A Fault-Tolerant Ethernet for Hard Real-Time Adaptive Systems

David Gessner, Julián Proenza, *Senior Member, IEEE*, Manuel Barranco and Alberto Ballesteros

***Abstract*—Distributed embedded systems (DESs) that perform critical tasks in unpredictable environments must be reliable, hard real-time, and adaptive. Since a DES comprises nodes that rely on a network, the network must provide adequate support: it must be reliable, convey messages on time, and meet new real-time requirements as the nodes adapt. Ethernet is ill-suited for such hard real-time adaptive systems, but it can be made suitable. The Flexible Time-Triggered (FTT) paradigm already supports hard real-time message exchanges and the necessary flexibility to meet evolving hard real-time requirements, but its Ethernet implementations had reliability limitations. To address these, we designed FTTRS, a communication subsystem that tolerates permanent and transient faults, even if they occur simultaneously, while keeping the paradigm's key features: support for both the timely exchange of periodic and sporadic real-time messages, and support for updating the real-time parameters of these messages at runtime. In this paper we present FTTRS, the first Ethernet-based communication subsystem specifically designed for highly reliable hard real-time adaptive DESs.**

***Keywords*—Ethernet, fault tolerance, hard real-time systems, adaptive systems, distributed embedded systems, dependable systems**

I. Introduction

During the 2011 nuclear disaster in Fukushima, mobile rescue robots were deployed, but only after significant delays. Valuable time was lost because the robots were not sufficiently reliable and adaptive: their "hardware reliability, communication systems, and basic sensors were considered inadequate" and first had to be "retrofitted for the disaster response missions" [1]. Had the robots been more reliable and adaptive, they could have been deployed earlier.

Rescue robots are just one application where reliable, hard real-time, and adaptive distributed embedded systems (DESs) are advantageous. Others include self-driving cars, smart cities, smart grids, and in general any application where a computerized system needs to provide an uninterrupted service while interacting with an unpredictable physical environment.

Regardless of the application, a DES has a set of computing nodes that exchange messages through a network. Thus, for a DES to be reliable, hard real-time, and adaptive, it needs support from its network. The Flexible Time-Triggered (FTT) paradigm [2] can provide the necessary support. It already provided hard real-time communication with the necessary operational flexibility, i.e., with the ability to change the real-time requirements and reconfigure the traffic without interrupting the communication services, for adaptive applications. However, its Ethernet implementations lacked the necessary reliability. The *Flexible Time-Triggered Replicated Star for Ethernet* (FTTRS) addresses this limitation by adding tolerance to both permanent and transient faults, without sacrificing FTT's support for hard real-time communication and flexibility to meet evolving hard real-time requirements.

D. Gessner, J. Proenza, M. Barranco and A. Ballesteros are with the Systems, Robotics and Vision group (SRV), Departament de Matematiques i Informatica, Universitat de les Illes Balears (UIB), 07122, Palma de Mallorca, Spain (e-mail: david.gessner@uib.es; julian.proenza@uib.es; manuel.barranco@uib.es; a.ballesteros@uib.es)

In particular, FTTRS **(S1)** consistently transports 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. The composite of S1, S2, and S3 is what we call an *FTT service*. Moreover, FTTRS provides an FTT service in the presence of faults. For this, it meets three requirements: **(R1)** the failure of a slave does not disrupt the FTT service provided to other slaves; **(R2)** the system is parameterizable to be able to tune it to tolerate transient faults of arbitrary maximum duration; and **(R3)** it provides an FTT service even if any arbitrary component suffers a permanent fault.

This paper for the first time shows how the different FTTRS mechanisms [3]–[5] are put together to fulfill requirements R1, R2, and R3, and argues for the feasibility of FTTRS. Also, it highlights that FTTRS adds some degree of dynamic fault tolerance and that it preserves additional flexibility of FTT.

II. Basic Concepts and Terminology

A system is *reliable* if it has a high probability of providing a continuous correct service. It is *hard real-time* if it has to meet at least one *hard deadline*—a deadline that, if missed, may lead to system failure. A *failure* is the transition of a system's service from correct to incorrect. The part of a system's state that may lead to a failure is an *error*. The cause of an error is a *fault*. A *fault model* is a description of the types of faults we assume to act upon the system. *Permanent faults* are those that remain unless repaired; *transient faults* are those that disappear on their own. Faults, errors, and failures are causally linked: a fault causes an error, which, in turn, causes a failure [6].

Fault tolerance are means aimed at interrupting the causal link such that, despite faults, the last event—the failure—does not occur. It relies on redundancy. One type of redundancy is *hardware replication*, where two or more systems, called *hardware replicas*, are available for the same service. For this, however, the replicas must be *replica deterministic*: they must show correspondence in their outputs (e.g., transmit equivalent messages) given that they started in the same initial state and processed corresponding inputs within a given time interval.

This requires *replica determinism enforcement*, i.e., mechanisms that ensure replica determinism. Hardware replication, together with the necessary replica determinism enforcement, is the only way to overcome *single points of failure*—components whose failure inevitably leads to a global system failure. In hardware replication, replicas need not be equivalent. For instance, in *semi-active replication* all replicas process the same inputs and provide the same responses, but one of them, the *leader*, plays a privileged role with respect to the others, the *followers*.

The *failure semantics* of a system are the ways in which it can behave—or misbehave—after a failure and before *failure recovery*, i.e., before correct service is restored. The hardest failure semantics to deal with are *byzantine failure semantics*, which include all possible misbehaviors, even such tricky ones as two-faced behaviors and impersonations. More benign are *fail-silent failure semantics*, where a system remains silent once it fails. Since the difficulty of tolerating the failure of system components depends on their failure semantics, fault-tolerant systems often restrict failure semantics. Two ways of doing so are *guardians* (components that limit the misbehaviors of other components) and *internal duplication with comparison*.

III. Related Work

Current Ethernet-based protocols lack one or more of the following: reliable message delivery, support for hard real-time communication, or the flexibility to meet evolving hard real-time requirements.

For instance, the standard Ethernet protocol, and most subsequent non-industrial ones based on it, only provide best-effort delivery and cannot meet real-time requirements due to their nondeterminism. This is particularly true for protocols based on the original, now mostly obsolete, CSMA/CD randomized retransmission algorithm, whereas the modern full-duplex switched Ethernet may delay messages unpredictably because of queuing at switches and end nodes.

A class of switched Ethernet-based protocols add some reliability by tolerating permanent faults—such as STP and RSTP [7], TRILL [8], and SPB [7]—but they are still insufficient for hard real-time applications. Not only do they retain the above-mentioned nondeterminism, but they add new impediments: they have significant failure recovery times (from 100's of ms up to minutes [9]) and do not prevent the loss of messages during recovery. A network using such protocols can only tolerate transient faults with an additional transport layer that adds its own, typically unbounded, delays.

Another class of Ethernet-based protocols are suitable for real-time applications and have some fault tolerance. Several older ones are described in [10], namely NDDS, ORTE, RETHER, RT-EP, the MARS bus, and EtheReal. Some of the newer ones, like MRP [11], HSR [12] and PRP [12], can even provide seamless fault tolerance, thus zero failure recovery times. The Ethernet-based redundancy mechanisms use a ring topology or two physically independent networks. For instance, HSR [12] daisy chains nodes into a ring in which frames are duplicated and then transmitted in opposite directions along the ring; PRP [12] and AFDX [13] use two independent Ethernet networks simultaneously; and TTEthernet [14] can provide up to three independent communication channels. Moreover, the recent TSN standards provide seamless redundancy through multipathing [15]. When compared with the use of a ring topology or independent networks, multipathing increases the number of redundant paths that can be obtained (and thus the number of faults that can be tolerated) with a given investment in extra switches and cabling.

