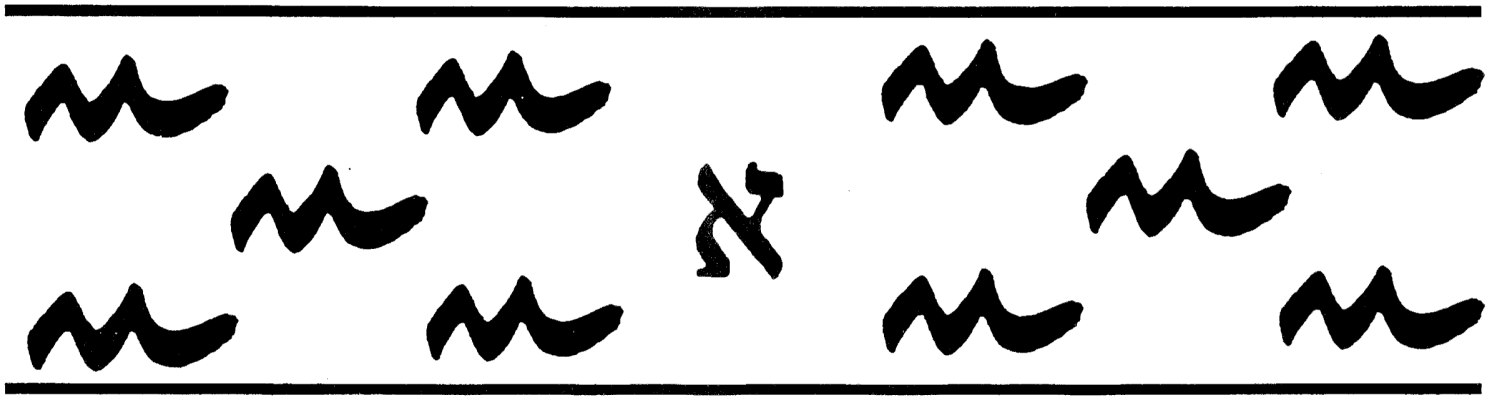
Universitat de les Illes Balears

Departament de Ciències
Matemàtiques i Informàtica

A description of the FTT-SE protocol

A. BALLESTEROS and J. PROENZA



TECHNICAL REPORT

December / 2013

A-06-2013

A description of the FTT-SE protocol

Alberto Ballesteros and Julián Proenza
DMI - Universitat de les Illes Balears, Spain
{a.ballesteros, julian.proenza}@uib.es

Abstract

The Flexible Time-Triggered (FTT) paradigm, proposed at the university of Aveiro in Portugal, makes it possible for distributed embedded systems (DES) to exchange real-time data in a flexible fashion, that is, it ensures that changes in the communication requirements can be managed. FTT has been implemented in different technologies and topologies. In this document we make a general description of the FTT Switched Ethernet (FTT-SE) protocol, an implementation of FTT on top of an Ethernet network with a star topology, that is, with a switch. To properly describe this protocol, we first discuss the FTT basics. Specifically, we explain how it organizes the communication, the kind of scheduling performed for the different types of traffics, the general design of the modules carrying out the protocol operation and other relevant features. After that, we focus on FTT-SE for which we expand all the topics discussed previously for FTT. Finally, we provide, in the form of appendixes, the list of the FTT-SE messages and an example scenario of one relevant FTT-SE mechanism.

Contents

1 Flexible Time-Triggered (FTT) basics

The Flexible Time-Triggered (FTT) paradigm [1] is a communication scheme proposed at the university of Aveiro in Portugal that makes it possible for Distributed Embedded Systems (DES) to exchange real-time data in a flexible fashion. That is, it ensures that changes in the communication requirements can be managed. For this, FTT implements various functionalities located in the second, third and fourth layers of the OSI model. In this sense, FTT must be deployed together with an existing network protocol implementing, at least, the first layer of the OSI model. Some examples of existing implementations can be found in Sec. 1.6.

The general FTT architecture, depicted in Fig. 1, is composed of a set regular nodes, called *slaves*, which are interconnected by a communication channel. Additionally, a special node called *Master* manages and coordinates the communication.

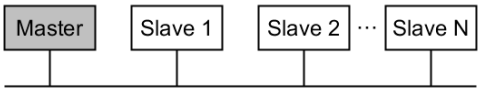


Figure 1: FTT architecture.

1.1 Elementary Cycle

The Master organizes the communication in fixed-duration time slots called *Elementary Cycles* (ECs) [4]. This makes it possible for DES application designers to easily map high-level application control loops to low-level communication.

As depicted in Fig 2, each EC is divided into two disjoint windows called *synchronous* and *asynchronous* windows. While the former is used to convey time-triggered traffic, herein called synchronous traffic because it is transmitted synchronously with the EC, the latter is used to convey event-triggered traffic, herein called asynchronous traffic because its transmission can be issued at any time. It is noteworthy that the relative position of these two windows depends on the specific implementation of the FTT protocol. More information about this last aspect can be found in Sec. 1.6.

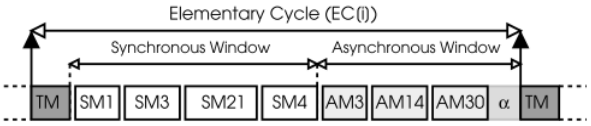


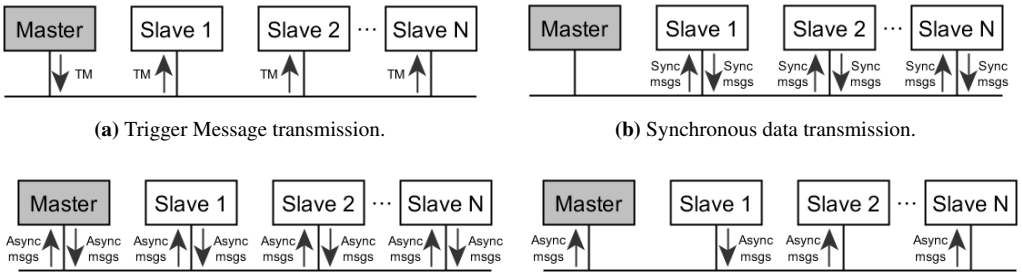
Figure 2: The Elementary Cycle structure (reproduced as it appears in [4]).

Each EC is triggered by the Master following a master-multislave scheme, by means of the so-called Trigger Message (TM). The purpose of this message is twofold:

- It marks the beginning of an EC and, thus, it is used to synchronize the slaves.
- It contains the EC-scheduling for the synchronous window, that is, it specifies the set of messages that slaves must transmit in said window.

The communication pattern defined by FTT for each EC is depicted in Fig. 3. First, the Master marks the beginning of the EC by sending the TM, which is received and decoded by slaves. After that, during the synchronous window, slaves exchange synchronous messages according to the EC-schedule contained in the TM. These messages carry data generated by the applications executed inside the slaves. After that, during the asynchronous window both, slaves and Master, exchange asynchronous messages, but now without any specific scheduling. It is noteworthy that these messages are not only used to transmit data from the applications, but also to manage the communication. All these types of messages are introduced later, in Sec. 1.2. Finally, note that slaves may consider that the EC is longer than how it really is. This is because they only synchronize when receiving the TM and, thus, when reaching the end of the EC they may be slightly desynchronized with the Master. In order to prevent EC overruns, that is, when a slave transmits after the end of the asynchronous window, an idle amount of time is reserved at the end of the EC. This gives some time to slower slaves to finish the transmission of their messages.

Finally, note that the size of the EC can be tuned to cope with the requirements set by the system. In this sense, longer ECs makes it possible to convey more data. That is, it decreases the weight of the overhead



(a) Trigger Message transmission.

(b) Synchronous data transmission.

(c) Asynchronous data transmission.

(d) Spare time at the end of the EC for slower slaves.

Figure 3: FTT communication scheme.

imposed by the time needed to transmit the TM and the idle time reserved at the end of the EC. In contrast, shorter ECs increases the responsiveness of the communication subsystem, as the transition between cycles is also shorter.

1.2 FTT messages

As can be seen in Fig. 4, messages that can be transmitted through an FTT network can be divided in two groups namely *data messages* and *FTT control messages*. Data messages are used by slaves to exchange synchronous and asynchronous application data. FTT control messages, in contrast, are used by both the Master and the slaves, and allow to manage the communication. This group of messages includes, apart from the TM, a set of asynchronous messages called *plug-and-play messages*, *slave request messages* and *Master command messages*.

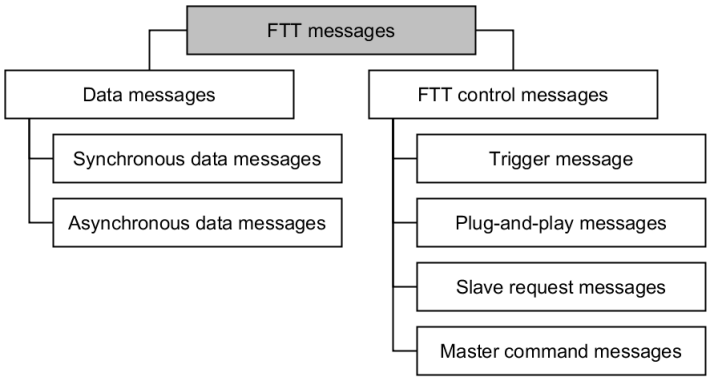


Figure 4: Types of messages in FTT.

First, plug-and-play messages are used to implement the plug-and-play mechanism. Specifically, a given slave willing to enter in the FTT network must first request permission to the Master. In case the access is granted, the Master responds by sending the current status of the network. This mechanism will be further discussed later, in Sec. 1.5. Second, slave request messages make it possible for slaves to request the Master for the modification of the communication requirements. Finally, Master command messages are used by the Master to respond to these requests by broadcasting the resources assigned to each kind of traffic.

1.3 FTT streams

FTT messages, both synchronous and asynchronous, are confined into virtual communication channels called *streams* [4, 1, 5, 2, 6]. As can be seen in Fig. 5, in FTT, nodes willing to communicate must be provided with a stream to which they can attach as transmitters or receivers. Note from the figure that, although various receivers can be attached to a given stream, FTT only allows the existence of one transmitter. In this sense, each stream is characterized by the type of traffic it conveys. The main benefit of this solution is that applications running on top FTT do not have to deal with the point-to-point communication. In fact, they are unaware of the other participants attached to the stream.

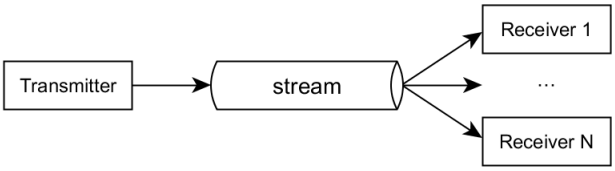


Figure 5: Stream-based communication scheme.

Streams are managed centrally by the Master, but all the operations are triggered distributively by the slaves. More precisely, they use slave request messages to request the creation, modification and removal of streams, as well as their attaching and detaching. First, a slave must request the creation of the stream by specifying the requirements of the communication. These requirements are defined as streams' attributes and are further described in Sec. 1.3.1 and 1.3.2. After that, the Master carries out an *admission control* to determine whether these requirements can be satisfied considering the available bandwidth and, if so, the stream is created. However, streams are initially disabled, as they do not have any slave attached. Slaves willing to communicate through this stream must notify the Master by sending an attaching request either as transmitters or receivers. Once the stream has a minimum number of transmitting and receiving slaves attached, the Master enables it. Moreover, the Master notifies this event to the slaves by sending a broadcast message informing all slaves about the existence of the new stream. Note that this message contains all the stream's attributes so slaves are able to know how to transmit and receive to and from said stream. When the communication is no longer needed, slaves may detach from the stream by notifying it to the Master. Conversely as in the enabling process, when the minimum number of attached transmitters and receivers is not fulfilled the Master disables the stream, and sends a broadcast notification message forcing the slaves to dismiss said stream. Finally, although multiple disabled streams may be stored inside the Master, slaves are allowed to remove them by means of a specific request.

As introduced in the previous paragraph, in FTT, the concept of stream is extended in order to take into account the temporal requirements of the data conveyed. In this sense, each stream has a set of attributes that describe both, the participants of the communication and the scheduling properties that are used by the Master to construct the scheduling for every EC. Note that these attributes are similar to the ones used to describe the temporal behaviour of tasks in real-time environments. For instance, it defines the transmission time, equivalent to the execution time, the period and the deadline. Moreover, this list of attributes depends on the type of traffic the stream conveys, that is, whether it is synchronous or asynchronous. Next we describe for each type of stream the set of attributes associated.

1.3.1 Synchronous streams

Streams conveying synchronous messages have the list of attributes shown in Def. 1. Note that this information is stored inside the Master, in the Synchronous Requirements Table (SRT). First, C_i specifies the transmission time of the message to be sent. The calculation of this value takes into account two factors. On the one hand, it involves the specific characteristics of the technology used. For instance, the transmission time will be shorter if Gigabit Ethernet is used instead of Fast Ethernet. On the other hand, in FTT, large messages are automatically divided in a sequence of fragments that are scheduled sequentially and individually by the Master. Apart from that, D_i specifies the relative deadline, T_i the period and O_i the offset. Note that these three attributes are expressed as a number of ECs and, thus, the deadline and the offset are relative to the EC specified in the period. Pr_i is a positive value specifying the priority of the stream, which is used by the scheduler to order the messages within the same EC. Finally, $*Xf_i$ is a pointer to an custom set of attributes that can be used to implement a specific QoS management policy. More information about this attribute, as well as its use, can be found in [4].

$$SRT = \{SM_i : SM_i = (C_i, D_i, T_i, O_i, Pr_i, *Xf_i), i = 1..N_s\} \tag{1}$$

C_i: Transmission time	D_i: Deadline (in ECs)
T_i: Period (in ECs)	O_i: Offset (in ECs)
Pr_i: Priority	$*Xf_i$: Pointer to a QoS attribute set

1.3.2 Asynchronous streams

The list of attributes of streams conveying asynchronous traffic is stored inside the Master, in the Asynchronous Requirements Table (ART). Note from Def. 2 that this list is similar to the one presented for synchronous streams. The only differences are the absence of offset and QoS attributes, as well as the presence of a minimum inter-arrival time ($Tmit_i$). This last value, equivalent to the period in a synchronous stream, is measured in ECs and defines the minimum number of ECs between two data transmissions.

$$ART = \{AM_i : AM_i = (C_i, Tmit_i, D_i, Pr_i), i = 1..N_a\} \tag{2}$$

C_i: Transmission time $Tmit_i$: Minimum inter-arrival time (in ECs)
D_i: Deadline (in ECs) Pr_i: Priority

1.4 Internal architecture of the FTT nodes

The internal design of the FTT nodes, depicted in Fig. 6, is constructed by means of a set of different interconnected modules [4, 5, 3]. Next, we list and describe all the modules conforming the internal architecture of the Master and the slaves.

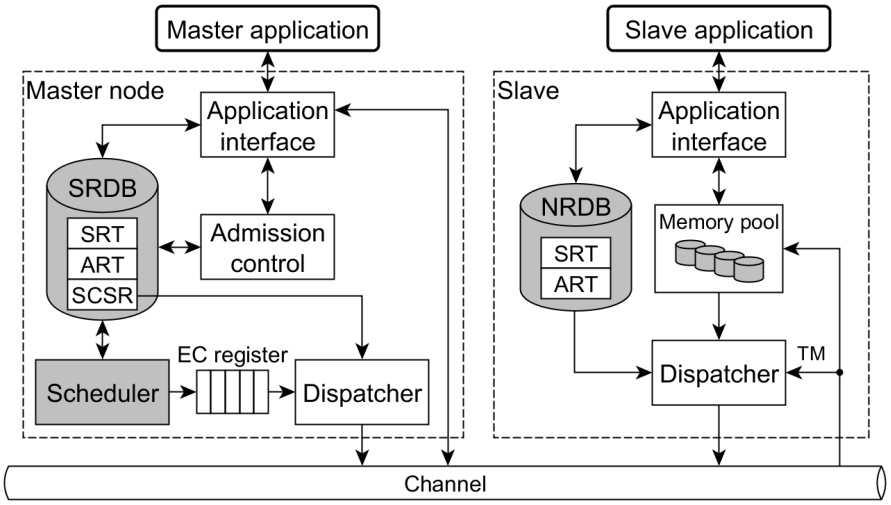


Figure 6: Master and slave internal architecture (based on a figure in [5]).

1.4.1 Master architecture

The internal design of the Master, shown in Fig. 6, makes it possible to coordinate the communication of the FTT network. First, the *Application interface* provides applications with a set of services to manage all the operation of the Master, like the management of the streams. Note from the figure, that these services can be accessed locally, from an application running on the Master, and externally, by an applications running on the slaves.

Second, the *System's Requirements Database* (SRDB) contains all the streams' attributes, as well as the Master operation parameters. On the one hand, as explained in Sec. 1.3.1 and 1.3.2, the SRT and ART store the attributes of the synchronous and asynchronous streams, respectively. On the other hand, the *System Configuration and Status Record* (SCSR) maintains all the data related to the configuration of the system, like the duration of the EC and the speed of the communication channel. The SRDB is managed by the Application interface, by means of the *Admission control* module, which supervises all modifications in order to keep the of streams schedulable.

Finally, the tuple consisting of the *Scheduler*, the *EC register* and the *Dispatcher* make it possible to compose and trigger the set of messages that slaves must transmit in each EC. The Scheduler scans every EC the SRDB in order to determine the set of messages that have to be transmitted. This list of messages are kept into the EC register until the beginning of the next EC. At this point, the Dispatcher constructs the TM, from the information provided by the EC register, and triggers the transmission of said message.

1.4.2 Slave architecture

The internals of the slaves are much simpler than the ones of the Master. This is because slaves just obey the instructions of the Master. The most relevant internal components of a slave are next described. First, similarly as in the Master, the *Application interface* provides applications with a set of services to manage and use the streams. On the one hand, this module allows requesting the creation, modification, removal of streams, as well as the attaching and detaching to/from them. On the other hand, applications may also request the transmission and reception of messages through a given stream.

Second, the *Node Requirements Database* (NRDB) is the SRDB counterpart in the slaves. Specifically, it stores all the streams together with their list of attributes. This database is managed by the Application interface, but the operations are triggered by the Master by means of FTT control messages.

Third, the *Memory pool* stores all the messages received and pending to be transmitted in a set of queues, each of which is associated with a stream. From the point of view of the reception, the Application interface needs to consult this pool for both, forwarding data messages to the application, and carrying out the appropriate modifications in the NRDB, according to the Master command messages. From the point of view of the transmission, the Application interface enqueues data and slave request messages coming from the application.

Finally, the *Dispatcher* is responsible for transmitting the messages according to the TM, in the adequate window. For this, the TM is captured and compared with the NRDB. More specifically, the Dispatcher identifies all those messages for which the slave is a transmitter. After that, messages are sent to the channel following the traffic separation defined by the EC.

1.5 Flexibility in FTT

The flexibility in the FTT protocol is brought thanks to various of its features [4, 2]:

- The Master includes an on-line scheduler that calculates dynamically the schedule for the synchronous traffic every EC, considering the current synchronous streams' attributes.
- Slaves do not need to be aware of the particular scheduling policy of each kind of traffic. They simply follow the schedule that the Master generates and later broadcasts by means of the TM.
- Streams are managed by the Master, but all the operations are triggered dynamically by slaves through the slave request messages. For this, the Master includes an Admission control module, which accepts or rejects those requests devoted to modify the stream attributes, considering the availability of its associated resources. This allows to assess the schedulability of new configurations before committing them.
- Slaves can connect to the FTT network without a previous knowledge of the state of the system. For this, the protocol defines a slave plug-and-play (PnP) process, in which the Master is responsible for registering them and for sending any update in the streams' attributes.
- The Master is able to efficiently manage the bandwidth assigned to the synchronous streams by means of a Quality of Service (QoS) approach. QoS guarantees both the minimum resources for all streams and the higher performance possible. For this, the FTT QoS implementation supports the definition of different levels of importance, which can be defined using different policies.

1.6 FTT protocol family

The FTT communication paradigm has been implemented in different technologies and topologies.

On the one hand, the FTT-CAN protocol [1] defines how FTT can be used on top of the Controller Area Network protocol. The most significant characteristics of this implementation are: the limited size of its TM, which devotes one bit to each of the messages that can be transmitted; and the placement of the synchronous window, which is situated at the end of the EC. This last feature makes it possible for low computation capacity slaves to use the duration of the asynchronous window to decode the TM.

On the other hand, FTT has also been implemented on top of the Ethernet protocol. More precisely, there are implementations for different topologies namely FTT-Ethernet [5] for a bus topology, FTT-Switched Ethernet [2] for a star topology with a legacy Ethernet switch and HaRTES [6] for a star topology with a custom switch. In this document we will focus on the FTT-Switched Ethernet implementation, which introduces several improvements with respect to the FTT-Ethernet proposal.

2 FTT-Switched Ethernet (FTT-SE) basics

Ethernet, due to its high bandwidth and wide use, is a promising alternative for implementing the communication subsystem of DES. However, the specific communication requirements set by these systems call for extensions in the protocol. The FTT-Switched Ethernet [2] communication paradigm achieves this by implementing FTT on top of a micro-segmented Ethernet network, that is, based on standard switches and with only one station connected to each port (see Fig. 7).

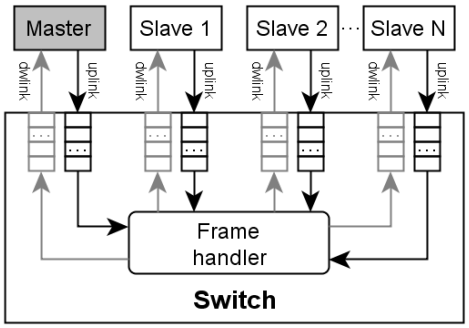


Figure 7: Switched topology.

The most relevant feature of the this topology, when compared to regular fieldbuses, is the existence of dedicated links for each node and transmission direction. Specifically, as can be seen in Fig. 7, each node is connected to the network by means of two separated links. On the one hand, the *uplink* connects the node to the switch, and is used to transmit. On the other hand, the *downlink* connects the switch to the node, and is used to receive. From the point of view of the protocol, this configuration results in two important benefits:

- *Absence of collisions*. This eases the implementation of both the Master and the slaves. More specifically, the Master does not need to enforce a collision-free medium access. Note that in FTT-Ethernet the scheduler has to specify the transmission instant of each message so no collisions occur. In FTT-SE, from the point of view of the slaves, they can transmit their pending synchronous messages immediately after decoding the TM. Likewise, asynchronous messages can be transmitted after sending the synchronous ones. The switch is the one responsible for serializing all these messages.

- *Parallel forwarding*. This provides each slave with dedicated transmission links, which can be exploited by the scheduler to increase the overall throughput. This can be done by building schedules that consider the messages following disjoint paths, that is, paths that have different source and destination nodes.

Fig. 8 depicts an EC as seen by a slave in each of its links (uplink and downlink). There are two important aspects to highlight from this figure. On the one hand, note that the existence of the uplink and downlink makes it possible for said slave to transmit messages as the same time as other messages are being received. For instance, during the transmission of the synchronous messages SM1, SM2, SM3 and SM4, the slave is able to receive SM5, SM6, AM4 and SM7. On the other hand, the arrangement of the windows depends on the messages conveyed and, thus, the set of windows dividing the EC is different in each link. The complete structure of the EC, as well as the traffic conveyed in each window, will be described later, in Sec. 3.

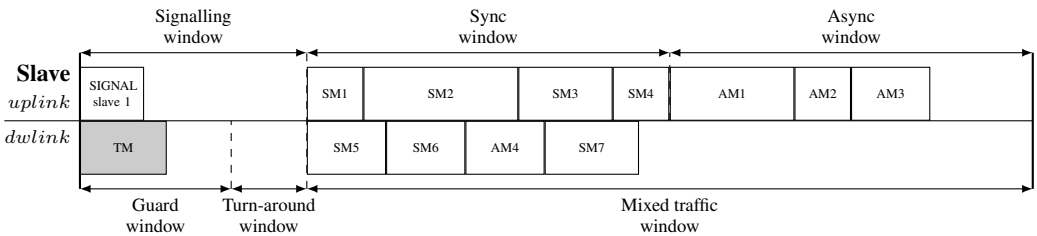


Figure 8: Elementary Cycle as seen by a slave.

3 The Elementary Cycle in FTT-SE protocol

As introduced in Sec. 1, in FTT time is organized in fixed duration slots called Elementary Cycles (ECs) which are, in turn, divided in different windows containing the different types of messages. However, in FTT-SE we have dedicated transmission links available for the Master and the slaves. Consequently, the arrangement and size of these windows is different depending on the point of view. More precisely, it depends on the specific node, and whether we look at the uplink or at the downlink.

Next, we describe the different EC schemes, in terms of window divisions, resulting for the Master and the slaves in both the uplinks and downlinks.

3.1 The Elementary Cycle in the uplink of the Master

From the point of view of the uplink of the Master, each EC is divided in three windows. These windows, depicted in Fig. 9, are described next.

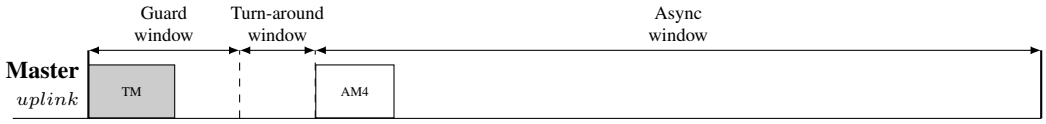


Figure 9: Structure of an Elementary Cycle as seen at the uplink of the Master.

- *Guard window*. It is the length of time reserved for the transmission of the TM. Its duration is pre-fixed and depends directly on the TM's maximum size, which is determined by the maximum amount of messages that can be exchanged within the EC.
- *Turn-around window*. It is the maximum necessary amount of time needed by the slaves to decode the TM. The size of this window is also pre-fixed and must be set considering the performance of the slaves. Note that a system containing low computation capacity slaves will demand a longer-lasting turn-around window, due to the longer time needed to decode the TM.
- *Asynchronous window*. It is used by the Master to transmit Master command messages. The duration of this window is equal to the remaining EC time. Note that the Master is not able to transmit synchronous messages (apart from the TM which has a dedicated window) and, thus, there is no synchronous window. In case these messages are added in the future, the corresponding synchronous window would be identical to the one implemented in the slaves (explained in Sec. 3.3).

3.2 The Elementary Cycle in the downlink of the Master

As concerns the downlink of the Master, each EC is divided in two windows, as shown in Fig. 10. Each of these windows is described next.

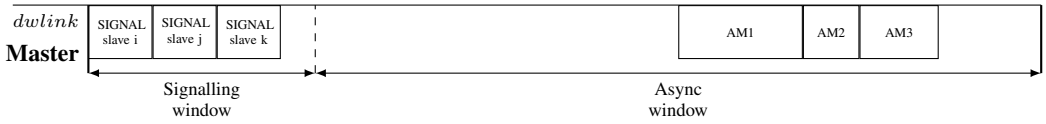


Figure 10: Structure of an Elementary Cycle as seen at the downlink of the Master.

- *Signalling window*. It is the length of time reserved for receiving all the messages related to the so-called signalling mechanism, explained later in Sec. 6.2.1. Note that, thanks to the parallel forwarding capabilities of FTT-SE, these messages can be received as the same time as the Master transmits the TM, without interference. The size of this window is pre-fixed and common to all the nodes in the system. Moreover, this size is equal to the sum of the guard and turn-around windows seen in the Master's uplink.
- *Asynchronous window*. It is used by the Master to receive all the slave request messages. Note from Fig. 10 that, since these messages are asynchronous, they are transmitted by the slaves after the synchronous ones and, thus, they are normally received some time after the end of the signalling window. More information about how slaves transmit messages can be found in Sec. 3.3. The position and duration of this window are equal to the ones set for said window in the Master's uplink.

3.3 The Elementary Cycle in the uplink of a slave

From the point of view of the uplink of a given slave, each EC is divided in three windows. These windows, depicted in Fig. 11 are described next.

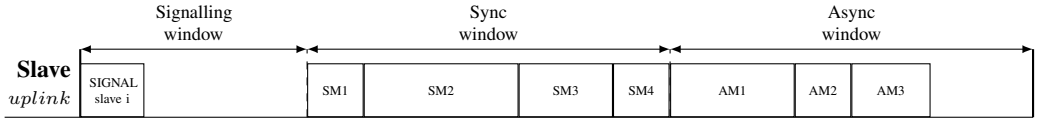


Figure 11: Structure of an Elementary Cycle as seen at the uplink of a slave.

- *Signalling window*. It is the length of time reserved for transmitting the message related to the so-called signalling mechanism, explained later in Sec. 6.2.1. Note that, thanks to the parallel forwarding capabilities of FTT-SE, this message can be transmitted as the same time the slave receives the TM, without interference. The position and size of this window are identical to the ones defined for the signalling window in the Master's downlink.

- *Synchronous window*. It is the time actually used by the slave to transmit synchronous data messages. The duration of this window depends on the amount of messages that are actually transmitted and, thus, it is different for each slave and EC.

- *Asynchronous window*. It is the remaining time until the end of the current EC, and it is used by the slave to transmit its pending asynchronous messages. Unlike the synchronous window, this window conveys different kinds of traffic, namely asynchronous data messages and slave request messages. It is noteworthy that the position and duration of the asynchronous window depend directly on the duration of the synchronous one and, thus, these values are different for each slave and EC.

To sum up, in the uplinks the size of the signalling window is the same for all the slaves. In contrast, the sizes of the synchronous and asynchronous windows depend on the specific slave and EC. More precisely, a given slave transmits first the synchronous messages, then the real-time asynchronous messages and finally, if there is still time, the non-real-time asynchronous ones. Note from this behaviour that synchronous messages have more priority than asynchronous ones, and thus, they may use the whole EC. To ensure that a minimum set of asynchronous messages can always be allocated in the EC, FTT-SE makes it possible to define an upper bound for the duration of the synchronous window.

3.4 The Elementary Cycle in the downlink of a slave

From the perspective of the downlink of a slave, each EC is divided in three windows. These windows, depicted in Fig. 12, are described next.

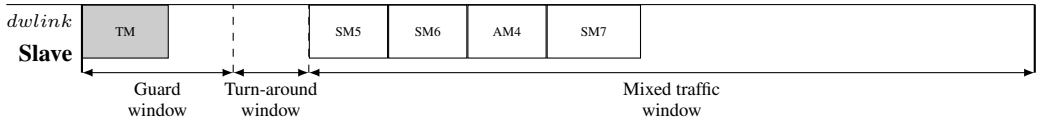


Figure 12: Structure of an Elementary Cycle as seen at the downlink of a slave.

- *Guard window*. It is the length of time reserved to receive the TM. Note that the position and size of this window are common to all slaves, and identical to the ones defined for said window in the Master's uplink.

- *Turn-around window*. It is the maximum necessary amount of time needed by the slave to decode the TM. The position and size of this window are also common to all slaves and equal to the ones described for said window in the Master's uplink.

- *Mixed traffic window*. It is the amount of time reserved to receive the messages transmitted by the Master and other slaves. Note from Fig. 12, that these messages are mixed, that is, there is no segregation between synchronous and asynchronous traffic as it did happen in the slave's uplink. This mixture is provoked by the different sizes of the synchronous and asynchronous windows among the different transmitters within the EC. This may happen, for instance, when a slave does not have

pending synchronous messages to transmit. This slave will send its asynchronous messages just after the signalling window, while another slave transmits its synchronous ones. All these messages will be forwarded by the switch to the downlinks sequentially, according to their order of arrival, without distinguishing whether they are synchronous or asynchronous.

4 FTT-SE messages

In FTT-SE, as in all FTT implementations on top of Ethernet, messages are encapsulated within Ethernet frames. More precisely, as can be seen in Fig. 13, the Ethernet frame has a header, a data section, and a footer. The header contains the source and destination MAC addresses of the frame, and the *ethertype*. The latter identifies the type of upper-layer message encapsulated in the data section. Since we currently only consider FTT messages to be exchanged over the network, the ethertype has for all Ethernet frames the value corresponding to FTT messages, that is, `0x8ff0`. The data section contains the payload of the frame, in this case, the FTT-SE message. Note that, due to the fragmentation mechanism described in Sec. 1.3.1, when a message, synchronous or asynchronous, is bigger than the maximum payload of the frame, the FTT protocol automatically divides the message in various fragments. Consequently, the data section can only contain part of the whole message. Finally, the footer carries a Cyclic Redundancy Check (CRC) code for the header and the data sections.

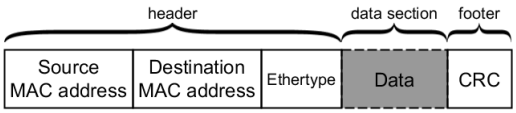


Figure 13: Ethernet frame structure.

In Sec. 1.2 we distinguished the different types of messages that the FTT paradigm defines. However, every FTT implementation defines its own set of messages, either by extending this set and/or redefining the content of some messages. As shown in Fig. 14, FTT-SE appends the so-called *queues status message*, that is, an FTT-SE control message that makes it possible for the slaves to inform the Master about the status of their asynchronous queues [2]. This message is related to the signalling described in Sec. 6.2.1.

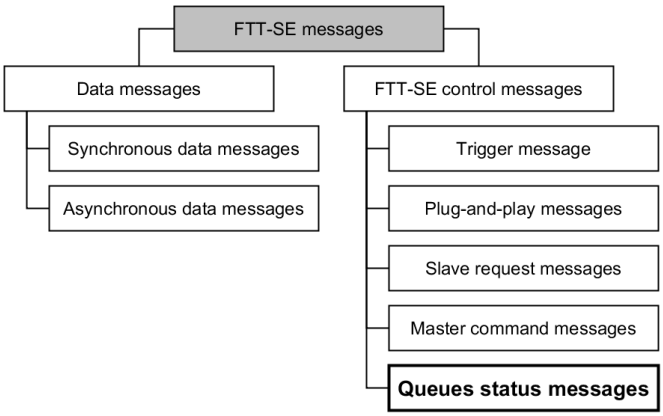


Figure 14: FTT-SE messages' division.

The details of all the messages defined in FTT-SE can be found in App. A. Specifically, it describes, for each type of message, its purpose, the stream through which it is sent, the related nodes and its internal structure.

5 FTT-SE streams

As introduced in Sec. 1.3, FTT provides a logical tool that makes it possible for the communication participants to exchange data without the need of knowing who is the transmitter or the receiver for said data. In FTT-SE the concept of stream is extended in order to take into account the dedicated transmission links. More precisely, we can move from a broadcast-based producer/consumer cooperation model in buses to a multicast publisher/subscriber cooperation model [2].

Once created, slaves willing to communicate through a given stream must attach to it. Specifically, transmitters attach as *publishers*, whereas receivers attach as *subscribers*. The attaching process comprises the registering of a new communication endpoint in the stream, which implies the reservation of some bandwidth either in the uplink or in the downlink. Note that, although the publisher/subscriber cooperation model does not impose any restriction on the number of entities that can attach to a given stream, in FTT-SE the number of publishers is limited to one.

As concerns the properties of the communications, FTT-SE extends the list of attributes of both synchronous and asynchronous streams to support the new cooperation model adopted. More precisely, as can be seen in Def. 3 and 4, two attributes are added. While S_i identifies the node (slave or Master) attached as publisher, $\{R_i^1..R_i^{k_i}\}$ identifies the set of nodes attached as subscribers.

$$SRT = \{SM_i : SM_i = (C_i, D_i, T_i, O_i, Pr_i, *Xf_i \boldsymbol{S_i}, \boldsymbol{\{R_i^1..R_i^{k_i}\}}), i = 1..N_s\} \quad (3)$$

$$ART = \{AM_i : AM_i = (C_i, Tmit_i, D_i, Pr_i, \boldsymbol{S_i}, \boldsymbol{\{R_i^1..R_i^{k_i}\}}), i = 1..N_a\} \quad (4)$$

Apart from the type of scheduling applied, FTT-SE streams can also be divided considering the purpose of the messages conveyed. On the one hand, *data streams* are those used to convey data generated at the application and, thus, they are only used by the slaves. On the other hand, *FTT-SE control streams* are used by both the Master and the slaves to exchange FTT-SE control messages.

Due to their interest when studying the operation of the protocol, all the FTT-SE control streams are next listed and described.

5.1 FTT-SE control streams

As introduced previously, FTT-SE control streams are used by the Master and the slaves to exchange information regarding the operation of the protocol [2]. Specifically, these streams are those devoted to carrying signalling and synchronization protocol messages. Next, we list and describe all the types of FTT-SE control streams that FTT-SE defines.

- *Master stream.* This is a hard-coded stream used by the Master to transmit broadcast messages to all slaves at the same time. The purpose of this stream is twofold. On the one hand, it is used by the Master to transmit synchronization messages devoted to inform the slaves about changes in the network status. For instance, when a stream is created, modified or removed. This synchronization mechanism is described later, in Sec. 7.1. On the other hand, the Master stream is also used by the Master during the PnP process when no other stream is available.

- *Signalling stream.* This is a hard-coded stream used by the slaves to transmit the signalling messages conforming the signalling mechanism. As will be explained in Sec. 6.2.1, this stream allows the slaves to transmit a single limited-size unicast messages every EC to the Master without the need of scheduling it. Additionally, this stream is also used by slaves during the PnP process when no other stream is available.

- *Dedicated Master-slave stream.* This stream is created during the PnP process of a slave and makes it possible for the Master to transmit unicast messages to the new slave.

- *Dedicated slave-Master stream.* This stream is created by the Master during the PnP process of a slave, and makes it possible for the said slave to transmit unicast messages to the Master. More specifically, the stream is used to convey slave request messages.

6 The scheduling in the FTT-SE protocol

In FTT the scheduling of the messages to be transmitted by the slaves in a given EC is carried out on-line and centrally in the Master. Then, the EC-scheduling is disseminated over the network by means of the TM. In FTT-SE, the TM contains the set of messages, both synchronous and asynchronous, that have to be transmitted during the EC. After receiving and decoding the TM, slaves transmit the pending messages according to the instruction contained in the TM.

In order to build the scheduling, the master needs some external information. Moreover, this information is different depending on the type of traffic, synchronous or asynchronous, that is considered. Next we describe, for each type of traffic, how the Master gets the information needed and how the scheduling process is carried out.

6.1 Synchronous traffic

Synchronous messages must be transmitted by the slaves according to time patterns, which are specified by means of the stream's attributes. For this, the Master constructs the synchronous schedule taking into account the size, period and priority of all synchronous streams.

Regarding the slaves, they all answer to the TM poll by transmitting all the pending synchronous traffic as soon as possible, that is, after the turn-around window. The precise transmission instants of the messages inside the synchronous window cannot be known as they depend on the speed of the slaves. However, the periodic traffic is transmitted in a burst, or nearly, with practically no wasted time between consecutive messages.

6.2 Asynchronous traffic

Asynchronous messages are generated from unpredictable events occurring in the slaves. Consequently, they cannot be scheduled automatically from the value of streams' attributes, as in the case of synchronous ones. In this sense, in order to build this scheduling the Master needs to regularly collect information regarding the existence of pending asynchronous messages from the slaves.

The procedure of informing the Master about the pending asynchronous messages is carried out by means if the so-called *signalling mechanism* explained in Sec. 6.2.1. Once the Master has gathered this information, the scheduling can be performed similarly as done for synchronous messages.

6.2.1 Signalling mechanism

The signalling mechanism [2] is a process in which every slave connected to the FTT-SE network regularly produces a message, the so-called queue status message (see App. A.8), containing the status of the queues of the asynchronous streams, both real- and non-real-time, for which the slave is a publisher. As introduced in Sec. 5.1, these unicast messages are transmitted to the Master through a hard-coded stream called *signalling stream*. This stream makes it possible for the slaves to transmit a single limited-size message every EC to the Master, without the need of scheduling it.

In order to provide all its features, this signalling stream uses the signalling window. That is, the length of time consisting of the guard and the turn-around windows, in which the Master's downlink and the slaves' uplinks remain unused. In this sense, this stream takes advantage of the parallel forwarding provided by the switched topology. Finally, note that, the lack of a need for a scheduling exhibited by this stream results from: a suitable tuning of the size of the signalling window and the constraint in the number and size of the messages sent through it.

There are two remarkable issues to take into account regarding the implementation of the signalling mechanism. On the one hand, since all the slaves' reports have to fit within the signalling window, its size indirectly limits the maximum number of slaves in the FTT-SE network. Therefore, this window has to be carefully scaled to cope with the amount of slaves to be connected. On the other hand, the signalling mechanism makes the system less reactive to aperiodic events. This is because it introduces a delay in the transmission of asynchronous messages. More precisely, it takes at least two ECs since an asynchronous message is generated until it can be transmitted.

7 The FTT-SE middleware

The FTT-SE implementation of an usual DES can be divided in layers. Specifically, at the lowest level there are a set of physical DES nodes interconnected by means of a communication channel, in this case, switched Ethernet. At the upper level each of these nodes executes an application, which implements the logic for said node. In between, the FTT-SE protocol provides a communication layer. More specifically, the FTT-SE protocol is implemented as a middleware [3], which confers two advantageous features to the implementation. On the one hand, in a distributed system, the middleware makes it possible for applications to be implemented independently from the physical node in which they will be executed. Basically, it offers a common application interface to all applications while all the internal complexity of the FTT protocol and its operation are hidden. On the other hand, the middleware provides a set of communication services beyond those provided by the Ethernet protocol.

A detailed view of the structure of the FTT-SE middleware can be seen in Fig. 15. The first important detail to highlight is the presence of the FTT network participants, that is, the Master and the slaves. On the one hand, the Master manages the traffic in order to fulfil the communication scheduling and QoS

requirements. On the other hand, the slaves implement the logic of the DES nodes. As concerns the relation between logical and physical components, it is important to remark that the implementation of FTT-SE is modular, which makes it possible for the Master and a slave to be executed in the same physical node. This, in turn, allows to save on costs, as there is no need to add one dedicated node for the Master.

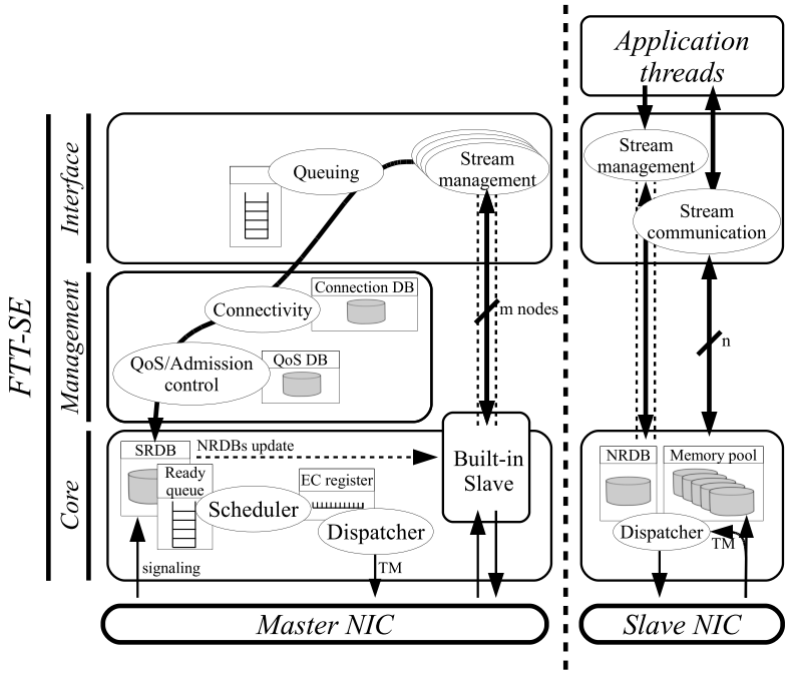


Figure 15: Detailed FTT-SE DES architecture (reproduced as it appears in [2]).

Another noteworthy element of the figure is the internal design of the FTT-SE middleware. As can be seen, it is divided in three layers called *Core*, *Management* and *Interface*. Next, we describe more in detail each of these layers.

7.1 Core layer

The Core layer contains the protocol basic communication mechanisms. In order to do so, this layer defines the concept of Core layer stream, or simply stream. As explained in Sec. 1.3, a stream is a tunnel used to transport information between a transmitter and a set of receivers. Additionally, each stream has an inherent scheduling and QoS properties that are kept by the protocol. Next we describe the two mechanisms that FTT-SE implements to manage the communication of streams, that is, the *traffic scheduling* and the *transmission control*,

On the one hand, the traffic scheduling is performed by Master, who constructs the scheduling for each EC from the information stored in the SRDB. More specifically, in each EC the Scheduler scans the SRDB and fills the *Ready queue* with the messages, both synchronous and asynchronous, pending to be transmitted by the slaves. After that, those messages that will be triggered in the next EC are selected. This scheduling is placed in the EC register, so the Dispatcher can later construct and send the TM. It is noteworthy that, as explained in Sec. 6, the traffic scheduling is slightly different depending on the type of the traffic. While the scheduling of synchronous traffic is carried out from the streams attributes, the scheduling of asynchronous traffic is carried out from the reports gathered from the signalling mechanism.

On the other hand, the transmission control involves both the Master and the slaves, and its purpose is to manage the transmission of the slaves by means of the TM. Specifically, the Master transmits the TM, which contains the scheduling constructed during the traffic scheduling process. After that, each slave receives, decodes and inspects said TM. This inspection involves the comparison of the TM's content against the NRDB. More precisely, the slave checks if it is the transmitter of any of the messages listed in the TM and, if so, the Dispatcher fetches the message from the Memory pool and orders its transmission. Finally, note that the FTT protocol contains a synchronization mechanism to keep NRDBs updated with the content of the SRDB. This mechanism is automatic, dynamic and ensures that streams' updates are simultaneously committed in the Master and the slaves.

The Core layer is also responsible for managing the slaves connected to the network. As explained in

Sec. 1.5, this is done by means of the PnP process. Specifically, a slave willing to enter in the network must be authorized by the Master. If so, the slave is provided with some necessary information. On the one hand, the Master assigns a *node id*, that is, an identifier for the physical component, that is bound to the MAC address of the Ethernet interface connected to the network. On the other hand, as explained in Sec. 5.1, the Master provides the slave with the dedicated Master-slave and slave-Master streams, as a mean to exchange FTT-SE control messages. This PnP process continues at the interface layer and, thus, it is further described in Sec. 7.3.

A complete description of all the phases conforming the PnP process can be found in App. B.

7.2 Management layer

The Management layer, only present in the Master, has a twofold purpose. On the one hand, it performs a high-level session control. That is, it maintains a database, the so-called *Connection DB*, containing all the streams together with the list of nodes attached to them. This is an interesting feature that allows applications to implement communications as interactions with the streams and, thus, to hide the participants of the communications.

On the other hand, the management layer regulates on-line the amount of resources assigned to each stream, based on the rules set by the application. The adjustments are controlled by a *QoS manager* that distributes the network capacity among the streams, based on the needs and importance of each. Note that, in order to do so this layer maintains a database, called *QoS DB*, with the current load of all the links.

It is noteworthy that, as explained in Sec. 1.3, the registering of new streams requires a previous checking of the available resources. In this sense, this layer includes the so-called *Admission Control* module, which is responsible for assessing whether there is enough bandwidth to allocate new streams. This module works closely to the QoS Manager, as it needs access to the QoS DB to know the available bandwidth, as well as to update the database in case a new stream is authorized to be allocated.

7.3 Interface layer

The Interface layer provides the applications with a common set of communication and management services. In order to do so, the Master needs to register the applications running in the slaves. This registering is part of the PnP process and, as explained in App. B, is triggered by the slave once it receives its node id. In this part of the PnP process the slave is provided with an application id, that is, an identifier of the application running on said slave. In this sense, at this level, FTT considers an application as a high-level abstraction of a slave. For this, this layer defines a namespace that maps applications ids to node ids.

Likewise, the communication services that this layer provides to the applications are possible thanks to the concept of *application stream*, that is, a high-level alias for the data streams. Specifically, the namespace defined in this layer also maps application streams to data streams. From the point of view of the application, application streams have the same attributes as defined for data streams, both synchronous and asynchronous.

This namespacing performed for applications and application streams, together with the high-level session control performed in the Management layer makes possible building location-independent code.

Finally, the management services provided by this layer allow applications to handle the application streams' lifecycle, that is, to create/modify/remove them, as well as to attach/detach to/from them. Since application streams are a high-level abstraction of the streams, this lifecycle is identical to the one presented in Sec. 1.3. Moreover, all the operations defined in this layer are mapped into their low-level implementation inside the Management and Core layer.

Appendix A List of FTT-SE messages

In this appendix we list and describe all the messages that can be sent through an FTT-SE network. This list of messages corresponds to the classification carried out in Sec. 1.2 and 4 and depicted in Fig. 14, in which we distinguished seven different types of messages. Note from the table of contents presented for this appendix that, although Slave PnP request and Application PnP request messages are introduced separately, they belong to the same type, that is, the Plug-and-play messages.

For each of the messages presented in this appendix we specify its name, its purpose, the stream through which it is sent, the related nodes and its internal structure. To describe the latter we first provide a figure depicting all the fields of the message. Note that the divisions performed in the fields represent each of the nibbles conforming said field. Finally, together with the figure we shortly explain the purpose of each of these fields.

FTT-SE messages appendix contents

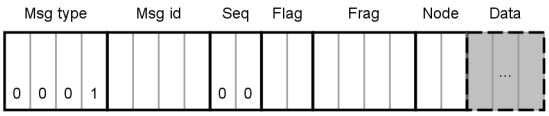
A.1 Synchronous data message

Description This message allows a slave to send synchronous data messages to other slaves.

Stream Synchronous data stream

Transmitter Slave

Receiver Slave



Message type Code of the message.

Message id Stream through which the message is sent.

Sequence Reserved for future use. Its value is always set to ”00”.

Flag Freshness of the data. It is used at the application level to know how old is the data carried and, thus, to know if the message is a repetition.

Fragment Number of fragment that is sent.

Node Node id of the slave sending the message.[1]

Data Data that is transmitted. It can be omitted if there is no data.

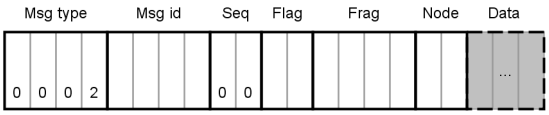
A.2 Asynchronous data message

Description This message allows a slave to send asynchronous data messages to other slaves.

Stream Application asynchronous stream

Transmitter Slave

Receiver Slave



Message type Code of the message.

Message id Stream through which the message is sent.

Sequence Number of message within the sequence. Since this message does not follow any sequence this value is always set to ”00”.

Flag Reserved for future use.

Fragment Number of fragment that is sent.

Node Node id of the slave sending the message.[1]

Data Data that is transmitted. It can be omitted if there is no data.

[1]This field is part of a mechanism that makes possible to increase reliability of the system. Specifically, multiple redundant publishers of the same information can be attached to a given stream and, if one fails, another can transparently assume the publishing role.

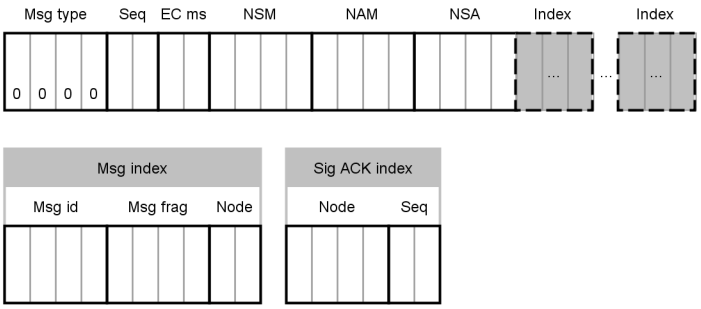
A.3 Trigger message

Description This message allows the master to specify the beginning of an EC. Moreover, it includes information about the messages to be transmitted in this EC.

Stream None

Transmitter Master

Receiver Broadcast



Message type Code of the message.

Sequence Number of trigger message within the sequence. When it reaches the "FF" value it is reset. Thus, it does not indicate the number of the current EC.

EC ms Duration of the EC expressed in milliseconds.

NSM Number of synchronous messages that will be sent in the EC.

NAM Number of asynchronous messages that will be sent in the EC.

NAM Number of signalling acknowledgements[1] that will be sent in the EC.

Indexes Information about the synchronous messages, asynchronous messages and signalling ack messages that will be send in the EC. For each message there is one index. The set of indexes are carried following the next order: indexes for asynchronous messages, indexes for synchronous messages and indexes for signalling ack messages. This field can be omitted if nothing has to be sent in the current EC.

Format of each message index

Message id	Stream through which the message will be sent.
Message fragment	Number of fragment that will be sent.
Node	Identifier of the node that will send the message.

Format of each signalling acknowledgement index

Node	Identifier of the node that will send the signalling message.
Sequence	Number of missing signalling message.

.

[1] This is part of the signalling mechanism that allows the retransmission of missing signalling messages (see App. A.8)

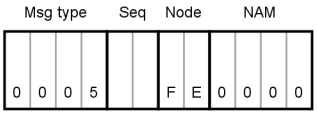
A.4 Slave PnP request message

Description Asynchronous message that allows a slave to request its admission in the FTT-SE network.

Stream Signalling stream

Transmitter Slave

Receiver Master



Message type Code of the message.

Sequence Number of signalling message within the signalling sequence. A missing signalling message can be detected by the master thanks to this value. When so, the master requests a retransmission by means of the trigger message.

Node id Node id of the slave. The slave can specify this value, however, the default value is "`FE`" which forces the master to assign the next available value.

NAM Since this message is constructed from the queues status message scheme, this field is mandatory. This value is always "`0000`" due to the fact that this type of message does not contain more information.

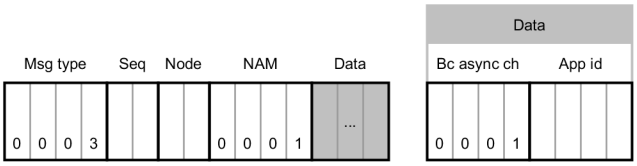
A.5 Application PnP request message

Description Asynchronous message that allows a slave to request the registering of a new application.

Stream Signalling stream

Transmitter Slave

Receiver Broadcast



Message type Code of the message.

Sequence Number of signalling message within the signalling sequence. A missing signalling message can be detected by the master thanks to this value. When so, the master requests a retransmission by means of the trigger message.

Node Node id of the slave sending sends the message.

NAM Since this message is constructed from the queues status message scheme, this field is mandatory. In this case it indicated the number of data blocks contained in the next field. This value is always "`0001`" due to the fact that this type of message always contains one data block.

Data Information about the application.

Data

Broadcast async channel	Stream id of the Master stream. It is hard-coded to be "`0001`".
Application id	Identifier of the application to be registered. Usually it is set to "`0000`" so the master can specify it later.

A.6 Slave request message

Description This message allows an application running inside a slave to send requests to the master.

Stream Dedicated slave-Master stream

Transmitter Slave

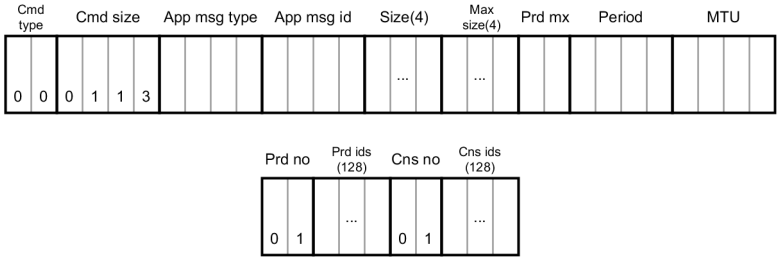
Receiver Master

Message type	Code of the message.
Message id	Stream through which the message is sent.
Sequence	Number of message within the sequence. Since this message does not follow any sequence this value is always set to "00".
Flag	Reserved for future use.
Fragment	Number of fragment that is sent.
Node	Node id of the slave sending the message.[1]
Command number	Number of requests carried.
Commands	Set of requests that are sent to the master. A given request can be of 11 different types.

[1] This field is part of a mechanism that makes possible to increase reliability of the system. Specifically, multiple redundant publishers of the same information can be attached to a given stream and, if one fails, another can transparently assume the publishing role.

A.6.1 Add full message

Description This message allows an application to request the creation of a new application stream. Note that this stream will not be available until a minimum set of publishers and subscribers are attached to it.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0113" bytes (275 in decimal)

App message type Type of traffic that the application stream will hold.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

Size Size in bytes of the data.

Max size Maximum size in bytes of the data.

Producer max Maximum number of publishers.

Period Number of ECs between data transmissions.

MTU Maximum Transfer Unit of a message's payload. Since we use Ethernet it should be set to 1500. However, note that the header of the message should be taken into account and, thus, it is usually set to 1450.

Producer number Predefined number of applications publishing the data. It is set to "1".

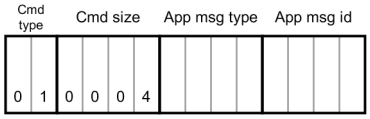
Producer ids Application ids of the set of publishers. Only the first application id is read.

Consumer number Predefined number of applications subscribed to the data. It is set to "1".

Consumer ids Application ids of the set of subscribers. Only the first application id is read.

A.6.2 Delete full message

Description This message allows an application to request the destruction of an existing application stream. Publishers and subscribers are not taken into account.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0004" bytes (4 in decimal)

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

A.6.3 Add/modify load requirements

Description This message allows an application to request the creation or modification of an application stream. Note that the application stream will not be available until a minimum set of publishers and subscribers are attached to it.

Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0011" bytes (17 in decimal).

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

Size Size in bytes of the data.

Max size Maximum size in bytes of the data.

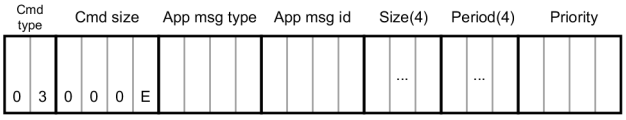
Producer max Maximum number of publishers.

Period Number of ECs between data transmissions.

MTU Maximum Transfer Unit of a message's payload. Since we use Ethernet it should be set to 1500. However, note that the header of the message should be taken into account and, thus, it is usually set to 1450.

A.6.4 Add/modify QoS requirements

Description This message allows an application to add or modify the QoS requirements of an existing application stream.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "000E" bytes (14 in decimal)

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

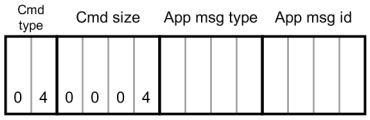
Size Size in bytes of the data.

Period Number of ECs between data transmissions.

Priority QoS priority for the stream. It is defined as a positive number. Greater value implies greater priority.

A.6.5 Remove load requirements

Description This message allows an application to request the detaching of the timing properties from an application stream. This leads to removing the stream from Core layer, but the connections remain in the Master Interface layer associating the application stream to the slaves.



Command type Code of the request.

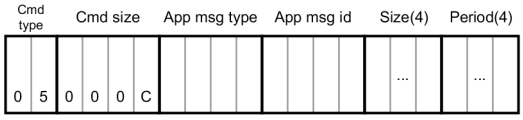
Command size Size in bytes of the content of the request. In this case the content has an occupation of "0004" bytes (4 in decimal)

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

A.6.6 Add QoS pair requirements

Description This message allows an application to request the modification of two QoS requirements, namely the size and the period, of an application stream.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0000C" bytes (12 in decimal)

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

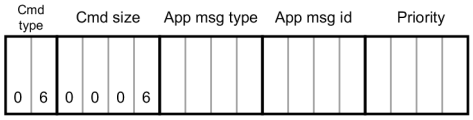
App message id Identifier of the application stream.

Size Size in bytes of the data.

Period Number of ECs between data transmissions.

A.6.7 Add QoS priority requirements

Description This message allows an application to request the modification of the priority of an application stream.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0006" bytes (6 in decimal)

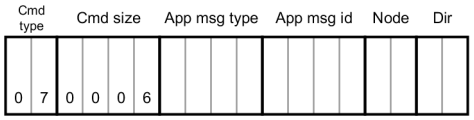
App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

Priority QoS priority for the stream. It is defined as a positive number. Greater value implies greater priority.

A.6.8 Add one end connection

Description This message allows an application to request the attaching to an application stream.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0006" bytes (6 in decimal)

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

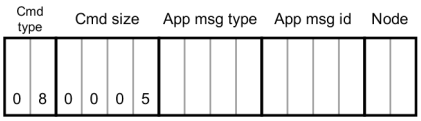
App message id Identifier of the application stream.

Node Identifier of the application.

Dir Role performed by the application.
"00" Publisher
"01" Subscriber

A.6.9 Delete one end connection

Description This message allows an application to request the detaching from an application stream.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0005" bytes (5 in decimal)

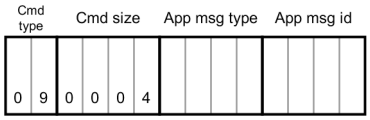
App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

Node Identifier of the application.

A.6.10 Get FTT id details

Description This message allows an application to request information about the an application stream. It is basically used to know what is the identifier of the stream bound to the application stream.



Command type Code of the request.

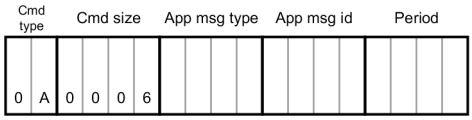
Command size Size in bytes of the content of the request. In this case the content has an occupation of "0004" bytes (4 in decimal)

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

A.6.11 Change period

Description This message allows an application to request the modification of the period of an application stream.



Command type Code of the request.

Command size Size in bytes of the content of the request. In this case the content has an occupation of "0004" bytes (4 in decimal)

App message type Type of traffic that the application stream holds.
"0000" Sync traffic
"0001" Async hard RT traffic
"0002" Async soft RT traffic
"0003" Async best-effort traffic

App message id Identifier of the application stream.

Period Number of ECs between data transmissions.

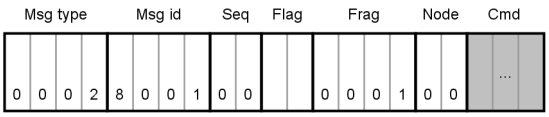
A.7 Master command message

Description This message allows the master to send commands to the slaves. It is mainly used keep the NRDBs consistent with the SRDB. However, it is also used in the PnP process.

Stream Master stream

Transmitter Master

Receiver Broadcast



Message type Code of the message.

Message id Stream through which the message is sent. This value always corresponds to the master stream identifier (”8001”).

Sequence Reserved for future use. Its value is always set to ”00”.

Flag Reserved for future use.

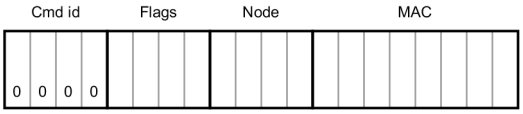
Fragment Number of fragment that is sent. Since none command overcomes the ethernet payload there is no frame fragmentation and, thus, this value is always set to ”0001”.

Node Identifier of the node transmitting the message. Since this message is always transmitted by the master its values is always set to ”00”.

Command Command that is sent to the slaves. This command can be of 6 different types.

A.7.1 Set node id

Description This command allows the master to set the node id of a specific slave during the PnP process.



Command id Code of the command.

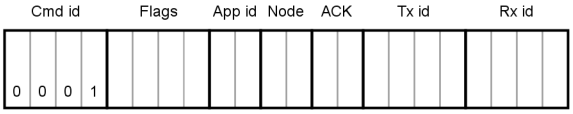
Flags Reserved for future use.

Node Identifier to be assigned to the slave.

MAC MAC address of the slave.

A.7.2 Set application id

Description This command allows the master to set the application id of a specific slave during the PnP process. Moreover, it also includes the code for the dedicated Master-slave and slave-Master streams.



Command id Code of the command.

Flags Reserved for future use.

Application id Identifier of the application.

Node Identifier of the slave.

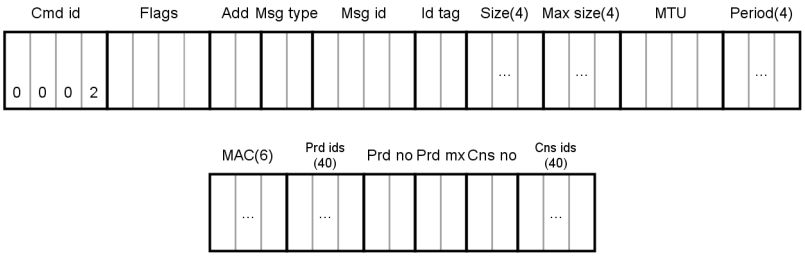
ACK Boolean value that indicates whether the application id could be carried out ("`01`"), or not ("`00`"). When not, it could be either because the application identifier was already assigned to a node, or any of the dedicated streams could not be created.

Tx id Identifier of the dedicated slave-Master stream.

Rx id Identifier of the dedicated Master-slave stream.

A.7.3 Add message

Description This command allows the master to inform that a new stream is available. That is, the stream has been registered and the minimum number of publishers and subscribers has been attached.



Command id Code of the command.

Flags Reserved for future use.

Adding Boolean value that indicates whether the information here specified must update the information already set for the stream ("00"), or the registering of a new stream must be forced ("01").

Message type Type of traffic that the stream will hold.
"00" Sync traffic
"01" Async traffic

Message id Identifier of the stream.

Id tag Identifier of the stream tag.[1]

Size Size in bytes of the data.

Max size Maximum size in bytes of the data.

MTU Maximum Transfer Unit of a message's payload. Since we use Ethernet it should be set to 1500. However, note that the header of the message should be taken into account and, thus, it is usually set to 1450.

Period When notifying a sync stream it contains the number of ECs between message transmissions. However, when notifying an async stream it contains the Tmit (minimum inter-arrival time).

MAC MAC address of the publisher.

Producer ids Node ids of the set of slaves producing the data. It is limited to 40 slaves.

Producer number Minimum number of publishers needed to make the stream available.

Producer max Maximum number of publishers.

Consumer number Minimum number of subscribers needed to make the stream available. In case it is set to 0, the stream is supposed to be broadcast.

Consumer ids Node ids of the set of slaves consuming the data. It is limited to 40 slaves. In case the "Consumer number" field is set to 0, the stream is supposed to be broadcast and, thus, this list of identifiers remains with a constant "0" value.

[1] This field is part of an FTT-SE mechanism that ensures a seamless modification of the stream attributes. More specifically, note that, when the Master authorizes the modification of the attributes of a given stream there could be pending messages in the slaves queues that have to be served according to the previous attributes. In this scenario, said stream will have two lists of attributes that coexist until all these pending are served. The tag field identifies the specific list of attributes associated with the stream.

A.7.4 Delete message

Description This command allows the master to inform that an existing stream is no longer available.

Command id Code of the command.

Flags Reserved for future use.

Message type Type of traffic that the stream holds.
"00" Sync traffic
"01" Async traffic

Message id Identifier of the stream.

A.7.5 Delete old tag

Description This command allows the master to inform that a specific stream tag will no longer be used.[1]

Command id Code of the command.

Flags Reserved for future use.

Message type Type of traffic that the stream holds.
"00" Sync traffic
"01" Async traffic

Tag id Identifier of the stream tag.

A.7.6 Dummy

Description This command allows the master to block the operation of the slaves. It is used to ensure that the master has enough time to update its local database, when a modification is performed.

Command id Code of the command.

Flags Reserved for future use.

[1] This message is part of an FTT-SE mechanism that ensures a seamless modification of the stream attributes. More specifically, note that, when the Master authorizes the modification of the attributes of a given stream there could be pending messages in the slaves queues that have to be served according to the previous attributes. In this scenario, said stream will have two lists of attributes that coexist until all these pending are served.

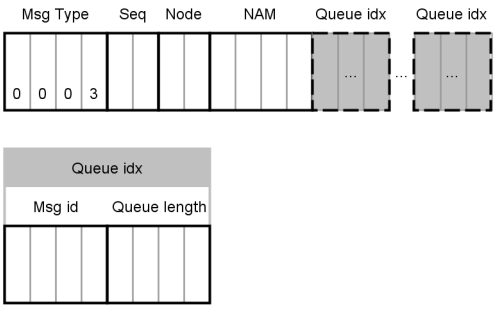
A.8 Queues status message

Description Signalling message that allows a slave to report the master about the current state of its asynchronous transmission queues.

Stream Signalling stream

Transmitter Slave

Receiver Master



Message type Code of the message.

Sequence Number of signalling message within the signalling sequence. A missing signalling message can be detected by the master thanks to this value. When so, the master requests a retransmission by means of the trigger message.

Node Node id of the slave sending the message.

NAM Number of queues having pending messages to be transmitted. It indicates the number of queue indexes contained in the next field.

Queue indexes Information about the state of the node's queues. Specifically, for each queue there is one queue index.

Queue index

Message id	Stream for the corresponding queue.
Queue length	Number of asynchronous messages for said stream that are pending to be transmitted.

Appendix B Slave plug-and-play scenario

In this scenario one slave requests the Master for the right of entering in the FTT-SE network. In response, the Master assigns a node id and an application id to the slave, as well as it creates the dedicated slave-Master and Master-slave streams. As explained in Sec. 5.1, these two streams are used by the Master and the slave to exchange FTT control messages. Next, we describe all the four ECs within which this handshake is carried out. Note that, for each EC we include a digram showing the interactions among the Master and two nodes; one requesting the plug-and-play (*Slave i*) and another one behaving like an idle slave (*Slave n*). The presence of this last slave helps to understand which messages are received and transmit by a non-related slave. Each of these diagrams distinguishes the different windows of the EC according to what is transmitted by the Master, that is, from the point of view of Master's uplink.

Once the slave is physically connected to the network, it is able to receive the TMs transmitted by the Master. In this sense, it is able to synchronize with the FTT-SE network and acquire the master's MAC address. What happens next, can be seen in Fig. 16. Specifically, at the beginning of the EC, *Slave i* sends a slave PnP request message (see App. A.4) via the signalling stream. This message is received by the Master together with the queues status messages transmitted by the already connected slaves, here represented by *Slave n*. Simultaneously, the Master, apart from transmitting the TM, registers the slave request. The response of the Master to this request takes some time and, thus, the PnP process continues in the next EC.

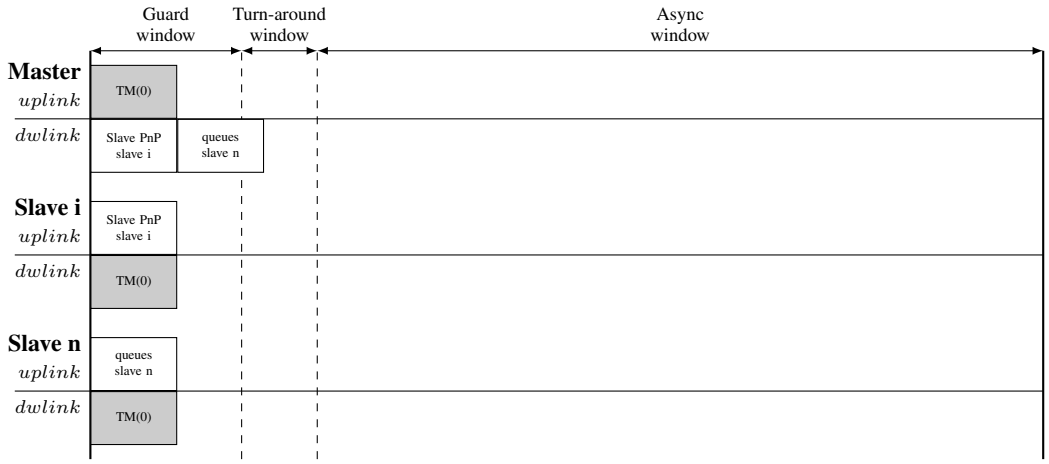


Figure 16: First EC of the slave plug-and-play scenario. The slave sends a slave plug-and-play request.

The sequence of events in the next EC can be seen in Fig. 17. First, *Slave i* sends the slave PnP request message again. This is because slaves repeat this behaviour until the Master assigns a node id to it. At the same time, the Master sends a TM notifying the slaves about its will to transmit a message, the set node id message (see App. A.7.1), through the Master stream. Later on, after the turn-around window, the Master sends this message which, thanks to a specific field containing the destination MAC address, is accepted and processed just by *Slave i*.

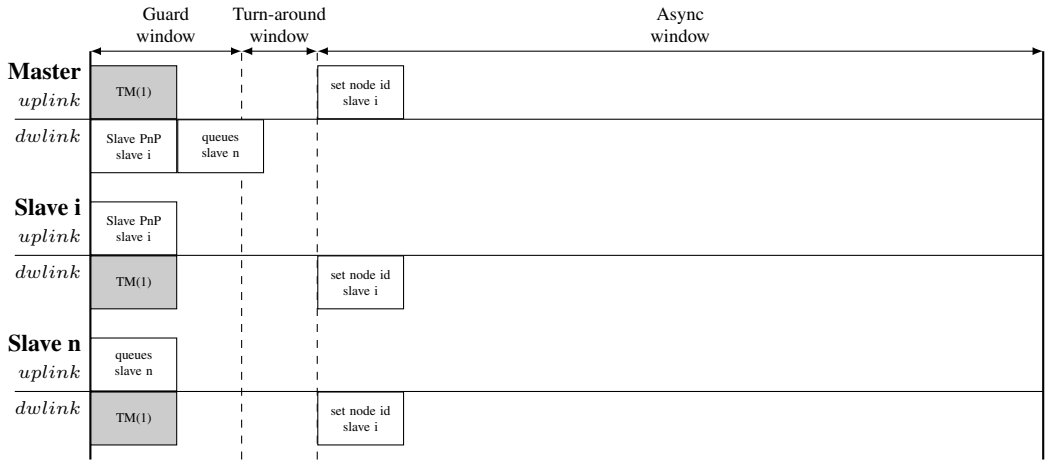


Figure 17: Second EC of the slave plug and play scenario. The Master responds by assigning a node ids.

Once the *Slave i* is registered as an active node, it should register its application. For this, as can be seen in Fig. 18, *Slave i* sends an application PnP request (see App. A.5). This message is very similar to the slave PnP request message in terms of its internal structure. Moreover, it is also sent through the signalling stream, as the dedicated Master-slave and slave-Master streams have not been yet created. Again, the response of the Master takes some time to be generated and, thus, it is carried out in the next EC.

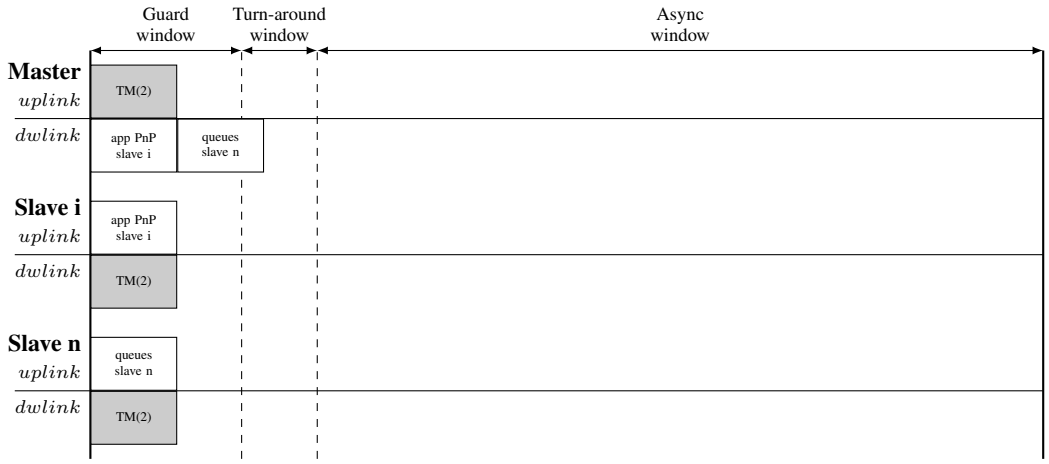


Figure 18: Third EC of the slave plug and play scenario. The slave sends an application plug-and-play request.

Finally, in the fourth EC, shown in Fig. 19, the Master finishes the handshake by sending the application id assigned and the stream ids of the dedicated slave-Master and Master-slave streams. On the one hand, as can be seen in the guard window, *Slave i* now is able to inform the Master about the status of its asynchronous queues. As concerns the Master, it notifies its will to transmit four asynchronous messages through the master channel.

On the other hand, after the turn-around window, the Master transmits the sequence of broadcast messages listed next. The first two messages, the so-called add messages (see App. A.7.3), inform *Slave i* about the creation of two streams, the dedicated slave-Maser and Master-slave streams. The third message, called dummy message (see App. A.7.6), forces the slaves to wait until the Master local database is updated. Finally, the set application id message (see App. A.7.2) informs *Slave i* about its assigned application id. Note that, this last message is unicast, however, it is sent in broadcast just like the other previous messages. As a mean for the slaves to accept or reject it, this message contains the node id of the receiver, in this case the node id of *Slave i*.

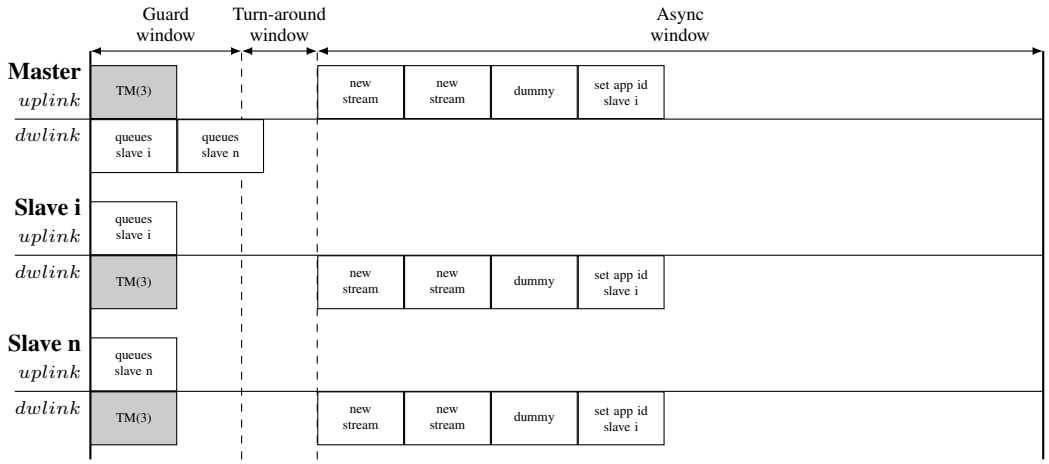


Figure 19: Fourth EC of the slave plug and play scenario. The Master responds by assigning an application id and the two dedicated Master-slave streams.

References

[1] L. Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN protocol: why and how. *IEEE Transactions on Industrial Electronics*, 49(6):1189–1201, December 2002.

[2] R. Marau. Real-time communications over switched Ethernet supporting dynamic QoS management. 2009.

[3] R. Marau, L. Almeida, M. Sousa, and P. Pedreiras. A middleware to support dynamic reconfiguration of real-time networks. *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, pages 1–10, September 2010.

[4] 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*. IEEE Comput. Soc, 2001.

[5] P. Pedreiras, P. Gai, L. Almeida, and G. C. Buttazzo. FTT-Ethernet: a flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems. *IEEE Transactions on Industrial Informatics*, 1(3):162–172, 2005.

[6] R. Santos. *Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications*. PhD thesis, Universidade de Aveiro, 2011.