Universitat de les Illes Balears

Implementation and Testing of the Node Replication Scheme of the FT4FTT Architecture

MSc Candidate
Alberto Ballesteros

A MSc thesis submitted to *Departament de Ciències Matemàtiques i Informàtica* of the University of Balearic Islands in accordance with the requirements for the degree of
Màster Universitari Enginyeria Informàtica (MINF)

Author
Alberto Ballesteros

MSc Supervisor
Julián Proenza

MSc Supervisor
Manuel Barranco

MINF Director
Antònia Mas Pichaco

21/09/2016

Implementation and Testing of the Node Replication Scheme of the FT4FTT Architecture

Alberto Ballesteros
Tutor: Julián Proenza y Manuel Barranco
Treball de fi de Màster Universitari Enginyeria Informàtica (MINF)
Universitat de les Illes Balears
07122 Palma de Mallorca
ballesteros.alberto@gmail.com

Abstract

Distributed Embedded Control Systems (DECSs) typically have stringent real-time and dependability requirements. Moreover, nowadays they are starting to be deployed in dynamic environments where they must also be flexible to adapt to the unexpected changes in the operation conditions. The FT4FTT project aims at providing an architecture for a complete distributed embedded system which supports applications that are real-time, highly-reliable and adaptive. This project addresses all these requirements for both the network and the nodes of the DECS. In this work we describe the implementation and testing of the fault-tolerance mechanisms devised to provide the nodes with tolerance to permanent and temporary faults, as well as to recover a faulty node when a temporary fault provokes its permanent discoordination with the rest of the system.

Keywords: Distributed Embedded Systems, Control Systems, Adaptive Systems, Dependability, Fault Tolerance, Industrial Communications, Ethernet, FTT, HaRTES, FT4FTT, Node Replication.

1 Introduction

A Distributed Embedded Control Systems (DECS) is a specific-purpose computer system devoted to control or support the operation of an equipment, machinery or plant in which it is spread. These systems typically have stringent real-time and dependability requirements. Moreover, nowadays they are starting to be deployed in dynamic environments and, thus, these DECSs must also be flexible to adapt to the unexpected changes in the operation conditions. An example of a flexible DECS is a system that is able to start new operations and communications, at the expense of less critical ones, in response of new operational requirements.

The FT4FTT (Fault Tolerance for Flexible Time-Triggered Ethernet) project [1] aims at providing an architecture for a complete distributed embedded system which supports distributed control applications that are real-time, highly-reliable and adaptive. This project addresses all these requirements for both the network and the nodes of the DECS.

As concerns the network, the FT4FTT architecture relies on the Flexible Time-Triggered communication paradigm (FTT) [13]. FTT is a time-triggered solution that makes it possible for a set of nodes, to exchange both periodic and aperiodic traffic in a predictable and flexible manner. FTT uses a master/multi-slave scheme, that is, a dedicated node called master is responsible for managing the communication among a set of slaves, which are the regular nodes of the DECS. For this, the master divides the communication in fixed-duration slots called Elementary Cycles (ECs). At the beginning of each EC the master broadcasts the so-called Trigger Message (TM) which contains the EC schedule, that is, the list of messages that the slaves must transmit during that EC.

Note that in FTT the scheduling of the messages is calculated online in a central point, the master. This eases the fulfilment of the slaves' communication requirements as any change on the scheduling can be done locally. In this regard, the master ensures the predictability of the traffic, that is, we know in advance the messages to are going to be transmitted in a given EC and, thus, we can enforce a real-time behaviour. Furthermore, the master also provides flexibility since the slaves can request the modification of the communication requirements at any time.

There are several implementations of FTT. Here we are interested in a switched Ethernet implementation called Hard Real-Time Ethernet Switching (HaRTES) [14]. In HaRTES all the slaves are interconnected by means of a custom Ethernet switch that embeds the master. This configuration yields important benefits from the point of view of the management of the aperiodic traffic, the implementation of fault-tolerance mechanisms and the delay of master messages.

Nevertheless, HaRTES was not developed to provide high reliability in the communications. That is why in the scope of the FT4FTT project we developed a fault-tolerant version of HaRTES called Flexible Time-Triggered Replicated Star

for Ethernet (FTTRS) [12][11]. FTTRS tolerates permanent and temporary hardware faults affecting the switches and the links. On the one hand, permanent faults are tolerated by means of space redundancy. Specifically, as sketched in Fig. 1, slaves are interconnected among them through two custom switches each one embedding an FTT master. Note that, FTTRS switches are also connected between them through several links called interlinks in order to coordinate their operation. On the other hand, temporary hardware faults are tolerated by means of time redundancy, that is, critical messages are transmitted several times.

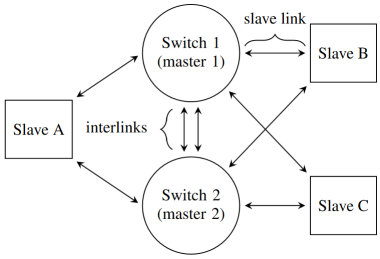


Fig. 1: FTTRS architecture.

As concerns the nodes, the FT4FTT architecture also provides mechanisms to tolerate permanent and temporary hardware faults in the context of control applications [10]. For this, active replication with majority voting is used. The active replication technique consists in replicating each node performing a critical task. For example, Node 3 in Fig. 2 performs the control of a plant and, thus, a fault affecting this node can lead to a failure in said plant. Thus, to make the system more reliable, the control is performed by several replicas.

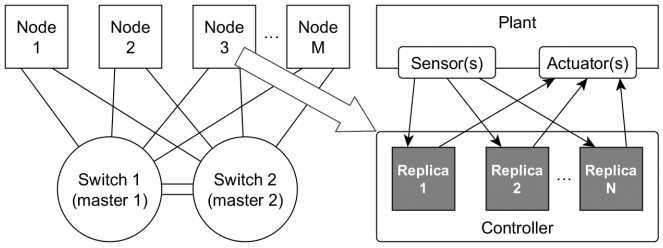


Fig. 2: Active replication.

