Towards a Layered Architecture for the Flexible Time-Triggered Replicated Star for Ethernet

David Gessner, Ignasi Furió, Julián Proenza
DMI, Universitat de les Illes Balears, Spain
davidges@gmail.com, {ignasi.furio,julian.proenza}@uib.es

***Abstract*—Distributed embedded systems (DES) have traditionally been designed assuming that the requirements they need to satisfy are known in advance. If this is not the case, and a DES should operate autonomously without interruption, it needs to be adaptive. For this, flexible approaches are necessary and this applies in particular to the network of the DES. However, if the probability of faults occurring is non-negligible, then flexibility alone is not enough and fault tolerance is also necessary. The Flexible Time-Triggered Replicated Star for Ethernet (FTTRS) is a set of protocols and mechanisms together with a specific network topology that builds on a switched-Ethernet implementation of the Flexible Time-Triggered (FTT) communication paradigm by enhancing it to not only provide flexibility, but also fault tolerance. This paper describes our efforts towards a layered architecture for FTTRS to benefit from the well-known advantages of these architectures, such as making the complexity manageable and easier to communicate, and making the design more future proof by allowing changes in one layer without affecting other layers.**

I. Introduction

A distributed embedded system (DES), like any embedded system, is deployed in a physical environment with which it interacts and which imposes requirements upon it. Traditionally such requirements were assumed to be static or at least predictable. More recently these assumptions have been relaxed and there is an interest in flexible solutions that allow a DES to keep operating autonomously even if the requirements can change in unpredictable ways. The *Flexible Time Triggered Replicated Star for Ethernet* (FTTRS) [1] is one such solution for the network of a DES. In contrast to other Ethernet-based solutions, it has been designed with both flexibility and high reliability in mind. More specifically, FTTRS aims at providing the necessary flexibility to adapt to changing real-time requirements, while providing high reliability by means of fault tolerance. For this it adds fault tolerance to a switched Ethernet implementation of the Flexible Time-Triggered (FTT) communication paradigm [2].

The FTT paradigm follows a master/multi-slave communication model where a master schedules the transmission of messages based on the current real-time requirements and then broadcasts that schedule by means of a so-called *trigger message* (TM). The slaves then comply by exchanging messages according to this schedule before the next schedule is broadcast. The time between the broadcast of one schedule and the next is called an *elementary cycle* (EC) and it has a fixed duration. Each EC thus begins with the broadcast of a TM. The flexibility of FTT comes from the fact that slaves can request changes to the current real-time requirements and these changes, if the master allows it, dictate future schedules.

978-1-4673-7929-8/15/$31.00 © 2015 IEEE

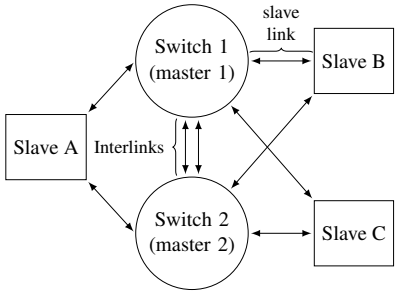

Figure 1. FTTRS architecture.

The architecture of FTTRS is shown in Figure 1. It comprises an arbitrary number of FTT slaves connected by means of *slave links* to two custom Ethernet switches, each of which embeds an FTT master and is connected to the other switch by means of redundant *interlinks*. The network components therefore form a topology without any single points of failure and with a redundant path between any pair of slaves.

In FTTRS the schedule for each EC is calculated by the two masters and this is done in a mutually consistent manner [7]. Moreover, each schedule is broadcast by both switches simultaneously in redundant TMs to maximize the probability that at least one TM gets through to each slave even in the presence of transient channel faults. Regarding the use of the spatially redundant network by the slaves, each slave transmits each message through both its slave links simultaneously. Moreover, each switch forwards all slave messages to the other switch. As a result, in the absence of faults, each slave receives the same messages through each of its slave links.

