekeh et al. "Exploring alternatives to scale FTT-SE to large networks". In: *Proceedings of the 6th IEEE International Symposium on Industrial and Embedded Systems (SIES 2011).* IEEE. 2011, pp. 107–110; Mohammad Ashjaei et al. "Evaluation of Dynamic Reconfiguration Architecture in Multi-Hop Switched Ethernet Networks". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014).* IEEE. 2014, pp. 1–4; Mohammad Ashjaei et al. "A Compact Approach to Clustered Master Slave Ethernet Networks". In: *Proceedings of the 9th IEEE International Workshop*

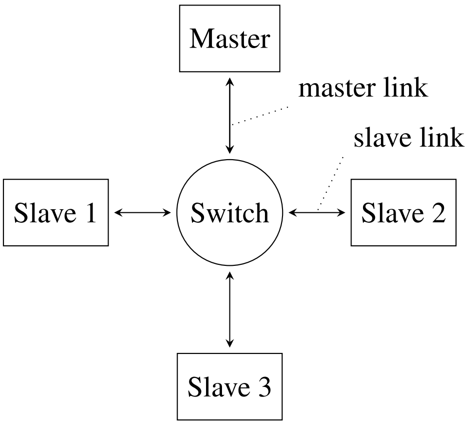


Figure 7.7: FTT-SE architecture.

in parallel without collisions: while one subset of nodes exchange one set of messages, other subsets may exchange other messages. To take advantage of this, FTT-SE no longer uses a producer/consumer approach at the application layer, but a publisher/subscriber approach. And at the Ethernet layer, frames are no longer systematically broadcast, but transmitted with unicast or multicast addressing when appropriate. As a result, there is no longer a single shared view of what occurs within each elementary cycle: one node may see one set of messages traveling up and down its link, while another node may see a different set. All nodes, however, still agree on when each elementary cycle starts and ends, and on what messages have been scheduled overall.

To exploit the multiple parallel transmission paths, the FTT-SE master stores additional information. It stores what message exchanges are unicast, multicast, or broadcast, and which end nodes are involved. Using this information, it then computes which messages follow disjoint paths, which helps it build schedules that exploit the parallelism.[18]

There are still more differences with respect to FTT-Ethernet.

Since FTT-SE uses a switch with full-duplex links, collisions can no longer occur and it is unnecessary for the trigger message to specify message transmission offsets. Hence, slaves transmit messages immediately after decoding the trigger

on Factory Communication Systems (WFCS 2012). IEEE. 2012, pp. 157–160; Mohammad Ashjaei et al. "Dynamic Reconfiguration in Multi-Hop Switched Ethernet Networks". In: *ACM SIGBED Review* 11.3 (2014)

[18] Marau, "Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management", p. 49.

message and the switch takes care of serializing and forwarding the messages to their destinations within each elementary cycle.[19]

Another key difference between FTT-SE and FTT-Ethernet is that blind polling is no longer used for asynchronous messages. Instead, an approach is used that we may call **signaled polling**: slaves use special control messages to signal to the master when they have pending asynchronous messages. Thus, the master can now do an informed polling: it only schedules asynchronous messages when the slaves have asynchronous messages ready for transmission.[20] The control messages used for the signaling are transmitted at the same time as the trigger message[21] and are neither synchronous nor asynchronous messages, but a category on their own, like the trigger message.[22]

FTT-SE also differs from FTT-Ethernet in how non-real-time messages are handled. The master no longer has a separate table for them, but instead considers them as asynchronous messages with a minimum priority and infinite relative deadlines.[23] This means that signaled polling is also used for non-real-time messages.

Reliability Limitations of FTT-SE

The single points of failure in FTT-SE are the master, the master link, the Ethernet switch, and the trigger message. At first glance we might therefore conclude that the reliability limitations in FTT-SE are essentially the same as in FTT-Ethernet when a hub is used. But on second thought we should realize that FTT-SE's use of switched and full-duplex communication is a major advantage: timing failures are now significantly less problematic.

Since message collisions can no longer occur, an untimely message can no longer postpone all other transmissions. For instance, if in Figure 7.7 slave 1 is faulty and transmits messages to slave 3 while the master is broadcasting its trigger message, the trigger message will only be postponed on the link of slave 3—not because of message collisions, but because of queuing at the switch output port. Moreover, a babbling idiot can now no longer monopolize the whole communication since

[19] Marau, "Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management", p. 49.

[20] Ricardo Marau, Paulo Pedreiras, and Luís Almeida. "Asynchronous Traffic Signaling over Master Slave Switched Ethernet Protocols". In: *6th International Workshop on Real-Time Networks (RTN 2007)*. 2007.

[21] Ballesteros and Proenza, *A Description of the FTT-SE Protocol*, pp. 9, 10.

[22] Ibid., p. 11.

[23] Marau, "Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management", p. 59.

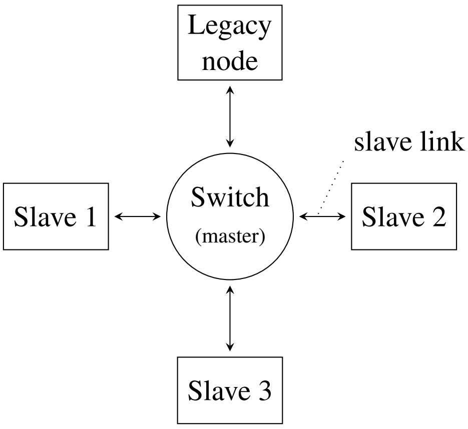


Figure 7.8: HaRTES architecture.

parallel transmissions are now possible (although it can still interfere with the timing of other messages—again, because of queuing).

Finally, an additional advantage of full-duplex links, which we already pointed out earlier (Section 6.4.2 on page 120), is that a fault affecting a link may only affect messages traveling in one direction.

The main features of FTT-SE are summarized in the third column of Table 7.1 on page 131.

7.2.3 HaRTES

FTT-Ethernet and FTT-SE are software-only implementations of the FTT paradigm. They work with standard Ethernet hardware. In many contexts this is an advantage: such hardware is cheap and widely available. But it also has its downsides. Both FTT-Ethernet and FTT-SE are vulnerable to timing failures since standard hardware has no mechanisms to guarantee timely transmissions. In fact, both FTT-Ethernet and FTT-SE trust the slaves to transmit when the master tells them to, even if this trust is unwarranted—nodes may be faulty or may completely ignore the master's instructions if they are non-FTT compliant legacy nodes. Moreover, the solutions they adopt to work with standard hardware have performance limitations: FTT-Ethernet wastes bandwidth with its blind polling, while FTT-SE adds unwanted latency to the transmission of asynchronous messages with its signaled polling. To solve these issues HaRTES was developed.

HaRTES[24] stands for hard real-time Ethernet switching. Figure 7.8 shows its architecture. It is based on a simplex (non-replicated) microsegmented star topology with full-duplex links.[25] The center of the star is a custom Ethernet switch, called an **FTT-enabled switch**[26] and whose internals are shown in Figure 7.9. The switch incorporates an FTT master (the shaded area at the top left of Figure 7.9). This moves the master to a privileged central position within the network from which it can exert tight transmission control. Since the master is now part of the switch, the switch has direct access to the SRDB and therefore can discard messages from slaves that do not comply with the contents of the SRDB. Moreover, the switch, or more specifically the dispatcher and the port dispatchers (Figure 7.9), now have access to each elementary cycle schedule and can apply traffic shaping to all messages to ensure that these comply with the bandwidth allotted to them for each elementary cycle (in Section 8.7.2 we will further discuss how messages are transmitted during an elementary cycle in HaRTES).

The master no longer needs to poll asynchronous messages at all. Slaves are free to transmit their asynchronous messages as soon as they are released. If the timing of these messages complies with the minimum interarrival time stored in the asynchronous requirements table (ART), the switch puts them into queues (boxes labeled “Packet list” in the figure) and forwards them at appropriately scheduled times (using the port dispatchers shown below the packet lists). Otherwise, they fail the validation process (boxes labeled “Validate” on the right-hand side of the figure), are considered the result of a timing failure, and the switch drops them.[27] This makes the handling of asynchronous messages in HaRTES more efficient than in both FTT-Ethernet and FTT-SE.

By moving the transmission control to the switch, HaRTES can now also work with non-FTT compliant legacy nodes. These nodes transmit their messages when-

[24] Santos et al., “A Synthesizable Ethernet Switch with Enhanced Real-Time Features”; Rui Gabriel Viegas dos Santos. “Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications”. PhD thesis. Universidade de Aveiro, 2010, chap. 3.

[25] The architecture we consider is single-hop. For research on using HaRTES on multi-hop networks you may want to refer to the following papers: Mohammad Ashjaei et al. “Supporting Multi-Hop Communications with HaRTES Ethernet Switches”. In: *Proceedings of the IEEE Real-Time Systems Symposium Work-in-Progress session (RTSS 2013).* Vancouver, Canada, Dec. 2013; Mohammad Ashjaei et al. “Improved Message Forwarding for Multi-Hop HaRTES Real-Time Ethernet Networks”. In: *Journal of Signal Processing Systems* 84.1 (2016), pp. 47–67; Mohammad Ashjaei et al. “Dynamic Reconfiguration in HaRTES Switched Ethernet Networks”. In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016).* IEEE. 2016, pp. 1–8.

[26] Rui Santos et al. “Designing a Costumized [sic] Ethernet Switch for Safe Hard Real-Time Communication”. In: *Proceedings of the 7th IEEE International Workshop on Factory Communication Systems (WFCS 2008).* IEEE. 2008, pp. 169–177.

[27] Santos, “Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications”, p. 65.

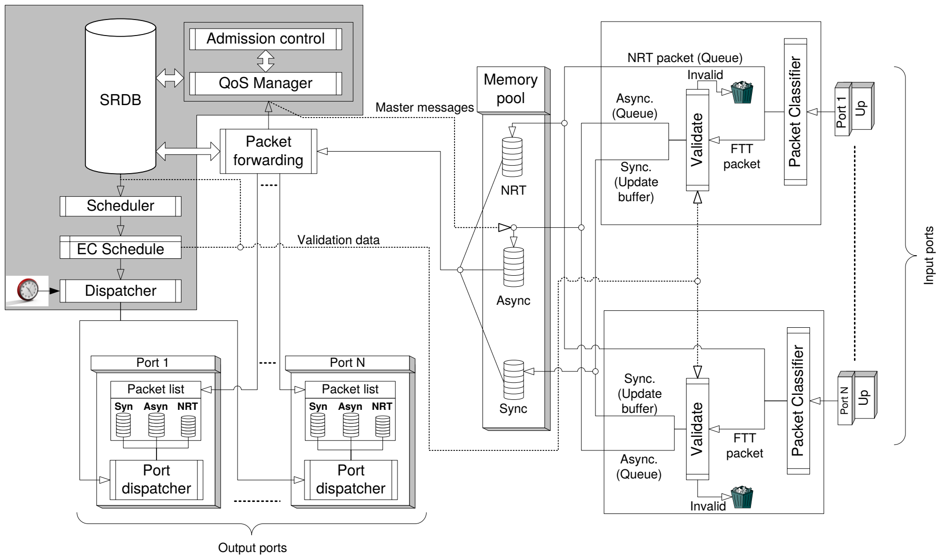


Figure 7.9: HaRTES switch internals. (Reprinted from Rui Gabriel Viegas dos Santos. "Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications". PhD thesis. Universidade de Aveiro, 2010, p. 64.)

ever they want, ignorant of any elementary cycles or schedules. The FTT-enabled switch takes care of queuing these messages separately (queues labeled "NRT" in Figure 7.9) and forwards them in the background using standard MAC-address-based forwarding, at times that do not interfere with the timely delivery of FTT messages. Thus, a HaRTES switch essentially behaves towards non-real-time traffic as a commercial off-the-shelf Ethernet switch, although one with reduced bandwidth because non-real-time traffic gives way to real-time traffic within the switch.[28]

Regarding synchronous messages, they are still polled elementary cycle by elementary cycle, like in all FTT versions. The difference is that the polling message—the trigger message—now originates at the switch (box labeled "Dispatcher" within the shaded area). Moreover, synchronous messages are also subject to a validation process: if they arrive at the switch unscheduled, they are dropped.[29]

(The QoS manager shown in Figure 7.9, to the right of the SRDB, is an optional component in HaRTES[30] and we will ignore it.)

At the application layer HaRTES uses a publisher/subscriber cooperation model,

[28] Ibid., p. 65.
[29] Ibid., p. 65.
[30] Ibid., p. 64.

like FTT-SE. Who is the publisher and who are the subscribers for a particular FTT message is stored in the SRDB, along with the real-time parameters of the given message.

Reliability Limitations of HaRTES

The single points of failure in HaRTES are only two: the FTT-enabled switch and the trigger message. This is certainly an advantage over the previous FTT versions for Ethernet. However, we also need to factor in that an FTT-enabled switch is a more complex device than a commercial off-the-shelf switch, hub, or bus. It might therefore have a higher failure rate.

HaRTES is also the first FTT version we have seen that has fault-tolerance mechanisms beyond the stock Ethernet error detection. Specifically, the validation process to which messages are subjected at the switch eliminates certain faulty behaviors of slaves that are caused by timing failures. The validation process therefore essentially implements a guardian (Section 3.5.2 on page 64).

The main features of HaRTES are summarized in the fourth column of Table 7.1 on page 131.

7.2.4 FTT-CAN

FTT-CAN stands for Flexible Time-Triggered communication for Controller Area Network (CAN). It is not based on Ethernet, but on CAN, a fieldbus that originated in the automotive industry during the 1980s.

Like the original shared Ethernet, CAN relies on a single shared broadcast bus. Other than that, it is quite different. When message collisions occur they are resolved not at the frame level, as in CSMA/CD, but at the bit level. Moreover, the resolution is deterministic by being based on priorities: if several nodes compete for access to the bus, the one transmitting the message with the highest priority (lowest CAN identifier) wins. This scheme is known as **carrier sense multiple access with bitwise arbitration**, or **CSMA/BA**. For the details you may want to refer to the CAN specification,[31] the ISO standard,[32] or the book by Voss.[33]

FTT-CAN puts FTT's master/multi-slave access control on top of CSMA/BA. The master is located in a node that periodically broadcasts the trigger message on

[31] Robert Bosch GmbH, *CAN Specification Version 2.0.*

[32] ISO, *CAN – Part 1: Data Link Layer and Physical Signalling.*

[33] Wilfried Voss. *A Comprehensible Guide to Controller Area Network.* Copperhill Media, 2008.

the shared CAN bus. As usual, the trigger message polls synchronous messages. Contrary to FTT-Ethernet, no transmission offsets are specified. Instead, all slaves transmit the scheduled synchronous messages within the elementary cycle, with contention being resolved using CAN's message priorities and the CSMA/BA algorithm. As to asynchronous and non-real-time messages, these are not polled by the trigger message, but transmitted within a confined interval of each elementary cycle, called the asynchronous window. Within this window messages are transmitted as in native CAN. For non-real-time messages, the FTT-CAN master stores an additional table in its SRDB called, like in FTT-Ethernet, the non-real-time table (NRT). This table stores the size of the longest non-real-time message that can be produced by each slave, which is useful for determining the size of the asynchronous window and thus prevent window overruns.[34]

Since our focus is on Ethernet and fault-tolerance, let us leave the basics of FTT-CAN at that and move on to the fault-tolerance mechanisms that have been designed for it, which will be more interesting to us. You can find more details on FTT-CAN in the paper by Almeida, Pedreiras, and Fonseca[35] and the thesis by Pedreiras.[36]

Fault Tolerance in FTT-CAN

As we have seen, the different Ethernet-based versions of FTT all have single points of failure. So do the initial implementations of FTT-CAN. There the single points of failure were the master node, the CAN bus, and the trigger message. Contrary to the Ethernet versions of FTT, however, mechanisms have already been designed that eliminate these single points of failure and make FTT-CAN fault tolerant.

The two main lines of work involved master replication[37] and bus replication.[38]

[34] Valter Filipe da Silva. "Flexible Redundancy and Bandwidth Management in Fieldbuses". PhD thesis. Universidade de Aveiro, 2010, p. 93.

[35] Almeida, Pedreiras, and Fonseca, "The FTT-CAN Protocol: Why and How".

[36] Pedreiras, "Supporting Flexible Real-Time Communication on Distributed Systems".

[37] Ferreira, "Fault-Tolerance in Flexible Real-Time Communication Systems"; Joaquim Ferreira et al. "Combining Operational Flexibility and Dependability in FTT-CAN". in: *IEEE Transactions on Industrial Informatics* 2.2 (2006), pp. 95–102; Ricardo Marau et al. *Assessment of FTT-CAN Master Replication Mechanisms for Safety Critical Applications*. Tech. rep. SAE Technical Paper, 2006.

[38] Valter Filipe Silva, José Alberto Fonseca, and Joaquim Ferreira. "Adapting the FTT-CAN Master for multiple-bus operation". In: *Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN 2007)*. Vol. 1. IEEE. 2007, pp. 305–310; Valter Filipe da Silva, Joaquim Ferreira, and José Alberto Fonseca. "Master Replication and Bus Error Detection in FTT-CAN with Multiple Buses". In: *Proceedings of the 12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2007)*. IEEE. 2007, pp. 1107–1114; Silva, "Flexible Redundancy and Bandwidth Management in Fieldbuses".

Both also dealt with the replication of the trigger message. Can we reuse these mechanisms for Ethernet? Unfortunately no because they rely on particularities of the CAN protocol.

As one example, the mechanisms used to tolerate message omissions due to transient channel faults rely on CAN's error signaling, which purportedly globalizes errors and thus ensures that a non-faulty transmitter knows when a transmission is corrupted. In Ethernet, however, there is no error signaling. (CSMA/CD provides signaling of collisions by means of a collision enforcement jam signal. This signaling, however, is only designed to detect collisions, not errors, and is not used at all in full-duplex switched Ethernet—collisions are not errors, but an integral part of the CSMA/CD algorithm.)

By the way, above I said that CAN "purportedly" globalizes errors because there are some scenarios where CAN's error signaling actually does not globalize errors.[39] They have to do with certain error combinations occurring during the last few bits of a CAN frame that may lead to inconsistent message omissions: some nodes accept a message, while others reject it. That CAN globalizes errors is therefore, strictly speaking, false. Nevertheless, CAN's error globalization may work sufficiently well in practice: Ferreira concluded from experiments that inconsistent message omissions occur less than 10^{-9} times per hour, "the commonly accepted threshold for safety-critical applications".[40]

As another example of incompatibility, FTT-CAN uses a passive replication mechanism for the masters, which also relies on CAN's properties. There is one active master and one or more backup masters. All masters are fail-silent by being internally duplicated and compared.[41] Thus, when the active master transmits a message, it either does so correctly and everybody receives the message without errors, or the transmission fails due to a channel error and everybody is aware of the error because it has been globalized by one or more nodes. Thus, when a backup master at the start of an elementary cycle does not receive the trigger message from the active replica, nor an error signal, it knows that the active master has not transmitted the trigger message. And this, because the masters are fail-silent, can only occur if the active master failed. Thus, in FTT-CAN the backup masters know at the beginning of each elementary cycle if the active replica has failed and one

[39] Jose Rufino et al. "Fault-Tolerant Broadcasts in CAN". in: *Proceedings of the 28th IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS 1998).* Washington, DC, USA: IEEE Computer Society, 1998, p. 150; Julián Proenza and José Miro-Julia. "MajorCAN: A Modification to the Controller Area Network Protocol to Achieve Atomic Broadcast". In: *IEEE International Workshop on Group Communication and Computations.* Taipei, Taiwan, 2000.

[40] Ferreira, "Fault-Tolerance in Flexible Real-Time Communication Systems", p. 166.

[41] Joaquim Ferreira et al. "Achieving Fault Tolerance in FTT-CAN". in: *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS 2002)* (2002), pp. 125–132.

of them can immediately take over. In Ethernet, in contrast, this does not work. In Ethernet not receiving a particular message from a fail-silent node does not allow us to conclude that the node has failed—the fail-silent node might have transmitted the message correctly and it was only the reception that failed at the receiver's end. Hence, a backup master should not necessarily take over just because it failed to receive the trigger message from the active one.

In FTT-CAN there is also a mechanism to detect permanent bus errors if bus replication is used.[42] It is based on having backup masters check that they receive the trigger message from the active master on each of the replicated buses. That is, "a bus fault is detected whenever a backup master receives, in a given elementary cycle, less trigger messages than the number of buses".[43] But this again relies on CAN's error globalization. Without it, a node not receiving an expected message through a particular channel (or bus) cannot conclude that the channel has failed. It might simply be that the transmitter did not detect the error—since it was not globalized—and thus did not retransmit the message.

Marques and several others have also worked on enhancing the scheduler of FTT-CAN to reschedule messages that have been corrupted due to transient channel errors.[44] This mechanism, however, is not directly applicable to Ethernet either because it relies, once again, on CAN's error globalization: they assume that channel faults are symmetric, by which they mean that message corruptions are consistently detected by all nodes. But error globalization does not exist in shared nor switched Ethernet. Hence, in Ethernet there would be no immediate way for a master to know whether the transmission of a message failed somewhere in the network and thus should be rescheduled.

Finally, FTT-CAN can also be made fault-tolerant, at least in theory, by combining master replication with ReCANcentrate,[45] a fault-tolerant alternative to the

[42] Silva, Ferreira, and Fonseca, "Master Replication and Bus Error Detection in FTT-CAN with Multiple Buses".

[43] Silva, "Flexible Redundancy and Bandwidth Management in Fieldbuses", p. 79.

[44] Luís Marques et al. "Towards Efficient Transient Fault Handling in Time-Triggered Systems". In: (2011); Luís Marques et al. *Towards Efficient Transient Fault Handling in Time-Triggered Systems*. Tech. rep. Aveiro: Universidade de Aveiro, 2011; Luís Marques et al. "Tolerating Transient Communication Faults with Online Traffic Scheduling". In: *IEEE International Conference on Industrial Technology (ICIT 2012)*. Athens, 2012; Luís Marques et al. "Comparing Scheduling Policies for a Message Transient Error Recovery Server in a Time Triggered Setting". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014, pp. 1–6; Luís Marques et al. "Error Recovery in Time Triggered Communication Systems Using Servers". In: *Proceedings of the 8th IEEE International Symposium on Industrial and Embedded Systems (SIES 2013)*. IEEE. 2013, pp. 205–212.

[45] Manuel Barranco, Julián Proenza, and Luís Almeida. "Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate". In: *Computer* 42.5 (May 2009), pp. 66–73.

traditional CAN bus that relies on a replicated star topology comprised of two CAN hubs that provide error containment. Some of my colleagues, and I myself, worked on this, but it remained only preliminary work.[46]

The main features of FTT-CAN are summarized in the last column of Table 7.1 on page 131.

7.3 Discussion

So what version of FTT is most promising as the basis for our own fault-tolerant communication subsystem? The answer is quite clearly HaRTES. FTT-CAN, although fault tolerant, is incompatible with Ethernet. FTT-Ethernet, by being based on shared Ethernet is bandwidth inefficient and quite vulnerable to timing failures. FTT-SE is less vulnerable, but has undesired latency for asynchronous messages and by having the master located in a node has one more single point of failure than HaRTES, namely, the master link. Moreover, it has no traffic policing, while HaRTES does.

Overall, HaRTES is the most bandwidth efficient FTT solutions for Ethernet. It is based on a star topology, which can be replicated without suffering from spatial proximity faults. It can already police the traffic by having a switch with direct access to the SRDB and thus knowledge about what constitutes correct traffic patterns. And it has the fewest number of single points of failure. Thus, qualitatively it is the winner.

Now, having made a choice, it is time to come up with our very own solution: a communication subsystem based on Ethernet and FTT—and more specifically on HaRTES—that can tolerate both permanent and transient faults while still supporting the main features of the FTT paradigm—the timely exchange of both periodic and sporadic real-time messages and the support for updating the real-type parameters of these messages at runtime.

[46] Manuel Barranco et al. “Towards the Integration of Flexible Time Triggered Communication and Replicated Star Topologies in CAN”. in: *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2011).* IEEE, 2011, pp. 1–4; David Gessner et al. “A First Qualitative Evaluation of Star Replication Schemes for FTT-CAN”. in: *Proceedings of the 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2012).* IEEE, 2012; Julián Proenza et al. “Using FTT and Stars to Simplify Node Replication in CAN-based Systems”. In: *Proceedings of the 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2012).* IEEE, Sept. 2012.

Part II

Main contribution

Chapter 8

The Flexible Time-Triggered Replicated Star for Ethernet

> To them, equipment failure is terrifying. To me, it's "Tuesday".
>
> Mark Watney, in Andy Weir's *The Martian*

We decided upon using HaRTES as a foundation for our communication subsystem (Chapter 7). Now it is time to build on HaRTES and design this subsystem to prove our thesis—to prove that, using Ethernet as the underlying technology, fault tolerance can be added to the FTT communication paradigm.

We will begin by listing our requirements and some preliminary design choices that will simplify our design (Section 8.1). Then we will define a fault model (Section 8.2), which is an essential first step for building any fault-tolerant system. With the fault model and requirements in mind, we will define the architecture for our communication subsystem (Section 8.3). This will give us the main components of our system. To tolerate faults within these components our communication subsystem will have to be equipped with appropriate fault-tolerance mechanisms; and to design these mechanisms we will have to know how the components can fail. For this in turn we must know what the components are supposed to do when they are correct. In other words, we will first have to define what the function of each type of component is (Section 8.4). Only then will we be able to define a first failure model that specifies how components can deviate from their function, i.e., fail (Section 8.5). With the failure model as the basis, we will then restrict the failure semantics of the components to benign ones (Section 8.6) and finally design fault-tolerance mechanisms that exploit these benign failure semantics (Section 8.7).

As we discuss the specific mechanisms of our communication subsystem, you may also want to refer to Appendix B, which contains a pseudocode reference implementation of the main components of the communication subsystem that we will design in this chapter.

8.1 Requirements and Other Preliminaries

Let us begin by stating our requirements, our non-requirements, some guiding design principles, and some preliminary design choices.

8.1.1 The Requirements

We stated our thesis at the very beginning (Section 1.3 on page 10). As a reminder, here it is again:

> We can add tolerance to coincident permanent and transient faults to the FTT communication paradigm, using Ethernet as the underlying technology, while maintaining the paradigm's main features: support for the timely exchange of both periodic and sporadic real-time messages, and support for updating the real-time parameters of these messages at runtime.

To prove the thesis, we must design a communication subsystem whose service is to

(S1) consistently transport real-time messages, which may be sporadic or periodic, from a sender to the appropriate receivers,

(S2) doing so without violating any deadlines and

(S3) while allowing the real-time parameters of the messages to change at runtime.

For easier reference, let us refer to the composite service comprised of the subservices S1, S2, and S3 as **FTT service**.

Besides providing an FTT service, the communication subsystem must also satisfy the following requirements, which are directly derived from the thesis statement:

R1: The communication subsystem is based on the FTT paradigm.

R2: The communication subsystem is based on Ethernet.

R3: The communication subsystem provides an FTT service even if both permanent and transient faults occur.

R4: The communication subsystem can be implemented in practice.

To satisfy requirements R1 and R2 we will design the communication subsystem using HaRTES as a building block. The last requirement, R4, ensures that fault tolerance can indeed be added to Ethernet-based FTT; we will deal with it through prototyping and experimentation, which we shall postpone until after we have a design (Chapter 9). Requirement R3 merits further discussion.

We want our communication subsystem to provide the FTT service despite permanent and transient faults. However, no system can be made to provide its service indefinitely. All systems fail at some point. Thus, we must be clear about what exactly our system should tolerate. This leads us to additional requirements, which qualify requirement R3 and which I have chosen because I expect them to be necessary for our communication subsystem to be more reliable than previous Ethernet-based versions of FTT. The additional requirements are these:

R3.1: The failure of a slave node does not disrupt the FTT service provided to other slave nodes.

R3.2: The maximum duration of transient faults that the communication subsystem can tolerate can be parameterized.

R3.3: The communication subsystem provides an FTT service even if any one of its components, no matter which one, suffers a permanent fault.

As to requirement R3.1, remember that we are working at the level of the communication subsystem. We are thus agnostic towards any particular application running on the nodes. Hence, whether a particular node is important, or even critical, is out of our scope. We will therefore not try to tolerate node failures. Nevertheless, we should not permit a node failure to disrupt the communication subsystem, which *is* within our scope. Requirement R3.1 forces us to add proper error containment.

R3.2 ensures that our communication subsystem can tolerate all transient faults, as long as their maximum duration is upper bounded and known. With known bounds, it is just a matter of properly parameterizing the system to tolerate transient faults.

Lastly, by meeting R3.3 we prevent the permanent failure of a single network component (link or switch) from disrupting the *FTT service*. In addition, since the application running on top of the communication subsystem presumably requires an FTT service, we also prevent the permanent failure of a single network component from disrupting the *application*. Thus, by satisfying requirement R3.3 we ensure that the permanent failure of a network component cannot disrupt the service of the whole system that includes the communication subsystem and the application on top of it. Hence, we eliminate all single points of failure from the communication subsystem, which is crucial for high reliability. As to why I have made it a requirement to tolerate a permanent fault in only one network component, and not more, the reason is to keep the complexity and cost down. In fact, precisely because of the high cost of tolerating multiple permanent faults, many, if not most, fault-tolerant systems are designed to only tolerate one permanent fault. Nevertheless, as we will see by the end of this chapter (Section 8.8), our communication subsystem will also be able to tolerate multiple component failures in many scenarios.

To satisfy R3.1, R3.2, and R3.3, when the communication subsystem tolerates faults, the activation of fault-tolerance mechanisms must not interrupt the FTT service. Thus, fault-tolerance mechanisms must not provoke deadline misses that would not occur in the absence of faults (which would be a violation of subservice S2). For instance, if we know that a message from each of the streams a, b, and c must be exchanged before deadlines D_a, D_b, and D_c, and this can be done in the absence of faults, then a message from each of those streams must also be exchanged before deadlines D_a, D_b, and D_c when one or more faults are tolerated. One implication of this is that our fault-tolerance mechanisms must not introduce temporal uncertainty: the maximum time for message exchanges must be predictable and the overhead for tolerating faults must be upper bounded.

Finally, all requirements must be satisfied at the same time. Thus, the communication subsystem must provide the FTT service even if any one of its components fails, at the same time some nodes fail, and transient faults occur for which the system has been parameterized. For this reason, we cannot rely on the redundancy used to tolerate permanent faults to also tolerate transient faults. The two types of redundancy must be orthogonal. Similarly, the error-containment mechanisms for the nodes must not interfere with the simultaneous tolerance of transient and permanent faults.

Now, before we embark on designing a communication subsystem that satisfies our requirements, let us also be explicit about some requirements that our system will not need to satisfy.

8.1.2 Non-requirements

The first non-requirement is the following:

NR1: We do not need to keep compatibility with non-FTT compliant legacy nodes.

A HaRTES switch is designed to provide its service even if non-FTT nodes, such as ordinary laptops, are connected to it. Our system does not need to support such nodes.[1]

NR2: We do not need to keep backward compatibility with HaRTES.

For instance, if we take a HaRTES slave and plug it into our fault-tolerant communication subsystem, we do not need to ensure that the slave is able to communicate. A slave to work in our system may require different software and hardware.

NR3: Message streams that are schedulable in HaRTES do not need to be schedulable in our system.

This non-requirement allows our communication subsystem to consume more time for message exchanges than HaRTES, which enables us to use temporal redundancy (message retransmissions) to tolerate transient faults.

Without this non-requirement we would have to rely on spatial redundancy to tolerate transient faults. This, however, is undesirable. Spatial redundancy is the only way to tolerate permanent faults. But since we must also tolerate simultaneously occurring transient faults, we cannot use the same spatial redundancy we are using for permanent faults to also tolerate transient faults. This means that we would need *additional* spatial redundancy just for transient faults, which is expensive. Thanks to non-requirement NR3 we can use cheaper temporal redundancy.

NR4: We do not need to rely solely on commercial off-the-shelf components. We can use custom hardware.

We inherited this non-requirement from HaRTES, which already used custom hardware.

[1] Nevertheless, as we will see in the next chapter (Section 9.6), our communication subsystem will be able to work with legacy nodes as long as these remain non-faulty.

NR5: We do not need to tolerate the failure of slaves.

This non-requirement has been anticipated earlier (when we talked about requirement R3.1). Our focus is the communication subsystem. Whether particular slaves are single points of failure or not depends on the application and we want to be application agnostic.

NR6: We do not need to support FTT messages with large payloads.

That is, we can assume that each FTT message fits into a single Ethernet frame and is not fragmented and conveyed in distinct frames.

NR7: We do not need to support plug-and-play.

A person may need to configure the system before it is put into operation and we do not need to support the addition of new hardware components at runtime. Thus, we can assume that the system parameters in the SCSR—the system configuration and status record (Section 7.1)—are fixed and preconfigured, and that the number of switches, links, and slaves does not change during the lifetime of the system. (The latter is a restriction with respect to HaRTES and FTT-SE, both of which allow adding new slaves at runtime.)

8.1.3 Our Guiding Design Principles

To make sure that we will satisfy our requirements, we should adopt some design principles:

DP1: Minimize temporal uncertainty.

Temporal uncertainty threatens providing an FTT service. In particular, it threatens transporting real-time messages without violating their deadlines (which corresponds to subservice S2 on page 148). We should therefore design our fault-tolerant communication subsystem to minimize it.

DP2: Use error compensation over rollback, rollforward, and fault handling.

This is in line with design principle DP1. Using error compensation will ensure that our fault-tolerance mechanisms do not delay the successful exchange of messages and provoke deadline misses. That is, with error compensation it takes the same amount of time to exchange messages in the presence as in the absence of faults, which makes it easier to satisfy requirement R3. In particular, it makes it easier to provide a proper FTT service without violating any deadlines (subservice S2).

DP3: Use seamless replication techniques, such as active replication, over other types of replication.

This also follows from design principle DP1. If we use replication techniques with zero failover time, we once again do not have to worry about messages missing deadlines when faults are tolerated. The mechanism to implement design principle DP3 will be error compensation, i.e., design principle DP2.

DP4: Keep the masters within the switches.

This is one of the key advantages of HaRTES over other versions of FTT (Section 7.2.3 on page 137). It grants a switch direct access to the SRDB and elementary cycle schedules, enabling it to drop non-FTT-conforming traffic. This allows enhanced error containment, which is important to satisfy requirement R3.1.

DP5: Restrict the failure semantics of the network components to benign failure modes.

The closer we can get a faulty component to misbehave with the innermost failure mode of the inclusion hierarchy (Figure 2.9 on page 45), the easier it will be for the rest of the system to tolerate that component's failure. Restricting the failure semantics of components will thus help us satisfy requirements R3.1 and R3.3.

8.1.4 Preliminary Design Choices

System designers always have to make choices. The following is a list of some I have made to simplify our work:

- Differences in propagation and transmission time among links shall be negligible.

All links should have approximately the same length and operate at the same speed. As a case in point, we should not have some links operating at 100 Mbps, while others are operating at 10 Gbps; nor should we have some links spanning a few meters, while others span kilometers. This will make it easier to synchronize the elementary cycles among slaves and masters in our communication subsystem.

- All switches shall be store-and-forward switches; none should be cut-through switches.

Relying only on store-and-forward switches means that all switches receive frames in full, verify their frame check sequence, and only forward them if the verification passes. Thus, using store-and-forward switches will already provide significant error containment, without any further mechanisms, making it easier to meet requirements R3.1 and R3.3.

- There shall be no application running on top of a master.

This means that, contrary to what happens in some other versions of FTT, SRDB update requests will only originate from slaves, doing so as slave control messages. This simplifies the design because it reduces the possible sources that can generate changes in the real-time requirements.

Having defined our requirements we know what our communication subsystem has to accomplish. And with the additional non-requirements, design principles, and preliminary design choices we have narrowed down how to meet these requirements. But we cannot yet go ahead and design any concrete fault-tolerant architecture or fault-tolerance mechanisms. First we must define a fault model so that we are clear about what specific types of faults our system should tolerate.

8.2 The Fault Model

As we know, a fault model describes the faults we assume and the rate with which we assume their occurrence (Section 3.4 on page 57). A good way of defining a fault model is by taking the 31 combined fault classes that Avižienis et al. identified (Section 2.5.1 on page 41), choosing among them a subset to specifically deal with, and excluding the rest.

(Excluding some faults from a fault model does not mean that our system will necessarily fail when such an excluded fault occurs. It only means that the system is not specifically designed to tolerate them.)

Figure 8.1 highlights the faults that will be part of our fault model. The chosen ones are highlighted in black and correspond to the columns labeled 12–21. All

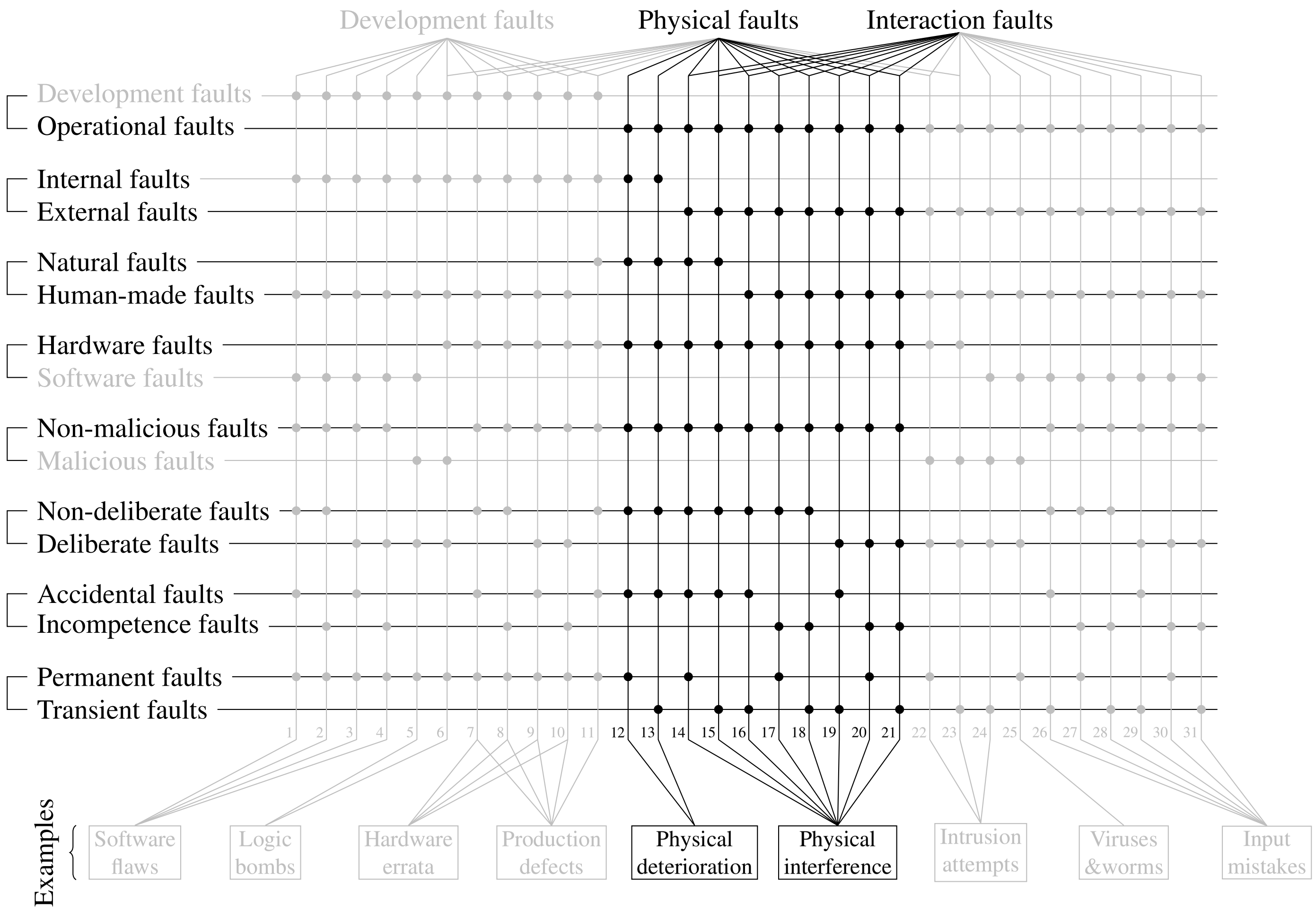


Figure 8.1: Fault model. (The figure is based on the figure of combined fault classes in Avižienis et al.[a] It differs in that I have highlighted the faults that belong to our fault model.)

[a] Algirdas Avižienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

other faults, the grayed out ones, will not be part of our fault model. As we can see on the left-hand side of the figure, we are excluding all development, software, and malicious faults. This is in line with the scope I defined at the very beginning (Section 1.4 on page 11). The faults that we *are* considering are non-malicious operational hardware faults. These may be internal or external; natural or human made; deliberate or non-deliberate; due to accidents or incompetence; and permanent or transient. Physical deterioration and interference, as shown at the bottom of the figure, are examples of these faults.

As to the rate with which faults occur, let us distinguish between permanent and transient faults.

For faults that persist permanently, let us not make any assumptions on how likely they are to occur. Coming up with accurate fault rates and distributions is difficult, and even more so during the design and prototyping stages. Moreover, our requirements (and thesis) say nothing about tolerating any particular rate with which permanent faults occur.

Regarding transient faults, which persist for only some time before they disappear on their own, let us not make any assumptions about their rate either. Except, that is, for transient faults affecting links. For these we should assume that their rate (and distribution in time) makes them detectable by the Ethernet frame check sequence. That way the corruption of frames due to link faults simply results in lost frames and not the acceptance of erroneous frames by correct receivers. This can greatly simplify our design, makes sure that using store-and-forward switches has a point, and we can benefit of the error containment that standard Ethernet controllers already provide. Luckily, assuming that all link faults are detected by the Ethernet frame check sequence is reasonable.

The frame check sequence used in the IEEE 802.3 standard is a 32-bit CRC polynomial that is guaranteed to detect up to 3 independent bit errors (Hamming distance of 4) in a maximum-sized Ethernet frame (12 144 bits, i.e., 1518 bytes)—if the frames are smaller than about 372 bytes (2975 bits, excluding the CRC field), the detection of even more independent bit errors is guaranteed.[2] Moreover, the Ethernet CRC is guaranteed to detect any burst errors affecting 32 or less bits and all odd numbers of bit inversions. There are other bit error patterns for which error detection is not guaranteed—but still extremely likely. According to Fujiwara, Kasami, and Lin,[3] for a binary symmetric channel, with independent bit-error rate

[2] Philip Koopman. "32-bit cyclic redundancy codes for Internet applications". In: *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2002)*. IEEE. 2002, pp. 459–468.

[3] Toru Fujiwara, Tadao Kasami, and Shu Lin. "Error detecting capabilities of the shortened Hamming codes adopted for error detection in IEEE standard 802.3". In: *IEEE Transactions on*

of up to 0.5, the probability of undetected errors in a frame is upper bounded by $2^{-32} \approx 2.328 \cdot 10^{-10}$ for any valid Ethernet frame length (512 bits, i.e., 64 bytes, up to 12 144 bits, i.e., 1518 bytes). That is, even if up to half the bits transmitted on a link can be affected by independent transient errors, the probability of a receiver accepting a corrupted frame is at most $2.328 \cdot 10^{-10}$. For more realistic bit error rates, and especially for the small frame sizes common in embedded applications, the probability of the Ethernet CRC failing to detect a corrupted frame is even lower. Thus, as stated a moment ago, assuming that all link faults are detected by the Ethernet frame check sequence is reasonable.

Having clarified our requirements and fault model, it is time to decide upon an architecture for our communication subsystem.

8.3 The Architecture

We are building on HaRTES and we know that it has as components slave nodes, links, and an FTT-enabled switch, i.e., a switch with a master within (Figure 7.8 on page 137). These are the initial building blocks for our system. Of these, two are single points of failure in HaRTES: the switch and the master it embeds. If they fail, the whole communication system fails. In contrast, the failure of slaves and links does not necessarily imply a global communication failure. First, they may fail with a benign failure mode, i.e., without disturbing the FTT service provided by the communication subsystem. Second, some slave nodes may be inessential for the given application and the overall system may accomplish its mission without them. Thus, to satisfy requirements R3.1 and R3.3, our architecture must replicate the switch and its master, whereas for slaves, error containment is enough since the need for slave replication is application dependent (and out of this dissertation's scope).

Now, we cannot assume failure independence between a switch and the master embedded in it—they are sharing resources and are subject to common mode and spatial proximity failures. Thus, the master replicas should not all be within the same switch, but be distributed among the switch replicas. That is, we should not have all masters embedded within one switch, and another switch with no embedded masters. So one decision is clear: we will have at least two switch replicas and each will have at least one master.

The next question is how we should connect the slaves to the replicated switches.

For all slaves to tolerate the failure of all but one switch, we require links from

Communications 37.9 (1989), pp. 986–989, Fig. 1.

each slave to each switch. That way, as long as there is one switch operating, all slaves can in principle still exchange messages.

As to the number of switch replicas, duplication should already give us a significant increase in reliability without the often prohibitive cost of higher levels of replication. Moreover, going beyond duplication will increase the complexity of our system without providing a clear advantage. The additional complexity may even decrease the reliability (it is harder to keep three or more replicas replica deterministic than two). So let us heed Gall's law and not introduce unnecessary complexity.[4]

Finally, to tolerate the failure of one replica out of a replica group, any replication scheme needs to ensure that surviving replicas can continue to provide a correct service when other replicas fail. For this we need to ensure replica determinism. Replica determinism usually requires some form of communication between the replicas. The only exception is when the replicas are internally deterministic and their inputs are guaranteed under all circumstances to be the same. This is not the case for the duplicated switches and masters: each slave has a dedicated link to each switch; hence, the switch replicas receive their inputs through different links and a fault may occur in one link and not the other. As a result, in the presence of faults it is impossible to guarantee that the two switches (and thus masters) receive identical inputs from each node. We therefore require a communication path between switches to resolve any potential nondeterminism. Moreover, this communication path should also be replicated. Otherwise, its failure would ultimately lead to a loss of replica determinism.

The above discussion leads us to the architecture shown in Figure 8.2. The replicated communication subsystem comprises two active HaRTES-based switches, each with its own master. The slaves are connected to both switches by means of **slave links**. Since these links are full duplex, we will find it convenient to sometimes distinguish between the two directions in which messages propagate. We can therefore think of each slave link as being constituted by two sublinks: a link from the slave to the switch, which we may call **uplink**, and a link from the switch to the slave, which we may call **downlink**. Figure 8.2 also shows several redundant links between the switches called **interlinks**. These serve as a communication channel between the masters, allowing them to exchange messages to achieve replica determinism. Moreover, if we use the interlinks to also forward

[4] Gall's law is a rule of thumb that advocates simplicity when designing a new system. It comes from Gall's 1977 book *Systemantics: How Systems Work and Especially How They Fail*, p. 52, where he wrote "A complex system that works is invariably found to have evolved from a simple system that worked. The parallel proposition also appears to be true: a complex system designed from scratch never works and cannot be made to work."

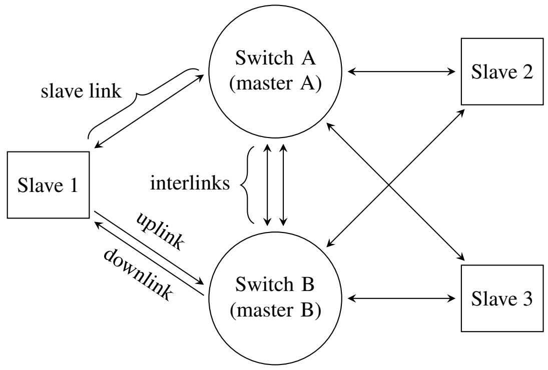


Figure 8.2: Preliminary FTTRS architecture.

slave messages from one switch to another, we benefit from a further redundant path between any pair of slaves. This reduces the probability of the slaves being partitioned into subsets that cannot communicate with each other.

The architecture of Figure 8.2 is then our preliminary architecture. It is a replicated star topology and since we will put flexible time-triggered communication on top of it, I suggest we call our communication subsystem *flexible time-triggered replicated star*, or **FTTRS**, from here onwards.

Having settled on the architecture of Figure 8.2, we are certain that the main components of our communication subsystem will be slave nodes, HaRTES-based switches, and links. Now, to make this architecture fault tolerant we will have to design appropriate fault-tolerance mechanisms. And to properly design these mechanisms we must know how the components can fail. That is, we must know their failure semantics. We will describe these failure semantics shortly in our failure model (Section 8.5). But before we do that, we must first know what a component is supposed to do when it is correct. Otherwise there is no way of telling whether a component is behaving correctly or not.

8.4 The Function of Each Component

A failure of a component is a deviation of the component's service from the component's function—or, as we put it earlier (Section 2.3), the departure of the service from a yellow brick road that represents the component's function. If we want to design a system that tolerates such failures, we must know when a component's service is correct—within the yellow brick road—and when it is faulty—outside the yellow brick road. For this we need to know where the yellow brick road lies.

In other words, we need to know the component's function. Once we know that, we can define a first failure model (Section 8.5) that describes how components can misbehave when they deviate from their function (the road) and then begin to add mechanisms to restrict these misbehaviors (Section 8.6).

To define the function of a component we should treat it as a black box and only look at its external behavior. After all, as long as its external behavior corresponds to what we want it to be, the component implements its function and what precisely it does internally is secondary (as an example, an analog and a digital watch provide the same function, i.e., showing the time of day, but internally work quite differently). Indeed, using a black box approach is common to define the function of systems and then prove their correctness.[5] Moreover, as we defined it (Section 2.3), the function of a system is a subset of its *external* states over time, and not of its total or internal states. Let us thus define the function of our components in terms of how they interact with other components, without worrying about what happens inside of them.

In FTTRS, as in any message passing system, the interaction consists of an exchange (or the lack of an exchange) of messages. Hence, the idea is to define the function of each component in terms of when and what messages it should transmit, and what messages it should omit. For this we can define component properties that we may call **correctness properties**.

Some of these properties will be somewhat vague. This is so because there is a circular dependency between a failure model and the design of a fault-tolerant system: to design a fault-tolerant system we need a failure model, and to define a failure model we must know the function of each component of the system, but the exact function of each component depends on the system's design, which again depends on the failure model. Luckily, this is not much of a problem. Our goal is not a formal verification. It is only to get a sense of how each component should behave when it is correct so that we can define a failure model and then continue designing FTTRS to deal with this failure model.

8.4.1 The Function of a Slave

The function of a slave is mostly determined by the FTT paradigm. It is the paradigm that tells us when and what message transmissions correspond to a well-behaved slave. Specifically, the paradigm determines the following correctness properties

[5] For instance, the black box approach is used when input/output automata are used to prove the correctness of a message passing system (Nancy A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996, p. 216). It is also used in so-called black box conformance testing.

for a slave s:

SF1: *Timely polling response:* if s receives a trigger message tm, it responds in time by transmitting the scheduled synchronous messages for which s is the publisher.

SF2: *No transmission of unscheduled synchronous messages*: the slave only transmits synchronous messages if it was polled. If s was not polled by a trigger message, then s does not transmit any synchronous message during the corresponding elementary cycle.

SF3: *Rate-constrained asynchronous transmissions:* the transmission by s of any two asynchronous messages am_i and am_j belonging to the same asynchronous stream are separated by the minimum interarrival time specified for that stream.

SF4: *Timely NRDB update:* if s receives commands from a master to update its NRDB, then subsequent transmissions by s reflect these updates (e.g., if the update involved making s the publisher of a given stream, then s will begin transmitting synchronous messages belonging to that stream when polled).

SF5: *Transmission of valid metadata:* if s sends a message, that message has FTT-compliant metadata and valid Ethernet metadata. An example of FTT compliant metadata would be a stream identifier, publisher, and set of subscribers that match one of the streams that have been negotiated between the slaves and the masters. An example of valid Ethernet metadata would be an Ethertype that always indicates that the payload is an FTT message.

(We do not have a property that requires the transmission of valid data—as opposed to metadata—because the data is application-dependent and we want to remain agnostic with respect to any particular application.)

Besides these FTT-derived properties, there are others that we can foresee that are related to fault-tolerance. As we know, fault tolerance relies on redundancy. Hence, a slave must correctly use the redundancy enabled by FTTRS to be correct. Even though we have not yet defined any specific fault-tolerance mechanisms, we know that with the FTTRS architecture a slave will have to rely on two main types of redundancy to tolerate simultaneous permanent and transient faults: spatially-redundant (parallel) transmissions through its links and temporally-redundant (re)transmissions. Therefore, when a slave s uses such redundancy, it must satisfy the following two additional correctness properties to be non-faulty:

SF6: *Spatially-consistent transmissions:* if s transmits a message through both its slave links, the message replicas are identical (or differ only in well-defined ways, such as in the source MAC address).

SF7: *Temporally-consistent transmissions:* if s transmits the same message several times, the message replicas are identical (or, again, differ only in well-defined ways, such as in sequence numbers).

8.4.2 The Function of a Link

The function of a link is simple:

LF1: *Message propagation:* the link propagates messages from the transmitting endpoint to the receiving endpoint.

LF2: *No message alteration:* the link does not alter any messages that it propagates.

8.4.3 The Function of an FTTRS Switch

The function of an FTTRS switch can be expressed as two separate classes of correctness properties: one set of properties related to the switching and forwarding of messages; and one set of properties related to the master functionality implemented by the switch. For this reason, it is best to think of an FTTRS switch as having two separate components: one that performs the switching, which we may call **internal switch**, and another that performs the FTT master functionality, which we may call **embedded master**. This is a subdivision similar to the one used in a HaRTES switch (in Figure 7.9 on page 139 the embedded master would correspond to the shaded area and everything else to the internal switch). The internal switch implements the switching circuitry, checks the frame check sequence of incoming Ethernet frames, performs a validation process, manages the different message queues, and performs traffic shaping. The embedded master, in contrast, computes the elementary cycle schedules, subjects SRDB update requests to an admission control, generates master command and trigger messages, and broadcasts these messages at appropriate times.

Let us then examine the correctness properties of an internal switch and an embedded master.

The Function of an Internal Switch

The switching-related properties for an FTTRS switch w include the following:

FF1: *Subscriber-based forwarding:* if w receives a message msg, then it forwards msg to those slave links to which subscribers of msg are attached.

FF2: *Timely forwarding of messages:* if w receives a timely message msg on any of its slave links or interlinks, it forwards msg in time.

FF3: *No message alteration:* the switch w does not alter any messages that it forwards.

FF4: *Dropping of untimely FTT messages:* if w receives a synchronous message that was not scheduled, or an asynchronous message that violates its minimum interarrival time, then w does not forward the message.

FF5: *Dropping of corrupted Ethernet frames:* if w receives a message conveyed in a corrupted Ethernet frame, then w does not forward the frame to any link.

The above properties are all derived from HaRTES, although in FTTRS we may implement them differently. The last one, in particular, may differ: an FTTRS switch, to improve the containment of errors, may be more strict about what message contents should be deemed valid (e.g., it may not suffice to only check the Ethernet frame check sequence).

Finally, we have one more property that is FTTRS specific:

FF6: *Interlink forwarding:* the switch w forwards every message it receives from a slave link through all its interlinks.

This last property ensures that the switches take full advantage of the FTTRS architecture to maximize the number of paths through which any pair of slaves communicate. For instance, referring back to Figure 8.2 on page 159, messages from slave 1 can reach slave 2 not only through single-hop paths (from slave 1, to switch A, to slave 2; and from slave 1, to switch B, to slave 2), but also through double-hop paths (from slave 1, to switch A, to switch B, to slave 2; and from slave 1, to switch B, to switch A, to slave 2).[6]

Now onto the master-related function of a switch.

[6] For this to properly work, the schedulers within the embedded masters must be aware of the additional delay of double-hop paths while computing the schedules. Since scheduling is out of the scope of this thesis, I will simply assume that the schedulers properly take into account double-hop paths.

