Universitat de les Illes Balears

Treball Final de Grau

GRAU D'ENGINYERIA ELECTRÒNICA INDUSTRIAL I AUTOMÀTICA

Implementation and Validation of the Fundamental Mechanisms of the Flexible Time-Triggered Communication Paradigm for Ethernet-based Highly-Reliable Systems

SERGI ARGUIMBAU GUARINOS

Tutor
Alberto Ballesteros Varela

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, October 2017

Quiero agradecer este trabajo de final de grado a mi familia por todo el apoyo que me han dado. A mi madre por su cariño y amor que me da fuerzas para continuar. A mi padre por guiarme siempre por el mejor camino para llegar hasta el objetivo. Y a mi hermana por su constante apoyo moral.

En especial quiero agradecer a mi tutor Alberto por todas las ayudas que me ha ofrecido para completar el TFG. Sobre todo agradecer todo el tiempo que me ha podido dedicar para responder a todas mis dudas. Agradeciendo cada una de sus explicaciones detalladas de los conceptos técnicos que tengo que conocer. Y por su puesto su ayuda a la hora de resolver los problemas y cuestiones que han ido surgiendo durante el TFG.

Contents

LIST OF FIGURES

Acronyms

ADM	Asynchronous Data Message
AM	Asynchronous Message
AW	Asynchronous Window
CAN	Controller Area Network
DB	DataBase
DECS	Distributed Embedded Control Systems
DFT4FTT	Dynamic Fault Tolerance for Flexible Time Triggered
EC	Elementary Cycle
FTT	Flexible Time-Triggered
FTT-CAN	Flexible Time-Triggered Controller Area Network
FTT-SE	Flexible Time-Triggered Switched Ethernet
HaRTES	Hard Real-Time Ethernet Switching
MCM	Master Command Message
SM	Synchronous Message
SRM	Slave Request Message
SW	Synchronous Window
SDM	Synchronous Data Message
TM	Trigger Message

Abstract

Ethernet is nowadays the most widespread communication standard for local networks in the domestic and office environment. Its main advantages are: high bandwidth, low price of its components and compatibility with other communication standards. For that reason it is considered interesting to use Ethernet in industrial systems.

Industrial systems have additional requirements not present in domestic or office environments. Specifically, they have real-time and dependability (reliability, availability and/or security) requirements. In addition, it is not uncommon that this kind of systems are deployed in dynamic environments, that is, environments where the operational conditions can change unexpectedly. Unfortunately, Ethernet by itself does not provide the necessary services to fulfil all these requirements.

To overcome this limitation, the Dynamic Fault Tolerance for Flexible Time Triggered (DFT4FTT) project aims at providing a complete infrastructure to support applications with real-time, reliability and adaptivity requirements. Specifically, the DFT4FTT architecture is based on the Flexible Time-Triggered (FTT) communication paradigm. FTT makes it possible to exchange periodic and aperiodic traffic with different criticality levels in a real-time manner. Moreover, it allows to modify the real-time attributes of the traffic dynamically. The DFT4FTT architecture modifies FTT to achieve high reliability by means of fault-tolerance mechanisms. This is done by replicating the network and the nodes.

The main problem when implementing the DFT4FTT architecture is that FTT was not designed having fault tolerance in mind. Moreover, fault tolerance mechanisms are typically not orthogonal to the operation of the system. Consequently, it is very costly to extend the FTT software to include these mechanisms. In this regard, it was decided to implement FTT from a new design which removes unnecessary and non-reliable functionalities, and makes room for the new fault tolerance mechanisms.

This project represents the first step towards a new implementation of FTT for highly-reliable systems. Specifically, this project consisted in the implementation and validation of a basic FTT network which can then be easily extended to implement the necessary fault tolerance mechanisms.

CHAPTER 1

Introduction

1.1 Background and motivation

Nowadays Ethernet is the most widespread communication standard for local networks in the domestic and office environment. The main advantages [1] that it presents are: high bandwidth, low price of its components and compatibility with other communication standards. For that reason it is considered interesting to use Ethernet in other areas, like industrial systems. An example of these systems are the Distributed Embedded Control Systems (DECS), where a set of interconnected devices with specific purpose hardware cooperate to control a certain system. Some examples of DECS are: the control system of an aircraft or a car, the automatic assembly chain in factories and the domotic systems in a building.

Industrial systems have additional requirements not present in domestic or office environments. Specifically, they have real time and dependability (reliability, availability and/or security) requirements. In addition, it is not uncommon that this kind of systems are deployed in dynamic environments, that is, environments where the operational conditions can change unexpectedly. Unfortunately, Ethernet does not provide services to fulfil these requirements. To take advantage of the benefits Ethernet can provide, complying with the requirements of the new environments, new Ethernet-based protocols and standards have been developed. An example is the Flexible Time-Triggered (FTT) [2], a master/multi-slave communication paradigm, which makes it possible for the nodes of a DECS to exchange real-time traffic. In particular, FTT supports the transmission of periodic and aperiodic messages with different criticality levels. Moreover, the flexibility of FTT comes from the fact that the real-time attributes of the traffic can be dynamically modified by requests issued by the nodes.

In FTT, the communication is divided in several fixed-duration time slots called Elementary Cycles (ECs). Each EC starts with the master transmitting the Trigger Message (TM). The purpose of this message is twofold. On the one hand, it marks the start of the EC. On the other hand, it contains the EC-schedule, which is the set of data messages slaves must transmit in said EC. In response to the TM slaves transmit the requested messages. The source and destination of the data messages are applications executed inside the slave nodes.

FTT by itself does not provide highly-reliability and thus, cannot be used to implement systems with reliability requirements. To overcome this limitation, the Dynamic Fault Tolerance for Flexible Time Triggered (DFT4FTT) [3] project aims at adding dynamic fault tolerance mechanisms on top of FTT. The DFT4FTT architecture is based on the Hard Real-Time Ethernet Switching (HaRTES) [4], a switch-Ethernet implementation of FTT in which all the slaves are interconnected by means of a custom software switch embedding the master. As can be seen in figure 1.1, two of the most important modifications performed on HaRTES to achieve high reliability were the replication of the network and the slaves. On the one hand, to tolerate permanent errors affecting the network the switch is replicated. On the other hand, to tolerate permanent errors affecting critical nodes the corresponding slaves are also replicated.

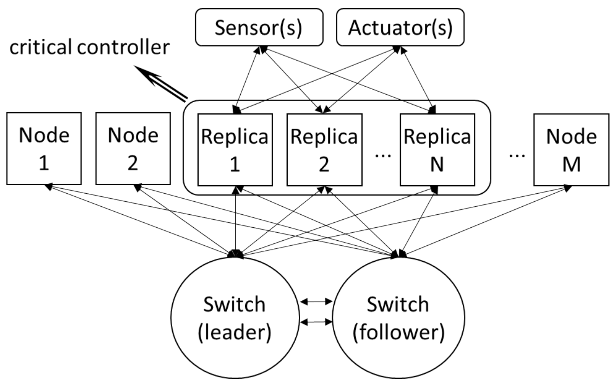


Figure 1.1: DFT4FTT architecture (reprinted as it appears in [5] Fig. 1)

HaRTES was not designed having fault tolerance in mind. Moreover, fault tolerance mechanisms are typically not orthogonal to the operation of the system. Consequently, it is very costly to extend the FTT software to include these mechanisms. In this regard, the main responsible for the development of the DFT4FTT architecture have decided to implement HaRTES from a new design which removes unnecessary and non-reliable functionalities, and makes room for new fault tolerance functionalities.

1.2 Project objective

The purpose of this project is to implement in software a basic FTT network, which can be easily extended to implement the necessary fault tolerance mechanisms. With basic we mean that only the core functionalities of the master and slaves are implemented. The implementation of low-level communication mechanisms and the dynamic features of DFT4FTT are out of the scope of this project. On the one hand, the HaRTES custom switch is not implemented. Instead, to prototype we use the Flexible Time-Triggered Switched Ethernet (FTT-SE) configuration [6], that is, the master and nodes are interconnected by means of a legacy switch. On the other hand, the real-time attributes of the traffic are predefined and kept static.

Specifically, this bachelor thesis has three goals:

- Implement a basic FTT master capable of storing the traffic requirements of the slaves for the periodic messages, as well as producing and transmitting the TMs at the beginning of every EC with the correct EC-schedule.
- Implement a basic FTT slave capable of receiving, decoding and processing the TMs in order to transmit the requested periodic messages.
- Implement an application interface so that the application in a given slave can deliver its messages for transmitting, as well as receive the messages addressed to said application.

The outcome is a functional prototype with one master and several slaves each one executing a specific application devoted to transmit or receive messages.

1.3 Realized tasks

In this section we detail all the tasks that were done to accomplish with the project objective:

Familiarize with the concepts of FTT and FTT-SE

The first step was to study several documents about the FTT basics [2, 6] and some of its main concepts: the Elementary Cycle (EC), the utility of the Trigger Message (TM), and periodic and aperiodic messages. All these concepts will be explained further in this document in the following section 2.

Prepare a virtual environment for communication

Instead of using a legacy switch and physical node connections, it was possible to test all the functioning in a single computer thanks to a virtual switch [7] set up in a Linux environment. The virtual switch is easily modifiable to test different cases and in overall it facilitates this work.

Create and define a stream database

The stream DataBase (DB) provides full control of the messages that are scheduled in the system. Due to using a static database in this project, the same stream DB structure was defined in the master and the slaves.

Software implementation of a reduced master

The logic of the master was implemented progressively during the working period of this project. As some general files were inherited from a previous work, it was important to understand how to use them and then if necessary, modify some parts.

Software implementation of a reduced slave

Once we gained the needed experience with the master implementation previously described, our objective was to implement the slaves. The logic of the slaves is more complex than the one from the master because they do several actions at the same time. Slaves have to receive the Trigger Message (TM) sent by the master, process it and then if required, send Synchronous Data Messages (SDMs) to other slaves. Another difficulty was that slaves have to interact with custom applications they execute.

Create an interface between an application and a slave

As we said before, the slaves have to interact with custom applications they execute. To accomplish this, we provide them with inter middle buffers that store the data of the streams they have to interact with.

Create simple applications to test the project

Every slave executes a custom application that represents its role in the system. These applications could be individual tasks such as the ones of a control system: Sampling, Control and Actuation. Two applications were created to test this project. The first one simulates the sampling of a sensor by generating and transmitting random data. The second one simulates a control application by receiving outputting the data received. To simplify it, the output will just be showing the data received as we are only interested in the correct transmission and reception of the messages.

CHAPTER 2

Previous work

In this section we explain briefly the previous work which constitutes the basis for the developing of the current project. Specifically, we will explain more in detail the basics of the Flexible Time-Triggered (FTT) communication paradigm.

The FTT paradigm, proposed at the University of Aveiro (Portugal), is a communication model that supports the exchange of data in a flexible manner. For this purpose, FTT provides mechanisms to ensure that the communication requirements can be managed dynamically.

The FTT architecture follows a master/multi-slave scheme (see figure 2.1). There is a master node which manages and coordinates the communication. The rest of the nodes are the slaves, which follow the master instructions to exchange data through streams.

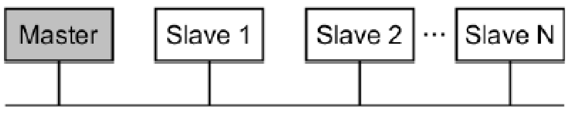


Figure 2.1: FTT architecture (reprinted as it appears in [6] Fig. 1)

Nodes perform their communications by means of virtual communication channels called streams. A particular stream has one publisher and several subscribers (see figure 2.2). The publisher node is the only transmitter of the stream; and the nodes that receive the data containing that stream are the so-called subscribers. Each stream is identified with a stream ID and has a set of real-time attributes associated: period, offset, deadline, priority...

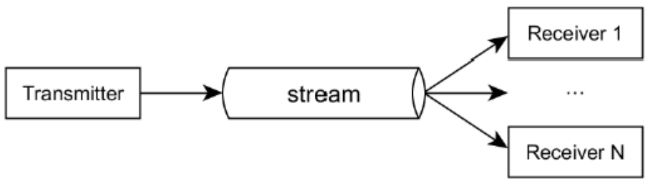


Figure 2.2: Stream-based communication scheme (reprinted as it appears in [6] Fig. 5)

The master organizes the communications in time slots called Elementary Cycles (ECs). The start of the EC is marked by the master with the Trigger Message (TM), which contains the list of messages that the slaves have to exchange between each other. The messages sent by the slaves can be either transmitted in the Synchronous Window (SW) or the Asynchronous Window (AW), depending if they are time-triggered or event-triggered, respectively. We can observe the structure of a EC in the figure 2.3:

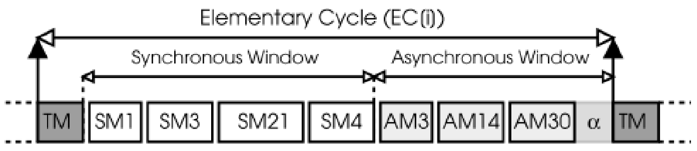


Figure 2.3: Elementary Cycle structure (reprinted as it appears in [2] Fig. 1)

The messages appearing in the SW are periodic and are called Synchronous Messages (SMs). In contrast to the messages appearing in the AW, which are aperiodic and are called Asynchronous Messages (AMs). All the SMs are Synchronous Data Messages (SDMs), that is the messages used by the slaves to convey periodic application data. Regarding the AMs, they can be divided in two types of messages. On the one hand, the Asynchronous Data Messages (ADMs) are the messages used by the slaves to convey aperiodic application data. On the other hand, the AMs can also be control messages, such as Slave Request Messages (SRMs) or Master Command Messages (MCMs). The SRMs are messages sent by the slaves to notify the master about the modification of the communication requirements. The master receives the SRMs and decides whether to accept or reject these changes depending on the availability of the resources (bandwidth, duration of EC...). If the changes are accepted the master responds with a MCM, forcing the slaves to modify their local stream databases with the new configuration.

Regarding the scheduling of periodic messages, each EC is identified with a sequence number, included as a part of the TM. This sequence number is used by the master to schedule the messages that will be sent in the current EC. Specifically, streams have a period and an offset. With the period, we define the periodicity in ECs with messages are transmitted; and with the offset, we are able to delay the transmission of the messages to a specific EC. The master scheduling condition formula is the one appearing in figure 2.4:

$$num\_EC \; \% \; T_i - O_i == 0$$

T_i: period of stream i

O_i: offset of stream i

Figure 2.4: Master scheduler formula

Finally, FTT can be used on different communication protocols. It has been already defined for the Controller Area Network (CAN), the Flexible Time-Triggered Controller Area Network (FTT-CAN) [8] and another approach for Ethernet with the FTT-Ethernet [9]. The more relevant FTT implementation, based on Ethernet, are FTT with switched-Ethernet, that is FTT-SE [6], which avoids message collisions thanks to a legacy switch that serializes all the messages; and HaRTES [4], which is the evolution of FTT-SE, integrating the master with the legacy switch to improve its global network synchronization, among other benefits. This project is focused on a basic FTT-Ethernet to further implement the fault tolerance mechanisms that will provide high reliability in the project DFT4FTT.

CHAPTER 3

DESIGN

In this section we will show the designed scheme of the master and the slave, and also explain how they work. Some important notes on this section are that most of the design was specified by the people that are behind the main project, which is DFT4FTT. Part of my work was to discuss this design, which can be found in the Annex A, and adapt it to the requirements of my project goal.

3.1 Master

In first place we will show the designed scheme of the Master (see figure 3.1):

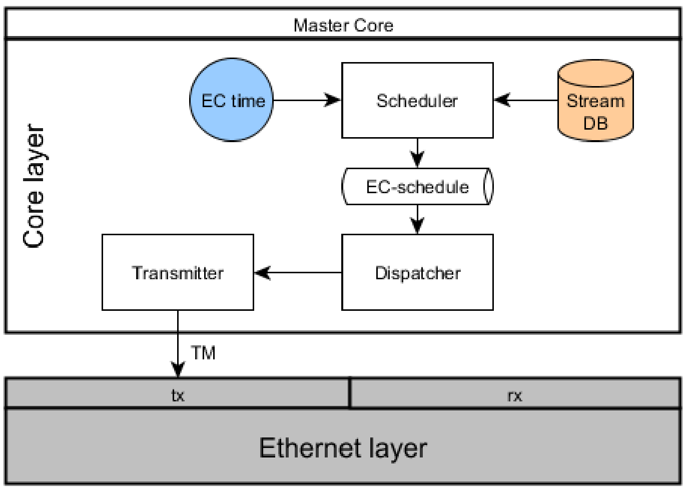


Figure 3.1: Master design

Described shortly, the master marks the start of a new Elementary Cycle sending the Trigger Message through Ethernet. In more detail, the master consists in two layers: Core layer and Ethernet layer.

Note that the Ethernet layer is painted in gray because it has not been directly part of my work; instead, it was just utilized to transmit or receive Ethernet messages.

Regarding the Core layer, each EC is triggered with a time event called EC time. It executes the Scheduler, block that reads the Stream DataBase (DB) and generates the EC-schedule, which is the list of the messages that have to be sent in the current EC. After this is done, the Dispatcher is executed, which takes the EC-schedule and constructs the content of the TM. Finally, this TM is transmitted through the Ethernet layer, ready to be received by all the slaves.

3.2 Slave

Coming up next, the design of the Slaves (see figure 3.2):

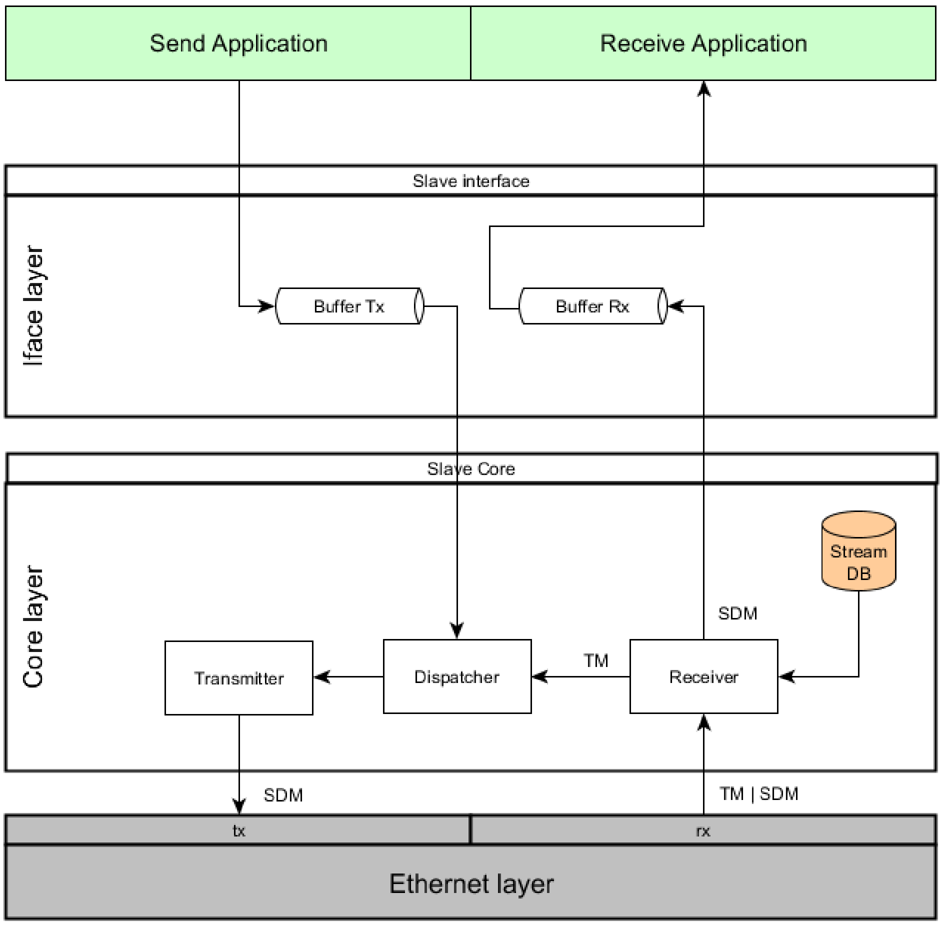


Figure 3.2: Slave design

In the same way as before, we will describe in brief the functionality of the slaves: every slave receives the Trigger Message and checks if it has to send any message. If that is the case, said slave constructs and sends a Synchronous Data Message (SDM) identified with a stream ID. This SDM is then received by all the slaves subscribed to that stream ID.

The design of a slave consists in three layers and applications: the applications represent the tasks that need to be executed by the slaves, so they are very diverse and can make anything that the user wants. In this example are showed simple Send and Receive applications to see more clearly its interaction with the slaves through the Interface layer. The slave layers from the top to the bottom are: Interface layer, Core layer and Ethernet layer.

The principal layer that represents the functionality of the slave is the Core layer, so we will start from here. Initially, each slave receives the TM from the Ethernet layer (which is the same as the one used in the master). It is analyzed by the Receiver module with the help of the Stream DB, which in this case contains the streams that the slave is a publisher or a subscriber. The TM received contains a list of the messages that have to be sent in the current EC. Specifically, this is a list of the node IDs and the stream IDs for every message. If the slave is the publisher of any of these stream IDs, said slave has to send a SDM with the data associated to this stream.

The SDM is constructed in the Dispatcher module, which takes the data stored in the Buffer Tx from the Interface layer. This Buffer Tx has been previously written by any application that has to send data, for example the simple Send Application. Note that the there is one Buffer Tx and Buffer Rx associated to each of the streams, so multiple streams can be accessed in the same EC. Once the Dispatcher has ended constructing the SDM with the corresponding data, it proceeds transmitting this message using the Transmitter module and sending it through the Ethernet layer.

Slaves, at the same time, can also receive a SDM from other slave. Following the same steps as before, the SDM comes through the Ethernet layer and is received with the Receiver module, checking in the Stream DB if the node ID is subscribed to the received stream. If that is the case, the final step is to save the received data in the Buffer Rx of the Interface layer, ready to be read by the Receive Application.

CHAPTER 4

IMPLEMENTATION

In this chapter we explain the implementation of the developed project. To put in context the implementation, an initial section will be dedicated to explain the tools and technology used in the project. Then, the master implementation is explained, followed by implementation of the slaves. And finally, it is explained the additional libraries that complement the master and slaves, which are the generic and common FTT libraries.

4.1 Used tools/technology

This software project was developed using C language compiled with the CMake tool, running on a computer with Linux OS. To test the correct functionality of the project during its development two main tools were used, first one is the own terminal emulator, using commands in the console to compile, run and debug the project; and second one is Wireshark, which helps to control precisely the timing and value of the Ethernet frames sent or received by our links.

The tools and technologies used to carry out this project are described in more detail below.

Linux OS

The entire project was developed with Linux OS, which is an open source operating system and it is commonly used in embedded systems [10] since it allows controlling many electronic devices at low level. The versatility and compatibility it provides with the technology used in this project makes it the optimal OS to work with. Concretely, we have used the Ubuntu 16.04 version.

C programming language

This programming language is one of the most widely used nowadays [11]. Especially it is the preferred one when having to develop embedded systems due to the really low, hardware level control and support that it provides, without entering in long and tedious assembler programs.

The generated code is also compatible with many different compilers and operating systems, facilitating a cross-platform programming and thus providing a better flexibility.

CMake tool

This one is the preferred tool to compile all the project files. Before using this tool, we compiled our code with the tool Makefile [12], but with the addition of new folders and subfolders to the project, the difficulty to write the Makefile properly, increased. In brief, we can just say that CMake is more user friendly in the end, that is why it was chosen.

To use CMake correctly, we have to edit the "CMakeLists.txt" file [13]. There is a principal file with this name that compiles the entire project. Additionally, there are additional files to compile libraries in the different subfolders of the project.

Wireshark

Wireshark is an open source Network Analyzer [14] that can be used for network troubleshooting and communications protocol development, amongst other applications. From its interface it is possible to select the link to be monitored, as depicted in figure 4.1. When selecting an Ethernet capture, we observe the exact time and value of the sent Ethernet frames (see figure 4.2).

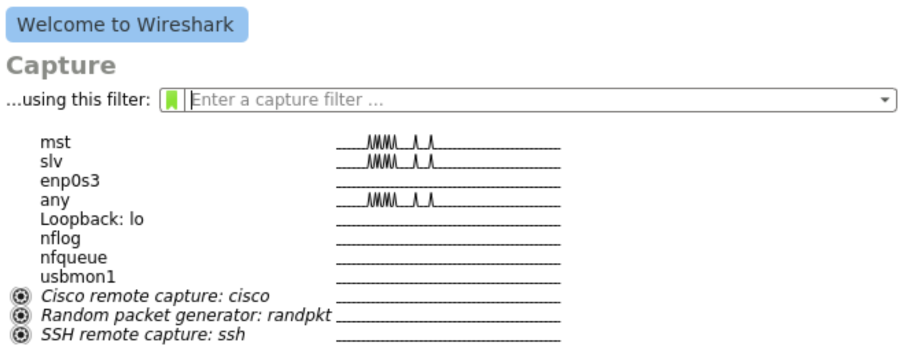
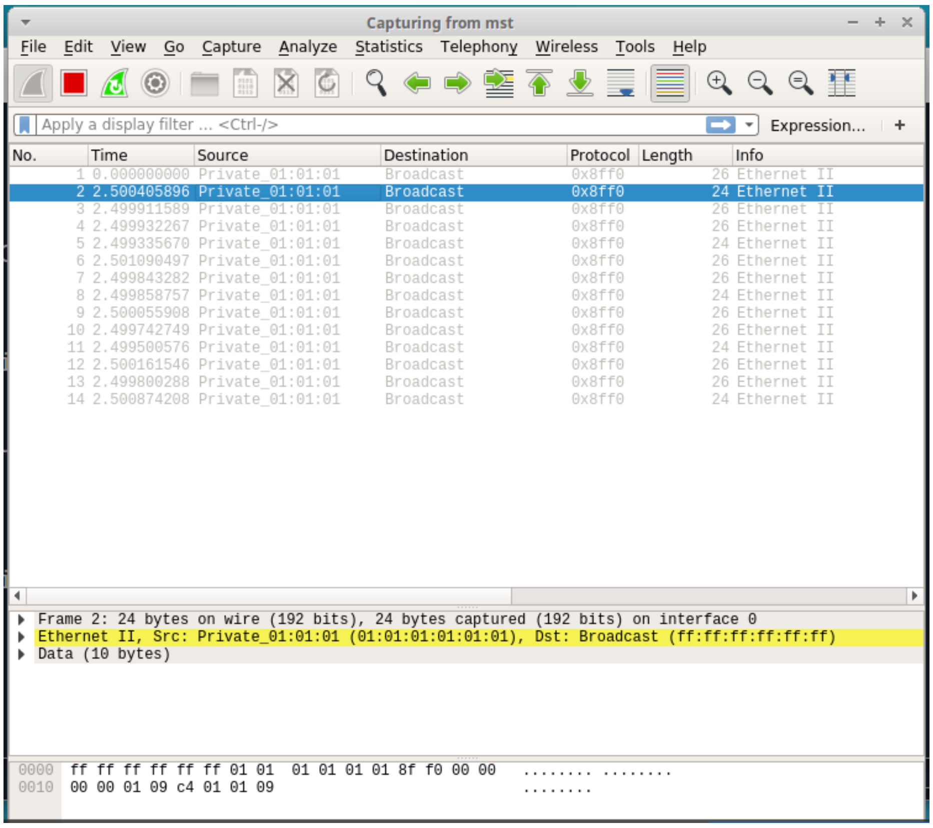


Figure 4.1: Start screen in Wireshark

Wireshark also has useful utilities like the option to filter the displayed frames or also to color them depending on certain conditions, to facilitate its differentiation.


Capturing from mst
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
+

```
No.   Time         Source              Destination   Protocol   Length Info
    1 0.000000000  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
    2 2.500405896  Private_01:01:01    Broadcast     0x8ff0     24 Ethernet II
    3 2.499911589  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
    4 2.499932267  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
    5 2.499335670  Private_01:01:01    Broadcast     0x8ff0     24 Ethernet II
    6 2.501090497  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
    7 2.499843282  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
    8 2.499858757  Private_01:01:01    Broadcast     0x8ff0     24 Ethernet II
    9 2.500055908  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
   10 2.499742749  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
   11 2.499500576  Private_01:01:01    Broadcast     0x8ff0     24 Ethernet II
   12 2.500161546  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
   13 2.499800288  Private_01:01:01    Broadcast     0x8ff0     26 Ethernet II
   14 2.500874208  Private_01:01:01    Broadcast     0x8ff0     24 Ethernet II

▸ Frame 2: 24 bytes on wire (192 bits), 24 bytes captured (192 bits) on interface 0
▸ Ethernet II, Src: Private_01:01:01 (01:01:01:01:01:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Data (10 bytes)

0000  ff ff ff ff ff ff 01 01  01 01 01 01 8f f0 00 00   ........ ........
0010  00 00 01 09 c4 01 01 09                            ........
```

Figure 4.2: Wireshark interface

4.2 Master

The reduced master we have implemented is divided in two layers: the Ethernet layer, which manages sent or received Ethernet frames; and the Core layer, which contains the basic behavior of the master itself.

The main code of this reduced master is found in the "master.c" file, which follows the design showed in the previous chapter. Of course, this file is not independent, as it needs other libraries to get the correct functionality. These libraries are shown in figure 4.3:

```c
//generic libraries
#include "eth.h"
#include "ts.h"

//common FTT libraries
#include "ftt_msg.h"
#include "stream_db.h"
```

Figure 4.3: "master.c" libraries

We find that the master uses generic and common FTT libraries. These libraries are explained in detail in sections 4.4 and 4.5, respectively. The "eth.h" library provides functions for Ethernet communications. The "ts.h" library provides functions for time control. The "ftt_msg.h" library has structs and defines to manage Ethernet and FTT messages. Finally, the "stream_db.h" library let us create, access and edit a stream database.

4.2.1 Ethernet Layer

This is the layer where the communications take place through Linux sockets. These sockets, formerly called Unix domain sockets [15] create an endpoint for communication to exchange data between processes on the same machine.

The messages are sent and received with Ethernet so a library for this purpose was added to the project, which name is "eth.h". This library is explained with more detail in the section 4.4.1.

In the "master.c" file, we use the *Eth_raw_init* function of the mentioned library to initialize the Linux sockets. Moreover, we use the *Eth_send* function to send the messages, which, in this case, are exclusively Trigger Messages.

4.2.2 Core Layer

This layer is the responsible to coordinate the communications with the connected slave nodes, sending informative TM at the start of a new EC. For this purpose, the master sends these messages in broadcast mode, in this way, all the slaves are supposed to receive it.

Before entering in the "master.c" file to see the Core Layer implementation, we should know that the master needs some input parameters to start its execution. These parameters are the following: EC size, Interlink name, Link name and Master Role. The EC size is the duration of the EC, measured in milliseconds. The Interlink name is the communication link connected between two masters. The Link name is the communication link that, in our case, is connected to the legacy switch. Finally, the Master Role, as its own name describes, configures the master with a certain role, with the possibility to be Leader, Follower or Standalone.

The input of these parameters is directly performed with commands from the Linux terminal emulator, as shown in table 4.1

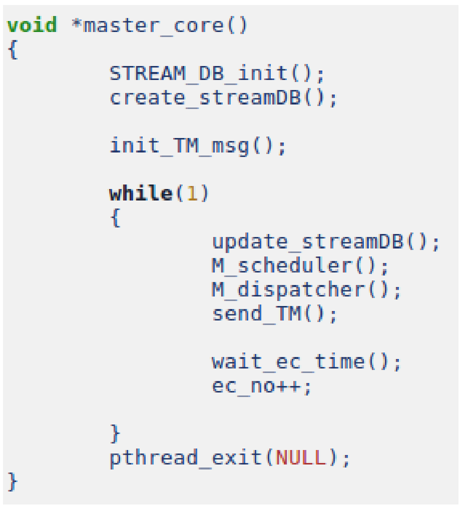
Parameter	**Command**	**Content**	**Example**
EC size	-e	Number (value in ms)	-e 1000
Interlink name	-I	String (ilink name)	-I ilink1
Link name	-i	String (link name)	-i mst
Master Role	-L / -F / -S	Leader / Follower / Standalone	-S

Table 4.1: Master parameters

Some of these parameters, like the Interlink name and the Master Role, are not utilized for this project but are there for future expansions. The Interlink will be omitted and the Master Role will always be Standalone, without any effect whatsoever.

Therefore, the command to start a master would be: sudo ./master -e 1000 -i mst -S In this example, the Elementary Cycles would last 1000 ms and the Ethernet communications will be done using the link named 'mst'. The Standalone (-S) indication does not affect in the current project.

Once the master program is initiated, it saves all the parameters received and starts a thread called *master_core* that will have a permanent cycle until the process is terminated [16]. The code structure of the thread is found in figure 4.4:



```
void *master_core()
{
        STREAM_DB_init();
        create_streamDB();

        init_TM_msg();

        while(1)
        {
                update_streamDB();
                M_scheduler();
                M_dispatcher();
                send_TM();

                wait_ec_time();
                ec_no++;

        }
        pthread_exit(NULL);
}
```

Figure 4.4: Code executed by the master_core thread

At the start, the stream DataBase (DB) is initialized making use of the *STREAM_DB_init* function, which is part of "stream_db.h" library. More details about this library are found in section 4.5.2. Then, the stream DB is defined at the start with *create_streamDB*, creating the streams with a certain stream ID, period, offset and message size; also configuring which are the publisher and the list of subscribers of these streams. A final note about the stream DB is that it has the possibility to be updated at the start of every Elementary Cycle, using the *update_streamDB* function. The possible updates that can be done in this function are: create a new stream, modify the period, offset or message size of an active stream, change the publisher node, add new subscribers, delete a subscriber, erase the list of subscribers or delete an active stream.

Before entering in the loop, a general structure for the TM is defined with the *init_TM_msg*, as observed in figure 4.5:

In the *init_TM_msg* function, the ETH_MSG and FTT_MSG_TM struct data types are used. These structs belong to the "ftt_msg.h" library, which is explained with all the details in section 4.5.1. The ETH_MSG is a struct that contains the fields of an Ethernet message (see figure 4.27) and the same for the ETH_MSG_TM (see figure 4.30), but

```
void init_TM_msg(){
        ETH_MSG* eth_msg = (ETH_MSG*) msg;
        for (i=0; i<ETH_MAC_LENGTH; i++) eth_msg->eth_src[i] = MAC_address[i];
        for (i=0; i<ETH_MAC_LENGTH; i++) eth_msg->eth_dst[i] = BCAST_MAC[i];
        eth_msg->eth_etype = htons(ETH_FTT_TYPE);

        tm = (FTT_MSG_TM*) eth_msg->eth_payload;
        tm->ftt_msg_head.type = FTT_TM;
        tm->ec_size = htons(ec_time);
}
```

Figure 4.5: Initialize TM message

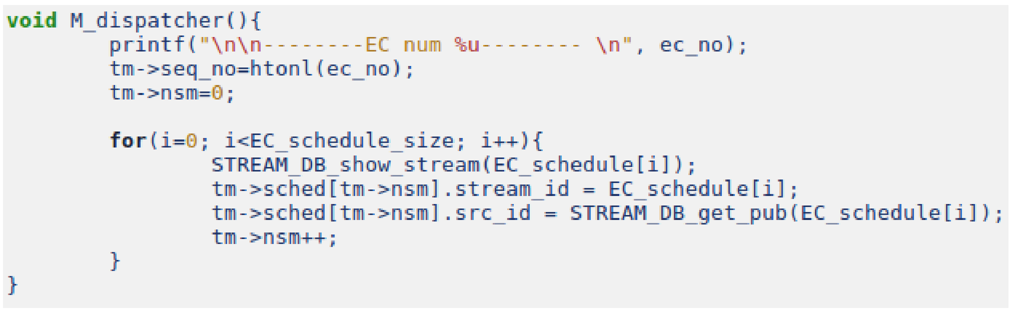
containing the fields of a Trigger Message. Looking at the *init_TM_msg* function we can see that the Header of the Ethernet message (source, destination and Ethertype) is determined, because it will not be modified during the execution. Finally, the TM is defined as the Payload of the Ethernet message, which has the type and the EC time determined too. The TM structure message can be observed with more detail in the figure 4.29. The rest of the TM fields will be modified in the infinite loop on every EC, in fact, a cycle in the infinite loop is a EC.

In the infinite loop, the master determines the list of streams that need to be transmitted in the current EC and stores them into the *EC_schedule* array, which is a global variable. This is done in *M_scheduler* function, shown in the figure 4.6:

```
void M_scheduler(){
        uint8_t nsm = 0;

        for(uint8_t stream_id = STREAM_DB_first();
            stream_id != INVALID_STREAM_ID ;
            stream_id = STREAM_DB_next(stream_id)){
                T_STREAM stream;
                STREAM_DB_get_stream(stream_id, &stream);
                if((ec_no%stream.period)-stream.offset == 0){
                        //save streams on EC schedule
                        EC_schedule[nsm]=stream_id;
                        nsm++;
                }
        }

        EC_schedule_size = nsm;
}
```

Figure 4.6: Master scheduler implementation

Note that the *M_scheduler* function always starts with zero number of synchronous messages (nsm). In the for loop, the master searches into the stream DB all the active streams configured in the stream DB. For every stream found, the master uses the Scheduler condition formula, previously commented in figure 2.4. When the condition is met, the stream ID is saved into the EC-schedule array, followed by the increase of the nsm, which, in the end is the same as the EC-schedule size. The next step is to read the content of the array *EC_schedule* to construct the TM, this is what is called the *M_dispatcher* (see figure 4.7):

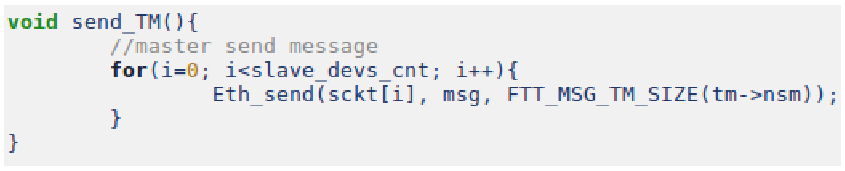


```
void M_dispatcher(){
        printf("\n\n--------EC num %u-------- \n", ec_no);
        tm->seq_no=htonl(ec_no);
        tm->nsm=0;

        for(i=0; i<EC_schedule_size; i++){
                STREAM_DB_show_stream(EC_schedule[i]);
                tm->sched[tm->nsm].stream_id = EC_schedule[i];
                tm->sched[tm->nsm].src_id = STREAM_DB_get_pub(EC_schedule[i]);
                tm->nsm++;
        }
}
```

Figure 4.7: Master dispatcher implementation

In the *M_dispatcher* function, the rest of the TM fields are constructed. One of the fields that is updated in the TM is the sequence number of the EC (ec_no variable). The rest of the TM fields are related to the EC-schedule array previously constructed in the M_scheduler function. The TM contains for every nsm the stream ID and its corresponding publisher slave (node ID). In the *M_dispatcher* function we also print debug information to see which EC the master is handling. Additionally, we also show information of all the streams triggered in each EC thanks to the *STREAM_DB_show_stream* function.

Once the TM is properly constructed, it is sent to the other slave nodes in broadcast mode, executing the *send_TM* function (see figure 4.8). This function consists in a for loop that sends the messages through the Ethernet Layer using the *Eth_send* function. The loop is executed as many times as Link names have been defined (remember Table 4.1).



```
void send_TM(){
        //master send message
        for(i=0; i<slave_devs_cnt; i++){
                Eth_send(sckt[i], msg, FTT_MSG_TM_SIZE(tm->nsm));
        }
}
```

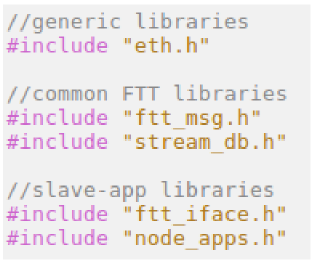
Figure 4.8: Master sending TM

Finally, after all this process has been done, the master waits the time until the next EC starts, waiting *ec_time* (in ms). To have consistent EC durations, we use the *clock_nanosleep* function that let us sleep until a certain timespec (see section 4.4.2). At the start, we get an initial timespec value corresponding to the current time. Then, in every EC we use the *timespec_add_ns* function, which belongs to the generic "ts.h" library, to add the *ec_time* to the timespec value we have already saved. The updated timespec value is the absolute time that the *clock_nanosleep* function will have to wait. When the time arrives, the number of EC is increased by one unity and the process of sending the TM starts again.

4.3 Slaves

The reduced slaves we have implemented are structured in three layers: Ethernet layer, to send or receive Ethernet frames; the Slave Core layer, which is the logical behavior of the slaves; and the Slave Interface layer, capable of communicating with the custom apps slaves execute.

Slave implementation is programmed in the "slave.c" file and it includes the following libraries:



```
//generic libraries
#include "eth.h"

//common FTT libraries
#include "ftt_msg.h"
#include "stream_db.h"

//slave-app libraries
#include "ftt_iface.h"
#include "node_apps.h"
```

Figure 4.9: "slave.c" libraries

The slave uses generic, common FTT and slave-app libraries. As the generic and common FTT libraries were commented in the previous 4.2 section, we will explain directly the slave-app libraries which are the new ones in this file. The "ftt_iface.h" library provides functions to communicate an application with a slave using the data streams. The "node_apps.h" library has all the available applications to be executed for the slaves.

4.3.1 Ethernet Layer

This layer works in the exact same way as in the previous 4.2.1 section, utilizing the "eth.h" generic library. The slaves use the Ethernet communications to send SDM to other nodes, and they also receive the FTT messages sent by the master or other nodes, which can be either the TM from the master or a SDM from other node.

In the "slave.c" file, we use the *Eth_raw_init* function to initialize the Linux sockets, the *Eth_recv* function to receive the FTT messages (TM or SDM) and also the *Eth_send* function to send SDM.

4.3.2 Core Layer

The reduced slaves have a different behavior than the master in its core. Slaves nodes receive the TM sent by the master, process it and if necessary, they send another FTT message with the stream data, this message is called Synchronous Data Message (SDM). The data contained in the SDM has as source or destination the applications that the own slaves execute.

Before entering in the "slave.c" file to see the Core Layer implementation, we should know that the slaves need some input parameters to start its execution. These parameters are: Link name, Node ID and App ID. The Link name is the communication link connected to the legacy switch. The Node ID is the unique identifier for the slaves, so that this ID can be used to identify which slaves are publishers or subscribers for the streams. The App ID indicates which application the slave has to execute, as there is a list of the available apps identified with this number (see section 4.3.4).

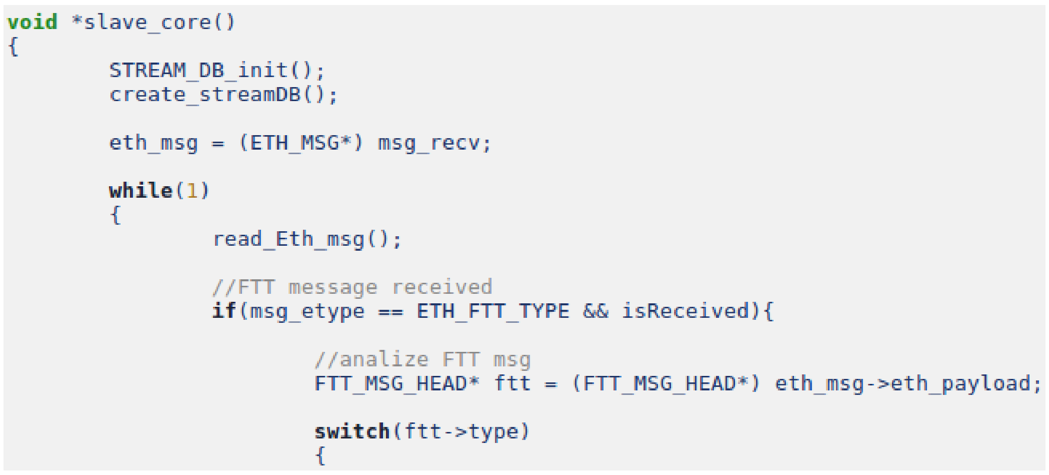
The input of these parameters is directly performed with commands from the Linux terminal emulator, as shown in table 4.2:

Parameter	**Command**	**Content**	**Example**
Node ID	-n	Number (ID value)	-n 3
App ID	-a	Number (ID value)	-a 1 ilink1
Link name	-i	Character (link name)	-i slv

Table 4.2: Slave parameters

Therefore, the command to start a slave would be: sudo ./slave -n 3 -a 1 -i slv
In this example, the selected slave will have a node ID = 3 and execute the application with ID = 1. The Ethernet communications would be done using the Ethernet link named 'slv'.

Entering in the implementation, the slave executes a thread called *slave_core* once the start slave command is executed. We can see the initial part of this thread in figure 4.10:

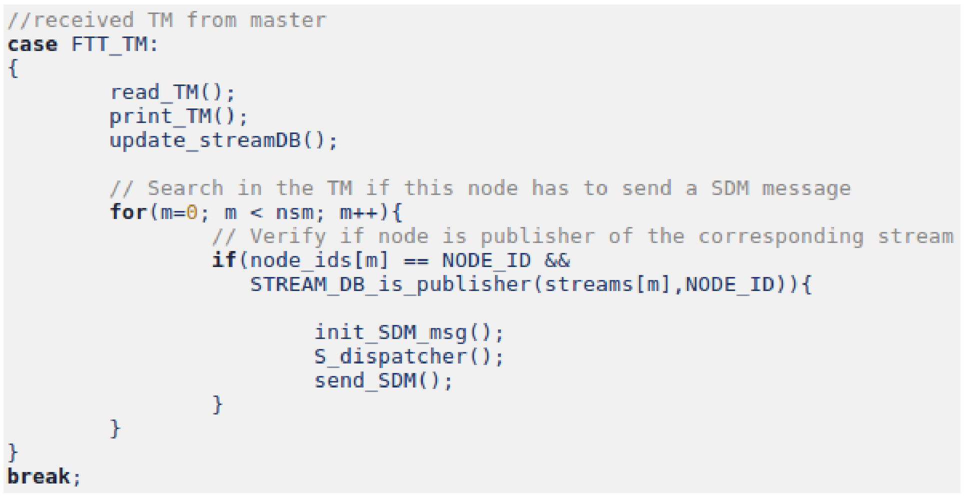


```
void *slave_core()
{
        STREAM_DB_init();
        create_streamDB();

        eth_msg = (ETH_MSG*) msg_recv;

        while(1)
        {
                read_Eth_msg();

                //FTT message received
                if(msg_etype == ETH_FTT_TYPE && isReceived){

                        //analize FTT msg
                        FTT_MSG_HEAD* ftt = (FTT_MSG_HEAD*) eth_msg->eth_payload;

                        switch(ftt->type)
                        {
```

Figure 4.10: Initial part of slave_core thread

As we can see in the figure 4.10, the initial step of the *slave_core* thread is initializing and creating the same stream database than in the master. Then it is known that an Ethernet message will be received at some point of the execution, that is why it calls *read_Eth_msg*, which waits until an Ethernet message is received. This Ethernet message is just checked by its etype and if it is equal to the FTT Ethertype (0x8FF0), the message payload is structured as an FTT message. After this process is done, there are two possible options in the FTT message type, which are differentiated in the switch.

The first case is that the received FTT message is a Trigger Message (see figure 4.11):



```
//received TM from master
case FTT_TM:
{
        read_TM();
        print_TM();
        update_streamDB();

        // Search in the TM if this node has to send a SDM message
        for(m=0; m < nsm; m++){
                // Verify if node is publisher of the corresponding stream
                if(node_ids[m] == NODE_ID &&
                   STREAM_DB_is_publisher(streams[m],NODE_ID)){

                        init_SDM_msg();
                        S_dispatcher();
                        send_SDM();
                }
        }
}
break;
```

Figure 4.11: Slave receives TM from master

In the case that the slave has received the TM from the master, it will proceed to read this TM with the function *read_TM*, then the content of this TM is printed through the terminal emulator with *print_TM* to facilitate the visibility of its content, which, at the same time acts as a debug tool to guarantee a correct functionality. Next step for the slave is to check if its own node ID appears in the list of messages that have to be sent in the current EC, and also, verifying that it is actually the publisher of the corresponding stream ID too. Whenever this situation happens, the slave has to send a SDM with the corresponding data for the selected stream ID. Starting with the *init_SDM_msg* function, the basic information of the SDM is set up. Then, the slave calls *S_dispatcher* to fill the SDM with the data that its own executing application can provide, as we can see in the figure 4.12:

The slave dispatcher simply reads the transmitted data that its own application has written in the transmission buffer, which is identified with the stream ID. To access the data, the slave uses the Slave-App Interface *FTT_IFACE_recv_slave* function, which is explained in more detail in section 4.3.3. Additionally, the data size that the streams have is defined in the stream DB so, if they do not match with the specified size, the terminal will show eventual Warnings or Errors regarding this issue. These Warnings or Errors do not appear in the figure 4.12 as they are too long to show up. In either case, when the data size is more than zero, it means that there was data in the transmission buffer, so the slave proceeds to copy it to the SDM which will be transmitted. Once the SDM is constructed, it is sent with the function *send_SDM* (see figure 4.13), also in broadcast mode, in this way all the nodes will be able to receive it.

```
void S_dispatcher(){

        stream_id_send = streams[m];
        printf("*Send stream %u* \n", stream_id_send);

        // dataTx is filled with the data that
        // the app has prepared for this stream
        FTT_IFACE_recv_slave(stream_id_send, dataTx, &dataTx_size);

        if(dataTx_size>0){

                printf("Data_send: ");
                for(i=0; i<dataTx_size; i++){
                        sdm_send->data[i] = dataTx[i];
                        printf("%u ", sdm_send->data[i]);
                }
                printf("\n");
        }
}
```

Figure 4.12: Slave dispatcher implementation

```
void send_SDM(){

        //Slave send message
        for(i=0; i<slave_devs_cnt; i++){
                Eth_send(sckt[i], msg_send, FTT_MSG_SDM_SIZE(dataTx_size));
        }
}
```

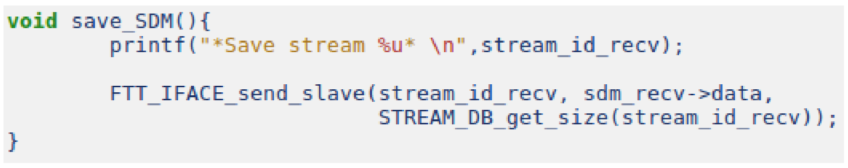
Figure 4.13: Slave sending SDM

The other case that can be found in the switch is that the FTT message corresponds to a Synchronous Data Message (see figure 4.14):

```
//received SDM from other node
case FTT_SDM:
{
        read_SDM();

        //save data to the app if node is subscribed
        if(STREAM_DB_is_subscriber(stream_id_recv,NODE_ID)){

                save_SDM();
        }

}
break;
```

Figure 4.14: Slave receives SDM from other node

The received SDM is sent by other node and contains only the stream ID and the data, as we can observe in the figure 4.31. The slave in first place has to read the SDM to find if it is subscribed to the received stream ID. If indeed is subscribed, the data is saved, or in other words, the data is sent to the application that the current slave is executing. To implement it, we can observe the function *save_SDM* function in the figure 4.15:



```
void save_SDM(){
        printf("*Save stream %u* \n",stream_id_recv);

        FTT_IFACE_send_slave(stream_id_recv, sdm_recv->data,
                             STREAM_DB_get_size(stream_id_recv));
}
```

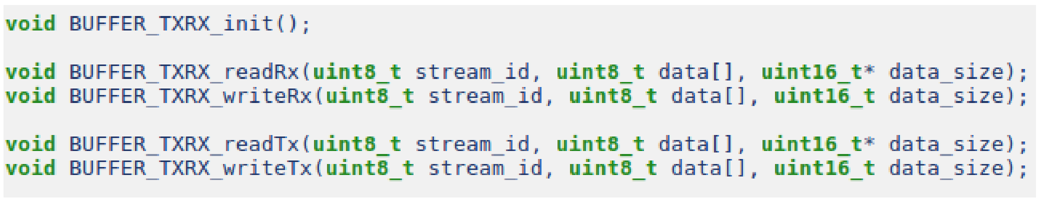
Figure 4.15: Initialize TM message

The *save_SDM* function sends the data contained in the SDM to its application using the Slave-App Interface *FTT_IFACE_send_slave* function, which is described in next section.

4.3.3 Interface Layer

This layer is the one that allows the communication between the slaves and the apps. As we can see in the slave design (see figure 3.2), it uses transmission and reception buffers, which are identified by the stream IDs and are capable of storing the data of these streams. For the implementation of this layer, two libraries were created: "ftt_iface.h" and "buffer_txrx.h". The first library mentioned is the principal one for the FTT interface, and their functions are the necessary ones to access the data streams from the slave or the apps. The library "buffer_txrx.h" is a secondary library that represents the internal work of the "ftt_iface.h" functions.

Let's take a look in the figure 4.16 at the functions available in the "buffer_txrx.h" because they will be used later for the implementation of the main library:

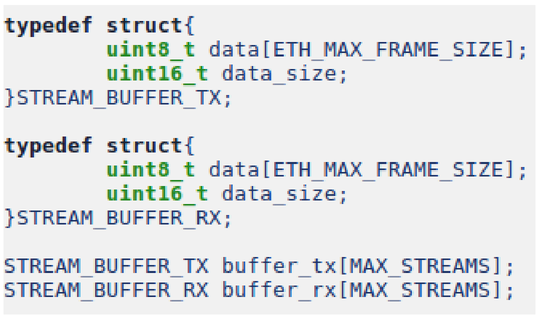


```
void BUFFER_TXRX_init();

void BUFFER_TXRX_readRx(uint8_t stream_id, uint8_t data[], uint16_t* data_size);
void BUFFER_TXRX_writeRx(uint8_t stream_id, uint8_t data[], uint16_t data_size);

void BUFFER_TXRX_readTx(uint8_t stream_id, uint8_t data[], uint16_t* data_size);
void BUFFER_TXRX_writeTx(uint8_t stream_id, uint8_t data[], uint16_t data_size);
```

Figure 4.16: Functions of "buffer_txrx.h" library

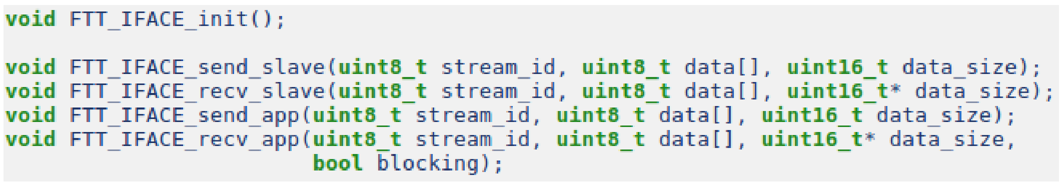
Basically, we have a *init* function to prepare the transmission/reception buffers. The most important part of this library is the *read/write* functions in the reception (Rx) or transmission (Tx) buffers. In the "buffer_txrx.c" file, an array of these buffers is created (see figure 4.17) and the index for the array is the stream ID.

The functions of the "buffer_txrx.h" library are directly used in the "ftt_iface.h" library, which is the one actually used by the apps or the slaves. The content of the "ftt_iface.h" is the one we can see in figure 4.18:



```
typedef struct{
        uint8_t data[ETH_MAX_FRAME_SIZE];
        uint16_t data_size;
}STREAM_BUFFER_TX;

typedef struct{
        uint8_t data[ETH_MAX_FRAME_SIZE];
        uint16_t data_size;
}STREAM_BUFFER_RX;

STREAM_BUFFER_TX buffer_tx[MAX_STREAMS];
STREAM_BUFFER_RX buffer_rx[MAX_STREAMS];
```

Figure 4.17: Array of buffers Tx/Rx



```
void FTT_IFACE_init();

void FTT_IFACE_send_slave(uint8_t stream_id, uint8_t data[], uint16_t data_size);
void FTT_IFACE_recv_slave(uint8_t stream_id, uint8_t data[], uint16_t* data_size);
void FTT_IFACE_send_app(uint8_t stream_id, uint8_t data[], uint16_t data_size);
void FTT_IFACE_recv_app(uint8_t stream_id, uint8_t data[], uint16_t* data_size,
                        bool blocking);
```

Figure 4.18: Functions of "ftt_iface.h" library

As we can see, comparing both libraries, is that they present the same structure because indeed they are directly related. The only addition is the blocking possibility in the *recv_app* function, useful to only receive data when the slave has in fact interacted with the Buffer Rx.

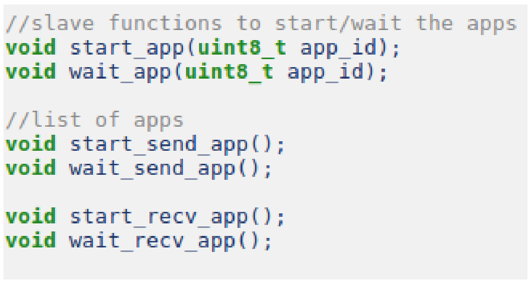
The names of the functions describe their functionality (*send* or *recv*) and where they can be used *(slave* or *app*). Below there is a quick relationship between the functions of the libraries:

- Send_slave →writeRx
- Recv_slave →readTx
- Send_app →writeTx
- Recv_app → readRx

4.3.4 Apps

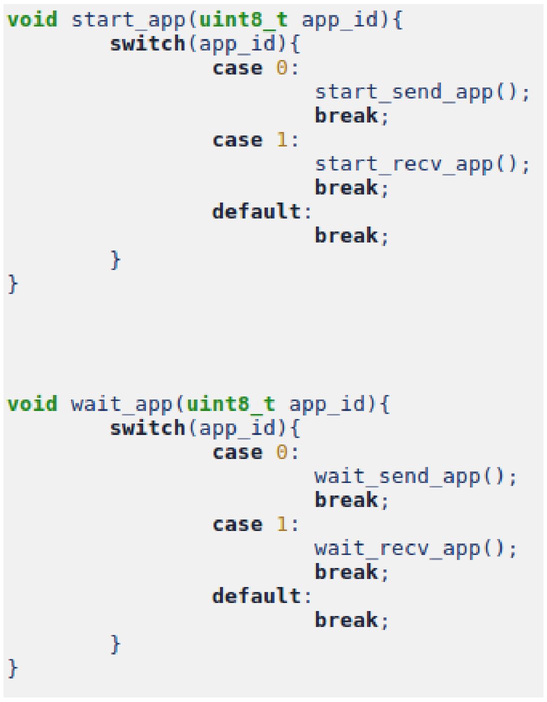
In these section will be shown two examples of simple applications that can interact with the slaves, but it has to be clear that the complexity of the apps is not limited by any means. In this project the apps realized are as simple as a Send App, which just emulates sending data (random values) through the Interface layer; and also a Receive App, which just receives the data available from the Interface layer and prints it in the console.

Some relevant information about the apps in general is that there is a library called "node_apps.h" which stores the list of the available apps (see figure 4.19), so if any application is created, it must be included in said file. The apps are executed continuously as threads, so for every application two functions are necessary: a start function to activate the thread; and a wait function to indicate the join of the same thread. In the library "node_apps.h" there is also a general function to start or wait any application, using the application ID. The configuration of the application IDs is managed inside the functions in "node_apps.h" library file, as can be seen in figure 4.20:



```
//slave functions to start/wait the apps
void start_app(uint8_t app_id);
void wait_app(uint8_t app_id);

//list of apps
void start_send_app();
void wait_send_app();

void start_recv_app();
void wait_recv_app();
```

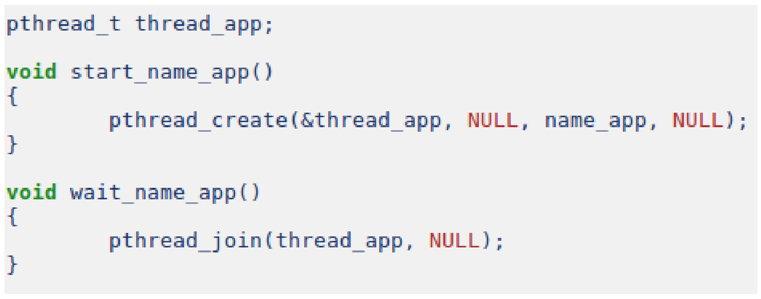
Figure 4.19: Functions in "node_apps.h" library



```
void start_app(uint8_t app_id){
        switch(app_id){
                case 0:
                        start_send_app();
                        break;
                case 1:
                        start_recv_app();
                        break;
                default:
                        break;
        }
}



void wait_app(uint8_t app_id){
        switch(app_id){
                case 0:
                        wait_send_app();
                        break;
                case 1:
                        wait_recv_app();
                        break;
                default:
                        break;
        }
}
```

Figure 4.20: General start/wait functions selected with application ID

The general code structure of the start and wait functions of any application is observed in figure 4.21:

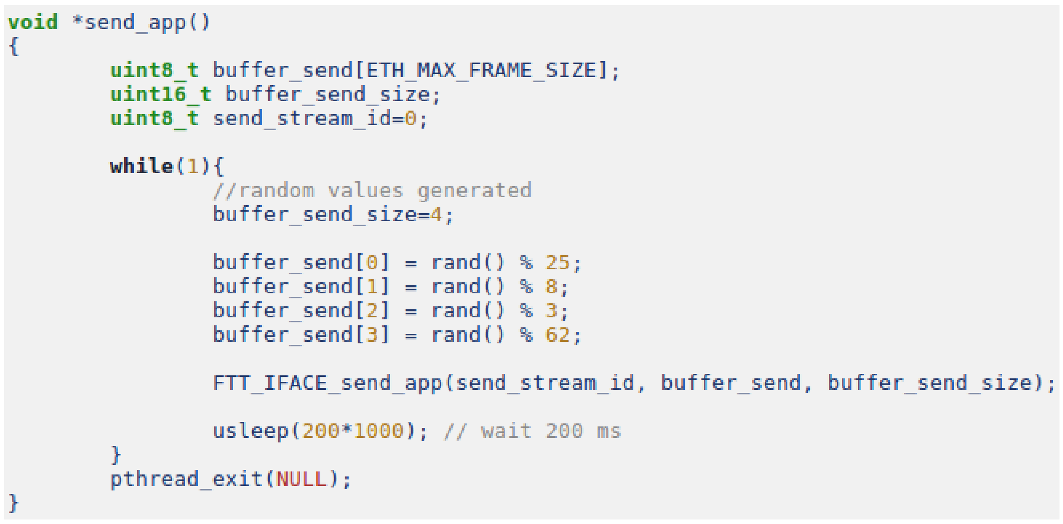


```
pthread_t thread_app;

void start_name_app()
{
        pthread_create(&thread_app, NULL, name_app, NULL);
}

void wait_name_app()
{
        pthread_join(thread_app, NULL);
}
```

Figure 4.21: Structure of start and wait functions of any application

So the only difference between apps is the code executed in the "name_app" function thread.

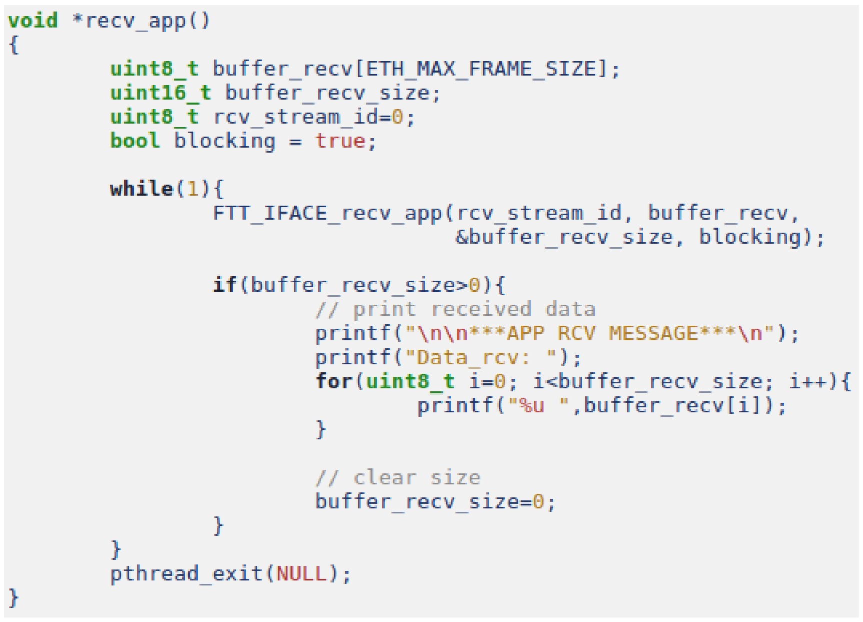
As we commented previously, we did two simple apps to test this functionality, one for sending random data and the other for receiving data and printing it. We have configured both apps so that they interact with the same stream (ID = 0). These apps are called *send_app* and *recv_app*, they can be found in figures 4.22 and 4.23, respectively.



```
void *send_app()
{
        uint8_t buffer_send[ETH_MAX_FRAME_SIZE];
        uint16_t buffer_send_size;
        uint8_t send_stream_id=0;

        while(1){
                //random values generated
                buffer_send_size=4;

                buffer_send[0] = rand() % 25;
                buffer_send[1] = rand() % 8;
                buffer_send[2] = rand() % 3;
                buffer_send[3] = rand() % 62;

                FTT_IFACE_send_app(send_stream_id, buffer_send, buffer_send_size);

                usleep(200*1000); // wait 200 ms
        }
        pthread_exit(NULL);
}
```

Figure 4.22: Example of send application

For the implementation of the *send_app*, the application needs a buffer array and a buffer size variable. The buffer size variable should coincide with the specified data size of the stream ID in the stream DB. To simulate the reading of a sensor, the application generates random values to fill the buffer array. Right after, the data is sent with the *FTT_IFACE_send_app* function, which as we know will put the data into the Transmission buffer array, indexed in the 0 position as it is the selected stream ID. Finally, the application waits 200 ms to generate again the values in the buffer. This wait time is arbitrary, but it has a meaning. The sensors of DECS work at a certain speed, and it is important to adequate the EC time in the system in a way that the sensor does not makes an excessive oversampling or otherwise, that the EC time is shorter than the sample, which would cause no available data to be transmitted.



```
void *recv_app()
{
        uint8_t buffer_recv[ETH_MAX_FRAME_SIZE];
        uint16_t buffer_recv_size;
        uint8_t rcv_stream_id=0;
        bool blocking = true;

        while(1){
                FTT_IFACE_recv_app(rcv_stream_id, buffer_recv,
                                   &buffer_recv_size, blocking);

                if(buffer_recv_size>0){
                        // print received data
                        printf("\n\n***APP RCV MESSAGE***\n");
                        printf("Data_rcv: ");
                        for(uint8_t i=0; i<buffer_recv_size; i++){
                                printf("%u ",buffer_recv[i]);
                        }

                        // clear size
                        buffer_recv_size=0;
                }
        }
        pthread_exit(NULL);
}
```

Figure 4.23: Example of receive application

For the implementation of the *recv_app*, the application at least needs a buffer array and a buffer size variable. We can also select to block or not block the *FTT_IFACE_recv_app* function to wait until the data is received or just check if there is data available, and if that is not the case, continue executing the rest of the code. With the mentioned function we will receive the data available from the Reception buffer array of the Interface Layer. The only condition to check if new data was received is if the buffer size variable is greater than zero. If the condition is met, the application shows the data received using the terminal emulator and clears the buffer size.

4.4 Generic modules

In this section it will be explained the generic libraries used in this project. These libraries were directly added to the project, so its implementation was not mine. The included libraries are: Ethernet and Timespec. The Ethernet library is mainly used to send or receive Ethernet messages. The Timespec library has several functions that can operate the timespec struct [17].

4.4.1 Ethernet

The Ethernet library, called "eth.h", provides functions for Ethernet communications. In the project, the Ethernet layer is implemented with the use of this library. The content of this library is shown in figure 4.24:

```c
#define ETH_MAC_LENGTH 6
#define ETH_MAX_FRAME_SIZE 1514

#define BCAST_MAC ((uint8_t[]){0xFF,0xFF,0xFF,0xFF,0xFF,0xFF})

#define dstMAC(_buf)    ((unsigned char *)_buf+0)
#define srcMAC(_buf)    ((unsigned char *)_buf+6)
#define etypeAddr(_buf) ((unsigned char *)_buf+12)
#define dataAddr(_buf)  ((unsigned char *)_buf+14)

int  Eth_raw_init (char* iface_name);
int  Eth_send     (int sckt, uint8_t* buffer, uint16_t  buffer_size);
int  Eth_recv     (int sckt, uint8_t* buffer, uint16_t* buffer_size);
void Eth_close    (char* iface_name, int sckt);
```

Figure 4.24: Initialize TM message

As we can see, the library has several defines, highlighting the ETH_MAC_LENGTH (value=6), which is the length of a MAC direction; the ETH_MAX_FRAME_SIZE (value=1514), which is the maximum length of an Ethernet frame; and the BCAST_MAC, which is the MAC that we have to use if we want to send a message in broadcast mode.

Regarding the functions, we have functions to open or close Linux Sockets; and then functions to send or receive Ethernet messages using these sockets.

4.4.2 Timespec

The Timespec library, called "ts.h", provides functions for time control. Basically it contains functions to operate the timespec struct, as we can see in the figure 4.25:

```c
#define NSEC_PER_SEC 1000000000L
#define NSEC_PER_MS  1000000

struct timespec timespec_from_ns(unsigned long ns);
unsigned long long timespec_getns(struct timespec ts);

unsigned char timespec_gt(struct timespec ts1, struct timespec ts2);

struct timespec timespec_add(struct timespec a, struct timespec b);
void timespec_add_ns(struct timespec *a, unsigned long ns);

unsigned long long timespec_diff(struct timespec start, struct timespec end);
signed long long timespec_sub(struct timespec ts1, struct timespec ts2);
void timespec_sub_ns(struct timespec *a, unsigned long ns);

void timespec_print(struct timespec ts);
```

Figure 4.25: Content of the "ts.h" generic library

The timespec struct contains a precise time in the following format: absolute time in seconds and the nanoseconds value of that second. The library contains defines such as the NSEC_PER_SEC and NSEC_PER_MS, which are conversion factors between nanoseconds-seconds and nanoseconds-milliseconds.

We are not going to explain all of the available functions in the library, but we will highlight the *timespec_add* function, which operates the sum of two timespecs; the *timespec_add_ns*, which adds a nanoseconds value to a certain timespec. In the same way we have the substract operator, with the *timespec_sub* and *timespec_sub_ns*. And a useful function also could be the *timespec_diff*, which returns the value of the difference between two timespecs, in nanoseconds.

4.5 Common FTT modules

In this section it will be explained the common FTT libraries used in this project. These libraries are used in both master and slaves. The included libraries are: FTT messages and Stream DB. The library of FTT messages has several structs and defines to construct the messages like TM, SDM, or even a general Ethernet message. The Stream DB library has all the necessary functions to create, modify, delete or consult streams.

4.5.1 FTT messages

This library has the name "ftt_msg.h" in the source code. As we previously commented, this library has structs and defines to manage all related to the structures of FTT messages.

To start with this library, we will look at the general Ethernet frame structure in the figure 4.26:

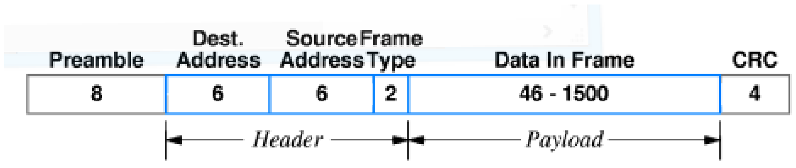
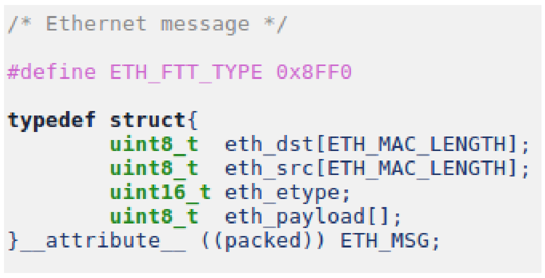


Figure 4.26: Ethernet frame structure

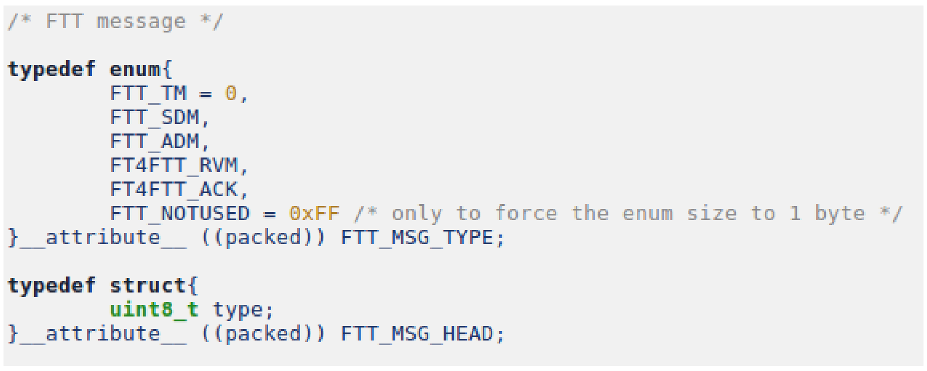
When we have to handle Ethernet messages we just have to use the part colored in blue, corresponding to the Header and the Payload. The Header is composed by three parts: Destination MAC Address (6 bytes), Source MAC Address (6 bytes) and Ethertype (2 bytes). The final part of the frame is the Data, which can be from a minimum of 46 bytes to a maximum of 1500 bytes. In our case, we will use the Payload part to build our own FTT messages; and they will be identified with an Ethertype equal to 0x8FF0. To see the Ethernet struct in the code see figure 4.27:

As we commented, in the Ethernet frame Payload we will build the FTT message, so it is necessary a field that identifies the different kind of FTT messages (see figure 4.28). In addition, we can use a general structure for an FTT message that only contains this identifier (1 byte). The general structure is useful when another component receives a FTT message. For example, the slaves use it when they receive the message and they do not know if it is a TM or SDM until they read the identifier.



```
/* Ethernet message */

#define ETH_FTT_TYPE 0x8FF0

typedef struct{
        uint8_t  eth_dst[ETH_MAC_LENGTH];
        uint8_t  eth_src[ETH_MAC_LENGTH];
        uint16_t eth_etype;
        uint8_t  eth_payload[];
}__attribute__ ((packed)) ETH_MSG;
```

Figure 4.27: Ethernet message in the code



```
/* FTT message */

typedef enum{
        FTT_TM = 0,
        FTT_SDM,
        FTT_ADM,
        FT4FTT_RVM,
        FT4FTT_ACK,
        FTT_NOTUSED = 0xFF /* only to force the enum size to 1 byte */
}__attribute__ ((packed)) FTT_MSG_TYPE;

typedef struct{
        uint8_t type;
}__attribute__ ((packed)) FTT_MSG_HEAD;
```

Figure 4.28: FTT message identifiers and FTT message head structure

For instance, the TM has the first byte of the FTT message equal to 0; and the SDM equal to 1. There are more identifiers that are not used in this project, such as the ADM which would be equal to 2.

The FTT messages that we use in this project are TM and SDM. Every one of these messages have a particular format. The structure of a TM is the one shown in the figure 4.29:

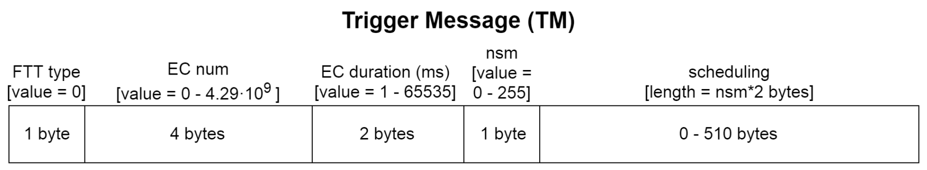
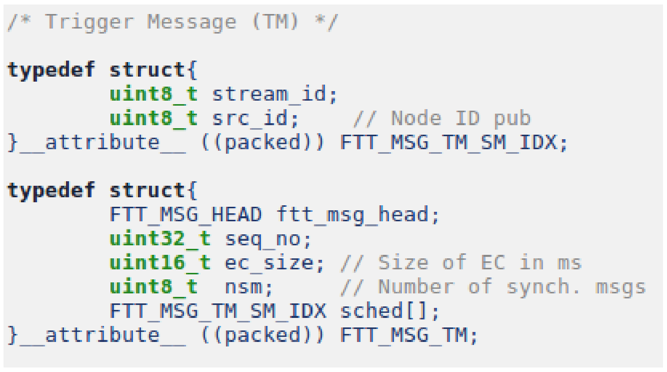

Trigger Message (TM)
FTT type [value = 0]
EC num [value = 0 - 4.29·10^9]
EC duration (ms) [value = 1 - 65535]
nsm [value = 0 - 255]
scheduling [length = nsm*2 bytes]
1 byte
4 bytes
2 bytes
1 byte
0 - 510 bytes

Figure 4.29: Trigger Message structure

As we can see from the above figure, the TMs will always start with a 0 that identifies them. The next field is the number of EC, which uses 4 bytes to have a wide range of values. Another informative field is the EC duration, using 2 bytes and represented in milliseconds. In the last part of the TM, there are fields that inform about the messages that have to be sent: one byte tells how many synchronous messages (nsm) are sent in the current EC. And then the scheduling part has a length corresponding to 2 times the nsm value. For every nsm, the scheduling indicates which stream ID has to be sent, also indicating which node ID is the publisher of that stream.

In the library, we are interested in a struct that formats the TM properly (see figure 4.30):



```
/* Trigger Message (TM) */

typedef struct{
        uint8_t stream_id;
        uint8_t src_id;    // Node ID pub
}__attribute__ ((packed)) FTT_MSG_TM_SM_IDX;

typedef struct{
        FTT_MSG_HEAD ftt_msg_head;
        uint32_t seq_no;
        uint16_t ec_size; // Size of EC in ms
        uint8_t  nsm;     // Number of synch. msgs
        FTT_MSG_TM_SM_IDX sched[];
}__attribute__ ((packed)) FTT_MSG_TM;
```

Figure 4.30: TM struct

And the other FTT message that we also use in the project is the SDM, which has a much simpler format, as it is just composed by the FTT Type equal to 1, then one byte to identify the stream ID and finally the data of said stream. The SDM structure can be seen in figure 4.31 and its implementation in the code in figure 4.32:

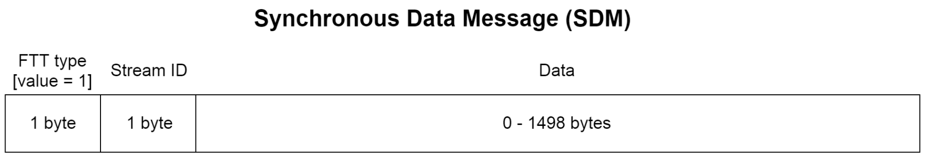
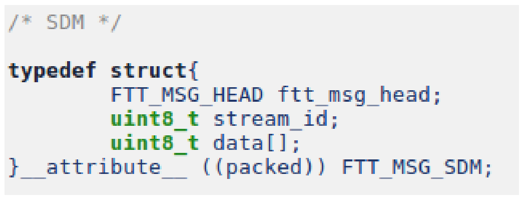


Figure 4.31: Synchronous Data Message structure



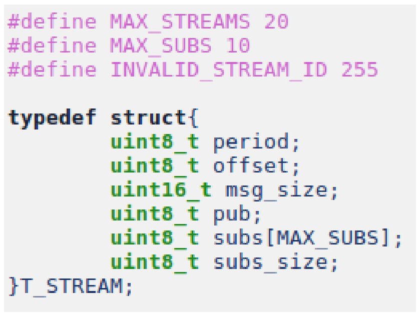
```
/* SDM */

typedef struct{
        FTT_MSG_HEAD ftt_msg_head;
        uint8_t stream_id;
        uint8_t data[];
}__attribute__ ((packed)) FTT_MSG_SDM;
```

Figure 4.32: SDM struct

4.5.2 Stream DB

The Stream DB is a library called "stream_db.h", which has several functions to manage the stream database. The logic of these functions is programmed in its respective "stream_db.c" file. But as there is a wide variety of functions we will just show them and explain in which situations can be useful.

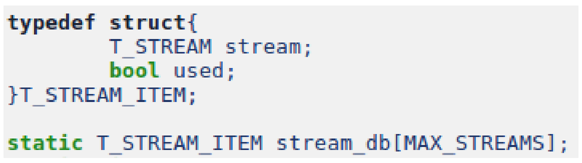
Starting with the Stream DB library, we have to define the parameters that form a stream. In this case, the only streams available are data streams, but in future extensions there could be also control streams that could have other parameters. The data streams have the parameters: period, offset, message size (bytes), publisher (node ID), list of subscribers (node IDs) and the quantity of subscribers. We can see the struct of the stream in figure 4.33:



```
#define MAX_STREAMS 20
#define MAX_SUBS 10
#define INVALID_STREAM_ID 255

typedef struct{
        uint8_t period;
        uint8_t offset;
        uint16_t msg_size;
        uint8_t pub;
        uint8_t subs[MAX_SUBS];
        uint8_t subs_size;
}T_STREAM;
```

Figure 4.33: Stream struct

As we can see from the above figure, the stream ID is not one of those parameters. Instead, the stream ID is saved internally inside the stream DB. To implement it, in the .c file there is an array of all the streams that uses another struct called T_STREAM_ITEM (see figure 4.34), being the index the stream ID.

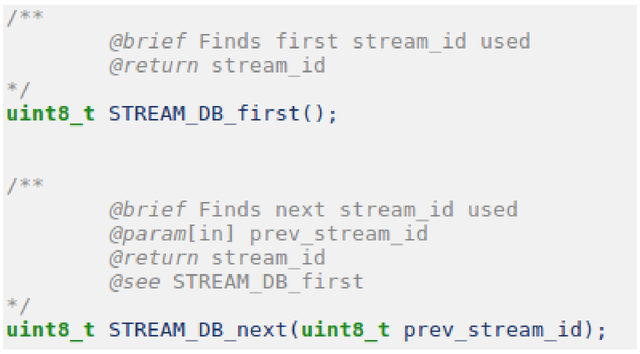


```
typedef struct{
        T_STREAM stream;
        bool used;
}T_STREAM_ITEM;

static T_STREAM_ITEM stream_db[MAX_STREAMS];
```

Figure 4.34: Stream struct internally in database

The T_STREAM_ITEM struct is composed by the T_STREAM struct (figure 4.33) in addition with a variable "used" that identifies if a particular stream is active or not. The database is limited by the MAX_STREAMS number, which is set to 20, but can be modified if necessary. This would mean that the available stream ID values are, a minimum of 0 and a maximum of 19, and any number superior would be out of the range of the stream_db array and would cause an error. The goal of the "used" variable for every stream is simple, when a stream is created is set to true, and when deleted is set to false.

When another module, for example the master, has to search inside the stream DB without knowing what streams are online, it has the functions of the figure 4.35 to cover this necessity:

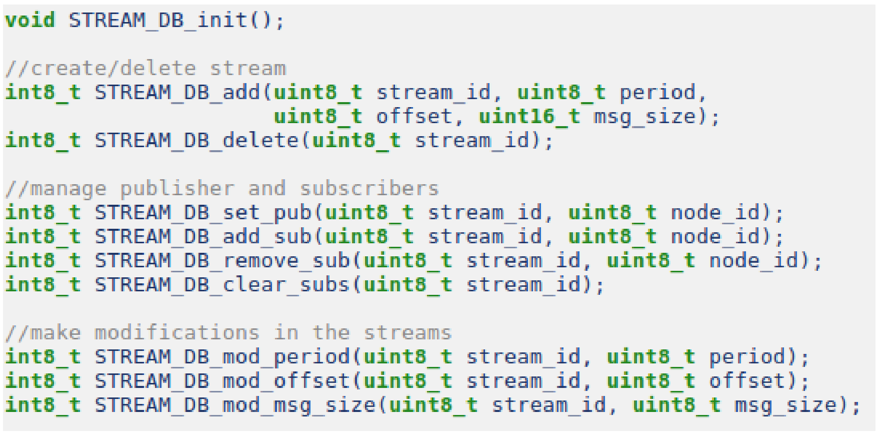


```
/**
        @brief Finds first stream_id used
        @return stream_id
*/
uint8_t STREAM_DB_first();


/**
        @brief Finds next stream_id used
        @param[in] prev_stream_id
        @return stream_id
        @see STREAM_DB_first
*/
uint8_t STREAM_DB_next(uint8_t prev_stream_id);
```

Figure 4.35: Functions to search inside the database

Both functions search inside the Stream DB and return values of active stream IDs. If the database does not find more active stream IDs, then they return an INVALID_STREAM_ID (value=255). An example of usage of these functions can be seen in the Dispatcher module (see figure 4.7).

The rest of the Stream DB functions can be divided in two big blocks: on one hand, functions to make changes in the stream DB (see figure 4.36); and on the other hand, functions to consult the stream DB (see figure 4.37).

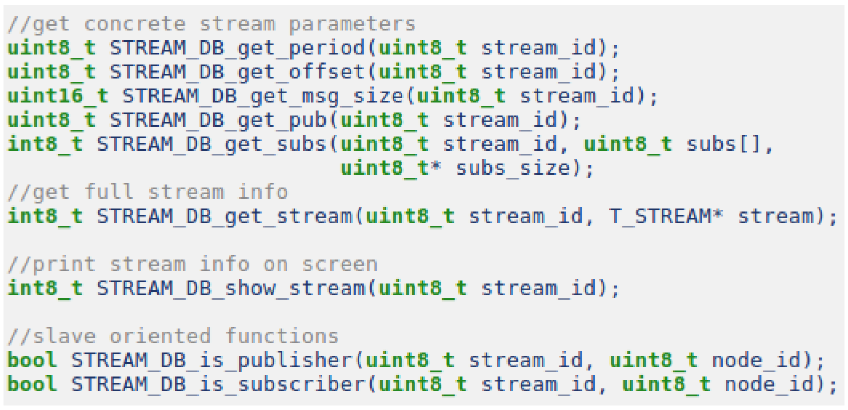


```
void STREAM_DB_init();

//create/delete stream
int8_t STREAM_DB_add(uint8_t stream_id, uint8_t period,
                     uint8_t offset, uint16_t msg_size);
int8_t STREAM_DB_delete(uint8_t stream_id);

//manage publisher and subscribers
int8_t STREAM_DB_set_pub(uint8_t stream_id, uint8_t node_id);
int8_t STREAM_DB_add_sub(uint8_t stream_id, uint8_t node_id);
int8_t STREAM_DB_remove_sub(uint8_t stream_id, uint8_t node_id);
int8_t STREAM_DB_clear_subs(uint8_t stream_id);

//make modifications in the streams
int8_t STREAM_DB_mod_period(uint8_t stream_id, uint8_t period);
int8_t STREAM_DB_mod_offset(uint8_t stream_id, uint8_t offset);
int8_t STREAM_DB_mod_msg_size(uint8_t stream_id, uint8_t msg_size);
```

Figure 4.36: unctions to make changes in Stream DB

In the first block we find a function to initiate the stream DB, of course functions to create and delete an stream, then functions to manage the publisher and subscribers of the streams and last but not least, functions to make modifications in certain stream parameters, such as the period, offset and the message size.

In this second block we find functions to get concrete stream parameters, such as period, offset, message size, the publisher and the list of subscribers of any stream. We also have the option to get the full stream with the get_stream function, which will return the struct of the figure 4.33. And if we want to print the stream parameters on



```
//get concrete stream parameters
uint8_t STREAM_DB_get_period(uint8_t stream_id);
uint8_t STREAM_DB_get_offset(uint8_t stream_id);
uint16_t STREAM_DB_get_msg_size(uint8_t stream_id);
uint8_t STREAM_DB_get_pub(uint8_t stream_id);
int8_t STREAM_DB_get_subs(uint8_t stream_id, uint8_t subs[],
                          uint8_t* subs_size);
//get full stream info
int8_t STREAM_DB_get_stream(uint8_t stream_id, T_STREAM* stream);

//print stream info on screen
int8_t STREAM_DB_show_stream(uint8_t stream_id);

//slave oriented functions
bool STREAM_DB_is_publisher(uint8_t stream_id, uint8_t node_id);
bool STREAM_DB_is_subscriber(uint8_t stream_id, uint8_t node_id);
```

Figure 4.37: Functions to consult the Stream DB

screen, we can use the *show_stream* function. Finally, there are slave oriented functions, that let the slaves know if they are publisher or subscribers of a particular stream.

CHAPTER 5

FUNCTIONAL VERIFICATION

In this chapter we describe the set of experiments carried out to verify the correct functionality of the project's final state. For any experiment that can be done with this project, there are in total three variable parameters: the first one is change the content of the Stream DB, which defines the interaction between the slaves through the streams; the second one is the number of slaves, which adds complexity to the system; and the third one is the application these slaves execute, which defines the role of every slave in the system.

First, during the implementation phase, we did small and specific experiments to verify the correctness of the individual components, such as the master, the slaves or the applications. For instance, once the Stream DB was finished, we tested if the master was constructing and sending correctly the Trigger Messages in every EC.

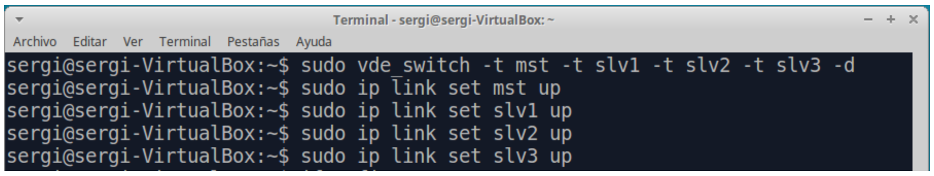
When the project was built in its entirety, we performed two final experiments to verify that it worked correctly. In the first experiment a slave transmits a periodic message to other two slaves, verifying the correct functioning of the basic mechanisms. The setup for this experiment is simple as it is performed in the same machine using a virtual switch. In the second experiment the infrastructure is hardware and allows us to demonstrate that the system is able to deal with different types of traffic in a real environment.

5.1 Simple setup

Our goal in this first experiment is to test the communication between three slaves through a single stream, one being the publisher and other two, the subscribers. The master will indicate to the publisher to transmit the data associated with the stream every 2 ECs and the subscribers will receive it.

Regarding the setup for this experiment, both the master and slaves are executed in the same computer. More specifically, the project was tested in my laptop Sony VAIO model SVF1531C4E, which has a processor Intel Core i7-4500U @ 1,80 GHz and 8 GB of RAM. In my case, to run the project in Linux (Ubuntu), the Oracle VM VirtualBox was used on top of Windows 8.1.

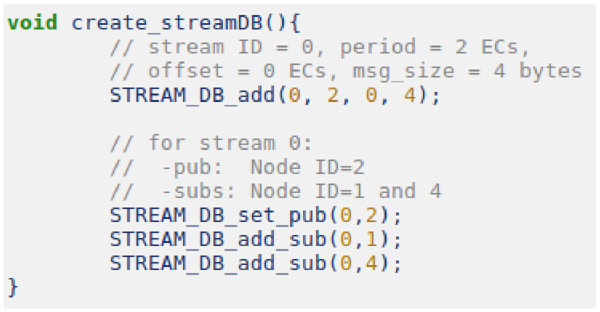
The master and the three slaves communicate with a virtual switch configured with the terminal emulator as shown in figure 5.1:


Terminal - sergi@sergi-VirtualBox: ~
Archivo Editar Ver Terminal Pestañas Ayuda

```
sergi@sergi-VirtualBox:~$ sudo vde_switch -t mst -t slv1 -t slv2 -t slv3 -d
sergi@sergi-VirtualBox:~$ sudo ip link set mst up
sergi@sergi-VirtualBox:~$ sudo ip link set slv1 up
sergi@sergi-VirtualBox:~$ sudo ip link set slv2 up
sergi@sergi-VirtualBox:~$ sudo ip link set slv3 up
```

Figure 5.1: Virtual switch master and three slaves

Before running the program for each of these links, the Stream DB needs to be modified with the required parameters. This Stream DB is static and can only be modified in the code, using the function *create_streamDB* (see figure 5.2). In this case, the *update_streamDB* function will be empty as we do not want to make any changes.



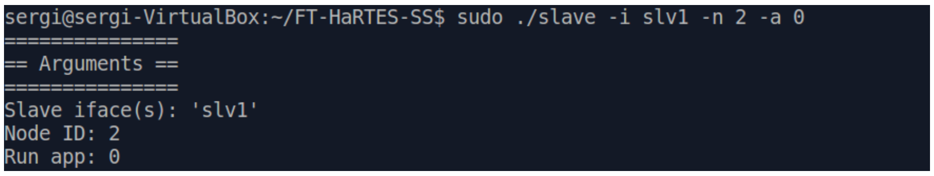
```
void create_streamDB(){
        // stream ID = 0, period = 2 ECs,
        // offset = 0 ECs, msg_size = 4 bytes
        STREAM_DB_add(0, 2, 0, 4);

        // for stream 0:
        //  -pub:  Node ID=2
        //  -subs: Node ID=1 and 4
        STREAM_DB_set_pub(0,2);
        STREAM_DB_add_sub(0,1);
        STREAM_DB_add_sub(0,4);
}
```

Figure 5.2: Stream DB one stream

As observed in figure 5.2, we are just interested in adding one stream that is sent every 2 ECs, without offset and with a data size of 4 bytes. Then, for this same stream ID, the publisher is set to the slave with ID=2, and there are multiple subscribers, including slaves with ID=1 and ID=4. Note that the ID values are arbitrary.

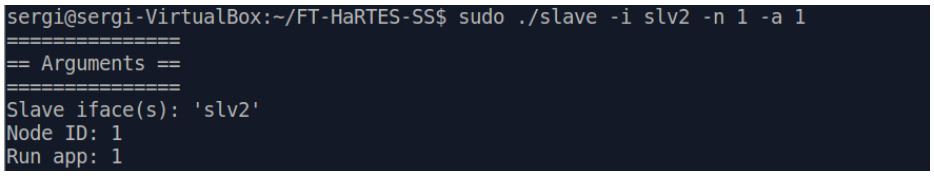
Next step is to decide which are the applications that slaves will execute. This decision is important, because it will determine the whole functionality of the setup. For this experiment, the applications used will be: for the publisher slave, an application that generates random data (4 bytes, which is the stream data size); and for the subscriber slaves, the selected application is the same for both of them, it will just receive the data of the subscribed stream and print the data received in the console. These applications are the same as we can see in the figures 4.22 and 4.23, respectively.

Once all has been setup, we just have to start the slaves, giving them the parameters of the Table 4.2 to start functioning. We can see the executed commands to start slave 1, 2 and 3 in the figures 5.3, 5.4 and 5.5:



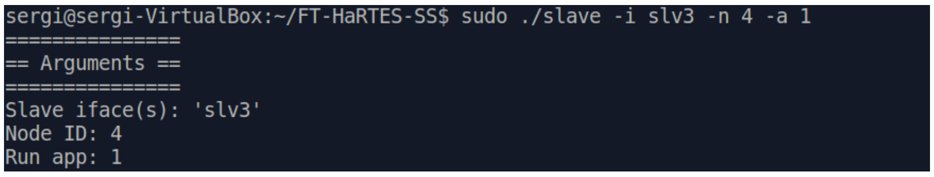
```
sergi@sergi-VirtualBox:~/FT-HaRTES-SS$ sudo ./slave -i slv1 -n 2 -a 0
===============
== Arguments ==
===============
Slave iface(s): 'slv1'
Node ID: 2
Run app: 0
```

Figure 5.3: Start slave 1



```
sergi@sergi-VirtualBox:~/FT-HaRTES-SS$ sudo ./slave -i slv2 -n 1 -a 1
===============
== Arguments ==
===============
Slave iface(s): 'slv2'
Node ID: 1
Run app: 1
```

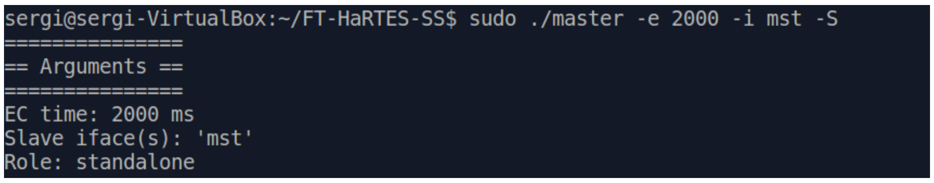
Figure 5.4: Start slave 2



```
sergi@sergi-VirtualBox:~/FT-HaRTES-SS$ sudo ./slave -i slv3 -n 4 -a 1
===============
== Arguments ==
===============
Slave iface(s): 'slv3'
Node ID: 4
Run app: 1
```

Figure 5.5: Start slave 3

When every slave is started, they do not do any action because they are waiting to receive FTT messages like TM or SDM. So the final step is to start the master (see figure 5.6) with the parameters of the Table 4.1, which will start sending the Trigger Messages:

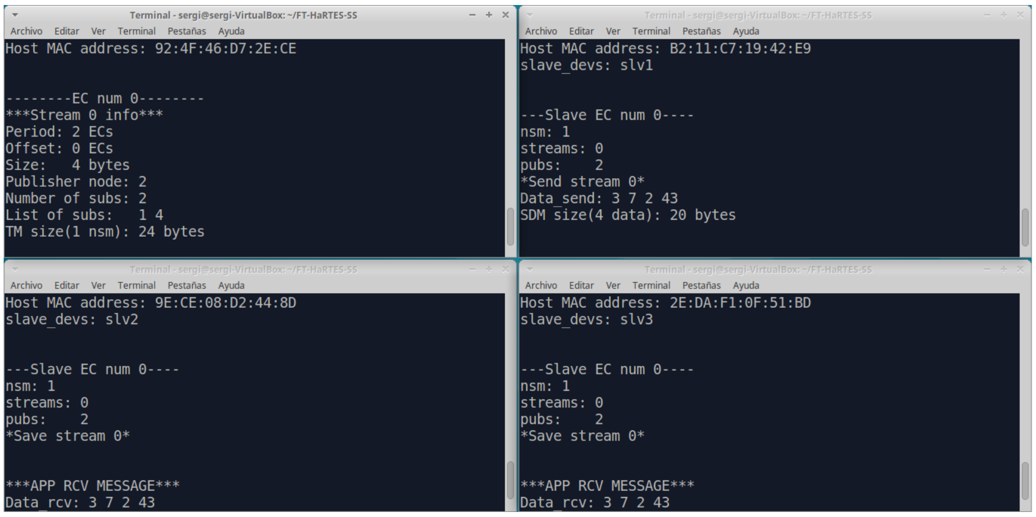


```
sergi@sergi-VirtualBox:~/FT-HaRTES-SS$ sudo ./master -e 2000 -i mst -S
===============
== Arguments ==
===============
EC time: 2000 ms
Slave iface(s): 'mst'
Role: standalone
```

Figure 5.6: Start master

Now that everything is running, the TM will be sent every 2 seconds (2000 ms). Every EC is identified by a sequence number, which defines the periodicity of every stream. As we previously introduced, in this experiment there is only one stream, which is the stream 0 with a period of 2 and an offset of 0. Consequently, the transmission of the messages associated to this stream will be triggered in every even EC.

To see the results at a glance, all four windows are placed in the screen at the same time, as we can see in the figures 5.7 and 5.8; being the top-left corner the master, top-right corner the slave 1, bottom-left corner the slave 2 and in the bottom-right corner the slave 3.


Terminal - sergi@sergi-VirtualBox: ~/FT-HaRTES-SS

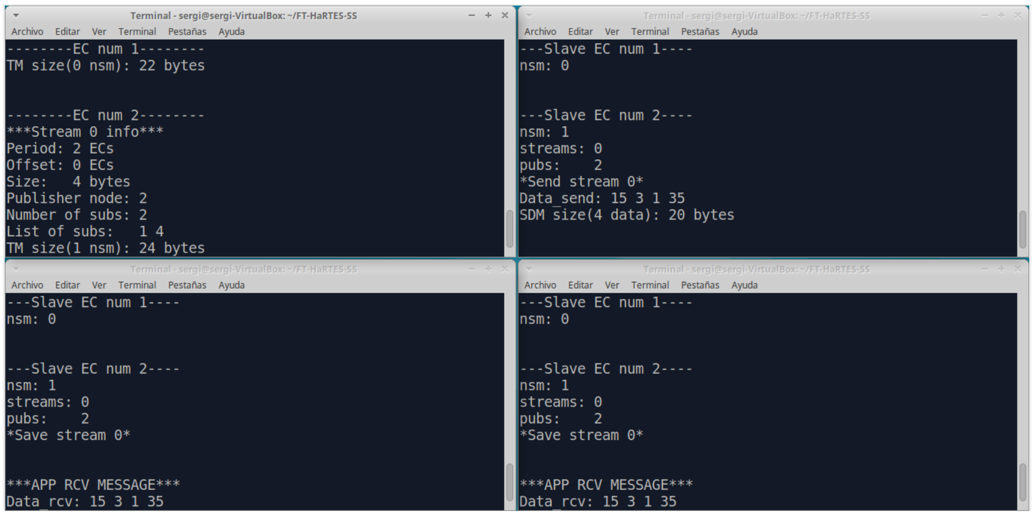
```
Host MAC address: 92:4F:46:D7:2E:CE


--------EC num 0--------
***Stream 0 info***
Period: 2 ECs
Offset: 0 ECs
Size:   4 bytes
Publisher node: 2
Number of subs: 2
List of subs:   1 4
TM size(1 nsm): 24 bytes
```

```
Host MAC address: B2:11:C7:19:42:E9
slave_devs: slv1


---Slave EC num 0----
nsm: 1
streams: 0
pubs:    2
*Send stream 0*
Data_send: 3 7 2 43
SDM size(4 data): 20 bytes
```

```
Host MAC address: 9E:CE:08:D2:44:8D
slave_devs: slv2


---Slave EC num 0----
nsm: 1
streams: 0
pubs:    2
*Save stream 0*


***APP RCV MESSAGE***
Data_rcv: 3 7 2 43
```

```
Host MAC address: 2E:DA:F1:0F:51:BD
slave_devs: slv3


---Slave EC num 0----
nsm: 1
streams: 0
pubs:    2
*Save stream 0*


***APP RCV MESSAGE***
Data_rcv: 3 7 2 43
```

Figure 5.7: Results EC=0


Terminal - sergi@sergi-VirtualBox: ~/FT-HaRTES-SS

```
--------EC num 1--------
TM size(0 nsm): 22 bytes


--------EC num 2--------
***Stream 0 info***
Period: 2 ECs
Offset: 0 ECs
Size:   4 bytes
Publisher node: 2
Number of subs: 2
List of subs:   1 4
TM size(1 nsm): 24 bytes
```

```
---Slave EC num 1----
nsm: 0


---Slave EC num 2----
nsm: 1
streams: 0
pubs:    2
*Send stream 0*
Data_send: 15 3 1 35
SDM size(4 data): 20 bytes
```

```
---Slave EC num 1----
nsm: 0


---Slave EC num 2----
nsm: 1
streams: 0
pubs:    2
*Save stream 0*


***APP RCV MESSAGE***
Data_rcv: 15 3 1 35
```

```
---Slave EC num 1----
nsm: 0


---Slave EC num 2----
nsm: 1
streams: 0
pubs:    2
*Save stream 0*


***APP RCV MESSAGE***
Data_rcv: 15 3 1 35
```

Figure 5.8: Results EC=1 and EC=2

The results of the above figures show printed text which was generated during the execution of the program. When the master prints "—EC num X—" it indicates the start of a new EC; showing right below the information of the scheduled streams that contain the TM. Additionally, when the TM is sent through Ethernet, it prints the TM size in bytes.

About the slaves, when a slave prints "—Slave EC num X—" it means that the slave has received the TM, and as the own TM contains the sequence number of the EC, the slaves just have to read it to know in which EC they are. Below that message, slaves print the rest of the content of the TM, which are: number of synchronous messages (nsm), list of stream IDs sent in the current EC and their respective publishers. Slaves can also print "*Send stream X*" or "*Save stream X*" to indicate that they have sent or received an SDM.

In the figure 5.7, it is displayed the output of the EC 0, in which the master indicates that the TM only contains 1 nsm, being the stream 0 the one that is going to be transmitted. All the three slaves print that they have received this same TM in the current EC. This particular TM indicates that the stream 0 has to be sent by the slave with node ID=2. The slave that has this ID is the slave 1, so it indicates with a "*Send stream 0*" that is going to send it with the corresponding data (4 bytes). Immediately after, the slaves 2 (ID=1) and 3 (ID=4) receive and save the SDM sent by the slave 1, we can see it because both of them print "*Save stream 0*" as both are subscribed to the stream 0. The last print is from their respective applications, showing the received data for this stream, which coincides with the original data sent by the publisher.

In the figure 5.8 we can observe the next EC, which is the EC 1. In this EC the master does not have any stream scheduled to be sent, so the total number of nsm is zero. Slaves receive the info but do not have to do anything more. The sequence is repeated for all the following EC numbers, as we can see for example in the EC 2, which does the same as the EC 0 but with different data values.

5.2 Advanced setup

The goal in this final experiment is ensure that the project works in a real environment, with more communication between the slaves and also testing the flexibility. The setup in this case will be with physical components as shown in the figure 5.9:

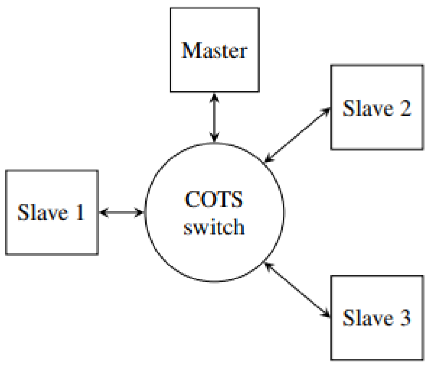


Figure 5.9: Experiment setup (reprinted as it appears in [18] Fig. 1a)

This setup is composed by a master and three slaves. The hardware of the master is a desktop computer with a processor Intel Core i7-4770 CPU @ 3,40 GHz to take the most advantage of the parallelism and 8 GB of RAM. Additionally, this PC has a network card Intel Ethernet I350-T4 that allows modifying low-level communication parameters. The slaves have all the same hardware, a slave is an embedded computer with a processor Intel Atom CPU D525 @ 1,80 GHz and 2 GB of RAM.

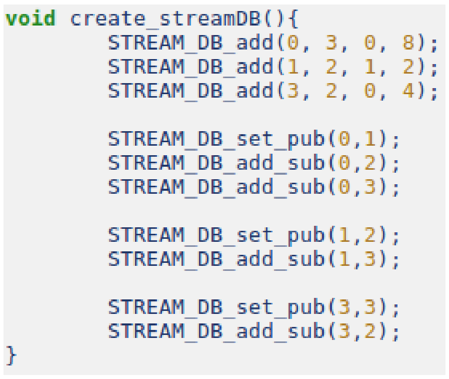
All the hardware components can be accessed in remote mode from any computer connected in the network. It is important to clarify that this setup is not designed and mounted by myself, as it has been used for the development of other projects.

For the three slaves that form this setup, they have the following role:

- Slave 1 (ID=1): Publisher of the stream 0 and 2. Not subscribed to any stream
- Slave 2 (ID=2): Publisher of the stream 1. Subscribed to stream 0, 2* and 3
- Slave 3 (ID=3): Publisher of the stream 3. Subscribed to stream 0, 1 and 2

2*: the slave 2 is subscribed 1 EC after the stream is added in the Stream DB

The Stream DB will be created at the start (see figure 5.10) and will be updated in every Elementary Cycle (see figure 5.11). In this project, the update of the Stream DB is checked at the start of a new EC, before the Scheduling module. This update wants to prove some future mechanisms that provide flexibility in communications.



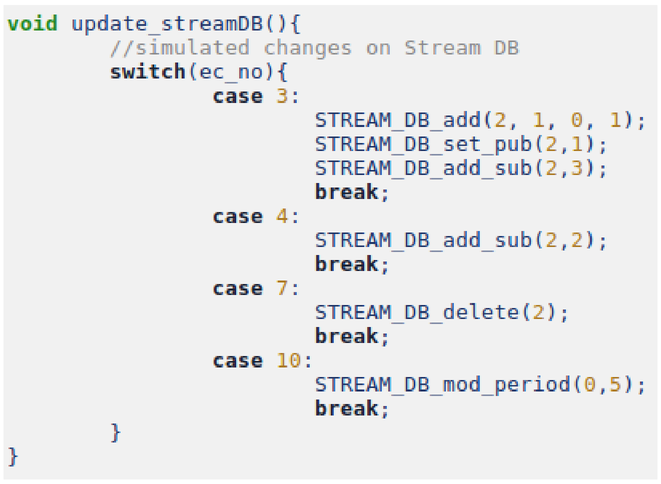
```
void create_streamDB(){
        STREAM_DB_add(0, 3, 0, 8);
        STREAM_DB_add(1, 2, 1, 2);
        STREAM_DB_add(3, 2, 0, 4);

        STREAM_DB_set_pub(0,1);
        STREAM_DB_add_sub(0,2);
        STREAM_DB_add_sub(0,3);

        STREAM_DB_set_pub(1,2);
        STREAM_DB_add_sub(1,3);

        STREAM_DB_set_pub(3,3);
        STREAM_DB_add_sub(3,2);
}
```

Figure 5.10: Stream DB initial creation

We will configure the streams so that there are multiple ways of communication between the slaves:

- Multicast: a slave will send a stream to multiple slaves
- Unicast: a slave will send a stream to another slave
- Bidirectional: two slaves will interact between them, for example to simulate a synchronization. To satisfy this communication they need two streams, one per slave.

To carry out this experiment four streams will be used. The streams 0, 1 and 3 will be added from the start, but the stream 2 will appear in EC 3. This new stream could mean that a new event occurred in the system and it needs to transmit this information



```
void update_streamDB(){
        //simulated changes on Stream DB
        switch(ec_no){
                case 3:
                        STREAM_DB_add(2, 1, 0, 1);
                        STREAM_DB_set_pub(2,1);
                        STREAM_DB_add_sub(2,3);
                        break;
                case 4:
                        STREAM_DB_add_sub(2,2);
                        break;
                case 7:
                        STREAM_DB_delete(2);
                        break;
                case 10:
                        STREAM_DB_mod_period(0,5);
                        break;
        }
}
```

Figure 5.11: Dynamic update of Stream DB

to the slaves, but it starts only having one subscriber (node ID=3). Then in the next EC it will be added one subscriber more (node ID=2), and later, in the EC 7, this same stream will be deleted.

Also, we can observe that in the EC 10 the period of the stream 0 will be changed to 5 ECs, so it will be transmitted less frequently compared to the initial period, which was 3 ECs. This change could mean that the requirements of the system changed, and in the consequence the period of this stream was modified.

With this configuration, we have that the slave 1 is performing a multicast communication using the stream 0, received by slaves 2 and 3. At the same time, the slaves 2 and 3 are alternating EC to communicate between them with the streams 1 and 3, this is the bidirectional communication. And finally there is the special case for the stream 2, which appears online in the EC 3 but only with the slave 3 subscribed; as there is only one transmitter and one receiver, it is a unicast communication. But then in the next EC, with the new subscription of the slave 2, the stream 2 transforms into multicast, as there is more than one receiver. Finally this stream is deleted in the EC 7, meaning that its last appearance is in the EC 6. The interactions between the slaves are represented in the figure 5.12:

Each of the slaves executes a custom application, which are the app1, app2 and app3, respectively. These applications are just a combination of the simple Send and Receive applications (see section 4.3.4), because in this case a slave will have to send or receive several streams simultaneously. Since we are only interested in checking if the SDMs are correctly transmitted and received, they contain random data. These applications can be seen with more detail in Annex B.

At this point the experiment is ready to start. We proceed to compile the project in the hardware of the respective master and slaves. The master and the slaves are started with the commands that we already know from the Tables 4.1 and 4.2. These are the commands executed for every one of the components:

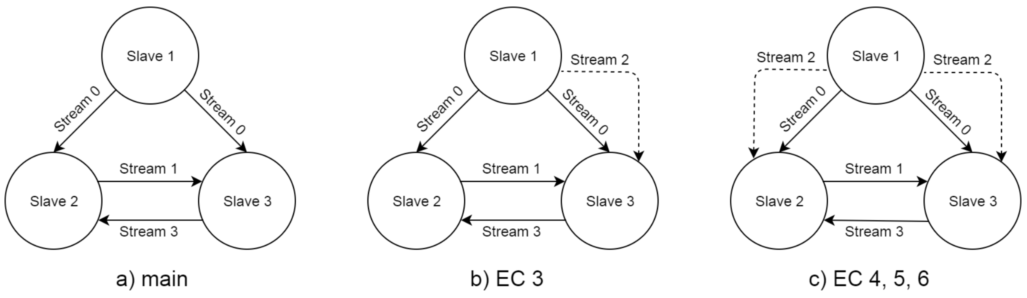


Figure 5.12: Interaction between slaves

- Master → sudo ./master -e 1000 -i slave4 -S
- Slave 1 → sudo ./slave -n 1 -a 1 -i eth2
- Slave 2 → sudo ./slave -n 2 -a 2 -i eth2
- Slave 3 → sudo ./slave -n 3 -a 3 -i eth2

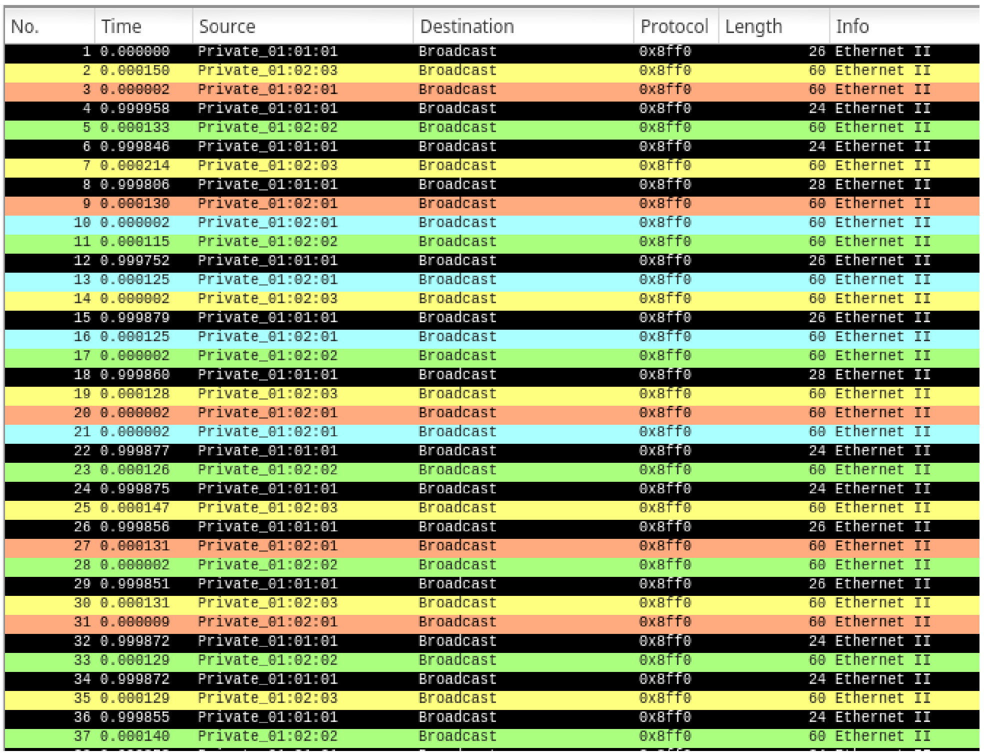
We decided that every EC lasts 1 second (1000 ms) and, for the slaves, we use intuitive node and application IDs to avoid confusions with them. Also we can observe that all the slaves use the same link, which is the "eth2". This is because all three slaves have identical hardware and, thus, the name of the interface connected to the switch is the same.

With this experiment we obtain two different kinds of outputs: one is the own terminal text output and the other is the Wireshark network capture that registered all the Ethernet messages transmitted during the execution

The output of the terminal is very useful because we can observe directly on the console all the details on what is happening while it is executing, but as the complexity raised compared to the first experiment, it would be tedious to verify all the cases in this way, although this output can be found in the Annex C.

Instead, we will use Wireshark because it will show all the FTT messages sent by the master and the slaves. Just registering the activity of the master is enough because all the messages are sent in Broadcast mode, so including the TM sent by itself, it will also receive the SDM messages sent by the slaves. For a better visual comparison, we can use the Wireshark coloring rules to color the messages. The TM will be colored in black and the stream 0, 1, 2 and 3 will be colored in red, green, blue and yellow, respectively. With Wireshark all the interesting output can be showed in just 2 screenshots, which are the figures 5.13 and 5.14:

With the Wireshark output we can see very clearly the messages thanks to the colors displayed. Remember that every EC start with a TM, which is a black message in the figures. Below every TM, there are one or several SDMs properly colored as we previously described, according with the Stream DB.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
2	0.000150	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
3	0.000002	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
4	0.999958	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
5	0.000133	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
6	0.999846	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
7	0.000214	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
8	0.999806	Private_01:01:01	Broadcast	0x8ff0	28	Ethernet II
9	0.000130	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
10	0.000002	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
11	0.000115	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
12	0.999752	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
13	0.000125	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
14	0.000002	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
15	0.999879	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
16	0.000125	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
17	0.000002	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
18	0.999860	Private_01:01:01	Broadcast	0x8ff0	28	Ethernet II
19	0.000128	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
20	0.000002	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
21	0.000002	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
22	0.999877	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
23	0.000126	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
24	0.999875	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
25	0.000147	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
26	0.999856	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
27	0.000131	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
28	0.000002	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
29	0.999851	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
30	0.000131	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
31	0.000009	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
32	0.999872	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
33	0.000129	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
34	0.999872	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
35	0.000129	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
36	0.999855	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
37	0.000140	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II

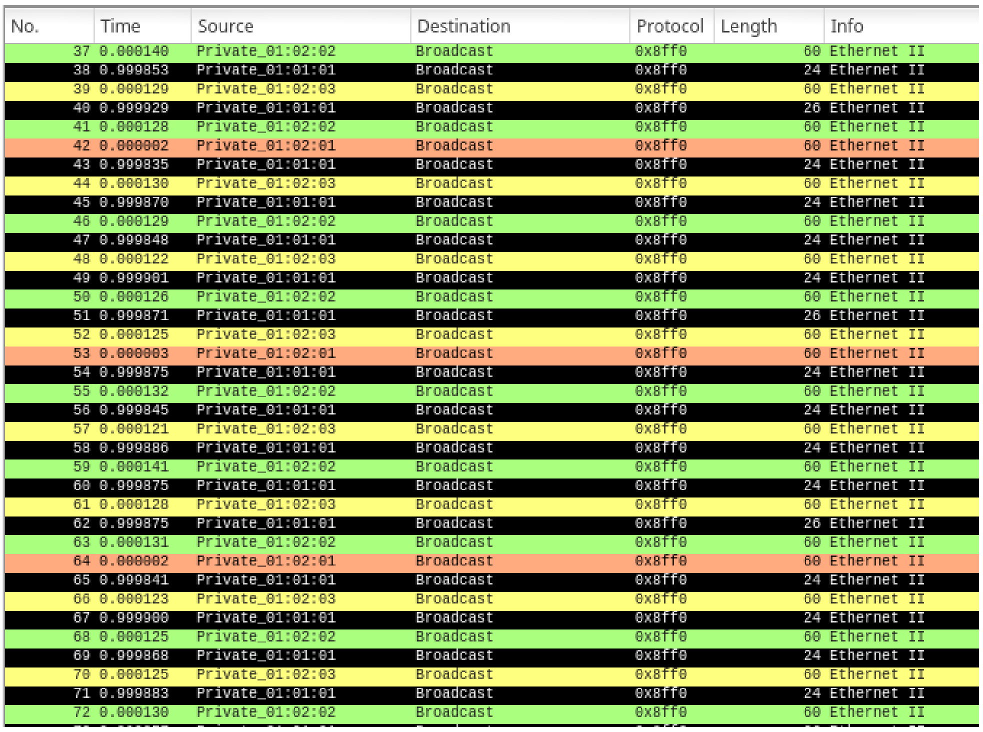
Figure 5.13: Wireshark global output (1)

In figure 5.13 we show from EC 0 to 13, that is, when all the stream DB updates take place. More concretely, we can observe that the stream 1 (yellow) and stream 3 (green) is alternating every 2 ECs. Moreover, the stream 0 (red) is sent with a period of 3 ECs, but in the EC 10 the period of this stream is changed to 5 ECs, that is why the stream 0 is sent in the EC 9 and in the EC 10, as expected. Also, the stream 2 (blue) appears right when expected, at the EC 3, and lasts with a period of 1 EC until the EC 6.

Finally, in the figure 5.14 we can see the pattern that will continually follow the experiment, as there are not more changes programmed to the Stream DB. The stream 1 and 3 keep alternating every EC and the stream 0 (red) is sent less frequently, every 5 ECs.

Another interesting information that we can observe from the Wireshark output is related to the 'Time' column. This column shows for every message the relative time (in microseconds) compared to the previous one. Observing only the Trigger Messages, its duration is always approximately 1 second, which is the duration of a Elementary Cycle.

But we can actually go further with this information and analyze which is the average delay (in microseconds) in the slaves to receive the TM, decode it and send the corresponding SDM. This time could also be called the response time of the slaves. To make this measurement we just have to calculate the mean between several values of a particular SDM, which is sent by a particular slave. We can indeed observe the node ID



No.	Time	Source	Destination	Protocol	Length	Info
37	0.000140	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
38	0.999853	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
39	0.000129	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
40	0.999929	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
41	0.000128	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
42	0.000002	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
43	0.999835	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
44	0.000130	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
45	0.999870	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
46	0.000129	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
47	0.999848	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
48	0.000122	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
49	0.999901	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
50	0.000126	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
51	0.999871	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
52	0.000125	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
53	0.000003	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
54	0.999875	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
55	0.000132	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
56	0.999845	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
57	0.000121	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
58	0.999886	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
59	0.000141	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
60	0.999875	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
61	0.000128	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
62	0.999875	Private_01:01:01	Broadcast	0x8ff0	26	Ethernet II
63	0.000131	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
64	0.000002	Private_01:02:01	Broadcast	0x8ff0	60	Ethernet II
65	0.999841	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
66	0.000123	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
67	0.999900	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
68	0.000125	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II
69	0.999868	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
70	0.000125	Private_01:02:03	Broadcast	0x8ff0	60	Ethernet II
71	0.999883	Private_01:01:01	Broadcast	0x8ff0	24	Ethernet II
72	0.000130	Private_01:02:02	Broadcast	0x8ff0	60	Ethernet II

Figure 5.14: Wireshark global output (2)

in the last digit of the source MAC. The results are available in Table 5.1.

Slave	**Delay** (μs)
1	126
2	131
3	124

Table 5.1: Response time of the slaves

The slaves have the same hardware so the response time should be similar between them. The mean response time of a slave is around 127 μs, which is faster enough to ensure Elementary Cycles of 1 ms (minimum EC time).

CHAPTER 6

CONCLUSIONS

In this chapter we expose the conclusions of the project. First, we will make a summary of the project current state and all the goals accomplished. Finally, it is proposed a future work that would improve the functionalities of the project.

6.1 Summary

This project consisted in the implementation of a basic FTT network on top of Ethernet. The outcome is a functional prototype with a master, scheduling the messages that the slaves have to send in time periods called ECs; and the slaves sending those messages that the master is indicating for every EC. The data contained in the slave messages has as source or destination the applications that the own slaves are executing, which represent a particular task that the slave is performing in the system.

To ensure the correct functioning of the project we made a functional verification using both a virtual and a real environment. In the virtual environment a single computer was used to execute the master and several slaves interconnect by means of a virtual switch. This configuration was very handy to use during the phase of implementation. In the real environment the master and each of the slaves was executed in a dedicated computer and all of them were connected to a legacy switch. This configuration is more realistic than the virtual one and allowed us to take real measures of the performance. The results obtained from the experiments showed that the implementation operates as expected, so it is ready to be expanded with future additions to reach the objectives of the DFT4FTT final goal.

Finally, I want to say that this project has been an overall great academic experience due to the new knowledge acquired and the more it can be obtained regarding the topic we are dealing with. Also, the decision of writing this report in English was to follow the line of the other related FTT projects, in this way it can be fully understandable for anyone that knows this language, although it is not my native language.

Concluding, this project is only a fraction of a bigger project that wants to reach a new functional protocol for highly-reliable systems. It is motivating for me to discover the laboratory workplace where all these investigations happen and of course, be part of it.

6.2 Future work

This project can be expanded to make improvements or add more functionalities. Some of the future additions that can be made to the current project are the following:

- *Extend the App-Slave Interface:* More functions could be added in the Interface Layer of the slave to let the applications know more about the state of the system. For instance, applications might want to know which is the EC duration, when is the exact time EC starts or in which EC they are currently. These functions would improve the applications capabilities because they could do the tasks according to the EC defined in the system..

- *Implement the transmission of Asynchronous Data Messages (ADMs):* In the current project we just use the TM and the SW in the EC. But as we explained in the section 2, the EC has an Asynchronous Window (AW) that it was not used in this project. In the AW the ADMs could be sent. The difference compared to the SDMs is that these messages are event-triggered, for instance, an alarm. To make this point, we would have to consider in the Slave Core, the possibility to convey this kind of messages after all the SDMs are sent. Additionally, streams with different parameters would need to be defined. The period and offset are not useful parameters in an event-triggered message, instead, we could set a priority and a deadline, measured in ECs.

- *Implement the dynamic management of the communication requirements:* Right now the management of the communication requirements is static, as it can only be modified inside the code. The next step would be to implement the mechanisms that let the slaves ask for changes in the stream DB. Consequently, we would need to create the SRM and MCM types of FTT messages. As a result of this implementation, we would also separate the stream DB of the master and the slaves, being the one in the slaves more reduced and personalized, as it only would contain the streams they are publishers or subscribers.

- *Implement fault tolerance mechanisms:* One of the first fault tolerance mechanisms that could be implemented is the Reliable TM mechanism, based on the replication of the TM in k copies. This replication is done because, if the TM is not transmitted successfully, the slaves would misunderstand what to do in the current EC, resulting in a failure of the system. Due to the replicated TM, the slaves need to coordinate the reception of the TMs and determine when they have to start sending their messages in the SW.

Finally, as we already know, this project will be expanded to reach new objectives, which is having the DFT4FTT protocol in working conditions. All the mentioned points in this section should help with this final goal, as they can even be considered previous steps to start with the full implementation of DFT4FTT.

APPENDIX A

Annex A: The DFT4FTT architecture

The scheme shown in figure A.1 represents the architecture of the DTF4FTT project. As the project is in development, this is not the definitive version, but it is a good approach of the architecture that will be finally implemented. We will not focus on a detailed explanation of the scheme, just a quick explanation.

The master is structured in Interface Layer, Core Layer and Ethernet Layer. Additionally, there is a Master Manager which is capable of modifying the Stream DB externally. The Interface Layer is capable of making the communication possible between the Master Manager and the Core Layer. The Core Layer is the layer that contains the behavior of the master. And the Ethernet Layer is an implementation of a software switch.

The slaves are structured in FT4FTT Layer, Interface Layer, Core Layer and Ethernet Layer. Moreover, the slaves execute Applications which make a certain task in the system. The FT4FTT Layer has Transmission and Reception modules that let the Applications communicate with the Core Layer. The Interface Layer of the slave abstracts the FTT services for the Applications. The Core Layer has all the logic behavior of the slave. Finally, the Ethernet Layer is an implementation of a software switch configured specially for the slaves.

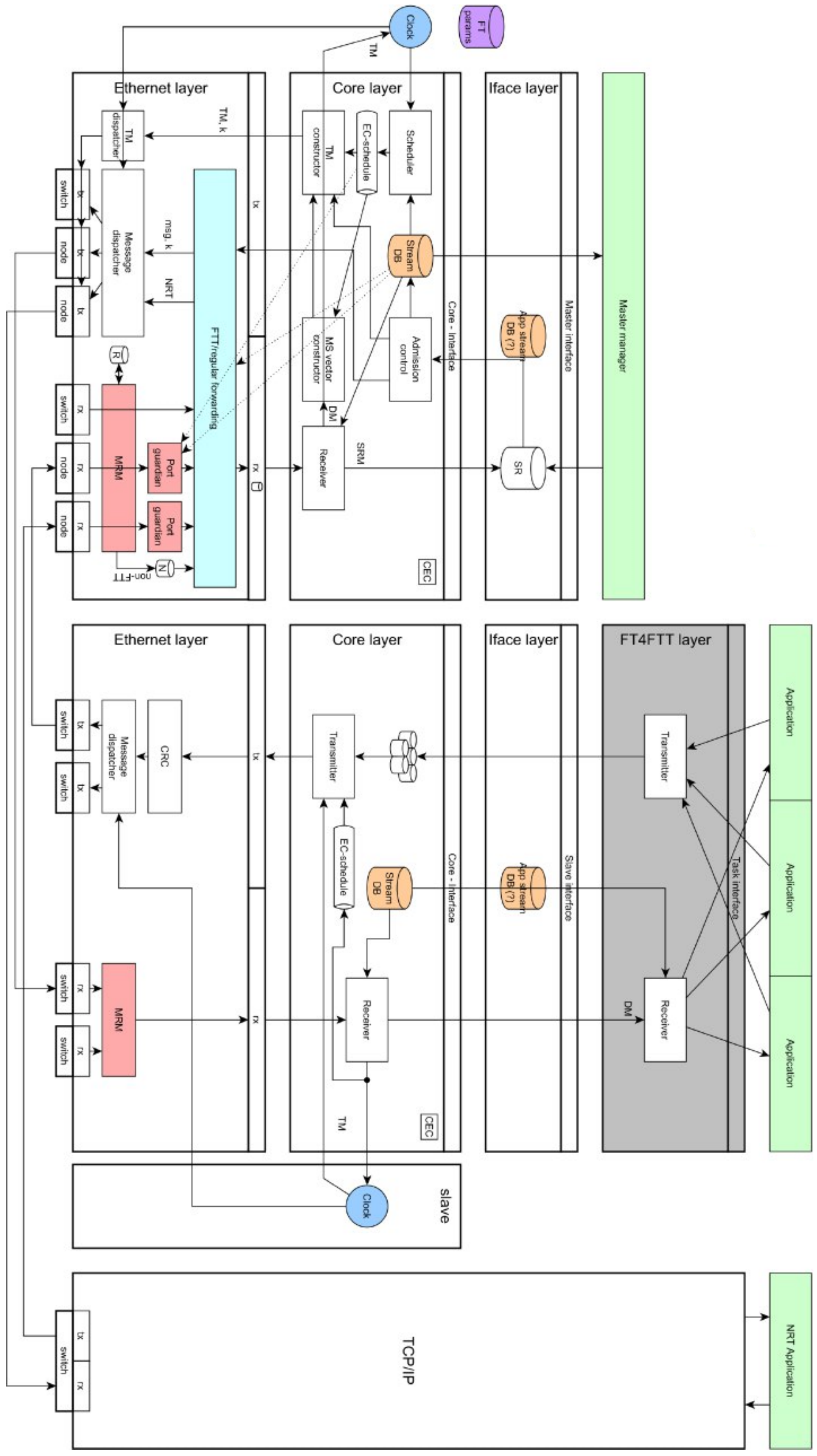


Figure A.1: DFT4FTT architecture

APPENDIX B

Annex B: Apps of the second experiment

The app IDs (configured in "node_apps.c" file) are the following:

```
void start_app(uint8_t app_id){
    switch(app_id){
        case 1: start_app1_app(); break;
        case 2: start_app2_app(); break;
        case 3: start_app3_app(); break;
        default : break;
    }
}

void wait_app(uint8_t app_id){
    switch(app_id){
        case 1: wait_app1_app(); break;
        case 2: wait_app2_app(); break;
        case 3: wait_app3_app(); break;
        default : break;
    }
}
```

B.1 App 1

```
void *app1_app()
{
    uint8_t buffer_send1[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_send1_size;
    uint8_t send1_stream_id=0;

    uint8_t buffer_send2[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_send2_size;
    uint8_t send2_stream_id=22;

    while(1){
        //generate data for stream 0
        buffer_send1_size=8;
        buffer_send1[0] = rand() % 25;
        buffer_send1[1] = rand() % 8;
        buffer_send1[2] = rand() % 3;
        buffer_send1[3] = rand() % 62;
        buffer_send1[4] = rand() % 25;
        buffer_send1[5] = rand() % 8;
        buffer_send1[6] = rand() % 3;
        buffer_send1[7] = rand() % 62;

        FTT_IFACE_send_app(
            send1_stream_id,
            buffer_send1,
            buffer_send1_size
        );

        //generate data for stream 2
        buffer_send2_size=1;
        buffer_send2[0] = rand() % 10 + 50;

        FTT_IFACE_send_app(
            send2_stream_id,
            buffer_send2,
            buffer_send2_size
        );

        usleep(300*1000); // wait 300 ms
    }
    pthread_exit(NULL);
}
```

B.2 App 2

```
void *app2_app()
{
    bool blocking = false;

    uint8_t buffer_rcv1[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_rcv1_size;
    uint8_t rcv1_stream_id=0;

    uint8_t buffer_rcv2[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_rcv2_size;
    uint8_t rcv2_stream_id=2;

    uint8_t buffer_rcv3[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_rcv3_size;
    uint8_t rcv3_stream_id=3;

    uint8_t buffer_send[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_send_size;
    uint8_t send_stream_id=1;

    while(1){
        //receive stream 0
        FTT_IFACE_recv_app(
            rcv1_stream_id, buffer_rcv1,
            &buffer_rcv1_size, blocking
        );
        if(buffer_rcv1_size>0){
            print_recv_data(rcv1_stream_id, buffer_rcv1, buffer_rcv1_size);
            buffer_rcv1_size=0;
        }

        //receive stream 2
        FTT_IFACE_recv_app(
            rcv2_stream_id, buffer_rcv2,
            &buffer_rcv2_size, blocking
        );
        if(buffer_rcv2_size>0){
            print_recv_data(rcv2_stream_id, buffer_rcv2, buffer_rcv2_size);
            buffer_rcv2_size=0;
        }

        //receive stream 3
        FTT_IFACE_recv_app(
            rcv3_stream_id, buffer_rcv3,
            &buffer_rcv3_size, blocking
        );
        if(buffer_rcv3_size>0){
            print_recv_data(rcv3_stream_id, buffer_rcv3, buffer_rcv3_size);
            buffer_rcv3_size=0;
        }

        //generate data for stream 1
        buffer_send_size=2;
        buffer_send[0] = rand() % 5;
        buffer_send[1] = rand() % 3;

        FTT_IFACE_send_app(send_stream_id, buffer_send, buffer_send_size);

        usleep(100*1000); // wait 100 ms
    }
    pthread_exit(NULL);
}

static void print_recv_data(
    uint8_t rcv_stream_id,
    uint8_t buffer_rcv[],
    uint16_t buffer_rcv_size
){
    printf("\n\n***APP2␣RCV␣MESSAGE␣%u***\n", rcv_stream_id);
    printf("Data_rcv:␣");
    for(uint8_t i=0; i<buffer_rcv_size; i++){
        printf("%u␣",buffer_rcv[i]);
    }
    printf("\n");
}
```

B.3 App 3

```
void *app3_app()
{
    bool blocking = false;

    uint8_t buffer_rcv1[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_rcv1_size;
    uint8_t rcv1_stream_id=0;

    uint8_t buffer_rcv2[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_rcv2_size;
    uint8_t rcv2_stream_id=1;

    uint8_t buffer_rcv3[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_rcv3_size;
    uint8_t rcv3_stream_id=2;

    uint8_t buffer_send[ETH_MAX_FRAME_SIZE];
    uint16_t buffer_send_size;
    uint8_t send_stream_id=3;

    while(1){
        //receive stream 0
        FTT_IFACE_recv_app(
            rcv1_stream_id, buffer_rcv1,
            &buffer_rcv1_size, blocking
        );
        if(buffer_rcv1_size>0){
            print_recv_data(rcv1_stream_id, buffer_rcv1, buffer_rcv1_size);
            buffer_rcv1_size=0;
        }

        //receive stream 1
        FTT_IFACE_recv_app(
            rcv2_stream_id, buffer_rcv2,
            &buffer_rcv2_size, blocking
        );
        if(buffer_rcv2_size>0){
            print_recv_data(rcv2_stream_id, buffer_rcv2, buffer_rcv2_size);
            buffer_rcv2_size=0;
        }

        //receive stream 2
        FTT_IFACE_recv_app(
            rcv3_stream_id, buffer_rcv3,
            &buffer_rcv3_size, blocking
        );
        if(buffer_rcv3_size>0){
            print_recv_data(rcv3_stream_id, buffer_rcv3, buffer_rcv3_size);
            buffer_rcv3_size=0;
        }

        //generate data for stream 3
        buffer_send_size=2;
        buffer_send[0] = rand() % 5;
        buffer_send[1] = rand() % 3;

        FTT_IFACE_send_app(send_stream_id, buffer_send, buffer_send_size);

        usleep(100*1000); // wait 100 ms
    }
    pthread_exit(NULL);
}

static void print_recv_data(
    uint8_t rcv_stream_id,
    uint8_t buffer_rcv[],
    uint16_t buffer_rcv_size
){
    printf("\n\n***APP3␣RCV␣MESSAGE␣%u***\n", rcv_stream_id);
    printf("Data_rcv:␣");
    for(uint8_t i=0; i<buffer_rcv_size; i++){
        printf("%u␣",buffer_rcv[i]);
    }
    printf("\n");
}
```

APPENDIX C

ANNEX C: TERMINAL EMULATOR OUTPUT TEXT IN SECOND EXPERIMENT

C.1 Master

```
===============
== Arguments ==
===============
EC time: 1000 ms
Slave iface(s): 'slave4'
Role: standalone
************************************************************
Successfully opened socket: 3
Successfully got interface index: 2
Host MAC address: 00:11:11:11:11:08

----------EC num 0----------
***Stream 0 info***
Period: 3 ECs
Offset: 0 ECs
Size:    8 bytes
Publisher node: 1
Number of subs: 2
List of subs:    2 3
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:    2 bytes
Publisher node: 3
Number of subs: 1
List of subs:    2
TM size(2 nsm): 26 bytes

----------EC num 1----------
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:    2 bytes
Publisher node: 2
Number of subs: 1
List of subs:    3
TM size(1 nsm): 24 bytes
```

```
---------EC num 2---------
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

---------EC num 3---------
***Stream 0 info***
Period: 3 ECs
Offset: 0 ECs
Size:   8 bytes
Publisher node: 1
Number of subs: 2
List of subs:   2 3
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
***Stream 2 info***
Period: 1 ECs
Offset: 0 ECs
Size:   1 bytes
Publisher node: 1
Number of subs: 1
List of subs:   3
TM size(3 nsm): 28 bytes

---------EC num 4---------
***Stream 2 info***
Period: 1 ECs
Offset: 0 ECs
Size:   1 bytes
Publisher node: 1
Number of subs: 2
List of subs:   3 2
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(2 nsm): 26 bytes

---------EC num 5---------
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
***Stream 2 info***
Period: 1 ECs
Offset: 0 ECs
Size:   1 bytes
Publisher node: 1
Number of subs: 2
List of subs:   3 2
TM size(2 nsm): 26 bytes

---------EC num 6---------
***Stream 0 info***
Period: 3 ECs
Offset: 0 ECs
Size:   8 bytes
Publisher node: 1
Number of subs: 2
List of subs:   2 3
***Stream 2 info***
Period: 1 ECs
Offset: 0 ECs
Size:   1 bytes
Publisher node: 1
Number of subs: 2
List of subs:   3 2
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(3 nsm): 28 bytes

---------EC num 7---------
```

```
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

————————EC num 8————————
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

————————EC num 9————————
***Stream 0 info***
Period: 3 ECs
Offset: 0 ECs
Size:   8 bytes
Publisher node: 1
Number of subs: 2
List of subs:   2 3
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(2 nsm): 26 bytes

————————EC num 10————————
***Stream 0 info***
Period: 5 ECs
Offset: 0 ECs
Size:   8 bytes
Publisher node: 1
Number of subs: 2
List of subs:   2 3
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(2 nsm): 26 bytes

————————EC num 11————————
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

————————EC num 12————————
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

————————EC num 13————————
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

————————EC num 14————————
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

————————EC num 15————————
```

```
***Stream 0 info***
Period: 5 ECs
Offset: 0 ECs
Size:   8 bytes
Publisher node: 1
Number of subs: 2
List of subs:   2 3
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(2 nsm): 26 bytes

————————EC num 16————————
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

————————EC num 17————————
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

————————EC num 18————————
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

————————EC num 19————————
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

————————EC num 20————————
***Stream 0 info***
Period: 5 ECs
Offset: 0 ECs
Size:   8 bytes
Publisher node: 1
Number of subs: 2
List of subs:   2 3
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(2 nsm): 26 bytes

————————EC num 21————————
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

————————EC num 22————————
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

————————EC num 23————————
```

```
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

---------EC num 24---------
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

---------EC num 25---------
***Stream 0 info***
Period: 5 ECs
Offset: 0 ECs
Size:   8 bytes
Publisher node: 1
Number of subs: 2
List of subs:   2 3
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(2 nsm): 26 bytes

---------EC num 26---------
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

---------EC num 27---------
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes

---------EC num 28---------
***Stream 3 info***
Period: 2 ECs
Offset: 0 ECs
Size:   2 bytes
Publisher node: 3
Number of subs: 1
List of subs:   2
TM size(1 nsm): 24 bytes

---------EC num 29---------
***Stream 1 info***
Period: 2 ECs
Offset: 1 ECs
Size:   2 bytes
Publisher node: 2
Number of subs: 1
List of subs:   3
TM size(1 nsm): 24 bytes
```

C.2 Slave 1

```
===============
== Arguments ==
===============
Slave iface(s): 'eth2'
Node ID: 1
Run app: 1
************************************************************
slave_cnt: 1
Successfully opened socket: 3
Successfully got interface index: 4
Host MAC address: 00:30:18:B0:0D:1A
slave_devs: eth2

---Slave EC num 0----
nsm: 2
streams: 0 3
pubs:    1 3
*Send stream 0*
Data_send: 17 1 1 2 13 4 1 10
SDM size(8 data): 24 bytes

---Slave EC num 1----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 2----
nsm: 1
streams: 3
pubs:    3

---Slave EC num 3----
nsm: 3
streams: 0 1 2
pubs:    1 2 1
*Send stream 0*
Data_send: 7 6 0 10 2 3 0 36
SDM size(8 data): 24 bytes
*Send stream 2*
Data_send: 52
SDM size(1 data): 17 bytes

---Slave EC num 4----
nsm: 2
streams: 2 3
pubs:    1 3
*Send stream 2*
Data_send: 58
SDM size(1 data): 17 bytes

---Slave EC num 5----
nsm: 2
streams: 1 2
pubs:    2 1
*Send stream 2*
Data_send: 51
SDM size(1 data): 17 bytes

---Slave EC num 6----
nsm: 3
streams: 0 2 3
pubs:    1 1 3
*Send stream 0*
Data_send: 12 6 1 53 6 2 1 42
SDM size(8 data): 24 bytes
*Send stream 2*
Data_send: 52
SDM size(1 data): 17 bytes

---Slave EC num 7----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 8----
```

```
nsm: 1
streams: 3
pubs:    3

---Slave EC num 9-----
nsm: 2
streams: 0 1
pubs:    1 2
*Send stream 0*
Data_send: 0 2 1 20 24 5 1 21
SDM size(8 data): 24 bytes

---Slave EC num 10-----
nsm: 2
streams: 0 3
pubs:    1 3
*Send stream 0*
Data_send: 15 7 1 22 14 6 0 6
SDM size(8 data): 24 bytes

---Slave EC num 11-----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 12-----
nsm: 1
streams: 3
pubs:    3

---Slave EC num 13-----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 14-----
nsm: 1
streams: 3
pubs:    3

---Slave EC num 15-----
nsm: 2
streams: 0 1
pubs:    1 2
*Send stream 0*
Data_send: 17 5 1 21 2 3 2 4
SDM size(8 data): 24 bytes

---Slave EC num 16-----
nsm: 1
streams: 3
pubs:    3

---Slave EC num 17-----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 18-----
nsm: 1
streams: 3
pubs:    3

---Slave EC num 19-----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 20-----
nsm: 2
streams: 0 3
pubs:    1 3
*Send stream 0*
Data_send: 6 0 0 15 3 6 2 16
SDM size(8 data): 24 bytes

---Slave EC num 21-----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 22-----
nsm: 1
streams: 3
pubs:    3

---Slave EC num 23-----
nsm: 1
streams: 1
pubs:    2

---Slave EC num 24-----
nsm: 1
streams: 3
```

```
pubs:      3

−−−Slave EC num 25−−−−
nsm: 2
streams: 0 1
pubs:      1 2
∗Send stream 0∗
Data_send: 2 1 0 22 1 3 1 58
SDM size(8 data): 24 bytes

−−−Slave EC num 26−−−−
nsm: 1
streams: 3
pubs:      3
```

```
−−−Slave EC num 27−−−−
nsm: 1
streams: 1
pubs:      2

−−−Slave EC num 28−−−−
nsm: 1
streams: 3
pubs:      3

−−−Slave EC num 29−−−−
nsm: 1
streams: 1
pubs:      2
```

C.3 Slave 2

```
===============
== Arguments ==
===============
Slave iface(s): 'eth2'
Node ID: 2
Run app: 2
*******************************************************
slave_cnt: 1
Successfully opened socket: 3
Successfully got interface index: 4
Host MAC address: 00:30:18:AF:39:C2
slave_devs: eth2

---Slave EC num 0----
nsm: 2
streams: 0 3
pubs:     1 3
*Save stream 3*
*Save stream 0*

***APP2 RCV MESSAGE 0***
Data_rcv: 17 1 1 2 13 4 1 10

***APP2 RCV MESSAGE 3***
Data_rcv: 3 0

---Slave EC num 1----
nsm: 1
streams: 1
pubs:     2
*Send stream 1*
Data_send: 4 0
SDM size(2 data): 18 bytes

---Slave EC num 2----
nsm: 1
streams: 3
pubs:     3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 0 1

---Slave EC num 3----
nsm: 3
streams: 0 1 2
pubs:     1 2 1
*Send stream 1*
Data_send: 2 1
SDM size(2 data): 18 bytes
*Save stream 0*

***APP2 RCV MESSAGE 0***
Data_rcv: 7 6 0 10 2 3 0 36

---Slave EC num 4----
nsm: 2
streams: 2 3
pubs:     1 3
*Save stream 2*
*Save stream 3*

***APP2 RCV MESSAGE 2***
Data_rcv: 58

***APP2 RCV MESSAGE 3***
Data_rcv: 0 0

---Slave EC num 5----
nsm: 2
streams: 1 2
pubs:     2 1
*Send stream 1*
Data_send: 1 1
SDM size(2 data): 18 bytes
```

```
*Save stream 2*

***APP2 RCV MESSAGE 2***
Data_rcv: 51

---Slave EC num 6----
nsm: 3
streams: 0 2 3
pubs:    1 1 3
*Save stream 3*
*Save stream 0*
*Save stream 2*

***APP2 RCV MESSAGE 0***
Data_rcv: 12 6 1 53 6 2 1 42

***APP2 RCV MESSAGE 2***
Data_rcv: 52

***APP2 RCV MESSAGE 3***
Data_rcv: 4 1

---Slave EC num 7----
nsm: 1
streams: 1
pubs:    2
*Send stream 1*
Data_send: 2 0
SDM size(2 data): 18 bytes

---Slave EC num 8----
nsm: 1
streams: 3
pubs:    3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 0 1

---Slave EC num 9----
nsm: 2
streams: 0 1
pubs:    1 2
*Send stream 1*
Data_send: 4 2
SDM size(2 data): 18 bytes
*Save stream 0*

***APP2 RCV MESSAGE 0***
Data_rcv: 0 2 1 20 24 5 1 21

---Slave EC num 10----
nsm: 2
streams: 0 3
pubs:    1 3
*Save stream 3*
*Save stream 0*

***APP2 RCV MESSAGE 0***
Data_rcv: 15 7 1 22 14 6 0 6

***APP2 RCV MESSAGE 3***
Data_rcv: 0 2

---Slave EC num 11----
nsm: 1
streams: 1
pubs:    2
*Send stream 1*
Data_send: 4 1
SDM size(2 data): 18 bytes

---Slave EC num 12----
nsm: 1
streams: 3
pubs:    3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 3 2

---Slave EC num 13----
nsm: 1
streams: 1
pubs:    2
*Send stream 1*
Data_send: 0 0
SDM size(2 data): 18 bytes

---Slave EC num 14----
nsm: 1
streams: 3
pubs:    3
*Save stream 3*
```

```
***APP2 RCV MESSAGE 3***
Data_rcv: 0 2

---Slave EC num 15----
nsm: 2
streams: 0 1
pubs:    1 2
*Send stream 1*
Data_send: 4 2
SDM size(2 data): 18 bytes
*Save stream 0*

***APP2 RCV MESSAGE 0***
Data_rcv: 17 5 1 21 2 3 2 4

---Slave EC num 16----
nsm: 1
streams: 3
pubs:    3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 1 0

---Slave EC num 17----
nsm: 1
streams: 1
pubs:    2
*Send stream 1*
Data_send: 3 1
SDM size(2 data): 18 bytes

---Slave EC num 18----
nsm: 1
streams: 3
pubs:    3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 0 1

---Slave EC num 19----
nsm: 1
streams: 1
pubs:    2
*Send stream 1*
Data_send: 3 1
SDM size(2 data): 18 bytes

---Slave EC num 20----
nsm: 2
streams: 0 3
pubs:    1 3
*Save stream 3*
*Save stream 0*

***APP2 RCV MESSAGE 0***
Data_rcv: 6 0 0 15 3 6 2 16

***APP2 RCV MESSAGE 3***
Data_rcv: 2 0

---Slave EC num 21----
nsm: 1
streams: 1
pubs:    2
*Send stream 1*
Data_send: 0 0
SDM size(2 data): 18 bytes

---Slave EC num 22----
nsm: 1
streams: 3
pubs:    3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 2 2

---Slave EC num 23----
nsm: 1
streams: 1
pubs:    2
*Send stream 1*
Data_send: 4 2
SDM size(2 data): 18 bytes

---Slave EC num 24----
nsm: 1
streams: 3
pubs:    3
*Save stream 3*
```

```
***APP2 RCV MESSAGE 3***
Data_rcv: 4 0

---Slave EC num 25----
nsm: 2
streams: 0 1
pubs:     1 2
*Send stream 1*
Data_send: 0 2
SDM size(2 data): 18 bytes
*Save stream 0*

***APP2 RCV MESSAGE 0***
Data_rcv: 2 1 0 22 1 3 1 58

---Slave EC num 26----
nsm: 1
streams: 3
pubs:     3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 3 2

---Slave EC num 27----
nsm: 1
streams: 1
pubs:     2
*Send stream 1*
Data_send: 0 0
SDM size(2 data): 18 bytes

---Slave EC num 28----
nsm: 1
streams: 3
pubs:     3
*Save stream 3*

***APP2 RCV MESSAGE 3***
Data_rcv: 4 1

---Slave EC num 29----
nsm: 1
streams: 1
pubs:     2
*Send stream 1*
Data_send: 4 2
SDM size(2 data): 18 bytes
```

C.4 Slave 3

```
===============
== Arguments ==
===============
Slave iface(s): 'eth2'
Node ID: 3
Run app: 3
*******************************************************
slave_cnt: 1
Successfully opened socket: 3
Successfully got interface index: 4
Host MAC address: 00:30:18:B0:08:72
slave_devs: eth2

----Slave EC num 0-----
nsm: 2
streams: 0 3
pubs:     1 3
*Send stream 3*
Data_send: 3 0
SDM size(2 data): 18 bytes
*Save stream 0*

***APP3 RCV MESSAGE 0***
Data_rcv: 17 1 1 2 13 4 1 10

----Slave EC num 1-----
nsm: 1
streams: 1
pubs:     2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 4 0

----Slave EC num 2-----
nsm: 1
streams: 3
pubs:     3
*Send stream 3*
Data_send: 0 1
SDM size(2 data): 18 bytes

----Slave EC num 3-----
nsm: 3
streams: 0 1 2
pubs:     1 2 1
*Save stream 0*
*Save stream 2*
*Save stream 1*

***APP3 RCV MESSAGE 0***
Data_rcv: 7 6 0 10 2 3 0 36

***APP3 RCV MESSAGE 1***
Data_rcv: 2 1

***APP3 RCV MESSAGE 2***
Data_rcv: 52

----Slave EC num 4-----
nsm: 2
streams: 2 3
pubs:     1 3
*Send stream 3*
Data_send: 0 0
SDM size(2 data): 18 bytes
*Save stream 2*

***APP3 RCV MESSAGE 2***
Data_rcv: 58

----Slave EC num 5-----
nsm: 2
streams: 1 2
pubs:     2 1
```

```
*Save stream 2*
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 1 1

***APP3 RCV MESSAGE 2***
Data_rcv: 51

---Slave EC num 6----
nsm: 3
streams: 0 2 3
pubs:    1 1 3
*Send stream 3*
Data_send: 4 1
SDM size(2 data): 18 bytes
*Save stream 0*
*Save stream 2*

***APP3 RCV MESSAGE 0***
Data_rcv: 12 6 1 53 6 2 1 42

***APP3 RCV MESSAGE 2***
Data_rcv: 52

---Slave EC num 7----
nsm: 1
streams: 1
pubs:    2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 2 0

---Slave EC num 8----
nsm: 1
streams: 3
pubs:    3
*Send stream 3*
Data_send: 0 1
SDM size(2 data): 18 bytes

---Slave EC num 9----
nsm: 2
streams: 0 1
pubs:    1 2
*Save stream 0*
*Save stream 1*

***APP3 RCV MESSAGE 0***
Data_rcv: 0 2 1 20 24 5 1 21

***APP3 RCV MESSAGE 1***
Data_rcv: 4 2

---Slave EC num 10----
nsm: 2
streams: 0 3
pubs:    1 3
*Send stream 3*
Data_send: 0 2
SDM size(2 data): 18 bytes
*Save stream 0*

***APP3 RCV MESSAGE 0***
Data_rcv: 15 7 1 22 14 6 0 6

---Slave EC num 11----
nsm: 1
streams: 1
pubs:    2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 4 1

---Slave EC num 12----
nsm: 1
streams: 3
pubs:    3
*Send stream 3*
Data_send: 3 2
SDM size(2 data): 18 bytes

---Slave EC num 13----
nsm: 1
streams: 1
pubs:    2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 0 0

---Slave EC num 14----
```

```
nsm: 1
streams: 3
pubs:     3
*Send stream 3*
Data_send: 0 2
SDM size(2 data): 18 bytes

---Slave EC num 15----
nsm: 2
streams: 0 1
pubs:     1 2
*Save stream 1*
*Save stream 0*

***APP3 RCV MESSAGE 0***
Data_rcv: 17 5 1 21 2 3 2 4

***APP3 RCV MESSAGE 1***
Data_rcv: 4 2

---Slave EC num 16----
nsm: 1
streams: 3
pubs:     3
*Send stream 3*
Data_send: 1 0
SDM size(2 data): 18 bytes

---Slave EC num 17----
nsm: 1
streams: 1
pubs:     2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 3 1

---Slave EC num 18----
nsm: 1
streams: 3
pubs:     3
*Send stream 3*
Data_send: 0 1
SDM size(2 data): 18 bytes

---Slave EC num 19----
nsm: 1
streams: 1
pubs:     2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 3 1

---Slave EC num 20----
nsm: 2
streams: 0 3
pubs:     1 3
*Send stream 3*
Data_send: 2 0
SDM size(2 data): 18 bytes
*Save stream 0*

***APP3 RCV MESSAGE 0***
Data_rcv: 6 0 0 15 3 6 2 16

---Slave EC num 21----
nsm: 1
streams: 1
pubs:     2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 0 0

---Slave EC num 22----
nsm: 1
streams: 3
pubs:     3
*Send stream 3*
Data_send: 2 2
SDM size(2 data): 18 bytes

---Slave EC num 23----
nsm: 1
streams: 1
pubs:     2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 4 2

---Slave EC num 24----
nsm: 1
```

```
streams: 3
pubs:    3
*Send stream 3*
Data_send: 4 0
SDM size(2 data): 18 bytes

---Slave EC num 25----
nsm: 2
streams: 0 1
pubs:    1 2
*Save stream 1*
*Save stream 0*

***APP3 RCV MESSAGE 0***
Data_rcv: 2 1 0 22 1 3 1 58

***APP3 RCV MESSAGE 1***
Data_rcv: 0 2

---Slave EC num 26----
nsm: 1
streams: 3
pubs:    3
*Send stream 3*
Data_send: 3 2
SDM size(2 data): 18 bytes

---Slave EC num 27----
nsm: 1
streams: 1
pubs:    2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 0 0

---Slave EC num 28----
nsm: 1
streams: 3
pubs:    3
*Send stream 3*
Data_send: 4 1
SDM size(2 data): 18 bytes

---Slave EC num 29----
nsm: 1
streams: 1
pubs:    2
*Save stream 1*

***APP3 RCV MESSAGE 1***
Data_rcv: 4 2
```

BIBLIOGRAPHY

[1] "Ethernet in Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Ethernet 1.1

[2] P. Pedreiras and L. Almeida, "The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems," in *Proceedings International Parallel and Distributed Processing Symposium*. IEEE Comput. Soc, 2003, p. 9. [Online]. Available: http://ieeexplore.ieee.org/document/1213243/ 1.1, 1.3, 2.3

[3] "DFT4FTT Project." [Online]. Available: http://srv.uib.es/dft4ftt/ 1.1

[4] R. Santos, "Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications," Ph.D. dissertation, Universidade Aveiro, 2011. 1.1, 2

[5] A. Ballesteros, S. Derasevic, M. Barranco, and J. Proenza, "First Implementation and Test of Reintegration Mechanisms for Node Replicas in the FT4FTT Architecture," in *Proc. 21st IEEE Int. Conf. on Emerging Tech. and Factory Autom. (ETFA)*, Berlin, 2016. 1.1

[6] A. Ballesteros and J. Proenza, "A description of the FTT-SE protocol," Tech. Rep., 2013. 1.2, 1.3, 2.1, 2.2, 2

[7] "Virtual Distributed Ethernet - VDE Switch in Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Virtual{_}Distributed{_}Ethernet{#}VDE{_}switch 1.3

[8] L. Almeida, P. Pedreiras, and J. A. G. Fonseca, "The FTT-CAN protocol: Why and how," *IEEE Transactions on Industrial Electronics*, vol. 49, no. 6, pp. 1189–1201, dec 2002. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1097741 2

[9] 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*, vol. 1, no. 3, pp. 162–172, 2005. 2

[10] "Linux on Embedded Systems in Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Linux{_}on{_}embedded{_}systems 4.1

[11] "The 2017 Top Programming Languages in IEEE Spectrum." [Online]. Available: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages 4.1

[12] "Make Software." [Online]. Available: https://en.wikipedia.org/wiki/Make{_}(software) 4.1

[13] "CMake Tutorial." [Online]. Available: https://cmake.org/cmake-tutorial/ 4.1

[14] "Wireshark in Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Wireshark 4.1

[15] "Unix Domain Sockets in Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Unix{_}domain{_}socket 4.2.1

[16] "Signal - SIGINT in Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Signal{_}(IPC){#}SIGINT 4.2.2

[17] "Timespec definition in cppreference." [Online]. Available: http://en.cppreference.com/w/c/chrono/timespec 4.4

[18] I. Alvarez, L. Almeida, and J. Proenza, "A first qualitative comparison of the admission control in FTT-SE, HaRTES and AVB," in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*. IEEE, may 2016, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/7496524/ 5.9