However, it is not enough to add more nodes. It is also necessary to coordinate the operation of the replicas. Consequently, in the FT4FTT architecture a distributed majority voting algorithm is used to keep replicas coordinated and to compensate the errors that they could generate, as long as there is a majority of non-faulty replicas. For this, software executed by the replicas is partitioned into segments. For each segment each replica produces an output, which is exchanged and voted upon to reach a consensus. This consensus value is then used by the replicas as input for the next segment.

More recently, in [8] the FT4FTT project has addressed the tolerance of temporary faults affecting the replicas. These faults are more likely to occur than permanent ones and, depending on their kind and duration, can lead a replica to a loss of coordination at the communication and/or computation level. If this happens, said replica is permanently disabled and cannot be used anymore. This phenomenon is called *redundancy attrition* and is a big issue as we are not taking full advantage of the investment done for the redundancy. Therefore, in [8] additional fault-diagnosis and reintegration mechanisms were proposed. These mechanisms make it possible to identify temporary faulty replicas and bring them back to a coordinated operation with the non-faulty ones.

In this document we present the prototyping and testing of the node replication mechanisms designed for the FT4FTT architecture. Theses mechanisms include the ones devised to allow the tolerance of permanent and temporary faults affecting the nodes and the ones devised to allow the diagnosis and reintegration of permanently discoordinated replicas.

The rest of this paper is organized as follows. First, Sec. 2 further describes the mechanisms devised to tolerate faults affecting the nodes. Second, in Sec. 3 we explain how all these mechanisms were implemented. Third, Sec. 4 is devoted to give some details about the testbed used for the testing. Fourth, in Sec. 5 we describe the tests carried out and discusses the results obtained. Finally, Sec. 6 summarizes the contribution and points out some future work.

2 Design of Fault-Tolerance Mechanisms for Faults Affecting the Nodes

This section summarizes the most important fault-tolerance mechanisms addressed to tolerate faults affecting the nodes. For this purpose, we first explain how the FT4FTT architecture physically replicates the nodes in the context of control systems. Then, we distinguish the actions carried out by a replica to both perform the control and to achieve the coordination with the rest of the replicas. After that, we describe the mechanism responsible for triggering each of these actions in all the replicas and in a synchronized manner. Finally, we list the additional mechanisms designed to identify and reintegrate a permanently discoordinated replica.

2.1 Node Replication for a Control System

As introduced previously, in the FT4FTT architecture permanent and temporary hardware faults affecting the nodes are firstly tolerated by means of active replication. The active replication technique consists in identifying each of the nodes whose failure could provoke a failure in the whole system, and then adding active nodes that execute the same code in parallel. The main advantage of this approach is that faults can be tolerated transparently with no downtime.

In Fig. 3 we show how the FT4FTT architecture uses this technique in the context of a control system. As depicted in the left-most part of this figure, we distinguish the plant, which represents a physical system that has to be controlled; and the control system, which contains the controller, that is,

a device executing a control application that regulates the behaviour of the plant. The interaction between plant and control system is carried out by means of a sensor and an actuator.

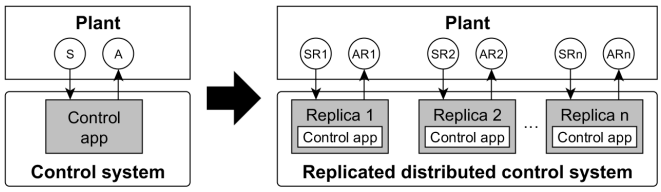


Fig. 3: System architecture.

The failure of the controller can lead to the failure of the plant and, thus, in the right-most part of this figure we show a more reliable version of the system in which the control system is replaced by a replicated distributed control system. This system is composed of several interconnected replicated nodes. Replicas are processing units with identical hardware that execute exactly the same tasks: control the plant and coordinate the replicas. Finally, note that now each replica interacts with the plant through dedicated sensors and actuators.

2.2 Extended Control Application Cycle

Apart from physically replicating the nodes, it is necessary to include on them some logic to ensure that they operate in a coordinated manner. In the FT4FTT architecture this is achieved by means of a distributed majority voting algorithm. Next we explain the additional actions a replica performs to periodically synchronize with the other replicas

Typical control applications can be divided in three phases: *sense*, read the state of the plant by means of sensors; *control*, determine the actuation to be performed at the plant to change its current state to the desired one; and *actuation*, carry out the action to change the state of the plant. The duration of one control cycle is called sampling period (T_s).

In the FT4FTT architecture new phases are introduced to deal with the tasks related to management of the node replication. Specifically, replicas need additional time to exchange and vote on the values of the sensors and on the result of the actuation. That is why a more complex scheme composed of seven phases was proposed:

1. *Sense* (S). Each replica retrieves, from the sensor attached to it, the current state of the plant.
2. *Exchange of sensor values* (ES). Each replica sends to the other replicas, through the fault-tolerant communication channel, its local view of the state of the plant.
3. *Voting on sensor values* (VS). Each replica votes on all the sensor values, that is, the ones received from the other replicas and the one acquired from the sensor attached. The result of this vote is the so-called *consensus sensor value*. It should be noted that the type of voting performed depends on the type of data on which the vote takes place.
4. *Control* (C). Each replica uses the consensus sensor value to determine the actuation to be performed at the plant.
5. *Exchange of actuation values* (EA). Each replica sends to the other replicas, through the fault-tolerant communication channel, the output of the control algorithm.
6. *Voting on actuation values* (VA). Each replica votes on all the actuation values, i.e., the ones received from the other replicas and the one obtained from the control algorithm. The result of this vote is the so-called *consensus actuation value*. Note that in this phase the type of voting performed always assumes that all the actuations are identical.
7. *Actuate* (A). Each replica sends its consensus actuation value to the actuator subsystem. This subsystem consolidates the received values and performs the actuation.