The Function of an Embedded Master

The FTT paradigm determines that masters must periodically broadcast trigger messages and respond to update requests. This gives us the following correctness properties for an embedded master m:

MF1: *Periodic trigger message broadcast:* the master m broadcasts a trigger message every LEC units of time (with some precision), where LEC is the duration of an elementary cycle.

MF2: *Spatially-consistent transmissions:* if w transmits a message through multiple links, the message replicas are identical (or differ only in well-defined ways, such as in the source MAC address).

MF3: *Eventual master response:* if m receives an update request from a slave, then it will eventually broadcast a response to each slave.[7]

The fact that we have two masters in FTTRS and that we want to rely on active replication (design principle DP3) gives us the following additional correctness properties:

MF4: *Consistent master transmissions:* if a master m_j in one switch sends a control message to a slave, then the master m_i in the other switch transmits the same (or at least equivalent) message to the same slave.

MF5: *Synchronized master transmissions:* if a master m_j in one switch sends a control message to a slave whose timing is critical, then the master m_i in the other switch sends the same (or at least equivalent) message at approximately the same time.

These last two properties are particularly relevant for the trigger message: for the active replication to properly work, the trigger messages must be transmitted consistently and in a synchronized manner.

Finally, our design will almost surely require a master to retransmit messages. In that case we have an additional correctness property:

[7] HaRTES does not satisfy this property for rejected update requests: when a HaRTES switch rejects a slave's request, it never answers. This is a flaw according to Inés Álvarez et al. "A First Performance Analysis of the Admission Control in the HaRTES Ethernet Switch". In: *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*. IEEE. 2016.

MF6: *Temporally-consistent transmissions:* if the master m_i transmits the same message several times, the message replicas are identical (or, once again, only differ in well-defined ways).

The above concludes our list of correctness properties. Our next task is to think about how the FTTRS components, given our fault model, may violate these properties, which will tell us what failure semantics we should assume.

8.5 The Unrestricted Failure Model

In the previous section we defined the function of each type of component. Now let us see if and how components may reasonably deviate from that function given our fault model. That is, let us specify for each type of component its failure semantics. Together, these failure semantics will give us a first failure model for FTTRS, which we may call an **unrestricted failure model**. (The restricted failure model will be the one we obtain in Section 8.6 once we restrict the failure semantics of components.)

Let us go through each type of component again, this time with an eye to how they may fail.

8.5.1 The Failure Semantics of a Slave

With our fault model, unless we do something about it, we should expect a slave to have byzantine failure semantics. The reason is the following.

From the correctness properties of a slave (Section 8.4.1) it is clear that the function of a slave has much to do with proper timing. In particular, if any of the properties SF1 (timely polling response), SF3 (rate-constrained asynchronous transmissions), and SF4 (timely NRDB updates) is violated, we have a timing failure. Moreover, if a scheduled synchronous message is sent in the wrong elementary cycle, this is a violation of property SF2 (no transmission of unscheduled synchronous messages) and also a timing failure. Unfortunately, all these properties can be violated with our fault model, i.e., with non-malicious hardware faults. A simple example is the deterioration of the oscillator that drives the CPU of the slave. We therefore must place the failure semantics of a slave at least at the level of the timing failure mode of the inclusion hierarchy (Figure 2.9, page 45).

A non-malicious hardware fault, however, can also lead to a value deviation in an FTT message (Figure 8.3 shows what subset of the inclusion hierarchy corresponds

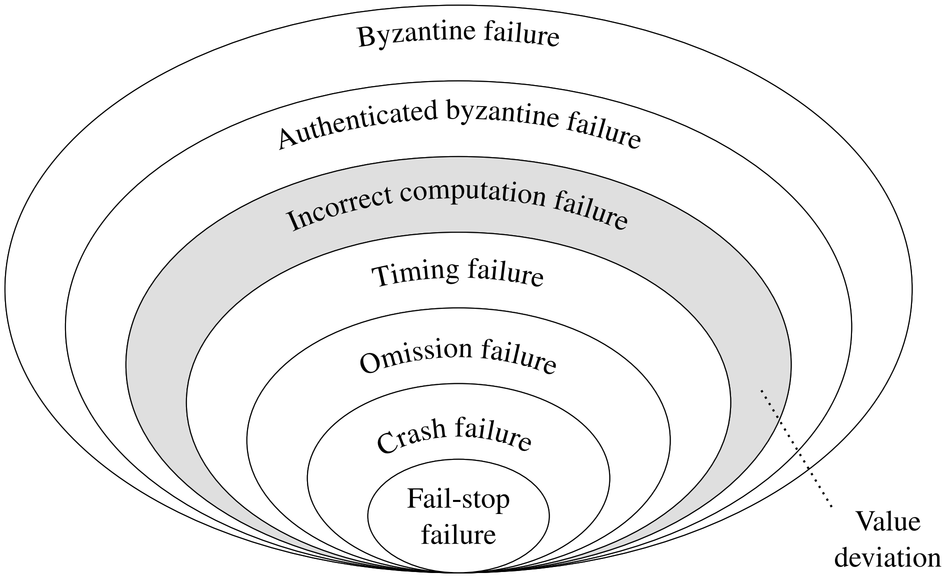


Figure 8.3: Location of the value deviation failure mode. Contrary to incorrect computation failures, value deviations do not include timing failures.

to a value deviation, namely, the subset of incorrect computation that excludes timing failures). For instance, a bit flip in the NRDB may lead a slave to send a message with an incorrect stream identifier, violating correctness property SF5 (transmission of valid metadata). This raises the failure semantics of a slave from the timing failure mode to the incorrect computation failure mode (which includes value deviations and timing failures).

An incorrect computation failure mode, however, is not the worst possible behavior for a slave either. The requirement for spatially-consistent transmissions (correctness property SF6) leads to potential two-faced behaviors in the presence of faults. For instance, a bit flip may affect the transmission controller of one slave link, but not the other. Moreover, bit flips may lead a slave to transmit messages with a sender ID, or a source MAC address, corresponding to another slave, and this would correspond to an impersonation. Given the potential for two-faced behaviors and impersonations, we have to put the slave failure semantics squarely at the top of the inclusion hierarchy. That is, we should expect slaves to be byzantine.

8.5.2 The Failure Semantics of a Link

We should expect a link to have value-deviation/omission failure semantics. The justification is this. The function of a link is to propagate messages, doing so without modification. In the presence of non-malicious hardware faults, however, propagated messages might be altered (e.g., due to electromagnetic interferences, which would correspond to a value deviation). Moreover, message propagation may be interrupted permanently (e.g., if the link is shorted, which would correspond to a crash) or temporarily (e.g., if the link has a loose connector at one end, which would be an omission). On the other hand, a link cannot present two-faced behaviors nor impersonate other components; and neither can it significantly delay the propagation of messages (temperature variations might affect the conductivity of the cable's wires, but this should have a negligible effect on propagation time). Byzantine behaviors and timing deviations are therefore excluded from the failure semantics of a link, which leaves links with value-deviation/omission failure semantics.

8.5.3 The Failure Semantics of an FTTRS Switch

For an FTTRS switch, as before (Section 8.4.3), let us differentiate between the part corresponding to the switching (the internal switch) and the part corresponding to the master (the embedded master).

The Failure Semantics of an Internal Switch

We should consider the failure semantics of an internal switch to be byzantine. Why? Because it is plausible under our fault model for an internal switch to present two-faced behaviors (or n-faced, where n is the number of ports). One way this could happen is when non-malicious hardware faults prevent the correct forwarding of messages (violating correctness property FF1 on page 163, i.e., subscriber-based forwarding). For instance, imagine a given message should be forwarded to ports p_1 and p_2. To accomplish this, a HaRTES switch (and thus a similarly implemented FTTRS switch) has queues of pointers to a memory pool shared by all ports where the messages to be forwarded are stored.[8] A bit flip in one of these pointers could cause the internal switch to transmit different messages through p_1 and p_2, when the correct behavior would have been to forward the same message. Another way in which a two-faced behavior could occur is when a given message is timely

[8] Santos, "Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications", p. 66.

forwarded through one port, but not another. In that case the service provided through both ports would differ when it should be the same.

Non-malicious hardware faults may also cause two-faced behaviors in another way. Again, imagine that the same message msg should be forwarded through ports p_1 and p_2. The head of the pointer queue of p_1 points to msg and subsequently msg is forwarded through p_1. So far so good. But then a bit flip alters msg within the memory pool, turning it into msg'. Next, the head of the pointer queue of p_2 points to the message, which now is msg'. Once p_2 completes the forwarding, it will have forwarded msg' instead of msg.

The above justifies why a violation of correctness property FF1 (subscriber-based forwarding) can plausibly lead to two-faced behaviors, which justifies why we should consider internal switches to be byzantine. For completeness, let us also take a quick look at the other properties to see some ways in which they might be violated.

Considering the strength of the Ethernet CRC, it seems unlikely that a non-malicious hardware fault may lead a switch to accept a corrupted Ethernet frame and violate correctness property FF5 (dropping of corrupted Ethernet frames). In contrast, there are plausible ways in which the other correctness properties can be violated with our fault model. Failing to drop untimely messages (a violation of property FF4) could occur upon a bit flip in the SRDB that modifies a real-time parameter of a stream; a failure to forward messages in time (a violation of property FF2, i.e., timely forwarding of messages) could occur due to the degradation of the oscillator used by the switch; alteration of messages (violation of property FF3, i.e., no message alteration) could occur upon bit flips in the memory pool; and failing to forward a message through all interlinks (violation of property FF6, i.e., interlink forwarding) could occur when bit flips corrupt the pointers of the queues in the interlink ports.

The Failure Semantics of an Embedded Master

Finally, what failure semantics should we assume for an embedded master? Since an embedded master must provide the same service to multiple users (property MF2 on page 164, spatially consistent transmissions), it could exhibit two-faced (or n-faced) behaviors and thus be byzantine. And indeed, with our fault model the probability of two-faced behaviors is not negligible. In a HaRTES switch (and thus in an FTTRS switch) an embedded master has direct access to the output ports (labeled “port dispatchers” in Figure 7.9 on page 139). Thus, when it broadcasts a trigger message to all slaves, a bit flip may affect the trigger message when it

Table 8.1: Unrestricted failure model of FTTRS.

Type of component	Failure semantics
Slave	Byzantine
Link	Value-deviation/omission
Internal switch	Byzantine
Embedded master	Byzantine

is copied to one port, but not when it is copied to others. This would lead to the transmission of different trigger messages on different ports, which would constitute a two-faced behavior.

Regarding the other correctness properties of an embedded master, these are also likely to be violated in the presence of non-malicious hardware faults. For instance, the degradation of the oscillator used by the master may prevent the periodic broadcasts of trigger messages (violation of property MF1); bit flips might corrupt an update request stored in memory beyond recognition such that the master cannot formulate a response (violation of property MF3); consistency and synchronization among masters (properties MF4 and MF5) will certainly be violated, even in the absence of faults, until we add mechanisms for replica determinism enforcement (Section 8.7.4); and temporally-consistent transmissions (property MF6) can be violated when a bit flip acts on one temporal message replica and not another.

This concludes our first failure model. For reference, Table 8.1 provides a summary. Following design principle DP5 (page 153), next we have to restrict the failure semantics summarized in the table to more benign ones. That way the fault-tolerance mechanisms we will design afterwards (Section 8.7) will remain sufficiently simple to not become unreliable themselves.

8.6 Restricting the Failure Semantics

As we have seen (Section 8.5), with our fault model and architecture we should expect slaves, internal switches, and embedded masters to have byzantine failure semantics. This is bad. Such failure semantics are hard to deal with. Byzantine components may misbehave in tricky ways, making it exceedingly difficult to build a reliable system. For this reason, before we design concrete fault-tolerance mechanisms, we should first restrict the failure semantics of the FTTRS components to more benign ones.

Let us see how we can do that. We will begin with the switches and their embedded masters (Section 8.6.1) and then we will consider the slaves (Section 8.6.2). The links we will leave as they are since they are already sufficiently benign: link failures can only lead to a message being omitted, either because the link itself omits the message or because it corrupts the message, which then causes the message to be rejected at the recipient due to a CRC error.

8.6.1 Restricting the Failure Semantics of the FTTRS Switches

Let us restrict the failure semantics of FTTRS switches, which include their internal switches and embedded masters, beyond the omission failure mode. That is, let us not only prevent byzantine misbehaviors, but also incorrect timing deviations and omissions. Otherwise, our system will be what in distributed systems theory is called an **asynchronous system**: a system where no bounds exist on computing and communication delays. And in that case the famous **FLP impossibility result** applies, which proves that in an asynchronous system it is impossible to solve agreement problems with a deterministic algorithm.[9] We, however, will have to solve agreement problems (for instance, the switches will have to agree on the contents of their SRDBs) and we will want to do this with deterministic algorithms to facilitate replica determinism.

(Other ways around the FLP impossibility besides eliminating timing deviations and omissions include using randomized algorithms, failure detectors, and settling for approximate agreement.[10])

Our goal, therefore, is to restrict the failure semantics of FTTRS switches to a failure mode more benign than omission failures. In other words, we must ensure that switches have crash or fail-stop failure semantics. Luckily, we already know how to do this: internal duplication with comparison (Section 3.5.1 on page 62).

Indeed, if we internally duplicate a switch and then, as soon as the internal comparison detects a mismatch, the switch turns itself off, we get a switch with crash failure semantics. Such failure semantics are just about the most benign failure semantics we can wish for: a component either does what it is intended to do or it remains silent. Only a fail stop, where the failure of a component can be reliably detected by others, is more benign. As mentioned earlier (Section 3.5), if a component has crash failure semantics we say that it is fail silent.

We should note, however, that internal duplication with comparison also has its downsides.

[9] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[10] Lynch, *Distributed algorithms*, Ch. 21.

First, an internally duplicated and compared switch is more likely to fail (although in a more benign way) than one that is not duplicated. This is not only because it has more components, but also because internal duplication with comparison may convert transient faults into permanent ones, thus potentially wasting resources by causing unnecessary redundancy attrition. (If the redundancy attrition is a problem, this may be addressed in future work by designing appropriate mechanisms to reintegrate faulty switches.)

Furthermore, internal duplication with comparison requires hardware modifications to the components, can be costly, and can be difficult to implement. This is undesirable, but acceptable. After all, by using HaRTES-based switches we are already using custom hardware anyway and although the cost increase is significant, it is bearable for two FTTRS switches. Moreover, internal duplication with comparison has already been used for decades to simplify the design of fault-tolerant distributed real-time systems.[11] Even in the context of FTT and Ethernet we would not be the first ones to use it: FTT-CAN already used internal duplication with comparison for FTT masters[12] and at least one fail-silent Ethernet switch has been developed that uses the technique as well.[13]

8.6.2 Restricting the Failure Semantics of the Slaves

We argued for internally duplicating and comparing the switches to restrict their failure semantics. Now, what about the slaves? Here we should favor guardians (Section 3.5.2 on page 64).

First, internal duplication with comparison is costly. We made that point a moment ago for the switches, but disregarded it. We can justify that because we only have two switches. The number of nodes, however, could be significantly higher than two. So for them it is harder to justify using internal duplication with comparison, even if we repeatedly emphasize that cost is a minor concern for us.

Second, we should avoid requiring non-standard hardware for the slaves. After all, we already sacrificed using commercial off-the-shelf switches. If we now require slaves to also use non-standard hardware, we are throwing away much of the

[11] Examples include Johannes Reisinger and Andreas Steininger. “The Design of a Fail-Silent Processing Node for the Predictable Hard Real-Time System MARS”. in: *Distributed Systems Engineering* 1.2 (1993), p. 104; Powell, “Distributed Fault Tolerance: Lessons from Delta-4”; Proenza, “RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance”.

[12] Ferreira et al., “Achieving Fault Tolerance in FTT-CAN”; Ferreira, “Fault-Tolerance in Flexible Real-Time Communication Systems”.

[13] Stefan Poledna. “Method and Switching Unit for the Reliable Switching of Synchronization of Messages”. US Patent App. 14/391162. Mar. 2015.

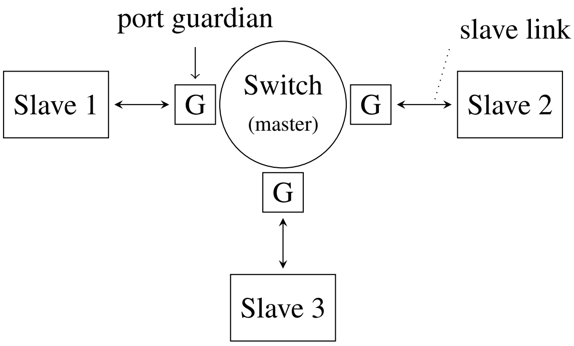


Figure 8.4: Architecture of HaRTES/PG.

advantage of using cheap, standardized, and widely available Ethernet components.

Finally, there is simply no need for internally duplicating and comparing the slaves. By having fail-silent switches, which occupy a privileged position within the network, these can sufficiently restrict the failure semantics of the slaves. Indeed, HaRTES switches already discarded untimely transmissions from slaves. Moreover, Ballesteros et al.[14] studied how to enhance HaRTES switches by enhancing them with **port guardians**: guardians that restrict the failure semantics of slaves further by policing the traffic arriving at the ports of a switch. The resulting architecture is called HaRTES/PG. We can use these port guardians for our own purposes, so let us examine HaRTES/PG in more detail. Afterwards, we will see how to apply these guardians, which were designed for a simplex star topology, to FTTRS, which has a duplicated star topology.

HaRTES/PG

Figure 8.4 shows the architecture of a **HaRTES/PG** network, a HaRTES network enhanced with port guardians. For the port guardians to work, all nodes must be FTT slaves. Legacy nodes are excluded. Moreover, the guardians must be fail silent.

The guardians, by being part of a switch that embeds an FTT master, have access to the SRDB and the elementary cycle schedules. This allows them to filter out any non-FTT compliant traffic. Specifically, the guardians do the following:[15]

[14] Alberto Ballesteros et al. “Towards Preventing Error Propagation in a Real-Time Ethernet Switch”. In: *Proceedings of the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2013)*. IEEE, 2013, I am a co-author of this paper, but the credit for this work should primarily go to Alberto Ballesteros.

[15] Ibid.

1. They eliminate impersonations by requiring a static assignment of MAC addresses to slaves and then checking that slaves transmit their messages with correct source addresses.
2. They drop trigger messages, master command messages, and any other master-exclusive messages coming from a slave link. That way slaves cannot impersonate masters.
3. They eliminate all FTT messages with incorrect destination addresses. To check if the destination address is correct, they consult the master's SRDB, which specifies who the subscribers of a particular FTT message are.
4. They drop any messages that, according to the SRDB, exceed the transmission time specified for the stream they belong to.
5. They drop synchronous messages that have not been scheduled. For this they check the current elementary cycle schedule.
6. They drop asynchronous messages that arrive with a frequency higher than allowed, which they check by consulting the minimum interarrival time specified in the ART of the SRDB.
7. In general, they drop all messages whose FTT or Ethernet metadata does not agree with what is specified in the SRDB.

With the first two actions listed above, the port guardians eliminate impersonations. With the third action, they eliminate two-faced behaviors: if a slave s_1 sends a message msg_A to a slave s_2 and subsequently another message $\mathrm{msg}_B \neq \mathrm{msg}_A$ to a slave s_3 when msg_A and msg_B should have been sent to both s_2 and s_3, then the guardian at the end of s_1's link would drop both messages msg_A and msg_B because they did not, as dictated by the SRDB, have a destination address matching both s_2 and s_3.

As a result of the first three actions, slaves in HaRTES/PG no longer present byzantine failure semantics. The fourth, fifth, and sixth actions eliminate incorrect timing deviations. The seventh action eliminates many, but not all, value deviations. Specifically, it does not eliminate deviations affecting the FTT message payload. (And unfortunately there is nothing HaRTES/PG can do about that. The payload is application dependent and determined by information local to a slave. For instance, the payload may be the value of a slave's sensor or the result of a local computation.)

The resulting failure semantics of a slave in HaRTES/PG is illustrated in Figure 8.5: the failure semantics is restricted to value deviations in payloads (payload errors) and omission failure semantics (which includes crash and fail-stop failures).

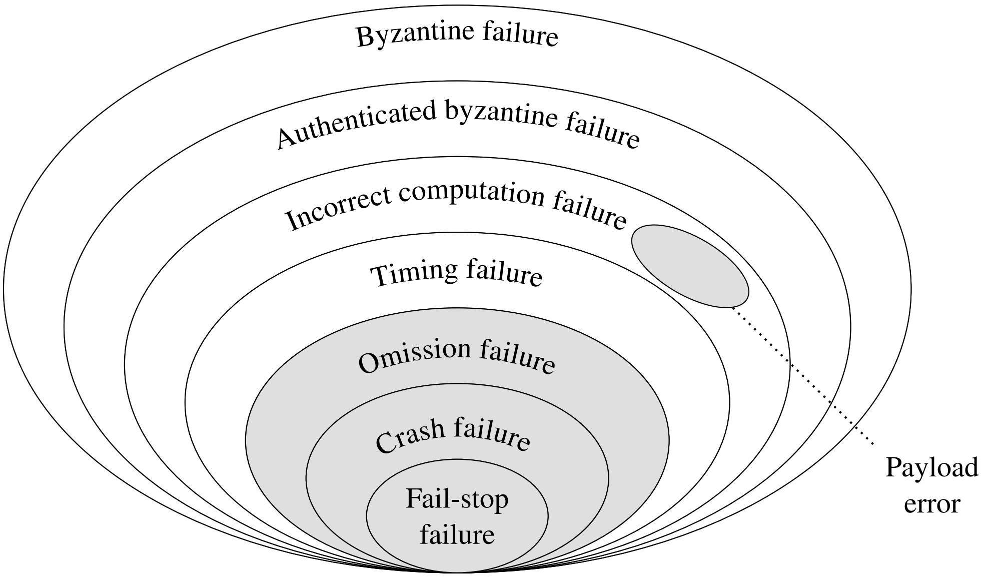


Figure 8.5: The shaded areas corresponds to the failure semantics of a slave in HaRTES/PG.

It would thus be accurate to refer to the failure semantics of a slave in HaRTES/PG as incorrect computation failure semantics since it is the innermost failure mode of the hierarchy that includes both payload errors and omission failures. But it would not be very precise. Better if we call them payload-error/omission failure semantics.

Are these failure semantics good enough for FTTRS? Let us see.

Payload errors in *data messages* (Figure 7.3 on page 126) are no problem for us who are designing the communication subsystem. Although they can affect the application executed by the slaves, they cannot affect the correct functioning of the communication subsystem itself. So this limitation is no reason to disregard the HaRTES/PG port guardians for FTTRS. (In any case, the application could deal with such errors by using, for instance, node replication.)

Payload errors in *slave request messages*, on the other hand, might be troublesome. Being unable to prevent payload errors means that the HaRTES/PG guardians cannot prevent slaves from sending incorrect update requests (unless the guardians drop all update requests from slaves, but this would eliminate all flexibility). This means that a slave can send a bogus update request and, if this update request passes the admission control, this request will update the SRDB and affect the scheduling of future elementary cycles.

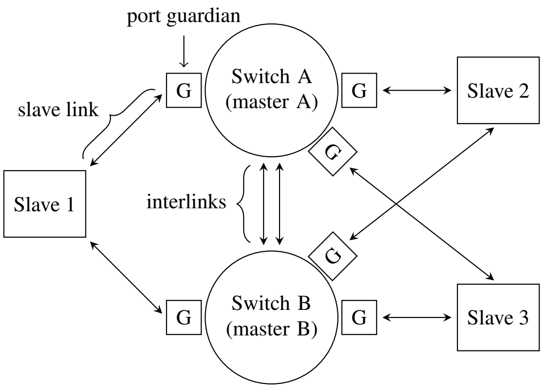


Figure 8.6: Final FTTRS architecture. We have added port guardians to restrict the failure semantics of the slaves.

Despite the risk for bogus slave requests, we should not discard the HaRTES/PG guardians for FTTRS. After all, the risk can be mitigated: FTTRS masters could only take into account update requests from trusted slaves (e.g., ones that are internally duplicated and compared); or the fail-silent masters could be the only ones that are allowed to request changes; or slaves could be replicated and update requests would then only be processed if a majority is requesting the same update. Which of the three solutions is best is application dependent. Thus, to continue with our agnosticism towards any particular application, let us only worry about keeping SRDBs and NRDBs consistent, but not about their specific contents nor the specific contents of update requests.

Adding Port Guardians to FTTRS

If we add the port guardians of HaRTES/PG to FTTRS, we obtain the architecture shown in Figure 8.6. Each switch now has a guardian for each slave link. For the interlinks, in contrast, there are no guardians because our guardians can only restrict the failure semantics of slaves. Moreover, switches are fail-silent anyway.

So what is the result? Do slaves in FTTRS now have non-byzantine failure semantics as in HaRTES/PG? Unfortunately no. The guardians were envisioned for the simplex (non-replicated) microsegmented topology of HaRTES (Figure 8.4). If we simply add them to our duplicated architecture (as in Figure 8.6), slaves have byzantine failure semantics again. Why? Because the HaRTES/PG guardians do not consider two-faced behaviors resulting from the transmission of different messages on different slave links, which can happen with our architecture and fault model.

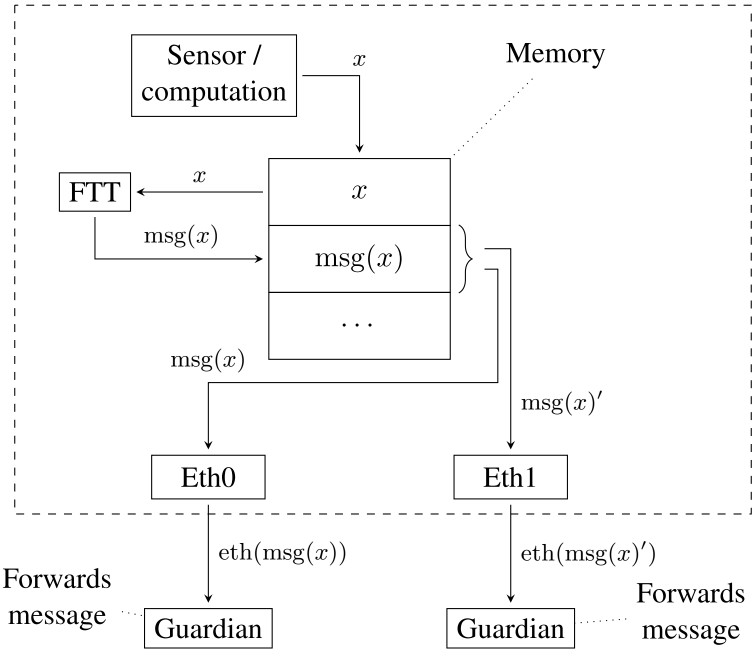


Figure 8.7: Potential two-faced behavior of FTTRS slaves.

Figure 8.7 shows how a two-faced behavior can occur. We have a slave node (dashed box), its internals, as well as the two port guardians (at the bottom of the figure) that restrict the slave's failure semantics. The slave has a payload x that it wants to transmit in an FTT message through both its links. This payload originated from a sensor or from a local computation. Either way, the payload is temporarily stored in memory and then encapsulated in an FTT message $\mathrm{msg}(x)$, which is again stored in memory. Next, the message is copied from memory to each of the two Ethernet controllers. However, the message is copied correctly only to one Ethernet controller (Eth0). The other (Eth1) receives a corrupted version $\mathrm{msg}(x)'$ of the message. This could happen due to a hardware fault causing a bit flip in one of the disjoint paths from the memory location to each of the Ethernet controllers. For instance, it could happen due to a bit flip occurring in an internal data bus. It may even happen due to a bit flip affecting the memory: the message $\mathrm{msg}(x)$ is copied to one Ethernet controller, then a bit flip occurs in the memory location where $\mathrm{msg}(x)$ is stored, $\mathrm{msg}(x)$ becoming $\mathrm{msg}(x)'$, and then the incorrect message $\mathrm{msg}(x)'$ is copied to the other Ethernet controller. Whatever the cause, one controller sends an Ethernet frame $\mathrm{eth}(\mathrm{msg}(x))$ containing the message $\mathrm{msg}(x)$, while the other sends an Ethernet frame $\mathrm{eth}(\mathrm{msg}(x)')$ containing the message $\mathrm{msg}(x)'$. Assuming that no FTT metadata was affected by the bit flip, but only the data portion of the FTT

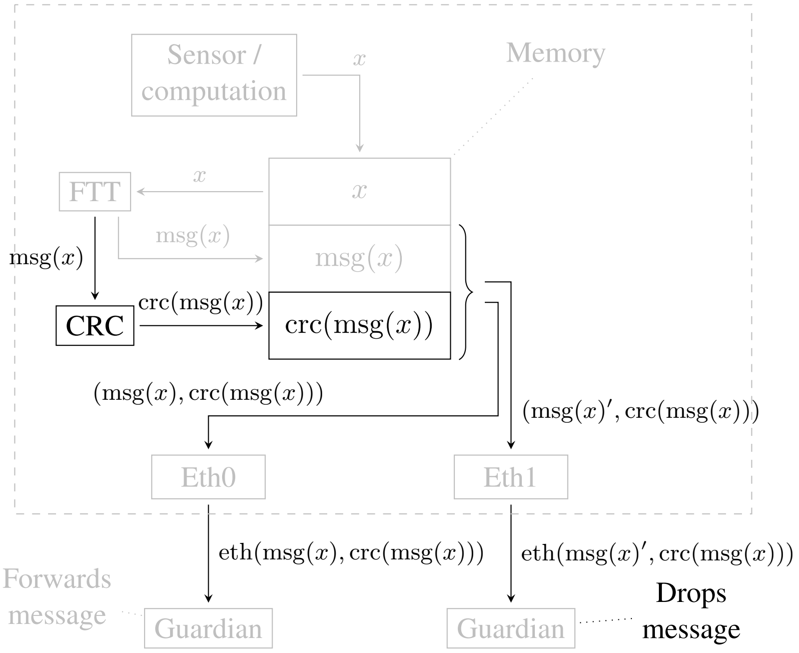


Figure 8.8: Adding a CRC to prevent two-faced behaviors of FTTRS slaves.

message, each port guardian determines the incoming Ethernet frame to carry an FTT-compliant message and forwards it. The Ethernet frame check sequence does not prevent this: each Ethernet controller calculates the frame check sequence *after* the bit flip occurred. Thus, with the hardware faults that are part of our fault model, slaves can present two-faced behaviors towards the switches and other slaves, even with the HaRTES/PG port guardians.

So what can we do? If we analyze the above scenario carefully, we will realize that the source of the two-faced behavior is an error that affects one transmission path (in the figure, the path from the memory to Eth1), but not the other (the path from the memory to Eth0). If only we could do something to make such errors detectable. Well, we can. All we have to do is compute an error detection code, such as a CRC, *before* the transmission path splits in two, then transmit the code through both links, together with the message, and have the guardians check that the message and the code match. This can be done completely in software—which has no bugs according to our fault model.

Figure 8.8 shows this schematically (the figure is a modification of Figure 8.7, with the parts that remain unchanged grayed out). When the payload x is encapsulated in an FTT message $\mathrm{msg}(x)$, that message is not only stored in memory, but also subjected to a CRC computation. The resulting error detection code, $\mathrm{crc}(\mathrm{msg}(x))$, is then stored alongside the message. Then, the tuple $(\mathrm{msg}(x), \mathrm{crc}(\mathrm{msg}(x)))$ becomes the Ethernet payload that is send to each Ethernet controller, and from there to each guardian. If the tuple in one of the disjoint paths is affected by an error, there will no longer be a match between the FTT message and its CRC. As an example, in the figure the FTT message $\mathrm{msg}(x)$ is corrupted on the right-hand path, yielding the tuple $(\mathrm{msg}(x)', \mathrm{crc}(\mathrm{msg}(x)))$. When this tuple reaches the right-hand guardian in an Ethernet frame, that guardian extracts $\mathrm{msg}(x)'$ and $\mathrm{crc}(\mathrm{msg}(x))$ from the Ethernet frame, calculates $\mathrm{crc}(\mathrm{msg}(x)')$, and detects that it does not match $\mathrm{crc}(\mathrm{msg}(x))$. Hence, it drops the message instead of forwarding it towards its switch.

Someone might object that we still have a two-faced behavior. Although we are no longer transmitting a message $\mathrm{msg}(x)$ past one guardian and a message $\mathrm{msg}(x)'$ past the other, we now have a successful transmission through one guardian and an omission through the other. And this does not correspond to an identical behavior towards both switches. That is true. We have an inconsistent omission, which, strictly speaking is a two-faced behavior. But it is much easier to deal with such an inconsistent omission than dealing with two diverging messages that should be the same. Moreover, this behavior is the best we can do with links that are not perfectly reliable: it is always possible for a transient fault to corrupt an Ethernet frame on one link, but not the other.

By adding the CRC mechanism, which is assumed to be implemented by fault-free software according to our fault model, the property of spatially consistent transmissions (SF6 on page 162) can no longer be arbitrarily violated past the port guardians. The only way the property can be violated is through omissions. Moreover, the same CRC mechanism also ensures that the property of temporally consistent transmissions (correctness property SF7) cannot be arbitrarily violated past the guardians either. Like the previous property, it can only be violated through omissions. This is so because our CRC mechanism works the same whether a message is read from memory twice to be transmitted through parallel links or whether it is read twice to be transmitted consecutively through the same link.

Figure 8.9 illustrates the resulting restricted failure semantics of a slave in FTTRS. Past the port guardians, the messages that slaves send through both their links can now only be incorrect in the following ways: one message is omitted, but the other is not (inconsistent omission); both messages have the same incorrect payload (payload error); or both messages are omitted (omission failure mode, which includes crash and fail-stop). Similarly, if a slave transmits the same message multiple times, the worst that can happen is that a subset of the temporal replicas do not get past a guardian. It is no longer possible for temporal replicas with different

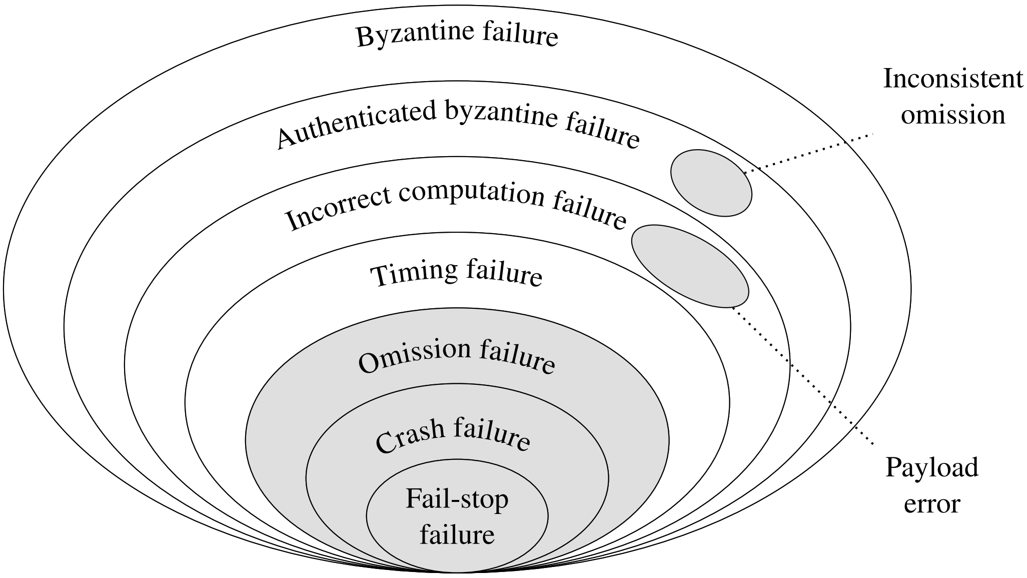


Figure 8.9: Restricted failure semantics of a slave in FTTRS.

contents to reach the switches or other slaves. Finally, a slave may cease to transmit messages altogether (e.g., if its power supply fails), which corresponds to a crash or fail-stop failure. If a slave stops transmitting through just one of its links, this again is an inconsistent omission—although a permanent one.

Table 8.2: Restricted failure model of FTTRS.

Type of component	Failure semantics
Link	Value-deviation/omission
Internal switch	Fail silent
Embedded master	Fail silent
Port guardian	Fail silent
Slave	Inconsistent-omission/ payload-error/omission

Table 8.3: The service that FTTRS must provide.

S1	Consistently transport real-time messages, which may be sporadic or periodic, from a sender to the appropriate receivers,
S2	doing so without violating any deadlines and
S3	while allowing the real-time parameters of the messages to change at runtime.

8.6.3 The Restricted Failure Model

With the mechanisms we have added to restrict the failure semantics of slaves and FTTRS switches, we now have a new failure model, which we may call a **restricted failure model**. Table 8.2 summarizes it.

The failure semantics of the links remain unchanged: when links fail, they can still only cause value deviations or (transient or permanent) omissions in the frames they propagate. Moreover, if they cause value deviations, these are always detectable by a correct receiver thanks to the Ethernet frame check sequence.

The FTTRS switches—and thus their internal switches, embedded masters, and guardians—are fail silent. Their internal duplication with comparison ensures that if a switch is operating, it is doing so correctly. And if it fails, it can only fail by remaining silent.

Finally, slaves can only exhibit well-defined misbehaviors towards other slaves and the switches: they can transmit messages with incorrect payload (which would have to be addressed at the application layer); they can successfully send a message through one link and omit it through the other; they can omit a message through both links; they can omit a subset of temporal message replicas; or they can remain permanently silent.

This concludes the restriction of the failure semantics. In line with design principle DP5 (Section 8.1.3, page 152), the components of FTTRS now all have benign failure semantics. The idea now is to design mechanisms that take advantage of these failure semantics to make the communication fault tolerant and thus satisfy requirement R3 (page 149)—all the way doing so using FTT and Ethernet as the underlying technologies to satisfy requirements R1 and R2.

Table 8.4: Requirements that FTTRS must satisfy.

R1	The communication subsystem is based on the FTT paradigm.
R2	The communication subsystem is based on Ethernet.
R3	The communication subsystem provides an FTT service even if both permanent and transient faults occur.
R3.1	The failure of a slave node does not disrupt the FTT service provided to other slave nodes.
R3.2	The maximum duration of transient faults that the communication subsystem can tolerate can be parameterized.
R3.3	The communication subsystem provides an FTT service even if any one of its components, no matter which one, suffers a permanent fault.
R4	The communication subsystem can be implemented in practice.

8.7 Fault-Tolerance Mechanisms for FTTRS

For easier reference, Tables 8.3 and 8.4 summarize the service and requirements we defined earlier (Section 8.1.1).

Since the switches used in FTTRS are based on HaRTES, we are so far on the right track to satisfy requirements R1 and R2. Our focus now is requirement R3. (Requirement R4 is still postponed until the next chapter.)

To satisfy R3, FTTRS must tolerate both permanent and transient faults. Specifically, to satisfy requirement R3.1, FTTRS must ensure that the failure of a slave does not disrupt the service provided to other slaves; to satisfy requirement R3.2, FTTRS must tolerate all transient faults that do not exceed a parameterizable duration; and to satisfy requirement R3.3, FTTRS must tolerate the permanent failure of any one of its components, whether it is a link or a switch (again, slaves are not part of the communication subsystem).

The plan is to meet these requirements by adding appropriate mechanisms to FTTRS. We will start with requirement R3.1 (Section 8.7.1) since we already provided a mechanism to satisfy it, namely, the port guardians. Then we will tackle requirement R3.2 by making messages fault tolerant in the presence of transient faults (Section 8.7.2). Since this will affect the structure of the elementary cycles we will also have to reconsider how the elementary cycles are synchronized across the network. After dealing with requirement R3.2, we will address requirement R3.3 (Section 8.7.3). In particular, we will see that for this last requirement we need the switches to be replica deterministic and we will thus discuss how to achieve this replica determinism (Section 8.7.4). In particular, we will see how the fault tolerant exchange of messages that helps us meet R3.2 will also help us achieve replica determinism by allowing the switch replicas to communicate reliably. Finally, we will finish discussing a few additional issues such as how to deal with an unexpected loss of switch replica determinism and how to have masters process more than one update request per elementary cycle (Section 8.7.5).

8.7.1 Preventing Faulty Slaves from Disrupting the FTT Service

As Table 8.3 shows, the service that FTTRS must provide is comprised of three subservices. Thus, to satisfy requirement R3.1 we must ensure that the failure of a slave

- does not prevent the consistent transport of real-time messages from a non-faulty slave to the appropriate receivers (subservice S1),
- does not cause the violation of deadlines corresponding to messages transmitted by other slaves (S2), and
- does not prevent other slaves from requesting changes to the real-time parameters nor masters from accepting and communicating these changes (S3).

Luckily, all three disruptions are already prevented under our fault model because of the restricted failure semantics enforced by the port guardians. First, general two-faced behaviors are prevented by the port guardians. This ensures that a faulty slave cannot provoke inconsistencies in the network, which would be deleterious to subservice S1. Second, any message with incorrect metadata sent by a slave is dropped at the corresponding guardian. Similarly, any untimely message is dropped as well. Thus, a faulty slave cannot send messages that unduly claim resources needed to transport other messages, which would be detrimental to subservice S2. For instance, a message cannot be sent at an incorrect time or be addressed to incorrect subscribers and thus steal bandwidth by being inappropriately forwarded. Furthermore, if the metadata is correct, but the payload is not, then the message will get past the guardians, but it will not prevent other slaves and the masters from communicating. This is so because if only the payload is incorrect, then the message must be timely and be addressed to correct recipients. As a result, the message cannot inappropriately block the transmission of other messages and thus cannot interfere with subservice S3. Finally, because software faults are excluded from our fault model, a faulty slave cannot transmit a message that causes the failure of a recipient, like it occurred with the infamous ping of death.[16]

[16] Malachi Kenney. *Ping of Death*. 1996. URL: http://insecure.org/sploits/ping-o-death.html (visited on 2016-12-16); MITRE Corporation. *CVE-1999-0128*. 2008. URL:

8.7.2 Tolerating Transient Faults

As I mentioned earlier (Section 8.6.1), given that the switches are internally duplicated and compared, they cannot suffer transient faults—any transient fault within a switch is converted into a permanent fault. Moreover, according to requirement R3.1, transient faults within slaves do not need to be tolerated—they only need to be prevented from disrupting the communication among other components. Thus, the only transient faults that need to be tolerated are those affecting links. And these, because of the failure semantics of links and the strong Ethernet CRC, can only lead to message omissions. Now, to satisfy requirement R3.2, what FTTRS must tolerate is all transient faults that do not exceed a parameterizable duration. If we know the maximum rate with which messages can be transmitted, the duration of a transient fault can be converted to a maximum number of messages that may be corrupted and, thus, omitted. Hence, to satisfy requirement R3.2, we can design FTTRS to tolerate a parameterizable number of message omissions.

So, how can we deal with message omissions? One way is to simply try, try, try again: if a message should be fault tolerant, transmit it repeatedly, k times, regardless of whether faults occurred or not. With a well-chosen k, a parameter that we may call the **redundancy level** of the given message, the probability of a transmission succeeding can then approach unity.

Some may object that this is wasteful. Why do we not adopt an approach where we only retransmit when necessary? Let us see.

The way to retransmit only when necessary is to use **automatic repeat request**, or **ARQ** for short, a backward error correction technique. As its name suggests, ARQ is based on a receiver automatically requesting failed transmissions to be repeated.[17] Although there are several variations of ARQ, it is generally based on a receiver sending acknowledgment messages (ACKs) back to the transmitter upon correct message reception and, when the message is not received correctly, requesting a retransmission. The retransmission request may be explicit—by having the receiver send non-acknowledgment messages (NACKs) back—or it may be implicit—by having the receiver remain silent, which then causes a timeout to expire at the transmitter that triggers a retransmission.

Many protocols use ARQ successfully (e.g., the Internet transmission control protocol, TCP, uses it). Why don't we? The main objection is that using ACKs or timeouts is inefficient in the worst case. With ARQ we cannot predict how long

https://www.cvedetails.com/cve/CVE-1999-0128/ (visited on 2016-12-16).

[17] Shu Lin, Daniel J. Costello Jr, and Michael J. Miller. "Automatic-Repeat-Request Error Control Schemes". In: *IEEE Communications Magazine* 22.12 (1984).

a successful transmission takes. At best it takes the time to get a message from the transmitter to the receiver, plus the time to get an ACK back. But it takes a different amount of time if the initial transmission or the ACK is lost, or if either one is lost and then the retransmission is lost, or if the second ACK is also lost, or if two successive retransmissions are lost. Since the time it takes to successfully get a message through is unpredictable, the scheduler would have to assume the worst case, that is, it would have to assume that the transmission of each message times out the maximum amount of times (which must be upper bounded). And if the scheduler has to take into account the worst case, it is more efficient to simply retransmit proactively and not waste bandwidth for ACKs or time waiting for a timer to expire. Thus, proactive retransmissions are more efficient for providing a fault tolerant FTT service.

Others have also recognized the advantages of proactive retransmissions and adopted it for their own real-time communication subsystems. Examples include the MARS approach,[18] TTP,[19] and the GOOSE protocol, which is specified in IEC standard 61850.[20]

Having settled on using proactive retransmissions, let us now discuss some particularities on how to implement this approach for messages sent by slaves and trigger messages sent by masters. (Master command messages will be made fault tolerant by piggybacking them on fault-tolerant trigger messages, as we will see later on in Section 8.7.4.)

Making Messages Sent by Slaves Fault Tolerant The scheduler of an embedded master needs to be aware of the redundancy level of each message sent by a slave, which may be a request message or data message (Figure 7.3, page 126). To make the scheduler aware, the easiest and most flexible approach is to store the redundancy level in the synchronous and asynchronous requirements tables.

When we covered these tables (Section 7.1 on page 124), we defined them as follows:

$$\begin{aligned}\text{SRT} &= \{\text{msg}_i \mid \text{msg}_i = (C_i, D_i, T_i, O_i, P_i), i \in [1, N_\text{S}]\}\\ \text{ART} &= \{\text{msg}_i \mid \text{msg}_i = (C_i, D_i, I_i, P_i), i \in [1, N_\text{A}]\}.\end{aligned}$$

[18] Hermann Kopetz et al. “Distributed Fault Tolerant Real-Time Systems: the MARS Approach”. In: *IEEE Micro* 9.1 (1989), pp. 25–40.

[19] Hermann Kopetz and Gunter Grunsteidl. “TTP — A Protocol for Fault-Tolerant Real-Time Systems”. In: *Computer* 27.1 (1994), pp. 14–23.

[20] International Electrotechnical Commission (IEC). *Communication networks and systems in substations — Part 8-1: Specific Communication Service Mapping (SCSM) — Mappings to MMS (ISO 9506-1 and ISO 9506-2) and to ISO/IEC 8802-3*. IEC 61850-8-1 (IEC).

The idea now is to add to each tuple msg_i that defines a synchronous or an asynchronous stream a new parameter that specifies the redundancy level. If we denote this parameter by k_i for a stream msg_i, then the SRT and ART are now specified like this:

$$\begin{aligned}\text{SRT} &= \{\mathrm{msg}_i \mid \mathrm{msg}_i = (C_i, D_i, T_i, O_i, P_i, k_i), i \in [1, N_{\text{S}}]\}\\ \text{ART} &= \{\mathrm{msg}_i \mid \mathrm{msg}_i = (C_i, D_i, I_i, P_i, k_i), i \in [1, N_{\text{A}}]\}.\end{aligned}$$

Moreover, the admission control and scheduler would have to be revised to take into account that transmitting the messages of a stream msg_i now requires $k_i C_i$ units of time.

Thus enhanced, the scheduler in an embedded master has the necessary information to determine how many instances of a particular message need to be transmitted by a slave. It can then take this into account when computing the elementary cycle schedules. Moreover, since the slaves also have a copy of the SRT and ART in their NRDBs, they will know how many copies of each message to transmit. And in case a slave misbehaves and transmits more replicas than specified by the redundancy level, the port guardians, which also have access to the tables, can simply drop the excessive replicas.

An additional advantage of storing the redundancy level in the SRDBs and NRDBs is that this parameter also becomes flexible: it enables different message streams to have different redundancy levels and it allows these redundancy levels to change over time, like any other stream parameter. FTTRS thus exploits the operational flexibility that FTT already provides for changing real-time parameters to also change the redundancy level of messages sent by slaves. (The admission control performed by the embedded masters needs to be updated to ensure that an increase in the redundancy level of a message is only accepted if all message replicas, together with all other messages, still results in a schedulable set of messages; but this is out of the scope of this dissertation.)

Finally, receivers must be prepared to handle multiple copies of the same message. One way to do this is to write an FTTRS software driver for the slaves that properly deduplicates. Another is to leave the deduplication to the application or a layer on top of FTTRS. I favor the latter because some higher layers might want to keep track of how many replicas out of the k_i get through (e.g., an application might want to request increasing the value of k_i if it detects that only few replicas get through). Another way of dealing with multiple copies of the same message is by ensuring that all messages are idempotent, meaning that processing the same message multiple times yields the same result regardless of how often it is processed.

Making the Trigger Message Fault Tolerant To make trigger messages fault tolerant we can also use our proactive retransmission approach. Each switch then transmits each trigger message k times, where k is the trigger message redundancy level. Indeed, this approach is particularly suitable for the trigger message. Imagine we were using an ACK-based approach instead. In that case, for a switch to be sure that it should not retransmit a trigger message, the embedded master would have to receive an ACK from each slave. But this causes scalability issues: if there were 100 slaves, the master would have to receive 100 ACKs, one from each slave. If non-acknowledgment messages (NACKs) were used instead, there might be similar scalability problems. Moreover, there would be the risk of a faulty slave sending a stream of bogus ACKs or NACKs.

Contrary to slave messages, the trigger message redundancy level should not be stored in an ART or SRT. This is so because in FTT the trigger message is not handled as an ordinary stream. A more appropriate location is the system configuration and status record (SCSR), which, as we saw earlier (Section 7.1), stores system parameters such as the duration of elementary cycles.

As to the flexibility of the trigger message redundancy level, in principle we could make it dynamic, allowing it to change at runtime. This would be in line with what we did for slave messages. However, there is an important difference between the trigger messages and slave messages: trigger messages are absolutely critical for the correct functioning of the communication subsystem, whereas slave messages are not (although they might be critical for the application). Thus, allowing tampering with the redundancy level of trigger messages requires careful thought—it must never be lowered below a level that makes the trigger message unreliable. But carefully analyzing who may modify the trigger message redundancy level, and when they may do so, is not required to prove our thesis. Hence, let us not unnecessarily complicate matters and keep the trigger message redundancy level static.[21]

Effect of Message Replication on the Elementary Cycle

The duplicated architecture of FTTRS and the proactive retransmission of messages impact the structure of the elementary cycle. As in all versions of FTT, each elementary cycle still starts with the transmission of a trigger message (Figure 7.2 on page 125), but beyond that, FTTRS differs in some important ways from HaRTES.

[21] Making the trigger message redundancy level dynamic is one of the goals for a new research project, titled dynamic fault-tolerance for flexible time-triggered Ethernet (DFT4FTT), which was recently funded by the Spanish *Ministerio de Economía y Competitividad* (grant TEC2015-70313-R) and in part by FEDER funding.

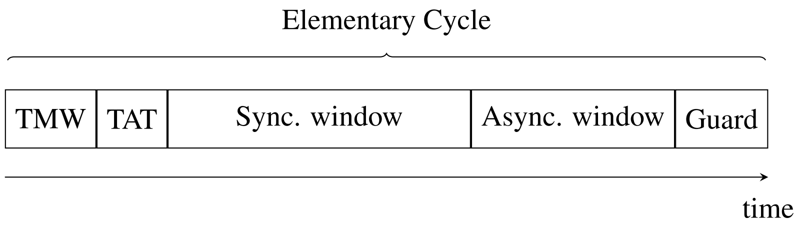


Figure 8.10: Elementary cycle structure on HaRTES downlinks. The elementary cycle is divided into a trigger message window (TMW), a synchronous window, and an asynchronous window. The turnaround time (TAT) and guard time are located between adjacent windows to prevent message transmissions running over from one window to the next. Moreover, the turnaround time gives slaves the time to decode the trigger message they received during the trigger message window.

To better appreciate these differences, we should first review how the elementary cycles are structured in HaRTES. Then we will take a look at the elementary cycles in FTTRS.

The Elementary Cycle in HaRTES Figure 8.10 shows the structure of an elementary cycle in HaRTES—specifically, it shows the structure on the downlinks of a HaRTES network (on the uplinks there is no enforced structure). On each downlink each elementary cycle is divided into windows. There is a **trigger message window** (TMW) in which the HaRTES switch broadcasts the trigger message; a **turnaround time** (TAT) that gives the slaves time to decode the trigger message; a **synchronous window** during which the HaRTES switch forwards the synchronous messages transmitted by the slaves; an **asynchronous window** during which the switch forwards asynchronous messages; and a **guard window** that ensures that slave messages do not block the transmission of the trigger message of the next elementary cycle. The duration of the synchronous and asynchronous windows may vary from elementary cycle to elementary cycle depending on what messages have been scheduled. The duration of a trigger message window depends on the size of the transmitted trigger message; the duration of the turnaround time depends on the processing speed of the nodes; and the duration of the guard window corresponds to the time to transmit a maximum sized Ethernet frame. Finally, as we know (Section 7.1), the duration of the elementary cycle itself is fixed.

Figure 8.11 shows an example of the messages that in HaRTES might be exchanged during an elementary cycle. Time progresses from left to right. There are three slaves s_1, s_2, and s_3, each connected to a single HaRTES switch. This corre-

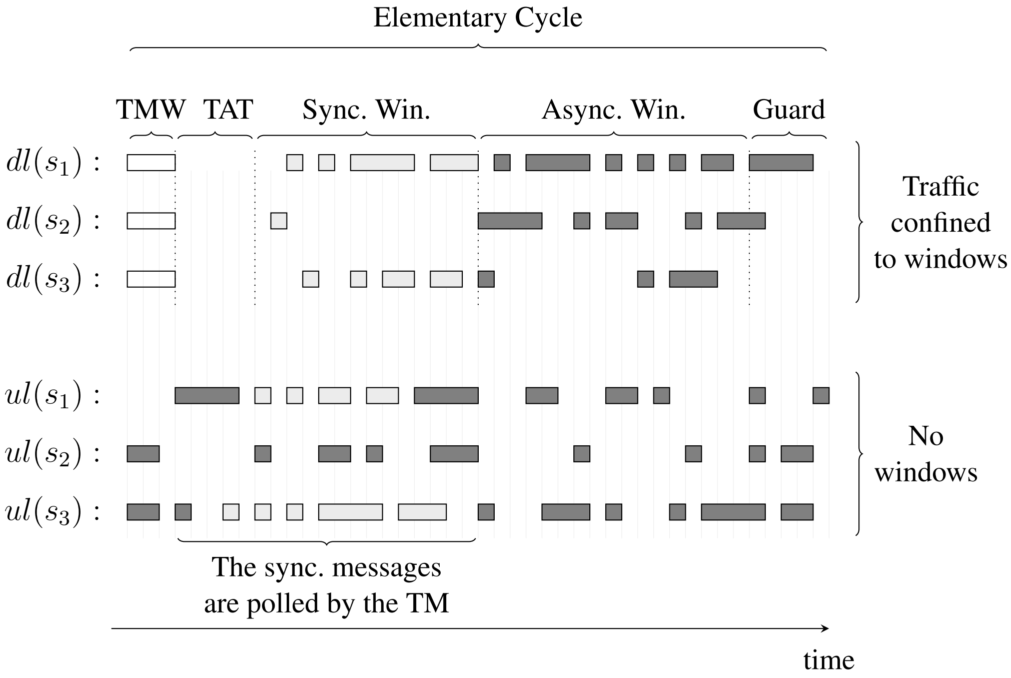


Figure 8.11: Example traffic on the downlinks and uplinks of a HaRTES switch. The notation $dl(s_i)$ denotes the downlink from switch A to a slave s_i and $ul(s_i)$ denotes the uplink to switch A from a slave s_i. Messages drawn in white are trigger messages, messages drawn in light gray are synchronous messages, and messages drawn in dark gray are asynchronous messages.

sponds to the architecture of Figure 8.4 on page 172. The three rows of messages labeled $dl(s_1)$, $dl(s_2)$, and $dl(s_3)$ show the messages transmitted from the HaRTES switch to each of the slaves. The notation dl stands for downlink. The three rows of messages labeled $ul(s_1)$, $ul(s_2)$, and $ul(s_3)$ show the messages transmitted in the opposite direction, from the slaves to the HaRTES switch. The notation ul stands for uplink. Messages drawn in white are trigger messages, messages drawn in light gray are synchronous messages, and messages drawn in dark gray are asynchronous messages.