These particular protocols have overcome Ethernet's nondeterminism in various ways. AFDX uses rate-constrained transmission, i.e., a transmitting node has to wait a certain amount of time, called the bandwidth allocation gap (BAG), between successive transmissions through a given AFDX virtual link. This constrains the rate with which messages are transmitted and prevents excessive queuing in switches to provide hard real-time guarantees. TTEthernet [14], in contrast, overcomes Ethernet's nondeterminism with time-triggered messages that are transmitted according to a pre-calculated cyclic executive schedule and special Ethernet switches that prioritize these messages over other non-real-time critical ones. Basically, TSN overcomes Ethernet's nondeterminism in a similar way as TTEthernet does [16]. Time-triggered hard real-time traffic is achieved by configuring nodes and switches according to a cyclic executive schedule. The schedule divides the communication into cycles, which in turn are split into different windows to isolate the time-triggered traffic from the soft real-time asynchronous one. Each node transmits its time-triggered traffic at the calculated instants of time; whereas each switch uses a set of *gates* to enable/disable the forwarding from its different queues, so as to shape the traffic according to the schedule [16].

Nevertheless, all these protocols lack the necessary flexibility regarding the real-time requirements to change in unforeseen ways. AFDX provides seamless fault tolerance and meets hard deadlines, but only when the real-time requirements remain static because the calculated BAGs remain fixed at runtime. Similarly, in TTEthernet the schedule for the time-triggered messages cannot be changed later on without halting the network.

In regard to TSN's flexibility, the IEEE Std 802.1Qcc [17] standard includes mechanisms to ask for changes in the real-time requirements at runtime to reconfigure nodes and switches according to a new cyclic executive schedule. However, it does not explicitly provide any specific mechanism to actually re-calculate the schedule, since this aspect is out of scope of TSN. Some recent works aim at providing this re-scheduling capability. For instance, [18] proposes a set of re-scheduling algorithms for the *fully centralized* approach of Qcc. In this approach the nodes ask a *centralized agent* for changes. In case the agent can provide a new schedule, the agent reconfigures the nodes and switches accordingly. As we will see later, this approach resembles the one of FTT, whose centralized master is in charge of re-scheduling and re-configuring the network at runtime.

Nevertheless, TSN presents flexibility limitations when compared with FTT. On the one hand note that TSN uses a cyclic executive scheduler, whereas FTT can use any scheduling policy. To calculate a cyclic executive schedule is known to be a NP-complete problem [19], and normally requires a significant

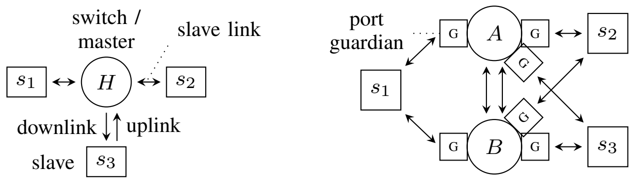


Figure 1: HaRTES versus FTTRS architecture.

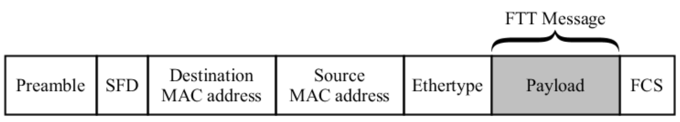


Figure 2: FTT message in an Ethernet frame.

amount of time. This directly reduces the flexibility of the system to timely reconfigure and react to critical situations when compared with FTT, in which it is even possible to reconfigure the traffic in each communication cycle. Research is currently being conducted to reduce the reschedule time in TSN. But even the most recent results [18] show that this time is considerably higher than the one needed to reschedule in FTT [20]. Moreover, although the time to react to changes could be reduced in TSN by switching among different pre-calculated modes (pre-calculated schedules of specific sets of messages), flexibility is still within the rigid margins of these modes. On the other hand, it is also noteworthy that no one of the flexibility mechanisms TSN provides is fault tolerant. In particular, the centralized agent is not replicated and thus constitutes a single point of failure. Moreover, TSN provides no time redundancy, since it does not include any message retransmission mechanism.

As to relevant research protocols from academia, we have Atacama [21] and the Ethernet versions of FTT.

Atacama uses time division multiple access (TDMA) arbitration with some flexibility to meet deadlines even when real-time requirements change at runtime. It does so by supporting dynamic switching between TDMA schedule alternatives [22]. However, if the real-time requirements change unpredictably, the foreseen schedule alternatives may not be appropriate. Moreover, it lacks the necessary fault tolerance.

As to the Ethernet versions of FTT, namely FTT-Ethernet [23], FTT Switched Ethernet [24], and HaRTES [25], although they can meet hard real-time requirements and have the necessary flexibility, they cannot guarantee the delivery of messages in the presence of faults.

Thus, neither of these Ethernet-based protocols, nor any other we are aware of, can currently be used to fulfill the requirements of Section Section I.

IV. The FTT Paradigm and HaRTES

FTT is a master to multi-slave approach: a master, by means of a single message, called *trigger message* (TM), polls multiple slave nodes at once. Each slave, in turn, responds by transmitting the requested messages for which it is the publisher (or producer, depending on the implementation of FTT) and which are intended for other slaves, which are the subscribers (or consumers). The polling by the TM occurs at fixed time intervals that divide the communication time into communication cycles called *Elementary Cycles* (ECs).

In each EC part of the bandwidth is allocated to periodic real-time messages, called *synchronous messages*. The rest of the EC is allocated to the referred to as *asynchronous messages*. These later ones include both real-time messages that are sporadic (aperiodic real-time messages with a minimum inter-arrival time); and non-real-time messages, which can be either sporadic or purely aperiodic. Synchronous messages are always polled by the TM, whereas asynchronous messages may or may not, depending on the implementation. The support for hard real-time communication comes from the master polling the messages according to its internal scheduler. This scheduler calculates the real-time schedule every communication cycle (each EC) based on the contents of a *System Requirements Database* (SRDB) that stores for all messages their real-time parameters (such as deadlines, periods, minimum interarrival times, etc.). The flexibility comes from the fact that nodes may request changes to the SRDB. Changes are only accepted by the master if the resulting set of all messages remains schedulable. If that occurs, the slaves are notified, and these then update their *Node Requirements Databases* (NRDBs), which are the counterpart in each slave to the SRDB.

FTTRS builds on top of a switched-Ethernet FTT implementation called *Hard Real-Time Ethernet Switching* (HaRTES) [25]. As shown on the left of Figure 1, HaRTES implements a simplex (non-replicated) microsegmented star topology, with the HaRTES switch (H in the figure) as a central element that provides the most relevant functions of FTT. In particular, the switch embeds the FTT master and thus not only forwards messages, but also stores the SRDB, computes the EC schedules, periodically broadcasts the TM, and accepts or rejects requests from any slave s_i to update the SRDB. Links are full-duplex, which enables simultaneous bidirectional communications in each link. Thus, we treat each slave link as if it were comprised of an uplink and a downlink, as shown in Figure 1. Moreover, the HaRTES switch already restricts the failure semantics of slaves to some extent. It does so by integrating validation units that discard unscheduled synchronous messages and asynchronous messages that violate their minimum interarrival time. As it will be explained in Section Section V-C), FTTRS extends the error-containment capabilities of these units in what we call *port guardians* (labeled G in Figure 1).

We chose HaRTES as our starting point because of its higher potential for high reliability. Not only because it additionally includes the just-mentioned validation units, but also because, contrary to FTT-Ethernet [23] and FTT Switched Ethernet [24], HaRTES only has one single point of failure (the switch). FTT-Ethernet and FTT-SE, by contrast, have two each: the master node and the Ethernet bus or hub in case of FTT-Ethernet, and the master node and Ethernet switch in case of FTT-SE.

HaRTES uses a regular Ethernet frame to convey each FTT message, as it is shown in Figure 2.

In HaRTES, as in all versions of FTT, slaves send their

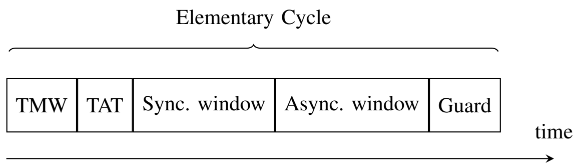


Figure 3: Elementary cycle as seen on the downlink of a slave.

synchronous messages initiated by the TM. Asynchronous messages, in contrast, are sent by the slaves whenever they become ready. Nevertheless, they do not interfere with the broadcast of the TM and are not interleaved with synchronous messages. This is so because the HaRTES switch shapes the traffic and ensures that the traffic it forwards in each EC is confined into windows. As shown in Figure 3, on the downlinks there is one window for transmitting the TM (TMW); a turn-around-time (TAT) that gives the slaves time to process the TM; a synchronous window where the switch forwards the synchronous messages polled by the TM; an asynchronous window where the switch forwards asynchronous messages; and a guard window that ensures that an asynchronous message does not overrun into the next EC, which could delay the next EC's TM. HaRTES follows a publisher/subscriber communication model and thus forwarding is not based on MAC addresses as in standard Ethernet, but on who the subscribers of a particular message are. The switch checks the FTT-related information in the payload of each Ethernet frame to identify the FTT message, and the SRDB in order to learn who are the subscribers of that message.

V. Design of FTTRS

A. Fault Model

Following the taxonomy established by Avižienis *et al.* [6], our fault model includes non-malicious operational hardware faults, which 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 are examples of these faults. The fault model excludes development, software, and malicious faults. We exclude for instance manufacturing defects, bugs, and intrusion attempts. As to the rate with which faults occur, we do not make any assumptions except that transient faults affecting links shall be detectable by the Ethernet Frame Check Sequence (FCS) shown in Figure 2 and, to properly parameterize FTTRS, have a known maximum duration.

B. The Architecture of FTTRS

FTTRS eliminates the single point of failure that the switch in HaRTES constitutes by using a duplicated architecture, as shown on the right of Figure 1. Specifically, FTTRS has a replicated star topology with two switches A and B in the center, each of them embedding an FTT master, just like in HaRTES. Although duplication increases the hardware realization costs with respect to HaRTES, it is the minimum replication needed to eliminate the single point of failure that the switch constitutes in HaRTES. The FTTRS switches are also interconnected using interlinks so that they can communicate with each other and resolve any inconsistencies that might prevent them from being replica deterministic, i.e., from providing a consistent FTT service towards the slaves. Also, to prevent a network partition in case of an interlink failure, the interlinks are replicated. Finally, each slave s_i is connected to each switch by means of a separate Ethernet link, each called a slave link and each subdivided into an uplink and downlink.

C. Failure Semantics of FTTRS Components

To make it easier to meet the requirements (Section I), FTTRS restricts the failure semantics of switches and slaves. Each switch is made fail silent through internal duplication with comparison. Although costly, for switches that need to be custom-made anyway, this is an acceptable trade-off to make the system fault tolerant. Moreover, internal duplication with comparison is a well-established mechanism to restrict the failure semantics of components.

The failure semantics of the slaves are mitigated by means of fail-silent port guardians located at the switch-side of each slave link (labeled G in Figure 1). These, through direct access to the SRDB of the corresponding embedded master, and like the original validation units in HaRTES, drop all incoming slave messages that violate the real-time communication requirements. Contrary to the HaRTES validation units, however, the FTTRS port guardians also prevent impersonations and two-faced behaviors. Impersonations are prevented because each slave has a statically assigned source identifier that the slave includes in each message it sends, as part of the FTT-related information within the payload of the corresponding Ethernet frame (see Figure 2). The port guardians simply drop any FTT message that does not include the source identifier assigned to that port. As to two-faced behaviors, Figure 4 illustrates how these could in principle occur within a slave: a payload x originates from a sensor or local computation; is stored in memory; and is encapsulated in an FTT message $\mathrm{m}(x)$, which is again stored in memory and then corrupted into $\mathrm{m}(x)'$ in one of the disjoint paths from the memory to the Ethernet controllers. Because the corruption occurs before the Ethernet controllers compute the Ethernet FCS, it is not detected. The solution is illustrated in Figure 5: a CRC of the message $\mathrm{m}(x)$ is computed within a slave before $\mathrm{m}(x)$ follows disjoint paths towards the Ethernet controllers (note that this CRC is a new one, different to FCS since it only considers the FTT-related information and not all the bits of the Ethernet frame); the CRC, together with the message, is handed over to the Ethernet controllers, which use it as the payload for an Ethernet frame eth (see Figure 2); and finally each guardian only allows the incoming frame to pass if it has a CRC matching the encapsulated FTT message. Thus, in case a slave sends a message $\mathrm{m}(x)$ to one switch and a message $\mathrm{m}(x)'$ to the other, only the correct message will be accepted by the receiving switch (and forwarded to the other switch via the interlinks), whereas the wrong message will be dropped once the non-matching CRC is detected by the port guardian. Thereby the two-faced behaviour is not perceived by the rest of the nodes. Finally, the guardians also check the

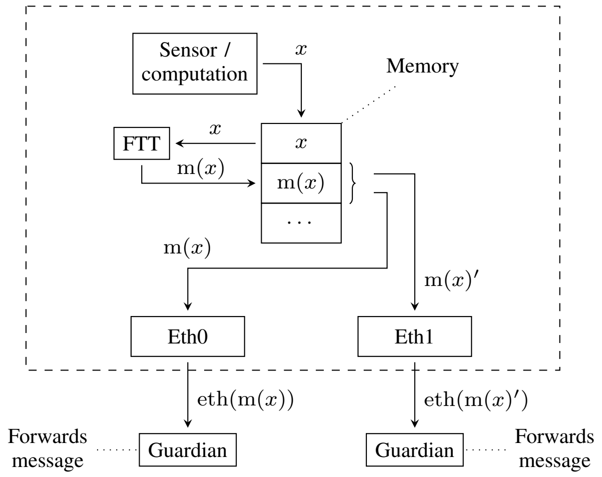


Figure 4: The potential for two-faced behaviors.

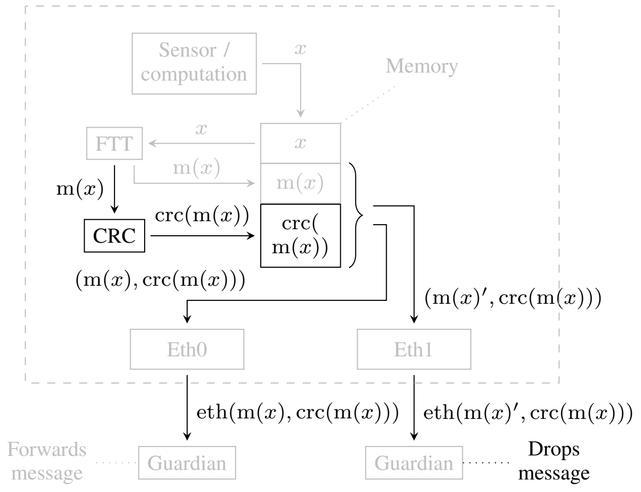


Figure 5: The CRC mechanism to prevent two-faced behaviors.

metadata of all incoming slave messages to verify that it is consistent with what is specified in the SRDBs.

D. Fault-Tolerance Mechanisms

To meet the requirements (Section I), FTTRS employs fault-tolerance mechanisms that exploit the restricted failure semantics and the duplicated architecture.

1) R1—Preventing Faulty Slaves from Disrupting the FTT Service: R1 is satisfied thanks to the port guardians. Specifically, the port guardians mitigate 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. Specifically, each guardian only lets FTT messages pass that are timely and have correct metadata (e.g., a valid value for the FTT message type and a correct source identifier). 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 the latter to lose legitimate frames. Moreover, given our fault model, which excludes development and software faults, frames sent by a faulty slave cannot crash a switch nor another slave or otherwise cause the failure of a switch or slave.

2) R2—Tolerating Transient Faults: Transient faults may occur in slaves, switches, or links. In slaves they must only be prevented from interfering with the communication (R1), but do not need to be tolerated by FTTRS since tolerating them is out of the scope of a communication subsystem. In any case, they can be tolerated at the application level through node redundancy. 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 its decisions compared. FTTRS therefore only has to deal with transient faults in links. These can only manifest as message omissions due to the Ethernet FCS, the port guardians, and the fail-silent switches. To tolerate these omissions, each message has a parameter called *redundancy level*, which specifies for a message how many copies of it the corresponding transmitter should send in a given EC through each link. Thus, a redundancy level of $k = 3$ for the TM means that each switch broadcasts three TM replicas in each EC, 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 tells that slave to transmit three replicas of the message within the same EC through each of its two links. FTTRS thus adopts a proactive retransmission approach where the number of retransmissions is specified by a redundancy level. This ensures that it meets requirement R2 even though the parameterization is not in terms of the maximum duration of transient faults. Once it is decided which is the maximum duration of transient faults that the system has to be capable of tolerating and the maximum rate with which frames can be transmitted, it can be calculated how many consecutive frames may be corrupted at most and choose a redundancy level higher than the calculated number. The redundancy level for slave messages can be changed at runtime, just like the real-time parameters, which adds some dynamic fault tolerance.

The proactive retransmissions consume a significant amount of bandwidth when compared to HaRTES. Nevertheless, retransmissions are unavoidable if we want to tolerate simultaneous transient and permanent faults without incurring the cost of higher levels of replication than duplication. Compared to acknowledgment-based retransmissions, proactive retransmissions have two key advantages. First, they enable simple fault masking instead of more complex error detection and recovery. Second, they reduce the worst-case latency to get a message to its destination. After all, by proactively retransmitting we avoid the time penalty of timeouts or acknowledgment messages that are otherwise necessary. Thus, proactive retransmissions enable the scheduling of more FTT messages per EC since the masters, to avoid messages missing their deadline, must in any case assume that the maximum (bounded) number of messages are lost per EC when they compute EC schedules.

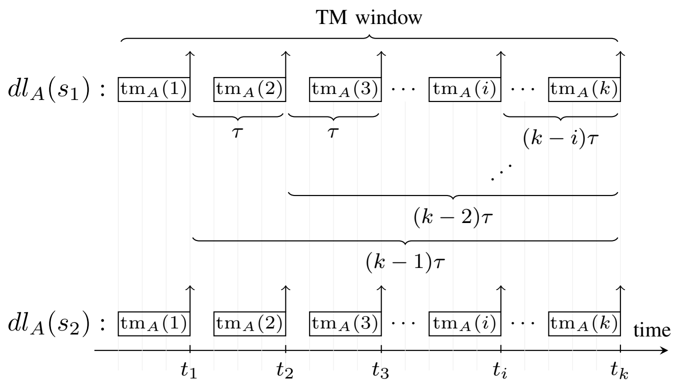


Figure 6: The EC synchronization.

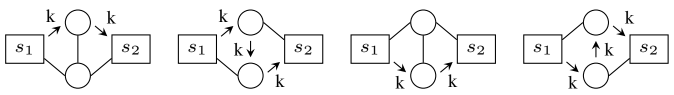


Figure 7: Replica radiation due to 4 redundant paths

Having met requirement R2 through proactive retransmissions raised a new problem. In HaRTES, and all other versions of FTT, slaves decide that a new EC begins as soon as they receive a TM. If there is only one TM per EC, and it is broadcast in parallel through all links, then all slaves receive the TM at approximately the same time and achieve a precise EC synchronization (assuming all links have the same propagation delay and bit rate). But FTTRS transmits k TMs per elementary cycle. Thus, to continue to provide a precise EC synchronization, FTTRS revises how slaves decide when an EC starts. The approach is to synchronize all slaves with the arrival of the last TM in each EC. And to make this work even if a subset of TMs, including the last one, is lost due to transient faults, the switches broadcast the TMs in such a way that the arrival time of the last TM can be predicted by a slave as long as it receives at least one uncorrupted TM. Specifically, the switches broadcast the TMs isochronously (with a fixed intertransmission time τ) and in lockstep. Moreover, TMs have sequence numbers, which are reset at the beginning of each EC. That way, as shown in Figure 6, whenever a slave s_1 receives through a downlink $dl_A(s_1)$ the ith TM out of k from a switch A in an EC at an instant of time t_i, it knows that the last TM should arrive at time $t_k = t_i + (k-i)\tau$. Another slave s_2 will reach the same conclusion whenever it receives at least one out of the k TMs. Thus, upon the reception of the ith TM, a slave simply sets a timer to expire after $(k-i)\tau$ units of time to know when it should consider the turn-around time of the new EC to begin.

3) R3—Tolerating Permanent Faults: To meet R3, FTTRS must tolerate a permanent fault in any one of its components, no matter which one. Since slaves are not part of the communication subsystem, this means that FTTRS has to ensure that it tolerates any permanent fault occurring in an interlink, slave link, or switch. This it does because of several key features of FTTRS. First, because of its architecture (Figure 1), 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. 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 slave link. 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 times the k replicas of each message due to a phenomenon we call *replica radiation*. More specifically, as depicted in Figure 7, FTTRS provides 4 parallel redundant paths between any pair of slaves, e.g., s_1 and s_2. Thus, since each one of the k replicas of a message should traverse each one of these paths, s_2 receives up to $4k$ times the message sent by s_1.

Note that we preferred not to include within the design of FTTRS any specific mechanism for eliminating the redundant copies of the same message that the slaves receive in the absence of faults. Instead, we leave this mechanism to higher layers, because they may want to know how many copies of a message have been received, e.g. for detecting the need to increase the redundancy level of a particular message. However, this does not necessarily mean that the application itself has to implement or to be aware of this mechanism. Actually, in the final proof-of-concept prototype of FTTRS (see references [26] and [27] in Section VI), we implemented this mechanism in a layer between FTTRS and the application, thereby leaving the application agnostic with respect to the redundant copies. To better understand how said intermediate layer carries out its function, first note that all the copies of every edition of a given FTT message are identifiable by means of the same univocal *FTT message identifier*. Second, no more than one edition of a given FTT message can be scheduled per EC, and all the edition copies are transmitted in the same EC (each in a separated frame). Thus, it is relatively simple to detect and eliminate redundant copies. Basically, when a slave of the mentioned prototype receives a frame, the intermediate layer inspects the FTT message identifier; if the slave has already received a copy of that FTT message in the current EC, it simply discards that frame.

The third feature of FTTRS that contributes to tolerate any permanent fault occurring in an interlink, slave link, or switch is that switches are fail silent and links have inherently benign failure semantics—they can only lead to the omission of messages. None of these faults can prevent slaves from exchanging messages through remaining non-faulty links. Nevertheless, slaves simply being able to exchange messages is not enough to tolerate a permanent fault. It is also necessary for the switches to be replica deterministic to provide corresponding outputs. Otherwise, a slave could receive contradicting services through each of its links. For instance, a slave might receive inconsistent EC schedules through its links, inconsistent timing information (e.g., different EC durations), or one switch might tell it to update the real-time requirements in one way, while the other tells it to update the requirements in another.

4) Enforcing the Necessary Replica Determinism: This means ensuring that replicated components provide corresponding outputs. For FTTRS switches this means that a) they must forward frames in the same way; b) their port guardians, when they monitor the same slave, must consistently reject or accept frames from that slave; and c) their embedded masters must be replica deterministic.

Since the switching circuitry and guardians are internally deterministic (e.g., they do not use true random number generators), the first two items can be ensured if the last one is. This is so because for masters to be replica deterministic, they must have the same SRDB contents and they must provide consistent timing, i.e., agree in which window of which EC the system currently finds itself. And the SRDBs and timing are precisely what determine how frames are forwarded or rejected. An FTTRS switch, just like a HaRTES one, forwards frames, whether they carry synchronous or asynchronous messages, not according to Ethernet MAC addresses, but according to the EC window in which the system finds itself and according to the SRDB, which specifies who the subscribers of a given message are. Similarly, whether a port guardian rejects a frame or not is also determined by the contents of the corresponding SRDB and by when the frame arrives (in which window of which EC).

To achieve the replica determinism of the embedded masters we distinguish between the time and value domain.

In the time domain we ensure that the masters agree when each EC starts and that they broadcast their TMs in lockstep. For this we use a semi-active replication approach, where one master is designated the leader and the other the follower. This approach uses a two-phase mechanism [3]: at system startup the masters execute an initialization phase based on a two-way time transfer (similar to IEEE 1588) and from then onwards the follower maintains its ECs synchronized with the leader by means of a process we call *periodic lockstep resynchronization*. In the periodic lockstep resynchronization, no switch ever waits for the other. Instead, each starts broadcasting its trigger messages for a given EC when its internal clock tells it that a new EC has started. What does occur, however, is that the follower does slight readjustments of its own clock if it detects that its own transmissions are not sufficiently in lockstep with the leader. These readjustments occur in each and every EC and, thus, are always small. As a result, the lockstep synchronization does not introduce significant delays beyond the one required for transmitting k trigger messages isochronously. The periodic lockstep resynchronization ensures a fault tolerant synchronization because it is enough for the follower to receive one out of the k TMs that the leader broadcasts per EC. Finally, when one of the masters fails, the other continues unperturbed, broadcasting its TMs according to its own clock and thus continuing to provide the systemwide timing.

Note that due to the impossibility of perfect synchronization, one switch may receive an asynchronous message in one EC, while the other receives it in another. This is only a problem for asynchronous messages that are update requests, which we address below in the replica determinism enforcement in the value domain. For other asynchronous messages, agreement on which EC they belong to is not necessary to provide an FTT service and meet our requirements.

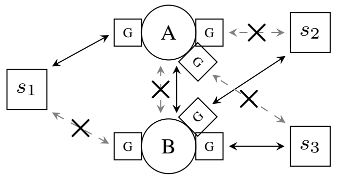
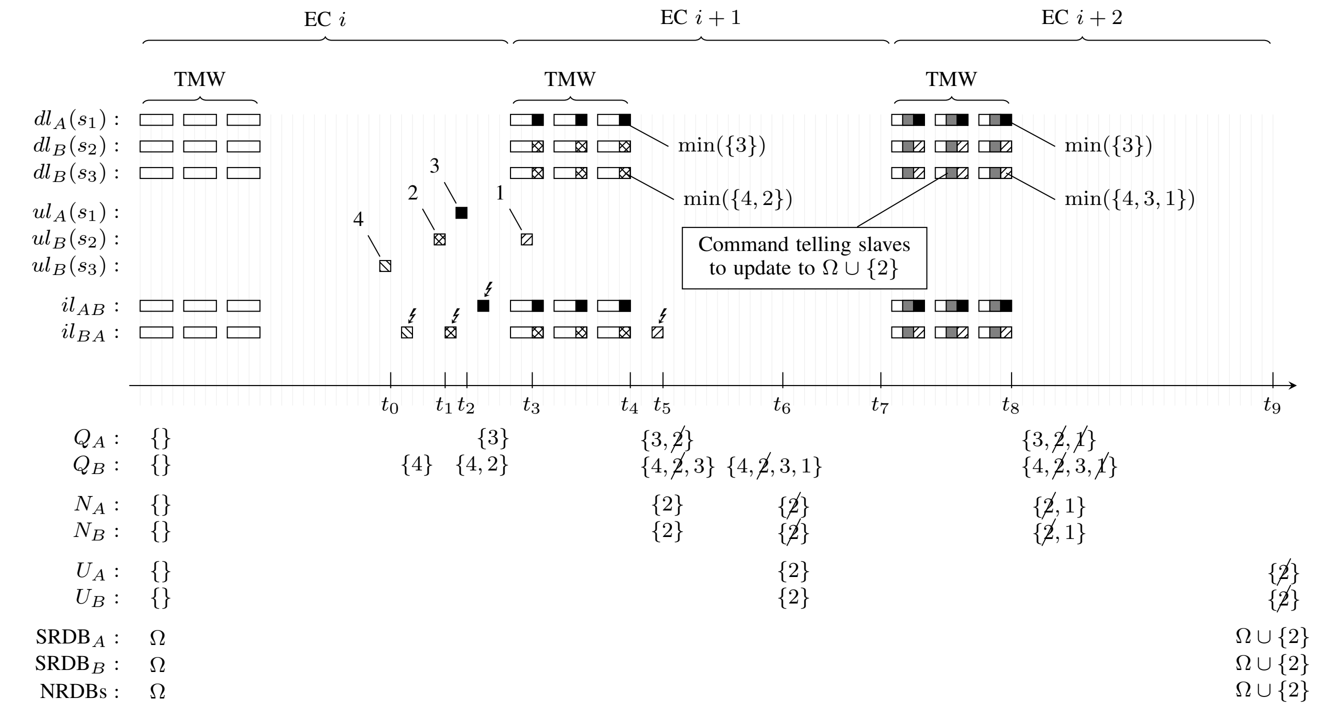


Figure 8: Still functioning FTTRS architecture after the loss of several links.

As to the replica determinism in the value domain, it comes down to ensuring three things: having the masters start out with consistent SRDBs, ensuring that the masters are internally deterministic by avoiding any non-deterministic logic, and ensuring that masters apply the same SRDB updates at the same time. The difficulty lies in the last. 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 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-tolerant mechanism for the embedded masters to agree on which update request to process next in case they did not receive the same requests.

We will illustrate this mechanism using the degraded FTTRS architecture of Figure 8, where several links have been lost (depicted as gray dashed arrows marked with a *cross*), but all slaves can still communicate. Specifically, Figure 9 shows a timeline of how, for the degraded architecture, an inconsistency among the masters is resolved (to keep the figure simple, it only shows TMs and update request messages). As to the notation, $dl_j(s_i)$ denotes the downlink from a switch $j \in \{A, B\}$ to a slave s_i and $ul_j(s_i)$ denotes the corresponding uplink in the opposite direction. The full-duplex interlink is denoted by il_{AB} for the direction from A to B, and il_{BA} for the reverse. The mechanism relies on the TMs, whose redundancy level k, contrary to the one for update requests and other messages, is always assumed to be high enough to tolerate transient faults of maximum duration. In Figure 9, $k = 3$ and thus there are three TMs in the TMWs of each EC. The mechanism relies on update requests from slaves being 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, i.e., a binary relation that is antisymmetric, transitive, and total. The relation can be implemented in different ways (one option is to order slaves by their identifiers and additionally use sequence numbers for update requests). Each master $j \in \{A, B\}$ keeps a set of pending update requests Q_j, which is a set containing all update requests it has received so far. In Figure 9, we assume that at the beginning of EC i there are no pending requests and thus initially $Q_A = Q_B = \{\}$. To keep the example simple, we assume that all update requests have a redundancy level of 1. Now, at time t_0, slave s_3 sends an update request with ordinal number 4 (according to the total order) through $ul_B(s_3)$,



EC i
EC $i+1$
EC $i+2$
TMW
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} :
4
2
3
1
$\min(\{3\})$
$\min(\{4,2\})$
$\min(\{3\})$
$\min(\{4,3,1\})$
Command telling slaves to update to $\Omega \cup \{2\}$
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
Q_A : $\{\}$ $\{3\}$ $\{3,\cancel{2}\}$ $\{3,\cancel{2},\cancel{1}\}$
Q_B : $\{\}$ $\{4\}$ $\{4,2\}$ $\{4,\cancel{2},3\}$ $\{4,\cancel{2},3,1\}$ $\{4,\cancel{2},3,\cancel{1}\}$
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 $\Omega \cup \{2\}$
SRDB$_B$: Ω $\Omega \cup \{2\}$
NRDBs : Ω $\Omega \cup \{2\}$

Figure 9: Example illustrating a consistent update of the system requirements across the whole system.

which due to a transient fault in il_{BA} only reaches B. Thus, shortly after t_0 we have $Q_B = \{4\}$, while Q_A remains empty (we use the ordinal numbers to identify requests and thus the number 4 in Q_B represents the request with that ordinal number). Similarly, at time t_1 slave s_2 sends an update request 2 through $ul_B(s_2)$, which also only reaches B; and at time t_2 slave s_1 sends a request 3 through $ul_A(s_1)$, which due to a fault in il_{AB} only reaches A. Thus, at the beginning of EC $i+1$, we have that $Q_A = \{3\}$, while $Q_B = \{4, 2\}$. To agree on which update request to process next in a given EC, at the start of that EC 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*. Thus, at the start of EC $i+1$, the local minimum of A is $\min(Q_A) = \min(\{3\}) = 3$ and the local minimum of B is $\min(Q_B) = \min(\{4, 2\}) = 2$. To not interfere with this process, update requests arriving during a TMW, like the one with ordinal number 1 at time t_3, are only forwarded and added to the pending requests after the TMW. In addition to choosing the local minimum, each master piggybacks it on the subsequent TMs it broadcasts. In Figure 9, this is shown on the downlinks during the TMW of EC $i+1$. At the end of that EC's TMW, each master has the other's local minimum, which it then adds to its set of pending requests. Thus, shortly after time t_4, we have that $Q_A = \{3, 2\}$ and $Q_B = \{4, 2, 3\}$. 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*. In Figure 9, both A and B select 2 as the global minimum, remove it from their set of pending requests, and assign it to N_A and N_B, which store for A and B, respectively, which update request to subject to admission control next. Once the global minimum is selected at the end of the synchronized TMW, both masters subject it to admission control. Both masters then conclude the admission control within the same EC because they started it at the beginning of the same EC and because it takes them approximately the same amount of time as they are internally deterministic replicas of each other. In Figure 9, the admission control is assumed to take less than one elementary cycle and to be completed at around time t_6. Edge cases where the admission control concludes just at the end of an EC can be avoided by forcing the admission control time to be extended with sufficient idle time, e.g., the time of half an EC. After completing the admission control, the masters, since they are internally deterministic and started out with consistent SRDBs, will reach the same conclusion on whether to accept or reject it. Once completed, both masters will have either accepted or rejected the request. In the figure, it is assumed to be accepted and thus at t_6 the request is eliminated from N_A and N_B and added to U_A and U_B, which store for A and B, respectively, the most recently accepted update request. Next, since both masters have accepted the same update request, 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

TMs and broadcast by the two masters during the next EC. Since the TMs are fault tolerant and broadcast in lockstep by the masters, it is ensured that all non-faulty slaves receive the same commands by the end of the TM window, which enables a systemwide update of the real-time requirements at the end of the corresponding EC. In Figure 9, the masters both piggyback in the TMs of EC $i+2$ the command to update the system requirements from Ω to $\Omega \cup \{2\}$, where Ω represents the previous system requirements—which were assumed to be consistent at the beginning of EC i and thus the SRDBs and NRDBs all contained that value. Through this mechanism, 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 is satisfied.

VI. Feasibility of FTTRS

To show that FTTRS is feasible, each of its mechanisms has to be proven and it must be possible to put them together.

Making FTTRS switches fail silent relies on internally duplicating them and comparing their results. This 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 proposed [28].

Similarly, the feasibility of mitigating the failure semantics of slaves by means of port guardians is already substantiated by prior work. Indeed, HaRTES already mitigates the failure semantics of slaves, although in the corresponding publication the guardians were called *validation units* [25]. FTTRS just extends the capabilities of these units.

As to the feasibility of the FTTRS architecture, we demonstrated it through proof-of-concept prototypes [5]. 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 of slave links and interlinks.

Making message transmission more reliable through proactive retransmissions is undoubtedly feasible. This is a technique that has already been used by other protocols, e.g., TTP [29], and we have implemented it ourselves for TMs [5].

The feasibility of the slave EC synchronization has been demonstrated by two proof-of-concept prototypes, one involving two slaves and a master within a virtualized switch, and one involving two slaves and a master communicating through a COTS switch [4]. The corresponding experiments showed that the slaves, as desired, synchronized their ECs (although with limited precision due to the nature of the proof-of-concept) and they further validated the correctness of the EC synchronization. Figure 10 shows a histogram of the measured absolute EC offset among two slaves in the experiment using a COTS switch. The mean measured EC offset among the two slaves was $0.69\,\mu s$, with a standard deviation of $1.36\,\mu s$, among $225\,000$ samples in which all possible TM loss combinations for a redundancy level of $k = 4$ TMs were injected repeatedly (1000 times per combination). The EC length was $1\,\mathrm{ms}$, the TMs fit within

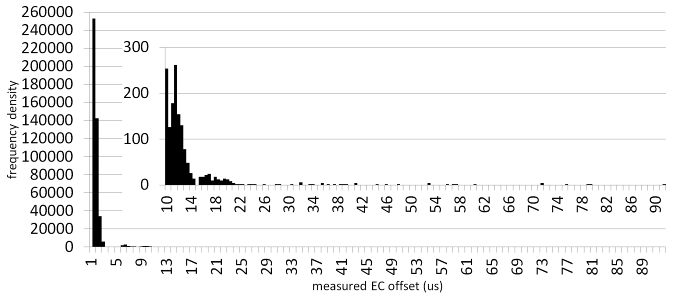


Figure 10: Results of the slave synchronization experiment. The bin size of the histogram is $0.5\,\mu s$. The inset shows a close-up of the right tail of the histogram.

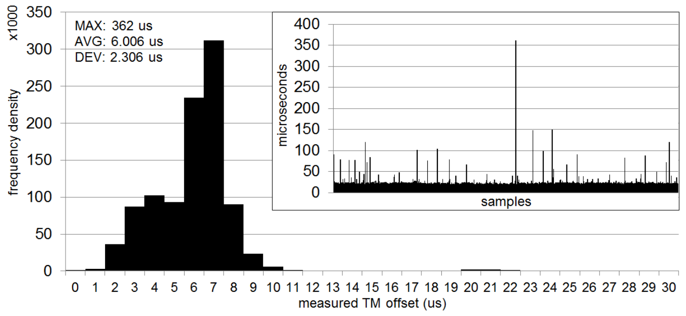


Figure 11: Results of the master synchronization experiment. The bin size of the histogram is $1\,\mu s$. The inset shows the value for each sample.

frames of size 72 bytes, the TM intertransmission time τ was $1\,\mu s$, and the bit rate was 100 Mbps.

To demonstrate the feasibility of isochronous lockstep transmission of TMs by masters, we implemented an additional proof-of-concept prototype, involving two proof-of-concept masters sending TMs to a monitoring station [3]. The measured absolute offsets with which the TMs arrived at the monitoring station verified that the prototype corresponded to the design and also further validated the lockstep synchronization. The results are shown in Figure 11.

The replica determinism of the FTTRS switches is also feasible. Making FTTRS switches start out with consistent SRDBs is just a matter of properly preconfiguring them. 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 EC; that is, one switch would not take significantly longer than the other for a computation. Since minimum update requests are piggybacked onto TMs, ensuring that FTTRS switches can reliably exchange them relies on the TMs being proactively retransmitted, which we already pointed out to be feasible. It is also feasible to have a total ordering of update requests. For instance, as suggested earlier, 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 aforementioned experiments related to synchronizing two FTTRS masters. 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 explained earlier, internal switch and port guardian replica determinism is a consequence of master replica determinism).

The system wide update of real-time requirements 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. Second, as discussed above, FTTRS switches can be made replica deterministic in practice. Third, master command messages can be piggybacked onto the fault-tolerant TMs as long as they fit within a maximum sized Ethernet frame payload (1500 bytes) together with the other information carried in TMs. 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 because the timeframe available is always determined by the length of the ECs and this length is a parameter that can be increased before deployment (assuming the application can work with larger ECs).

The final proof-of-concept prototype put a control application on top of FTTRS [26] and showed the feasibility of integrating the different mechanisms of FTTRS. It also showed 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. A video demonstrating the final prototype can be found online [27]. As can be seen in the prototype, the injection of faults did not lead to a loss of service to the control application—unless more faults were injected than the system was designed to tolerate. Although not shown in the video, we have also injected all possible permanent faults that should be tolerated by the control application and verified that they were indeed tolerated.

VII. Applicability of FTTRS

As already said, the chosen mechanism for synchronizing the switches does not introduce significant delays beyond the one required for transmitting additional trigger message replicas. Moreover, proactive retransmissions reduce the worst-case latency to get a message to its destination when compared to as-needed retransmissions. Nevertheless, it is clear that any temporal replication of messages always consumes bandwidth and therefore leads to a reduction in the network performance for the exchange of data messages. This may reduce the suitability of FTTRS for applications in which a very high bandwidth is necessary, such as intense streaming of high quality audio/video.

The impact of FTTRS on the available bandwidth can be analysed on a EC by EC basis and it is twofold. First, there is always a reduction caused by the proactive retransmission of trigger messages. And second, it is also necessary to consider the bandwidth reduction caused by proactively retransmitted slave messages. Only the critical slave messages will be retransmitted and the decision of which ones are critical is application-dependant. In this discussion we are going to assume the worst case in which all messages are critical.

Before continuing with the analysis it is important to clarify the potential impact of the replica radiation phenomenon described in Section V. As can be seen in Figure 7, for each slave message $2k$ replicas are transmitted in each one of the downlinks of any receiving slave. This level of redundancy is not only unnecessary, since the value of k is chosen to provide enough reliability in each link, but it is also undesirable due to its impact on performance. Furthermore, this impact of replica radiation on downlinks is easily avoidable by simply designing the switches to only transmit on the downlinks k replicas of only one of the replicas received for each slave message (actually it has been already avoided in the implementation shown in the video [27]). Therefore for the rest of this discussion we are going to assume that downlinks only have to accommodate k replicas of each slave message.

Based on all the assumptions indicated above and in the absence of faults, the links that have to accommodate more messages per EC are the interlinks, which convey the traffic from one switch to the other. In them, first, the k replicas of the TMs are transmitted, and second, all slave messages received by one switch are transmitted to the other for fault tolerance, what in the absence of faults accounts for all the slave messages exchanged in the network. Note that the TMs are transmitted between switches in both directions for implementing the replica determinism enforcement illustrated in Figure 9, and in the direction from the leader to the follower for the additional purpose of allowing the synchronization between the switches. Obviously if one slave has to receive all slave messages of the EC, then its downlinks have to convey the same traffic as the mentioned interlinks.

In each of these interlinks, which constitute the bottleneck of the network, the TMW has a bigger size (when compared to regular FTT) to accommodate the k TM retransmissions, thus reducing the time available for the other windows by $k1$ times the duration of the TM. Moreover each one of the slave messages will use an additional interval of $k1$ times the duration of said message. In any case, all these extra needs will be known in advance by the scheduler, which will decide if the messages are schedulable in each EC. Once the specific application is known, it is important to keep in mind these extra needs of critical traffic when considering the performance of more bandwidth-demanding traffic.

Similar to FTT Ethernet, FTTRS is primarily intended to support relatively small adaptive DESs that rely on a single hop switch Ethernet network. The major advantage of FTTRS with respect to FTT is that FTTRS provides high reliability while being very flexible with respect to mixing control and data traffics with different levels of criticality. There are many examples of DESs that can benefit from FTTRS, such as vision-based mobile robots, complex industrial machinery, automobiles, forest vehicles, medium-large size UAVs, small airplanes, among others.

When the size of the considered DES increases, it will be necessary to scale FTTRS as a multi-hop network. This will be the case of DESs controlling large vehicles, e.g. commercial planes, or IoT applications such as those in the context of smart cities, smart grids, etc. Thus, it will be a matter of future work to investigate how to scale FTTRS for those kinds of systems. For this investigation a suitable starting point would be the one recently done on extending FTT Ethernet to multi-hop architectures [30].

VIII. Conclusion

FTTRS not only meets the requirements (Section I), but surpasses them by means of introducing hardware redundancy and proactive retransmissions, determined by a configurable redundancy level. FTTRS, like all FTT versions, provides flexibility for changing real-time requirements, and also for other requirement changes, such as which nodes are publishers or subscribers of particular messages. Moreover, FTTRS adds 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 at design time, 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, such as the permanent failure of multiple components as long as there is still a path between any pair of slaves and one interlink remains. FTTRS can tolerate a scenario where each slave has lost a link and all but one interlink have failed, as in Figure 8.

Acknowledgement

This work was supported by projects DPI2011-22992 and TEC2015-70313-R (Spanish Ministerio de economía y competividad) and by FEDER funding.

The authors also wish to thank the following individuals for their useful comments on the preliminary ideas of this work: Luis Almeida, Inés Álvarez, Paulo Pedreiras, Guillermo Rodríguez-Navas, Ignasi Furió and Thilo Sauter.

References

[1] K. Nagatani *et al.*, "Emergency response to the nuclear accident at the Fukushima Daiichi nuclear power plants using mobile rescue robots", *Journal of Field Robotics*, vol. 30, no. 1, pp. 44–63, 2013.

[2] P. Pedreiras *et al.*, "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.

[3] A. Ballesteros *et al.*, "Achieving elementary cycle synchronization between masters in the flexible time-triggered replicated star for Ethernet", in *Proc. 19th IEEE Int. Conf. Emerging Technologies Factory Automat. (ETFA'14)*, IEEE, 2014.

[4] D. Gessner *et al.*, "Towards an experimental assessment of the slave elementary cycle synchronization in the flexible time-triggered replicated star for Ethernet", in *Proc. 19th IEEE Int. Conf. Emerging Technologies Factory Automat. (ETFA'14)*, IEEE, 2014.

[5] D. Gessner *et al.*, "Experimental evaluation of network component crashes and trigger message omissions in the flexible time-triggered replicated star for Ethernet", in *Proc. 11th IEEE World Conf. Factory Commun. Syst. (WFCS'15)*, IEEE, 2015.

[6] A. Avižienis *et al.*, "Basic concepts and taxonomy of dependable and secure computing", *IEEE Trans. Depend. Sec. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[7] IEEE, *Standard for local and metropolitan area networks - bridges and bridged networks*, IEEE Std 802.1Q-2014, Aug. 2014.

[8] *Routing bridges (RBridges): Base protocol specification*, RFC, 2011.

[9] M. Huynh *et al.*, "Resilience technologies in ethernet", *Computer Networks*, vol. 54, no. 1, 2010.

[10] L. Almeida *et al.*, "Approaches to enforce real-time behavior in Ethernet", in *The Industrial Communication Technology Handbook*, R. Zurawski, Ed., CRC Press, 2005, ch. 20.

[11] International Electrotechnical Commission (IEC), *Industrial communication networks - high availability automation networks - part 2: media redundancy protocol (mrp)*, IEC 62439-2, 2016.

[12] ——, *Industrial communication networks - high availability automation networks - part 3: parallel redundancy protocol (PRP) and high-availability seamless redundancy (hsr)*, IEC 62439-3, 2016.

[13] M. Li *et al.*, "Reliability enhancement of redundancy management in afdx networks", *IEEE Transactions on Industrial Informatics*, vol. 13, no. 5, pp. 2118–2129, 2017.

[14] Society of Automotive Engineers (SAE), *Time-triggered Ethernet*, SAE, 2011.

[15] TSN Task Group, *Ieee standard for local and metropolitan area networks - frame replication and elimination for reliability*, IEEE Std 802.1CB-2017, Oct. 2017.

[16] W. Steiner *et al.*, "Traffic planning for time-sensitive communication", *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 42–47, Jun. 2018.

[17] TSN Task Group, *Ieee draft standard for local and metropolitan area networks - media access control (mac) bridges and virtual bridged local area networks amendment: stream reservation protocol (srp) enhancements and performance improvements*, IEEE P802.1Qcc/D2.2, Jan. 2018.

[18] M. L. Raagaard *et al.*, "Runtime reconfiguration of time-sensitive networking (tsn) schedules for fog computing", in *2017 IEEE Fog World Congress (FWC)*, Oct. 2017, pp. 1–6.

[19] A. Burns *et al.*, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, 4th. Addison-Wesley, 2009.

[20] M. Ashjaei *et al.*, “Evaluation of dynamic reconfiguration architecture in multi-hop switched ethernet networks”, in *Proc. of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sep. 2014, pp. 1–4.

[21] G. Carvajal *et al.*, “Integrating dynamic-tdma communication channels into cots ethernet networks”, *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1806–1816, Oct. 2016.

[22] A. Azim *et al.*, “Generation of communication schedules for multi-mode distributed real-time applications”, in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, IEEE, 2014, pp. 1–6.

[23] P. Pedreiras *et al.*, “Ftt-ethernet: A flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems”, *IEEE Trans. Ind. Informat.*, vol. 1, no. 3, pp. 162–172, Aug. 2005.

[24] R. Marau *et al.*, “Enhancing real-time communication over COTS Ethernet switches”, in *Proc. 6th IEEE Int. Workshop Factory Commun. Syst. (WFCS’06)*, IEEE, vol. 6, 2006, pp. 295–302.

[25] R. Santos *et al.*, “A synthesizable Ethernet switch with enhanced real-time features”, in *Proc. 35th Annu. Conf. IEEE Industrial Electronics Society (IECON’09).*, IEEE, 2009, pp. 2817–2824.

[26] A. Ballesteros *et al.*, “First implementation and test of a node replication scheme on top of the flexible time-triggered replicated star for Ethernet”, in *Proc. 12th IEEE World Conf. Factory Commun. Syst. (WFCS’16)*, IEEE, 2016.

[27] A. Ballesteros. (2017). FT4FTT final prototype demo, [Online]. Available: http://srv.uib.es/ft4ftt-final-prototype-demo/.

[28] S. Poledna, “Method and switching unit for the reliable switching of synchronization of messages”, US Patent App. 14/391162, Mar. 2015.

[29] Society of Automotive Engineers (SAE), *Ttp communication protocol*, SAE AS6003, 2011.

[30] M. Ashjaei *et al.*, “Improved message forwarding for multi-hop HaRTES real-time ethernet networks”, *Journal of Signal Processing Systems*, vol. 84, no. 1, pp. 47–67, 2016.

David Gessner received the first degree in informatics engineering (Ingeniería Superior en Informática), the Masters degree in information and communication technologies and the doctorate in information and communication technologies from the University of the Balearic Islands, Palma de Mallorca, Spain, in 2010, 2011 and 2017 respectively.

His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems, dependable communication topologies, and field-bus and industrial networks.

Julián Proenza (SM12) received the first degree in physics (Licenciatura en Ciencias Físicas) and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 1989 and 2007, respectively.

He is currently a Lecturer with the Department of Mathematics and Informatics, UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems, clock synchronization, dependable communication topologies, and field-bus and industrial networks.

Dr. Proenza is a Member of the IEEE Industrial Electronics Society (IES) since 2009 and Senior Member since 2012. He is also a member of the IES Technical Committee on Factory Automation.

Manuel Barranco received the first degree in informatics engineering (Ingeniería Superior en Informática) and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2003 and 2010, respectively.

He is currently a Lecturer with the Department of Mathematics and Informatics, UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems, dependable communication topologies, and field-bus and industrial networks.

Dr. Barranco is a member of the IES Technical Committee on Factory Automation.

Alberto Ballesteros received the first degree in informatics engineering (Ingeniería Superior en Informática) and the Masters degree in computing engineering from the University of the Balearic Islands, Palma de Mallorca, Spain, in 2012 and 2016, respectively.

He is currently working towards the Ph.D. degree in information and communications technologies in the same university. His research interests include dependable and real-time systems, fault-tolerant distributed systems, adaptive systems, dependable communication topologies, and field-bus and industrial networks.