FTTRS thus provides several services to an application. First, it provides services that it inherits from FTT: it provides a synchronization service that ensures that the applications running on the slaves have a common time base with a granularity of the EC length, it enables the exchange of messages with real-time requirements, and it allows an application to request changes to these requirements. Second, FTTRS provides services not provided by any other Ethernet-based implementation of FTT: fault-tolerant communication that can be used by an application in a transparent way and, if required, a total-order multicast service [6].

 doi: http://dx.doi.org/10.1109/ETFA.2015.7301636

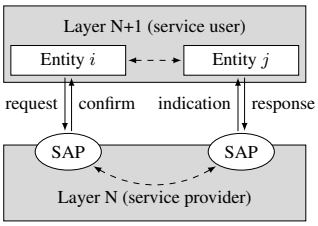

Figure 2. Layer interaction model.

Providing the above services requires several protocols and mechanisms. For instance, the masters must transmit their TMs synchronously and redundantly, the slaves must use the received TMs to agree on the start of an EC, and the masters must agree on the schedule they produce for each EC and do so even if the real-time requirements have changed. This results in a significant amount of complexity. We therefore propose a layered architecture for FTTRS that follows the abstract definitions and architecture described in the *Open Systems Interconnection Reference Model* (OSI model) [3].

This has several advantages. It makes the complexity more manageable by allowing us to focus on one set of features at a time; it makes it easier to obtain a description of the FTTRS architecture that is more amenable to formal verification, e.g., layers may be verified formally one by one; and it makes the FTTRS design more future proof by allowing the modification of certain features that are encapsulated in one layer, without having to modify the other layers.

The paper is organized as follows. Section II introduces the OSI layer interaction model. Section III proposes a layered architecture for FTT to serve as a starting point for a layered architecture of FTTRS. Section IV discusses the latter. Finally, Section V concludes the paper.

II. OSI Layer interaction

The main goal of the OSI model is to provide a common basis for the development of standards that allow the interconnection of systems. This model divides the complex process of establishing a communication between systems into several simpler processes and organizes them in a layered architecture.

Figure 2 shows how a layered system is organized and works. Each layer contains *entities*, which are abstract objects that execute some logic and interact with other entities within the same layer. A set of such interacting entities are called *peer entities* (in the figure, Entity i and Entity j would be peer entities). The way peer entities interact is defined by means of *protocols*. How entities or protocols are implemented is irrelevant from the specification point of view.

If interacting entities are within the same machine, they can interact directly with each other. Otherwise, they can only interact indirectly (dashed lines in Figure 2) using *services* that are provided by the immediate lower layer. In this way layer $N+1$ is called the *service user* of layer N, while layer N is the *service provider* of layer $N+1$. The services provided by layer N are supplied by means of protocols between layer N peer entities, which use the services provided by layer $N-1$, and so on. Only peer entities at the bottom layer (layer 1) interact through a physical link.

Service users and providers exchange information through interfaces called *service access points* (SAPs). The reference model uses *service primitives* to describe this information exchange. Like entities or protocols, primitives are also described in an abstract way. The four most important types of primitives are shown in Figure 2 and are

- **Request**: used when the service user requests a service from the service provider.
- **Confirm**: used when a service provider acknowledges the completion of an activity initiated by a request primitive.
- **Indication**: used by a service provider to indicate to a service user that an event occurred as a consequence of providing a service.
- **Response**: used by a service user to acknowledge the reception of an indication.

Information exchanged between peer entities is transferred in discrete units called *Protocol Data Units* (PDUs). Every PDU is formed by *metadata* (i.e., a header or footer) followed by some data (called *payload*). The metadata contains information that can be understood by a peer entity, while payloads have no meaning for the peer entity and must be transferred up to the service user. A layer $N+1$ PDU is passed down to layer N, which considers it as a *Service Data Unit* (SDU). The SDU is accompanied by *Interface Control Information* (ICI), which is understood by a layer N entity. A layer $N+1$ SDU becomes the payload of a layer N PDU. This procedure is known as *encapsulation*. When finally the physical layer PDU is communicated over a physical link and reaches the destination machine, the process is inverted there (*decapsulation*): on every layer corresponding headers and footers are processed, while each payload is extracted and delivered to the immediate upper layer.

III. Layered architecture for FTT

The FTT paradigm has been implemented on Controller Area Network (CAN), as well as shared and switched Ethernet. FTT can thus be thought of as a layer that provides services to higher-layer applications running on slaves, while it uses services provided by a lower layer communication system. Unfortunately, however, there does not exist any abstract specification of the service interface between FTT and its adjacent layers, so we analyzed FTT to obtain one.

Our initial model has three layers as shown in Figure 3. At the top we have an application layer with application entities, which may be implemented by processes or threads executing within an operating system. These entities run on slave nodes and communicate with other application peer entities running on other nodes by using FTT services. The communication provided by the FTT services follows a publisher/subscriber model and satisfies real-time requirements that are specified by (possibly changing) attributes such as deadlines, periods, and minimum inter-arrival times. A logical connection with a certain set of attributes that is established between one application entity that acts as a publisher and multiple application peer

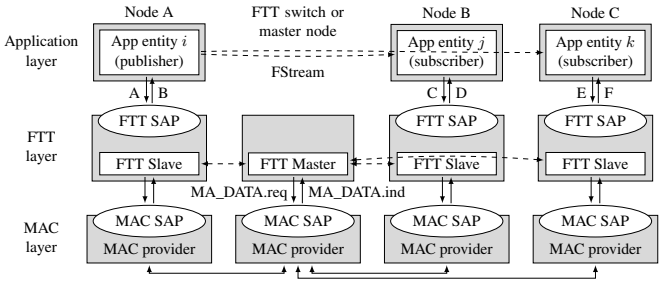

Figure 3. Example of layered behaviour for FTT.

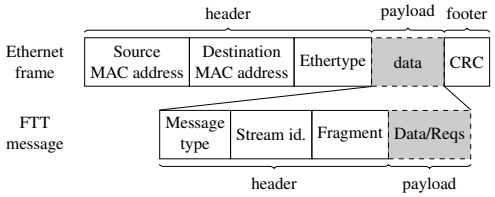

Figure 4. Encapsulation of FTT messages into Ethernet.

entities that act as subscribers is called a *flexible real-time publisher/subscriber stream* (FStream). Such Fstreams are made possible by the FTT layer using FTT SAPs. The interaction between application layer and FTT layer that enables FStreams is done by means of service primitives represented by arrows labeled A–F in Figure 3. The FTT layer not only acts as the service provider for the application layer, but also as a user of services provided by the MAC (*medium access control*) layer. The MAC layer and its corresponding physical layer are formed by Ethernet NICs in slaves, an Ethernet NIC at the master (if it is a node) or switching circuitry (if it is embedded in an FTT-enhanced switch), and connection cables. Slaves and master use primitives MA_DATA.req and MA_DATA.ind for interaction with the lower MAC layer [4]. For implementations over CAN similar identifications can be made.

Figure 3 also represents a particular example of the interaction between three slaves and an FTT master node (or FTT-enhanced switch). The application entity in node A acts as the publisher of an FStream, while the application entities in nodes B and C act as subscribers. This interaction between peer application entities through an FStream is provided by the FTT layer by means of specific types of FTT SAPs: an *FStream publisher SAP* in node A and *FStream subscriber SAPs* in nodes B and C. In addition, there also exist *FStream management SAPs*, which are used to create and maintain publisher/subscriber FStream SAPs. Primitives for these SAPs are used to attach application layer entities as publishers or subscribers to previously created streams, for data transmission and reception, etc. Details about all service primitives are out of the scope of this document, but some examples could be

- **Fadd_stream.req**: used by an application entity to request the creation of an FStream with certain atributes and identifier. This primitive is sent to the FStream management SAP. Every successful stream creation also creates a new SAP of type publisher or subscriber.
- **Fupdate_stream.req**: request which, if confirmed, provokes changes in attributes of a certain FStream.
- **Fattach_rx.conf**: confirmation of a previous Fattach_rx.req. It confirms to an application entity that it has been successfully attached to an FStream as a subscriber.
- **Fsend.req**: requests the transmission of data through a certain FStream where the sender entity has been attached as a publisher.

The information exchange in the FTT layer is controlled by the FTT master entity, which periodically transmits TMs to all FTT slave entities. From the TM every FTT slave entity knows for which FStreams data has been scheduled and thus which FTT SAPs it needs to access to deliver or receive application data. In Ethernet implementations, each FTT PDU, called *FTT message*, is individually encapsulated inside an Ethernet frame as can be seen in Figure 4. The FTT message format shown is just an example, but all FTT messages start with a type field from which all other header fields can be determined. Some other fields that may be present are stream identifiers, fragment identifiers (needed when data cannot be sent in a single payload), or a sender identifier. The header is followed by a payload formed with application data or requests.

Once ready, the FTT message is handed down the stack and reaches the MAC provider, which encapsulates it into the data field of an Ethernet frame. As Figure 4 shows, these frame headers are formed by two MAC address fields which contain NIC identifiers for source and destination interfaces. The *ethertype* field identifies the kind of information transported in the payload field, which always corresponds to FTT.

The main point of this three-layer model is to make the FTT layer independent from the applications running on slaves and the underlying physical communication system. This independence will allow us to develop a more complex model for FTTRS.

IV. Layered architecture for FTTRS

As we already said, FTTRS adds fault tolerance to FTT through replication of masters, slaves, and links. A proposal for managing this redundancy has been presented in [5] and [7]. Moreover, total-order multicast can also be provided [6]. This section will explain how the proposed techniques can be distributed into a layered architecture to isolate FTT from all fault-tolerance mechanisms and services, as well as services corresponding to providing total-order multicast.

A first diagram of the proposed layered architecture is shown in Figure 5. It represents the stack of layers corresponding to two slave nodes and two FTTRS switches. Although the switches are replicated, a main objective of the proposed layer architecture is to hide this replication from FTT slave entities in the FTT layer and any other entities at higher layers. We represent this by having FTT slave entities interact with a single *virtual* FTT master entity that is actually comprised of two FTT master entities. In this way it should be possible to take the FTT layer described in the previous section and

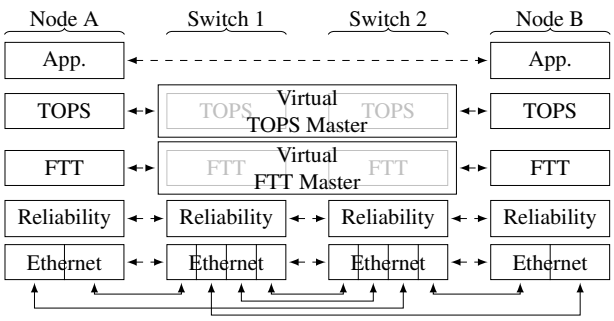

Figure 5. FTTRS layers architecture.

insert it unmodified in the slave nodes of the FTTRS layer architecture. On the other hand, the layers below the FTT layer are perfectly aware of the replication and one of their tasks will be to hide the replication from upper layers.

As usual in layered architectures, the bottom layer represents the physical communication system, which in this case consists of multiple independent Ethernet interfaces. Each slave node has two interfaces, each one connected to one of the replicated switches; while masters need two interfaces for the interlinks, plus another one for every attached node. This interconnection may result in multiple replicated frames when a single FTT message is handed down by an FTT slave entity. Since the original FTT slave entities do not expect any kind of replication, a service that deals with it is necessary. This is one of the main purposes of the *Reliability* layer.

Another important task of the reliability layer consists in ensuring that all FTT entities agree when each EC starts and ends, despite the fact that in FTTRS each TM is replicated and broadcast redundantly by each switch to make the TM more robust to transient link faults [5], [8]. For this, the nodes must be prepared to not only handle the TM replication, but also to handle the redundant reception due to replicated masters. So another of the services provided by the reliability layer is the replication of trigger messages that are handed down from the FTT layer by FTT master entities. Moreover, within a slave node, the reliability layer will have to collect all TM replicas and pass only one copy up to the FTT layer.

An example of tasks that can be done in layers above the FTT layer is the approach described in [6] to achieve total-order multicast. This approach assumes a single switch with an embedded FTT master, but thanks to our layered approach, can be integrated with a replicated channel relatively easily. The basic idea would work as follows. As shown in Figure 5, a *Total-Order Publish/Subscribe* (TOPS) layer can optionally be located between the application layer and the FTT layer. In order for that layer to be transparent to the application layer, the TOPS layer should mimic the interface of the FTT layer by providing streams that behave from the application's point of view just like FStreams. The difference, however, would be that these streams would ensure that the data exchange between application entities satisfies total-order properties. These new types of stream could be called *total-order flexible real-time publisher/subscriber streams* (TOStreams).

V. Conclusions and future work

The FTT paradigm has been implemented over different communications systems, but until now there was no abstract specification of the service interface between FTT and the physical communication system over which it was implemented. This paper proposes a first such abstract specification in form of a layered architecture for FTT that comprises an application layer, an FTT layer, and a MAC layer for the underlying communication system. This architecture has then allowed us to add fault tolerance and total-order multicast to FTT by adding appropriate layers and "sandwiching" the FTT layer between them. The result of this is the layer architecture for FTTRS that was presented in this paper.

The presented layered architectures should serve as a good basis for further enhancements to FTT. For instance, the reliability layer might be modified to consider arbitrary network topologies and a node redundancy layer could be put on top of the FTT layer instead of the TOPS layer.

Future work includes further developing the details of the interface between the different layers we presented, as well as developing a formal specification (such as state machines) for the entities of each of the layers.

Acknowledgements

This work was supported by project DPI2011-22992 and grant BES-2012-052040 (Spanish Ministerio de economía y competitividad), by FEDER funding, and by the Portuguese government through FCT grant Serv-CPS PTDC/EEA-AUT/122362/2010. We also like to thank Luis Almeida and Paulo Pedreiras for their support during the early stages of this work, and Ricardo Marau for his helpful explanations on the inner workings of the FTT-SE code base.

References

[1] D. Gessner, J. Proenza, M. Barranco, and L. Almeida. "Towards a Flexible Time-Triggered Replicated Star for Ethernet." In *18th IEEE Int. Conf. on Emerging Technologies & Factory Automation*. Sept. 2013.

[2] 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,* page 9. IEEE Comput. Soc, 2001.

[3] ITU-T Recommendation X.200, "Information Technology - Open Systems Interconnection - Basic Reference Model.", ITU-T, 1994.

[4] "IEEE Standard for Ethernet," IEEE Std. 802.3-2012, 2012.

[5] D. Gessner, J. Proenza, M. Barranco. "A Proposal for Managing the Redundancy Provided by the Flexible Time-Triggered Replicated Star for Ethernet." In *Proc. of the 10th IEEE Int. Workshop on Factory Communication Systems (WFCS 2014)*, 2014, Toulouse, France.

[6] G. Rodriguez-Navas, 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 2013)*, 2013, Cagliary, Italy.

[7] D. Gessner, J. Proenza, M. Barranco. "A Proposal for Master Replica Control in the Flexible Time-Triggered Replicated Star for Ethernet." In *Proc. of the 10th IEEE Int. Workshop on Factory Communication Systems (WFCS 2014)*, 2014, Toulouse, France.

[8] A. Ballesteros, J. Proenza, D. Gessner, G. Rodriguez-Navas and T. Sauter. "Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star or Ethernet." In *Proc. of the 19th IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA 2014)*, 2014, Barcelona, Spain.