As we can see in Figure 8.11, the HaRTES switch's traffic shaping only enforces the elementary cycle structure of Figure 8.10 on the downlinks. On the uplinks, in contrast, slaves transmit asynchronous messages at arbitrary times and synchronous messages after the reception and decoding of a trigger message. In other words, the traffic is only confined to windows on the downlinks. The figure also illustrates that each window has the same duration on all downlinks. For instance, the synchronous window has the same duration on downlinks $dl(s_1)$, $dl(s_2)$, and $dl(s_3)$. Similarly, the trigger message window, the asynchronous window, and the guard window each have the same duration on each of the downlinks. Moreover, we can see that on the downlinks—$dl(s_1)$, $dl(s_2)$, and $dl(s_3)$—the HaRTES traffic shaping confines synchronous messages to the synchronous window and asynchronous messages to the asynchronous window, with some potential overshooting into the guard window, as illustrated on downlinks $dl(s_1)$ and $dl(s_2)$. On the uplinks—$ul(s_1)$, $ul(s_2)$, and $ul(s_3)$—there is no such confinement and there are no windows. Finally, we can see that, despite the overshooting on downlinks $dl(s_1)$ and $dl(s_2)$, the guard window prevents the overshooting asynchronous messages from overrunning the next elementary cycle. For example, without the guard window, the last asynchronous message transmitted on downlink $dl(s_1)$ would block the transmission of the trigger message on that downlink during the next elementary cycle.

Preventing the blocking of trigger messages is crucial to ensure that these reach all slaves at approximately the same time. And the simultaneous arrival is crucial in turn for the arrival of trigger messages to act as an **elementary cycle synchronization event**; that is, an event that synchronizes the elementary cycles among all slaves.

The Elementary Cycle in FTTRS FTTRS inherits its elementary cycle structure from HaRTES. As in HaRTES (Figure 8.10), in FTTRS the traffic on the downlinks is divided into a trigger message window, a turnaround time, a synchronous window, an asynchronous window, and a guard window. Moreover, like in HaRTES, in FTTRS synchronous and asynchronous messages are confined to their respective windows on the downlinks thanks to the traffic shaping applied by the switches. On the uplinks, as in HaRTES, there are no windows and the only structure is the one derived from the regular polling by the trigger message.

Despite the similarities, how messages are exchanged in FTTRS differs from HaRTES. This is due to three choices we made to achieve error compensation: we have two Ethernet switches, towards which each slave transmits its messages in parallel; switches exchange through their interlinks all messages they receive from uplinks; and both slaves and embedded masters use proactive retransmissions.

An example of the traffic we might observe in FTTRS is shown in Figure 8.12. The traffic shown still corresponds to three slaves s_1, s_2, and s_3. Each slave, however, now has two slave links, one for each of two switches A and B. As a result, the figure shows two downlinks and two uplinks for each slave. Thus, a slave s_i has a downlink $dl_A(s_i)$ from switch A and a downlink $dl_B(s_i)$ from switch B. Similarly, a slave s_i has an uplink $ul_A(s_i)$ to switch A and an uplink $ul_B(s_i)$ to switch B. (To not further complicate the figure, the messages exchanged on the interlinks are not shown.)

The figure reflects that each slave transmits the same traffic in parallel through both its uplinks. Thus, for example, the traffic on the uplinks of slave s_1, i.e., $ul_A(s_1)$ and $ul_B(s_1)$, has the same pattern. It also shows that the trigger messages are transmitted three times from each of the switches A and B through each downlink (top left of the figure). The trigger message redundancy level k is therefore 3 in the example. Finally, we can see that more slave messages are transmitted on the downlinks than on the uplinks. This is due to a phenomenon that involves messages originating at slaves and that we may call **replica radiation**.

Figure 8.13 illustrates replica radiation. Since the switches exchange all the traffic they receive from uplinks, we have that when no messages are lost, a slave receives four copies of a message when another slave transmits two copies in parallel through its uplinks. In particular, of the four copies received, two are received through each downlink. As a result, there is more traffic on the downlinks than the uplinks. This is also reflected in Figure 8.12. For instance, the number of synchronous messages that a slave s_i receives through a downlink is twice the number of messages that have been transmitted to s_i through the uplinks. Hence, the duration of the synchronous window is longer in FTTRS than it was in HaRTES. In the same way, the asynchronous window needs to be longer as well since we can observe the same replica radiation phenomenon for asynchronous messages. For trigger messages, in contrast, there is no replica radiation. This is so because replica radiation does not involve messages originating within switches since switches do not forward each other's trigger messages through the downlinks. To take into account replica radiation and the consequent bottleneck that downlinks constitute, in FTTRS the rate with which messages are transmitted through uplinks must be

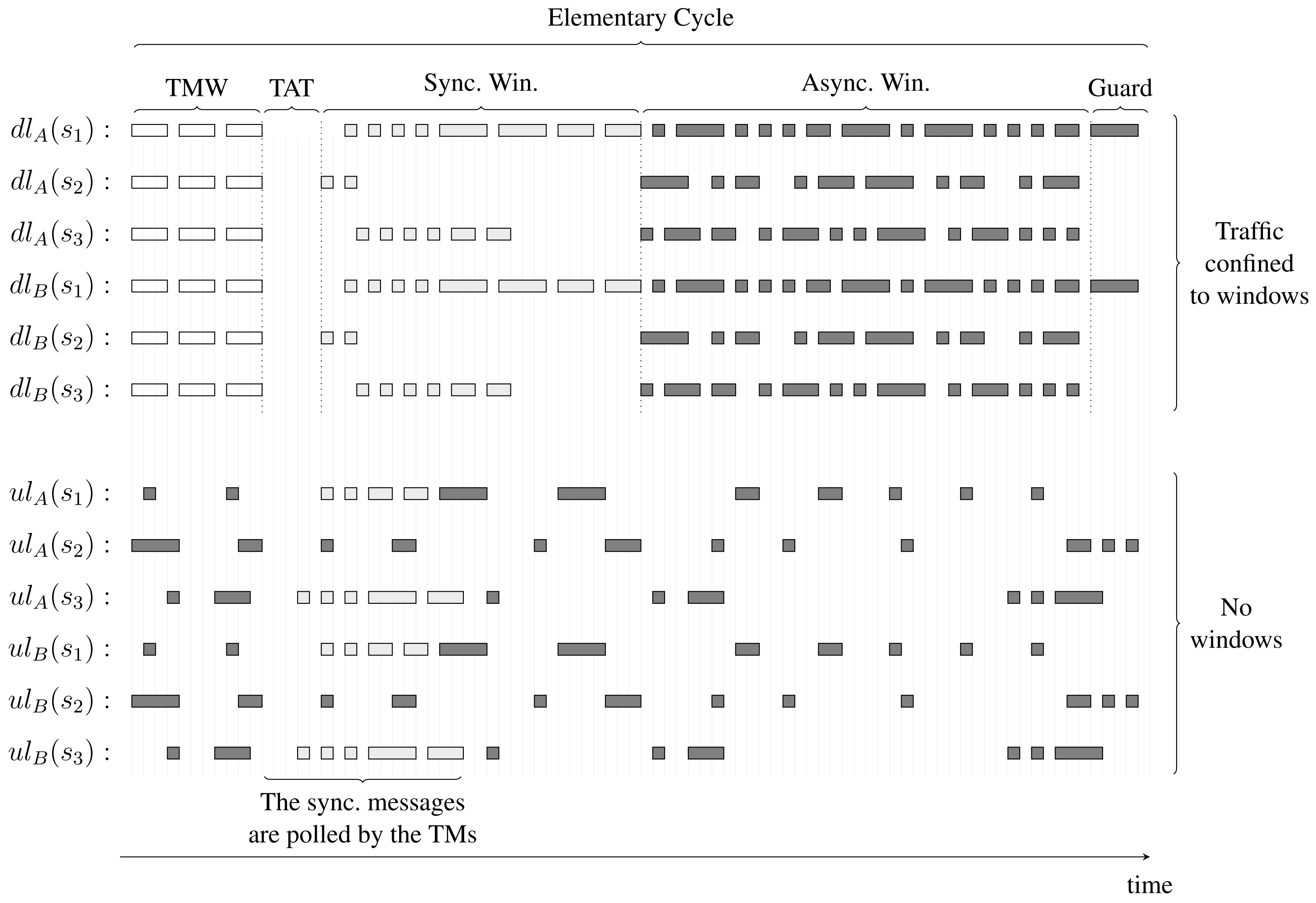


Figure 8.12: Example traffic on the downlinks and uplinks of an FTTRS network. The notation $dl_A(s_i)$ denotes the downlink from switch A to a slave s_i and $ul_A(s_i)$ denotes the uplink to switch A from a slave s_i. Similarly, $dl_B(s_i)$ and $ul_B(s_i)$ denote the downlink and uplink from s_i to the other switch, B. Messages drawn in white are trigger messages, messages drawn in light gray are synchronous messages, and messages drawn in dark gray are asynchronous messages.

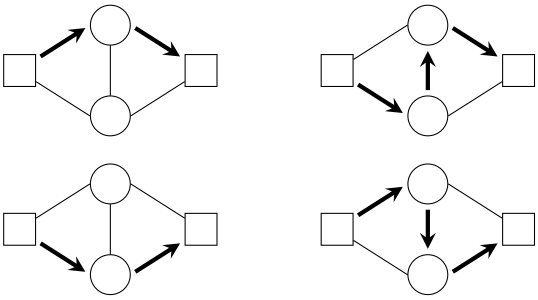


Figure 8.13: Replica radiation in FTTRS. As a result of the switches exchanging all the traffic they receive, there are four possible paths from a given source node to a given destination node. This also means that if a source transmits two messages in parallel through its links, four copies will be received at the destination, two through each downlink.

reduced with respect to HaRTES.

(The way to take into account the bottleneck is by properly setting the redundancy level, period, and minimum interarrival time in the ART and SRT of the SRDBs and to improve the admission control and scheduler to consider replica radiation. This, however, is out of the scope of this dissertation.)

Finally, there is an additional issue related to replicating the trigger message: in FTT the slaves learn when a new elementary cycle starts upon the reception of a trigger message. If there is only one trigger message, which is broadcast to all slaves in parallel, then all slaves that receive it will do so at approximately the same time, ensuring a synchronized start of the corresponding elementary cycle. This is how it works in HaRTES. If, however, there are k trigger message replicas, some of which may be lost due to faults in the links, then the elementary cycle synchronization must be revised. Otherwise some slaves may start their elementary cycles before others. This may happen, for instance, if one slave only receives the first trigger message and another only receives the last.

Achieving Elementary Cycle Synchronization among Slaves

In the FTT paradigm the arrival of a trigger message prompts a new elementary cycle in the slaves. Hence, if all slaves receive a trigger message simultaneously, the elementary cycles among the slaves are synchronized. Figure 8.14 shows how this would work in HaRTES for three slaves s_1, s_2, and s_3. The arrival of the trigger message in a downlink $dl(s_i)$ of a slave s_i acts as a synchronization event. Since the switch broadcasts the trigger messages in parallel through all downlinks and

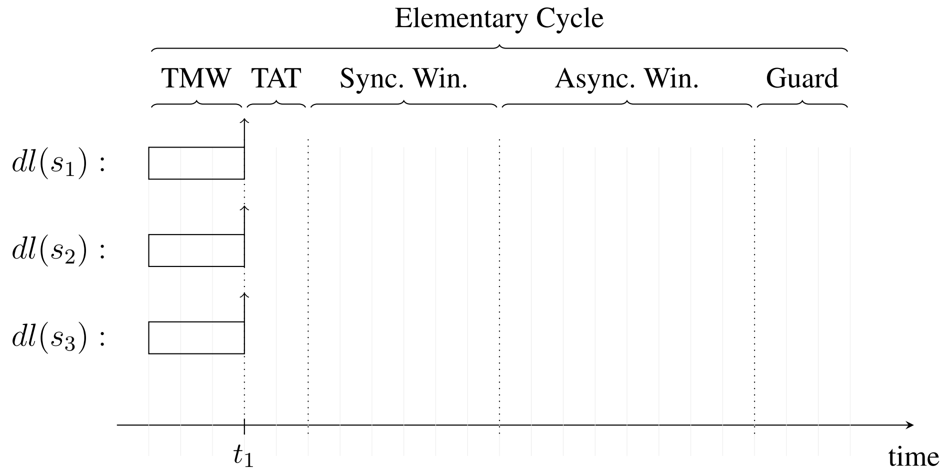


Figure 8.14: Elementary cycle synchronization among slaves in HaRTES. All three slaves s_1, s_2, and s_3 detect the arrival of the trigger message at t_1. The synchronized trigger message arrivals can therefore act as a synchronization events in HaRTES.

differences in propagation delays are negligible, the arrivals occur simultaneously, at time t_1 in the figure, and, thus, all slaves begin the turnaround time of the new elementary cycle in sync.

In FTTRS, however, there is no longer a single trigger message per elementary cycle. As a consequence there is no single trigger message arrival per downlink that can act as a synchronization event. Instead there are between 1 and k trigger message arrivals—assuming that k, the trigger message redundancy level, has a value such that at least one trigger message gets through any link.

Figure 8.15 shows an example for $k = 3$. As in Figure 8.12, the notation $dl_j(s_i)$ labels the traffic observed on a downlink from a switch j to a slave i. The first slave, s_1, receives the first trigger message at time t_1, but no other trigger messages. Similarly, slave s_2 only receives the second trigger message, doing so at time t_2, and slave s_3 only receives the third trigger message, doing so at time t_3. Thus, if each slave immediately initiated its elementary cycle upon the arrival of any trigger message, as in previous versions of FTT, slave s_1 would start its elementary cycle at time t_1, slave s_2 would do so at time t_2, and slave s_3 at time t_3. From the point of view of the slaves, some elementary cycles might then be shorter than others. Is this a problem? Maybe not. The scheduler and admission control could ensure that all schedules are dimensioned to fit in the shortest possible elementary cycles, i.e., those starting at t_3. In that case, the fact that the slaves of Figure 8.15 received the

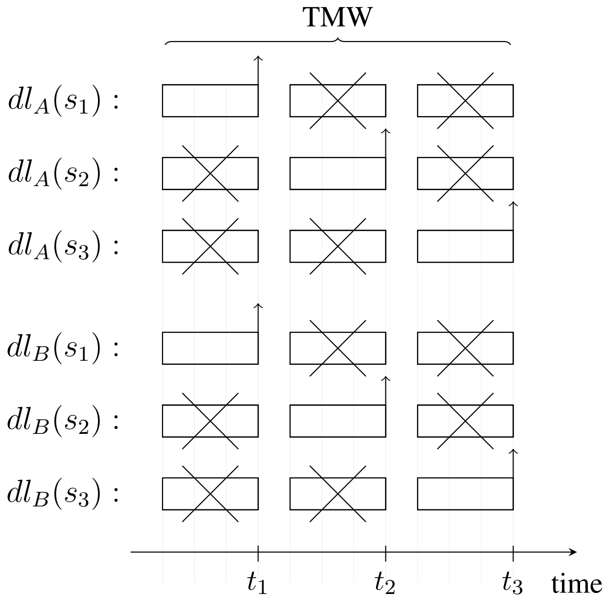


Figure 8.15: In FTTRS, when messages are lost, slaves may observe different trigger message arrival times.

trigger messages at different times would simply lead to slave s_1 being polled before slaves s_2 and s_3; and s_2 being polled before s_3, with all polled messages being temporarily buffered by the traffic-shaping switches and then forwarded through the downlinks at the start of the synchronous window. However, slaves starting their elementary cycles at different times can be undesirable: the application running on top of an FTTRS slave might want to be notified when each elementary cycle starts and then use this notification for its own purposes. In that case the application might benefit from the notifications occurring with predictable periodicity, instead of occurring with jittery quasi-periodicity that depends on the random loss of trigger messages. This is particularly important if we want to put a control application on top of FTTRS where tasks are triggered by elementary cycles.[22]

Figure 8.16 shows another example of how slaves may observe different trigger message arrival times: some slaves observe three arrival times on a downlink, others observe two, and yet others observe only one. Moreover, the same slave may

[22] An example of a control application where a low-jitter periodic elementary cycle is used to trigger the execution of tasks is described by Sinisa Derasevic, Julián Proenza, and Manuel Barranco. “Using FTT-Ethernet for the Coordinated Dispatching of Tasks and Messages for Node Replication”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014.

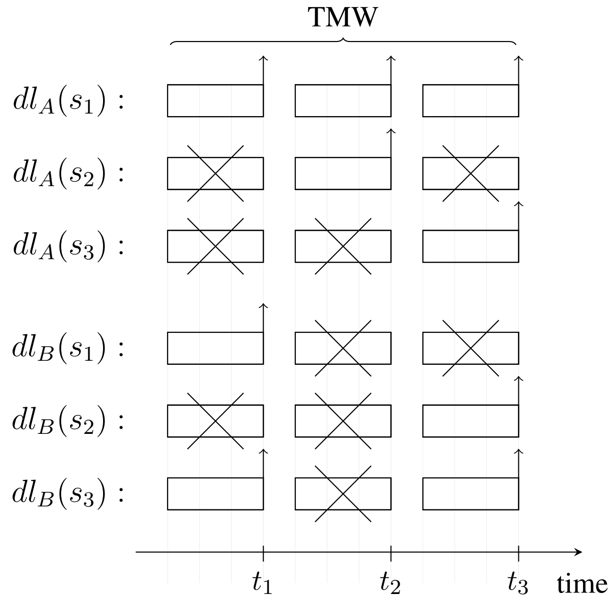


Figure 8.16: Another example showing that in FTTRS, when messages are lost, slaves may observe different trigger message arrival times.

observe different arrival times on each of its downlinks. Whether the scenario is like in Figure 8.15, like in Figure 8.16, or like in any other scenario where each slave receives at least one trigger message, the slaves should all consider their elementary cycles—or more specifically, their turnaround times—to start at the same time.

So what can we do? How can we ensure that in FTTRS all non-faulty slaves agree on when each elementary cycle starts, regardless of which subset of the trigger messages they receive?

One solution is to ensure that during each trigger message window each switch transmits its trigger messages isochronously and in lockstep with the other switch.

Isochronous Trigger Message Transmission

As I just said, to ensure that in FTTRS all non-faulty slaves agree on when each elementary cycle starts, one solution is to ensure that each switch transmits its trigger messages isochronously and in lockstep with the other switch. But what does this mean?

The prefix *iso* means equal and *chronous* means time. **Isochronous** therefore

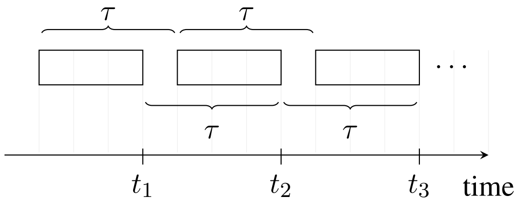


Figure 8.17: Isochronous message transmission in FTTRS.

means equal in duration or frequency, or occurring at regular intervals. Saying that messages are transmitted isochronously is thus saying that they are transmitted with fixed periodicity, with the time between successive transmissions being constant. Figure 8.17 illustrates this for a fixed intertransmission time τ or, equivalently, for a fixed interarrival time τ. All messages are assumed to have the same duration.

The key of isochronous transmissions is that they allow us to predict the arrival of any future message if we have received at least one previous message. For instance, and referring again to Figure 8.17, if we know that the first message arrived at time t_1, we immediately know that the second message will arrive at $t_2 = t_1 + \tau$, that the third message will arrive at $t_3 = t_1 + 2\tau$, and that the ith message will arrive at $t_1 + (i - 1)\tau$. Hence, if the trigger messages are transmitted isochronously, then, as long as each slave receives at least one trigger message of a given elementary cycle, it can predict when the last trigger message should arrive. For instance, in Figure 8.15, slaves s_1 and s_2 could predict when the third trigger message should have arrived. All three slaves can thus synchronize their elementary cycles by either using the actual arrival time of the third trigger message or by using the predicted arrival time.

For this to work, however, slaves must know whether they have received the first, second, third, or, more generally, the ith trigger message of a given elementary cycle. This calls for sequence numbers that take values from 1 to k. The trigger messages must also all have the same size, which we can ensure through padding, if necessary. Moreover, slaves need to know the value of the intertransmission time τ and of the trigger message redundancy level k. Assuming the values of τ and k are constant and do not change at runtime, these values could be preconfigured within the slaves. The sequence numbers, however, will have to be carried within the trigger messages. Thus, we have to add at least a sequence number to the trigger messages. And while we are at it, we can just as well also add the values of k and τ to the trigger messages. That way we open the possibility for making these parameters modifiable at runtime (in this work, however, we will assume that they are constant).

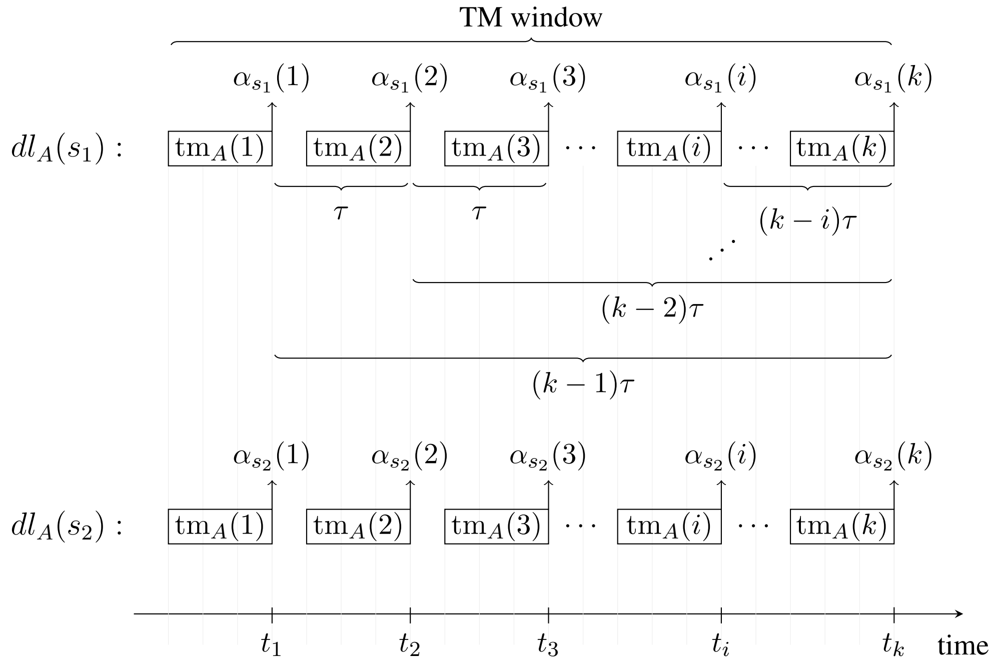


Figure 8.18: Isochronous trigger message transmission. The figure shows that trigger messages from a switch A reach two slaves s_1 and s_2 with a fixed interarrival time τ. This allows a slave to predict the end of the trigger message window as long as it receives at least one out of the k trigger messages.

Figure 8.18 illustrates how two slaves s_1 and s_2, which receive isochronously transmitted trigger messages through the downlinks corresponding to a switch A, can use the timing parameters for elementary cycle synchronization (to keep the example simple, we will only consider one switch for the moment). When a slave s_j receives a trigger message $\mathrm{tm}_A(i)$, it observes an **arrival event** $\alpha_{s_j}(i)$ at time t_i. When this occurs, it extracts from the trigger message the sequence number i, the redundancy level k, and the interarrival time τ. Using these parameters, it then predicts that the arrival event $\alpha_{s_j}(k)$, i.e., the arrival of the last trigger message, will occur within $(k - i)\tau$ units of time. Thus, to synchronize the elementary cycles among the slaves, all each slave has to do when it receives the ith trigger message of an elementary cycle is to set a timer to elapse after $(k - i)\tau$ units of time. That way, the timers of all slaves will elapse at the same time, t_k, and the timeout can be used as the elementary cycle synchronization event.

For this to work, we must assume that the clocks used by the slaves can only negligibly drift apart during the duration of a trigger message window. Although if the drift is not negligible, it can also be corrected. How? The slaves could simply predict the arrival times of the trigger messages of the next elementary cycle and then compute the difference in time between predicted trigger message arrival times and actual arrival times. Knowing this difference, a slave could then adjust the rate of its clock. For this to work, the slaves must know the duration of the elementary cycles, but this value is already conveyed in the trigger messages along with the other timing parameters.

We now have a mechanism to synchronize the elementary cycles among slaves if there is a single switch transmitting trigger messages isochronously. In FTTRS, however, there are two switches. To work in that case as well, the mechanism requires that the two switches both broadcast the trigger messages isochronously; that they do so in lockstep, meaning that when one switch initiates the broadcast of the ith trigger message, the other does the same quasi-simultaneously; and that the quasi-simultaneously transmitted trigger messages include the same data concerning elementary cycle synchronization (e.g., sequence numbers). If these conditions are met, then slaves can synchronize their elementary cycles regardless of the link through which they receive trigger messages. To meet these conditions, however, the switches must be replica deterministic, both from the point of view of the time domain (isochronous lockstep transmission) and the value domain (trigger messages must carry corresponding parameters, e.g., the same value for τ and k). We will talk more about the replica determinism of the switches in the context of tolerating permanent faults.

8.7.3 Tolerating Permanent Faults

To satisfy requirement R3.3, our communication subsystem must tolerate a permanent fault in any one of its components, no matter which one. Since we do not consider slaves as being part of the communication subsystem, this means that FTTRS must tolerate any permanent fault occurring in an interlink, slave link, or switch. Luckily, much of what we need is already in place.

To tolerate a permanent fault in an interlink we already decided upon having multiple redundant interlinks (Section 8.3). All that we have to ensure now is that switches use the interlinks in parallel whenever they exchange messages.

To tolerate a permanent fault in a slave link, and in line with design principle DP2, according to which we favor error compensation (Section 8.1.3), each slave must transmit each message in parallel through both its links. As a result, the failure of a slave link is already almost tolerated since FTTRS satisfies the following:

- An FTTRS network always starts out with at least two disjoint paths between any pair of slaves (Figure 8.6 on page 175).
- Slaves are prevented from presenting two-faced behaviors towards the switches and other slaves because of the Ethernet CRC (Section 8.2) and because port guardians drop incorrect messages (Section 8.6.2). Specifically, whenever a slave transmits a message in parallel through its links, the message is transmitted consistently in space (correctness property SF6 on page 162) or, if there is an error, the faulty transmission—on either one or both of the links—is turned into an omission.
- FTTRS switches are fail silent (Section 8.6.1) and will thus correctly forward messages according to their SRDBs[23] or not transmit nor forward any message at all.
- Receivers properly deduplicate repeated messages or, alternatively, messages are idempotent (Section 8.7.2).
- Each switch forwards each message it receives from a slave to the other switch, which enables additional redundant paths between slaves (FF6 on page 163).

Given the above, what remains to be done to tolerate the failure of any slave link is to ensure that the two switches simultaneously provide the same FTT service and,

[23] Remember that in FTTRS, like in HaRTES, data messages from slaves are not forwarded using traditional MAC-based Ethernet forwarding, but using forwarding based on who the subscribers of a particular message stream are, as specified in the SRDBs.

in particular, that they always have the same SRDB contents at the same time. In other words, we must enforce replica determinism for the switches so that a slave perceives a consistent FTT service, regardless of whether it has one or two correct slave links.

Finally, to tolerate a permanent fault in a switch we must also ensure that the switches are replica deterministic. Otherwise the permanent fault of one switch would not be masked by the other: they would not provide corresponding services and a surviving switch would not be able to seamlessly carry on with a consistent FTT service.

8.7.4 Enforcing the Necessary Replica Determinism

As we just outlined (Section 8.7.3), we have to make the switches replica deterministic to ensure that slaves can continue to communicate upon the failure of one of their links or of one of the switches. Moreover, as we said earlier (Section 8.7.2), switch replica determinism is also necessary to ensure that the masters transmit trigger messages in lockstep, which in turn we need for the slaves to achieve proper elementary cycle synchronization in the presence of transient link faults. Links and slaves, on the other hand, do not need to be made replica deterministic: links are simply conduits for messages to which the concept of replica determinism does not apply and the replication of slaves is out of the scope. We will thus focus on the switches exclusively.

The type of replication we use for the switches determines what it means for them to be replica deterministic. In our case, in accordance with design principle DP3 (page 153), we will be using active replication for the value domain and semi-active replication for the time domain. This means that to be replica deterministic the switches must simultaneously provide a consistent FTT service, with no switch acting as a passive backup. To discuss how we can achieve this, let us again partition each switch into distinct functional parts—an internal switch, a set of port guardians, and an embedded master—and see how to enforce replica determinism for each of these parts.

Enforcing Replica Determinism for the Internal Switches

Before we discuss replica determinism for the internal switches, let us first clarify what exactly constitutes a correct service for them.

The service of an internal switch consists in forwarding Ethernet frames and, as we will see now, an internal switch performs this service in different ways depending

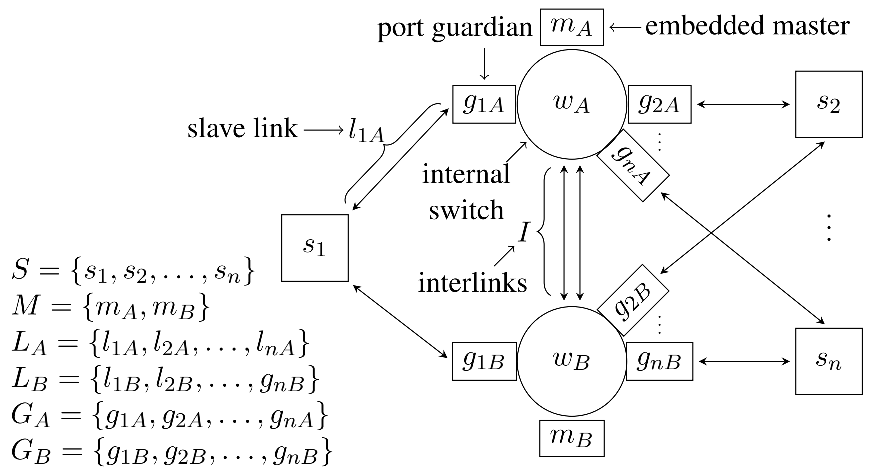

port guardian
m_A
embedded master
g_{1A}
w_A
g_{2A}
s_2
slave link
l_{1A}
g_{nA}
internal switch
I
interlinks
s_1
$S = \{s_1, s_2, \ldots, s_n\}$
$M = \{m_A, m_B\}$
$L_A = \{l_{1A}, l_{2A}, \ldots, l_{nA}\}$
$L_B = \{l_{1B}, l_{2B}, \ldots, g_{nB}\}$
$G_A = \{g_{1A}, g_{2A}, \ldots, g_{nA}\}$
$G_B = \{g_{1B}, g_{2B}, \ldots, g_{nB}\}$
g_{2B}
g_{1B}
w_B
g_{nB}
s_n
m_B

Figure 8.19: Notation for the different components of the FTTRS architecture. See the text for an explanation.

on the type of FTT message encapsulated within the frames (Figure 7.3 on page 126). Moreover, how an internal switch forwards a frame is not only determined by the type of encapsulated message, but also by the location at which the frame is handed over to the internal switch.

To make it easier to discuss how an internal switch forwards frames, let us use shorthand labels for the different components of an FTTRS network. Figure 8.19 shows the architecture of FTTRS again, but this time each switch's embedded master and internal switch are drawn separately and labels are introduced for the different components: s_i for a slave; w_A and w_B for the internal switch of an FTTRS switch A and B, respectively; m_A and m_B for the corresponding embedded masters; g_{iA} and g_{iB} for the corresponding guardians that restrict the failure semantics of a slave s_i; l_{iA} and l_{iB} for the two slave links of a slave s_i; I for the set of interlinks; S for the set of all slaves of the network; M for the set that comprises the two embedded masters m_A and m_B; L_A and L_B for the set of slave links attached to switch A and B, respectively; and G_A and G_B for the set of port guardians of switch A and B, respectively.

Using these labels, the forwarding performed by an internal switch w_A can be summarized as shown in Table 8.5. The table indicates for each type of message (first column) how the frame encapsulating that message is forwarded when it is received by w_A. As in the table, let us refer to the message to be forwarded as msg and to the Ethernet frame carrying it as f.

If msg is a synchronous message, then there are two sets of destinations to which

Table 8.5: Forwarding by an internal switch w_A of an Ethernet frame f carrying a message msg. The notation $\mathrm{sub}_A(\mathrm{msg})$ denotes the set of subscribers of msg according to the SRDB of switch A. Master command messages are piggybacked on trigger messages as explained in Section 8.7.4.

Message type of msg	Source of f	Destination of f	Forwarding time	Buffering time
Sync. data message	$g_{iA} \in G_A$	$\{l_{jA}, \mid s_j \in \mathrm{sub}_A(\mathrm{msg}) \subset S\} \cup I$	SW[a]	TAT[b]
Sync. data message	I	$\{l_{jA}, \mid s_j \in \mathrm{sub}_A(\mathrm{msg}) \subset S\}$	SW	TAT
Async. data message	$g_{iA} \in G_A$	$\{l_{jA}, \mid s_j \in \mathrm{sub}_A(\mathrm{msg}) \subset S\} \cup I$	AW[c]	TMW[d], TAT, SW, Guard[e]
Async. data message	I	$\{l_{jA}, \mid s_j \in \mathrm{sub}_A(\mathrm{msg}) \subset S\}$	AW	TMW, TAT, SW, Guard
Request message	$g_{iA} \in G_A$	$\{m_A\} \cup I$	TAT, SW, AW	TMW, Guard
Request message	I	m_A	TAT, SW, AW	TMW, Guard
Trigger message	m_A	$L_A \cup I$	TMW	—
Trigger message	I	m_A	TMW	—

[a] Synchronous window.

[b] Turn-around time.

[c] Asynhronous window.

[d] Trigger message window.

[e] Guard window.

f may be forwarded (first two rows of Table 8.5).

If f was handed over to w_A by a guardian $g_{iA} \in G_A$ (first row of Table 8.5), then the destination to which f is forwarded includes both the subscribers of msg and the other switch—switch B. Specifically, the frame is forwarded through the set of interlinks I and the set of slave links $\{l_{jA} \mid s_j \in \mathrm{sub}_A(\mathrm{msg})\}$, where $\mathrm{sub}_A(\mathrm{msg})$ denotes the set of subscribers of msg according to the SRDB of switch A.

If f was handed over to w_A by one or more interlinks (second row of Table 8.5), then w_A forwards f, as before, through the set of slave links $\{l_{jA} \mid s_j \in \mathrm{sub}_A(\mathrm{msg})\}$. This time, however, w_A refrains from forwarding f through the set of interlinks I. Why? Because f came through the interlinks in the first place and forwarding it back would not only be unnecessary as the other internal switch, w_B, already has a copy, but it would create a loop.

One thing to note is that in the second case (the second row of Table 8.5), where the source of f is the set of interlinks I, multiple copies of f may arrive at w_A in parallel (one copy per correct interlink). Hence, to avoid increasing the number of messages that may be transmitted through each slave link with each additional interlink, w_A should deduplicate the spatially redundant copies and only forward one copy through each destination slave link. For instance, if there are two interlinks and two copies of f arrive at w_A in parallel, then w_A should forward only one copy of f through each slave link in the set $\{l_{jA} \mid s_j \in \mathrm{sub}_A(\mathrm{msg})\}$. The same deduplication process should also be applied to spatially replicated frames arriving through the interlinks that are carrying other types of FTT message.

So much for the value domain. To completely describe the forwarding of synchronous messages, however, we must also cover the time domain. Indeed, when forwarding a frame, an internal switch should not only take into account the forwarding destination, but also the forwarding time. That is, contrary to a regular Ethernet switch, an internal FTTRS switch shapes traffic and may need to buffer incoming frames to ensure that they are forwarded to their destination during the appropriate window of the elementary cycle. In the case of frames carrying synchronous messages (again, the first two rows of Table 8.5), the appropriate time to forward them is the synchronous window of the current elementary cycle (first two rows of the next to last column of Table 8.5). However, a frame carrying a synchronous message may arrive at the internal switch earlier, during the turnaround time. This is so because slaves may take different amounts of time to process the trigger message and some may thus respond to the polling earlier than others. In that case, the frame is buffered during the turnaround time by the internal switch until the synchronous window starts (first two rows of the last column of Table 8.5) and only then the internal switch forwards the frame. This ensures that on the downlinks synchronous messages are confined to the synchronous window.[24]

The buffering of synchronous messages only occurs for messages transmitted by slaves during the turnaround time. If synchronous messages are transmitted at any other point in time, they are not buffered. In fact, unless they were transmitted during the synchronous window, they would not even reach the internal switch. After all, any frames carrying synchronous message transmitted outside the turnaround time or the synchronous window would have been deemed untimely and dropped by the corresponding port guardian.

On to asynchronous messages (Figure 7.3 on page 126). Here we have to distinguish between asynchronous data messages (third and fourth row of Table 8.5) and update request messages (fifth and sixth rows).

Like synchronous data messages, asynchronous data messages may also be handed over to an internal switch w_A by a guardian g_{iA} (third row of Table 8.5) or by the set of interlinks I (fourth row). In either case, the behavior of w_A in the value domain is analogous to the behavior we just saw for synchronous data messages. That is, if msg denotes the asynchronous data message and f the frame carrying it, and if f was handed over by a guardian (third row), then w_A forwards f through the set of slave links $\{l_{jA} \mid s_j \in \text{sub}_A(\mathrm{msg})\}$ and through the interlinks I. On the other hand, if f was handed over to w_A by the set of interlinks I (fourth row), then f is forwarded through $\{l_{jA} \mid s_j \in \text{sub}_A(\mathrm{msg})\}$ only.

Regarding the time domain, asynchronous data messages should be forwarded by an internal switch during the asynchronous window and buffered when they arrive during any other time of the elementary cycle. This is reflected in the last two columns of the third and fourth row of Table 8.5. Contrary to synchronous data messages, slaves are allowed to transmit asynchronous data messages during any of the elementary cycle windows—assuming that the slaves' transmissions do not violate the minimum interarrival time specified in the asynchronous requirements table. Thus, frames carrying asynchronous data messages are buffered by an internal switch during all windows of the elementary cycle except the asynchronous window, during which they are forwarded.

Now, what if an internal switch w_A forwards a frame f that carries a slave update request message msg (fifth and sixth rows of Table 8.5)? Like any other frame ultimately originating at a slave, f may reach w_A through a port guardian or through the interlinks. If f reaches w_A through a port guardian $g_{iA} \in G_A$ (fifth row of Table 8.5), then it is delivered to the local master m_A. Moreover, f is also forwarded through the set of interlinks I. The forwarding through the interlink gives the other

[24] This buffering is precisely what happened in Figure 8.12 on page 191 with the first synchronous message transmitted by s_3 through uplinks $ul_A(s_3)$ and $ul_B(s_3)$.

internal switch, w_B, another chance of receiving a copy of f if it did not receive any through the slave link l_{iB}. If, on the other hand, f reaches w_A through the interlinks (sixth row of Table 8.5), then f is only forwarded to the local master m_A since w_B must already have a copy of f.

As to the time domain, frames carrying update request messages are buffered during the trigger message window and the guard window (last column of Table 8.5), and forwarded during all other windows (next to last column). The buffering ensures that update requests cannot interfere with the isochronous transmission of trigger messages through the interlinks during the trigger message window. This will be important later on when we design a mechanism for the masters to synchronize their transmission of trigger messages (see the section about replica determinism enforcement for masters on page 212). As to why the asynchronous update request messages are also forwarded during the turnaround time and synchronous window, and not confined to the asynchronous window exclusively, the reason is that an internal switch only confines asynchronous messages to the asynchronous window on the slaves' downlinks, but update requests are not sent through downlinks—they are sent to masters through interlinks and dedicated links that interconnect an internal switch with its embedded master.

Next up is the forwarding of frames originating at embedded masters (last two rows of Table 8.5), which in FTTRS will carry trigger messages only (master command messages will be piggybacked within trigger messages, as we will see in the section about updating the system requirements across the whole network, which starts on page 226).

When a frame f carrying a trigger message msg arrives at an internal switch w_A, w_A may have received f from its local master m_A (next to last row in Table 8.5) or from the set of interlinks I (last row). If f came from m_A, then w_A must forward it through all slave links attached to A (set L_A) and through the set of interlinks I. That way w_A broadcasts f. On the other hand, if f came through the interlinks I, then it must have been broadcast by m_B. In that case w_A forwards the trigger message carrying frame f to m_A. The master m_A will then be able to use the trigger message msg to ensure replica determinism with m_B, as we will see shortly (section on enforcing replica determinism between masters, page 207). As to why w_A does not forward msg to the slave links L_A when it came from m_B through the interlinks (last row), this is so because the links in the set L_A are already receiving the isochronously transmitted trigger messages from master m_A and the forwarding would interfere with the isochrony of these messages. Moreover, since we use active replication, the trigger messages from m_A and from m_B will have corresponding values anyway. Thus, having w_A forward the trigger messages from m_B through the downlinks is unnecessary if it is already forwarding the ones from m_A.

A further point to note is that to maintain the isochrony of the trigger messages it is critical that an internal switch forwards each incoming trigger message with minimum delay and jitter. Achieving this is a matter of having internal switches forward trigger messages immediately, without any buffering.

This concludes the overview of how frames are forwarded by internal switches in FTTRS. Now, how do we ensure that the internal switches are replica deterministic? Well, first we have to define what replica determinism means for internal switches. Broadly speaking, we can define two internal switches w_A and w_B as being replica deterministic if they forward frames in an analogous way. More precisely, three conditions have to be met:

W1: The internal switch w_A should forward frames as described in Table 8.5 and w_B should forward frames as described in the same table when we substitute each occurrence of the subscript A by B.

W2: The elementary cycles and their windows must be synchronized in time among the two internal switches.

W3: The two internal switches must agree on who the subscribers for each message are; that is, for each message msg we must have that $\mathrm{sub}_A(\mathrm{msg}) = \mathrm{sub}_B(\mathrm{msg})$.

If these three requirements are met, the internal switches will behave identically when they receive the same sequence of frames within the same elementary cycle window. Of course, due to message losses and other errors, it is unlikely for two switches to receive identical inputs. Nevertheless, according to our definition, they will still be replica deterministic.

As to how to meet these requirements, this is a matter of properly implementing the switches for W1 and ensuring replica determinism for the embedded masters for W2 and W3. After all, with respect to W2, it is the masters who determine when each elementary cycle starts and ends. Moreover, with respect to W3, it is the masters who determine the contents of the SRDB—and thus what $\mathrm{sub}_A(\mathrm{msg})$ and $\mathrm{sub}_B(\mathrm{msg})$ return.

In conclusion, internal switches will be replica deterministic if they implement the same logic (i.e., Table 8.5) and their corresponding embedded masters are synchronized in time and have identical contents in the SRDB. Hence, internal switches will be replica deterministic as a consequence of the replica determinism of the embedded masters. Now, before we discuss the replica determinism of the masters, let us first consider the port guardians.

Enforcing Replica Determinism for the Port Guardians

As we saw earlier (Section 8.6.2), the function of a port guardian is to restrict the failure semantics of a slave by dropping frames sent by the slave that have incorrect metadata or that carry untimely FTT messages, while letting all others pass. Thus, two port guardians will be replica deterministic, i.e., show corresponding outputs if, when they receive the same frame within the same window of the same elementary cycle from the same slave, they either both drop the frame or they both accept it. On the other hand, if two guardians do not receive the same frame within the same window from the same slave, they do not need to act in unison: one may drop the frame and the other may accept it, depending on whether the frame is correct or not. In fact, this is how we prevented two-faced behaviors earlier, in Section 8.6.2.

Whether a guardian is dropping a frame or accepting a frame, to make the decision, it takes into account the local master's SRDB and information about the current elementary cycle and window. Thus, for the guardians to be replica deterministic, their masters need consistent SRDBs and they need to be synchronized in time. This is the same situation as before for the internal switches. Hence, guardian replica determinism, just like internal switch replica determinism, will be a direct consequence of master replica determinism. Our main concern therefore is to make the masters replica deterministic.

Enforcing Replica Determinism between Masters

For a replica group to be replica deterministic, the replicas of the group must provide *corresponding outputs* to their users (Section 4.2). But what exactly does that mean for the embedded masters? When are masters providing corresponding outputs?

The outputs that an embedded master produces are the following:

- Trigger messages.
- Master command messages.
- The schedule for each elementary cycle.
- Updates to the SRDB.
- Timing information about when each elementary cycle and each of its windows starts and ends.

Moreover, given an embedded master m_A, its users are the following (see Figure 8.19 on page 201 for the notation):

Table 8.6: Users of the output of an embedded master m_A. See Figure 8.19 for the notation used.

Output	User	Usage
Trigger messages	S	EC synchronization, extraction of EC schedule and piggybacked master command messages.
Master command messages[a]	S	Extraction of NRDB update commands[b].
EC schedule	G_A	Dropping of unscheduled synchronous messages.
SRDB updates	G_A[c]	Dropping of messages with incorrect metadata.
	w_A[c]	Finding the subscribers to which to forward each data message.
Timing information	G_A	Dropping of untimely messages.
	w_A	Traffic shaping.

[a] Master command messages are piggybacked on trigger messages. See the section on the consistent update of system requirements on page 226.

[b] Other commands, such as those used for plug and play, are out of the scope of this dissertation.

[c] Used indirectly, by accessing the SRDB of switch A.

- The set of slave nodes S.
- The internal switch w_A.
- The set of guardians G_A.

Table 8.6 summarizes which output of a master m_A each of these users uses: trigger messages are used by the set of slaves S for elementary cycle synchronization, to find out what synchronous messages they should transmit according to the elementary cycle schedule, and to receive master command messages that are piggybacked onto the trigger messages (see the section on the consistent update of system requirements on page 226 for more on the piggybacking); the piggybacked master command messages are used by the set of slaves S to learn how they should update their node requirements databases (NRDBs) after the masters have accepted a previous update request; the schedules for each elementary cycle are used by the set of port guardians G_A to identify unscheduled messages that should be dropped; SRDB updates are used indirectly by the set of port guardians G_A when they access the updated SRDB of switch A, which in turn they use to identify messages with incorrect metadata; SRDB updates are also indirectly used by the corresponding

Table 8.7: Master output correspondency requirements in the time and value domains.

Output	Time domain	Value domain
Trigger messages	Lockstep transmission.	Same EC synchronization information, same EC schedule, and corresponding master command messages.
Master command messages	Transmission during same EC.	Identical command.
EC schedules	Produced within the same EC.	List the same synchronous message streams.
SRDB updates	Update applied at the end of the same EC.	Apply the same change.
Timing information	Agreement on when each window of each EC starts.	—

internal switch to determine the subscribers of each data message and thus how to forward such a message; the timing information about when each elementary cycle and each of its windows starts and ends is used by G_A to identify incorrectly timed messages from slaves; and, finally, the timing information is also used by the internal switch w_A for shaping the traffic, i.e., for confining messages on the downlinks to the appropriate elementary cycle windows.

The users of the output of the other master, m_B, are analogous. We can obtain a table summarizing what type of output they use by substituting each instance of the subscript A in Table 8.6 by a subscript B.

To specify what it means for the masters to provide corresponding outputs we have to specify a correspondency requirement (see Section 4.2) for each type of output, both in the time and value domain. This is summarized in Table 8.7.

A trigger message tm_A broadcast by master m_A and a trigger message tm_B broadcast by master m_B will correspond in time if

- tm_A and tm_B are broadcast in lockstep, i.e., at the same time (up to a certain precision), by m_A and m_B (this is necessary for the slaves' elementary cycle synchronization);

and they will correspond in value if

- tm_A and tm_B carry the same elementary cycle synchronization information, i.e., the same sequence number, the same trigger message interarrival time τ, the same elementary cycle length, and the same trigger message redundancy level k (this is also necessary for the slaves' elementary cycle synchronization);
- tm_A and tm_B carry the same elementary cycle schedule (this is necessary for a consistent polling of slaves); and
- if tm_A piggybacks a master command message msg_A, then tm_B piggybacks a master command message msg_B and, moreover, msg_A and msg_B satisfy the correspondency requirement defined for master command messages (this is necessary for consistent piggybacking).

A command message msg_A sent by master m_A and a command message msg_B sent by master m_B will correspond in time if

- msg_A and msg_B are transmitted during the same trigger message window (this helps ensure that all slaves can update their NRDBs at the end of the same elementary cycle);

and they will correspond in value if

- msg_A and msg_B contain identical instructions, i.e., they both contain the same instruction on how to update the NRDBs (this ensures a consistent update of the databases).

An elementary cycle schedule sched_A produced by master m_A and an elementary cycle schedule sched_B produced by master m_B will correspond in time if

- sched_A and sched_B are produced within the same elementary cycle;

and they will correspond in value if

- sched_A and sched_B list the same synchronous message streams.

The correspondency in time and value of the elementary cycle schedules ensures that the trigger messages of the two masters carry consistent schedules and that the guardians of each switch consistently drop unscheduled synchronous messages.

An SRDB update u_A applied by master m_A and an SRDB update u_B applied by master m_B will correspond in time if

- u_A and u_B are applied to the SRDB of m_A and m_B, respectively, at the end of the same elementary cycle;

and they will correspond in value if

- u_A and u_B apply the same change, e.g., they modify the same parameter of the same stream within the same table (ART or SRT).

The correspondency of SRDB updates is necessary for the masters to generate identical elementary cycle schedules, for the master command messages to carry consistent NRDB update instructions, for the guardians to consistently drop messages with incorrect metadata, and for the internal switches to consistently forward messages to the appropriate subscribers.

The timing information of master m_A and master m_B correspond if

- when master m_A signals to its corresponding internal switch and guardians a transition to a given window of a given elementary cycle within a given time interval, then m_B signals a transition to the same window of the same elementary cycle within the same time interval (the shorter the time interval, the tighter the synchronization between masters).[25]

Agreement on the timing of elementary cycles and their windows is necessary for the masters to consistently start their lockstep transmission of trigger messages, for the guardians of each switch to consistently drop untimely messages, and for the two internal switches to consistently shape the traffic on the downlinks.

Having defined the correspondency requirement for each of the outputs of the masters, the next step for ensuring that the masters are replica deterministic is to make sure that these correspondency requirements are satisfied. For this we will tackle the time and value domains separately.

[25] In Appendix B the system's timing is provided by a synchronization unit in each switch that is modeled as a separate entity from the master. To keep the explanation simpler, however, in this chapter I do not introduce a separate synchronization unit and consider it part of the master.

Enforcing Replica Determinism between Masters in the Time Domain As we saw in Table 8.7 on page 209, embedded masters will be replica deterministic in the time domain if

- they transmit trigger messages in lockstep,
- they respond to update requests by encapsulating master command messages in the trigger messages of the same elementary cycle,
- they produce a given elementary cycle schedule in the same elementary cycle,
- they update their SRDBs at the end of the same elementary cycle, and
- they agree (up to a certain precision) when each window of each elementary cycle starts.

Luckily, we can solve all these timing problems in one fell swoop by ensuring that the masters agree when each elementary cycle starts. The point is that when they agree on the start of each elementary cycle, they can use relative time offsets with respect to the start of each elementary cycle to transmit trigger messages in lockstep, to simultaneously respond to update requests, to simultaneously produce the elementary cycle schedules, to simultaneously update their SRDBs, and to agree when each window of the current elementary cycle starts.

To make the masters agree on the start of each elementary cycle we can adopt a semi-active replication approach that exploits the isochronous transmission of trigger messages.

Figure 8.20 shows how the trigger messages during each trigger message window should align for the transmissions to be isochronous and in lockstep. The figure shows the traffic as seen by a switch A, with the other FTTRS switch being switch B. The messages shown at the top, labeled as “Downlinks”, correspond to the messages that the embedded master of A broadcasts to the slaves. These same messages are also broadcast through the interlinks to the other switch. In the figure this corresponds to the row of messages labeled “Outlinks”. Finally, at the bottom of the figure we can see the messages that switch A receives from switch B through the interlinks. These are labeled as “Inlinks”.

The lockstep behavior of Figure 8.20 can be achieved by adopting a semi-active replication mechanism that takes advantage of the k trigger message replicas. We consider one of the masters the **leading master** (or **leader**) and the other the **follower master**. The leader broadcasts trigger messages isochronously, doing so according to its own internal clock, without synchronizing with anyone. The

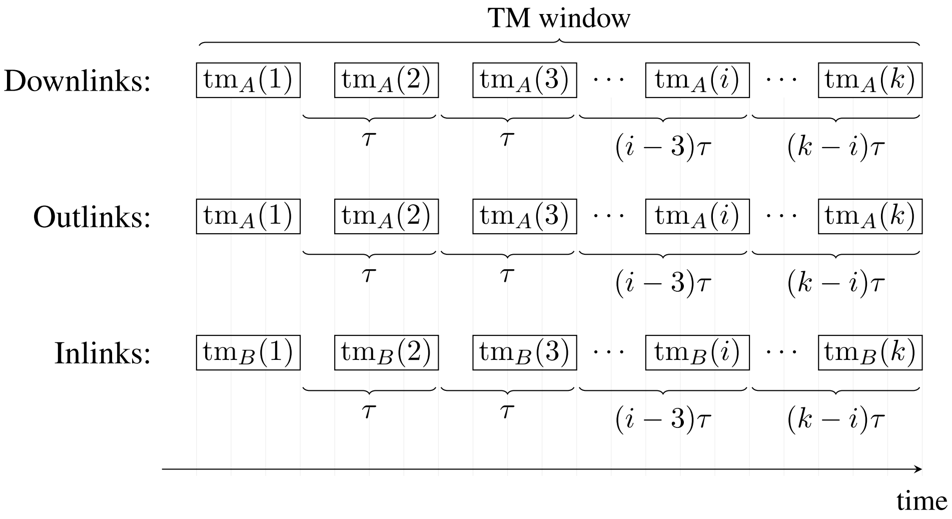

TM window
Downlinks:
$\mathrm{tm}_A(1)$ $\mathrm{tm}_A(2)$ $\mathrm{tm}_A(3)$ $\cdots$ $\mathrm{tm}_A(i)$ $\cdots$ $\mathrm{tm}_A(k)$
τ τ $(i-3)\tau$ $(k-i)\tau$
Outlinks:
$\mathrm{tm}_A(1)$ $\mathrm{tm}_A(2)$ $\mathrm{tm}_A(3)$ $\cdots$ $\mathrm{tm}_A(i)$ $\cdots$ $\mathrm{tm}_A(k)$
τ τ $(i-3)\tau$ $(k-i)\tau$
Inlinks:
$\mathrm{tm}_B(1)$ $\mathrm{tm}_B(2)$ $\mathrm{tm}_B(3)$ $\cdots$ $\mathrm{tm}_B(i)$ $\cdots$ $\mathrm{tm}_B(k)$
τ τ $(i-3)\tau$ $(k-i)\tau$
time

Figure 8.20: Traffic in FTTRS during the trigger message window, as seen by a switch A, with the other switch being B. "Outlinks" refers to the traffic going from A to B. "Inlinks" refers to traffic going in the other direction.

broadcast occurs on the slave links of the switch where the leader is located, as well as on the interlinks. The follower also broadcasts through its slave links and through the interlinks according to its own internal clock. Contrary to the leader, however, in each elementary cycle it resynchronizes its trigger message transmissions using the arrival times of the leader's trigger messages, which it receives through the interlinks. Specifically, it uses the arrival time of each received trigger message to decide whether it needs to defer or advance the start of its next trigger message transmission in order to be in sync with the leader.

This mechanism to achieve lockstep transmissions requires that the follower successfully receives *sufficient* trigger messages from the leader. What "sufficient" precisely means depends on how long it takes the follower to get out of sync if it does not synchronize its transmissions with the leader. It thus depends on the masters' time-stamping accuracy, the transmission jitter, the quality of the oscillators that drive the masters' clocks, and on the relative drift of these clocks (i.e., how quickly the frequencies of the clocks diverge from one another over time). For instance, if skipping a synchronization during one elementary cycle causes significant frequency offset between the clocks of the masters, then at least one trigger message must get through from the leader to the follower in each elementary cycle. If, on the other hand, a significant frequency offset accumulates only after, let us say, 10 elementary cycles, then in principle—and for synchronization purposes only—it suffices for the follower to receive a trigger message every tenth elementary cycle (for replica determinism in the value domain, however, a message must get through during each and every elementary cycle, as we will see shortly).

We should point out, however, that this follower/leader approach would not work if the clocks of the masters can diverge significantly during the time spanning the turnaround time, synchronous window, asynchronous window, and guard window (see Figure 8.10 on page 187). This is so because the follower only resynchronizes during the trigger message window. Luckily, a significant loss of synchronization is unlikely if we ensure that the fail-silent switches include mechanisms to make the oscillators fail silent[26] and these oscillators are of sufficient quality. After all, for a realistic application we can expect the duration of elementary cycles to be on the order of at most a few milliseconds, which is not enough time for correctly functioning clocks to diverge significantly. For instance, even if we use a crystal oscillator with an accuracy of only 20 ppm (parts per million), the accumulated error after the end of a 1 ms (1 000 000 ns) elementary cycle would only be ± 20 ns. This means that after a perfect synchronization at the beginning of one elementary cycle, by the end of the 1 ms elementary cycle the clocks of the two masters would have diverged at most $2 \cdot 20 = 40$ ns. If this is too much, higher-quality oscillators can be used, such as temperature compensated crystal oscillators, which perform significantly better. Moreover, if the oscillators are not of sufficient quality, this may be compensated by shortening the duration of the elementary cycles. This increases the number of trigger message windows per unit of time and thus the frequency with which the follower resynchronizes.

Ensuring that for synchronization purposes sufficient trigger messages get from the leader to the follower, despite transient link faults, is just a matter of appropriately setting the value of the trigger message redundancy level k. In fact, because we already made the trigger message fault tolerant by choosing a sufficiently high value for k to tolerate transient faults (Section 8.7.2), we already ensured that the follower can resynchronize in each and every elementary cycle. Resynchronization can then only fail to occur when either switch crashes or all interlinks are affected by permanent faults. In case a switch crashes, FTTRS tolerates this transparently since the surviving switch will simply go on transmitting trigger messages according to its own clock. If the loss of resynchronization is due to permanent faults affecting all interlinks, with both switches still operating, then the network will be partitioned in two. In that case maintaining synchronization is impossible. The problem can be

[26] Ferreira discusses a mechanism for CAN that might be adapted to Ethernet. See Ferreira, "Fault-Tolerance in Flexible Real-Time Communication Systems", Sec. 7.3 — Internal replication and temporized agreement.

mitigated by ensuring that there are enough interlinks.

Based on this general idea of using a leader/follower approach for achieving lockstep synchronization, Alberto Ballesteros worked out the details.[27]

The solution Ballesteros proposed involves two phases:[28] an initialization phase, called **rendezvous phase**, and a subsequent normal operation phase, called **periodic lockstep resynchronization**.

The rendezvous phase is based on the IEEE 1588 standard, also known as the Precision Time Protocol (PTP).[29] It essentially involves a two-way time transfer, which is not only used by PTP, but also by other clock synchronization protocols such as the Network Time Protocol (NTP).[30] The periodic lockstep resynchronization, on the other hand, is based on the leader-follower approach I just described. For the full details, please refer to Ballesteros et al.'s paper.[31]

Although we used semi-active replication, which often has a non-zero failure recovery time (Section 4.4.5 on page 89), in this case the recovery time is zero, i.e., seamless. This is so because we have no need for a time-intensive leader election process. After all, we only have two switches and therefore there is only one choice when the leader fails.

Having chosen a mechanism to enforce replica determinism in the time domain, we can now move on to the value domain.

Enforcing Replica Determinism between Masters in the Value Domain With the rendezvous phase and the periodic lockstep resynchronization, switches can be made replica deterministic in the time domain: they can transmit trigger messages in lockstep, encapsulate command messages in trigger messages within the same elementary cycle, produce schedules during the same elementary cycle, apply SRDB updates at the end of the same elementary cycle, and agree on when each window of each elementary cycle starts. To make them fully replica deterministic we now also have to ensure replica determinism in the value domain.

[27] I can take full credit for the general idea of using a leader/follower approach, but not for the subsequent details that were worked out by Ballesteros.

[28] Alberto Ballesteros et al. "Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE, 2014.

[29] Institute of Electrical and Electronics Engineers (IEEE). *Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. 1588-2008 (IEEE). 2008.

[30] Jean-Loup Ferrant et al. *Synchronous Ethernet and IEEE 1588 in Telecoms: Next Generation Synchronization Networks*. John Wiley & Sons, 2013, Sec. 2.3.2 Packet-based methods.

[31] Ballesteros et al., "Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star for Ethernet".

If we refer back to Table 8.7 on page 209 once again, we will recall that embedded masters will be replica deterministic in the value domain if

V1: in lockstep broadcast trigger messages carry

(a) the same elementary cycle synchronization information,

(b) the same elementary cycle schedule, and

(c) corresponding master command messages;

V2: corresponding master command messages, i.e., those that are piggybacked on trigger messages of the same elementary cycle, carry identical commands;

V3: elementary cycle schedules produced during the same elementary cycle list the same synchronous message streams; and

V4: SRDB updates applied at the end of the same elementary cycle apply the same change to an SRDB.

Having in lockstep transmitted trigger messages carry the same elementary cycle synchronization information (V1a) is easy since the only relevant information that is dynamic are sequence numbers, while the rest—the elementary cycle length, the trigger message intertransmission time, and the trigger message redundancy level—are static. We thus only have to ensure two things. First, the masters need to be initialized with the same contents in their system configuration and status records (SCSRs), which is where the static parameters are stored. Second, the masters must update the sequence numbers in the same deterministic way. This again is easy because the sequence numbers simply have to be reset at the beginning of each elementary cycle and increased by 1 after each trigger message broadcast. Together with the replica determinism in the time domain, this ensures that corresponding trigger messages will carry the same elementary cycle synchronization information.

Meeting the other correspondency requirements (V1b, V1c, V2, V3, and V4), in contrast, is harder. But—assuming that the masters are internally deterministic and start out with identical SRDB contents—they can all be addressed by addressing the last correspondency requirement (V4). This is so because if the last requirement is satisfied, i.e., if corresponding SRDB updates apply the same change, then the SRDBs will be kept consistent at all times given that they started out with the same contents. If, in turn, the SRDBs are kept consistent at all times, then, when the masters compute the schedule in the same elementary cycle (thanks to the replica determinism in the time domain), they will obtain the same result. Thus, when the masters generate the trigger messages (in lockstep thanks to the replica determinism in the time domain), they will encapsulate the same schedule within their trigger messages and meet requirement V1b. Also, since they deterministically compute schedules during the same elementary cycle from the same SRDB contents, the masters will also list the same synchronous message streams in their schedules and also meet requirement V3. Finally, if the masters agree on how to update their SRDBs, they will also agree on how slaves should update their NRDBs in turn to keep them consistent with the SRDBs. Thus, they will agree on the master command messages to send. Hence, by keeping the SRDBs consistent we will also be ensuring requirement V2 and, thus, requirement V1c.

Our goal then is to meet the last correspondency requirement and the others will follow.

How, then, can we ensure that masters apply the same SRDB updates? Well, we can exploit the following facts: the masters are fail silent, free from any internal non-determinism, and the contents of the SRDBs can only be modified upon the reception of update requests from slaves. We can therefore use the following two-pronged approach. First, as mentioned before, we ensure that the masters start with consistent SRDBs. Second, we ensure that when one or more slaves send update requests, the masters agree on which one to subject to admission control next and at what time. From then onwards the internal determinism will ensure that both masters either accept or reject the request, and, if they accept the request, that the SRDBs are updated consistently.

Ensuring that the masters start with consistent SRDBs is simply a matter of properly preconfiguring the masters and ensuring that no update requests are processed before the very first elementary cycle, which starts when the follower has synchronized with the leader (as described earlier when we mentioned the rendezvous phase).

Ensuring that the masters agree on what update request to process next is more difficult. One solution is to use active replication with total-order broadcast (Section 4.4.1 on page 83). In that case, the slaves, in collaboration with the masters, would execute a total-order broadcast protocol that ensures that both masters always deliver to admission control the same update requests in the same order. The other solution is to use active replication with state reconciliation (Section 4.4.1 on page 82), which does not rely on the slaves to achieve consistency. Both approaches can work. Nevertheless, considering that switches have more benign failure semantics than slaves, it is safer to only rely on the fail-silent switches to enforce replica determinism among themselves and to not involve the more malicious slaves in this critical task; that is, it is safer to use active replication with state reconciliation.

If we use active replication with state reconciliation, some update requests might reach one master, but not the other. This is so because slaves can produce inconsistent omissions (Figure 8.9 on page 179). Thus, master m_A might receive a set of update requests Q_A, while master m_B receives a set of update requests Q_B, with $Q_A \neq Q_B$. Given that they have different sets of pending update requests, the masters must agree on which update request to process next among the ones in $Q_A \cup Q_B$. This is an instance of the **coordinated attack problem**, also known as the **two generals' problem**. The problem can be stated as follows:[32]

> Two divisions of an army, each commanded by a general, are camped on two hilltops overlooking a valley. In the valley awaits the enemy. It is clear that if both divisions attack the enemy simultaneously they will win the battle, while if only one division attacks it will be defeated. As a result, neither general will attack unless he is absolutely sure that the other will attack with him. In particular, a general will not attack if he receives no messages. The commanding general of the first division wishes to coordinate a simultaneous attack (at some time the next day). The generals can communicate only by means of messengers. Normally, it takes a messenger one hour to get from one encampment to the other. However, it is possible that he will get lost in the dark or, worse yet, be captured by the enemy. Fortunately, on this particular night, everything goes smoothly. How long will it take them to coordinate an attack?

In our case, the two generals are the masters, the valley is the set of unreliable interlinks, and the time of attack is what update request to submit to admission control next.

The answer to the question posed by the problem (how long will it take the generals to coordinate an attack?) is that they will never agree on a time of attack, even if no messenger is ever lost. This might be surprising, but it follows from the requirement of absolute certainty: "neither general will attack unless he is *absolutely sure* that the other will attack with him". If general A sends general B a message saying "attack at dawn", A cannot attack until he is sure that B received the message. Trying to remedy the situation, B might send a messenger back with an acknowledgment. But even if A receives this acknowledgment, neither general can attack. After all, now it is B who is unsure whether his acknowledgment was delivered. A might then send an acknowledgment of the acknowledgment. But this does not improve the situation either: now it is A again who is worrying about whether his latest message got through. Indeed, regardless of how many acknowledgments are successfully delivered, coordination cannot be attained.

[32] Ronald Fagin et al. *Reasoning about knowledge*. The MIT press, 1995, pp. 190-191.

Since the coordinated attack problem has no solution, does this mean we cannot achieve replica determinism in the value domain? Strictly speaking, yes. But there is no need to despair. "Communication is one of those delightful things that work [...] in practice"; even though "in theory it's impossible".[33]

The key is that in practice we do not need absolute certainty. We only need to be sufficiently certain that the masters will agree on what update request to process next. What "sufficient" means can be quantified in terms of the desired level of reliability. For instance, if we want a reliability of 0.9999 for a mission time of 1 hour, then the probability of a system failure must be below 0.0001 during that hour and, consequently, the probability of not agreeing must also be below 0.0001.

How, then, can the coordinated attack problem be solved in practice? One solution is for a general to send a whole flood of messengers through the valley so that it becomes exceedingly unlikely for all of them to be captured. This is the solution we shall adopt by using, once again, proactive retransmissions. Specifically, by adjusting the redundancy level of the messages that masters exchange among themselves to agree on what update request to process next, we can make the probability of not agreeing, and thus losing replica determinism, as low as necessary.

To recapitulate, we will make the masters replica deterministic in the value domain by ensuring that they start out with identical SRDB contents and then, when slaves sent update requests, by making them agree on which request to process next at what time. This will ensure that the masters have consistent SRDBs whenever they are about to generate an output—which may be a trigger message, master command, or elementary cycle schedule. And since these outputs will be derived deterministically from the contents of the SRDB and the agreed-upon update request, the outputs will correspond, making the masters replica deterministic in the value domain as well.

Now, before we tackle the specifics of how to make the masters agree on what update request to process next, it will be helpful to put this mechanism in the context of how to achieve a system-wide update of the real-time requirements, i.e., in the context of how to consistently update not only the SRDBs, but the NRDBs as well.

Figure 8.21 summarizes the whole process. In an elementary cycle i (labeled "EC i" in Figure 8.21) one or more slaves send an update request. In the next elementary cycle (EC $i + 1$), the masters solve the coordinated attack problem to agree on what update request to submit to admission control next. As we can see in the figure, and as I will explain in the next subsection in more detail, this agreement is reached during the trigger message window (TMW) of elementary cycle $i + 1$. Once the trigger message window is over and the masters have agreed on what update

[33] Brian Christian and Tom Griffiths. *Algorithms to Live By: The Computer Science of Human Decisions*. Henry Holt and Co, 2016, p. 210.

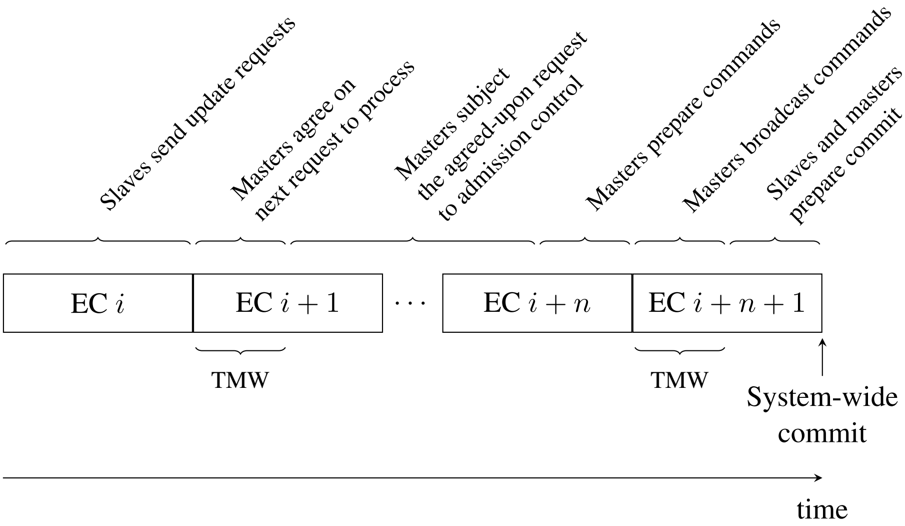

Slaves send update requests
Masters agree on next request to process
Masters subject the agreed-upon request to admission control
Masters prepare commands
Masters broadcast commands
Slaves and masters prepare commit
EC i
EC $i+1$
$\cdots$
EC $i+n$
EC $i+n+1$
TMW
TMW
System-wide commit
time

Figure 8.21: Phases of a system-wide update of the real-time requirements. The last phase, where slaves and masters prepare the commit, as well as the system-wide commit, are only executed if the request that was subjected to admission control was accepted.

request to process, the masters deterministically subject the agreed-upon request to admission control. The time to carry out the admission control may vary depending on the performance of the masters, the size of the SRDBs, the implementation of the scheduler, and other factors (in Figure 8.21 it takes n elementary cycles, from EC $i + 1$ to EC $i + n$). From the communication point of view the amount of time taken for the admission control does not matter as long as the masters complete it during the same elementary cycle. Once the admission control is completed (EC $i + n$), the masters will reach the same conclusion because they implement the same admission control algorithm, are internally deterministic, and started out with consistent SRDBs: the request is either accepted or rejected. Either way, the masters prepare command messages to be broadcast during the next trigger message window, where they will be piggybacked within trigger messages. These command messages will tell the slaves if and how to update their NRDBs. Moreover, they will be consistent among the two masters because they are computed deterministically based on the consistent SRDBs and the agreed-upon request. In the next elementary cycle (EC $i + n + 1$) at least one copy of the master command will reach each correct slave since trigger messages, which carry the commands, are fault tolerant thanks to being retransmitted proactively. At this point both the masters and the slaves will know how to update their SRDBs and NRDBs, respectively. They will thus prepare an appropriate database commit and execute it at the end of the current elementary cycle (again, EC $i + n + 1$). In this way a system-wide update of the real-time requirements is achieved.

Note that the steps that in Figure 8.21 take up elementary cycles $i + 1$ through $i + n$ could, in principle, be carried out in a single elementary cycle if the masters have sufficient computing power. The whole process, from request to system-wide commit, therefore consumes at least three elementary cycles. Nevertheless, we should also note that several of these phases can be executed in parallel for different requests. For instance, while during one trigger message window the masters agree on the next request to process, they can simultaneously carry out the admission control of an earlier request, and broadcast commands corresponding to an even earlier request. Moreover, the system-wide update process can also be implemented such that masters agree on processing multiple requests at once.

Now that we have the context within which masters must solve the coordinated attack problem, i.e., within which they have to agree on what update request to process next, let us discuss how exactly we can solve the problem.

Making Masters Agree on What Update Request to Process Next In FTT the update requests from a given slave are messages belonging to a single asynchronous stream.[34] This means that they have, like all asynchronous messages, a minimum interarrival time that is a multiple of the elementary cycle length. Thus, a slave can only request one change per elementary cycle and not two or more different changes within the same elementary cycle. Furthermore, in FTT there are no deadlines for when an update request must have been processed by the masters (the deadlines specified in the asynchronous requirements table, ART, are message exchange deadlines, which is different). Neither does the FTT paradigm specify in which order update requests must be processed by a master if several need to be subjected to admission control. Given all this, we can do the following to ensure that masters process update requests consistently, by which I mean that they agree on which update request to submit to admission control next and then either both reject the request or both accept it.

First, we define an a priori total order relation for the update requests. This means that update requests have an intrinsic order: for any two requests, a master can always tell which comes before another. There are different ways of defining such a relation and for consistency among masters the details do not matter. All that matters is that the relation is indeed a total order. As an example, if we have

[34] At least this is the case in FTT-SE, where update requests belong to a dedicated slave-master stream. See Ballesteros and Proenza, *A Description of the FTT-SE Protocol*, p. 12.

three slaves s_1, s_2, and s_3, we could define the total order as follows: all update requests a master receives from s_1 come before the ones by s_2, all the ones from s_2 come before the ones by s_3; and if a master receives two update requests from the same slave, the one transmitted in an earlier elementary cycle comes before another transmitted in a later elementary cycle (the order of transmission could be determined by adding sequence numbers to the update requests, with replicated update requests all having the same sequence number[35]). Other ways of defining a total order for update requests may be possible as well. Again, what is important is that there is a total order, but at the level of the communication subsystem the details of how exactly the order is defined do not matter.

With a total order, a master can always compare arbitrary update requests and unambiguously determine which among them is the *minimum* update request according to the total order. The idea then is the following. Each master collects the update requests it receives, doing so in a **set of pending update requests**. Each master then exchanges with the other the request that it determines to be the minimum within its set of pending requests, which we may call the given master's **local minimum**. Thus, if the set of pending requests of master m_A is Q_A and the one of master m_B is Q_B, then m_A sends its local minimum $\min(Q_A)$ to m_B and m_B sends its local minimum $\min(Q_B)$ to m_A. After this exchange, which must be fault tolerant, the set of pending requests of m_A becomes $Q'_A = Q_A \cup \min(Q_B)$ and the set of pending requests of m_B becomes $Q'_B = Q_B \cup \min(Q_A)$. The result of this is that now $\min(Q'_A) = \min(Q'_B)$. Hence, if now m_A subjects to admission control $\min(Q'_A)$ and m_B subjects to admission control $\min(Q'_B)$, they will be processing the exact same request, which we may call the **global minimum**. Having decided on whether to accept or reject the global minimum, they can then take measures to consistently update their SRDBs together with the NRDBs of the slaves.

For this mechanism to work, it is not only important that the update requests have a total order and that the local minimums are exchanged reliably between the masters, but that the local minimums are chosen and exchanged during the same elementary cycle. Moreover, we must ensure that between the time that masters exchange the local minimum from their respective sets of pending requests and the time they choose the global minimum, no new requests are added to either master's set of pending requests. We can achieve all this by piggybacking the local minimums on the fault tolerant and tightly synchronized trigger messages, while at the same time ensuring that during the trigger message window the internal switches do not forward any new requests to their respective embedded masters (internal

[35] Update requests are not only replicated spatially when a slave transmits them in parallel through slave links, but may also be replicated in time because, like all asynchronous messages, they have a redundancy level.

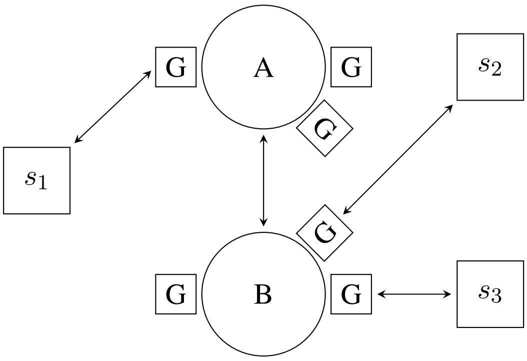


Figure 8.22: Still functioning FTTRS architecture after the loss of several links.

switches do precisely that as we can see if we refer back to the fifth and sixth row of Table 8.5 on page 202). That way, the local minimums are reliably exchanged during the trigger message window and then, during the turnaround time, the masters can consistently select the global minimum, which is then the update request each of them subjects to admission control next.

To illustrate the mechanism, let us consider a concrete example. Specifically, to keep it simple, let us consider an FTTRS network where several links have already failed, but communication is still possible among all slaves. Figure 8.22 shows the network. We have three slaves s_1, s_2, and s_3 and two FTTRS switches A and B. Slave s_1 has a link to switch A, while the two other slaves have links to switch B. With this topology in mind, let us now look at Figure 8.23. This second figure illustrates our consistency mechanism.

The figure shows the trigger messages and update requests exchanged during an elementary cycle i and during the trigger message window of the subsequent elementary cycle $i + 1$. Other messages are not shown to not further complicate the figure. The top row of messages, labeled $dl_A(s_1)$, denotes the messages that slave s_1 receives from switch A through its downlink. The next two rows, labeled $dl_B(s_2)$ and $dl_B(s_3)$, denote the messages that slaves s_2 and s_3 receive through their downlinks from switch B, respectively. The next three rows show the messages transmitted by the slaves through their uplinks: the label $ul_A(s_1)$ refers to the messages that s_1 sends through its uplink to switch A; and the labels $ul_B(s_2)$ and $ul_B(s_3)$ refer to the messages that s_2 and s_3 send through their uplinks to switch B, respectively. The label il_{AB} refers to the traffic that switch A sends through the interlink to switch B, and the label il_{BA} refers to the traffic in the opposite direction. Next, we find two rows, labeled Q_A and Q_B, which denote the contents of the sets of pending update requests of the masters m_A and m_B at different points in time.

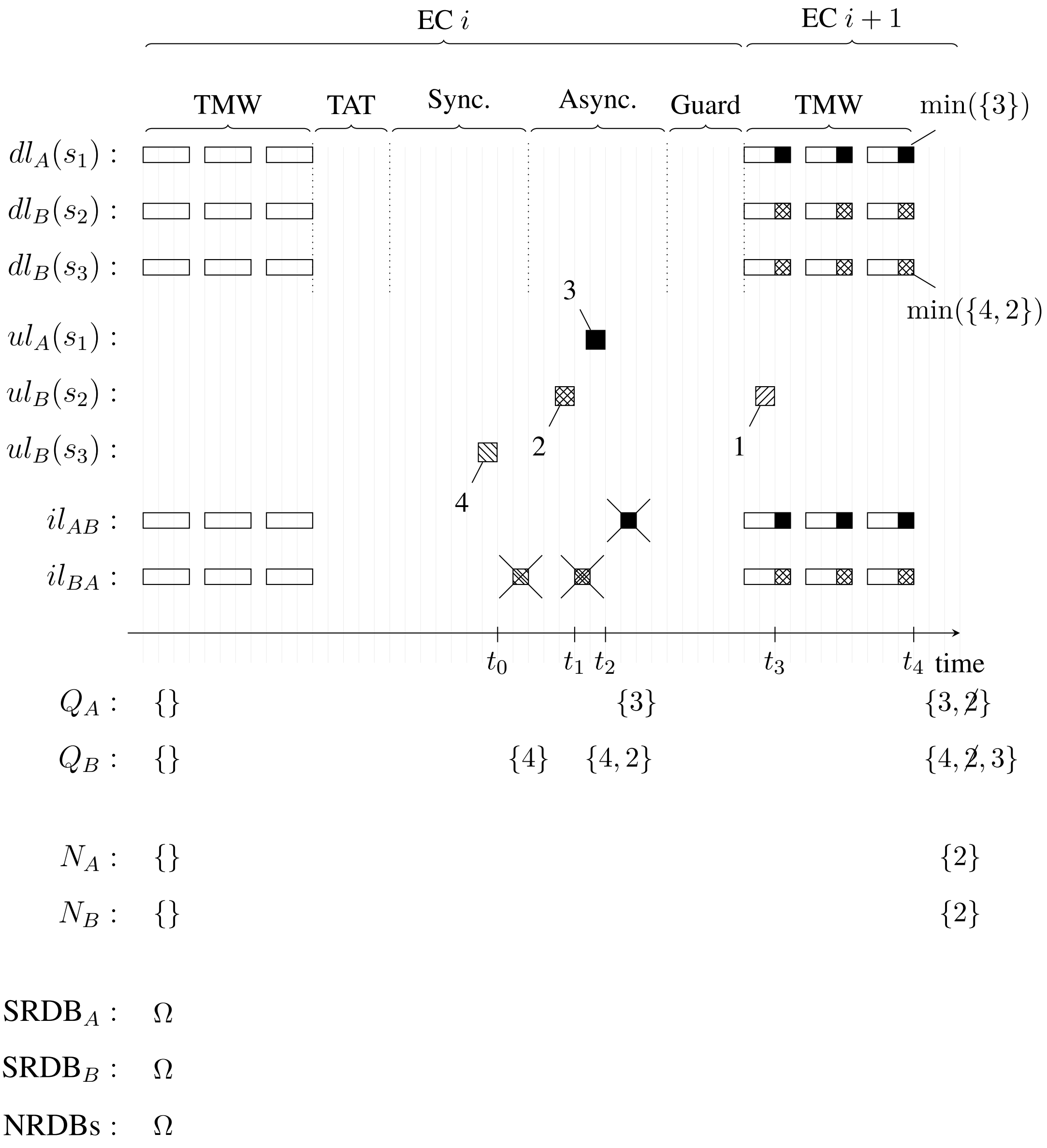


Figure 8.23: Example of exchanging minimum update requests to ensure that masters agree on what update request to subject to admission control next. Refer to the text for the details.

Below these, we find another two rows, labeled N_A and N_B; these denote updates that the masters m_A and m_B, respectively, have accepted to submit to admission control. Finally, at the bottom of the figure we find two rows showing the contents of the SRDB of each master and one row denoting the contents of the NRDBs (there is actually one NRDB per slave, but the figure shows only one for simplicity).

As we can see, at the beginning of the elementary cycle i the sets of pending update requests and of requests to process next are empty for both switches; that is, $Q_A = Q_B = N_A = N_B = \emptyset = \{\}$. Moreover, in all elementary cycles before the ith the switches and slaves have not lost consistency and hence their SRDBs and NRDBs have the same contents, denoted Ω. At time t_0 an update request (labeled 4) reaches switch B through uplink $ul_B(s_3)$. We assume this update request is in the fourth position according to the total order and we can therefore refer to it by its ordinal number, i.e., by the number 4. Hence, at time t_0 update request 4 is added to the pending requests of B and Q_B becomes $\{4\}$. Switch B forwards request 4 through the interlink il_{BA}, but due to a transient fault it is lost (this occurs shortly after t_0). Thus, Q_A remains empty and the pending requests of the two switches have diverged: $Q_A \neq Q_B$. Similarly, another update request, this time with ordinal number 2, reaches B from slave s_2, doing so at time t_1 through uplink $ul_B(s_2)$. This one also fails to reach A after being forwarded by B through il_{BA}. Hence, shortly after time t_1 we already have that $Q_B = \{4, 2\}$, while Q_A is still empty. At time t_2 another update request (labeled 3) reaches a switch, but this time it reaches A instead of B. This request has ordinal number 3 and, unfortunately, is also lost on the interlink. Now we have that $Q_A = \{3\}$ and $Q_B = \{4, 2\}$. No further requests arrive until the next elementary cycle.

The next elementary cycle, EC $i + 1$, begins as usual with a trigger message window. Hence, each switch begins to transmit k copies of the trigger message, with $k = 3$ in this example. Contrary, however, to the previous elementary cycle, when the trigger message window begins in EC $i + 1$, the switches do have pending requests. Specifically, at the beginning of the trigger message window of EC $i + 1$ switch A has $Q_A = \{3\}$ and switch B has $Q_B = \{4, 2\}$. Switch A selects its local minimum, i.e., $\min(\{3\}) = 3$, and piggybacks it on its trigger messages. Switch B does the same and thus piggybacks $\min(\{4, 2\}) = 2$ on its trigger messages. During the trigger message window, at time t_3, another update request reaches switch B. As the trigger message window has already started, the request is put on hold and not forwarded through the interlinks nor added to the sets of pending requests. At the end of the trigger message window, at time t_4, each switch adds the piggybacked local minimum it received from the other switch to its own set of pending requests. That is, m_A adds $\min(\{4, 2\}) = 2$ to Q_A and m_B adds $\min(\{3\}) = 3$ to Q_B. Thus, after time t_4 we have that $Q_A = \{3, 2\}$ and $Q_B = \{4, 2, 3\}$. Now the master embedded in each switch accepts for admission control the global minimum $2 = \min(Q_A) = \min(Q_B)$, thereby removing the request from their respective set of pending requests. In the figure this is represented by request 2 being crossed out in Q_A and Q_B and being added to the set of updates that are ready for admission control, resulting in $N_A = N_B = \{2\}$. With this, both masters have agreed at the end of the same elementary cycle to subject request 2 to admission control next. The next step is to carry out the admission control and, if the request passes it, to update the SRDBs and NRDBs. Let us see how this can be done.

Consistently Updating the System Requirements across the Whole System To update the system requirements across the whole system, masters must not only agree among themselves what update to submit to admission control and then incorporate it into their SRDBs if it passes, but they must also notify the slaves if they have to do a corresponding update of their NRDBs. In FTT this notification is done by means of master command messages (Figure 7.3 on page 126): upon processing an update request, a master generates a command message that tells the slaves to either update their NRDBs according to the accepted request or, if the request was rejected, to reject the update.[36]

In classical FTT, master command messages are asynchronous messages. As such they may be delivered to different slaves at different points in time within the same elementary cycle because of varying queuing delays in the switch output ports. Thus, if the masters simply broadcast their decision to reject or accept a request using such messages, slaves may process them at different times within an elementary cycle and consequently some of the slaves may not have enough time to prepare a corresponding database commit before the end of the current elementary cycle (see Figure 8.21 on page 220 again). As a result, slaves may have different versions of the system requirements active simultaneously in the next elementary cycle. In principle we could, as in FTT-SE, add further mechanisms to deal with different concurrent database versions.[37] For instance, we could enhance each port guardian to be aware of what version of the system requirements is active within the slave whose failure semantics it is restricting. However, to keep things simple, and thus more robust to failures, let us only allow one version of the system requirements to be active at a time.

If we only allow one version at a time, all slaves and masters must commit the

[36] Actually, in HaRTES a rejection is not communicated to the slaves. But this, as pointed out earlier (footnote 7 on page 164), is a flaw.

[37] FTT-SE employs a tagging mechanism that essentially labels different message streams as belonging to one version of a database or another. See Marau, "Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management", p. 71.

same changes to their databases at the end of the same elementary cycle. We can achieve this by once again piggybacking the necessary information onto trigger messages. After all, we have designed trigger messages to be broadcast by the masters in lockstep and to be processed by all slaves during the same elementary cycle. Moreover, trigger messages are the earliest messages transmitted in each elementary cycle, which gives slaves the maximum possible time for preparing a database commit before the next elementary cycle starts.

The idea then is the following. First, using the minimum update request exchange mechanism we saw before, the masters agree on the update to add to the list of requests to submit to admission control. Having agreed on which update to check for admission next, they do so. This may, in principle, and as illustrated in Figure 8.21 on page 220, take several elementary cycles. If the update was accepted, the next step is to consistently apply the update across all SRDBs and NRDBs. Thus, when the masters have accepted the update in a given elementary cycle, they prepare appropriate command messages and then wait until the next elementary cycle. In the next elementary cycle they broadcast their trigger messages as usual, but embedding within them the prepared update commands. Since the trigger messages are fault tolerant and processed by all slaves at the end of the trigger message window, during the subsequent turnaround time all slaves will extract the update command, begin preparing database commits, and respond to the trigger message polling based on their not-yet-updated NRDBs. Then, at the end of the elementary cycle, all slaves and the masters update their databases, upon which we will have achieved our goal: the system requirements are consistently updated across the whole system at the same time.

Figure 8.24 is a continuation of Figure 8.23 that illustrates how this works. The first elementary cycle shown is EC $i + 1$, which was the second elementary cycle in Figure 8.23. The trigger message window of that elementary cycle ended with the two switches agreeing to subject to admission control update 2. Hence, after that trigger message window, i.e., after t_4 in Figures 8.23 and 8.24, we have that $N_A = N_B = \{2\}$. The masters now subject this update request to admission control. In Figure 8.21 this took several elementary cycles. In Figure 8.24, in contrast, we assume that the admission control is completed within the same elementary cycle, at time t_6. Moreover, we assume that the request was accepted. Thus, at time t_6 the update request 2 is deleted from the set of requests to be subjected to admission control, i.e., it is deleted from N_A and N_B, and added by each master to the sets U_A and U_B, respectively, which represent the sets of accepted updates. Next, the masters each prepare an appropriate command message, which they complete by t_7. Then, during the trigger message window of elementary cycle $i + 2$, the masters broadcast the command within trigger messages. The slaves, who receive the

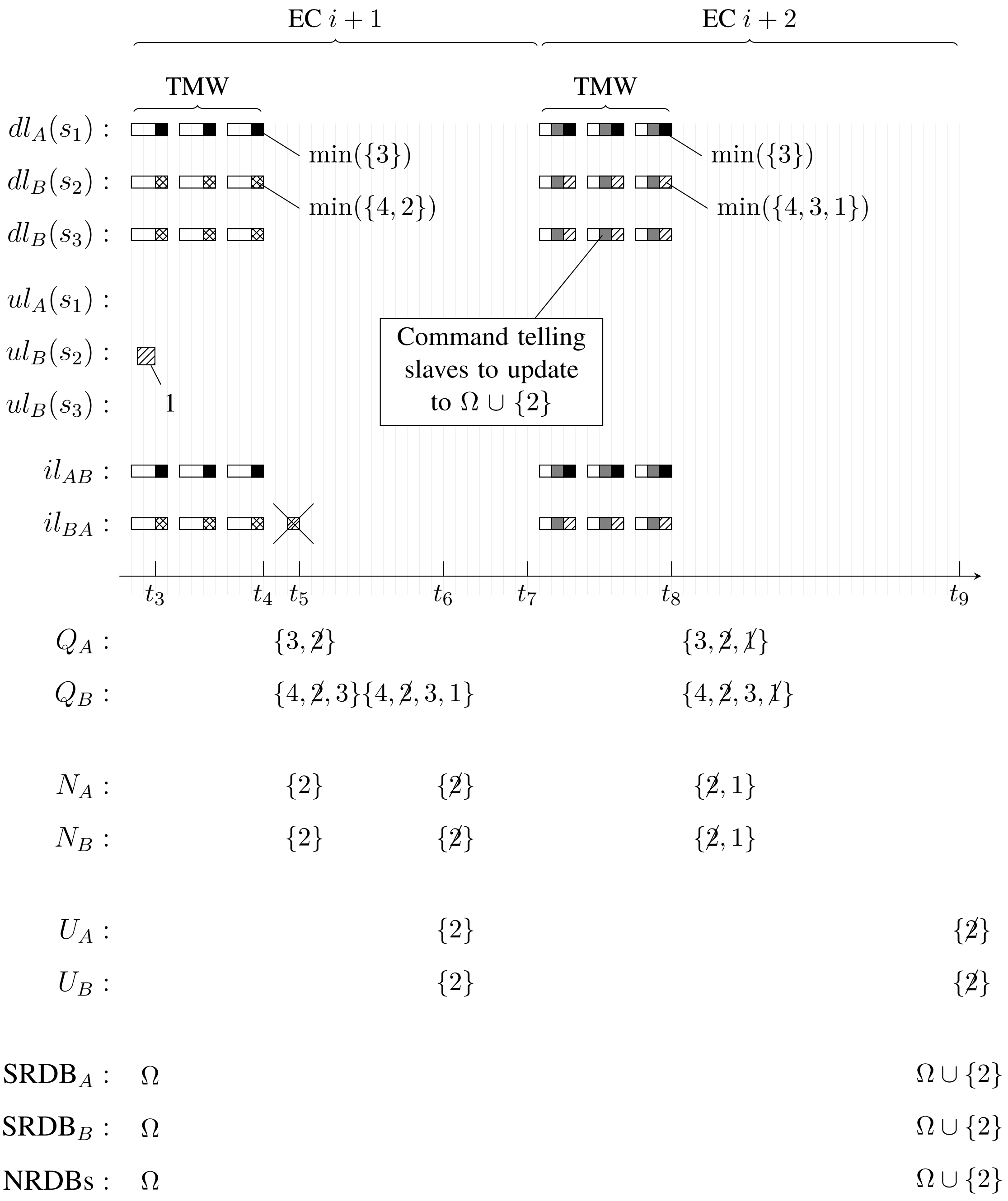

EC $i+1$
EC $i+2$
TMW
TMW
$dl_A(s_1)$:
$dl_B(s_2)$:
$dl_B(s_3)$:
$ul_A(s_1)$:
$ul_B(s_2)$:
$ul_B(s_3)$:
il_{AB} :
il_{BA} :
min({3})
min({4, 2})
min({3})
min({4, 3, 1})
1
Command telling slaves to update to Ω ∪ {2}
t_3 t_4 t_5 t_6 t_7 t_8 t_9
Q_A : $\{3, \cancel{2}\}$ $\{3, \cancel{2}, \cancel{1}\}$
Q_B : $\{4, \cancel{2}, 3\}\{4, \cancel{2}, 3, 1\}$ $\{4, \cancel{2}, 3, \cancel{1}\}$
N_A : $\{2\}$ $\{\cancel{2}\}$ $\{\cancel{2}, 1\}$
N_B : $\{2\}$ $\{\cancel{2}\}$ $\{\cancel{2}, 1\}$
U_A : $\{2\}$ $\{\cancel{2}\}$
U_B : $\{2\}$ $\{\cancel{2}\}$
SRDB$_A$: Ω Ω ∪ {2}
SRDB$_B$: Ω Ω ∪ {2}
NRDBs : Ω Ω ∪ {2}

Figure 8.24: Example illustrating a consistent update of the system requirements across the whole system. See the text for the details.

command, as well as the masters, then begin to prepare the appropriate database commit shortly after time t_8 and, at the end of elementary cycle $i+2$, at time t_9, they execute the commit. The result is that at time t_9 the whole system has agreed on the new system requirements, denoted as $\Omega \cup \{2\}$ in Figure 8.24.

Figure 8.24 also shows that the update request 1, which was put on hold during the trigger message window of elementary cycle $i+1$, is processed by switch B after that window, at time t_5. Specifically, at that point in time request 1 is forwarded by the internal switch w_B to master m_B and through the interlink—where it is corrupted by a transient fault. The request is thus only added to the set of pending requests of m_B, with Q_B becoming the set $\{4, \cancel{2}, 3, 1\}$ shortly after t_5. Moreover, the figure also shows how after the trigger message window of elementary cycle $i+2$, at time t_8, the two switches agree on subjecting to admission control update request 1. That is, during the trigger message window of elementary cycle $i+2$ master m_A broadcasts its local minimum $\min(\{3\}) = 3$ and m_B broadcasts its local minimum $\min(\{4,3,1\}) = 1$. At the end of the trigger message window, at time t_8, they both add the minimum of the two proposed requests, i.e., $\min(3,1) = 1$, to their respective set of pending update requests. Thus, Q_A becomes $\{3, \cancel{2}, 1\}$ and Q_B becomes $\{4, \cancel{2}, 3, 1\}$. Finally, both masters agree that the global minimum is $\min(Q_A \cup Q_B) = 1$, which is subsequently removed from Q_A and Q_B and added to both N_A and N_B. Thus, at t_8 Q_A becomes $\{3, \cancel{2}, \cancel{1}\}$, Q_B becomes $\{4, \cancel{2}, 3, \cancel{1}\}$, N_A becomes $\{\cancel{2}, 1\}$, and N_B becomes $\{\cancel{2}, 1\}$.

As to why the update request 1 is not added to U_A nor U_B by the end of elementary cycle $i+2$ in Figure 8.24, this is to illustrate two possible scenarios: either update request 1 was rejected by the admission control or the admission control is not completed by the end of elementary cycle $i+2$.

8.7.5 Further Considerations

We have now essentially seen all the fault-tolerance mechanisms of FTTRS. But before we move on to the final discussion of the chapter, let us just briefly consider two minor issues.

Dealing with an Unexpected Loss of Replica Determinism

The above mechanisms all rely on the trigger message redundancy k having a sufficiently high value for at least one copy of a given trigger message to reach its destination even in the presence of several transient faults in the links. However, since in practice the value of k cannot be set arbitrarily high, this means that it may

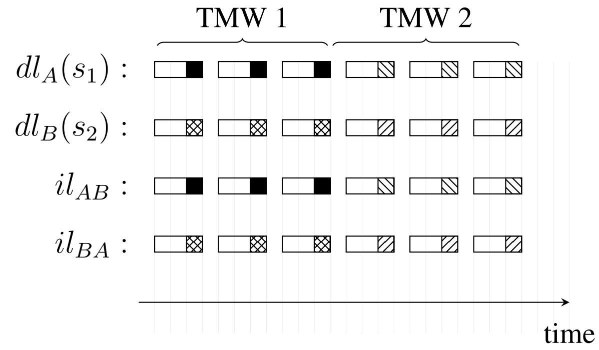


Figure 8.25: Increasing the size of the trigger message window to increase the amount of information that can be reliably broadcast with precise timing.

still be possible for an unexpectedly long error burst to corrupt all k replicas of a trigger message. This would lead to a loss of replica determinism of the masters. Thus, for FTTRS to continue to operate, we need an alternative approach (a "plan B") in case k turned out to have a too low value. We can deal with such a situation by having the slaves favor the leading master's opinion over the follower master's in case of loss of replica determinism, i.e., when they receive conflicting trigger messages. Note that the follower can detect the loss of replica determinism as long as it receives the trigger messages from the leader through the interlinks. In that case, we could force the follower to shut down its service to prevent any potential interference with the leader's service.

Optional Increase of the Trigger Message Window

To consistently update the system requirements we have relied on the piggybacking of minimum update requests and master command messages on trigger messages. This made the exchange of minimum update requests and master command messages fault tolerant and ensured that they reached all slaves quasi-simultaneously and at precise instants of time. But what if we want to piggyback more messages than can fit into a trigger message? For instance, what if we want the masters to process multiple update requests per elementary cycle to make the system more reactive? We could use the same approach we just saw (Sections 8.7.4 and 8.7.4) and not only piggyback the minimum update request from each set of pending requests, but the least n update requests. But what if n requests do not fit into a single trigger message?

A solution to this problem is to increase the size of the trigger message window and convey trigger messages that piggyback different information. An example of this is shown in Figure 8.25. The trigger message window is partitioned into two subwindows, TMW 1 and TMW 2. In each of these the trigger message is conveyed $k = 3$ times to make it fault tolerant within each subwindow. In this way we allocate a larger portion of the bandwidth to the trigger messages and increase the amount of information that can be reliably broadcast at precise instants of time across the whole network. For some applications, such as those requiring the processing of multiple update requests per elementary cycle, increasing the size of the trigger message window in this way may be worthwhile.

8.8 Discussion

We have now completed the design of FTTRS, a communication subsystem based on Ethernet that is fault-tolerant, flexible, and supports real-time communication. FTTRS inherits its flexibility and support for real-time communication from FTT, and more specifically, from HaRTES. The fault tolerance, in contrast, comes from our very own mechanisms, which we designed specifically to prove our thesis. In particular, we designed the mechanisms to meet a series of requirements that we derived from the thesis statement. The question now is whether we have indeed satisfied these requirements.

Let us now validate the design, i.e., go through the requirements one more time and see if FTTRS meets them.

The first two requirements were the following:

R1: The communication subsystem is based on the FTT paradigm.

R2: The communication subsystem is based on Ethernet.

These we have satisfied. FTTRS is based on HaRTES and HaRTES is an Ethernet-based version of FTT. In particular, in FTTRS we have kept Ethernet as the underlying technology and maintained the distinctive features of FTT, among them a master/multi-slave control of the communication; a subdivision of the communication time into periods of fixed duration called elementary cycles; a polling of messages at the beginning of each elementary cycle; databases to keep track of evolving system requirements, which enables operational flexibility to deal with unpredictable changes to real-time requirements; and a classification of messages into synchronous, asynchronous, and trigger messages.

What about the third requirement? Have we met it as well? Let us recall what it said:

> R3: The communication subsystem provides an FTT service even if both permanent and transient faults occur.

To further qualify this requirement we broke it down into the following three subrequirements:

> R3.1: The failure of a slave node does not disrupt the FTT service provided to other slave nodes.
>
> R3.2: The maximum duration of transient faults that the communication subsystem can tolerate can be parameterized.
>
> R3.3: The communication subsystem provides an FTT service even if any one of its components, no matter which one, suffers a permanent fault.

And yes, FTTRS meets all three.

It satisfies requirement R3.1, as explained in Section 8.7.1, thanks to the port guardians we added to the FTTRS architecture. These guardians restrict the failure semantics of the slaves to such an extent that their failure cannot prevent the switches from serving other slaves. Incidentally, having restricted the failure semantics in this way is not only useful to us, at the communication level, but also to higher levels. For instance, it makes it easier to design mechanisms on top of FTTRS to tolerate slave failures. Indeed, Sinisa Derasevic's work,[38] which is all about tolerating the failure of FTT slaves and dealing with payload errors, benefits from the restricted failure semantics.

[38] Sinisa Derasevic, Manuel Barranco, and Julián Proenza. "Appropriate Consistent Replicated Voting for Increased Reliability in a Node Replication Scheme over FTT". in: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014).* IEEE. 2014, pp. 1–4; Derasevic, Proenza, and Barranco, "Using FTT-Ethernet for the Coordinated Dispatching of Tasks and Messages for Node Replication"; Sinisa Derasevic et al. "First Experimental Evaluation of the Consistent Replicated Voting in the Hard Real-Time Ethernet Switching Architecture". In: *Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2015).* IEEE. 2015; Sinisa Derasevic, Manuel Barranco, and Julián Proenza. "Designing Fault-Diagnosis and Reintegration to Prevent Node Redundancy Attrition in Highly Reliable Control Systems Based on FTT-Ethernet". In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016).* IEEE. 2016.

As to requirement R3.2, recall that switches, being fail silent, only suffer permanent faults[39] and that faults in slaves do not need to be tolerated—they only need to be prevented from interfering with the FTT service. Thus, as pointed out earlier (Section 8.7.2), the only transient faults we have to tolerate are those occurring in links. To do so, we developed the proactive retransmission approach. This approach is parameterizable and thus ensures that FTTRS meets requirement R3.2. This is so even though the parameterization is done in terms of a redundancy level and not in terms of a duration, as R3.2 stipulates. The point is that given an upper bound on the maximum duration of transient faults we can always come up with an appropriate redundancy level. We simply take the maximum rate with which messages can be transmitted, which is given by the speed of the Ethernet links, and calculate how many messages can be lost at most during the maximum duration of a transient fault. The redundancy level needed to tolerate all transient faults must then be greater than the calculated maximum number of messages that can be lost.

In practice, in a given deployment the redundancy level for a given message may be too low because the faults that are encountered last longer than the faults that the communication subsystem has been parameterized to tolerate. This may occur because we underestimated the duration of faults; or because tolerating all transient faults requires a redundancy level that is too high to be practical (the higher the redundancy level for the message streams, the more reliable they become, but the less streams are schedulable within an elementary cycle). Either way, in these cases FTTRS alone will not tolerate all transient faults. Nevertheless, layers on top of FTTRS, or the application itself, may take some compensatory measures. For instance, the application may request a shorter period for synchronous messages and a shorter minimum interarrival time for asynchronous messages. That way, data messages are transmitted more frequently than strictly necessary and the failure to deliver a message in some elementary cycles may be tolerated by the application itself. This approach may work for both data messages and slave request messages. For trigger messages, on the other hand, the redundancy level must always be high enough.

Lastly, FTTRS satisfies requirement R3.3 because it provides an FTT service even if one of its links or switches suffers a permanent fault. First, the duplicated architecture of FTTRS, together with its redundant interlinks, ensures that regardless of whether it is a link, interlink, or switch that fails, there is still a path interconnecting any pair of slaves (see Figure 8.6 on page 175). Second, all available communication

[39] Transient faults, if they occur within an internally duplicated and compared switch, lead to the internal comparison detecting a mismatch. This leads to a shutdown of the switch and thus, in practice, the switches can only suffer permanent faults. As mentioned before (Section 8.6.1), this makes the switches fail silent, but also makes them more likely to fail.

paths are used all the time: slaves send each message through both their slave links and switches exchange all messages through the interlinks (recall replica radiation, illustrated in Figure 8.13 on page 192). Third, because of their failure semantics, neither the failure of a link nor of a switch can prevent slaves or a surviving switch from sending messages. Finally, because both switches keep their SRDBs consistent and synchronize their elementary cycles in such a way that each switch can continue at its own pace in the absence of the other, the switches provide a consistent FTT service and either switch continues the service unperturbed when the other fails. Thus, regardless of which link or switch suffers a permanent fault, FTTRS provides an FTT service and thus satisfies requirement R3.3.

In fact, FTTRS not only satisfies requirement R3.3, but surpasses it: there are multiple scenarios in which FTTRS can provide an FTT service to all slaves even if several of its components fail. For instance, the degraded topology we saw in Figure 8.22 on page 223 corresponds to an FTTRS network that suffered permanent faults in three slave links and all but one interlink. Yet, FTTRS can still provide an FTT service to all slaves in that scenario. More generally, FTTRS can provide an FTT service to all slaves as long as there is still a surviving communication path between any pair of slaves. And in case the application can tolerate the failure of some of its slaves, FTTRS can tolerate even more permanent faults (e.g., the failure of both slave links of a slave).

Finally, requirement R4 stated that it must be possible to implement the communication subsystem in practice. This will be the topic of the next chapter. Before we move on, however, let me be explicit about some of the limitations of the work presented.

8.8.1 Limitations of the Work Presented

Any research work has its limitations and the work presented here is no exception. Some of the limitations stem from the scope I defined for the work at the very beginning (Section 1.4) and the non-requirements I defined in this chapter (Section 8.1.2). Other limitations are the consequence of later design decisions, simplifications, and priorities.

The following are limitations due to the scope of the work:

- I did not design FTTRS to work with non-FTT nodes. This is a loss with respect to the original HaRTES, but not a loss with respect to other FTT versions, which are not compatible with legacy nodes either.
- I did not add a plug-and-play mechanism similar to the one used in FTT-

SE. In fact, there are good arguments against having such a mechanism in a subsystem designed for critical systems. For instance, a security-related argument is that plug-and-play mechanisms make it easier for an intruder to gain access to the system. A reliability-related argument is that plug-and-play mechanisms may make it more difficult to restrict the failure semantics of nodes (if an unknown node can be connected to the system at runtime, a port guardian has less information about it and therefore a harder time to distinguish correct behaviors from incorrect ones).

- I did not replicate slaves to tolerate their failure in case they are critical for a given application. Within the FT4FTT project this was the responsibility of Sinisa Derasevic, who is currently addressing this issue in his own dissertation.
- I did not evaluate how to modify the scheduler and admission control of the switches to take into account the proactive retransmission of messages or replica radiation. The focus of my dissertation is fault tolerance, not schedulability.
- I did not evaluate the impact on performance of the proactive retransmissions or the replica radiation. Again, the focus of the dissertation is fault tolerance. Performance was not a focus.
- The FTTRS design is based on HaRTES without server-based scheduling for asynchronous traffic.[40] I excluded server-based scheduling from the design because it is an optional improvement for handling asynchronous traffic whose main motivation was to support legacy applications and to make the handling of asynchronous messages more efficient; but both of these are goals that are out of the scope of this dissertation.

The following are limitations that fall within the scope of building a reliable and fault-tolerant communication subsystem, but which I did not address in this work for various reasons:

- I did not finish a quantitative reliability analysis of FTTRS and the other versions of FTT to find out how much of a reliability improvement FTTRS can provide with respect to previous solutions. I did start working on such a reliability analysis, but decided to postpone it until after this dissertation because it would have taken too much extra time and was unnecessary for

[40] Santos, “Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications”, Ch. 4.

proving the thesis. For proving the thesis statement (page 10) it was only necessary to show that FTT on Ethernet can be made fault tolerant; it was not necessary to quantify the reliability of FTTRS or any other version of FTT.[41]

- I did not formally verify FTTRS. Formal verification consists in building an abstract mathematical model of a system and then formally proving that the model satisfies desired mathematical properties. This can provide one of the strongest forms of evidence to show that a system satisfies certain properties. The main problem, however, is that for a complex system, such as FTTRS, abstracting it into a tractable mathematical model is only possible if the model is significantly simplified with respect to reality, which often leads to oversimplified models.[42] As an alternative one may then built multiple models for a single system, each capturing some aspect of it. But this also poses problems if the different aspects interact with each other. After considering formal verification for FTTRS—in particular model checking—I came to the conclusion that it was not a promising approach. First, the mechanisms for restricting the failure semantics of slaves and switches did not seem worthwhile to verify formally: both the internal duplication with comparison and guardians are well-established mechanisms to obtain components with restricted failure semantics. Second, tolerating transient faults by ensuring that at least one copy of a message gets through if we proactively retransmit the message k times is obviously true if the duration of transient faults is upper bounded and k is sufficiently large. Third, the mechanism of having masters exchange minimum update requests to agree on what request to process next is sufficiently simple to not warrant a formal verification—after all, the mechanism builds upon a reliable communication between switches and then relies on the simple mathematical fact that for two sets Q_A and Q_B of totally ordered update requests, $\min(Q_A \cup Q_B) = \min(\{\min(Q_A)\} \cup \{\min(Q_B)\})$, as explained earlier (page 222). For this simple fact it did not seem worthwhile building a model. Finally, regarding the mechanisms to achieve elementary cycle synchronization, I was unable to verify them using model checking since, as I discovered myself when I tried it, model checking is inadequate

[41] Nevertheless, I was able to publish a first work-in-progress paper on some aspects of the reliability analysis: David Gessner et al. "Towards a Reliability Analysis of the Design Space for the Communication Subsystem of FT4FTT". in: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE, 2014.

[42] Such oversimplified models are sometimes humorously called "spherical cows". The expression comes from the joke about the dairy farmer who asks a theoretical physicist for help in increasing milk production. The physicist accepts the challenge, builds a mathematical model, and a few weeks later presents a solution to the farmer, opening with the words "I have the solution, but it only works if we assume a cow is a sphere of uniform density in a vacuum".

for this task. The elementary cycle synchronization is driven by clocks and "[c]lock-driven systems [as opposed to, e.g., round-based systems] represent a problem for model checking".[43] And in any case, the experiments discussed in the next chapter already validated that the elementary cycle synchronization works.

- I did not design any process to reintegrate a faulty switch or a switch that has lost replica determinism. First, designing such a process was unnecessary to prove the thesis. Second, it is not clear that adding reintegration mechanisms would improve the reliability of FTTRS: reintegration mechanisms add more complexity to the system and more mechanisms that may go wrong. Also, if a switch has been declared faulty, it may be wiser not to trust it anymore and to not reintegrate it.

- I assume that the interlinks are replicated and that at least one is always available. Thus, I have not designed the system to tolerate the failure of all interlinks, which would lead to a partition of the network. As mentioned earlier, this can be mitigated by ensuring that the system starts out with sufficient interlinks.

- My design of FTTRS trusts slaves to request sensible changes to the SRDB. The failure semantics of a slave, however, includes payload errors. This means that a faulty slave could send bogus update requests and, if these pass the admission control of the masters, the system requirements would be updated incorrectly. For instance, a slave sending bogus requests could incorrectly delete existing message streams, create new ones, change the periodicity of an existing synchronous stream, modify deadlines, and so forth. Although this could not cause a failure of the communication, it could cause a failure of any application relying on the message streams to have sensible values. Luckily, as mentioned earlier (Section 8.6.2), the problem can be solved: masters could only take into account update requests from trusted slaves (e.g., slaves with at most omission failure semantics); the fail-silent masters themselves could be the only ones to request changes; or slaves could be replicated and update requests would then only be processed by the masters if a majority of the slave replicas request the same update.

- A babbling idiot slave could continuously send minimum update requests, i.e., update requests that are the minimum according to the total order relation

[43] Rodrıguez-Navas, "Design and Formal Verification of a Fault-Tolerant Clock Synchronization Subsystem for the Controller Area Network", As Rodriguez-Navas wrote, "in a model made up of timed automata it is not possible to have clocks increasing indefinitely together with an infinite number of time marks that are to be compared with the clock values", p. 91.

used by the minimum update request exchange mechanism (Section 8.7.4). By being minimum update requests, these requests have in practice highest priority: they are processed before other pending requests. Although the port guardians would drop those that violate the corresponding minimum interarrival time, those not violating it would get through, potentially delaying the processing of genuine requests. The problem can be solved, as the previous one, by only allowing requests from trusted (e.g., internally duplicated and compared) slaves; by only allowing masters themselves to request changes; or by replicating slaves and having them, or the masters, vote on proposed changes. We could also simply increase the minimum interarrival time for update request to mitigate the problem. Moreover, we could add error counters to the port guardians that track the rate with which slaves send untimely messages. A guardian may then permanently disconnect a slave if the latter sends too many update request messages per unit of time.

- If there is a shower of update requests, it may take the masters several elementary cycles to process them. This is so because with the exchange of minimum update requests the masters can only agree on a limited number of update requests per elementary cycle. Previous versions of FTT also failed to provide guarantees for the time to react to update requests.[44] In any case, the problem can be mitigated in FTTRS by using large trigger messages that can piggyback many update requests at the same time and by using multiple subwindows within the trigger message window (see Section 8.7.5).

- I have designed FTTRS to piggyback update requests and master command messages on trigger messages. This is only feasible as long as update requests and master commands are sufficiently small. Fortunately they are, especially in comparison with the maximum allowable size for Ethernet payloads (1500 bytes). Still, by piggybacking I increase the size of trigger messages, which makes them more vulnerable to transient faults. Then again, these faults can be addressed by increasing the trigger message redundancy level.

- The trigger message redundancy level, k, is static. It cannot change at runtime. Allowing it to change, like the redundancy level of synchronous and asynchronous messages, would allow FTTRS to adapt to electromagneticaly harsher environments than those that were expected. Fortunately, this is

[44] The non-real-time performance of processing update requests has been evaluated for FTT-SE and HaRTES by Álvarez et al., "A First Performance Analysis of the Admission Control in the HaRTES Ethernet Switch"; Inés Álvarez. "Study of the Admission Control in the Flexible Time-Triggered and the Audio Video Bridging Communication Protocols". MA thesis. Universitat de les Illes Balears, 2016.

feasible. Indeed, making the trigger message redundancy level dynamic is one of the planned tasks for the DFT4FTT project, the successor to the FT4FTT project (within which I developed FTTRS).

The above limitations leave ample room for future work, some of which I have already begun. None of these limitations, however, represent any problem for proving our thesis.

One limitation FTTRS does not have is a prove of feasibility: with the help of Alberto Ballesteros, Inés Álvarez, Andreu Adrover, and Francisca Font (who were bachelor or master students at the time), I could experimentally verify the feasibility of the main mechanisms of FTTRS. This shows that FTTRS can be implemented in practice and thus also satisfies requirement R4 (page 149). The next chapter is all about this.

Chapter 9

Feasibility of FTTRS

> They say no plan survives first contact with implementation. I'd have to agree.
>
> Mark Watney, in Andy Weir's *The Martian*

We now have a design for FTTRS (Chapter 8), which we showed to satisfy several requirements derived from our thesis. A design, however, is only a plan. Thus, to complete the proof of our thesis and also meet the last requirement (R4, page 149), we must show that the plan—the design—is feasible.

How, then, can we show that the design is feasible? One approach is to implement and test a complete prototype. That is, implement all aspects of the design, test each aspect in isolation, and finally test the complete prototype as a whole. But this is time consuming. Thus, to prove the feasibility of a design it is often more practical to realize a **proof of concept**: a physical system built purely to demonstrate that some concept or mechanism can be realized in practice and which is usually small and may or may not be complete. For some aspects of a design even a proof of concept may be unnecessary: the aspect may have already been implemented multiple times in the past in analogous ways for other systems or we may find other arguments to substantiate its feasibility. For instance, nowadays we know that it is feasible for non-faulty computers to perform addition and it would be silly to ask for a proof of concept of computerized addition just because the design of some architecture or protocol happens to rely on arithmetical operations. Similarly, if an integrated circuit can do nothing but addition, we do not need a proof of concept to show that it can also multiply: the knowledge that multiplication can be seen as repeated addition suffices. To demonstrate the feasibility of FTTRS I therefore chose the timesaving approach: proof of concept prototypes for the mechanisms that I considered most relevant (Sections 9.3 to 9.6) and arguments for the feasibility of the remaining unimplemented mechanisms (Section 9.7).

The prototypes I will present have not only been built, but obviously also been subjected to experiments to check that they have been built correctly, that is, according to the design. The main goal of these experiments was therefore verification, which is distinct from validation. **Verification** is the process of checking that an implementation complies with its functional specification or design. It tells us if a system implements its design without any bugs or other implementation errors. **Validation**, in contrast, is the process of checking that a system does what we intend it to do. It tells us if a design or its implementation achieves its high-level purpose and meets its requirements. The difference is also sometimes stated as a pair of questions: verification asks “Are we building the product right?”, whereas validation asks “Are we building the right product?”.[1] The focus of the experiments I will present shortly is verification since we already validated the design of FTTRS in the previous chapter: we showed through logical arguments that the design of FTTRS meets its requirements. Nevertheless, the experiments in this chapter do further validate the design. For instance, by building a prototype of the FTTRS architecture, injecting permanent faults into it, and observing that these are tolerated (Section 9.5), we not only verified that the prototype is implemented correctly, but also further validate that the duplicated architecture of FTTRS can indeed tolerate permanent faults (requirement R3.3, page 149).

For the prototyping and experimentation I could count with the help of several students. Specifically, the prototyping work, the execution of experiments, and the collection of results was carried out by Alberto Ballesteros, Inés Álvarez, and Andreu Adrover,[2] all three of which were either undergraduate or master students at the time.[3] Ballesteros in particular also supervised the implementation work done

[1] Hoang Pham. *System Software Reliability.* Springer Science & Business Media, 2007, p. 134.

[2] Their work has been published in several publications: David Gessner et al. “Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014).* IEEE, 2014; Ballesteros et al., “Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star for Ethernet”; David Gessner et al. “Experimental Evaluation of Network Component Crashes and Trigger Message Omissions in the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 11th IEEE World Conference on Factory Communication Systems (WFCS 2015).* IEEE, 2015; Alberto Ballesteros et al. “First Implementation and Test of a Node Replication Scheme on Top of the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016).* IEEE. 2016.

[3] Other students worked on other aspects of the FT4FTT architecture that are not directly related to

by Álvarez and Adrover, in addition to completing much of the implementation and experimentation himself. Francisca Font, another undergraduate student, also helped with the last prototype (Section 9.6). I provided the FTTRS design, defined the experiments to test the elementary cycle synchronization among slaves (Section 9.3) and the experiments to test network component crashes and trigger message omissions (Section 9.5), and provided supervision, guidance, and help for all prototypes and experiments. In particular, I gave regular feedback on the implementation of all prototypes—especially on the key implementation decisions—and provided guidelines to all experiments directly related to the mechanisms I developed (Chapter 8), with special attention to their purpose and the definition of the needed parameters. My involvement in the last experiment (the one that tests a control application with replicated nodes on top of FTTRS, Section 9.6) was less as it did not focus that much on FTTRS in particular, but more on FT4FTT as a whole (whose other key component, apart from FTTRS, is a node replication scheme developed by Sinisa Derasevic).

9.1 Inherited Codebase

When I began working on this thesis there were two previous Ethernet-based FTT implementations available: a software implementation of FTT-SE[4] and an FPGA-based hardware implementation of a HaRTES switch.[5] Both could have served as a starting point for implementing FTTRS. Nevertheless, although FTTRS is based on HaRTES, and not on FTT-SE, I favored building the FTTRS prototypes using the FTT-SE software implementation as a foundation. The main reason for this was that a software implementation generally allows faster prototyping than a hardware implementation. This I considered more important than the better performance that an FPGA-based implementation might have achieved.

The FTT-SE implementation was initially developed in the C programming language by Ricardo Marau while working on his dissertation.[6] In this implementation the slaves and masters are implemented as processes executed in user space on top of an x86-based computer running a GNU/Linux operating system. Thus, in a typical FTT-SE network based on this implementation, several computers running GNU/Linux are interconnected by means of a single commercial off-the-shelf Eth-

FTTRS and that I am therefore omitting.

[4] Marau, Almeida, and Pedreiras, “Enhancing Real-Time Communication over COTS Ethernet Switches”.

[5] Santos et al., “A Synthesizable Ethernet Switch with Enhanced Real-Time Features”.

[6] Marau, “Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management”.

ernet switch, with one of these computers executing the process for the FTT-SE master, and each one of the others executing an FTT-SE slave process.

Luis Silva, at the time a student at the University of Aveiro, began adapting the FTT-SE codebase to implement some of the functionalities of a HaRTES switch. In particular, he took the code related to an FTT master and added a software-implemented switch that forwards FTT messages and confines synchronous and asynchronous FTT messages into synchronous and asynchronous windows, respectively. The result was a software switch specifically implemented to forward FTT messages at the appropriate times and that embedded an FTT master. We inherited this codebase, which I will call the **Marau-Silva codebase**, and we adapted it to implement FTTRS. Thus, in our experiments the slaves, embedded masters, and switches were Linux processes within x86-based computers.

9.2 Prototype Hardware

We used two types of computers for the prototypes: small barebones and multicore PCs.

The *small barebones* were Jetway JBC373F38-525-B computers,[7] each with an Intel Atom D525 dual-core processor with hyperthreading support and 2 GB of RAM, as well as four Ethernet interfaces.

The *multicore PCs*, in contrast, were regular Intel i7-4770 multicore PCs with hyperthreading support and 8 GB of RAM each. Ballesteros provided each with several Intel i350-T4 Ethernet interfaces, which allow the management of multiple low-level operation parameters, a useful feature to decrease communication delays. Moreover, we used Asus Z87-WS motherboards, which each contain a PCIExpress switch that allows applications to receive and transmit Ethernet frames in parallel with little interference.

In the experiments that focused on the fault-tolerance mechanisms (Section 9.5) and the FT4FTT prototype (Section 9.6), the small barebones were used for the slaves and the multicore PCs for the FTTRS switches. For the experiments that tested the elementary cycle synchronization (Sections 9.3 and 9.4), on the other hand, only the multicore PCs were used, as we will see shortly.

Given this hardware, and the Marau-Silva codebase we inherited, we proceeded with the implementation of proof of concepts for the main mechanisms of FTTRS.

[7] Jetway Computer Corp. *Jetway JBC373F38-525-B*. URL: http://www.jetwaycomputer.com/JBC373F38.html (visited on 2017-01-03).

9.3 Testing the Fault-Tolerant Elementary Cycle Synchronization among Slaves

The first proof of concept implemented the elementary cycle synchronization mechanism for slaves (Section 8.7.2, page 192). To simplify this task, we abstracted away the presence of two FTTRS switches. This was a reasonable abstraction for a first experimental evaluation because the FTTRS switches are replica deterministic and thus provide an identical service from the slaves point of view. In addition, the abstraction allowed us to test the elementary cycle synchronization among slaves without first having to implement the elementary cycle synchronization among FTTRS switches.

As a starting point, Álvarez took the Marau-Silva codebase and then modified it to implement my design of the FTTRS elementary cycle synchronization. First, she modified the code of the master to make it proactively retransmit trigger messages for a total of k transmissions per trigger message window. Moreover, she added code for the transmissions to occur isochronously, with an intertransmission time τ. Both k and τ, as well as the desired elementary cycle duration, were passed as arguments to the executable for the master. Furthermore, she modified the code for the master to insert k and τ in all trigger messages, and to add sequence numbers to them (the elementary cycle duration was already encapsulated in the trigger messages in the Marau-Silva codebase).

Álvarez also modified the code of the slaves such that a slave predicts the arrival time of the last trigger message of the current trigger message window. Given a slave s_1 that receives a trigger message $\mathrm{tm}(i)$, where i denotes the message's sequence number for a given elementary cycle, the reception occurs at an instant of time t_i, when an arrival event $\alpha_{s_1}(i)$ occurs (refer back to Figure 8.18 on page 197). The instant of time that must be predicted is t_k, when the arrival event $\alpha_{s_1}(k)$ occurs, and which coincides with the expiration of the trigger message window. Its value is $t_k = t_i + (k - i)\tau$. Once the expiration time of the trigger message window is predicted, a slave needs to wait until the predicted time to properly synchronize with other slaves. To achieve this, Álvarez's implementation ensures that upon the reception of a trigger message $\mathrm{tm}(i)$, a slave creates a thread that waits until the instant of time t_k. She coded the wait using a busy wait, which prevents the thread from being put into sleep by the operating system. The advantage of a busy wait over timers is that it is more deterministic: with timers a slave process would be put to sleep by the operating system until the predicted expiration time and this is less deterministic because it necessarily involves context switches. The disadvantage is that if a slave actively waits for the instant of time t_k, then, if the slave is implemented on a machine with only a single CPU, it cannot execute anything else until t_k. In all our prototypes, however, the slaves had available hyperthreading and more than one CPU core. Moreover, in later experiments (Section 9.6) the busy wait did not have any remarkable impact on any higher layers that were put on top of FTTRS slaves. Finally, Álvarez added some instrumentation code to the slaves to evaluate the correct implementation of the elementary cycle synchronization.

With the modified code, we proceeded to test the implementation. For this we chose software implemented fault injection (SWIFI).[8] That is, Álvarez implemented each slave process to execute additional code to ignore certain trigger messages. In this way, we could test whether the implementation indeed tolerates trigger message omissions. In particular, the added SWIFI code chooses which trigger message must be ignored depending on their sequence number. This is done such that all possible combinations of losing up to $k-1$ trigger messages on the links of the different slaves are tested, which are all trigger message loss scenarios under which FTTRS is designed to synchronize slaves. The number of these trigger message loss combinations is given by

$$\left(\sum_{e=0}^{k-1}\binom{k}{e}\right)^n,$$

where k is the number of trigger messages per elementary cycle, e is the number of lost trigger messages, and n is the number of slaves attached to the network. The expression can also be expressed more simply. According to the binomial theorem, $\sum_{e=0}^{k}\binom{k}{e}1^{k-e}1^e = (1+1)^k = 2^k$; and since $\binom{k}{k} = 1$, the number of trigger message loss combinations can also be expressed as

$$(2^k-1)^n.$$

In addition to the SWIFI code, Álvarez also added instrumentation code to the slaves to timestamp the trigger message window expiration time of each elementary cycle. This allowed us to evaluate the precision with which the slaves synchronize their elementary cycles.

9.3.1 Test Setup

We used two test setups: one with a virtualized network, as seen in Figure 9.1, and another with a physical Ethernet switch, as seen in Figure 9.2. The virtualized setup, contrary to the physical one, did not take into account physical Ethernet

[8] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. “Fault injection techniques and tools”. In: *Computer* 30.4 (1997), pp. 75–82.

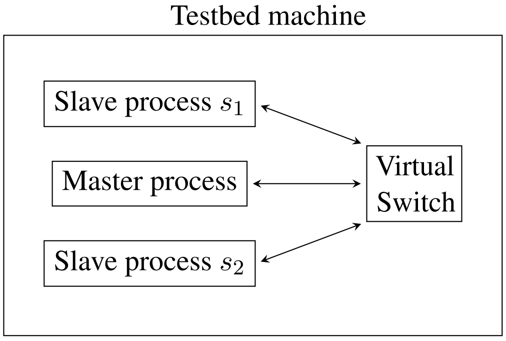


Figure 9.1: Virtualized network setup to test the slave elementary cycle synchronization.

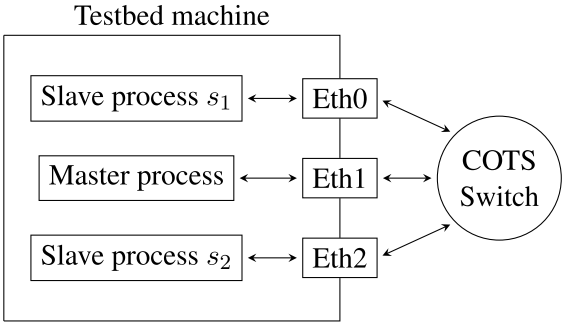


Figure 9.2: Physical network setup to test the slave elementary cycle synchronization.

interfaces, the propagation delay through physical cables, and the switching delays of a physical Ethernet switch.

In both setups we used the minimum number of slaves and masters required to evaluate the elementary cycle synchronization: one master and two slaves. Moreover, in both setups the processes implementing the master and slaves were executed on the same machine (one of the multicore PCs mentioned in Section 9.2), under the same GNU/Linux OS instance. This made it possible for the slaves to share the same clock, thereby providing a common timebase for timestamping (sharing a machine also had its disadvantages, which I will discuss in the next subsection). Finally, in both setups the master process and the slave processes were attached to a single 100 Mbps Ethernet switch.

The key difference between the two setups was that in the virtualized one the switch was virtual and thus the processes were attached to it through virtual Ethernet interfaces. For this Álvarez used a *virtual distributed Ethernet* (VDE) switch.[9] In the physical network setup, in contrast, the switch was a commercial off-the-shelf (COTS) Ethernet switch, to which each process was attached by means of a different physical Ethernet interface of the testbed machine. Having a virtualized setup in addition to a physical one allowed a first simpler experimentation.

9.3.2 Test Parameters and Results

To evaluate the results we defined the **absolute EC offset** between two slaves s_1 and s_2 in elementary cycle c as $|t_{s_1}(c) - t_{s_2}(c)|$, where $t_{s_1}(c)$ and $t_{s_2}(c)$ indicate the recorded timestamp in elementary cycle c by slaves s_1 and s_2, respectively. In other words, we measured by how much the ends of the trigger message windows deviate from one slave to the other in each elementary cycle. With perfect synchronization and no overheads, the observed deviation should have been zero if the elementary cycle synchronization mechanism correctly tolerates the loss of trigger messages. But since reality is not perfect, we did observe some deviations.

Table 9.1 shows the parameters Álvarez used in the tests. She set the elementary cycle length to $1000\,\mu\text{s}$, a suitable value for timely communication in typical control applications. Regarding the trigger message intertransmission time τ, it had to be set to a value greater than the transmission time of a trigger message, including the Ethernet interframe gap, which lasts 96 bit times—otherwise the Ethernet controllers of the nodes or the standard Ethernet switch could have queued the

[9] Renzo Davoli. “VDE: Virtual Distributed Ethernet”. In: *Proceedings of the 1st IEEE International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM 2005)*. IEEE. 2005, pp. 213–220.

Table 9.1: Experiment parameters.

	k	τ (µs)	EC length (µs)	# Test runs
Virtual Switch	4	100	1000	1000
COTS switch	4	100	1000	1000

trigger messages in their output ports, which could have interfered with the trigger messages' isochronous transmission. Since we used 100 Mbps Ethernet and the trigger messages did not carry any scheduling information, this means that we had to set τ to a value greater than $(72 \cdot 8 + 96)/100 = 6.72$ µs, where 72 is the number of bytes of a minimum Ethernet frame, including the preamble and start of frame delimiter. We chose a value of 100 µs, a round number well above 6.72 µs. We also chose to perform 1000 test runs, where each test run injected all possible combinations for losing trigger messages in both slaves with $k = 4$. That is, Álvarez performed $(\sum_{e=0}^{4-1} \binom{4}{e})^2 = (2^4 - 1)^2 = 225$ fault injections per test run, giving us 225 000 sample points. By investing more time, we could have executed more experiments with different parameters and we could have collected more sample points. However, the experiment we performed already showed the feasibility of the synchronization mechanism to tolerate the omission of trigger messages and it seemed wiser to proceed with different experiments to evaluate other aspects (Sections 9.4 to 9.6). Moreover, testing the elementary cycle synchronization for a value of $k = 4$, and not higher, seemed sufficient. After all, with $k = 4$, even if we assume that the bit error ratio is 10^{-6}, which is 100 times higher than the maximum bit error ratio acceptable according to the IEEE 802 standard,[10] the expected time for a slave not to receive any trigger message at all in a given elementary cycle through a given link would already be exceedingly unlikely—approximately 288 years on a 100 Mbps link with 1 ms elementary cycles, assuming that bit errors are independent and exponentially distributed (see Appendix C). Testing even more unlikely scenarios (values of k greater than 4) therefore seemed unnecessary. In any case, in a subsequent experiment (Section 9.5) we also used a value of $k = 6$ and then in the last experiment, which tests a full FT4FTT prototype and was concluded successfully (Section 9.6), we also made use of the slave elementary cycle synchronization mechanism, lending further support to the claim that the

[10] The standard states that "for wired or optical fiber physical media: Within a single access domain, the probability that a transmitted MAC frame (excluding any preamble) is not reported correctly at the Physical Service interface of an intended receiving peer MAC entity, due only to operation of the Physical layer, shall be less than 8×10^{-8} per octet of MAC frame length". Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks: Overview and Architecture*. 802-2001 (IEEE). 2001, Sec. 7.3a.

Table 9.2: Measured absolute EC offset results.

	# samples	mean (µs)	std. dev. (µs)	max (µs)
Virtual Switch	225 000	1.94	0.84	47.62
COTS switch	225 000	0.69	1.36	91.37

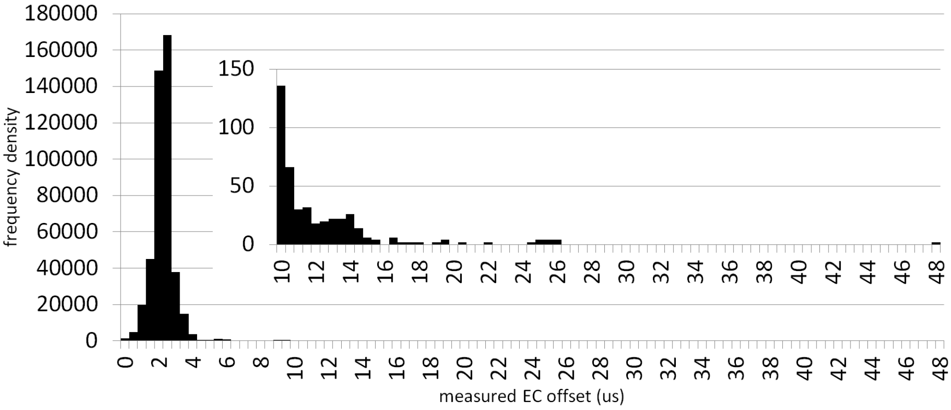


Figure 9.3: Histogram of measured absolute EC offset for shared machine with virtual switch. Bin size is 0.5 µs. The superimposed figure is a close-up of the right tail of the histogram.

mechanism is feasible.

The results are shown in Table 9.2. The mean and standard deviation of the absolute EC offset (1.94 and 0.84 for the virtual switch, 0.69 and 1.36 for the COTS switch) indicate that with both experimental setups the implementation achieves a good EC synchronization, on the order of 0.1–0.2% of the EC length (1000 µs), in most cases. However, the main measure of interest is the maximum absolute EC offset. This is so because FTTRS provides real-time communication, where the end of each elementary cycle constitutes a hard deadline for the round-based communication to take place. Unfortunately, with both experimental setups we also measured values of EC offsets significantly larger than the mean, with the largest values being 47.62 µs and 91.37 µs for the virtual switch and the COTS switch setup, respectively. These values constitute on the order of 5–10% of the elementary cycle length, which is also confirmed by the histograms of Figures 9.3 and 9.4. They both reveal that the distribution of the absolute EC offset has a long tail in our proof of concepts.

Overall, we can conclude that the results of the tests were promising: they showed

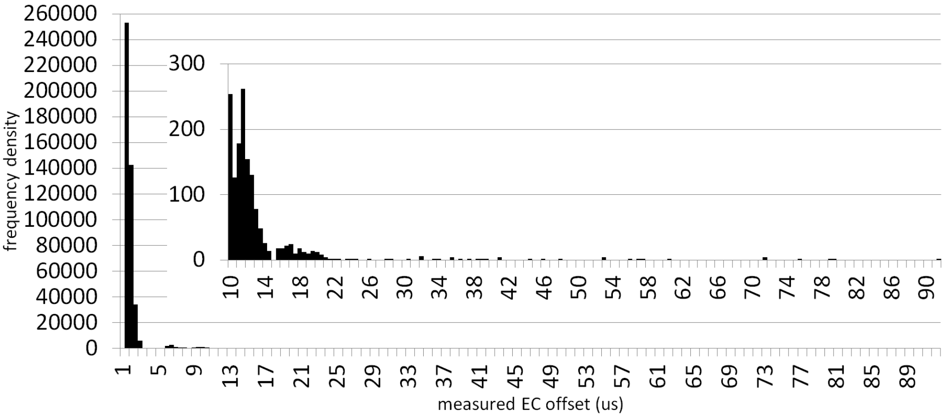


Figure 9.4: Histogram of measured absolute EC offset for shared machine with COTS switch. Bin size is 0.5 µs. The superimposed figure is a close-up of the right tail of the histogram.

that the implementation achieves a good elementary cycle synchronization most of the time and that trigger message losses were tolerated. Nevertheless, they also highlighted that occasionally a large absolute EC offset between the slaves could be observed. I think this is due to the slaves and the master being executed as user processes on top of a non-real-time OS. We inherited this from the Marau-Silva codebase. Moreover, the fact that the two slaves and the master were executed on a single machine may also have contributed to the occasional peak in the absolute EC offsets; after all, by being executed on a single machine, they were sharing resources and may have had to take turns to access some of these resources.

Depending on the application, a maximum absolute EC offset on the order of 50-100 µs, as we have measured, may not be a problem. If it is, then a real-time OS may be used or the master may be implemented in hardware. Alternatively, a large enough guard window between each pair of consecutive elementary cycles could be added. That way we can prevent one slave from transmitting messages of one elementary cycle during the next one even if there are occasional large EC offsets.

An important limitation of the experiments that Álvarez performed was that all slaves shared a common clock since they were executed as processes within the same machine. This enabled precise timestamping. Sharing a clock, however, also eliminated the natural source of desynchronization in a distributed embedded system, namely, clock drift. Thus, the experiments did not allow us to check the precision with which slaves would be able to synchronize when they are driven by different clocks. Even so, the experiments were useful to verify that the elementary cycle synchronization mechanism works and—most importantly—works when a proper subset of the trigger messages of each elementary cycle are corrupted. After all, in the experiments the slaves still had to receive a trigger message for synchronization, despite sharing a common clock. Moreover, if the synchronization had not worked correctly, we would have observed maximum EC offsets larger than the elementary cycle duration, i.e., 1000 µs.

In the context of this dissertation it was more important to test the fault tolerance of the synchronization mechanism than to test its precision when each slave has its own clock. Furthermore, the fact that the elementary cycle synchronization mechanism indeed tolerates faults is a general result; whereas any specific measurements to determine the precision with which distributed slaves can synchronize would depend on many implementation details, among them the quality of the oscillators that drive the slaves; whether the slaves execute a real-time OS, a generic OS, or no OS at all; and so forth.

In conclusion, the experiments showed that, as desired, the slave elementary cycle synchronization mechanism is feasible and that it is fault tolerant. Moreover, subsequent experiments (Sections 9.5 and 9.6) showed that the slave elementary cycle synchronization for the slaves, together with the slave elementary cycle synchronization among masters (Section 9.4), also worked in more realistic prototypes.

Finally, the description of some further experiments can be found in Álvarez's bachelor thesis.[11] In particular, she also describes experiments that measure the variability of the intertransmission time τ and of the duration of the trigger message window.

9.4 Testing the Elementary Cycle Synchronization between Masters

Besides synchronizing the elementary cycles among slaves, the FTTRS design also provides a lockstep synchronization of trigger message transmissions. The feasibility of this synchronization was verified in the next set of experiments. Specifically, Ballesteros carried out an experiment to assess the feasibility of the masters' rendezvous phase and periodic lockstep resynchronization (Section 8.7.4 on page 212). In the same experiment he also measured the precision of the corresponding proof of concept.

[11] Inés Álvarez. "Implementation and Verification of the Slave Elementary Cycle Synchronization Mechanism of the Flexible Time-Triggered Replicated Star for Ethernet". Bachelor Thesis. Universitat de les Illes Balears, 2014.

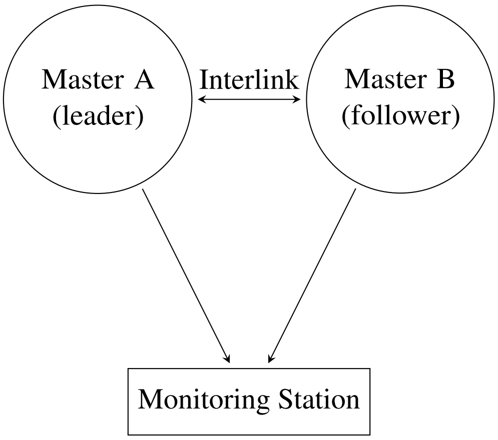


Figure 9.5: Setup for the proof of concept corresponding to the elementary cycle synchronization between masters.

As shown in Figure 9.5, the testbed comprised three stations: two interconnected master stations—one of which was the leader and the other the follower—and one monitoring station connected to both masters. The goal was for the masters to transmit their trigger messages in lockstep towards the monitoring station. Each master was implemented in one of the multicore PCs (Section 9.2). The code they executed was once again derived from the Marau-Silva codebase, appropriately modified to incorporate the FTTRS elementary cycle synchronization between masters. The monitoring station was built using one of the small barebones. The data that the station captured was the **trigger message offset**, i.e., the absolute value of the time elapsed between the reception of the trigger messages issued for the same elementary cycle by leader and follower. Thus, a trigger message offset of zero corresponds to perfect lockstep synchronization, i.e., maximum precision. The arrival times of the trigger messages were timestamped in the monitoring station using *tcpdump*, an operating system (OS) tool that provides a resolution of 1 μs. Finally, each station ran a regular GNU/Linux OS.

In the experiment Ballesteros recorded the reception of one million trigger messages using an elementary cycle length of 1 ms. Within 1 ms the maximum frequency offset (or clock skew) between the clocks of two non-faulty PCs is negligible. Thus, in the experiment we did not apply any frequency synchronization, i.e., adjustment of clock rate, which is also called *synto*nization (and which should not be confused with *synchro*nization). In other words, for the experiment Ballesteros did not implement a mechanism to slow down or speed up the clock of the follower master to ensure that the clocks of both masters progress at the same speed. After all,

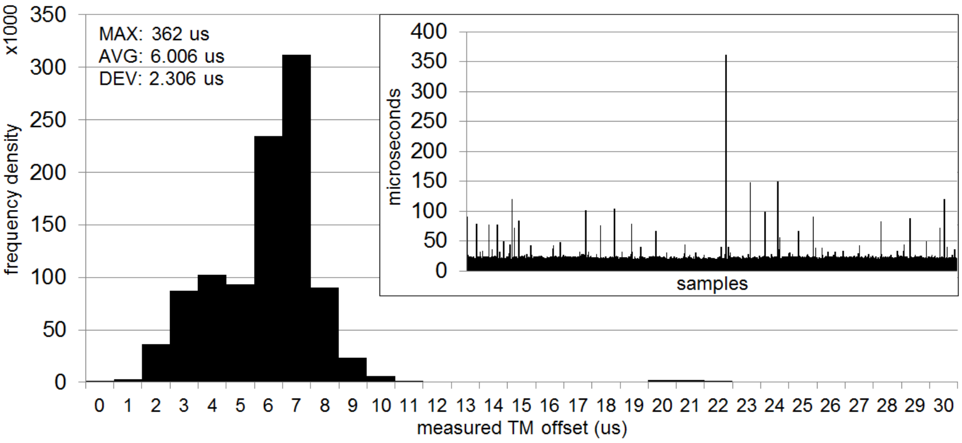


Figure 9.6: Results of the master synchronization experiment. The main figure is the histogram of the trigger message offsets with a bin size of 1 µs. The inset shows the value for each sample.

the masters would resynchronize about every millisecond anyway and within one millisecond the clocks of the non-faulty multicore PCs implementing the masters could not deviate significantly.

Figure 9.6 shows the relevant statistical results: a histogram of the trigger message offsets with a bin size of 1 µs and an inset bar diagram showing the value of each sample. As desired, we obtained low values for the mean and standard deviation (6.006 µs and 2.306 µs respectively). Indeed, the mean trigger message offset (6.006 µs) was only 0.6% of the 1 ms elementary cycle length and the standard deviation (2.306 µs) was only 0.23% of the elementary cycle length. The most frequent values appear in the bins corresponding to 6 and 7 µs, which embrace more than half of the samples. Finally, as the inset bar diagram shows, the maximum trigger message offset measured was 362 µs. This is disappointing. It shows that the follower can have a trigger message offset of more than one third of the 1 ms elementary cycle duration, which is not acceptable in many control applications. However, since such large trigger message offsets are rare (7 samples out of 1 million exceed 100 µs), we conclude that they are provoked by the nondeterminism of the OS. Hence, they can be reduced by devoting further efforts in tuning the software components of the system (e.g., using a real-time OS), or by implementing the masters in hardware.

As to the fault-tolerance of the mechanism to synchronize the masters, this was evaluated in the next set of experiments, which injected trigger message omissions.

9.5 Testing Network Component Crashes and the Omission of Trigger Messages

The two previous experiments focused on assessing the elementary cycle synchronization and did not comprise the full FTTRS architecture: they either only had a single switch (Section 9.3) or no slaves (Section 9.4). The next set of experiments, in contrast, provided a proof of concept of the full FTTRS architecture, which comprises two switches and several slaves. Specifically, the aim was to demonstrate the feasibility of the FTTRS architecture and to verify that the mechanisms of FTTRS do indeed tolerate faults. Adrover carried out these experiments with help from Ballesteros and under my guidance.

The experiments involved fault injection. Specifically, we verified that FTTRS can tolerate crash and omission failures of any single network component (i.e., link or switch), as well as the crash or omission failure of various combinations of multiple network components. On the other hand, we did not inject faults resulting in a crash of slaves or payload errors; tolerating such failure modes is out of the scope of FTTRS.[12] Nevertheless, Ballesteros did inject such faults in the final FT4FTT prototype (Section 9.6).

9.5.1 The FTTRS Prototype

Figure 9.7 shows the testbed we used. It consisted of an FTTRS prototype with two switches and three slaves, extended by a series of additional links connecting each of the switches and slaves to an instrumentation station.

The switches, slaves, and the instrumentation station were all implemented using commercial off-the-shelf personal computers. Specifically, each switch was implemented using one of the multicore PCs and each slave was built using one of the small barebones (Section 9.2).

Figure 9.8 shows a block diagram of the internals of one of our prototype FTTRS switches. The components that implemented the switch are the five Ethernet ports at the bottom of the figure (boxes labeled "Eth 1" through "Eth 5") and the *FTTRS*

[12] Indeed, in the FT4FTT project these failure modes are dealt with at the application layer and have been addressed by Derasevic, Barranco, and Proenza, "Appropriate Consistent Replicated Voting for Increased Reliability in a Node Replication Scheme over FTT"; Derasevic, Proenza, and Barranco, "Using FTT-Ethernet for the Coordinated Dispatching of Tasks and Messages for Node Replication"; Sinisa Derasevic, Manuel Barranco, and Julián Proenza. "Designing Fault-Diagnosis and Reintegration to Prevent Node Redundancy Attrition in Highly Reliable Control Systems Based on FTT-Ethernet". In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016)*. IEEE. 2016.

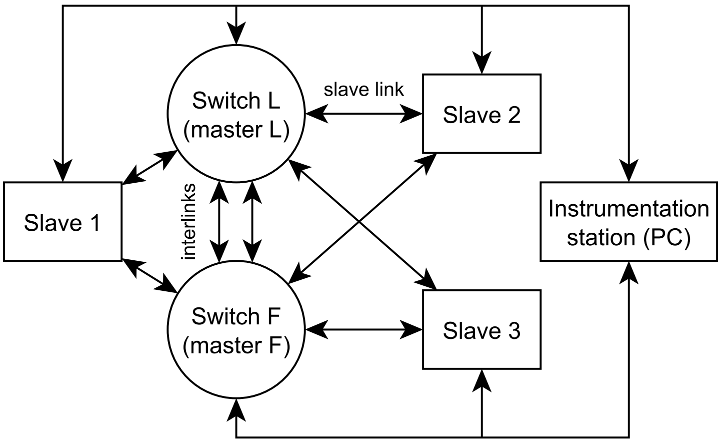


Figure 9.7: FTTRS fault-injection testbed.

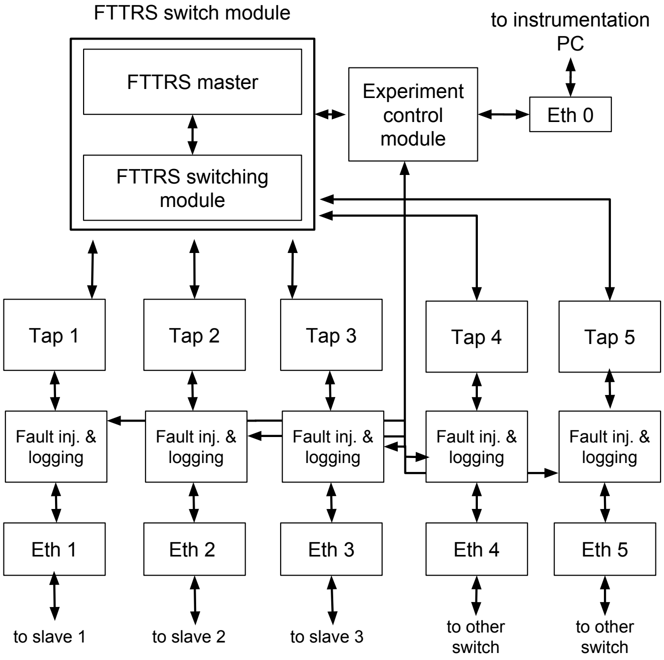


Figure 9.8: Internals of an FTTRS prototype switch.

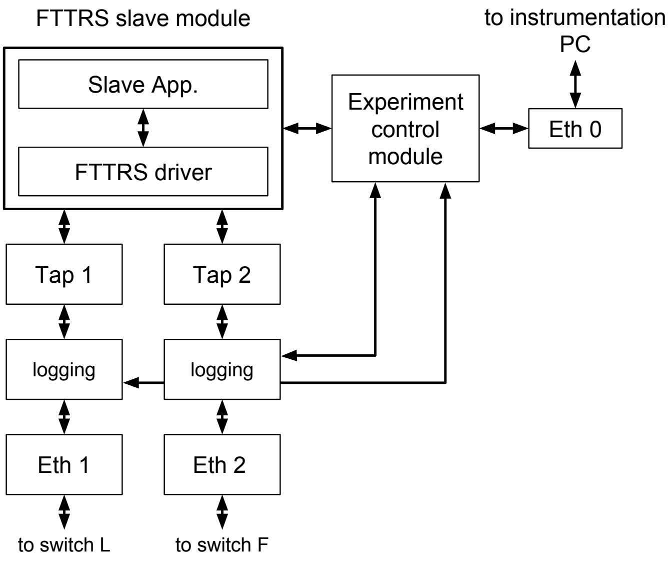


Figure 9.9: Internals of an FTTRS prototype slave.

switch module, shown at the top left corner, which includes the *FTTRS master* and the *FTTRS switching module*. The ports (again, boxes labeled “Eth 1” through “Eth 5”) are part of two network interface cards (NICs) installed in the PC implementing the corresponding switch. The interface to these ports (i.e., the bidirectional arrows shown immediately above them) is provided by the Linux kernel running on that PC. The FTTRS switch module was fully implemented in software and runs in the user space of the operating system. Like in the prototypes of the previous experiments, the code of the switch module is based on the Marau-Silva codebase. The remaining components (labeled “Fault inj. & logging”, “Tap 1” through “Tap 5”, “Experiment control module”, and “Eth 0”) are not strictly part of FTTRS, but are extensions to facilitate the execution of the experiments. These components are a fault injection and logging module that Adrover implemented in software for each of the links; several OS-provided virtual Ethernet interfaces (labeled “Tap” in the figure) that serve as an interface between the FTTRS switch module and each fault injection and logging module; and an experiment control module that used an additional Ethernet port and that allowed us to control the fault injection modules and the FTTRS switch module, as well as to retrieve any generated logs.

Figure 9.9 shows a block diagram of the internal structure of a prototype slave. Just like a switch, a slave had FTTRS specific components and additional components that made it easier to execute the experiments.

The FTTRS specific components were two Ethernet ports, one for each switch (labeled “Eth 1” and “Eth 2” at the bottom left of the figure); and a software-implemented *FTTRS slave module*, which included the application to be executed by the slave as well as an FTTRS driver that provided the application with primitives for parallel transmission and reception of frames.

As to the components that facilitated the execution of experiments, we again had an experiment control module connected to the instrumentation PC that allowed us to control the FTTRS slave module. We also had two logging modules that Adrover implemented and placed between the physical Ethernet ports and the FTTRS slave module with the help of two OS-provided virtual Ethernet *Tap* interfaces. Adrover did not perform fault injection at the slave end, but only at the switch end. This did not restrict the set of faults that we were interested in: for switch failures the switch end is the right place to inject the fault, for link failures and message omissions either end of a link is adequate for fault injection, and tolerating faults within slaves was out of the scope.

9.5.2 Experimental Evaluation of the Prototype

Next I will describe the application that Adrover executed on the three prototype slaves and the experiments he performed to verify that the FTTRS prototype correctly handled the crash of a switch and omission failures occurring in links. The logging information gathered during each experiment was stored and is available at the website of our research group.[13]

The Slave Application

The application Adrover executed on the slave nodes was a simplified replicated control application. All slaves performed the same task at the same time and then agreed upon the result of the control. Specifically, the operation of each slave consisted of three phases. In the first phase each slave generated a value. This value might have come from a sensor, but to keep things simple, we opted for using a simple counter. In the second phase this value was exchanged among the three nodes. Finally, the nodes executed a voting algorithm to reach a consensus on the value. The voting was a simple majority vote. Each phase was executed in a different elementary cycle, which means that the exchange of the counter values was scheduled by the masters every three elementary cycles.

[13] Andreu Adrover. *Log files for the FTTRS prototype fault-injection experiments*. URL: http://srv.uib.es/experimental-evaluation-of-fttrs/, The log files consist of 404 MB of uncompressed text files.

Since in the experiments (and in my whole dissertation) the focus was on the network, Adrover injected faults into network components only (switches and links) and tested that each slave could still provide the value of its counter to the other two slaves. If so, we considered the fault to have been correctly tolerated.

Testing the Tolerance to Switch Crash Failures

The first two experiments consisted in provoking the crash of one of the switches. In the first experiment Adrover made the leader crash and in the second the follower. In both cases the crash was only provoked after the rendezvous phase (Section 8.7.4 on page 212) that synchronized the trigger message transmissions of the two switches.

By inspecting the logs captured in the slaves we observed that, despite the crash of either switch, each slave was always able to share its counter value with the others and all slaves agreed on the value of the counter after voting. We therefore conclude that the crash of either switch was correctly tolerated.

Testing the Tolerance to Link Crashes

The goal of this fault injection campaign was to verify that the FTTRS prototype tolerates the crash of links. We achieved this by having the fault injectors in the switches logically disconnect the different links in all the possible combinations that can be tolerated by the FTTRS network architecture—this excluded a crash of all interlinks, which would partition the network, a situation that the current FTTRS design is not prepared to deal with. For practical reasons these injected faults were actually not permanent, but shortened to 20 elementary cycles. This allowed us to logically reconnect a link and try another combination of link failures without having to reinitialize all slaves and switches. Nevertheless, having injected faults that last 20 elementary cycles can be considered equivalent to a permanent fault. This is so because a fault lasting three or more elementary cycles would already have disrupted the three-phase application (subsection titled “The Slave Application”) in the same way as a permanent fault.

In our prototype with two switches and three slaves, there were two links per slave, giving a total of six slave links. In addition, there were two interlinks interconnecting the switches. We therefore had a total of eight links and therefore eight ways that a single link could suffer a permanent fault. Adrover tried all eight combinations of disconnecting one link at a time and disconnected each one of them for a duration of 20 elementary cycles, as shown in Table 9.3. In the tables, as in Figure 9.7, slaves are numbered 1–3 and the switches are labeled L and F, for leader and follower,

Table 9.3: Injected link crashes (one link at a time).

Target links	Affected ECs
l_{1L}	70–89
l_{1F}	90–109
l_{2L}	110–129
l_{2F}	130–149
l_{3L}	150–169
l_{3F}	170–189
l_{LF_1}	190–209
l_{LF_2}	210–229

respectively. Moreover, l_{ij_n} indicates the nth link between slave or switch i and switch j—if there is only one link between i and j, then n is omitted. All eight fault scenarios were tolerated.

After this experiment, Adrover proceeded with another one where he disconnected two links at a time. There are a total of $\binom{8}{2} = 28$ combinations for doing this. Of these 28 combinations, 24 can be tolerated such that communication is still possible among all three slaves. (The four combinations that do not allow all three slaves to communicate occur when both links of the first slave fail, both links of the second slave fail, both links of the third slave fail, and both interlinks fail.) Adrover tried these 24 combinations, as shown in Table 9.4, and verified that the faults were again tolerated.

Finally, there are total of $\binom{8}{3} = 56$ combinations in which three links can crash at the same time. Of these, 24 still allow all slaves to communicate. These 24 correspond to those combinations in which each slave has at least one non-faulty link connected to one of the switches and there is at least one non-faulty interlink between the switches. Adrover tried all these combinations, as shown in Table 9.5, again verifying that they were all tolerated thanks to the redundant paths provided by the network topology of FTTRS.

Injecting Transient Faults into Trigger Messages

Apart from the links and the FTTRS switches, another critical element for the correct operation of any FTT-based communication infrastructure are the trigger messages. In FTTRS each trigger message is proactively retransmitted k times by each master

Table 9.4: Injected link crashes (two links at a time).

Target links	Affected ECs	Target links	Affected ECs
l_{1L}, l_{2L}	70–89	l_{2L}, l_{3L}	310–329
l_{1L}, l_{2F}	90–109	l_{2L}, l_{3F}	330–349
l_{1L}, l_{3L}	110–129	l_{2L}, l_{LF_1}	350–369
l_{1L}, l_{3F}	130–149	l_{2L}, l_{LF_2}	370–389
l_{1L}, l_{LF_1}	150–169	l_{2F}, l_{3L}	390–409
l_{1L}, l_{LF_2}	170–189	l_{2F}, l_{3F}	410–429
l_{1F}, l_{2L}	190–209	l_{2F}, l_{LF_1}	430–449
l_{1F}, l_{2F}	210–229	l_{2F}, l_{LF_2}	450–469
l_{1F}, l_{3L}	230–249	l_{3L}, l_{LF_1}	470–489
l_{1F}, l_{3F}	250–269	l_{3L}, l_{LF_2}	490–509
l_{1F}, l_{LF_1}	270–289	l_{3F}, l_{LF_1}	510–529
l_{1F}, l_{LF_2}	290–309	l_{3F}, l_{LF_2}	530–549

Table 9.5: Injected link crashes (three links at a time).

Target links	Affected ECs	Target links	Affected ECs
l_{1L}, l_{2L}, l_{3L}	70–89	l_{1F}, l_{2F}, l_{3L}	310–329
l_{1L}, l_{2L}, l_{3F}	90–109	l_{1F}, l_{2F}, l_{3F}	330–349
l_{1L}, l_{2L}, l_{LF_1}	110–129	l_{1F}, l_{2F}, l_{LF_1}	350–369
l_{1L}, l_{2L}, l_{LF_2}	130–149	l_{1F}, l_{2F}, l_{LF_2}	370–389
l_{1L}, l_{2F}, l_{3L}	150–169	l_{2L}, l_{3L}, l_{LF_1}	390–409
l_{1L}, l_{2F}, l_{3F}	170–189	l_{2L}, l_{3L}, l_{LF_2}	410–429
l_{1L}, l_{2F}, l_{LF_1}	190–209	l_{2F}, l_{3L}, l_{LF_1}	430–449
l_{1L}, l_{2F}, l_{LF_2}	210–229	l_{2F}, l_{3L}, l_{LF_2}	450–469
l_{1F}, l_{2L}, l_{3L}	230–249	l_{2L}, l_{3F}, l_{LF_1}	470–489
l_{1F}, l_{2L}, l_{3F}	250–269	l_{2L}, l_{3F}, l_{LF_2}	490–509
l_{1F}, l_{2L}, l_{LF_1}	270–289	l_{2F}, l_{3F}, l_{LF_1}	510–529
l_{1F}, l_{2L}, l_{LF_2}	290–309	l_{2F}, l_{3F}, l_{LF_2}	530–549

in each elementary cycle. As long as in the experiments at least one of the k trigger messages reached each slave, consistency among slaves would be ensured: they would all agree on the number of elapsed elementary cycles, on which messages have been scheduled for each of them, and on what master command messages have been broadcast. It would therefore in principle suffice for a single trigger message to get through only one of the two links of a slave in each elementary cycle. However, one of the design goals of FTTRS was for the tolerance of permanent faults and transient faults to be orthogonal (see end of Section 8.1.1 and comments on non-requirement NR3 on page 151). Thus, according to the design, at least one trigger message should get through each of the two links of each slave, assuming the links have not suffered permanent faults.

Of the 2^k scenarios in which k trigger messages can be corrupted or not in a link, only a single one may not be tolerated, namely, the one in which all k trigger messages are corrupted (whether it is tolerated or not, depends on whether a trigger message got through another corresponding link). This means that there are $2^k - 1$ ways of losing trigger messages on a link that should definitely be tolerated. For s slaves with two links each, there are therefore a total of $(2^k - 1)^{2s}$ possible scenarios of corrupting trigger messages of a given elementary cycle that should be tolerated. In contrast, if each slave had only one link remaining, then there are a total of $(2^k - 1)^s$ possible scenarios that should be tolerated.

In order for our experimentation to be feasible and take a reasonable amount of time, Adrover did only two experiments, one where the slaves had two links and one where they only had one.

In the first experiment he used a value of $k = 4$, two switches, and two slaves connected to both switches ($s = 2$). This yielded $(2^k - 1)^{2s} = (2^4 - 1)^4 = 50\,625$ possible ways of injecting trigger message omissions that should be tolerated. Adrover injected trigger message omissions to bring about each one of these scenarios.

In the second experiment he used a value of $k = 6$, three slaves ($s = 3$), and two switches, but with all slaves only connected to one of the switches. In this case we obtained $(2^k - 1)^s = (2^6 - 1)^3 = 250\,047$ possible ways of injecting trigger message omissions that should be tolerated. Adrover tested each one of them.

Adrover verified that in both experiments all of the involved slaves agreed upon the number of elapsed elementary cycles and that they correctly transmitted the messages scheduled for each elementary cycle. We conclude from this that all the faults injected in the trigger messages were tolerated by the FTTRS prototype.

With more time and resources, we could have tested hybrid scenarios, i.e., scenarios were some slaves had one link and others had two links. We could also have tried scenarios with other parameters, e.g., more slaves or different values for the trigger message redundancy level k. And we could have tested scenarios were trigger messages omissions were also injected into the interlinks. Nevertheless, an exhaustive test of all scenarios that FTTRS should tolerate according to its design would not have been possible in a reasonable amount of time—or at all, if we would have wanted to verify that the prototype works with the infinite possible values for the parameters k and s.

Despite the limitations, with our proof-of-concept prototype we have been able to back up the claim that the FTTRS design is viable. Indeed, we experimentally verified that the prototype correctly tolerates a large number of the crash and omission failures that according to the FTTRS design it should tolerate. Specifically, we verified that it tolerates the crash of either switch, the crash of any single link, the crash of two or three links that still result in a connected network topology, and the omission of multiple trigger messages per elementary cycle. We did not encounter any scenario that according to the design should be tolerated, but was not.

9.6 Testing FTTRS as the Communication Subsystem for a Control Application with Replicated Nodes

I designed FTTRS within the context of the FT4FTT project. The goal of this project was the design and implementation of an infrastructure to provide fault tolerance and flexibility to all crucial parts of a real-time distributed embedded system. For this FTTRS was one of the key components; the other was a node replication scheme, which Derasevic designed.[14] To successfully conclude the FT4FTT project, we had to combine these two key components. Specifically, we had to test that Derasevic's fault-tolerant node replication scheme works on top of the fault-tolerant communication subsystem, i.e., FTTRS. The corresponding integration test was performed by Ballesteros.[15]

For the integration test Ballesteros developed a new experimental setup that allowed us to test the behavior of a system running in a real-time environment and with

[14] Derasevic, Proenza, and Barranco, "Using FTT-Ethernet for the Coordinated Dispatching of Tasks and Messages for Node Replication"; Derasevic, Barranco, and Proenza, "Designing Fault-Diagnosis and Reintegration to Prevent Node Redundancy Attrition in Highly Reliable Control Systems Based on FTT-Ethernet".

[15] Ballesteros et al., "First Implementation and Test of a Node Replication Scheme on Top of the Flexible Time-Triggered Replicated Star for Ethernet"; Alberto Ballesteros et al. "First Implementation and Test of Reintegration Mechanisms for Node Replicas in the FT4FTT Architecture". In: *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*. IEEE. 2016; Alberto Ballesteros. "Implementation and Testing of the Node Replication Scheme of the FT4FTT Architecture". MA thesis. Universitat de les Illes Balears, 2016.

mixed traffic requirements, i.e., both real-time and non-real-time traffic. The setup implemented an inverted pendulum using the **hardware-in-the-loop** technique, which consists in having a hardware system that controls a simulated plant. In our integration test specifically, the hardware was comprised of an FTTRS prototype with node replication on top. The simulated plant was the inverted pendulum, which was implemented in Simulink[16] and for which Francisca Font implemented the necessary interface to interact with the FT4FTT prototype. Using a simulated pendulum simplified the testing: we did not have to build or acquire an inverted pendulum, and we did not have to waste time maintaining a physical device that may break down or waste time physically setting up the pendulum for new experiments. Moreover, a simulated pendulum allowed us to use a lower gravitational acceleration than the Earth's standard acceleration of $9.8\,\mathrm{m\,s^{-2}}$. Specifically, we used a value of $5\,\mathrm{m\,s^{-2}}$. This made the pendulum a little bit more sluggish and ensured that we did not have to worry too much about the performance of our proof-of-concept prototype and could instead focus on other aspects that were key within the FT4FTT project (fault tolerance, error containment, not missing deadlines, replica determinism, etc.). In particular, having a more sluggish pendulum allowed us to use software implemented fault injection without having to worry about deadline misses caused by the extra latency induced by the fault injection mechanisms.

The final prototype comprised one PC simulating the inverted pendulum; two FTTRS switches; three FTTRS slaves, which were replicas controlling the inverted pendulum; and two legacy nodes, i.e., nodes with no specific FTTRS or FTT drivers, that exchanged a video stream. As in the previous prototypes, the FTTRS switches were implemented using a multicore PC for each and the slaves and legacy nodes were implemented using a small barebone for each. Figure 9.10 shows a picture of this final prototype. The tower PCs to the left and right are the switches, on top of each tower is one of the legacy nodes that exchange video (labeled "video server" and "video client"), and at the bottom, under the display, we find the three replicated slaves running Derasevic's node replication mechanism on top of FTTRS. Each replica was connected by a dedicated slave link to each of the switches, the switches were interconnected by means of two interlinks, and each legacy node was connected to only one of the switches.

The prototype executed three applications at the same time.

First, the replicas controlled the angle and position of the inverted pendulum by running two PID controllers each and by voting on the control outputs. This allowed us to check that the node replication scheme works on top of FTTRS and

[16] MathWorks. *Control of an Inverted Pendulum on a Cart*. URL: https://es.mathworks.com/help/control/examples/control-of-an-inverted-pendulum-on-a-cart.html (visited on 2016-11-03).

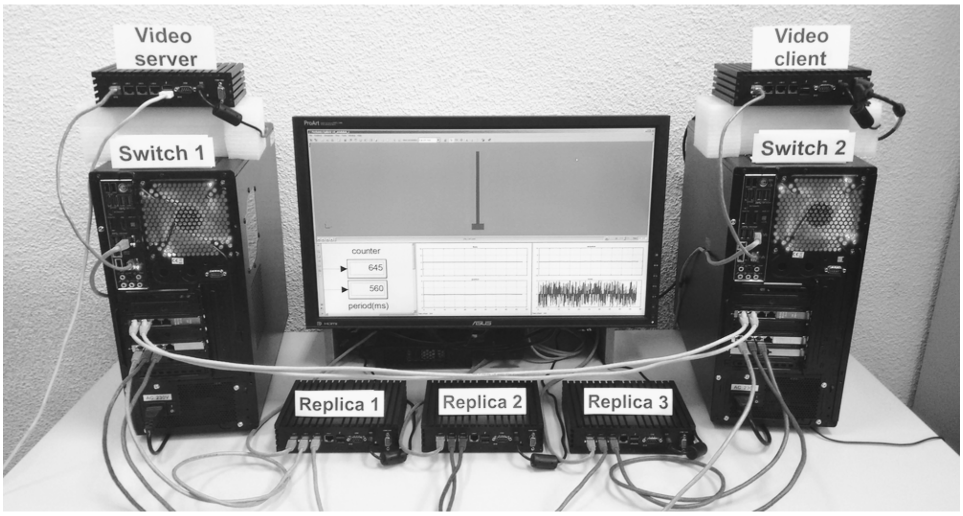


Figure 9.10: FT4FTT prototype that uses FTTRS as the underlying communication subsystem.

that FTTRS supports the exchange of real-time messages.

Second, replicas 2 and 3 not only executed the PID controllers, but also exchanged the value of a counter, whose periodicity could be changed at runtime. This allowed us to test the flexibility of the communication. In particular, the period specified how often the counter should be increased and how often its value should be exchanged via synchronous messages through the network. This period was stored as a real-time parameter in the synchronous requirements tables (SRTs) of the NRDBs and SRDBs. Upon a request from one of the slaves—and the subsequent consistent admission control executed by the FTTRS switches—this parameter could be changed at runtime. Ballesteros therefore also showed that the prototype has the desired operational flexibility.

Finally, the video server sent a non-real-time video stream to the video client, generating additional traffic that did not interfere with the real-time control of the inverted pendulum. This might be surprising since FTTRS, as we have designed it, does not take into account legacy nodes (see non-requirement NR1 in Section 8.1.2). In particular, FTTRS does not have mechanisms to restrict the failure semantics of such nodes. Nevertheless, the legacy video nodes did not cause problems in Ballesteros's experiments. First, they did not interfere with the transmission of trigger messages or real-time data messages because switches simply confined their traffic to any spare bandwidth at the end of each asynchronous window. Second, Ballesteros did not inject any faults into the video nodes and their behavior was therefore fault free. The fact that FTTRS worked correctly with non-faulty legacy nodes is good news since it supports the claim that FTTRS, even though it was not a design goal, can currently already support such nodes if these do not fail or have fail-silent failure semantics.

To demonstrate the tolerance of the whole system to permanent faults, Ballesteros carried out two fault-injection campaigns.

In the first, he tested the tolerance of FTTRS and the node replication scheme to the crash of either switch. Specifically, he carried out two experiments, each one involving the crash of one of the two switches. In both experiments the system was able to continue its operation seamlessly, without any notable disturbance of the control of the inverted pendulum.

In the second fault-injection campaign, Ballesteros tested the tolerance of FTTRS and the node replication scheme to permanent faults affecting the links. Specifically, he provoked all the tolerable combinations of permanent failures in the slave links and interlinks. That is, given that there were 8 links involved in the control of the pendulum (two for each of the three slaves and two for the interlinks), there were a total of $2^8 = 256$ different combinations for links to fail. Of these, FTTRS in combination with node replication can tolerate 162. The combinations that could not be tolerated were those where all interlinks failed and those that disconnected a majority—two or more—of the slave replicas. As intended, all 162 error scenarios were tolerated without the control of the inverted pendulum being noticeably disturbed.

Ballesteros also tested the tolerance of the whole FT4FTT prototype to faults within slaves and transient faults in links that last so long that they corrupt all temporal replicas of a given message. Specifically, he tested Derasevic's mechanism to reintegrate slave nodes that have become faulty due to such faults. In particular, the work showed that by using appropriate reintegration mechanisms for the slaves, a control system relying on FTTRS can continue to provide its service even if occasionally FTTRS fails to deliver a message to some slaves (e.g., because FTTRS was misconfigured and the redundancy level chosen for the trigger messages or data messages was too small given the actual number and duration of transient faults). In other words, the key idea in that work was that even if FTTRS fails to deliver a correct FTT service for some slaves, and these consequently become faulty, the service provided by the slaves is not necessarily doomed—the slaves can take measures to recover correct internal states and become correct again. For this Ballesteros performed several tests.

In the first set of tests he forced the omission of all trigger messages on a given pair of slave links during a given elementary cycle. That is, he made some slaves not receive any trigger message at all. This caused the corresponding slave to become faulty and lose replica determinism with the other slave replicas. Ballesteros's experiments, however, showed that the faulty slave was subsequently able to reintegrate itself into its replica group using Derasevic's reintegration mechanism.

In the second set of experiments Ballesteros injected transient link faults that prevented some slaves from transmitting or receiving their data messages. Derasevic's voting mechanism, working on top of FTTRS, was able to tolerate these scenarios.

In the third, fourth and fifth set of experiments, Ballesteros injected faults not into network components, but into the slaves themselves. Again, Derasevic's mechanisms on top of FTTRS tolerated these faults and allowed faulty slaves to reintegrate themselves into their replica group.

More details on these experiments can be found in the corresponding paper by Ballesteros et al.,[17] as well as in the master thesis of Ballesteros.[18] What is important in the context of this dissertation is that all these experiments show that FTTRS correctly integrates with a node replication and reintegration mechanism. Moreover, the experiments show that if FTTRS is configured with message redundancy levels that can occasionally be overwhelmed by longer-than-expected transient faults, the overall system may still be able to tolerate these faults by layering appropriate mechanisms on top of FTTRS.

Finally, a video showing a demonstration of the final FT4FTT prototype, which uses FTTRS as the underlying communication subsystem, can be found on the website of the UIB's Systems, Robotics, and Vision group.[19] This video not only shows that FTTRS supports a real-time application (an inverted pendulum), but also that it does so while the failure of network components is tolerated and the real-time requirements of messages are modified at runtime (the video shows how through a gamepad one of the slave nodes is instructed to request changes in the periodicity of the synchronous message stream that carries the previously mentioned counters, with the requests then consistently being accepted by the FTTRS switches). More details on this and other aspects of using FTTRS within an FT4FTT prototype can be found in the master thesis of Ballesteros.[20]

[17] Ballesteros et al., "First Implementation and Test of Reintegration Mechanisms for Node Replicas in the FT4FTT Architecture".

[18] Ballesteros, "Implementation and Testing of the Node Replication Scheme of the FT4FTT Architecture".

[19] Alberto Ballesteros. *FT4FTT final prototype demo*. 2017. URL: http://srv.uib.es/ft4ftt-final-prototype-demo/ (visited on 2017-03-08), Also available directly on YouTube: https://www.youtube.com/watch?v=tIU11dOpB_k.

[20] Ballesteros, "Implementation and Testing of the Node Replication Scheme of the FT4FTT Architecture".

9.7 Discussion

Our goal in this chapter was to show that the mechanisms of FTTRS are feasible, i.e., that they can be implemented in practice and that FTTRS thus meets the last requirement (R4, page 149). The goal was not to build an end product. Nor was it to validate FTTRS, i.e., to show that FTTRS meets its high-level purpose—we already did so previously (Chapter 8, in which we argued why FTTRS meets its requirements).

To convince ourselves that FTTRS is feasible we have to convince ourselves that each of its mechanisms is feasible and that they can be put together. The main mechanisms of FTTRS are the following:

- Restriction of the failure semantics of FTTRS switches by means of internal duplication with comparison.
- Restriction of the failure semantics of slaves by means of port guardians.
- An architecture to tolerate permanent faults through spatial redundancy.
- Proactive retransmissions to make messages fault tolerant when facing transient faults.
- Slave elementary cycle synchronization, which in turn relies on
  - isochronous transmission of trigger messages,
  - master lockstep synchronization, and
  - masters conveying consistent elementary cycle synchronization information in their trigger messages.
- Replica determinism of the FTTRS switches, which in turn relies on
  - the FTTRS switches starting out with consistent SRDBs and SCSRs,
  - the FTTRS switches being internally deterministic,
  - the FTTRS switches being able to reliably exchange minimum update requests piggybacked on trigger messages,
  - a total ordering of update requests, and
  - the FTTRS switches sharing a common notion of time, i.e., elementary cycle synchronization between masters.
- System-wide update of real-time requirements, which in turn relies on

- the slaves and masters starting out with consistent NRDBs and SRDBs, respectively,
- the FTTRS switches being replica deterministic,
- master command messages being piggybacked on trigger messages, and
- slaves and masters having sufficient processing power to complete relevant tasks within the necessary timeframe (e.g., as shown in the last elementary cycle of Figure 8.21 on page 220, slaves and masters must have sufficient processing power to prepare a database commit in time, before the commit has to be made effective at the end of the elementary cycle to trigger a system-wide commit).

Alberto Ballesteros, Inés Álvarez, Andreu Adrover, and Francisca Font have helped showing the feasibility of some of these mechanisms through proof-of-concept prototypes. For other mechanisms we still have to show their feasibility, which we shall do through arguments. Let us go through each of the above-listed mechanisms in turn and see why they are feasible.

Restricting the failure semantics of FTTRS switches to make them fail silent relies on internally duplicating and comparing them. Internal duplication with comparison is a well-known technique that others have already implemented for various devices and architectures over the decades. In fact, even for Ethernet switches the technique has already been used.[21] It therefore did not seem worthwhile within the scope of my PhD to invest the significant engineering work required to actually build an internally duplicated and compared FTTRS switch. The prior work of others already corroborates the feasibility sufficiently.

Similarly, the feasibility of restricting the failure semantics of slaves by means of port guardians is already substantiated by prior work. Indeed, the feasibility of guardians has already been demonstrated in several protocols, such as TTEthernet, FlexRay, TTP, and CAN. It has even been demonstrated for HaRTES, although in the corresponding publication the guardians were called *validation units*.[22] FTTRS just extends the capabilities of these units. Moreover, the feasibility of the port guardians is further corroborated by the fact that they implement functionality analogous to a firewall (e.g., deep packet inspection, dropping of unauthorized packets, dropping of packets whose arrival rate exceeds a threshold to prevent denial of service attacks, etc.) and network devices with built-in Ethernet switches and firewalls have long been commercially available.

[21] Poledna, “Method and Switching Unit for the Reliable Switching of Synchronization of Messages”.

[22] Santos et al., “A Synthesizable Ethernet Switch with Enhanced Real-Time Features”.

As to the feasibility of the FTTRS architecture, this we did demonstrate through proof-of-concept prototypes. Moreover, the corresponding experiments not only verified the correct implementation of the prototypes, but also further validated the capacity of the replicated architecture to tolerate the failure of either FTTRS switch and to tolerate permanent failures of slave links and interlinks (Section 9.5). For slave links and interlinks specifically, we validated all the tolerable scenarios involving up to two simultaneous permanent link crashes.

Making messages reliable through proactive retransmissions is undoubtedly feasible. Not only is it a technique that has already been used by other protocols (examples include MARS,[23] TTP,[24] and GOOSE[25]), but we have implemented it ourselves for trigger messages (Section 9.5).

That it is feasible to implement the slave elementary cycle synchronization mechanism of FTTRS has been demonstrated by two proof-of-concept prototypes, one involving a virtualized switch and one involving a COTS switch (Section 9.3). Through experiments we verified that the relevant mechanisms implemented in both prototypes corresponded to the design. Moreover, since the experiments showed that the slaves, as desired, synchronized their elementary cycles (although with limited precision due to the nature of the proof-of-concept prototypes), we also further validated the correctness of the elementary cycle synchronization mechanism.

To demonstrate the feasibility of isochronous lockstep transmission of trigger messages by masters, we implemented an additional proof-of-concept prototype (Section 9.4). The corresponding experiments verified that the prototype corresponded to the design and also further validated the mechanism to achieve lockstep synchronization.

The replica determinism of the FTTRS switches is also feasible. Making FTTRS switches start out with consistent SRDBs and SCSRs is just a matter of properly preconfiguring them, which is clearly feasible. Making them internally deterministic is also feasible: switches do not require any nondeterministic program constructs, true random number generators, or other components that exhibit nondeterministic behavior. If they are implemented identically in hardware they would also have the same processing power and could finish their calculations and decisions in the same elementary cycle; that is, one switch would not take significantly longer than the other for a computation. Since minimum update requests are piggybacked onto trigger messages, ensuring that FTTRS switches can reliably exchange them relies on the trigger messages being proactively retransmitted, which I already pointed

[23] Kopetz et al., “Distributed Fault Tolerant Real-Time Systems: the MARS Approach”.

[24] Kopetz and Grunsteidl, “TTP — A Protocol for Fault-Tolerant Real-Time Systems”.

[25] IEC, *Communication networks and systems in substations — Part 8-1.*

out to be feasible. It is also feasible to have a total ordering of update requests. For instance, as suggested earlier (Section 8.7.4), node identifiers and sequence numbers can be used for this, which are both standard features of many, if not most, protocols. Finally, having the FTTRS switches share a common notion of time is also feasible: we showed it ourselves in the experiments related to synchronizing two FTTRS masters (Section 9.4). Thus, since all this can be implemented in practice, the FTTRS switches can be replica deterministic in an actual implementation. In particular, the masters can be replica deterministic and keep their SRDBs consistent, and consequently the internal switches and port guardians can also be replica deterministic (as I explained in Section 8.7.4, internal switch and port guardian replica determinism is a consequence of master replica determinism).

Regarding a system wide update of real-time requirements, it is also feasible. First, it is feasible for the slaves and masters to start out with consistent NRDBs and SRDBs. It is just a matter of properly preconfiguring them before the system begins its operation. Second, as discussed above, FTTRS switches can be made replica deterministic in practice. Third, master command messages can be piggybacked onto the fault-tolerant trigger messages as long as they fit within a maximum sized Ethernet frame payload (1500 bytes) together with the other information carried in trigger messages. And fourth, it is generally possible to have slaves and masters with sufficient processing power to complete relevant tasks within the necessary timeframe. This is so because the timeframe available is always determined by the length of the elementary cycles and this length is a parameter that can be increased before deployment in case the slaves or masters are too slow (assuming the application can work with larger elementary cycles).

The final proof-of-concept prototype that put a control application on top of FTTRS (Section 9.6) demonstrated the feasibility of integrating the different mechanisms of FTTRS. It also demonstrated the feasibility of integrating these mechanisms with a higher-layer replicated control application, i.e., an application to control an inverted pendulum in a redundant manner. And although not a priority in the context of this dissertation, it also showed that FTTRS correctly integrates with the node replication mechanism developed by Derasevic within the FT4FTT project.

Finally, even though the goal of the experiments was to verify the feasibility of FTTRS, and not to exhaustively validate its correctness—for which experimentation is not the right approach anyway[26]—it is still worth pointing out that none of the

[26] Experimentation can reveal design faults, but generally it cannot prove the correctness of algorithms and protocols. As Dijkstra put it, "[...] testing can be used very effectively to show the presence of bugs but never to show their absence" (Edsger W. Dijkstra. *On the reliability of programs*. No date. URL: `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html` [visited on 2017-03-16]). Einstein is also reported to have said a similar thing: "No

experiments uncovered any flaws in the design of FTTRS.

amount of experimentation can ever prove me right; a single experiment can prove me wrong”.

Chapter 10

Conclusions and Future Work

I motivated this dissertation with our visions of the future—with the futuristic distributed embedded systems for which we have longed for in countless works of science fiction. I pointed out that many of these future systems would have to be highly reliable, hard real-time, and adaptive. I then further pointed out that to have these attributes, the system would have to rely on an internal network that is itself highly reliable and that provides support for hard real-time communication. Moreover, although the network would not have to be adaptive itself, it would have to be flexible. More specifically, it would have to provide operational flexibility, meaning that it would have to be amenable to being changed at runtime. In particular, for the network to support an adaptive application, it would have to be able to change upon requests from the application that uses the network.

I remarked on the value of building such highly reliable, hard real-time, and flexible networks using the pervasive Ethernet technology, which has many desirable properties, such as high bandwidth, low cost, and widespread expertise for it. I observed, however, that Ethernet in its current implementations is inadequate for such networks. Specifically, I said (page 7) that the problem is that *there is currently no Ethernet-based communication subsystem available that provides operational flexibility with hard real-time and reliability guarantees*.[1]

The above-mentioned problem established the context of our work. To make strides towards addressing it, we set out to tackle a narrower one in this dissertation: we decided to focus on fault tolerance, which is one of the means to achieve high reliability, and to focus on one specific type of operational flexibility, namely,

[1] The set of IEEE standards known as Time-Sensitive Networking (TSN) may turn out to have the necessary operational flexibility while supporting hard real-time applications and being highly reliable, but as of this writing, it is still actively being developed.

flexibility towards changing the real-time requirements of periodic and sporadic real-time messages. We chose as a starting point the Flexible Time-Triggered (FTT) communication paradigm, which already provides support for real-time communication and has the necessary flexibility. And we then set out to prove the following thesis (page 10):

> *We can add tolerance to coincident permanent and transient faults to the FTT communication paradigm, using Ethernet as the underlying technology, while maintaining the paradigm's main features: support for the timely exchange of both periodic and sporadic real-time messages, and support for updating the real-time parameters of these messages at runtime.*

To prove the thesis we expressed it as a series of requirements and then designed, validated, and showed the feasibility of a communication subsystem that meets them. The requirements we stated were as follows (Section 8.1.1):

> R1: The communication subsystem is based on the FTT paradigm.
>
> R2: The communication subsystem is based on Ethernet.
>
> R3: The communication subsystem provides an FTT service even if both permanent and transient faults occur.
>
> R4: The communication subsystem can be implemented in practice.

The third requirement, R3, is expressed in terms of an *FTT service*, which we defined (page 148) to consist of

> (S1) consistently transporting real-time messages, which may be sporadic or periodic, from a sender to the appropriate receivers,
>
> (S2) doing so without violating any deadlines and
>
> (S3) while allowing the real-time parameters of the messages to change at runtime.

Moreover, we qualified the third requirement by breaking it down into the following subrequirements:

> R3.1: The failure of a slave node does not disrupt the FTT service provided to other slave nodes.

> R3.2: The maximum duration of transient faults that the communication subsystem can tolerate can be parameterized.
>
> R3.3: The communication subsystem provides an FTT service even if any one of its components, no matter which one, suffers a permanent fault.

To come up with a communication subsystem that meets all requirements, and thus prove our thesis, we built on the most promising implementation of FTT for Ethernet in terms of reliability, namely, HaRTES (Section 7.2.3). The result was a new communication subsystem for distributed embedded systems.

Like any system designer, we began the design by making assumptions. The most important one was our fault model, which we assumed to include non-malicious operational hardware faults and to exclude development, software, and malicious faults (Section 8.2).

Given the fault model, we proceeded with our design. First, we eliminated the single point of failure that the switch in HaRTES constitutes. We did so by choosing a duplicated architecture for our communication subsystem. Specifically, we chose a replicated star topology with two switches in the center, each of them embedding an FTT master, just like in HaRTES. Moreover, we interconnected the switches using interlinks so that they could communicate with each other and resolve any inconsistencies that might prevent them from being replica deterministic. Also, we chose multiple redundant interlinks to prevent a network partition in case of an interlink failure. Finally, we connected each slave to each switch by means of a separate Ethernet link, each of which we called a slave link. The result was the architecture illustrated in Figure 8.2 on page 159. Since this architecture is based on FTT, a replicated star topology, and Ethernet, we named our communication subsystem Flexible Time-Triggered Replicated Star for Ethernet, or FTTRS for short.

To make it easier to meet our requirements, we proceeded by restricting the failure semantics of switches and slaves (Section 8.6). We chose to make switches fail silent through internal duplication with comparison. The failure semantics of the slaves, in contrast, we decided to restrict by means of port guardians located at the switch-side of each slave link (see Figure 8.6 on page 175). This resulted in slaves exhibiting restricted failure semantics towards the switches and other slaves, instead of byzantine failure semantics (Figure 8.9 on page 179 summarizes the exact failure semantics we obtained).

Next we took advantage of the restricted failure semantics and the duplicated architecture to design fault-tolerance mechanisms that allow FTTRS to meet requirement R3, all the while maintaining the key features of the FTT paradigm and continuing to use Ethernet as the underlying technology to satisfy requirements R1 and R2.

To meet requirement R3 we had to ensure that FTTRS satisfies each of the subrequirements R3.1, R3.2, and R3.3.

We began with R3.1 because it was the easiest to address (Section 8.7.1). After all, the port guardians we added to restrict the failure semantics of slaves already ensured that FTTRS meets requirement R3.1. Specifically, the port guardians already sufficiently restricted the failure semantics of slaves to ensure that these, upon failing, cannot disrupt the FTT service provided by the switches to other (non-faulty) slaves. Each guardian monitors the frames transmitted by a given slave and only lets them pass, and thus reach an embedded master or other slaves, when the frames carry FTT messages that are timely and have correct metadata. This ensures that a faulty slave cannot improperly delay legitimate messages from other slaves by sending more frames than it should or by sending them to inappropriate destinations. Similarly, a faulty slave cannot send so many frames to a switch that it overflows the queues of the switch, which, if it were possible, might lead a switch to lose legitimate frames. Moreover, given our fault model, which excludes development and software faults, no frames sent by a faulty slave can crash a switch, nor can they crash another slave or otherwise cause the failure of a switch or slave. Because of all this, a faulty slave cannot disrupt the FTT service provided to other slave nodes and requirement R3.1 is met.

Next, we took on requirement R3.2 because it would be a stepping stone towards meeting requirement R3.3.

To meet requirement R3.2 we only had to tolerate transient faults in links. Transient faults in slaves and switches did not have to be tolerated. Transient faults in slaves only need to be prevented from interfering with the communication (requirement R3.1), but do not need to be tolerated since tolerating slave faults is out of the scope of the communication subsystem. And as to transient faults in switches, they are in practice non-existent since they are effectively converted into permanent faults due to the switches being internally duplicated and compared. We therefore only had to deal with transient faults in links. Now, to tolerate transient faults in links we added a new parameter for each message which we called redundancy level (Section 8.7.2). The redundancy level specifies for a message how many copies of it the corresponding transmitter should transmit in a given elementary cycle through each link. Thus, a redundancy level of $k = 3$ for the trigger message would mean that each switch would broadcast three trigger message replicas in each elementary cycle, one after the other, through all slave links and interlinks attached to that switch. Similarly, a redundancy level of $k = 3$ for a message originating at a slave would tell that slave to transmit three replicas of the message within the same elementary cycle through each of its two links. In other words, we adopted a proactive retransmission approach where the number of retransmissions could be specified by a redundancy level. This ensured that FTTRS meets requirement R3.2 even though the parameterization is in terms of a redundancy level and not in terms of the maximum duration of transient faults. The key is that by knowing the maximum duration of transient faults and the maximum rate with which frames can be transmitted on the Ethernet links we can calculate how many consecutive frames may be corrupted at most and simply choose a redundancy level higher than the calculated number. That way it is guaranteed that at least one copy of each message on each link gets through uncorrupted and the transient fault is tolerated. In other words, that way FTTRS allows to parameterize the maximum duration of transient faults that can be tolerated and requirement R3.2 is met.

Having met requirement R3.2 through proactive retransmissions, however, raised a new problem. In HaRTES, and all other versions of FTT, slaves decide that a new elementary cycle begins as soon as they receive a trigger message. If there is only one trigger message per elementary cycle, and it is broadcast in parallel through all links, then all slaves receive the trigger message at approximately the same time and achieve a precise elementary cycle synchronization (assuming all links have the same propagation delay and bit rate). But we decided on transmitting k trigger messages per elementary cycle. Thus, to continue to provide a precise elementary cycle synchronization, we had to revise how slaves decide when an elementary cycle starts. The approach we chose was to synchronize all slaves with the arrival of the last trigger message in each elementary cycle. And to make this work even if a subset of trigger messages, including the last one, is lost due to transient faults, we decided to have the switches broadcast the trigger messages in such a way that the arrival time of the last trigger message can be predicted by a slave as long as it receives at least one uncorrupted trigger message. Specifically, we decided that the switches should broadcast the trigger messages isochronously and in lockstep. That way, whenever a slave receives the ith trigger message out of k in an elementary cycle at an instant of time t_i, it knows that the last one should arrive at time $t_k = t_i + (k - i)\tau$, where τ denotes the trigger message intertransmission time (see Figure 8.18 on page 197). Thus, upon the reception of the ith trigger message, the slave simply sets a timer to expire after $(k - i)\tau$ units of time to know when it should consider the new elementary cycle to begin—or more precisely, to decide when the turn-around time of the new elementary cycle should begin (see Figure 8.10 on page 187).

To meet the remaining requirement, R3.3, we had to ensure that FTTRS tolerates a permanent fault in any one of its components, no matter which one. Since we do not consider slaves part of the communication subsystem, this meant that we had to ensure that FTTRS tolerates any permanent fault occurring in an interlink, slave link, or switch. As we argued at the time (Section 8.7.3), this boiled down to ensuring that the switches are replica deterministic. Specifically, ensuring replica determinism of switches sufficed because of several key features of FTTRS that we already introduced. First, FTTRS has a duplicated star topology comprised of two HaRTES-based switches which are interconnected by redundant interlinks, with each slave node being connected to both switches by means of a dedicated slave link. As a result, FTTRS has a path interconnecting any pair of slaves even if any one of the components of the communication subsystem (switch, link, or interlink) fails (see Figure 8.6 on page 175). Second, in FTTRS all available communication paths are used all the time: each slave sends each message in parallel to both switches by means of a separate link for each switch. Moreover, switches exchange all messages they receive from slaves through the redundant interlinks. In the absence of faults this ensures that each slave receives multiple copies of each message due to a phenomenon we called replica radiation (Figure 8.13 on page 192); and in the presence of one permanent fault, it ensures that at least one copy gets through to the intended destination—even if there are additional transient faults (assuming that the redundancy level of the messages is sufficiently high). Third, switches are fail silent and links have inherently benign failure semantics (Section 8.6). As a result, neither the failure of a link nor of a single switch can prevent slaves and a surviving switch from exchanging messages through remaining non-faulty links. Nevertheless, slaves simply being able to exchange messages is not enough to meet requirement R3.3. What remained to be done was to ensure that either switch also continues to provide a consistent FTT service when the other fails. This called for enforcing the replica determinism of the switches.

To enforce the replica determinism of the FTTRS switches we had to enforce the replica determinism of each of their parts: their internal switches, their port guardians, and their embedded masters. We analyzed what it would entail for each of these components to be replica deterministic and concluded that the replica determinism of the internal switches and port guardians would be a consequence of the replica determinism of the embedded masters (see Section 8.7.4). Essentially, the argument was that how the port guardians and internal switches process a given frame is completely determined by their internal logic, by the contents of the SRDBs, and by when within a given elementary cycle (in which window) the frame is being processed. Thus, for port guardians and internal switches to be replica deterministic it is enough to ensure that they are implemented in the same way and without internal non-determinism and that the embedded masters are replica deterministic, which implies that they provide consistent timing information to the internal switches and guardians and that they have consistent SRDB contents.

To achieve the replica determinism of the embedded masters we distinguished between the time and value domain.

In the time domain we had to ensure that the masters agree when each elementary cycle starts and that they broadcast their trigger messages in lockstep. For this I proposed a semi-active replication approach, where one master would be designated the leader and the other the follower. Ballesteros, working out the details of my idea, then proposed a two-phase mechanism: at system startup the masters would execute an initialization phase based on a two-way time transfer and from then onwards the follower would maintain its elementary cycles synchronized with the leader by means of a process called periodic lockstep resynchronization. In the periodic lockstep resynchronization, the follower uses the proactively and isochronously broadcast trigger messages from the leader to decide whether it should advance or defer the broadcast of its own trigger messages. This ensures a fault tolerant synchronization because it is enough for the follower to receive one out of the k trigger messages that the leader broadcasts per elementary cycle, where k is the trigger message redundancy level. Moreover, when either master fails, the other continues unperturbed, broadcasting its trigger messages according to its own clock and thus continuing to provide the systemwide timing without the other.

As to the replica determinism in the value domain, after an analysis we concluded that it came down to ensuring three things: having the masters start out with consistent SRDBs, ensuring that the masters are internally deterministic by not using any non-deterministic logic (such as true random number generators), and ensuring that masters apply the same SRDB updates at the same time. The difficulty lies in the last: ensuring that masters apply the same updates at the same time. In most situations the masters should receive the same requests. After all, a slave transmits update requests multiple times in parallel through both its links and the switches forward the update requests to each other. The problem, however, is that in the presence of faults switches may nevertheless receive different requests—especially if the redundancy level used for the update requests is too low. Thus, we introduced an additional fault-tolerance mechanism for the embedded masters to agree on which update request to subject to admission control next in case they did not receive the same requests. This mechanism relies on the trigger messages, whose redundancy level, contrary to the one for update requests and other messages, is always assumed to be high enough to tolerate transient faults of maximum duration. The mechanism works as follows. We assume that all update requests from slaves are totally ordered, meaning that for any two update requests that are not replicas of each other we can always tell which comes before the other in terms of a total order relation. As we said at the time, the total order can be implemented in different ways and the details do not matter as long as it is indeed a total order (one option is to order slaves by their identifier and additionally use sequence numbers for update requests). Each master then keeps a set of pending update requests, which is a set containing all update requests it has received so far. To agree on which update request to submit to admission control next, each master first selects the minimum update request, according to the total order, among the ones it has in its set. We call this minimum the local minimum. In the next elementary cycle, each master then piggybacks its local minimum on the trigger messages it broadcasts. At the end of that elementary cycle's trigger message window, each master will then have the other's local minimum, which it then adds to its set of pending requests. Now each master selects a minimum again, but this time from its just updated set of pending requests. Both masters will select the same minimum because of the mathematical fact that for two totally ordered sets of pending requests Q_A and Q_B we have that $\min(\{\min(Q_B)\} \cup Q_A)) = \min(\{\min(Q_A)\} \cup Q_B))$. We call this identically selected minimum the global minimum. Once the global minimum is selected, both masters subject it to admission control. Now, due to the replica determinism in the time domain, both masters will conclude the admission control of the global minimum within the same elementary cycle. And since they are internally deterministic and started out with consistent SRDBs, they will reach the same conclusion on whether to accept or reject it. Thus, they will also generate the same master command messages to inform the slaves on the outcome of the admission control. These master command messages are then piggybacked on trigger messages and broadcast by the two masters during the next elementary cycle. Since the trigger messages are fault tolerant and broadcast in lockstep by the masters, this ensures that all non-faulty slaves receive the same commands by the end of the trigger message window, which enables a systemwide update of the real-time requirements at the end of the corresponding elementary cycle. Hence, not only are the masters kept replica deterministic in the value domain, but the whole system maintains consistent database (SRDB and NRDB) contents. And since replica determinism of the masters was the necessary prerequisite for tolerating the permanent failure of either switch, requirement R3.3 is satisfied.

The final requirement to meet was R4—we had to show that FTTRS is feasible. For this I decided to use proof-of-concept prototypes to demonstrate the feasibility of some mechanisms and to use arguments to show the feasibility of the remaining ones. The proof-of-concept prototypes implemented the mechanisms I considered to be the fundamental ones: elementary cycle synchronization among both slaves and masters, fault-tolerant trigger messages, and the fault-tolerant duplicated architecture of FTTRS that tolerates the failure of any one network component. All other mechanisms either built on these three or they have already been implemented in the past in analogous ways for various other protocols and architectures. We subjected the prototypes to experiments to verify them, i.e., to check that their behavior corresponded to the one specified by the design. Nevertheless, the experiments also served the purpose of further validating that FTTRS indeed meets requirements R1, R2, and R3.

By conceiving FTTRS, a communication subsystem that meets the requirements, we proved our thesis. In fact, in some aspects we even surpassed it. Notably, FTTRS, like all FTT versions, not only provides flexibility for changing real-time requirements, but also for other requirement changes, such as which nodes are publishers or subscribers of particular message streams. Moreover, FTTRS, contrary to any previous version of FTT, provides some degree of dynamic fault tolerance, that is, one of its fault-tolerance mechanisms is flexible itself. Specifically, the redundancy level of both synchronous and asynchronous slave messages is not only parameterizable offline, but at runtime, just like the real-time parameters of these messages. Also, in many scenarios FTTRS can tolerate more faults than required by R3.3. Specifically, it can tolerate the permanent failure of multiple components as long as there is still a path between any pair of slaves and one interlink remains. For instance, it can tolerate a scenario where each slave has lost a link and all but one interlink have failed.

10.1 Conclusions

Having proved our thesis, what can be observed or learned? What is important?

Well, most obviously we have learned that FTT, using Ethernet as the underlying technology, can indeed be made fault tolerant while maintaining its operational flexibility and support for hard real-time communication. That was, after all, what we set out to prove. But we have learned more.

We have learned that, in Ethernet, reliability, operational flexibility, and hard real-time support may not necessarily be mutually exclusive. Traditionally there was a chasm with highly reliable and hard real-time systems on one side, and flexible systems on the other. On the reliability and hard real-time side, communication requirements had to be static to provide high reliability and hard real-time guarantees. A prime example of this is avionics full-duplex Ethernet (AFDX), which is highly reliable and hard real-time, but has no operational flexibility (for instance, the number and parameters of AFDX virtual links have to be preconfigured and must remain static). On the flexibility side, requirements could change at runtime, but no guarantees for the timely and reliable delivery of messages could be given.

An example of this is traditional switched Ethernet in local area networks; there we can add or remove nodes at any time, increase or decrease at runtime the rate with which messages are exchanged, and run a myriad of different network, transport, and application layer protocols at various times without halting the network. Standard switched Ethernet therefore supports operational flexibility. It does, however, lack mechanisms to simultaneously also meet hard real-time and high reliability requirements. Despite these counterexamples, operational flexibility, hard real-time support, and high reliability surely can be combined in a single Ethernet network. First, the Ethernet implementations of the FTT paradigm showed that both operational flexibility and hard real-time support can be combined. FTTRS now added fault tolerance to the mix, showing that a single Ethernet network can also be simultaneously fault tolerant. And since fault tolerance is one of the means to achieve high reliability, FTTRS strongly suggests the viability of a single Ethernet network being highly reliable and supporting operational flexibility and hard real-time communication, all at the same time.

In fact, due to its flexibility, FTTRS might provide high reliability in environments where other highly reliable Ethernet-based communication subsystems cannot provide it. Specifically, FTTRS might provide high reliability in dynamic and unpredictable environments where continuous operation—and thus high reliability—can *only* be provided if the communication subsystem has operational flexibility. In such environments other competitors, such as the aforementioned AFDX, simply cannot provide a continued service, regardless of the reliability of their components or their fault-tolerance mechanisms. They would inevitably fail as soon as the environment changes and imposes new real-time requirements upon the communication.

This is not to say that there are no tradeoffs. In FTTRS we did combine fault tolerance, operational flexibility, and support for hard real-time communication, but we had to make some sacrifices. For instance, the proactive retransmissions reduce the bandwidth available for nodes to exchange data and thus the set of message streams that are schedulable in FTTRS is smaller than in HaRTES; the current design of port guardians to restrict the failure semantics of nodes cannot properly restrict the failure semantics of legacy nodes (e.g., ordinary laptops with no FTTRS-specific drivers); and FTTRS, as we designed it, lacks support for the addition of new nodes at runtime.

Regarding our emphasis on fault tolerance over other means to achieve reliability, such as fault prevention, it is worth pointing out that fault tolerance is becoming increasingly important for high reliability. This is so because newly manufactured computing devices have a tendency to be less reliable than those in the past. Indeed, with shrinking transistor sizes, growing transistor density, lower power voltages, and higher operating frequencies, integrated circuits are becoming more and more sensitive to fabrication defects, aging, electromagnetic interference, and other faults.[2] Thus, systems built out of modern high-performance components must rely more on fault tolerance to achieve the same overall reliability as in the past when lower-performance components were used. It has even gotten to the point that on-chip fault tolerance is increasingly required to make individual components sufficiently reliable.

We further learned that the FTT paradigm can be made fault tolerant for multiple underlying network technologies. In particular, it had been made fault tolerant for CAN and we have now also made it fault tolerant for Ethernet. Perhaps the paradigm might be applied to still further network technologies, where it might then also be made fault tolerant. Features that make the FTT paradigm especially suitable for adding fault tolerance to it include the following:

- Databases that store the current system parameters (SRDBs and NRDBs) and that can be used to distinguish correct from incorrect message transmissions. This is particularly helpful for error containment and the restriction of failure semantics.
- Related to the previous item, the predictability of what synchronous messages should be transmitted in the next elementary cycle is another feature that helps make FTT fault tolerant. That is, since an elementary cycle schedule dictates for each elementary cycle what synchronous messages should be transmitted, it is easy to detect when slaves are not transmitting the synchronous messages they should.
- The fact that asynchronous real-time messages are sporadic[3] and thus have minimum interarrival times. This makes it easier to detect timing failures, specifically, failures where a node transmits messages too soon.
- The fact that it does not rely on any type of randomness. This makes it easier to ensure that its components are internally deterministic.
- The division of the communication time into elementary cycles. This enables a divide-and-conquer approach for making FTT fault tolerant: by focusing on making the communication within an elementary cycle fault tolerant, we can

[2] Cristian Constantinescu. “Trends and Challenges in VLSI Circuit Reliability”. In: *IEEE Micro* 23.4 (2003), pp. 14–19; Todd Austin et al. “Reliable Systems on Unreliable Fabrics”. In: *IEEE Design and Test of Computers* 25.4 (2008).

[3] In some versions of FTT, such as FTT-SE, asynchronous messages also include aperiodic messages, i.e., those without minimum interarrival times and which are typically used for legacy non-real-time applications. But since aperiodic messages are treated differently in different versions of FTT, they do not have any features generic to the FTT paradigm that help us tolerate faults.

make the communication fault tolerant over the whole communication time. Moreover, the fact that the time granularity in FTT is the elementary cycle helps in enforcing replica determinism. After all, it simplifies how components (e.g., masters or slaves) reach agreement in the time domain: it is only by the end of an elementary cycle that they have to reach agreement, whether that is agreement on when a local event occured, which thus becomes global, or agreement on the values to be used as inputs by different components, or agreement on the order in which messages are delivered, or any other kind of agreement that may be necessary for replica determinism. Indeed, by only having to agree by the end of elementary cycles, the whole preceding time of each elementary cycle is available for the exchange of appropriate messages to reach agreement.

- The multitude of ways in which FTT can be implemented—such as with standalone masters, masters embedded within switches, bus topologies, and star topologies—which give the designer of a fault-tolerant communication subsystem a variety of options to choose from to maximize fault tolerance.

- The periodically broadcast trigger messages can act as a heartbeat, i.e., a periodic signal that indicates normal operation and whose absence helps detect the failure of the corresponding transmitter or one of its links.

There is, however, also one aspect of the FTT paradigm that I found to make it more difficult to make the paradigm work in the presence of faults. This is the fact that in FTT the elementary cycles are synchronized by means of the reception of a single trigger message by each slave in each elementary cycle. This makes it difficult to synchronize elementary cycles in protocols without error globalization, i.e., protocols such as Ethernet (as opposed to CAN) where not all nodes can easily agree on whether a frame has been successfully broadcast or not. After all, for the trigger message to be fault tolerant it may need to be retransmitted and without error globalization some slaves may only receive one copy of the trigger message and other slaves may only receive another. As we have seen, I solved the issue by having trigger messages be broadcast isochronously, which means that the arrival time of the last trigger message of an elementary cycle can be predicted given that at least one trigger message of that elementary cycle got through to each slave.

I also want to point out that each of the main mechanisms of FTTRS can be considered a contribution in its own right because each can be useful independently of the others. For instance, the port guardians provide error containment, which is useful on its own; the redundant transmission of trigger messages provides fault-tolerant synchronization of elementary cycles, which can also be useful on its own;

similarly, the fault-tolerant transmission of slave messages is useful on its own; and the replicated architecture that provides a path between any pair of slaves even if some link or switch fails is also useful independently of the other mechanisms.

Another important thing to note is that what we have created is a fault-tolerant communication subsystem with operational flexibility, but not an adaptive system. Adaptivity and flexibility are not the same (Chapter 1). A system is adaptive if it can change autonomously to deal with unpredictable circumstances. For that the system needs to be flexible, i.e., susceptible to being changed. More specifically, it must have operational flexibility, i.e., it must be susceptible to be changed at runtime. Operational flexibility is therefore a prerequisite of adaptivity. FTTRS provides this operational flexibility for Ethernet and therefore provides support for adaptive systems. To build a distributed embedded system that is adaptive, however, it is still necessary to put some intelligence on top of FTTRS that can sense environmental changes and respond to them, for which it might need to request appropriate changes to the communication subsystem.[4]

We also learned that star topologies, as was already the case for Controller Area Network (CAN),[5] are once again a viable means for providing a communication subsystem with significant error-containment capabilities and fault tolerance.

Finally, I think we can confidently say that FTTRS has a potential for being more reliable than HaRTES. We can conclude this even though we have not quantified the reliability of FTTRS nor HaRTES. Let us discuss this in more detail.

10.1.1 On the Reliability of FTTRS Versus HaRTES

As we know, reliability is the probability of continued correct service, i.e., the probability that a given system provides a continuous correct service without being interrupted by faults that lead to a failure. Thus, to see why FTTRS can be more reliable than HaRTES, let us see why FTTRS can be set up such that its service is less susceptible to fail due to faults. For this, let us consider transient and permanent faults separately.

[4] Putting an intelligence on top of FTTRS to obtain an adaptive system is currently being considered within the scope of the DFT4FTT project, which is the successor of the FT4FTT project and whose goal is to add dynamic fault tolerance to FTT.

[5] Barranco, Proenza, and Almeida, "Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate"; Manuel Barranco, Julián Proenza, and Luís Almeida. "Quantitative Comparison of the Error-Containment Capabilities of a Bus and a Star Topology in CAN Networks". In: *IEEE Transactions on Industrial Electronics* 58.3 (2011), pp. 802–813; Manuel Barranco, Julián Proenza, and Luís Almeida. "Quantitative Characterization of the Reliability of Simplex Buses and Stars to Compare their Benefits in Fieldbuses". In: *Reliability Engineering & System Safety* 138 (2015), pp. 163–175.

Transient Faults

Given the same environmental conditions, FTTRS can be set up such that transient faults are less likely to disrupt the FTT service in it than in HaRTES. This is so for several reasons.

First, if a transient fault occurs within an FTTRS switch, the fault effectively manifests the same way as a permanent crash because the switch is internally duplicated and compared. If that happens, however, the other switch continues providing a correct FTT service. In HaRTES, in contrast, although a transient fault within the HaRTES switch may not lead to a permanent crash and the switch is more likely to continue to provide some service, that service is likely to be incorrect. After all, in HaRTES there is no mechanism within the single switch to tolerate the fault. Hence, a transient fault in a HaRTES switch is likely to lead to a failure of the FTT service. For instance, a bit flip in a switch-internal data bus when reading the SRDB may corrupt the length of the elementary cycles read from the system configuration and status record (SCSR) and lead the trigger messages to be broadcast at the wrong time, or the bit flip may corrupt the deadline field of a synchronous message stream in the synchronous requirements table (SRT) and lead to incorrect scheduling and polling.

Second, although in this dissertation we have not added a mechanism to reintegrate a crashed FTTRS switch, there is the potential for doing so. In HaRTES, on the other hand, there is no such potential. In FTTRS therefore the system could continue working while a transiently faulty switch is recovering; whereas in HaRTES there is no potential for switch recovery while continuing to provide a correct FTT service.

Third, transient faults in slaves can cause byzantine behaviors in HaRTES (such as impersonations) that disrupt the whole communication, while in FTTRS they are prevented.

Fourth, if transient faults in links corrupt a message, then in HaRTES they inevitably lead to the loss of that message. And this is problematic: most types of messages must reach their destination for HaRTES to provide a correct service. For instance, trigger messages need to reach their destination to convey the elementary cycle schedule and timing, and real-time messages must reach their destination to not miss deadlines. The only messages whose loss may not lead to a failure are non-real-time messages from legacy applications. In FTTRS, on the other hand, when one message is corrupted, there is still the possibility for another copy of that message to reach its destination because each message is proactively retransmitted through multiple paths. In fact, if the redundancy level of the messages is correctly parameterized, then the probability of a scenario in which not a single copy of a message reaches its destination before negotiated deadlines is negligible (by negotiated deadlines I mean deadlines corresponding to message streams whose parameters have been accepted by the FTT admission control).

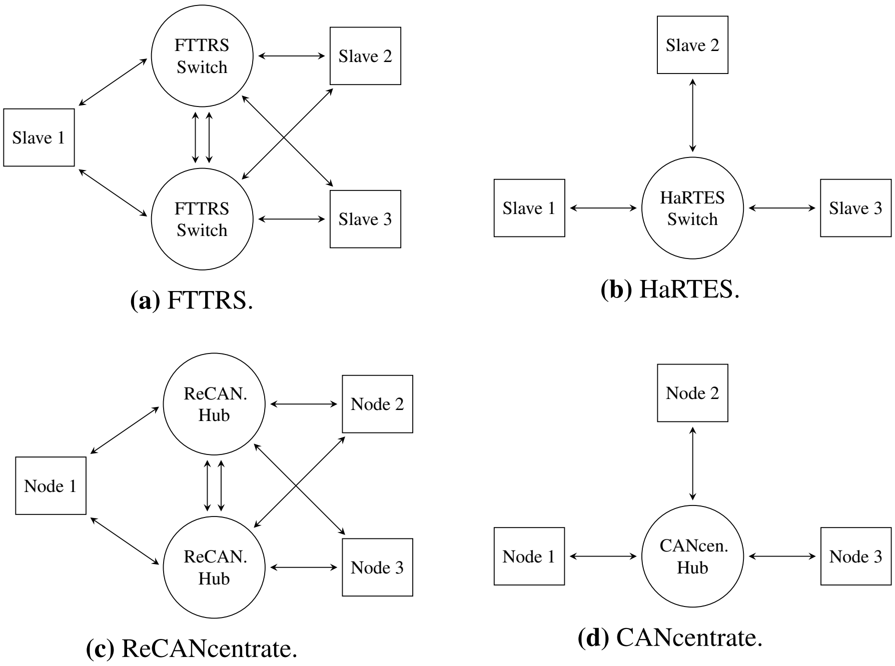
Finally, the additional hardware in FTTRS, and the consequent increase in the rate with which transient faults can occur, does not mean that FTTRS is less reliable. That is, FTTRS has more hardware than HaRTES and this means that under the same environmental conditions it can suffer more transient faults. Thus, the average rate with which transient faults occur in an FTTRS network is higher than in a HaRTES network serving the same number of slaves. Nevertheless, considering transient faults only, the reliability of FTTRS should still be higher since the additional transient faults that can occur in FTTRS are offset by the fact that a much smaller fraction of all transient faults can disrupt the FTT service in FTTRS. After all, although there are additional transient faults in FTTRS, these additional faults occur in additional components and these components, when they suffer transient faults, do not disrupt the FTT service. For instance, assume that a transient fault occurs, on average, once per minute in a HaRTES link and links are affected independently. With three slaves, there would be three links in HaRTES, and thus, on average, we would have three transient faults per minute in HaRTES. In FTTRS, under the same environmental conditions, we would still have one transient fault per minute per link. With two links per slave, three slaves, and two interlinks, FTTRS would have $2 \cdot 3 + 2 = 8$ transient faults per minute. Thus, FTTRS suffers more transient faults than HaRTES. However, if FTTRS has properly parameterized redundancy levels for its messages, these 8 transient faults are tolerated, whereas the three faults that occur per minute in HaRTES are, most likely, not.

In conclusion, if both a HaRTES network and an FTTRS network are subjected to the same rate of transient faults, then the FTT service in HaRTES is more likely to be interrupted and hence FTTRS is more reliable than HaRTES.

Permanent Faults

As to permanent faults, we can extrapolate to some extent the results published in a paper by my supervisors, Barranco and Proenza.[6] In the paper, Barranco and Proenza show that, when only permanent hardware faults are considered, ReCANcentrate—a duplicated star for CAN—generally has a higher reliability than CANcentrate—a simplex star for CAN. This is relevant for us because the architectures of these two star topologies for Controller Area Network are anal-

[6] Manuel Barranco and Julián Proenza. “Towards Understanding the Sensitivity of the Reliability Achievable by Simplex and Replicated Star Topologies in CAN”. in: *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2011)*. IEEE. 2011, pp. 1–4.



(a) FTTRS.

(b) HaRTES.

(c) ReCANcentrate.

(d) CANcentrate.

Figure 10.1: The architectures of FTTRS and HaRTES are analogous to the architectures of ReCANcentrate and CANcentrate, respectively.

ogous to the architectures of FTTRS and HaRTES, respectively. Essentially, the extrapolation is valid because what FTTRS does to HaRTES is analogous to what ReCANcentrate did to CANcentrate. The analogy is clear in

- the architecture, as can be seen in Figure 10.1, where each node is connected to two central elements that are interconnected to each other by means of several interlinks;
- the extra hardware of the double star compared with the simplex one, e.g., extra wires, connectors, communication controllers, transceivers, etc.;
- the additional error-containment of both duplicated stars, i.e., both of them include new mechanisms for each central element and new mechanisms for each node to contain errors that were not present in their simplex star counterpart; and
- the extra fault-tolerance capabilities, which make it possible for each node to communicate as long as it is connected to at least one non-faulty central element, no matter which one.

Because of this analogy, when facing permanent hardware faults only, we can conclude that FTTRS is more reliable than HaRTES under the same conditions under which ReCANcentrate is more reliable than CANcentrate. And what are these conditions? Essentially all conditions except those where the central element (the HaRTES switch or the CANcentrate hub) is extremely reliable. The edge case provides some intuition for this: if the central element never fails, then duplicating it is pointless for reliability. Actually, it is even worse than pointless: if the central element never fails, the additional cables and hardware in the duplicated central element only give more opportunities of failure for the duplicated star. If, however, the central element has a non-negligible probability of failing, a duplicated star can improve the reliability.

After discussing it with Barranco and Proenza, I have only found one aspect that may pose some uncertainty about how the results of ReCANcentrate and CANcentrate can be extrapolated to the comparison between FTTRS and HaRTES: the fact that in FTTRS each central element is internally duplicated and compared, whereas in ReCANcentrate they are not. As a result, the failure rate of the central element of the double star with respect to the simplex star is higher when we compare FTTRS to HaRTES than if we compare ReCANcentrate to CANcentrate. In other words, the amount by which the failure rate of an FTTRS switch surpasses the failure rate of a HaRTES switch is higher than the amount by which the failure rate of a ReCANcentrate hub surpasses the failure rate of a CANcentrate hub. The reason is twofold:

- in FTTRS the internal circuitry of the central element is more complex in comparison and, thus, more prone to suffering from permanent faults; and
- any transient fault affecting the FTTRS switch internal circuitry will be transformed into a permanent one.

Despite this difference, I think that the results of Barranco and Proenza can still be extrapolated to FTTRS and HaRTES.

First, the analysis of Barranco and Proenza assumed that the hubs in ReCANcentrate and CANcentrate had failure rates of the same order of magnitude and the switches in FTTRS and HaRTES should also have failure rates of the same order of magnitude. This is so because a pessimistic estimation based on the *parts-count method*[7] of the MIL-HDBK-217 standard[8] indicates that the rate of permanent faults affecting the internals of a switch is at worst twice as large in FTTRS than in HaRTES, which does not constitute a change of order of magnitude. (It is approximately twice as large because an FTTRS switch is internally duplicated whereas a HaRTES switch is not, and the additional hardware for the internal comparison and new mechanisms should be negligible.)

Second, one of the most interesting conclusions of Barranco and Proenza is the aforementioned fact that simplex stars only have higher reliability than duplicated ones when the central elements are extremely reliable. This, however, is not the case in HaRTES. The central element in HaRTES is not extremely reliable and because of all its complexity it is hardly possible to make it so. And if we look to the future, making the central element highly reliable in Ethernet-based FTT will become even more difficult. After all, several research groups are still working on adding more and more features to FTT, which will typically demand a more complex switch.

Because of the above two reasons, we can extrapolate the results by Barranco and Proenza and conclude that a duplicated FTTRS star is more reliable than a simplex HaRTES star, just like a ReCANcentrate duplicated star was deemed more reliable than a simplex CANcentrate star.

[7] The parts-count method is the simplest way to estimate the reliability of a system: we count the number of parts in the system, estimate the failure rate of each one of them, and define the failure rate of the whole system as the sum of these failure rates. It is pessimistic because with this approach we assume that the failure of any system element causes the whole system to fail, which is not necessarily the case.

[8] United States of America: Department of Defense. *Military Handbook: Reliability Prediction of Electronic Equipment: MIL-HDBK-217F*. Department of defense, 1991.

Finally, the results of Barranco and Proenza do not even consider potential sources of faults such as external impacts where duplicated stars are even more beneficial. Thus, their results do not show the full reliability potential of a duplicated star over a simplex one. If we would take such faults into consideration as well, the reliability benefits of FTTRS over HaRTES would be even bigger. Still, a complete quantitative evaluation should be carried out as future work to know the specific reliability increase that FTTRS can yield.

10.2 Future Work

The work presented in this dissertation can be extended in multiple directions. We could turn some of the non-requirements (Section 8.1.2) into requirements and modify FTTRS accordingly; we could change the preliminary design choices (Section 8.1.4) and redesign FTTRS; we could modify FTTRS to guarantee that it can handle a more inclusive fault model than the one we used in this work (Section 8.2); and we could also solve several of the limitations of the work presented in this dissertation (Section 8.8.1). Let me elaborate on each possibility and then conclude with a shortlist.

10.2.1 Turning Non-Requirements into Requirements

When we designed FTTRS we not only listed a set of requirements, but also a list of non-requirements (Section 8.1.2), i.e., a list of features that our communication subsystem would not need to have. This simplified the design. One source of future work could now be to turn some of these non-requirements into requirements and modify FTTRS accordingly. Let us go through each of the non-requirements:

- The first non-requirement, NR1 (page 151), was not having to be compatible with legacy nodes, i.e., non-real-time nodes that know nothing about FTT, such as ordinary laptops with stock Ethernet drivers. Turning NR1 into a requirement would mean requiring FTTRS to be compatible with legacy nodes. This is already possible as long as legacy nodes remain non-faulty (Section 9.6). When they are faulty, however, we need additional mechanisms to ensure that their failure does not lead to a disruption of the FTT service. That is, we need mechanisms to specifically contain the errors originating at legacy nodes. One approach might be to enhance the port guardians accordingly.
- The second non-requirement, NR2, was not having to ensure backward

compatibility with HaRTES. If we want to turn it into a requirement, we would have to add mechanisms for a HaRTES slave to be attached to an FTTRS network and still function. This should be possible by making an FTTRS switch behave like a HaRTES switch through particular ports to which HaRTES slaves could then be connected. For instance, an FTTRS switch should send only one trigger message per elementary cycle through such ports. Of course, the HaRTES slaves would then not benefit from the same fault tolerance as doubly-connected slaves with FTTRS drivers.

- The third non-requirement, NR3, was about message streams being schedulable in HaRTES not having to be schedulable in FTTRS. Turning it into a requirement would mean that every set of streams that are schedulable in HaRTES should also be schedulable in FTTRS. This is achievable, but we would then no longer be able to free up bandwidth for temporal message replicas by reducing the number of streams schedulable per elementary cycle in FTTRS when compared to HaRTES. Without the freed-up bandwidth, we would no longer be able to solely rely on temporal redundancy for tolerating transient faults. This would have deep implications for FTTRS. After all, it relies heavily on temporal redundancy in the form of proactive retransmissions. We would thus either have to add additional spatial redundancy to compensate for the lost temporal redundancy or we would have to sacrifice some of the fault tolerance of FTTRS. The latter might make sense if we compensate the lost tolerance to faults by focusing on other means to achieve reliability and dependability, e.g., fault prevention and fault removal (Section 2.4.3). For instance, the bit error rate on the links might be reduced by using fiber optic links or additional shielding.

- The fourth non-requirement, NR4, was not having to rely solely on commercial off-the-shelf components. Turning it into a requirement could be explored, but we would have to come up with an entirely new FTT-based communication subsystem. Neither FTTRS nor its predecessor HaRTES could be used as a starting point.

- The fifth non-requirement, NR5, was not having to tolerate slave failures. It was precisely what Derasevic addressed within the FT4FTT project. In fact, we have already seen in the experiments involving the final FT4FTT prototype (Section 9.6) that NR5 can be turned into a satisfiable requirement.

- The sixth non-requirement, NR6, was not needing to support FTT data messages with large payloads, i.e., payloads exceeding the amount of information that can be carried in a single Ethernet frame. Supporting large payloads should be fairly straightforward. It would involve fragmenting such messages and conveying each fragment in a different Ethernet frame within the same elementary cycle, with each such Ethernet frame being replicated both in space and time, i.e., being transmitted by a slave through both its links in parallel multiple times. Additionally, the scheduler used by FTTRS would have to be revised to take into account the additional Ethernet frames corresponding to the fragments of a large FTT message.

- The seventh and last non-requirement, NR7, was not having to support plug-and-play. Adding plug-and-play support could be explored. However, it might have important reliability implications. For instance, one of the mechanisms that a port guardian uses to restrict the failure semantics of a slave is to check messages for a correct MAC source address. But if an unknown slave could be attached to an FTTRS switch at runtime, the corresponding port guardian would have no way of knowing whether the slave is sending messages with a correct source address—although a port guardian could at least check that the slave is not impersonating another slave connected to some other port. In any case, it is clear that adding plug-and-play support would make FTTRS more flexible, but also that the reliability implications would have to be evaluated carefully.

10.2.2 Changing the Preliminary Design Choices

When we designed FTTRS we made some preliminary design choices (Section 8.1.4 on page 153). Thus, a second source of future work could be based on changing some of these choices:

- The first design choice was about requiring all links to have negligible differences in propagation and transmission time. If we want to change this, one implication would be that we would have to change the way in which FTTRS synchronizes. After all, the elementary cycle synchronization and system-wide commit relies on the fact that if a switch transmits a trigger message in parallel through all its links, then the trigger message replicas will reach the end of each link at approximately the same time. This, however, could no longer be guaranteed if there are significant differences between links. One alternative synchronization approach might be to use a clock synchronization protocol such as the precision time protocol (PTP), which uses specific mechanisms for compensating for the different delays in the communication links. We would then use time instants for synchronization instead of trigger message arrival times. For instance, slaves and masters could consider each elementary cycle to start whenever their synchronized local clocks have counted a time interval of one millisecond, where one millisecond would then be the duration of the elementary cycles. Trigger messages, however, would still have to be broadcast to convey the elementary cycle schedules and other information; it is just that their timing would be less important.

- The second design choice was about requiring switches to be store-and-forward switches. We could explore the alternative of FTTRS switches being cut-through switches instead. Nevertheless, I do not think that this would be promising. The main advantage of cut-through switches is that they can increase the performance of the network by having a switch forward a frame as soon as it knows the destination port. But this would be a significant problem for error containment: using cut-through switching would mean that an FTTRS switch would forward a frame before it has checked the correctness of the frame in its entirety. This would compromise the error containment capability of FTTRS and its ability to restrict the failure semantics of slaves.

- The last preliminary design choice was about not having applications running on top of FTT masters. We could change this and consider executing application-specific software on top of embedded masters. For instance, we could add sensors to the FTTRS switches and have masters detect environmental changes, which would then trigger changes in the SRDBs. If we then also prevent slaves from requesting changes, this would solve the problem of slaves sending bogus or babbling idiot update requests, while still allowing FTTRS to be flexible. The fail-silent masters would then be the only ones to trigger changes. Another potential benefit of application-specific software on top of masters is that it might allow us to further improve error containment. Specifically, the switch might then have information about what constitutes valid payloads in slave messages for a given application and the guardians could use that information to discard messages that otherwise could not be identified as carrying incorrect payloads.

10.2.3 Dealing with a More Inclusive Fault Model

Yet another source of future work could be a more inclusive fault model. We have designed FTTRS assuming non-malicious operational hardware faults (Section 8.2 on page 154), but no development faults, software faults, or malicious faults.[9] If we extend the fault model with new fault types, we would have to address these

[9] When referring to faults, malicious refers to human-made faults introduced with the objective to cause harm to the system or its environment. It does not refer to byzantine failures. See Section 2.5.1.

specifically. For instance, to address development software faults we could use N-version programming and to address malicious faults we would have to delve into the realm of security.

10.2.4 Addressing the Limitations of the Work Presented

Finally, another source of future work are the limitations of the work presented in this dissertation (Section 8.8.1, page 234). Let us go through the main limitations I consider worth addressing.

- Since addressing the scheduling issues was out of the scope, I did not evaluate how to modify the HaRTES scheduler to take into account the extra time consumed by proactive retransmissions and replica radiation. I assumed that the specific task of executing the admission control and of generating elementary cycle schedules would be performed correctly and consistently by the masters given that they are internally deterministic, synchronized in time, and have consistent SRDBs. Nevertheless, upgrading the HaRTES scheduler needs to be done for FTTRS to provide a correct service. Otherwise, the scheduler might misjudge how many FTT messages actually fit into an elementary cycle and this could cause a failure of the FTT service since meeting deadlines may then become impossible.[10]

- Since evaluating the performance of FTTRS was out of the scope of this dissertation, I did not evaluate such metrics as throughput or latency. Such metrics are in any case mostly irrelevant for hard real-time messages such as the trigger messages, synchronous messages, and asynchronous messages exchanged in FTTRS. This is so because for hard real-time messages it is usually irrelevant how soon messages reach their destination, as long as they reach it before the corresponding deadline. Nevertheless, we could evaluate the performance in future work. This might be particularly interesting if we add support for legacy applications that transmit non-real-time messages in the background, without interfering with the real-time messages. In that case we could evaluate the impact on performance of the proactive retransmissions and replica radiation. This should be a quantitative study, which might be carried out using simulation or analytical approaches.

- Because of time limitations, I did not complete a quantitative reliability analysis of FTTRS and the other Ethernet-based versions of FTT. We therefore

[10] In the prototypes we did not run into problems because the duration of the elementary cycles was oversized for the number of message streams used and their parameters.

do not have concrete numbers on how reliable each of the different Ethernet-based versions of FTT are. Nevertheless, we already have evidence that FTTRS, when properly parameterized for the target environment, can be more reliable than HaRTES (see Section 10.1.1).

- I did not design a process to reintegrate a faulty switch or a switch that has lost replica determinism. But this would be particularly interesting for cases where a switch suffered a transient fault, which currently has the same effect as a permanent fault due to the switches' internal duplication with comparison.
- I did not design a mechanism to allow FTTRS to tolerate the failure of all interlinks. Although this can be compensated by having enough redundant interlinks, it might still be interesting to explore other solutions. For instance, when all interlinks fail, and the network is partitioned into two networks controlled by uncoordinated masters, the slaves could try to determine which of the two networks has a majority of correctly functioning slaves attached and use that network in preference to the other.
- I did not make it a priority to minimize the time it takes the masters to process update requests. In future work we could explore ways to increase the reactivity of FTTRS when the real-time message parameters have to change. This might be particularly interesting for distributed embedded systems operating in dynamic environments that can change abruptly and might require the underlying communication subsystem to satisfy new requirements in a short amount of time. The current approach—which sufficed to prove the thesis—relies on a preestablished total order, which means that the order in which update requests are processed by the masters is preestablished. This precludes masters processing update requests using dynamic priorities based on environmental factors.
- In the current version of FTTRS, like in all other versions of FTT, there are no deadlines for the masters to process update requests. In future work we could change that and study how to provide real-time guarantees for the processing of update requests.
- Finally, we could study how to make the fault-tolerance mechanisms of FTTRS more dynamic. For instance, we could study how to increase or decrease the trigger message redundancy level at runtime, depending on the number of trigger messages that are actually being corrupted. In particular, allowing the trigger message redundancy level to be increased could enable an FTTRS-based application to survive in environments that turned out to

be harsher than anticipated during deployment. In fact, this task is already planned for the DFT4FTT project.

10.2.5 Future Work Shortlist

As we saw above, there are plenty of ways of extending the work. Nevertheless, the following are the issues that I, based on my experience researching dependable systems, consider most worthwhile to pursue next:

- *Quantify the reliability of FTTRS and its FTT competitors.* We designed FTTRS to be fault tolerant. This should make it more reliable than FTT Ethernet, FTT-SE, and HaRTES, all of which, contrary to FTTRS, have single points of failure and no fault-tolerance mechanisms. Indeed, we have already seen that FTTRS should be more reliable than HaRTES (Section 10.1.1). However, we do not have concrete numbers. We do not know how much more reliable FTTRS is. To find out a quantitative reliability evaluation is in order. Such a quantification will become particularly interesting once we have more mature prototypes (e.g., with hardware-implemented FTTRS switches), since then we will know better what exact components a deployable FTTRS system might have and thus what failure rates to assign to the different components in a quantitative reliability evaluation.

- *Prevent improper changes to the real-time parameters.* In FTTRS, and all other versions of FTT, any arbitrary slave can change the real-time communication parameters. If a slave is faulty, this can be a problem. A faulty slave could send bogus update requests and, if these pass the admission control of the masters, the system requirements would be updated incorrectly. For instance, a slave sending bogus requests could incorrectly delete existing message streams, create new ones, change the periodicity of an existing synchronous stream, modify deadlines, and so forth. Although this could not cause a failure of the communication, it could cause a failure of any application relying on the message streams to have sensible values. Earlier (Section 8.8.1) I pointed out that this could be solved by only allowing trusted slaves to request changes to the real-time parameters, by using voting, or by only allowing the fail-silent masters to request changes themselves. A careful study of which is the best approach and how exactly to implement it should be performed to address this current limitation of all FTT versions.

- *Upgrade the scheduler to take into account proactive retransmissions and replica radiation.* As I said before, without upgrading the scheduler of each embedded master, they might misjudge how many messages fit into an elementary cycle. If that happened, they would be unable to ensure that slaves can exchange messages without violating any deadlines (subservice S2 on page 148). This would result in a failure of the FTT service, which is why this limitation needs to be addressed.

- *Add further support for dynamic fault-tolerance.* One of the fault tolerance mechanisms of FTTRS can be parameterized to fit variously harsh environments. Specifically, the redundancy level of all messages can be adjusted. For slave messages this can even be done at runtime (for trigger messages, in contrast, the redundancy level can only be adjusted offline). This gives FTTRS not only flexibility towards changing real-time communication requirements, but also some flexibility towards changing reliability requirements. With further efforts FTTRS could be enhanced to provide full dynamic fault tolerance, i.e., mechanisms to automatically shift resources to increase the reliability of some aspects of the communication, while sacrificing other aspects such as performance. For instance, the communication subsystem could be improved to continuously monitor the bit error rate on the links and automatically increase or decrease the trigger message replication level accordingly, with a corresponding decrease or increase in the bandwidth available for other messages. In fact, within my research group we found adding dynamic fault tolerance so interesting and relevant that we are planning to address it in the DFT4FTT project, the successor to the FT4FTT project.

Finally, since I motivated this dissertation by the upcoming need for highly reliable real-time adaptive distributed embedded systems, it would be exciting to actually build a prototype of such a system using FTTRS. We have shown that FTTRS is flexible, but FTTRS on its own is not adaptive. Thus, to build an adaptive distributed embedded system we would have to put some intelligence on top of FTTRS that makes use of the flexibility of FTTRS. This intelligence might be application-specific software running on top of FTTRS that monitors the environment and responds to environmental changes by modifying itself and by sending appropriate update requests down to the FTTRS subsystem.

Part III

Appendices

Appendix A

A More General Formalization of a Correct Service

When I introduced the yellow brick road (or tunnel) in Section 2.3 on page 26, I did so under the assumption that both the time dimension and the external states are continuous and totally ordered, and that correct external states are contiguous. This allowed us to easily visualize a system's function as a road and its service as a curve that may either be on that road (when the service is correct) or off that road (when the service is incorrect). In this appendix we will see that it is not necessary for the external states and time to be continuous; nor is it necessary for the external states to be totally ordered and for correct external states to be contiguous; we can still formalize what it means to deliver a correct service.

As in Section 2.3, let us begin by considering a two-dimensional service space. Let X again be the set of external states that the system can be in and let T be a totally ordered set that represents the mission time. As we know, the service delivered by the system can be considered as a function $x\colon T \to X$ such that $x(t) = a$ if the system is in the external state a at time t (for now we are assuming only one user so there is only one family of external states).

The two-dimensional service space is the Cartesian product $T \times X$. Let us denote it by $\mathbb{S}$. The system's function is a region of the service space, i.e., a set of points $F \subseteq \mathbb{S}$, which spans the whole mission time, i.e., for every $t \in T$ there exists at least one point $(t, \cdot\ ) \in F$. The region F is thus what we previously (in Section 2.3) called the yellow brick road. Now, however, the region does not need to be connected. Still, the idea is the same: the system delivers a correct service at time t if the point $(t, x(t)) \in F$. Otherwise, the system delivers an incorrect service and is said to be faulty.

Let $x\restriction_{[t_0,t_1]}$ denote the restriction[1] of the service to the interval of time $[t_0, t_1] \in T$. In other words, $x\restriction_{[t_0,t_1]}$ is the set of points $\{(t, x(t)) \mid t \in [t_0, t_1]\}$, or, equivalently, $x\restriction_{[t_0,t_1]}$ is the same mathematical function as x, but is only defined on the interval of time $[t_0, t_1]$ instead of being defined for the whole mission time.[2] We say that the interval of time $[t_0, t_1]$ is a **service delivery** if $x\restriction_{[t_0,t_1]} \subseteq F$. On the other hand, $[t_0, t_1]$ is a **service outage** if $x\restriction_{[t_0,t_1]}$ and F are disjoint, that is, $x\restriction_{[t_0,t_1]} \cap F = \emptyset$.

If time is discrete, then a failure occurs at time t_k if $(t_{k-1}, x(t_{k-1})) \in F$ and $(t_k, x(t_k)) \notin F$, that is, if $x\restriction_{\{t_{k-1}\}} \in F$ and $x\restriction_{\{t_k\}} \notin F$. If we consider time to be continuous, then a failure occurs at time t if $(t - \Delta t, x(t - \Delta t)) \in F$ and $(t + \Delta t, x(t + \Delta t)) \notin F$ for $\Delta t \to 0$.

Similarly, if time is discrete, then a service restoration occurs at time t_k if $(t_{k-1}, x(t_{k-1})) \notin F$ and $(t_k, x(t_k)) \in F$. If we consider time to be continuous, then a service restoration occurs at time t if $(t - \Delta t, x(t - \Delta t)) \notin F$ and $(t + \Delta t, x(t + \Delta t)) \in F$ for $\Delta t \to 0$.

When we visualized the function as a road (Section 2.3), we were assuming that F is a region bounded by two curves. That is, $F = \{(t, x(t)) \mid t \in T, f_1(t) \leq x(t) \leq f_2(t)\}$, where $f_1(t)$ is the lower bound and $f_2(t)$ is the upper bound. However, this visualization only makes sense if the set of external states X is a totally ordered set—otherwise there may be points in the service space that are not comparable by a "$\leq$" relation and writing "$f_1(t) \leq x(t) \leq f_2(t)$" may not make any sense. One example where X is totally ordered is when the elements of X, the external states, can be represented by real numbers, as is the case for temperature values or oxygen levels. The external states, however, do not have to constitute a totally ordered set. For instance, if X is the set of possible messages that can be transmitted by a system through a network, then we usually do not have a total order for these messages. The above formalization does not require the external states to be totally ordered.

We can now extend our formalization to a service space of more than two dimensions. Assume that the system's function requires the delivery of n services. Each service corresponds to a function $x_i \colon T \to X_i$, where $i \in [1, n]$. Each set X_i is a set of external states corresponding to a given service. For instance, X_1 may be a range of temperatures and X_2 may be a range of oxygen levels. The service space corresponding to this system has $n + 1$ dimensions and is given by the Cartesian product $T \times X_1 \times \cdots \times X_n$. The system's function is now a region of this multidimensional service space, i.e., a set of points $F \subseteq T \times X_1 \times \cdots \times X_n$. Moreover, we can introduce the notion of a **composite service** as a function $\mathbf{x} \colon T \to X_1 \times \cdots \times X_n$

[1] The restriction of a function f with a domain D is another function $f\restriction_{D'}$ that is identical to f, except that it is only defined for a subdomain $D' \subseteq D$.

[2] Remember that a function $f \colon A \to B$ is a set of ordered pairs $\{(a, f(a)) \mid a \in A, f(a) \in B\}$.

such that $\mathbf{x}(t) = (x^{(1)}(t), \ldots, x^{(n)}(t))$. As a shorthand, let us write $\mathbf{x}{\restriction_t}$ for $(t, x^{(1)}(t), \ldots, x^{(n)}(t))$. The system then delivers a correct service at time t if the point $\mathbf{x}{\restriction_t} \in F$. Otherwise, it delivers an incorrect service and is said to be faulty.

Service delivery, service outage, service failure, and service restoration are defined analogously to the two-dimensional case by replacing two-dimensional points of the form $(t, x^{(1)}(t))$ with points of $n+1$ dimensions of the form $\mathbf{x}{\restriction_t} = (t, x^{(1)}(t), \ldots, x^{(n)}(t))$. Hence, the time interval $[t_0, t_1]$ is a service delivery if $\mathbf{x}{\restriction_{[t_0,t_1]}} \subseteq F$. In contrast, it is a service outage if $\mathbf{x}{\restriction_{[t_0,t_1]}} \cap F = \emptyset$. If we consider time to be discrete, then a failure occurs at time t_k if $\mathbf{x}{\restriction_{t_{k-1}}} \in F$ and $\mathbf{x}{\restriction_{t_k}} \notin F$. If we consider time to be continuous, then a failure occurs at time t if $\mathbf{x}{\restriction_{t-\Delta t}} \in F$ and $\mathbf{x}{\restriction_{t+\Delta t}} \notin F$ for $\Delta t \to 0$. A service restoration with discrete time occurs at time t_k if $\mathbf{x}{\restriction_{t_{k-1}}} \notin F$ and $\mathbf{x}{\restriction_{t_{k-1}}} \in F$. A service restoration with continuous-time occurs at time t if $\mathbf{x}{\restriction_{t-\Delta t}} \notin F$ and $\mathbf{x}{\restriction_{t+\Delta t}} \in F$ for $\Delta t \to 0$.

Appendix B

A Pseudocode Description of FTTRS

This appendix contains a pseudocode description of the flexible time-triggered replicated star (FTTRS). I wrote it as a reference and as a more detailed description of FTTRS. In particular, it shows how the different fault-tolerance mechanisms of FTTRS can be put together.

In the pseudocode, when I had to make a trade-off between readability and performance, I favored readability. Moreover, to keep the pseudocode simple, and since our fault model excludes software faults, I assume that requests are correctly invoked by the application layer.

The notation I use is based on the one that Cachin, Guerraoui, and Rodrigues use in their book *Introduction to Reliable and Secure Distributed Programming*.[1]

The pseudocode is structured as a set of concurrent modules, also called components, that interact with each other by means of events. In an actual implementation, these modules might be processes running under an operating system or hardware components.

There are three types of events: requests, indications, and internal events.

Requests "are used by a component to *invoke* a service at another component or to *signal* a condition to another component".[2]

Indications "are used by a component to *deliver* information or to *signal* a

[1] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.

[2] Ibid., p. 11.

condition to another component”.[3]

Internal events originate internally within a module. They “are triggered when some condition in the implementation becomes true”.[4] An example of an internal event is a Boolean expression becoming true. Other examples are an Init event, which is true when the system is initialized, and timeout events, which are true when a corresponding timer expires.

Events can have attributes, such as the module invoking the event or data structures. They are denoted as follows:

$\langle co, \mathrm{EventType} \mid \mathit{Attributes}, \ldots \rangle$

Here co denotes an instance of a module, or component, for which the event is defined. If we want another module to invoke the above event, we write a trigger statement within the pseudocode of the invoking module:

```
trigger$\langle\ co, \mathrm{EventType} \mid \mathit{Attributes}, \ldots\ \rangle$;
```

The module whose event is triggered must have a corresponding event handler, which we denote as follows:

```
upon event $\langle\ co, \mathrm{EventType} \mid \mathit{Attributes}, \ldots\ \rangle$ do
    Code that handles the event;
```

Event handlers for internal events are denoted as follows when they are triggered by a Boolean expression becoming true:

```
upon $\mathit{condition}$ do
    Handle internal event;
```

Above $\mathit{condition}$ denotes a Boolean expression.

Event handlers for internal events that model the expiration of a local timer are denoted like this:

```
upon event $\langle\ co, \mathrm{TimeoutX}\ \rangle$ do
    Handle timeout;
```

The timer is started by invoking a corresponding function $\mathrm{startTimerX}$, whose definition we assume and whose argument is a non-negative real number that represents the time that shall elapse until the timeout:

[3] Cachin, Guerraoui, and Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, p. 11.

[4] Ibid., p. 10.

$\text{startTimerX}(\textit{timeUntilTimeout});$

I assume that the execution of event handlers takes a negligible amount of time when compared with the time to transmit frames. This is in line with Lamport's definition of a distributed system, which he defines as a "multiprocessor system in which the time required for interprocess communication is large compared to the time between events in a single process [i.e., module]".[5]

Further notes on the notation follow:

- Functions or procedures that I will *not* define, and whose existence I simply assume, are typeset like this: undefinedFunction.
- Functions or procedures that I *will* define are typeset like this: DEFINED-FUNCTION.
- Variable names are typeset like this: $\textit{variable}$.
- Events are typeset like this: Event.
- An empty value is denoted like this: NIL.
- Assignment is denoted with a left pointing arrow. Thus, assigning the value of a variable x to a variable y is written like this: $y \leftarrow x$
- The notation $[x]^N$ for any symbol x denotes a vector $[x, \ldots, x]$ of N positions, indexed by integers or other enumerable types such as message stream identifiers.

B.1 FTTRS Modules

Figure B.1 shows the modules that make up the FTTRS reference implementation and how they interact by means of requests and indications. Requests are drawn as solid arrows and indications as dashed arrows. Dotted arrows represent a **requires** relationship between a module and a data structure, meaning that the pointed-to module requires the data structure and can directly access it. The only such data structures in the figure are NRDBs and SRDBs. The figure only shows the modules for one slave and one switch. The modules of the second switch (Switch B) are the same as those of the first (Switch A).

[5] Software Engineering Radio. *Episode 203: Leslie Lamport on Distributed Systems.* IEEE Computer Society. Apr. 2014.

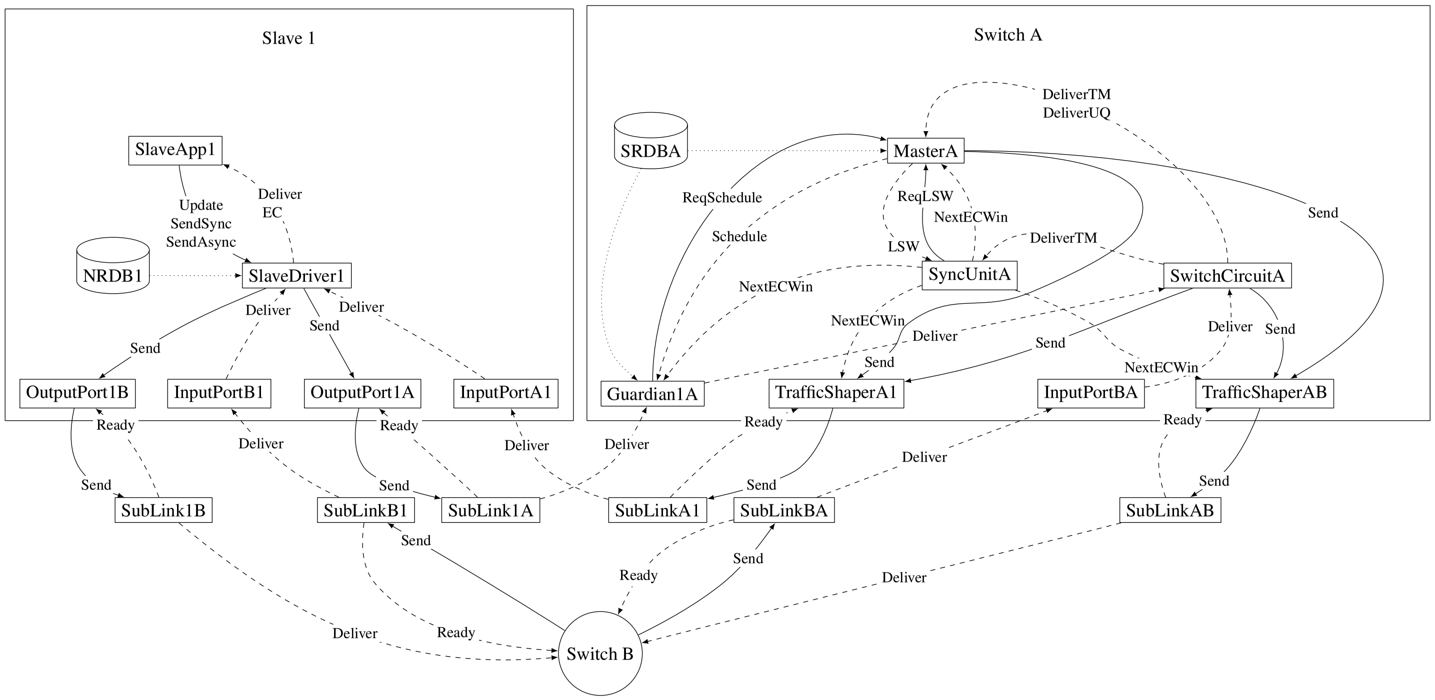


Figure B.1: Abstract modules that make up FTTRS slaves, switches, and links. Solid arrows represent requests, dashed arrows represent indications, and dotted arrows represent a *requires* relationship.

The modules that constitute FTTRS can be classified into three groups: those corresponding to an FTTRS slave, those corresponding to an FTTRS switch, and those corresponding to links.

Within a slave we find the following:

- A SlaveApp module, which models the application executed by an FTTRS slave. Since this is application dependent, I assume its existence, but will not define it.
- A SlaveDriver module, which models the software driver that manages the communication through an FTTRS network.
- An OutputPort module, which models that part of an Ethernet network interface card responsible of sending Ethernet frames through a link.
- An InputPort module, which models that part of an Ethernet network interface card responsible of receiving Ethernet frames from a link.

In addition to these modules, an FTTRS slave also includes an NRDB, to which the corresponding SlaveDriver has direct access.

A SlaveApp may request changes to the NRDB by sending an Update request to the SlaveDriver. Moreover, a SlaveApp may request the transmission of synchronous or asynchronous data by sending a $\mathrm{SendSync}$ or $\mathrm{SendAsync}$ to the SlaveDriver, respectively.

A SlaveDriver, in turn, delivers data to the SlaveApp using a $\mathrm{Deliver}$ indication. It also signals the start of a new elementary cycle using an EC indication. The interface of a SlaveDriver is listed in Module 5 on page 316 and its implementation is listed in Algorithm 4 on page 317.

A SlaveDriver, in addition to interacting with a SlaveApp, also interacts with two OutputPort modules and two InputPort modules. Each OutputPort is paired with an InputPort, together constituting the interface to one of the two slave links, which in turn are modeled by two SubLink modules to represent an uplink and a downlink.

A SlaveDriver can send a Send request to an OutputPort and may receive a $\mathrm{Deliver}$ indication from an InputPort. The former models the software driver requesting the transmission of an Ethernet frame through the network interface card and the latter the reception of an Ethernet frame. Module 3 and Algorithm 2 on page 313 show the interface and algorithm of an OutputPort, respectively; similarly, Module 2 and Algorithm 1 on page 312 and on page 313 show the interface and algorithm of an InputPort.

An OutputPort, upon receiving a Send request, will itself send a Send request to a SubLink. After some time, corresponding to the transmission and propagation time, the SubLink generates a Deliver indication to an InputPort or Guardian. Moreover, it generates a Ready indication upwards, to the module that previously invoked the Send. This tells the invoking module that the SubLink is ready to transmit another Ethernet frame. Module 1 on page 312 shows the interface of a SubLink. Since SubLink modules are simple hardware components—i.e., cables—I describe their implementation by means of a set of properties instead of by an algorithm.

A SubLink corresponding to an uplink interacts with a Guardian. Specifically, a SubLink can send a Deliver indication to the Guardian. This models the delivery of an Ethernet frame from the uplink to the corresponding port guardian.

A Guardian has direct access to the SRDB of its switch and interacts with several other modules within the switch. Specifically, it may send a ReqSchedule request to an embedded master Master, which in turn will respond with a Schedule indication. This allows a guardian to obtain the schedule of a given elementary cycle, which it needs to determine whether a frame that was delivered to it from a SubLink is timely or not. A Guardian also needs to know within which window of the elementary cycle the system currently finds itself. For this, it receives NextECWin indications from a SyncUnit module. This module keeps track of the system synchronization and tells other modules within the same switch when the next window of an elementary cycle starts (similar to a HaRTES dispatcher, shown within the shaded region of Figure 7.9 on page 139 and described in a little more detail in a paper by Santos et al.[6])). Finally, if a Guardian determines that a frame that was delivered to it is to be forwarded, it does so by delivering the frame to a SwitchCircuit module using a Deliver indication. Module 6 and Algorithm 5 on page 319 and on page 320 show the interface and algorithm corresponding to a Guardian module, respectively.

A SwitchCircuit module represents the switching logic within an FTTRS switch. It receives Ethernet frames from Guardian modules or InputPort modules by means of Deliver indications. If the delivered frame is a trigger message, then it is delivered to the SyncUnit and the Master module using a DeliverTM indication. The SyncUnit uses the arrival time of the trigger message to achieve synchronization with the SyncUnit in the other switch. The Master module, on the other hand, uses the trigger messages it receives to extract the update requests that may be piggybacked on the trigger messages. It then uses these to ensure replica determinism in the value domain with the Master module in the other switch. A SwitchCircuit also forwards to the Master any update requests coming from slaves in standalone Ethernet frames. For this, it uses the DeliverUQ indication. The interface and

[6] Santos et al., "Designing a Costumized [sic] Ethernet Switch for Safe Hard Real-Time Communication", Sec. 4. FTT-enabled Ethernet Switch Architecture.

algorithm that implement a SwitchCircuit are shown in Module 7 and Algorithm 6 on page 323 and on page 324, respectively.

A Master and a SyncUnit also interact with each other. Specifically, a SyncUnit tells a Master unit when a new elementary cycle window starts using a $\mathrm{NextECWin}$ indication. Furthermore, a SyncUnit periodically requests the length of the next synchronous window from the master. For this, it uses a ReqLSW request and receives, in turn, a LSW indication.

Module 9 and Algorithm 8 on page 328 and on page 329 show the interface and algorithm that implement a Master module. Similarly, Module 8 and Algorithm 7 on page 325 and on page 326 show the interface and algorithm that implement a SyncUnit.

Finally, we have TrafficShaper modules. These have essentially the same interface as OutputPort modules, except that they receive $\mathrm{NextECWin}$ indications from a SyncUnit to transmit frames such that they are confined within the different windows of an elementary cycle. The interface and algorithm of a TrafficShaper are shown in Module 4 and Algorithm 3 on page 314 and on page 315. It implements much of the same functionality as a port dispatcher in HaRTES (see Figure 7.9 on page 139)

Module 1 Interface and properties of a unidirectional sublink with omission failure semantics.

Name: SubLink **as** sl.
Requires: bit latency lat and frame-loss bound k.

Events:
Request: $\langle\ sl, \text{Send} \mid f\ \rangle$: Requests to send a frame f.
Indication: $\langle\ sl, \text{Deliver} \mid f\ \rangle$: Delivers the frame f.
Indication: $\langle\ sl, \text{Ready}\ \rangle$: Indicates that the sublink sl is ready to transmit another frame.

Properties:

SL1: *No duplication:* if a frame f is send once, then f cannot be delivered more than once.

SL2: *No creation:* if a frame f is delivered, then f was previously sent.

SL3: *k-bounded loss:* if a frame of maximum size is sent k times, and sl has not crashed, then the frame is delivered at least once.

SL4: *Eventual crash:* the sublink sl is going to crash eventually.

SL5: *Constant per bit latency:* the time it takes to transmit and propagate a bit (called latency) has a constant value of lat.

Module 2 Interface of an input port.

Name: InputPort **as** ip.

Events:
Indication: $\langle\ ip, \text{Deliver} \mid f\ \rangle$: Delivers a frame f.

Algorithm 1 Implementation of an input port. It simply forwards every frame it receives from a sublink.

```
Implements: InputPort as $ip$.
Uses: a SubLink $sl$.

upon event $\langle\ sl, \mathrm{Deliver} \mid f\ \rangle$ do
    trigger$\langle\ ip, \mathrm{Deliver} \mid f\ \rangle$;
```

Module 3 Interface of a FIFO output port.

```
Name: OutputPort as $op$.

Events:
    Request: $\langle\ op, \mathrm{Send} \mid f\ \rangle$: Requests to send a frame $f$.
```

Algorithm 2 Implementation of a FIFO output port. It sends an Ethernet frames one by one in FIFO order through a sublink.

```
Implements: OutputPort as $op$.
Uses: a SubLink $sl$.

upon event $\langle\ op, \mathrm{Init}\ \rangle$ do
    $pendingFrames \leftarrow$ empty FIFO queue;
    $linkIsBusy \leftarrow$ FALSE;

upon event $\langle\ op, \mathrm{Send} \mid f\ \rangle$ do
    $\mathrm{append}(pendingFrames, f)$;

upon $\neg\mathrm{linkIsBusy} \wedge \neg\mathrm{isEmpty}(pendingFrames)$ do
    $f \leftarrow \mathrm{popFirst}(pendingFrames)$;
    TRANSMITFRAME($f$);

upon event $\langle\ sl, \mathrm{Ready}\ \rangle$ do
    $linkIsBusy \leftarrow$ FALSE;

procedure TRANSMITFRAME($f$)
    $linkIsBusy \leftarrow$ TRUE;
    $fcs \leftarrow$ frame check sequence of $f$;
    append $fcs$ to $f$;
    trigger$\langle\ sl, \mathrm{Send} \mid f\ \rangle$;
```

Module 4 Interface of a traffic shaper.

```
Name: TrafficShaper as $ts$.

Events:
    Request: $\langle\ ts, \mathrm{Send} \mid f\ \rangle$: Requests to send $f$.
```

Algorithm 3 Implementation of a traffic shaper. It shapes traffic on downlinks to comply with the FTT windows.

```
Implements: TrafficShaper as $ts$.
Uses: a SubLink $dl$ and a SyncUnit $u$.

upon event $\langle\ ts, \text{Init}\ \rangle$ do
    $framesSync \leftarrow \emptyset$;
    $framesAsync \leftarrow \emptyset$;
    $linkIsBusy \leftarrow$ FALSE;
    $phase \leftarrow$ NIL;

upon event $\langle\ u, \text{NextECWin} \mid newWindow\ \rangle$ do
    $phase \leftarrow newWindow$;
    ▷ $newWindow$ is TMWIN, TAT, SYNCWIN, ASYNCWIN, or GUARD.

upon event $\langle\ ts, \text{Send} \mid f\ \rangle$ do
    if $f$ contains a trigger message from our master $\wedge \neg linkIsBusy \wedge phase =$ TMWIN then
        TRANSMITFRAME($f$);
    else if $f$ contains a synchronous message then
        $framesSync \leftarrow framesSync \cup \{f\}$;
    else if $f$ contains an asynchronous message then
        $framesAsync \leftarrow framesAsync \cup \{f\}$;
    else
        $\text{shutdown}()$;                         ▷ Something went wrong. Force crash.

upon $(phase =$ TAT $\vee\ phase =$ SYNCWIN$) \wedge \neg linkIsBusy \wedge \neg\text{isEmpty}(framesSync)$ do
    $f \leftarrow \text{selectFrame}(framesSync)$;
    TRANSMITFRAME($f$);
    $framesSync \leftarrow framesSync \setminus \{f\}$;

upon $phase =$ ASYNCWIN $\wedge \neg linkIsBusy \wedge \neg\text{isEmpty}(framesAsync)$ do
    $f \leftarrow \text{selectFrame}(framesAsync)$;
    TRANSMITFRAME($f$);
    $framesAsync \leftarrow framesAsync \setminus \{f\}$;

upon $phase =$ GUARD do
    ▷ Do nothing. This prevents starting transmissions that can block trigger messages.

upon event $\langle\ dl, \text{Ready}\ \rangle$ do
    $linkIsBusy \leftarrow$ FALSE;

procedure TRANSMITFRAME($f$)
    $linkIsBusy \leftarrow$ TRUE;
    trigger$\langle\ dl, \text{Send} \mid f\ \rangle$;
```

Module 5 Interface of the driver for FTTRS slaves.

Name: SlaveDriver **as** sd.

Requires: a node requirements database $NRDB$, a slave MAC address MAC, and a slave identifier $senderID$.

Events:

Request: $\langle\ sd, \mathrm{SendSync} \mid data, streamID\ \rangle$: Requests to send data through the synchronous stream identified by $streamID$.

Request: $\langle\ sd, \mathrm{SendAsync} \mid data, streamID\ \rangle$: Requests to send data through the asynchronous stream identified by $streamID$.

Request: $\langle\ sd, \mathrm{Update} \mid parameters, streamID\ \rangle$: Requests to update $streamID$ according to $parameters$.

Indication: $\langle\ sd, \mathrm{Deliver} \mid data, streamID\ \rangle$: Delivers data corresponding to the stream identified by $streamID$.

Indication: $\langle\ sd, \mathrm{EC} \mid c\ \rangle$: Notifies the upper layer that EC number c has started. The notification occurs at the end of the trigger message window.

Algorithm 4 Implementation of the driver for FTTRS slaves.

```
Implements: SlaveDriver as $sd$.
Requires: a node requirements database $NRDB$, a slave MAC address $MAC$,
and a slave identifier $senderID$.
Uses: A set $OP$ of OutputPorts and a set $IP$ of InputPorts.

upon event $\langle\ sd, \mathrm{Init}\ \rangle$ do
    $syncMessages \leftarrow \emptyset$;
    $receivedCmds \leftarrow \emptyset$;
    $schedule \leftarrow$ NIL;
    $hasReceivedTM \leftarrow$ FALSE;

upon event $\langle\ sd, \mathrm{SendSync} \mid data, streamID\ \rangle$ do
    $sm \leftarrow$ CREATEMESSAGE$(data, streamID)$;
    $syncMessages \leftarrow syncMessages \cup \{sm\}$;

upon event $\langle\ sd, \mathrm{SendAsync} \mid data, streamID\ \rangle$ do
    $am \leftarrow$ CREATEMESSAGE$(data, streamID)$;
    DISPATCHMESSAGES$(\{am\})$;

upon event $\langle\ sd, \mathrm{Update} \mid parameters, streamID\ \rangle$ do
    $am \leftarrow$ CREATEMESSAGE$(parameters, streamID)$;
    DISPATCHMESSAGES$(\{am\})$;

upon $\langle ip, \mathrm{Deliver} \mid f \rangle$, where $ip \in IP$ do
    msg $\leftarrow$ getPayload$(f)$;
    if msg is a trigger message $\wedge \neg hasReceivedTM$ then
        SYNCHRONIZEEC(msg);
        $schedule \leftarrow$ getSchedule(msg);
        $receivedCmds \leftarrow$ getCommands(msg);
        $hasReceivedTM \leftarrow$ TRUE;
    else                    ▷ msg is a synchronous or asynchronous data message.
        DELIVERMESSAGE(msg);                    ▷ Upper layer must deduplicate.

function CREATEMESSAGE$(data, streamID)$
    $c \leftarrow$ computeCRC$(data)$.
    return FTT message with sender $senderID$, payload $data$, CRC $c$, and
stream identifier $streamID$.
```

Algorithm 4 (Continued) Implementation of the driver for FTTRS slaves.

```
procedure SynchronizeEC(tm)
    $k, \tau, i \leftarrow \mathrm{getTimingParameters}(\mathrm{tm})$;
    $\mathit{timeUntilPolling} \leftarrow (k - i)\tau$;
    $\mathrm{startTimerPolling}(\mathit{timeUntilPolling})$;

upon event $\langle\ sd, \mathrm{TimeoutPolling}\ \rangle$ do
    trigger$\langle\ sd, \mathrm{EC} \mid c\ \rangle$;
    $\mathrm{commit}(\mathit{NRDB}, \mathit{receivedCmds})$;                    ▷ Update $\mathit{NRDB}$.
    SendPolledMessages();
    $\mathit{hasReceivedTM} \leftarrow$ FALSE;

procedure SendPolledMessages()
    $\mathit{polled} \leftarrow$ set of messages in $\mathit{syncMessages}$ that are scheduled according to $\mathit{schedule}$;
    DispatchMessages$(\mathit{polled})$;
    $\mathit{syncMessages} \leftarrow \mathit{syncMessages} \setminus \mathit{polled}$;

procedure DispatchMessages$(\mathit{messages})$
    for all $\mathrm{msg} \in \mathit{messages}$ do
        CreateFrame$(\mathrm{msg})$;
        $\mathit{streamID} \leftarrow$ stream identifier of $\mathrm{msg}$;
        $\kappa \leftarrow$ redundancy level of $\mathit{streamID}$ according to $\mathit{NRDB}$;
        for all $j \in [1, \kappa]$ do
            for all $op \in \mathit{OP}$ do trigger$\langle\ op, \mathrm{Send} \mid f\ \rangle$;

function CreateFrame$(\mathrm{msg})$
    $\mathit{dst} \leftarrow$ subscribers of $\mathrm{msg}$ according to $\mathit{NRDB}$;
    return frame with payload $\mathrm{msg}$, destination $\mathit{dst}$, source $\mathit{MAC}$, and an Ethertype corresponding to FTT.

procedure DeliverMessage$(\mathrm{msg})$
    $\mathit{data} \leftarrow$ payload of $\mathrm{msg}$;
    $\mathit{streamID} \leftarrow$ stream identifier of $\mathrm{msg}$;
    trigger$\langle\ sd, \mathrm{Deliver} \mid \mathit{data}, \mathit{streamID}\ \rangle$;
```

Module 6 Interface of a port guardian.

Name: PortGuardian **as** g.

Requires: slave MAC address $validMAC$, slave ID $validID$, and a systems requirements database $SRDB$.

Events:

Indication: $\langle\ g, \text{Deliver} \mid f\ \rangle$: Delivers the frame f.

Algorithm 5 Implementation of a port guardian.

```
**Implements:** PortGuardian **as** $g$.
**Requires:** slave MAC address $validMAC$, slave ID $validID$, and a systems requirements database $SRDB$.
**Uses:** an uplink $ul$, which is an instance of SubLink, a Master $m$, and a SyncUnit $u$.

upon event $\langle\ g, \text{Init}\ \rangle$ do
    $lastECseen \leftarrow [\text{NIL}]^{|\text{AM}|}$;    ▷ Tracks in which EC a given asynchronous message was seen for the last time.
    $replicaCount \leftarrow [0]^{|M|}$;    ▷ Tracks how many replicas of a given message we have seen in the current EC.
    $schedule \leftarrow [\text{NIL}]^2$;
    $phase \leftarrow \text{NIL}$;
    $currentEC \leftarrow 0$;
    $nextEC \leftarrow currentEC + 1$;
    trigger$\langle\ m, \text{ReqSchedule} \mid nextEC\ \rangle$;

upon event $\langle\ m, \text{Schedule} \mid newSchedule, c\ \rangle$ do
    $schedule[c \bmod 2] \leftarrow newSchedule$;

upon event $\langle\ u, \text{NextECWin} \mid newWindow\ \rangle$ do
    $phase \leftarrow newWindow$;
    ▷ $newWindow$ is TMWIN, TAT, SYNCWIN, ASYNCWIN, or GUARD.

upon $phase = \text{TMWIN}$ do
    $replicaCount \leftarrow [0]^{|M|}$;
    $currentEC \leftarrow currentEC + 1$;
    $nextEC \leftarrow currentEC + 1$;
    trigger$\langle\ m, \text{ReqSchedule} \mid nextEC\ \rangle$;

upon event $\langle\ ul, \text{Deliver} \mid f\ \rangle$ do
    if $\neg$ISVALIDFRAME$(f, validMAC)$ then
        return    ▷ Do not deliver the frame.
    if $\neg$containsFTTmessage$(f)$ then
        return    ▷ Do not deliver the frame.
    msg $\leftarrow$ payload of $f$;
    $streamID \leftarrow$ identifier of msg;
    $replicaCount[streamID] \leftarrow replicaCount[streamID] + 1$;
    $isValidMessage \leftarrow$ (isAsyncMessage(msg) $\vee$ isSyncMessage(msg)) $\wedge$ HASVALIDMETADATA(msg) $\wedge$ $\neg$ISIMPERSONATIONM(msg, $validID$) $\wedge$ HASVALIDCRC(msg) $\wedge$ ISTIMELY$(streamID, lastECseen, schedule, phase)$ $\wedge$ $\neg$HASTOOMANYREPLICAS$(streamID, replicaCount[streamID])$;
    if $isValidMessage$ then
        trigger$\langle\ g, \text{Deliver} \mid f\ \rangle$;
    $lastECseen[streamID] \leftarrow currentEC$;
```

Algorithm 5 (Continued) Implementation of a port guardian.

```
function ISVALIDFRAME($f$, $validMAC$)
    return hasValidFCS($f$) $\land \neg$ISIMPERSONATIONF($f$, $validMAC$);

function ISIMPERSONATIONF($f$, $validMAC$)
    $src \leftarrow$ source address of $f$;
    if $src \neq validMAC$ then
        return TRUE;
    else
        return FALSE;

function ISIMPERSONATIONM(msg, $validID$)
    $senderID \leftarrow$ sender ID of msg;
    if $senderID \neq validID$ then
        return TRUE;
    else
        return FALSE;

function HASVALIDMETADATA(msg)
    if metadata in msg is correct according to $SRDB$ then ▷ For instance, if the subscribers field and message length is correct.
        return TRUE;
    else
        return FALSE;

function ISTIMELY($streamID$, $lastECseen$, $schedule$, $phase$)
    if $streamID$ is an asynchronous stream then
        return ISTIMELYASYNCMSG($streamID$, $lastECseen$);
    else if $streamID$ is a synchronous stream then
        return ISTIMELYSYNCMSG($streamID$, $schedule$, $phase$);
    else
        return FALSE;
```

Algorithm 5 (Continued) Implementation of a port guardian.

```
function isTimelyAsyncMsg($streamID$, $lastECseen$)
    if $lastECseen[streamID] =$ NIL then    ▷ If first time ever we see $streamID$.
        return TRUE;
    $elapsedECs \leftarrow currentEC - lastECseen[streamID]$;
    $isTooSoon \leftarrow (elapsedECs < \text{getMinIntArrivTime}(SRDB, streamID))$;
    $isReplica \leftarrow (elapsedECs = 0)$;
    return $isReplica \lor \neg isTooSoon$;

function isTimelySyncMsg($streamID$, $schedule$, $phase$)
    $isScheduled \leftarrow (streamID \in schedule)$;
    return $isScheduled \land (phase =$ TAT $\lor$ $phase =$ syncWin$)$;

function hasTooManyReplicas($streamID$, $count$)
    $validReplicaCount \leftarrow$ replication level for $streamID$ according to $SRDB$;
    return $count > validReplicaCount$;

function hasValidCRC(msg)
    $receivedCRC \leftarrow$ CRC conveyed in msg;
    $payload \leftarrow$ payload of msg;
    $calculatedCRC \leftarrow \text{computeCRC}(payload)$;
    return $receivedCRC = calculatedCRC$;
```

Module 7 Interface of a switch circuit.

Name: SwitchCircuit **as** w.

Requires: a systems requirements database $SRDB$ and a forwarding table $shaperToSlave$ that tells us what slave is attached at each traffic shaper of the switch.

Events:

Indication: $\langle\ w, \mathrm{DeliverTM} \mid \mathrm{msg}\ \rangle$: Delivers a trigger message.

Indication: $\langle\ w, \mathrm{DeliverUQ} \mid \mathrm{msg}\ \rangle$: Delivers an update request.

Algorithm 6 Implementation of a switch circuit. It forwards Ethernet frames according to the stream identifier of FTT messages, doing the forwarding to those switch output ports that have subscribers of that stream attached.

```
Implements: SwitchCircuit as $w$.
Requires: a systems requirements database $SRDB$ and a forwarding table $shaperToSlave$ that tells us what slave is attached at each traffic shaper of the switch.
Uses: a set $TS$ of TrafficShapers connected to slave links; a set $ITS$ of TrafficShapers connected to interlinks; a set $G$ of Guardians attached to uplinks, and a set $IP$ of InputPorts attached to interlinks.

upon $\langle g, \mathrm{Deliver} \mid f\rangle$, where $g \in G$ do
    FORWARDTOOTHERSWITCH($f$);
    msg ← payload of $f$;
    if msg is an update request then
        trigger$\langle\ w, \mathrm{DeliverUQ} \mid \mathrm{msg}\ \rangle$
    else
        FORWARDTOLOCALSLAVES($f$);

upon $\langle p, \mathrm{Deliver} \mid f\rangle$, where $p \in IP$ do
    msg ← payload of $f$;
    if msg is a trigger message then
        trigger$\langle\ w, \mathrm{DeliverTM} \mid \mathrm{msg}\ \rangle$
    else if msg is an update request then
        trigger$\langle\ w, \mathrm{DeliverUQ} \mid \mathrm{msg}\ \rangle$
    else
        FORWARDTOLOCALSLAVES($f$);

procedure FORWARDTOLOCALSLAVES($f$)
    msg ← payload of $f$;
    $streamID$ ← stream identifier corresponding to msg;
    $subscribers$ ← subscribers of $streamID$ according to $SRDB$;
    $destinationTS \leftarrow \{ts \in TS \mid shaperToSlave[ts] \in subscribers\}$;
    for all $ts \in destinationTS$ do
        trigger$\langle\ ts, \mathrm{Send} \mid f\ \rangle$;

procedure FORWARDTOOTHERSWITCH($f$)
    for all $ts \in ITS$ do
        trigger$\langle\ ts, \mathrm{Send} \mid f\ \rangle$;
```

Module 8 Interface of the synchronization unit of an FTTRS switch.

Name: SyncUnit **as** u.

Requires: the duration LEC of an elementary cycle, the interlink propagation time PROP, the TM redundancy level k, the TM inter-transmission time τ, and the TM transmission time TXTM.

Events:

Indication: $\langle\, u, \mathrm{NextECWin} \mid \mathit{newWindow} \,\rangle$: Signals the start of a new window of an elementary cycle. The parameter $\mathit{newWindow}$ specifies which one and can take one of the following values: TMWIN, TAT, SYNCWIN, ASYNCWIN, and GUARD.

Algorithm 7 Implementation of the synchronization unit of an FTTRS switch.

```
Implements: SyncUnit as $u$.
Requires: the duration LEC of an elementary cycle, the interlink propagation time PROP, the TM redundancy level $k$, the TM inter-transmission time $\tau$, and the TM transmission time TXTM.
Uses: a Master $m$ and a SwitchCircuit $w$.

upon event $\langle\ u, \text{Init}\ \rangle$ do
    $isLeader \leftarrow$ preconfigured as TRUE or FALSE;   ▷ Must be TRUE for exactly one switch.
    ▷ The following durations must use the same units of time.
    $lengthTMWin \leftarrow k\tau$;
    $maxFrameLatency \leftarrow$ time to transmit and propagate a maximum-sized Ethernet frame;
    $maxTMdecodingTime \leftarrow$ maximum time for a slave to decode a trigger message;
    $lengthTAT \leftarrow \max(maxFrameLatency, maxTMdecodingTime)$;
    $lengthSyncWin \leftarrow$ NIL;
    $lengthAsyncWin \leftarrow$ NIL;
    $lengthGuardWin \leftarrow maxFrameLatency$;
    $expectedTMarrival \leftarrow [\text{NIL}]^k$;   ▷ Only used when $isLeader =$ FALSE.
    $\text{executeRendezvous}()$;
    $\text{startTimerEC}(0)$;

upon event $\langle\ u, \text{TimeoutEC}\ \rangle$ do
    $\text{startTimerEC}(\text{LEC})$;
    trigger$\langle\ u, \text{NextECWin} \mid \text{TMWIN}\ \rangle$;
    if $\neg isLeader$ then
        PREDICTTMARRIVALS();

procedure PREDICTTMARRIVALS()
    $now \leftarrow \text{getLocalTime}()$;
    for all $i \in [1, k]$ do
        $expectedTMarrival[i] \leftarrow now + \text{LEC} + \text{PROP} + (i-1)\tau + \text{TXTM}$;
```

Algorithm 7 (Continued) Implementation of the synchronization unit of an FTTRS switch.

```
upon event ⟨ $u$, TimeoutTAT ⟩ do
    trigger⟨ $u$, NextECWin | SYNCWIN ⟩
    startTimerSyncWin($lengthSyncWin$);

upon event ⟨ $u$, TimeoutSyncWin ⟩ do
    trigger⟨ $u$, NextECWin | ASYNCWIN ⟩
    startTimerAsyncWin($lengthAsyncWin$);

upon event ⟨ $u$, TimeoutAsyncWin ⟩ do
    trigger⟨ $u$, NextECWin | GUARD ⟩
    trigger⟨ $m$, ReqLSW ⟩;
    ▷ No need to set a new timer. The next timeout to elapse will be timeoutEC.

upon event ⟨ $m$, LSW | $length$ ⟩ do
    $lengthSyncWin \leftarrow length$;
    $lengthAsyncWin \leftarrow \text{LEC} - lengthTMWin - turnAroundTime - lengthSyncWin - lengthGuardWin$;

upon event ⟨ $w$, DeliverTM | tm ⟩ such that $\neg isLeader$ do
    $now \leftarrow$ getLocalTime();
    $i \leftarrow$ get sequence number of tm;
    adjustClock($now$, $expectedTMarrival[i]$);
```

Module 9 Interface of an embedded FTTRS master.

Name: Master **as** m.

Requires: a systems requirements database $SRDB$, a trigger message redundancy k, and a trigger message interarrival time τ.

Events:

Request: $\langle\ m, \text{ReqLSW} \mid c\ \rangle$: Requests the length of the synchronous window of elementary cycle c.

Request: $\langle\ m, \text{ReqSchedule} \mid c\ \rangle$: Requests the schedule for elementary cycle c.

Indication: $\langle\ m, \text{LSW} \mid \mathit{length}\ \rangle$: Indicates that the length of the next synchronous window is length.

Indication: $\langle\ m, \text{Schedule} \mid \mathit{schedule}, c\ \rangle$: Indicates that the schedule for EC c is $\mathit{schedule}$.

Algorithm 8 Implementation of an embedded FTTRS master.

```
**Implements:** Master **as** $m$.
**Requires:** a systems requirements database $SRDB$, a trigger message redundancy $k$, and a trigger message interarrival time $\tau$.
**Uses:** a SyncUnit $u$, a SwitchCircuit $w$, and a set of TrafficShapers $TS$.

upon event $\langle\ m, \text{Init}\ \rangle$ do
    $SRDB \leftarrow$ preconfigured contents;
    $commands \leftarrow$ NIL;
    $pendingRequests \leftarrow \emptyset$;
    $onHoldRequests \leftarrow \emptyset$;
    $currentEC \leftarrow 0$;
    $schedule \leftarrow \text{getSchedule}(SRDB, currentEC + 1)$;
    $phase \leftarrow$ NIL;
    $nextUpdate \leftarrow$ NIL;

upon event $\langle\ u, \text{NextECWin} \mid newWindow\ \rangle$ do
    $phase \leftarrow newWindow$;
    ▷ $newWindow$ is TMWIN, TAT, SYNCWIN, ASYNCWIN, or GUARD.

upon $phase =$ TMWIN do
    $currentEC \leftarrow currentEC + 1$;
    $seqNum \leftarrow 1$;
    $\text{uq} \leftarrow \min(pendingRequests)$;
    $\text{update}(SRDB, nextUpdate)$;
    $\text{tm} \leftarrow \text{createTM}(c, schedule, commands, \text{uq}, [k, \tau, seqNum])$;
    TRANSMITTM(tm);
    $\text{startTimerTM}(\tau)$;                    ▷ Ensure isochronous transmission.

upon event $\langle\ u, \text{TimeoutTM}\ \rangle$ do
    $seqNum \leftarrow seqNum + 1$;
    if $seqNum \leq k$ then
        $\text{uq} \leftarrow \min(pendingRequests)$;
        $\text{tm} \leftarrow \text{createTM}(c, schedule, commands, \text{uq}, [k, \tau, seqNum])$;
        TRANSMITTM(tm);
        $\text{startTimerTM}(\tau)$;                ▷ Ensure isochronous transmission.

procedure TRANSMITTM(tm)
    $f \leftarrow$ padded Ethernet frame encapsulating tm;      ▷ The padding ensures a constant length, and thus transmission time, for TMs.
    for all $ts \in TS$ do
        trigger$\langle\ ts, \text{Send} \mid f\ \rangle$; ▷ Triggering this Send event is assumed to take a negligible amount of time.
```

Algorithm 8 (Continued) Implementation of an embedded FTTRS master.

```
upon $phase = \textsc{syncWin}$ do
    if $pendingRequests \neq \emptyset$ then
        handleNextUpdateRequest();
    $schedule \leftarrow \mathrm{getSchedule}(SRDB, currentEC + 1)$;
    $pendingRequests \leftarrow pendingRequests \cup onHoldRequests$;
    $onHoldRequests \leftarrow \emptyset$;

procedure handleNextUpdateRequest()
    $\mathrm{uq} \leftarrow \min(pendingRequests)$;
    if $\mathrm{passesAdmissionControl(uq)}$ then
        $commands \leftarrow$ instructions on how the slaves should update their NRDBs;
        $nextUpdate \leftarrow \mathrm{uq}$;
    $pendingRequests \leftarrow pendingRequests \setminus \{\mathrm{uq}\}$;

upon event $\langle\, w, \mathrm{DeliverTM} \mid \mathrm{tm} \,\rangle$ do
    $\mathrm{uq} \leftarrow$ update request conveyed in $\mathrm{tm}$;
    $pendingRequests \leftarrow pendingRequests \cup \{\mathrm{uq}\}$;

upon event $\langle\, w, \mathrm{DeliverUQ} \mid \mathrm{uq} \,\rangle$ do
    if $phase \neq \textsc{guard} \wedge phase \neq \textsc{tmWin} \wedge phase \neq \textsc{tat}$ then
        $pendingRequests \leftarrow pendingRequests \cup \{\mathrm{uq}\}$;
    else
        $onHoldRequests \leftarrow onHoldRequests \cup \{\mathrm{uq}\}$;

upon event $\langle\, m, \mathrm{ReqLSW} \mid c \,\rangle$ do
    $\ell \leftarrow \mathrm{getLSW}(SRDB, c)$;
    trigger$\langle\, m, \mathrm{LSW} \mid \ell \,\rangle$;

upon event $\langle\, m, \mathrm{ReqSchedule} \mid c \,\rangle$ do
    $\mathrm{sched} \leftarrow \mathrm{getSchedule}(SRDB, c)$;
    trigger$\langle\, m, \mathrm{Schedule} \mid \mathrm{sched}, c \,\rangle$;
```

Appendix C

Probability of Losing All Trigger Messages of an Elementary Cycle

Let us assume that the duration of a trigger message is d. Moreover, let us assume that the bit error rate is λ and that bit errors arrive following a Poisson process. In that case, a trigger message is not corrupted on a link if during the duration of a trigger message no bit error arrives. Specifically, if the bit errors arrive following a Poisson process, then the probability of corrupting a trigger message only depends on the duration of the trigger message, and not on what has happened before or what is going to happen afterwards. This is due to the stationary increments property of a Poisson process which states that the number of arrivals counted in any time interval only depends on the length of the interval.[1] Thus, we just have to look at the counting process $\{N(t) \mid t \geq 0\}$ of a Poisson process of parameter λ and calculate the probability that the counting process counts zero arrivals in a time interval $(0, d]$. In other words, we simply have to calculate the following probability to obtain the probability that a trigger message is not corrupted:

$$\Pr\left(N(d) = 0\right)$$

Since we are talking about a Poisson process, the above probability is given by a Poisson distribution:

$$\Pr\left(N(d) = 0\right) = \frac{(\lambda d)^0 e^{-\lambda d}}{0!} = e^{-\lambda d}.$$

[1] Sheldon M. Ross. *Introduction to Probability Models*. 9th. Elsevier, 2007, Sec. 5.3. The Poisson Process, p. 303.

From the above it follows immediately that the probability that one trigger message is corrupted is

$$\Pr\left(N(d) \neq 0\right) = 1 - \Pr\left(N(d) = 0\right) = 1 - e^{-\lambda d}.$$

Now, if we know the probability that one trigger message is corrupted, we can calculate the probability that k trigger messages are corrupted quite easily due to the independent increments property of a Poisson process, which states that the number of arrivals counted in disjoint intervals are independent of each other.[2] We can do this by simply multiplying k times the probability that a single trigger message is corrupted. Thus, if we have k trigger messages per elementary cycle, the probability of all of them being corrupted is

$$\Pr(k \text{ TMs corrupted}) = \prod_{i=1}^{k} 1 - e^{-\lambda d} = (1 - e^{-\lambda d})^k.$$

Hence, the probability of losing k trigger messages is $(1 - e^{-\lambda d})^k$.

In the particular case of the elementary cycle synchronization experiments (Section 9.3), links are operating at 100 Mbps and the trigger messages have the minimum permissible length in Ethernet, i.e., 72 bytes, including the preamble and start of frame delimiter. We thus have that in those experiments the duration of a trigger message is $d = (72 \cdot 8)/100 = 5.76\,\mu\text{s}$. Moreover, in those experiments we chose a value of $k = 4$ for the trigger message redundancy, assumed a bit error ratio of 10^{-6}, and elementary cycles of $1000\,\mu\text{s}$. A bit error ratio of 10^{-6} corresponds to a bit error rate of

$$\lambda = \frac{1 \text{ error}}{10^6 \cancel{\text{bits}}} \cdot \frac{10^6 \cancel{\text{bits}}}{1 \cancel{\text{megabit}}} \cdot \frac{100 \cancel{\text{megabit}}}{1 \cancel{\text{s}}} \cdot \frac{1 \cancel{\text{s}}}{10^6 \mu\text{s}} = 10^{-4} \text{ errors}/\mu\text{s}.$$

Thus, the probability of losing all 4 trigger messages of a given trigger message window on a given link is

$$p = (1 - e^{-\lambda d})^k = \left(1 - e^{-10^{-4} \cdot 5.76}\right)^4 \approx 1.1 \cdot 10^{-13}.$$

Let us call p the **EC corruption probability** and let us refer to the event of all trigger messages of an elementary cycle being corrupted as an **EC corruption**.

We can now define a random variable X that gives the number of elementary cycles until an EC corruption occurs on a given link. In that case, X follows a geometric distribution with parameter p, i.e., $X \sim \text{Geo}(p)$. Since the expectation

[2] Ross, *Introduction to Probability Models*, Sec. 5.3. The Poisson Process, p. 303.

of a geometric distribution of parameter p is $\mathrm{E}(X) = 1/p$, we have that it takes on average $1/(1.1 \cdot 10^{-13}) \approx 9.09 \cdot 10^{12}$ elementary cycles until a first EC corruption. If the elementary cycles have a duration of $1000\,\mu\mathrm{s}$, an EC corruption occurs on average once every $9.09 \cdot 10^{15}$ µs, or roughly every 288 years. This means that with $k = 4$ the mean time to EC corruption on a single link is 288 years.

Bibliography

Abramson, Norman. “Development of the ALOHANET”. In: *IEEE Transactions on Information Theory* 31.2 (1985), pp. 119–123 (cit. on p. 102).

— “The ALOHA System: Another Alternative for Computer Communications”. In: *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*. ACM. 1970, pp. 281–285 (cit. on p. 102).

Adrover, Andreu. *Log files for the FTTRS prototype fault-injection experiments*. URL: http://srv.uib.es/experimental-evaluation-of-fttrs/ (cit. on p. 258).

Aeronautical Radio, Incorporated (ARINC). *Aircraft Data Network — Part 7, Avionics Full Duplex Switched Ethernet Network (AFDX)*. ARINC 664P7-1 (ARINC). 2009 (cit. on pp. 106, 108).

Albert, Amos. “Comparison of Event Triggered and Time Triggered Concepts with Regard to Distributed Control Systems”. In: *Embedded World* 2004 (2004), pp. 235–252 (cit. on p. 100).

Almeida, Luís, Paulo Pedreiras, and José A. Fonseca. “The FTT-CAN Protocol: Why and How”. In: *IEEE Transactions on Industrial Electronics* 49.6 (2002), pp. 1189–1201 (cit. on pp. 125, 141).

Almeida, Luís and Paulo Pedreiras. “Approaches to Enforce Real-Time Behavior in Ethernet”. In: *The Industrial Communication Technology Handbook*. Ed. by Richard Zurawski. CRC Press, 2005. Chap. 20 (cit. on p. 109).

Álvarez, Inés. “Implementation and Verification of the Slave Elementary Cycle Synchronization Mechanism of the Flexible Time-Triggered Replicated Star for Ethernet”. Bachelor Thesis. Universitat de les Illes Balears, 2014 (cit. on p. 252).

— “Study of the Admission Control in the Flexible Time-Triggered and the Audio Video Bridging Communication Protocols”. MA thesis. Universitat de les Illes Balears, 2016 (cit. on p. 238).

Álvarez, Inés, Mladen Knezic, Luis Almeida, and Julián Proenza. “A First Performance Analysis of the Admission Control in the HaRTES Ethernet Switch”.

In: *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*. IEEE. 2016 (cit. on pp. 164, 238).

Amari, Suprasad V., Albert F. Myers, Antoine Rauzy, and Kishor S. Trivedi. "Imperfect Coverage Models: Status and Trends". In: *Handbook of Performability Engineering*. Springer, 2008, pp. 321–348 (cit. on p. 59).

America: Department of Defense, United States of. *Military Handbook: Reliability Prediction of Electronic Equipment: MIL-HDBK-217F*. Department of defense, 1991 (cit. on p. 290).

American Association for the Advancement of Science. *Articles for* Science (cit. on p. xix).

American Institute of Physics. *AIP Style Manual*. 1990 (cit. on p. xviii).

American Psychological Association. *The Publication Manual of the American Psychological Association*. 2nd. American Psychological Association, 1974 (cit. on p. xviii).

Anderson, James M., Kalra Nidhi, Karlyn D. Stanley, Paul Sorensen, Constantine Samaras, and Oluwatobi A. Oluwatola. *Autonomous Vehicle Technology: a Guide for Policymakers*. Rand Corporation, 2014 (cit. on p. 5).

Arun, K. and H. Nitin. "Understanding Fault Tolerance and Reliability". In: *Computer* (1997) (cit. on pp. 56, 57).

Ashjaei, Mohammad, Moris Behnam, Thomas Nolte, Luís Almeida, and Ricardo Marau. "A Compact Approach to Clustered Master Slave Ethernet Networks". In: *Proceedings of the 9th IEEE International Workshop on Factory Communication Systems (WFCS 2012)*. IEEE. 2012, pp. 157–160 (cit. on p. 134).

Ashjaei, Mohammad, Yong Du, Luís Almeida, Moris Behnam, and Thomas Nolte. "Dynamic Reconfiguration in HaRTES Switched Ethernet Networks". In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016)*. IEEE. 2016, pp. 1–8 (cit. on p. 138).

Ashjaei, Mohammad, Paulo Pedreiras, Moris Behnam, Luís Almeida, and Thomas Nolte. "Dynamic Reconfiguration in Multi-Hop Switched Ethernet Networks". In: *ACM SIGBED Review* 11.3 (2014) (cit. on pp. 131, 135).

— "Evaluation of Dynamic Reconfiguration Architecture in Multi-Hop Switched Ethernet Networks". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014, pp. 1–4 (cit. on p. 134).

— "Supporting Multi-Hop Communications with HaRTES Ethernet Switches". In: *Proceedings of the IEEE Real-Time Systems Symposium Work-in-Progress session (RTSS 2013)*. Vancouver, Canada, Dec. 2013 (cit. on p. 138).

Ashjaei, Mohammad, Luís Silva, Moris Behnam, Paulo Pedreiras, Reinder J. Bril, Luís Almeida, and Thomas Nolte. "Improved Message Forwarding for Multi-

Hop HaRTES Real-Time Ethernet Networks”. In: *Journal of Signal Processing Systems* 84.1 (2016), pp. 47–67 (cit. on p. 138).

Austin, Todd, Valeria Bertacco, Scott Mahlke, and Yu Cao. “Reliable Systems on Unreliable Fabrics”. In: *IEEE Design and Test of Computers* 25.4 (2008) (cit. on p. 283).

Avizienis, Algirdas and Jean-Claude Laprie, eds. *Dependable Computing for Critical Applications*. Vol. 4. Springer Science & Business Media, 2012 (cit. on p. 56).

Avižienis, Algirdas, Jean-Claude Laprie, and Brian Randell. *Fundamental Concepts of Dependability*. University of Newcastle upon Tyne, Computing Science Newcastle upon Tyne, UK, 2001 (cit. on pp. 19, 20).

Avižienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 (cit. on pp. 19, 20, 22–24, 27, 29–34, 36–38, 41–44, 53, 54, 154, 155).

Baher, Jeffrey D. “Switch Pioneer Kalpana Now Offers Bidirectional Ethernet”. In: *PC Magazine* November 9 Issue (Nov. 1993) (cit. on p. 119).

Ballesteros, Alberto. *FT4FTT final prototype demo*. 2017. URL: http://srv.uib.es/ft4ftt-final-prototype-demo/ (visited on 2017-03-08) (cit. on p. 267).

— “Implementation and Testing of the Node Replication Scheme of the FT4FTT Architecture”. MA thesis. Universitat de les Illes Balears, 2016 (cit. on pp. 263, 267).

Ballesteros, Alberto, Sinisa Derasevic, Manuel Barranco, and Julián Proenza. “First Implementation and Test of Reintegration Mechanisms for Node Replicas in the FT4FTT Architecture”. In: *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*. IEEE. 2016 (cit. on pp. xv, 263, 267).

Ballesteros, Alberto, Sinisa Derasevic, David Gessner, Francisca Font, Inés Álvarez, Manuel Barranco, and Julián Proenza. “First Implementation and Test of a Node Replication Scheme on Top of the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016)*. IEEE. 2016 (cit. on pp. xv, 242, 263).

Ballesteros, Alberto, David Gessner, Julián Proenza, Manuel Barranco, and Paulo Pedreiras. “Towards Preventing Error Propagation in a Real-Time Ethernet Switch”. In: *Proceedings of the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2013)*. IEEE, 2013 (cit. on pp. xiv, 172).

Ballesteros, Alberto and Julián Proenza. *A Description of the FTT-SE Protocol*. Tech. rep. A-06-2013. Universitat de les Illes Balears, Dec. 2013 (cit. on pp. 127, 129, 136, 221).

Ballesteros, Alberto, Julián Proenza, David Gessner, Guillermo Rodriguez-Navas, and Thilo Sauter. "Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star for Ethernet". In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE, 2014 (cit. on pp. xiv, 215, 242).

Barborak, Michael, Anton Dahbura, and Miroslaw Malek. "The Consensus Problem in Fault Tolerant Computing". In: *ACM Computing Surveys (CSur)* 25.2 (1993), pp. 171–220 (cit. on p. 44).

Barranco, Manuel. "Improving Error Containment and Reliability of Communication Subsystems Based on Controller Area Network (CAN) by Means of Adequate Star Topologies". PhD thesis. Universitat de les Illes Balears, 2010 (cit. on p. 48).

Barranco, Manuel and Julián Proenza. "Towards Understanding the Sensitivity of the Reliability Achievable by Simplex and Replicated Star Topologies in CAN". In: *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2011)*. IEEE. 2011, pp. 1–4 (cit. on pp. 287, 289–291).

Barranco, Manuel, Julián Proenza, and Luís Almeida. "Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate". In: *Computer* 42.5 (May 2009), pp. 66–73 (cit. on pp. 143, 285).

Barranco, Manuel, Julián Proenza, and Luís Almeida. "Quantitative Characterization of the Reliability of Simplex Buses and Stars to Compare their Benefits in Fieldbuses". In: *Reliability Engineering & System Safety* 138 (2015), pp. 163–175 (cit. on p. 285).

— "Quantitative Comparison of the Error-Containment Capabilities of a Bus and a Star Topology in CAN Networks". In: *IEEE Transactions on Industrial Electronics* 58.3 (2011), pp. 802–813 (cit. on p. 285).

Barranco, Manuel, Guillermo Rodriguez-Navas, David Gessner, and Julián Proenza. "Towards the Integration of Flexible Time Triggered Communication and Replicated Star Topologies in CAN". In: *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2011)*. IEEE, 2011, pp. 1–4 (cit. on p. 144).

Bauer, Günther, Hermann Kopetz, and Wilfried Steiner. "Byzantine Fault Containment in TTP/C". In: *Proceedings of the 2002 International Workshop on Real-Time LANs in the Internet Age (RTLIA 2002)* (2002), p. 9 (cit. on p. 64).

Beck, Michael. *Ethernet in the First Mile: The IEEE 802.3ah EFM Standard (Communications Engineering Series)*. McGraw-Hill Education, 2005 (cit. on p. 105).

Boggs, D. R., J. C. Mogul, and C. A. Kent. "Measured Capacity of an Ethernet: Myths and Reality". In: *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM '88. Stanford, California, USA: ACM, 1988, pp. 222–234 (cit. on p. 115).

Bower, Joseph L. and Clayton M. Christensen. *Disruptive Technologies: Catching the Wave*. Harvard Business Review, 1995 (cit. on p. 121).

Budhiraja, Navin, Keith Marzullo, Fred B. Schneider, and Sam Toueg. "The Primary-Backup Approach". In: *Distributed systems* 2 (1993), pp. 199–216 (cit. on p. 86).

Burns, Alan and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. 3rd. Pearson Education, 2001 (cit. on p. 93).

Buttazzo, Giorgio. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. Vol. 24. Springer Science & Business Media, 2011 (cit. on pp. 92, 97, 98).

Bux, Werner. "Local-Area Subnetworks: a Performance Comparison". In: *IEEE Transactions on Communications* 29.10 (1981), pp. 1465–1473 (cit. on p. 115).

Cachin, Christian, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011 (cit. on pp. 305, 306).

Carter, William C. "A Time for Reflection". In: *Proceedings of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS-8)*. 1982, p. 41 (cit. on p. 18).

Choi, Jung Il, Mayank Jain, Kannan Srinivasan, Phil Levis, and Sachin Katti. "Achieving Single Channel, Full Duplex Wireless Communication". In: *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*. MobiCom '10. Chicago, Illinois, USA: ACM, 2010, pp. 1–12 (cit. on p. 103).

Christian, Brian and Tom Griffiths. *Algorithms to Live By: The Computer Science of Human Decisions*. Henry Holt and Co, 2016 (cit. on p. 219).

Constantinescu, Cristian. "Trends and Challenges in VLSI Circuit Reliability". In: *IEEE Micro* 23.4 (2003), pp. 14–19 (cit. on p. 283).

Coyle, Edward J. and Bede Liu. "A Matrix Representation of CSMA/CD Networks". In: *IEEE Transactions on Communications* 33.1 (1985), pp. 53–64 (cit. on p. 115).

— "Finite Population CSMA/CD Networks". In: *IEEE Transactions on Communications* 31.11 (1983), pp. 1247–1251 (cit. on p. 115).

Cristian, Flaviu. “Understanding Fault-Tolerant Distributed Systems”. In: *Communications of the ACM* 34.2 (1991), pp. 56–78 (cit. on p. 60).

Danielis, Peter, Jan Skodzik, Vlado Altmann, Eike Bjoern Schweissguth, Frank Golatowski, Dirk Timmermann, and Joerg Schacht. “Survey on Real-Time Communication via Ethernet in Industrial Automation Environments”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014 (cit. on p. 106).

Davoli, Renzo. “VDE: Virtual Distributed Ethernet”. In: *Proceedings of the 1st IEEE International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM 2005)*. IEEE. 2005, pp. 213–220 (cit. on p. 248).

Défago, Xavier, André Schiper, and Nicole Sergent. “Semi-Passive Replication”. In: *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS 1998)*. 96. 1998 (cit. on p. 88).

Défago, Xavier, André Schiper, and Péter Urbán. “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey”. In: *ACM Computing Surveys* 36 (2003) (cit. on pp. 82, 84).

Derasevic, Sinisa, Manuel Barranco, and Julián Proenza. “Appropriate Consistent Replicated Voting for Increased Reliability in a Node Replication Scheme over FTT”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014, pp. 1–4 (cit. on pp. 232, 255).

— “Designing Fault-Diagnosis and Reintegration to Prevent Node Redundancy Attrition in Highly Reliable Control Systems Based on FTT-Ethernet”. In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016)*. IEEE. 2016 (cit. on p. 232).

— “Designing Fault-Diagnosis and Reintegration to Prevent Node Redundancy Attrition in Highly Reliable Control Systems Based on FTT-Ethernet”. In: *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016)*. IEEE. 2016 (cit. on pp. 255, 263).

Derasevic, Sinisa, Maties Melia, Alberto Ballesteros, Manuel Barranco, and Julián Proenza. “First Experimental Evaluation of the Consistent Replicated Voting in the Hard Real-Time Ethernet Switching Architecture”. In: *Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2015)*. IEEE. 2015 (cit. on p. 232).

Derasevic, Sinisa, Julián Proenza, and Manuel Barranco. “Using FTT-Ethernet for the Coordinated Dispatching of Tasks and Messages for Node Replication”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014 (cit. on pp. 194, 232, 255, 263).

Digital Equipment Corporation, Intel Corporation, and Xerox Corporation. *The Ethernet, a Local Area Network, Data Link Layer and Physical Layer Specification*. Sept. 1980 (cit. on p. 103).

Dijkstra, Edsger W. *On the reliability of programs*. No date. URL: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html (visited on 2017-03-16) (cit. on p. 271).

Dudek, R. A., S. S. Panwalkar, and M. L. Smith. "The Lessons of Flowshop Scheduling Research". In: *Operations Research* 40.1 (1992), pp. 7–13 (cit. on p. 96).

Ethernet POWERLINK Standardisation Group. *Ethernet POWERLINK Communication Profile Specification — Version 1.3.0*. EPSG Draft Standard 301 (EPSG). 2016 (cit. on p. 108).

Fagin, Ronald, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about knowledge*. The MIT press, 1995 (cit. on p. 218).

Ferrant, Jean-Loup, Mike Gilson, Sebastien Jobert, Michael Mayer, Laurent Montini, Michel Ouellette, Silvana Rodrigues, and Stefano Ruffini. *Synchronous Ethernet and IEEE 1588 in Telecoms: Next Generation Synchronization Networks*. John Wiley & Sons, 2013 (cit. on p. 215).

Ferreira, Joaquim. "Fault-Tolerance in Flexible Real-Time Communication Systems". PhD thesis. Universidade de Aveiro, 2005 (cit. on pp. 128, 141, 142, 171, 214).

Ferreira, Joaquim, Luís Almeida, José A. Fonseca, Paulo Pedreiras, Ernesto Martins, Guillermo Rodriguez-Navas, Joan Rigo, and Julián Proenza. "Combining Operational Flexibility and Dependability in FTT-CAN". In: *IEEE Transactions on Industrial Informatics* 2.2 (2006), pp. 95–102 (cit. on p. 141).

Ferreira, Joaquim, Paulo Pedreiras, Luís Almeida, and José A. Fonseca. "Achieving Fault Tolerance in FTT-CAN". In: *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS 2002)* (2002), pp. 125–132 (cit. on pp. 142, 171).

Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382 (cit. on p. 170).

Frazier, Howard and Gerry Pesavento. "Ethernet Takes on the First Mile". In: *IT Professional* 3.4 (2001), pp. 17–22 (cit. on p. 105).

Fujiwara, Toru, Tadao Kasami, and Shu Lin. "Error detecting capabilities of the shortened Hamming codes adopted for error detection in IEEE standard 802.3". In: *IEEE Transactions on Communications* 37.9 (1989), pp. 986–989 (cit. on p. 156).

Gall, John. *Systemantics: How Systems Work and Especially How They Fail*. Times Books, 1977 (cit. on p. 158).

Gastel, Barbara and Robert A Day. *How to write and publish a scientific paper*. 8th. ABC-CLIO, 2016 (cit. on p. xviii).

Gessner, David, Inés Álvarez, Alberto Ballesteros, Manuel Barranco, and Julián Proenza. “Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE, 2014 (cit. on pp. xiii, 242).

Gessner, David, Alberto Ballesteros, Andreu Adrover, and Julián Proenza. “Experimental Evaluation of Network Component Crashes and Trigger Message Omissions in the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 11th IEEE World Conference on Factory Communication Systems (WFCS 2015)*. IEEE, 2015 (cit. on pp. xiii, 242).

Gessner, David, Manuel Barranco, Julián Proenza, and Michael Short. “A First Qualitative Evaluation of Star Replication Schemes for FTT-CAN”. In: *Proceedings of the 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2012)*. IEEE, 2012 (cit. on pp. xiii, 131, 144).

Gessner, David, Ignasi Furió, and Julián Proenza. “Towards a Layered Architecture for the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2015)*. IEEE, 2015 (cit. on p. xiv).

Gessner, David, Paulo Portugal, Julián Proenza, and Manuel Barranco. “Towards a Reliability Analysis of the Design Space for the Communication Subsystem of FT4FTT”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE, 2014 (cit. on pp. xiv, 236).

Gessner, David, Julián Proenza, and Manuel Barranco. “A Proposal for Managing the Redundancy Provided by the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 10th IEEE International Workshop on Factory Communication Systems (WFCS 2014)*. IEEE, 2014 (cit. on p. xii).

— “A Proposal for Master Replica Control in the Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 10th IEEE International Workshop on Factory Communication Systems (WFCS 2014)*. IEEE, 2014 (cit. on p. xii).

Gessner, David, Julián Proenza, Manuel Barranco, and Luís Almeida. “Towards a Flexible Time-Triggered Replicated Star for Ethernet”. In: *Proceedings of the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2013)*. IEEE, 2013 (cit. on p. xii).

Goldberg, David. “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. In: *ACM Computing Surveys (CSUR)* 23.1 (1991) (cit. on p. 79).

Gonsalves, Timothy A. and Fouad A. Tobagi. “On the Performance Effects of Station Locations and Access Protocol Parameters in Ethernet Networks”. In: *IEEE Transactions on Communications* 36.4 (1988), pp. 441–449 (cit. on p. 115).

Grottke, Michael, Rivalino Matias, and Kishor S. Trivedi. “The Fundamentals of Software Aging”. In: *IEEE Proceedings of Workshop on Software Aging and Rejuvenation, in conjunction with ISSRE. Seattle, WA*. 2008 (cit. on p. 40).

Hsueh, Mei-Chen, Timothy K. Tsai, and Ravishankar K. Iyer. “Fault injection techniques and tools”. In: *Computer* 30.4 (1997), pp. 75–82 (cit. on p. 246).

Huynh, Minh, Stuart Goose, and Prasant Mohapatra. “Resilience technologies in Ethernet”. In: *Computer Networks* 54.1 (2010) (cit. on p. 68).

Institute of Electrical and Electronics Engineers (IEEE). *Enhancements for Scheduled Traffic — Draft 2.1*. 802.1Qbv (IEEE). 2014 (cit. on p. 108).

— *Frame Preemption — Draft 1.1*. 802.1Qbu (IEEE). 2014 (cit. on p. 108).

— *Frame Replication and Elimination for Reliability — Draft 0.5*. 802.1CB (IEEE). 2014 (cit. on p. 108).

— *IEEE Authorship Series: How to Write for Technical Periodicals & Conferences*. URL: https://www.ieee.org/publications_standards/publications/authors/author_guide_interactive.pdf (visited on 2016-10-14) (cit. on pp. xvii, xx).

— *Path Control and Reservation — Draft 1.1*. 802.1Qca (IEEE). 2014 (cit. on p. 108).

— *Preparation of Papers for IEEE transactions and journals*. URL: https://www.ieee.org/documents/transactions_journals.pdf (visited on 2016-10-14) (cit. on p. xx).

— *Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. 1588-2008 (IEEE). 2008 (cit. on p. 215).

— *Standard for Ethernet—Section three*. 802.3-2015 (IEEE). 2015 (cit. on p. 104).

— *Standard for Information technology– Local and Metropolitan Area Networks — Part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment: Media Access Control Parameters, Physical Layers, and Management Parameters for Subscriber Access Networks*. 802.3ah-2004 (IEEE). 2004 (cit. on p. 105).

— *Standard for Local and Metropolitan Area Networks — Full Duplex Operation*. 802.3x-1997 (IEEE). 1997 (cit. on p. 119).

— *Standard for Local and Metropolitan Area Networks: Overview and Architecture*. 802-2001 (IEEE). 2001 (cit. on p. 249).

Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*. 802.1AS-2011 (IEEE). 2011 (cit. on p. 109).

— *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 10: Congestion Notification*. 802.1Qau-2010 (IEEE). 2010 (cit. on p. 105).

— *Standard for Local and Metropolitan Area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*. 802.1Qav-2009 (IEEE). 2009 (cit. on p. 109).

— *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 17: Priority-based Flow Control*. 802.1Qbb-2011 (IEEE). 2011 (cit. on p. 106).

— *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 18: Enhanced Transmission Selection*. 802.1Qaz-2011 (IEEE). 2011 (cit. on p. 106).

— *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 4: Provider Bridges*. 802.1ah-2005 (IEEE). 2005 (cit. on p. 105).

— *Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 7: Provider Backbone Bridges*. 802.1ah-2008 (IEEE). 2008 (cit. on p. 105).

— *Standard for Local and Metropolitan Area networks–Audio Video Bridging (AVB) Systems*. 802.1BA-2011 (IEEE). 2011 (cit. on p. 109).

— *Standard for Local and Metropolitan Area Networks—Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 20: Shortest Path Bridging*. 802.1aq-2012 (IEEE). 2012 (cit. on p. 107).

— *Standard for Local and Metropolitan Area networks–Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP)*. 802.1Qat-2010 (IEEE). 2010 (cit. on p. 109).

— *Standard for Local Area Network MAC (Media Access Control) Bridges*. 802.1D-1998 (IEEE). 1998 (cit. on p. 107).

— *Standard for Local Area Network MAC (Media Access Control) Bridges*. 802.1D-2004 (IEEE). 2004 (cit. on p. 107).

— *Standards for Local Area Networks: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. 802.3-1985 (ANSI/IEEE). 1985 (cit. on p. 104).

— *Stream Reservation Protocol (SRP) Enhancements and Performance Improvements — Draft 0.2*. 802.1Qcc (IEEE). 2014 (cit. on p. 108).

— *Timing and Synchronization: Enhancements and Performance Improvements — Draft 0.7*. 802.1ASbt (IEEE). 2014 (cit. on p. 108).

International Electrotechnical Commission (IEC). *Communication networks and systems in substations — Part 8-1: Specific Communication Service Mapping (SCSM) — Mappings to MMS (ISO 9506-1 and ISO 9506-2) and to ISO/IEC 8802-3*. IEC 61850-8-1 (IEC) (cit. on pp. 184, 270).

— *Industrial communication networks—high availability automation networks—part 2: media redundancy protocol (MRP)*. IEC 62439-2 (IEC) (cit. on p. 107).

— *Industrial communication networks—high availability automation networks—part 3: parallel redundancy protocol (PRP) and high-availability seamless redundancy (HSR)*. IEC 62439-3 (IEC) (cit. on p. 108).

International Organization for Standardization (ISO). *Information technology — Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO/IEC 7498-1:1994. Geneva, 1994 (cit. on pp. 107, 112).

— *Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling*. ISO 11898-1:2015 (ISO). 2015 (cit. on pp. 130, 140).

Jensen, E. Douglas, C. Douglas Locke, and Hideyuki Tokuda. "A Time-Driven Scheduling Model for Real-Time Operating Systems." In: *Proceedings of the IEEE Real-Time Systems Symposium (RTSS 1985)*. Vol. 85. 1985, pp. 112–122 (cit. on p. 93).

Jetway Computer Corp. *Jetway JBC373F38-525-B*. URL: http://www.jetwaycomputer.com/JBC373F38.html (visited on 2017-01-03) (cit. on p. 244).

Johnson, Barry W. *Design and Analysis of Fault Tolerant Systems*. Ed. by Harold S. Stone. Addison-Wesley Publishing Company, Inc., 1989 (cit. on p. 55).

Johnson, Selmer Martin. "Optimal Two and Three Stage Production Schedules with Setup Times Included". In: *Naval research logistics quarterly* 1.1 (1954), pp. 61–68 (cit. on p. 96).

Jonathan Amos. *Schiaparelli Mars probe's parachute 'jettisoned too early'*. 2016. URL: http://www.bbc.com/news/science-environment-37715202 (visited on 2016-11-14) (cit. on p. 4).

Kasim, Abdul, Prasanna Adhikari, Nan Chen, Norman Finn, Nasir Ghani, Marek Hajduczenia, Paul Havala, Giles Heron, Michael Howard, and Luca Martini. *Delivering Carrier Ethernet: Extending Ethernet Beyond the LAN*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2008 (cit. on p. 105).

Kempf, Mark F. *Bridge Circuit for Interconnecting Networks*. US Patent 4,597,078. June 1986 (cit. on pp. 115, 117).

Kenney, Malachi. *Ping of Death*. 1996. URL: http://insecure.org/sploits/ping-o-death.html (visited on 2016-12-16) (cit. on p. 182).

Kirrmann, H., K. Weber, O. Kleineberg, and H. Weibel. "Seamless and Low-Cost Redundancy for Substation Automation Systems (High Availability Seamless

Redundancy, HSR)". In: *Power and Energy Society General Meeting, 2011 IEEE*. July 2011, pp. 1–7 (cit. on p. 70).

Kirrmann, Hubert, Mats Hansson, and Peter Muri. "IEC 62439 PRP: Bumpless Recovery for Highly Available, Hard Real-Time Industrial Networks". In: *Proceedings of the 12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2007)*. IEEE. 2007 (cit. on p. 68).

Knuth, Donald E. "The toilet paper problem". In: *The American Mathematical Monthly* 91.8 (1984), pp. 465–470 (cit. on p. xxi).

Kohlhepp, Robert J. "The 10 Most Important Products of the Decade — Number Five: Kalpana EtherSwitch". In: *Network Computing* (2000) (cit. on p. 117).

Koopman, Philip. "32-bit cyclic redundancy codes for Internet applications". In: *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2002)*. IEEE. 2002, pp. 459–468 (cit. on p. 156).

Kopetz, Hermann. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science & Business Media, 2011 (cit. on p. 3).

Kopetz, Hermann, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. "Distributed Fault Tolerant Real-Time Systems: the MARS Approach". In: *IEEE Micro* 9.1 (1989), pp. 25–40 (cit. on pp. 184, 270).

Kopetz, Hermann and Gunter Grunsteidl. "TTP — A Protocol for Fault-Tolerant Real-Time Systems". In: *Computer* 27.1 (1994), pp. 14–23 (cit. on pp. 184, 270).

Lamport, Leslie. "The Part-Time Parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169 (cit. on p. xxi).

Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401 (cit. on pp. xxi, 48, 58).

Laprie, Jean-Claude. *Dependability: Basic Concepts and Terminology*. Springer Verlag, 1992 (cit. on pp. 18–20, 29, 36, 44).

— "Dependable Computing and Fault-Tolerance". In: *Digest of Papers FTCS-15* (1985), pp. 2–11 (cit. on pp. 18, 20, 29).

Latronico, Elizabeth Ann. "Reliability Validation of Group Membership Services for X-by-Wire Protocols". PhD thesis. Electrical and Computer Engineering Department, Carnegie Mellon University, May 2005 (cit. on p. xx).

Lin, Shu, Daniel J. Costello Jr, and Michael J. Miller. "Automatic-Repeat-Request Error Control Schemes". In: *IEEE Communications Magazine* 22.12 (1984) (cit. on p. 183).

Lynch, Nancy A. *Distributed algorithms*. Morgan Kaufmann, 1996 (cit. on pp. 160, 170).

Marau, Ricardo. “Real-Time Communications over Switched Ethernet Supporting Dynamic QoS Management”. PhD thesis. Universidade de Aveiro, 2009 (cit. on pp. 129, 132, 134–136, 226, 243).

Marau, Ricardo, Luís Almeida, José A. Fonseca, Joaquim Ferreira, and Valter F. Silva. *Assessment of FTT-CAN Master Replication Mechanisms for Safety Critical Applications*. Tech. rep. SAE Technical Paper, 2006 (cit. on p. 141).

Marau, Ricardo, Luís Almeida, and Paulo Pedreiras. “Enhancing Real-Time Communication over COTS Ethernet Switches”. In: *Proceedings of the 6th IEEE International Workshop on Factory Communication Systems (WFCS 2006)*. Vol. 6. IEEE. 2006, pp. 295–302 (cit. on pp. 8, 134, 243).

Marau, Ricardo, Paulo Pedreiras, and Luís Almeida. “Asynchronous Traffic Signaling over Master Slave Switched Ethernet Protocols”. In: *6th International Workshop on Real-Time Networks (RTN 2007)*. 2007 (cit. on p. 136).

Marques, Luís, V. Vasconcelos, Paulo Pedreiras, Luís Almeida, and Valter Silva. “Towards Efficient Transient Fault Handling in Time-Triggered Systems”. In: (2011) (cit. on p. 143).

Marques, Luís, Verónica Vasconcelos, Paulo Pedreiras, and Luís Almeida. “Comparing Scheduling Policies for a Message Transient Error Recovery Server in a Time Triggered Setting”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*. IEEE. 2014, pp. 1–6 (cit. on p. 143).

— “Error Recovery in Time Triggered Communication Systems Using Servers”. In: *Proceedings of the 8th IEEE International Symposium on Industrial and Embedded Systems (SIES 2013)*. IEEE. 2013, pp. 205–212 (cit. on p. 143).

— “Tolerating Transient Communication Faults with Online Traffic Scheduling”. In: *IEEE International Conference on Industrial Technology (ICIT 2012)*. Athens, 2012 (cit. on p. 143).

Marques, Luís, Verónica Vasconcelos, Paulo Pedreiras, Luís Almeida, and Valter Silva. *Towards Efficient Transient Fault Handling in Time-Triggered Systems*. Tech. rep. Aveiro: Universidade de Aveiro, 2011 (cit. on p. 143).

Matheus, Kirsten and Thomas Königseder. *Automotive Ethernet*. Cambridge University Press, 2014 (cit. on p. 106).

MathWorks. *Control of an Inverted Pendulum on a Cart*. URL: https://es.mathworks.com/help/control/examples/control-of-an-inverted-pendulum-on-a-cart.html (visited on 2016-11-03) (cit. on p. 264).

Merriam-Webster.com. *Dependable*. 2016. URL: http://www.merriam-webster.com/dictionary/dependable (visited on 2016-05-18) (cit. on p. 19).

Metcalfe, Robert M. and David R. Boggs. “Ethernet: Distributed Packet Switching for Local Computer Networks”. In: *Commun. ACM* 19.7 (July 1976), pp. 395–404 (cit. on p. 103).

Metro Ethernet Forum. *Website of the Metro Ethernet Forum*. 2014. URL: https://www.mef.net (visited on 2016-05-15) (cit. on p. 105).

MITRE Corporation. *CVE-1999-0128*. 2008. URL: https://www.cvedetails.com/cve/CVE-1999-0128/ (visited on 2016-12-16) (cit. on p. 182).

Mudoi, Uday. *Ethernet Evolves Again To Meet The Internet Of Things*. 2014. URL: http://electronicdesign.com/communications/ethernet-evolves-again-meet-internet-things (visited on 2017-01-05) (cit. on p. 106).

Nagatani, Keiji, Seiga Kiribayashi, Yoshito Okada, Kazuki Otake, Kazuya Yoshida, Satoshi Tadokoro, Takeshi Nishimura, Tomoaki Yoshida, Eiji Koyanagi, Mineo Fukushima, et al. “Emergency Response to the Nuclear Accident at the Fukushima Daiichi Nuclear Power Plants Using Mobile Rescue Robots”. In: *Journal of Field Robotics* 30.1 (2013), pp. 44–63 (cit. on p. 5).

Obermaisser, Roman. *Event-Triggered and Time-Triggered Control Paradigms*. 1st ed. Vol. 22. Real-Time Systems Series. Springer US, 2005 (cit. on p. 95).

— *Time-Triggered Communication*. CRC Press, 2011 (cit. on pp. 98, 99).

Odersky, Martin, Enno Runne, and Philip Wadler. “Two ways to bake your pizza—translating parameterised types into Java”. In: *Generic Programming*. Springer, 2000, pp. 114–132 (cit. on p. xxi).

Pedreiras, Paulo. “Supporting Flexible Real-Time Communication on Distributed Systems”. PhD thesis. University of Aveiro, 2003 (cit. on pp. 127–129, 141).

Pedreiras, Paulo and Luís Almeida. “The Flexible Time-Triggered (FTT) Paradigm: an Approach to QoS Management in Distributed Real-Time Systems”. In: *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2001, p. 9 (cit. on pp. 8, 124).

Pedreiras, Paulo, Luís Almeida, and P. Gai. “The FTT-Ethernet Protocol: Merging Flexibility, Timeliness and Efficiency”. In: *Proceedings of the 14th Euromicro Conference on Real-Time Systems* (2002), pp. 134–142 (cit. on p. 8).

Pedreiras, Paulo, P. Gai, Luís Almeida, and G.C. Buttazzo. “FTT-Ethernet: A Flexible Real-Time Communication Protocol that Supports Dynamic QoS Management on Ethernet-Based Systems”. In: *IEEE Transactions on Industrial Informatics* 1.3 (Aug. 2005), pp. 162–172 (cit. on pp. 130, 132).

Perlman, R., D. Dutt, S. Gai, and A. Ghanwani. *Routing Bridges (RBridges): Base Protocol Specification*. 6325 (RFC). 2011 (cit. on p. 107).

Perlman, Radia. “An Algorithm for Distributed Computation of a Spanning tree in an Extended LAN”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 15. 4. ACM. 1985, pp. 44–53 (cit. on pp. xxi, 117).

— *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Pearson Education, 2000 (cit. on p. 113).

Pham, Hoang. *System Software Reliability*. Springer Science & Business Media, 2007 (cit. on p. 242).

Pinker, Steven. *The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century*. Penguin Books, 2014 (cit. on p. xvi).

Pinker, Steven, Michael C. Munger, Helen Sword, Rachel Toor, and Theresa MacPhail. *Why Academics Stink at Writing—and How to Fix It*. The Chronicle of Higher Education, Sept. 2014 (cit. on p. xvii).

Poledna, Stefan. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996 (cit. on pp. 44, 48, 49, 56, 60, 74–81, 90).

— "Method and Switching Unit for the Reliable Switching of Synchronization of Messages". US Patent App. 14/391162. Mar. 2015 (cit. on pp. 171, 269).

Potts, Chris N. and Vitaly A. Strusevich. "Fifty Years of Scheduling: a Survey of Milestones". In: *Journal of the Operational Research Society* 60.1 (2009) (cit. on p. 96).

Powell, David, ed. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer, 1991 (cit. on pp. 19, 82, 84–86).

— "Distributed Fault Tolerance: Lessons from Delta-4". In: *IEEE Micro* 14.1 (1994), pp. 36–47 (cit. on pp. 87, 171).

— "Failure Mode Assumptions and Assumption Coverage". In: *Predictably Dependable Computing Systems*. Springer, 1995, pp. 123–140 (cit. on pp. 57–59).

Proenza, Julián. "RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance". PhD thesis. Universitat de les Illes Balears, 2007 (cit. on pp. 48, 54, 63, 64, 67, 78, 90, 171).

Proenza, Julián, Manuel Barranco, Joan Llodra, and Luís Almeida. "Using FTT and Stars to Simplify Node Replication in CAN-based Systems". In: *Proceedings of the 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2012)*. IEEE, Sept. 2012 (cit. on p. 144).

Proenza, Julián and José Miro-Julia. "MajorCAN: A Modification to the Controller Area Network Protocol to Achieve Atomic Broadcast". In: *IEEE International Workshop on Group Communication and Computations*. Taipei, Taiwan, 2000 (cit. on p. 142).

Pullum, Laura L. *Software Fault Tolerance Techniques and Implementation*. Artech House, 2001 (cit. on pp. 56, 57).

Ramakrishnan, K. K. and Henry Yang. "The Ethernet Capture Effect: Analysis and Solution". In: *Proceedings of the 19th IEEE Conference on Local Computer Networks (LCN 1994)*. IEEE. 1994, pp. 228–240 (cit. on p. 133).

Ramasamy, HariGovind V., Adnan Agbaria, and William H. Sanders. *Semi-Passive Replication in the Presence of Byzantine Faults*. Tech. rep. 2004 (cit. on p. 89).

Reis, George A, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. “SWIFT: Software Implemented Fault Tolerance”. In: *Proceedings of the 3rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2005)*. IEEE Computer Society. 2005, pp. 243–254 (cit. on p. 56).

Reisinger, Johannes and Andreas Steininger. “The Design of a Fail-Silent Processing Node for the Predictable Hard Real-Time System MARS”. In: *Distributed Systems Engineering* 1.2 (1993), p. 104 (cit. on p. 171).

El-Rewini, Hesham and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. Vol. 42. John Wiley & Sons, 2005 (cit. on p. 46).

Robert Bosch GmbH. *CAN Specification Version 2.0*. Tech. rep. Robert Bosch GmbH, 1991 (cit. on pp. 130, 140).

Rodrıguez-Navas, Guillermo. “Design and Formal Verification of a Fault-Tolerant Clock Synchronization Subsystem for the Controller Area Network”. PhD thesis. Universitat de les Illes Balears, 2010 (cit. on pp. 93, 237).

Ross, Sheldon M. *Introduction to Probability Models*. 9th. Elsevier, 2007 (cit. on pp. 331, 332).

Rufino, Jose, P. Veríssimo, Guilherme Arroz, Carlos Almeida, and L. Rodrigues. “Fault-Tolerant Broadcasts in CAN”. In: *Proceedings of the 28th IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS 1998)*. Washington, DC, USA: IEEE Computer Society, 1998, p. 150 (cit. on p. 142).

Rushby, John. “Bus Architectures for Safety-Critical Embedded Systems”. In: *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT 2001)*. (Tahoe City, CA, USA). Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Lecture Notes in Computer Science. Springer. 2001, pp. 306–323 (cit. on p. 134).

Santos, R., R. Marau, A. Vieira, P. Pedreiras, A. Oliveira, and Luis Almeida. “A Synthesizable Ethernet Switch with Enhanced Real-Time Features”. In: *Proceedings of the 35th Annual Conference of the IEEE Industrial Electronics Society (IECON 2009)*. IEEE. 2009, pp. 2817–2824 (cit. on pp. 8, 138, 243, 269).

Santos, Rui Gabriel Viegas dos. “Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications”. PhD thesis. Universidade de Aveiro, 2010 (cit. on pp. 138, 139, 167, 235).

Santos, Rui, Ricardo Marau, Arnaldo Oliveira, Paulo Pedreiras, and Luis Almeida. “Designing a Costumized [sic] Ethernet Switch for Safe Hard Real-Time Communication”. In: *Proceedings of the 7th IEEE International Workshop on Factory Communication Systems (WFCS 2008)*. IEEE. 2008, pp. 169–177 (cit. on pp. 138, 310).

Schneider, Fred B. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319 (cit. on p. 84).

Severance, Charles. “Bob Metcalfe: Ethernet at Forty”. In: *Computer* 46.5 (2013), pp. 6–9 (cit. on pp. 102, 103).

Shelton, D. Cragin. “Writing in the First Person for Academic and Research Publication”. In: *Proceedings of the EDSIG Conference*. 2015 (cit. on p. xix).

Silva, Valter Filipe da. “Flexible Redundancy and Bandwidth Management in Fieldbuses”. PhD thesis. Universidade de Aveiro, 2010 (cit. on pp. 141, 143).

Silva, Valter Filipe da, Joaquim Ferreira, and José Alberto Fonseca. “Master Replication and Bus Error Detection in FTT-CAN with Multiple Buses”. In: *Proceedings of the 12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2007)*. IEEE. 2007, pp. 1107–1114 (cit. on pp. 141, 143).

Silva, Valter Filipe, José Alberto Fonseca, and Joaquim Ferreira. “Adapting the FTT-CAN Master for multiple-bus operation”. In: *Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN 2007)*. Vol. 1. IEEE. 2007, pp. 305–310 (cit. on p. 141).

Society of Automotive Engineers (SAE). *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*. SAE J3016 (SAE). 2016 (cit. on p. 5).

— *Time-Triggered Ethernet*. SAE AS 6802 (SAE). 2011 (cit. on pp. 108, 110).

Software Engineering Radio. *Episode 203: Leslie Lamport on Distributed Systems*. IEEE Computer Society. Apr. 2014 (cit. on p. 307).

Spurgeon, Charles E. *Ethernet: The Definitive Guide*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2000 (cit. on pp. 104, 113).

Sword, Helen. *Stylish Academic Writing*. Harvard University Press, 2012 (cit. on pp. xix, xx).

Systems, Robotics, and Vision group (UIB). *Webpage of the DFT4FTT project*. 2017. URL: http://srv.uib.es/dft4ftt/ (visited on 2017-02-27) (cit. on p. xxiv).

— *Webpage of the FT4FTT project*. 2016. URL: http://srv.uib.es/ft4ftt/ (visited on 2016-11-12) (cit. on p. xxiv).

Takagi, Hideaki and Leonard Kleinrock. “Throughput analysis for persistent CSMA systems”. In: *IEEE Transactions on Communications* 33.7 (1985), pp. 627–638 (cit. on p. 115).

Tasaka, Shuji. “Dynamic behavior of a CSMA-CD system with a finite population of buffered users”. In: *IEEE Transactions on Communications* 34.6 (1986), pp. 576–586 (cit. on p. 115).

Thomas, Francis-Noël and Mark Turner. *Clear and simple as the truth: Writing classic prose*. Princeton University Press, 2011 (cit. on p. xvi).

Tobagi, Fouad A. and V. Bruce Hunt. "Performance analysis of carrier sense multiple access with collision detection". In: *Computer Networks (1976)* 4.5 (1980), pp. 245–259 (cit. on p. 115).

Turing, Alan M. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 42 (1936), pp. 230–265 (cit. on p. xx).

Urbán, Péter, Naohiro Hayashibara, André Schiper, and Takuya Katayama. "Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pp. 4–17 (cit. on p. 88).

Varghese, George. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. The Morgan Kaufmann Series in Networking. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004 (cit. on pp. 115, 116).

Veríssimo, Paulo. *Comunicação e Computação em Sistemas Distribuídos - Lição de Síntese*. URL: http://www.navigators.di.fc.ul.pt/docs/abstracts/ccsd-aula.html (visited on 2016-12-10) (cit. on p. 90).

Voss, Wilfried. *A Comprehensible Guide to Controller Area Network*. Copperhill Media, 2008 (cit. on p. 140).

Wadler, Philip. "Et tu, XML? The downfall of the relational empire". In: *Proceedings of the 27th Very Large Data Bases Conference (VLDB 2001)*. 2001, p. 15 (cit. on p. xxi).

Wadler, Philip and Robert Bruce Findler. "Well-typed programs can't be blamed". In: *European Symposium on Programming*. Springer. 2009, pp. 1–16 (cit. on p. xxi).

Wisniewski, Lukasz, Mohsin Hameed, Sebastian Schriegel, and Juergen Jasperneite. "A Survey of Ethernet Redundancy Methods for Real-Time Ethernet Networks and its Possible Improvements". In: *Fieldbuses and Networks in Industrial and Embedded Systems*. Vol. 8. 1. 2009, pp. 163–170 (cit. on p. 68).

Yekeh, Farahnaz, Mostafa Pordel, Luís Almeida, Moris Behnam, and Paulo Portugal. "Exploring alternatives to scale FTT-SE to large networks". In: *Proceedings of the 6th IEEE International Symposium on Industrial and Embedded Systems (SIES 2011)*. IEEE. 2011, pp. 107–110 (cit. on p. 134).

Index