In order for the FT4FTT network to assist in the triggering of the execution of the phases in the replicas, phases are mapped into ECs. The most intuitive mapping would be to devote one EC to each of the phases. However, this is not the only possible mapping. For instance, if the plant needs a very tight control, we can reduce the sampling period by executing several phases in one EC. In contrast, if the control system is working in a very harsh environment in which errors affecting the communication are likely to occur, we can devote several ECs for phases 2 and 5 so that replicas have various chances to exchange their messages.

2.3 Coordinated Triggering of Phases

Once the control application phases have been defined it is necessary to specify how to trigger their execution in all the replicas and at the same time. For this purpose the FT4FTT architecture contains the Coordinated Dispatching of tasks and messages for Node Replication (CD4NR) mechanism, which takes advantage of the TM to also trigger the execution of phases in the nodes.

More precisely, each replica is provided with an *EC Counter* and a *Triggering Table*. The former is a local copy of the so-called *TM sequence number*, which is a numerical value the masters insert into the TM to indicate number of the current EC. The second one is a table specifying the phase that has to be executed in each EC. Every time a replica receives a new TM it updates the value of its EC Counter following Eq. 1, where TM_seqno is the TM sequence number, and T_s is the sampling period of the application measured in ECs. Once the application has determined the value of the EC Counter, it consults the Dispatching Table to execute the corresponding phase.

$$EC\_counter = TM\_seqno \bmod T_s \quad (1)$$

The main advantage of this solution is that the communication subsystem remains unaware of the operation of the application. This is because, no changes are needed in the operation of FTT to support this dispatching scheme.

2.4 Fault-Diagnosis and Reintegration

The fault-tolerance mechanisms previously described make it possible to tolerate permanent, as well as transitory faults that prevent nodes from properly communicating or operating.

In particular these mechanisms require that the non-faulty replicas are coordinated among at both the communication and the application levels. On the one hand, to be coordinated at the communication or EC/phase level means that the replica follows the EC pace dictated by the network and, thus, executes the appropriate phase in the correct instant. On the other hand, to be coordinated at the application level means that the replica has the same correct *operational state* as the non-faulty replicas. The operational state is the information a non-faulty replica needs to produce results for the current and following phases. As explained in [8], the operational state includes information such as the value of sensors, actuators and the *control status*. This last value depends on the type of controller and, since we are assuming a PID, these are the error, the derivative and the integral terms.

Although the already described mechanisms allow tolerating faults, it may happen that a temporary fault, depending on its kind or duration, makes a replica to lose coordination with respect to the non-faulty replicas. In [8] authors thoroughly classified the temporary faults that may provoke this discoordination. Basically, a temporary fault affecting the communication capabilities of a replica may lead said replica to lose coordination if the fault lasts too much. This can happen, for instance, when the replica does not receive a TM or enough messages for voting. As regards a temporary fault affecting the capacity of a replica for correctly operating, note that it normally discoordinates the replica. This is because an incorrect operation, for instance, a wrong read operation, usually prevents the replica from producing correct results.

In most cases a temporary fault may lead a replica to be lost from the on and, thus, to behave as if it was permanently faulty unless it is reinitialized. This problem is a redundancy attrition as the discoordinated replica cannot participate in tolerating faults any longer. This problem is a big issue since temporary faults are more likely to happen than permanent ones.

In order to overcome this vulnerability, the FT4FTT architecture includes additional mechanisms to both diagnose replicas that are lost and reintegrate them. Note that this has to be done as soon as possible so the fault tolerance capabilities of the system are restored, and additional faults can be tolerated. These mechanisms are sketched next, for further details about them please refer to [8]:

1. *TM resync*. When a temporary fault prevents a replica from receiving the TM during one or more ECs, the value of its EC Counter gets desynchronized with respect to the other replicas. In order to reintegrate at the communication level the replica updates its EC Counter with the TM sequence number piggybacked at the TM.
2. *Voting Reint. Point*. Replicas use this mechanism to reintegrate at the application level, that is, to acquire the correct operational status of the non-faulty replicas. On the one hand, the extended control cycle is modified so that replicas use phases ES, VS and EA, VA to exchange and vote on, not only the values of the sensors and actuators respectively, but also on the control status. Note that the control status is a fundamental part of the operational status, as a replica needs the control status to correctly execute the control algorithm. On the other hand, a faulty replica reintegrates by updating any incorrect value of its operational state by the consensus one obtained when voting.
3. *Communication Error Counter (CEC)*. This mechanism is devoted to diagnose and reintegrate any replica from temporary faults that prevent it from correctly communicating, as long as the replica itself is not reinitialized. On the one hand, the replica increases or decreases the CEC after phases VS and VA, depending on its ability to communicate. For the replica to know this, the masters convey in every TM information concerning what messages the replica successfully transmitted and received during the last exchange phase. More details of the mechanism used for populating the communication status can be found in [9]. On the other hand, when the CEC exceeds a given threshold, the replica resets itself and then reintegrates using the two previously-explained reintegration mechanisms.
4. *Discrepancy Error Counter (DEC)*. This mechanisms is analogous to the CEC, but it is intended to diagnose and reintegrate from temporary node internal faults that prevent the replica from correctly operating, as long as the replica itself is not reinitialized. For instance, a temporary fault that corrupts a memory-stored value may lead the replica to behave incorrectly until the replica is resumed. The replica increases or decreases the DEC after each voting, basically depending on whether or not its local values for the operational state deviate from the consensus ones that result from the voting.
5. *You Are Alive (YAA) watchdog*. This is an external device connected to each one of the replicas that is able to detect a crash of the replica to whom it is connected and reset it if necessary. To avoid being considered as permanently faulty and, thus, being reset, the replica has to periodically forward the so-called YAA message that is piggybacked on every TM. This message is specifically generated by the masters and cannot be forged.

3 Implementation of the Node Replication Scheme

This section describes the implementation of all the fault-tolerance mechanisms previously introduced in Sec. 2. Specifically, we first describe the initial prototype built in the scope of the FT4FTT project. Then we explain how this prototype was modified to extend the control application cycle and

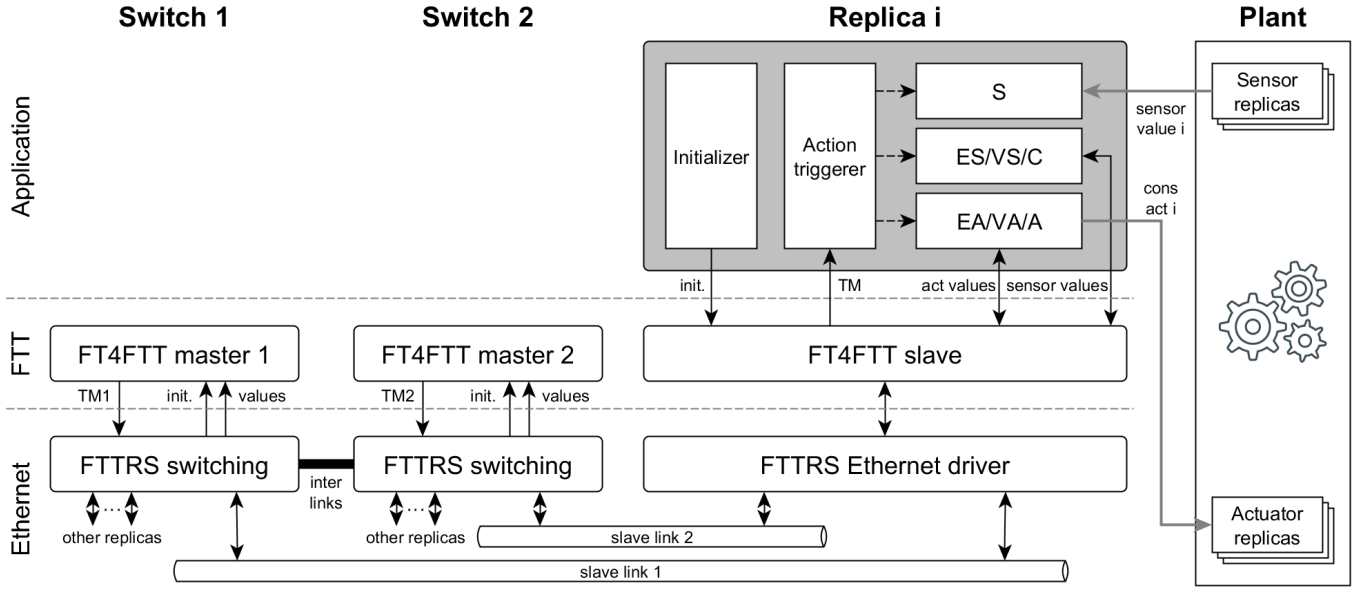


Fig. 4: Software diagram of the node replication implementation.

to include the coordinate triggering of the phases. Finally, we describe the implementation of the fault-diagnosis and reintegration mechanisms.

3.1 Initial prototype

The starting point for the implementation of the node replication scheme was a prototype containing the FTTRS network, as well as a basic node replication consisting in one round for the exchange voting on the intermediate results. The functionalities of this prototype can be seen in [2]. As shown in this video, it included a simple example program simulating a control algorithm and a basic triggering mechanism to activate each of the phases of the application.

This implementation was completely done using the C language. This language is well suited for embedded applications as the code generated is quiet efficient. In this sense, note that HaRTES was already developed in this language.

As concerns the hardware, the prototype is composed of two FT4FTT switches and three node replicas (see Fig. 6b). On the one hand, each switch is a regular Intel i7-4770 multicore PC with 8 GB of RAM provided with several Intel i350-T4 Ethernet interfaces. The main benefit of this configuration is that the Asus Z87-WS motherboard contains a PCI-Express switch which makes it possible for the applications to receive and transmit Ethernet frames in parallel with little interferences. Moreover, the selected Ethernet adapters allow the possibility of managing multiple low-level operation parameters so that we can configure them to decrease the delay in the communications. On the other hand, each slave is built using a commercial computer devised fer embedded devices, the Jetway JBC373F38-525-B. This is a small barebone containing an Intel Atom D525 processor with 2 GB of RAM and four Ethernet interfaces.

Finally, as concerns the operating system, switches and slaves run Ubuntu Linux 12.04 with Xenomai, a supplement that enables its use in systems with real-time requirements.

3.2 Extended Control Application Cycle and Coordinated Triggering of its Phases

This section discusses the additions carried out on top of the initial implementation to include the extended control application cycle and the CD4NR mechanism.

More precisely, the implementation consisted in the addition of specific features in various components and in different layers of the architecture. As sketched in Fig. 4, we can distinguish between the plant and the replicated distributed control system. This last is composed of a set of replicas (here only one is represented) and the switches. Additionally, the control system can be divided into three layers. First, the Ethernet layer is responsible for transmitting the Ethernet frames among the replicas and the switches. Second, the FTT layer contains the modules providing the FTT services, namely one FT4FTT master inside each switch and one FT4FTT slave inside each replica. Finally, the application layer is only implemented in the replicas and performs the control of the plant, as well as the management of their redundancy. Next we describe the operation of all of these components.

In the plant, every sensor replica is directly connected to the application of one of the replicas and periodically informs about the internal state of the plant. Similarly, the application of every replica is directly connected to the actuator subsystem. In this implementation this subsystem includes a consolidator that gathers the actuation commands of all the replicas, unifies them and applies the resultant actuation to the plant.

As concerns the control system, its operation starts with the switches broadcasting the TMs, which act as notifications for

the replicas to know that the communication channel is available. At this point, replicas request to the masters the creation of all the communication resources necessary to execute their applications. More specifically, the *Initializers* of the replicas inform the masters about the size and periodicity of the messages needed to be transmitted. If there is enough bandwidth available, masters change the schedule so that the TM triggers the transmission of these messages. It is noteworthy that, for the replicas to perform their actions in synchrony, the messages associated with these actions have to be scheduled in the same EC. However, there is no mechanism in FTT to specify the precise ECs in which messages have to be triggered. To solve this problem the Initializers send the initialization messages in an order and in instants of time that, then, lead the masters to schedule the messages in the desired ECs.

After the initialization the control system starts its regular operation. Specifically, every time a replica receives a TM the *Action triggerer* is notified so that the value of its EC counter can be updated. It should be noted that in this implementation we slightly modify Eq. 1 to also take into account the number of ECs used in the initialization. Specifically, as shown in Eq. 2, we introduce APP_start, which is the EC in which the initialization finishes and, thus, the application starts.

$$EC\_counter = (TM\_seqno - APP\_start) \bmod T_s \quad (2)$$

Next, we describe the actions carried out in every of the three tasks in which we have divided the extended control cycle (see Section 2.2) of the application.

First, the S task gathers from its associated sensor replica in the plant the last sensor value registered.

Second, the ES/VS/C task transmits the sensor value to the other replicas, waits for the reception of the sensor values from the other replicas, votes on all these values and executes the control algorithm with the result of the voting, in order to calculate the actuation value. In this implementation we assume that the state of the plant can be represented as a floating point value like, for instance, the temperature in a room. These values are not expected to be identical even if sensors are operating correctly. Therefore, replicas perform a type of voting that consists in removing the outliers and then calculating the average of the resulting values.

Finally, the EA/VA/A task transmits the result of the control algorithm, that is, the actuation value, waits for the reception of the actuation values from the other replicas, votes on all of them and sends the result of this vote to the actuator subsystem in the plant. As explained in Sec. 2.2, the type of voting performed on the actuations considers that their values are identical and, thus, the task implements a majority voting.

Note that, thanks to the clever initialization previously described in this section, masters generate the TMs according to the control scheme previously presented. That is, during the sense task no transmission is scheduled, but during the ES/VS/C and EA/VA/A tasks, masters schedule the transmission of all the sensor and actuation values respectively.

3.3 Fault Diagnosis and Reintegration

This section describes the extensions carried out in the node replication implementation presented in the previous section to include the fault-diagnosis and reintegration mechanisms explained in Sec. 2.4. It is noteworthy that, as shown in Fig. 5, to make the application robust and fast, in this new implementation the 7 phases of the extended control cycle are implemented by means of 5 tasks. Next we discuss the implementation of every fault-diagnosis and reintegration mechanism.

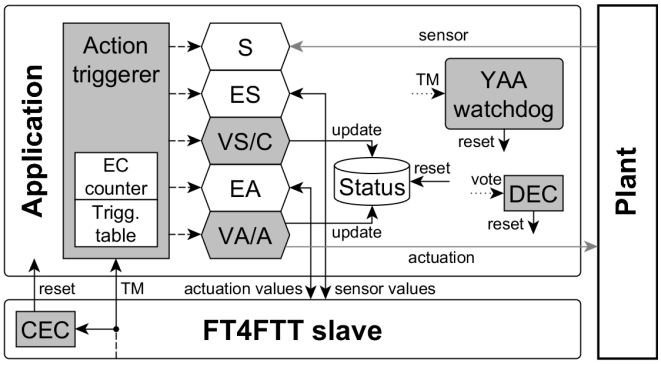


Fig. 5: Software diagram of the fault-diagnosis and reintegration implementation.

First, the *TM resync* mechanism did not need relevant modifications in the code as the *Action triggerer* already triggers the execution of each of the tasks upon the reception of a TM.

Second, to add the *Voting Reint. Point* mechanism we reprogrammed tasks VS/C and VA/A to not only respectively vote on the sensor and actuation values, but also to vote on the values that constitute the control status. The control status is labelled *Status* in Fig. 5. Note that the control status is also voted in the VA phase to give more chances to reintegrate during a given control cycle. Additionally, we also include the setpoint in the voting.

Third, as concerns the *Communication Error Counter* (CEC) mechanism, we implemented the counter itself at the FTT level so that it can update its value upon the reception of a new TM. As introduced in Sec. 2.4, every TM conveys the list of application messages correctly transmitted and received by the replicas. Thus, it serves to diagnose communication errors. If any communication error is detected, the value of the counter is increased, otherwise it is decreased unless it is zero. The value of this counter can be read by the application, which resets itself if it exceeds a certain threshold. Note that, for simplicity we perform a soft reset, which consists in reinitializing the operational state information rather than reinitializing the hardware components of the replica.

Fourth, the *Discrepancy Error Counter* (DEC) mechanism is similar to the CEC but for errors affecting the application. It is implemented at the application level and increases its value whenever the consensus value resulting from a voting differs from what the replica proposed. Otherwise it decreases the counter if it is not zero already. As for the CEC, the application reinitializes when the DEC exceeds a given threshold.

Finally, we add a simplified version of the *You Are Alive (YAA) watchdog*. Specifically, we implemented it as a software module that resets the application when several consecutive TMs are not received; that is, when the replica seems to be completely lost and cannot reset itself.

4 Testbed Setup

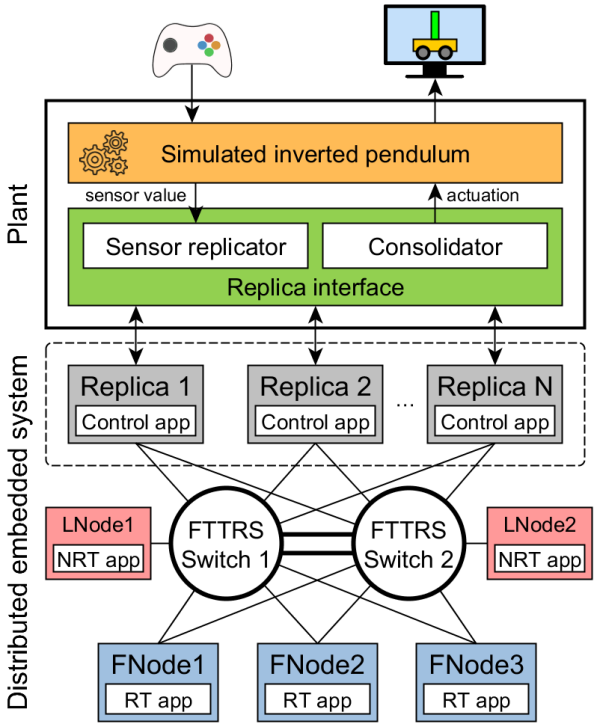
Up to this point of the document we have been assuming that the FT4FTT architecture only executes a replicated real-time control application. However, as stated in the introduction, this architecture also provides flexibility in the communications. In this regard, it is possible for the slaves to change the communication requirements online, as well as to connect legacy nodes that exchange non-real-time traffic. Consequently, we developed a complete new experimental setup that allowed us to test all the fault-tolerance mechanisms described in this document while running several applications with different real-time, reliability and adaptivity requirements. The complete design and implementation of this setup is sketched in Figs. 6a and 6b respectively.

Physically, this setup is composed of two FT4FTT switches, three node replicas and two legacy nodes. Each switch executes an F4FTT master, whereas each replica executes an FT4FTT slave implementing a typical control application. Finally, legacy nodes execute a multimedia application. Next we describe in more depth the applications executed in each node and the services we test in each case.

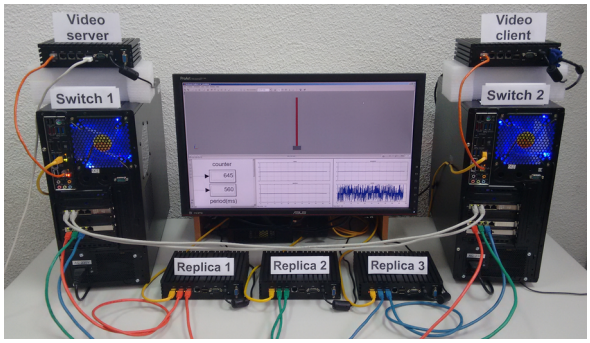
First, to test the support of replicated real-time traffic, we developed a control application that regulates the behaviour of a plant, a simulated inverted pendulum [4]. More specifically, as shown in Fig. 6a, we built a system using the *hardware-in-the-loop* [3] technique, where the plant is simulated in software while the control system is implemented as a realistic combination of hardware and software. The control application is executed in parallel by three node replicas, the gray-filled nodes in the figure. The implementation of the simulated plant is discussed below, in Sec. 4.1.

Second, to demonstrate the support of flexibility in real-time communications, blue-filled FT4FTT nodes in Fig. 6a perform an exchange of periodic information. Specifically, FNode1 creates and manages a virtual communication channel between FNode2 and FNode3. FNode2 periodically transmits the value of a counter to FNode3. The periodicity of this transmission can be modified online by FNode1. Finally, note that, although these three nodes are represented in the figure as independent physical nodes, their operation is carried out by the same physical nodes performing the control of the simulated inverted pendulum, that is, the gray-filled nodes.

Third, two legacy nodes, depicted as red nodes in Fig. 6a, exchange non-real-time information. Specifically, LNode1 acts as a video server and transmits a video stream to LNode2. This last node acts as a video client and outputs the video stream through a screen.



(a) Design of the experimental setup.



(b) Implementation of the experimental setup.

Fig. 6: Experimental setup.

4.1 Simulated plant

As indicated above, the experimental setup includes a simulated plant that makes it possible to assess the correct operation of the replicated control application in the presence of faults. The construction of this simulator is quite complex and, thus, was carried out within the final-year project of an informatics degree student. Specifically, the author of the present work was responsible for supervising said project and, more specifically, for designing the architecture of the plant, which includes the simulated inverted pendulum and its interface with the node replicas, and for performing various final modifications to adapt the solution to the existing prototype. On the other side, the student was responsible for finding and adapting an available graphical simulated inverted pendulum and implementing the interface between this simulator and the node replicas. Next we describe more deeply these tasks.

The initial design for the plant consisted in a document containing the main components of the plant and their interactions. Specifically, as sketched in Fig. 6a, the *Simulated inverted pendulum*, module has to simulate the physics of an inverted pendulum, as well as to graphically represent it, all in real time. Additionally, the *Replica interface* module has to send the status information to the replicas and the actuation to the inverted pendulum. On the one hand, this module has to obtain the angle of the pendulum and the position of the cart, replicate them and send them to the replicas. Note that here we can simulate sensor value discrepancies by introducing some errors into the readings. On the other hand, after the execution of the control algorithm, the Replica interface has to collect the actuations coming from the replicas, consolidate them and perform a single actuation into de plant.

The student made a thorough search to find an inverted pendulum implementation that could be easily adapted for our purposes. Specifically, we selected a Simulink implementation which can be adapted by tuning its operational parameters and by modifying and adding new blocks, that is, a Simulink element that carries out a predefined action. As shown in Fig. 6a, the Simulink model communicates with the Replica interface, and the latter one with the replicas. The communication between the Simulink model and the Replica interface was implemented by means of a network block, that is, a Simulink element that makes it possible to transmit and receive TCP/IP packets. Regarding the details of the implementation, the Replica interface is an independent program programmed in C code using network sockets for the communication with both, the Simulink model and the replicas. Note that the idea of communicating all these pieces of software through network connections makes the testing and deployment very flexible, as said pieces can be placed in the same machine or in different ones. Finally, the student also developed the control application running in the replicas. This application is composed of two PID controllers [5] that operate in parallel. One of them regulates the angle and the other one the position of the inverted pendulum.

At this point we were able to execute the simulated inverted pendulum and regulate it thanks to the control application. However, this was not performed in real time. That is why some modifications were carried out to accommodate the implementation to the prototype. First, we transformed the Simulink model so it ran in real time. This was done by adding and replacing some blocks and by adjusting some operation parameters. Second, we changed the communication protocol used by the Replica interface. Specifically, now it uses User Datagram Protocol (UDP) to communicate with the inverted pendulum model and the replicas. This communication protocol is widely used in real-time environments as its low overhead makes it very fast. Finally, the consolidator inside the Replica interface was enhanced to be more fault-tolerant. More precisely, now the consolidator is able to tolerate the faults we typically inject, for instance, omissions or untimely messages.

Finally, we extended the Simulink model to accept commands from a gamepad (see top of Fig. 6a). This device makes it possible for a user to apply some force on the pendulum and, thus, simulate a disturbance. This, in turn, allows us to check the responsiveness of the system.

5 Testing

Here we present the most relevant tests we have carried out to check that the implementation of the fault tolerance mechanisms was done properly and that they integrate correctly with the rest of the already implemented fault tolerance mechanisms. Moreover, these tests also serve to check that the design of the mechanisms was correct. Finally, we also obtain some preliminary measurements of the time needed for a replica to reintegrate.

5.1 Testing Tolerance to Permanent Faults

With these tests we assess the tolerance of the system to permanent faults in the channel. For this, we have carried out two test campaigns in which we simulate the crash of the two switches and the failure of the links of the control system.

5.1.1 Tests Conducted

In test 1 we assess the tolerance of the system to switch crashes. Specifically, we carried out two experiments each one involving the crash of one of the switches.

In test 2 we assess the tolerance of FTTRS and the node replication to permanent faults affecting the links of the control system. More precisely, we provoke all the possible combinations of disconnections in the replicas' links and in the interlinks. Since the control system contains 8 links (2 links for each of the three replicas and 2 interlinks) and each link can be non-faulty or faulty, there are $2^8 = 256$ different fault scenarios. However, for the system to work correctly, some assumptions were done in the design: at least one interlink must be online and at least a majority of the replicas (2 in this case) must have a non-faulty link with one of the switches. Consequently, we test 162 scenarios of these 256 possible scenarios. Note that, in 81 of the tested scenarios both links of one of the replicas were faulty, that is, these scenarios provoked behaviours equivalent to the failure of said replica.

5.1.2 Results

In both test campaigns the system was able to continue its operation normally. Moreover, no disturbances where noticed in the control of the plant when provoking the switch crashes and the link disconnections.

5.2 Testing Tolerance to Temporary Faults

With these tests we assess the fault-diagnosis and reintegration mechanisms when different types of temporary faults af-

fecting the links or the replicas do occur. Moreover, we get some preliminary measurements of the reintegration time, that is, the time elapsed since the fault stops provoking errors until the operational state of the affected replica is consistent with the ones of the other replicas.

It is important to mention that in each of the tests we provoke errors in only one of the replicas. Moreover, for each test we carry out several experiments in order to analyze the effects of the provoked errors in each of the five phases of the application (see Sec. 3.3). This represents a first step towards a complete characterization of the reintegration time.

Next we describe each of the tests we conducted. To assist in the explanation Table 1 summarizes the error injection, that is, the type of error provoked and the instant in which it is injected. We finish this section by presenting the reintegration times registered for each test and the conclusions we have extracted from them.

S	ES	VS/C	EA	VA/A	S	ES	VS/C	EA	VA/A
TM									
	TM					DM			
		TM							
			TM					DM	
				TM					
Mem							Mem		Mem
	Mem						Mem		
		Mem							
			Mem						Mem
				Mem			Mem		Mem

Table 1: Error injection for tests 1, 2, 3 and 4.

5.2.1 Tests Conducted

In test 1 we simulate temporary link faults that prevent the replica from receiving the TM of some of the phases (see top-left part of Table 1). This causes the replica to not execute the associated phases. In order to reintegrate, the replica uses the *TM resync* mechanism, so that the subsequent phases can be executed normally. In some cases, not executing a given phase results in the replica not properly voting, which leads to a loss of consistency with the other replicas. In these cases the replica also uses the *Voting Reint. Point.* mechanism.

In test 2 we simulate temporary link faults that prevent the replica from transmitting and/or receiving the application messages, herein called Data Messages (DMs). This test is composed of three sub-tests in which we analyze the behaviour of the replica when it is not able to receive them, transmit them and receive nor transmit them, during one phase. It is noteworthy that, as shown in the top-right part of Table 1, we only inject these error during the ES and EA phases, that is, when replicas exchange messages. In any other phase the fault does not provoke errors. Communication problems involving DMs result in the replica voting with a different subset of messages, which leads to an inconsistent operational state. When so, the replica uses the *Voting Reint. Point.* mechanism to reintegrate.

In test 3 we simulate temporary node faults that corrupt the operational state of the replica. Specifically, as shown in the bottom-left part of Table 1, in each phase we inject an error that modifies the values used and generated by the replica. As a result, said replica uses the *Voting Reint. Point.* mechanism to achieve consistency again.

In test 4 we simulate a replica crash that prevents said replica from correctly voting until it is restarted by the *DEC* mechanism. For this, as shown in the bottom-right part of Table 1, we corrupt the voting values during several consecutive voting phases. Every time the replica is not able to correctly vote the DEC is increased and, once reached a threshold, the replica resets itself. After this reset the replica uses the *TM resync* and the *Voting Reint. Point.* mechanisms to reintegrate in the time and in the value domain, respectively.

In test 5 we simulate a replica crash that prevents it from executing any action until it is restarted by the *YAA watchdog*. For this we force the replica to not receive several consecutive TMs. This error injection is similar to the one carried out in test 1 (see the top-left part of Table 1) but with the loss affecting several phases. As in test 4, once the watchdog resets the replica, the *TM resync* and the *Voting Reint. Point.* mechanisms make it possible to achieve consistency again in the time and in the value domain, respectively.

5.2.2 Results

In each of the tests executed the faulty replica was able to correctly reintegrate in a number of ECs shown in Table 2. Each row corresponds to one of the tests and each of the first five columns corresponds to one of the experiments conducted for each of these tests. Specifically, each of these columns refers to the phase in which the fault finishes. Finally, the last two columns show, for each test, the maximum and average of the reintegration time.

Note that the size of the EC for this test was 7 ms. However, since these results are expressed in ECs they do not depend on the size of the EC, as long as there is enough time in each EC to carry out the required actions. Next we present the main conclusions we extracted from the tests and Table 2.

	Last phase affected by the fault					Statistics	
	S	ES	VS/C	EA	VA/A	max	avg
1	2	3	2	0	0	3	1
2	-	3	-	0	-	3	1.5
3	2	3	2	0	0	3	1.4
4	-	-	2	-	3	3	2.5
5	2	3	2	4	3	4	2.8

Table 2: Time to reintegrate (in ECs) for every test and exp.

As can be concluded, the time needed to reintegrate is the time until the next successful voting. Specifically, according to the Voting Reint. Point mechanism, this happens after receiving the DMs from the other replicas and voting on them. In this sense, as can be seen in Table 2, the values for each column are almost identical. The only difference occurs when injecting errors after performing the control

More precisely, a temporary communication fault occurring after the VS/C phase does not make the replica inconsistent. This is because the actuation is already calculated and it must be identical in all the replicas. In contrast, when a restart is needed to stop the error, the initialization of the operational state forces the replica to wait for the next reintegration point, which occurs in the next application cycle.

6 Conclusions

We presented the implementation and testing of the node replication scheme designed in the scope of the FT4FTT Architecture. The implementation covered all those mechanisms devised to add redundancy to the nodes performing critical control tasks, as well as to diagnose permanently discoordinated replicas and reintegrate them. For this, we extended the previous FT4FTT prototype by adding new functionalities mostly at the application level, but also at the communication level. One relevant contribution of this work is the development of the experimental setup, using the hardware-in-the-loop technique, used to test all these mechanisms. This setup allowed us to test the behaviour of the system when nodes have different real-time, dependability and flexibility communication requirements.

The testing performed validated the design of the replication scheme and verified its implementation and integration with the rest of the fault tolerance mechanisms. Additionally, we obtained some preliminary results of the time (measured in ECs) needed to reintegrate a replica affected by temporary faults.

As a future work, we propose some extensions that could be carried out to improve some specific aspects of the implementation. First, it would be interesting to implement the YAA watchdog in hardware. With this the watchdog would not depend on the replica it is attached to and, thus, it would not suffer any fault affecting said replica. Second, incorporating the implementation of the hard reset would also be an interesting improvement. Specifically, this reset would be able to completely shutdown and restart the replica. This has two benefits. On the one hand, it would allow to recover a replica from a fault affecting the operating system or the hardware. On the other hand, it would allow to improve the characterization of the reintegration time as we could also consider the time needed to start the application and the time needed to enter into the communication.

As a final remark, it is noteworthy that the construction of the experimental platform was a hard work. This is because it is quite difficult to achieve a real-time behaviour with the current software implementation. Thus, to be able to control systems with more demanding real-time requirements it would be necessary to improve the responsiveness of the control system. This would require to analyse and try to enhance the implementation of the network stack and the FTT code.

Appendix A Publicacions of this work

The results of this work have been described in two papers which were presented and published in the proceeding of peer-reviewed international conferences.

First, [7] was presented in the 12th IEEE World Conference on Factory Communication Systems (WFCS 2016) and described the implementation and testing of the basic replication scheme, as wells as the construction of the experimental setup. This paper received the Best Work-in-Progress Award.

Second, [6] was presented in the 21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016) and described the implementation and testing of the fault-diagnosis and reintegration mechansism.

Appendix B Other publications

During the development of this work the author was involved in several final-year projects as a co-supervisor. These projects reflect part of the work described in this dissertation:

- Maties Melià. *Implementació i validació de mecanismes per a l'intercanvi consistent d'informació entre nodes d'un sistema encastat distribuït basat en HaRTES*. Master thesis for the *Màster Universitari en Tecnologies de la Informació*. Supervised by J. Proenza and A. Ballesteros.
- Andreu Adrover. *Infraestructura d'injecció de fallades per a Ethernet amb funcions específiques per a FTT*. Final-year project for the *Enginyeria Tècnica de Telecomunicacions*. Supervised by A. Ballesteros and J. Proenza.
- Adel Zendouh. *Dependable Distributed infrastructures for monitoring and automation in complex emerging applications*. Bachelor thesis in Université Constantine 2, Algeria. