A Proposal for Master Replica Control in the Flexible Time-Triggered Replicated Star for Ethernet

David Gessner, Julián Proenza, Manuel Barranco
DMI, Universitat de les Illes Balears, Spain
davidges@gmail.com, {julian.proenza, manuel.barranco}@uib.es

Abstract

Traditionally, distributed embedded systems have been designed using static approaches, i.e., assuming a static environment. However, when the environment is dynamic and imposes changing requirements on the system, flexible approaches are needed. If in addition the system must operate continuously, then high reliability is also required.

The necessary flexibility for real-time requirements may be provided by Hard Real-Time Ethernet Switching (HaRTES), which is an implementation of the master/multi-slave Flexible Time Triggered (FTT) communication paradigm. To help provide high reliability, the Flexible Time-Triggered Replicated Star (FTTRS) adds network fault tolerance by duplicating the HaRTES switch and master, as well as the network links. This paper discusses how to enforce replica determinism for the masters in FTTRS by ensuring that they show corresponding outputs both in the time and value domain. Moreover, it suggests how to proceed in case replica determinism is lost despite all efforts.

1. Introduction

If a distributed embedded system (DES) must operate continuously while satisfying unpredictable requirement changes, then it must be both highly reliable and flexible. In particular, this requires that the communication channel of the DES satisfies those attributes.

The goal of the *Flexible Time-Triggered Replicated Star for Ethernet* (FTTRS) [1] is to provide such a channel for systems with changing real-time requirements. It is part of the *Fault Tolerance for Flexible Time-Triggered Ethernet-based systems* (FT4FTT) project, which aims to provide high reliability and flexibility for all crucial parts of a DES.

FTTRS is based on *Hard Real-Time Ethernet Switching* (HaRTES) [2], a switched ethernet implementation of the Flexible Time Triggered (FTT) communication paradigm [3]. HaRTES implements a micro-segmented star topology with the HaRTES switch as a central element. The switch provides the most relevant functions of FTT by embedding the FTT master, which polls multiple slaves by means of a single periodic message called *trigger message* (TM). The master thereby divides the communication time into rounds of fixed duration, where each round is initiated by a new TM transmission. These rounds are called *elementary cycles* (ECs) and can be divided into a *synchronous window* followed by an *asynchronous window*. In each EC, the corresponding TM conveys a *schedule* that tells the slaves which synchronous messages to transmit during the EC's synchronous window. The schedule is calculated by the master according to a *system requirements database* (SRDB), which specifies the system's real-time requirements and whose contents may be changed by slaves using *SRDB update request messages*.

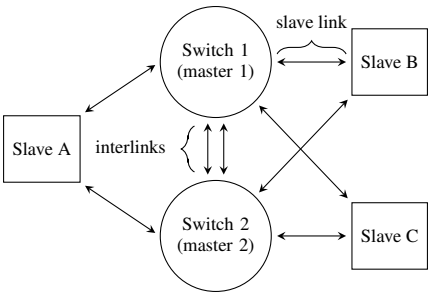

Figure 1. FTTRS architecture.

The architecture of FTTRS has been presented recently [1]. Figure 1 shows its main components. It is comprised of two HaRTES switches, each with its own master and SRDB. The slaves are connected to both switches by means of *slave links* and there are several redundant links between the switches called *interlinks*.

The masters in FTTRS also provide a total order multicast service [4] for the slaves, such that a synchronous message is either delivered by all or none of its intended recipient slaves. This service is being developed within FT4FTT under the name of *Reliable Total Order Publish/Subscribe* (ReTOPS). It assumes a single HaRTES switch, but can be integrated with a replicated channel. Also, it is based on a 4-phase commit protocol that relies on a reliable TM [4]. Slaves transmit synchronous messages as dictated by the schedule conveyed by the TM. Receiving slaves then acknowledge to the master the reception of each of those messages using dedicated acknowledgment messages (ACKs), but without yet delivering the received message to the locally executing application. If the master receives ACKs for a given message from all intended recipient slaves, it instructs those slaves to deliver the message in the next EC. This is done by adding a *status vector* field to the next EC's TM, which is a bit vector where each position corresponds to a given synchronous message that was scheduled in the previous EC. Only if such a position has a bit value of 1,

 doi:10.1109/WFCS.2014.6837599

the slaves deliver the corresponding message.

To replicate the switches and their ReTOPS-enhanced masters, FTTRS uses *active* and *semi-active replication* [5]. This means that both switches, and their masters, are simultaneously providing the same service — either as equals in the case of active replication or with one being distinguished from the other in the case of semi-active replication. Either way, since the replicas provide the same service and the slaves use both of them simultaneously, this allows the failure of either switch to be tolerated without interrupting the communication between slaves. However, for this to work, it is essential that the replicas are *replica determinate* [5]: starting from the same internal state, they must have *corresponding* outputs. What this means depends on the service provided by the replicated entities.

In the case of the masters, their service is threefold: they dictate when each EC begins, what synchronous messages should be transmitted in each EC by the slaves, and which synchronous messages transmitted in an EC should be delivered by the slaves. Using this definition for the master service, we can distinguish between output correspondence in the time and value domain. In the time domain, the masters are replica determinate if they transmit their TMs quasi-simultaneously, TM by TM. In the value domain, they are replica determinate if they convey the same schedule and status vector in the TMs. Enforcing replica determinism is called *replica control* [5], and in the time domain it is a time synchronization problem; whereas in the value domain it is a consensus problem: a set of distributed entities (the masters) must agree on a value.

Regarding the switches, their service is to drop erroneous frames and forward correct ones, where the latter is non-standard forwarding based on *streams* (see Section 3.1). We consider switches to show corresponding outputs if, when they receive the same frame, they consistently drop the frame or forward it to the same set of slaves. Both tasks take into account the local master's SRDB. Thus, for the switches to be replica determinate, their masters need consistent SRDBs. Since this is already necessary for master replica control, having master replica determinism helps achieve switch replica determinism.

This paper discusses replica control for the masters in FTTRS. Section 2 addresses the time synchronization problem and Section 3 the consensus problem. Section 4 explains what to do in case replica determinism is lost despite all efforts. Section 5 concludes the paper.

2. Elementary cycle synchronization

As stated previously, ReTOPS relies on a reliable TM. To provide this reliability, in FTTRS each EC starts with a *trigger message (TM) window*. During that window, each master transmits k replicas of the TM on all links and interlinks, where k is a function of the channel's bit error rate. If k is large enough, each slave with a link to a correct master receives at least one TM replica with a sufficiently high probability, even in the presence of transient link faults.

To synchronize the masters' TM transmissions, we propose a semi-active replication mechanism that takes advantage of the k TM replicas. We consider one of the masters the *leading master* (or *leader*) and the other the *follower master*. The leader transmits TMs according to its own internal clock and does not synchronize itself with anyone. The transmissions occur on the slave links of the switch where the leader is located, as well as on the interlinks. The follower also transmits on its slave links and the interlinks according to its own internal clock. However, in contrast to the leader, in each EC it resynchronizes its TM transmissions using the arrival times through the interlink of the incoming TMs. Specifically, it uses the arrival time of each received TM to decide whether it needs to defer or advance the start of its next TM transmission in order to be in sync with the leader. Note that this means that until the follower successfully receives a TM from the leader, its TM transmissions might be out of sync. However, if the EC duration and the value of k are appropriately set, and the clock drift is not excessive, there should be enough resynchronizations for the deviation in TM transmission times to be acceptable. Resynchronization can then only fail to occur when the leader crashes or all interlinks are affected by permanent faults. In case of a crash of the leader, this is tolerated transparently since the follower's clock will now dictate the ECs. If it is due to several permanent failures affecting all interlinks, then the network will be partitioned in two. In that case, maintaining synchronization is impossible. Nevertheless, we are working towards making the slaves handle this situation.

3. Towards consensus among masters

With perfectly reliable components, consensus problems can be solved by an exchange of messages [6]. However, when components can suffer faults, guaranteeing consensus may be impossible [6]. Nevertheless, solutions can be found that reach consensus with a certain probability.

The specific solutions depend on the failure assumptions. For instance, and using graph theory terminology, when only edge faults are considered, the problem can be modeled as the *coordinated attack problem* [6] and its probabilistic solutions may be adequate; or, if only vertex failures are considered, and these can fail arbitrarily, the problem might be modeled as the *Byzantine generals problem* [6] and its solutions might be applicable.

In FTTRS, both edge and vertex failures are considered, where edges are the slave links and interlinks, and the vertices are the slaves and the switches with their embedded masters. Links are assumed to have omission failure semantics, i.e., they might either transiently or permanently fail to forward a message. This makes sense considering that links cannot generate frames themselves and that both the HaRTES switches and the slaves drop corrupted frames. Regarding the masters and the switches, they have crash failure semantics, i.e., they either produce a correct result or remain permanently silent. Finally, the slaves have incorrect computation failure semantics in the value domain, i.e., they might transmit incorrect values, but they cannot

transmit at arbitrary times. Notably, they cannot impersonate other slaves and do not present two-faced behaviors, i.e., they cannot send a message m through one slave link and a message m' through the other such that m and m' purportedly provide the same information, while they actually convey different values. How the failure semantics are enforced is out of the scope of this paper, but for the slaves it mostly relies on port guardians implemented at the switches [7] and additional CRC-based integrity checks, and for the switches and masters it relies on internal duplication and comparison [1]. Finally, besides the failure semantics, for the masters there are two additional and realistic assumptions: they do not have any internal non-determinism and they are initialized with the same internal state, e.g., they start with the same SRDB contents.

With the above assumptions, and taking into account that the masters provide their service on an EC-by-EC basis, consistent outputs by the masters can be achieved by ensuring that they receive the same input within the same EC. Specifically, it may be achieved if, within the same EC, they receive the same SRDB update requests and they receive ACKs from the same set of slaves. However, in the presence of link faults, it is not sufficient for each slave to transmit the same message to each switch. After all, messages might be lost in only one channel, preventing the masters from receiving the same input.

To deal with message losses, a common approach is to retransmit only when it is likely that the original transmission has not reached its destination. This is usually done for bandwidth efficiency. Transmitting on an only-as-needed basis, however, can be problematic if the transmission must succeed before a tight and hard deadline. This is so because it may take a non-deterministic amount of time for the sender to detect that a retransmission is necessary.

In FTTRS, bandwidth efficiency is secondary. The main goal is high reliability. Moreover, the end of each EC constitutes a tight and hard deadline since masters must reach a consensus within each EC to be replica determinate. The solution we therefore propose is similar to the one used for reliable TMs: slaves proactively retransmit k times before the end of an EC any message that must be received by both masters within that EC, where k is again a function of the bit error rate in the links. Moreover, to ensure consistent input for the masters with a sufficiently high probability, the k messages are also forwarded through the interlinks in case some slaves only have a link to one of the switches. Note that the value of k can, but does not necessarily need to be, the same as the value of the k from Section 2. This solution is simple, takes a predictable and deterministic amount of time, and does not require any type of failure detection by a transmitting slave or switch. Next, we describe the details of this solution for the ACKs and then for the SRDB update requests.

3.1. The cumulative ACK slot vector mechanism

As explained above, we aim for both masters to receive ACKs from the same set of slaves by transmitting each ACK k times in an EC. This could be achieved by having each slave respond to a synchronous message reception with k dedicated ACK messages. If a given slave receives n synchronous messages, it would transmit nk dedicated ACK messages. We propose a more efficient solution.

In HaRTES and FTTRS, each synchronous message belongs to a *synchronous stream*, whose configuration is stored in the SRDBs. Such a stream defines, among other things, the period and deadline of its synchronous messages, and is used by the master as the scheduling unit, analogous to tasks in processor scheduling. Since the periods are integer multiples of the EC length, no two messages of the same stream are scheduled for the same EC. Moreover, a stream identifies a transmitter and set of recipients, which are common to all messages of the stream.

Let $\sigma_s = (\mu_1, \mu_2, \ldots, \mu_{n_s})$ be a tuple of the streams for which a slave s is a recipient, with n_s being the number of such streams. We denote the ith component of a tuple σ_s as $\sigma_s(i)$. We propose to store a tuple $v_s = (b_0, b_1, \ldots, b_{n_s})$ in each slave s, which we call a *cumulative ACK slot vector*, or *ACK vector* for short. Each slot $v_s(i)$ is initialized with a value of 0, and set to 1 when s receives a synchronous message from stream $\sigma_s(i)$. Moreover, $v_s(i)$ is reset to 0 when a status vector is received that instructs the delivery of the latest message of stream $\sigma_s(i)$. In a given EC, an ACK vector v_s therefore keeps track of the streams for which s has received, but not delivered, messages.

Whenever a slave transmits a message, which may be synchronous or asynchronous, the current contents of the slave's ACK vector are piggybacked. Thus, any message transmitted by a slave not only conveys the data specific to that message, but also acknowledges the reception of all the synchronous messages that the slave has received so far in the current EC. This means that whenever a slave receives a synchronous message m, it must subsequently transmit at least k messages before the end of the EC to acknowledge m. If the slave still had messages to transmit, the ACK vector will be piggybacked on them. Only if this is not the case, the slave would have to transmit additional messages whose payload is nothing but the ACK vector.

The above mechanism has several notable features. First, all received messages are acknowledged at least k times, but may be acknowledged more times if they are received by a slave that still has more than k messages to transmit in the same EC after the reception. This allows to maximize the probability of the masters instructing a delivery for a given synchronous message by transmitting it before others. In that case the message is received at the earliest possible time within an EC and thus the probability that a receiver has more than k transmissions pending is highest. Second, the ACK vectors can be used for error detection since an ACK slot should only be reset to 0 if the delivery of the corresponding message has been instructed through the status vector. If it is reset at any other time, this is evidence of an error. Third, the ACK vectors may help the masters infer the bit error rate on the slave links as missing ACKs may indicate corrupted frames. This could

be used to decide on a value for the parameter k. Finally, it requires less bandwidth than dedicated ACK messages. This is especially noticeable if the message payload is so small that padding must be used to comply with Ethernet's minimum payload size. In that case, the piggybacked ACK vectors would substitute part of the padding.

A disadvantage of the mechanism is that there is the danger of incorrect acknowledgments if an ACK vector gets corrupted such that 0s become 1s. However, with the above-mentioned error detection, and additional integrity checks, the probability of this can be made negligible.

3.2. Towards consistent SRDB updates

As already mentioned, for the masters to be replica determinate, they must have the same contents in their SRDBs whenever they calculate the schedule for the next EC. Since the masters start with the same SRDB contents, and they have crash-failure semantics and no internal non-determinism, inconsistencies in the SRDBs can only be due to them having processed different SRDB update requests.

SRDB update request messages may be significantly larger than an ACK vector. Also, their transmission occurs during the asynchronous window and thus may occur near the very end of an EC. For these messages we therefore favor an alternative to the piggybacking approach from Section 3.1. Specifically, we propose to use a reconciliation mechanism during the TM window. The idea is the following. First, we define an a priori total order relation for the update requests so that a master can always unambiguously determine which is the next one to process from a set of requests. We call this request that is the next to be processed the *minimum* of the set. During the TM window, each master exchanges with the other one the minimum of all the update requests it has received so far. If this exchange is done reliably, by the end of the TM window both masters will either have exchanged the same update request message, or different ones. In either case, after the exchange the set of pending requests in each master will contain the same minimum. By ensuring that the masters only process their local minimum before the asynchronous window of each EC, both masters will update their local SRDB equally.

Note that the above requires the reliable exchange of the minimum. For this we again use the proactive retransmission of k replicas. Moreover, since the interlinks during the TM window are also used to exchange TMs, there might not be enough time to exchange both the TMs and the local minimum of each master. This can be solved by either increasing the duration of the TM window or by increasing the available bandwidth in the interlinks.

4. Having a plan B

The above mechanisms all rely on the parameter k having a sufficiently high value for at least one copy of a given message (TM, ACK, or update request) 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 still be possible for an unexpectedly long error burst to corrupt all k replicas of a given 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 propose to 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 TMs. Note that the follower can detect the loss of replica determinism as long as it receives the TMs from the leader through the interlinks. In that case, we propose the follower to shut down its service to prevent any potential interference with the leader's service.

5. Conclusions and future work

This paper proposed how to enforce replica determinism for the masters in FTTRS, a fault-tolerant duplicated star topology based on a switched Ethernet implementation of the FTT paradigm. The guiding design principle for achieving replica determinism is to proactively retransmit any message that constitutes an input that must be consistent for the masters. Following this principle, we can provide replica determinism under the tight deadlines that the round-based FTT paradigm imposes.

Future work includes the design of a reintegration mechanism for the follower, in case replica determinism is lost; an analysis of the reliability achievable for different values of the parameter k, which gives the number of proactive retransmissions; and an implementation of a prototype of FTTRS based on the presented ideas.

Acknowledgements

This work was supported by project DPI2011-22992 and grant BES-2012-052040 (*Spanish Ministerio de economía y competividad*), and by FEDER funding.

References

[1] D. Gessner, J. Proenza, M. Barranco, and L. Almeida, "Towards a flexible time-triggered replicated star for Ethernet", in *18th IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, Sept. 2013, Cagliari, Italy.

[2] R. G. V. dos Santos, *Enhanced Ethernet switching technology for adaptive hard real-time applications*, PhD thesis, Universidade de Aveiro, 2010.

[3] P. Pedreiras and L. Almeida, "The Flexible Time-Triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems", in *Proc. Int. Parallel and Distributed Processing Symposium*, 2001. IEEE Comput. Soc.

[4] G. Rodriguez-Navas and J. Proenza, "A proposal for flexible, real-time and consistent multicast in FTT/HaRTES Switched Ethernet", in *18th IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, Sept. 2013.

[5] S. Poledna, *Fault-tolerant real-time systems: the problem of replica determinism*, Kluwer Academic Publishers, 1996.

[6] N. A. Lynch, *Distributed algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[7] A. Ballesteros, D. Gessner, J. Proenza, M. Barranco, and P. Pedreiras, "Towards preventing error propagation in a real-time Ethernet switch", in *18th IEEE Conf. on Emerging Technologies and Factory Automation*, 2013, Cagliari, Italy.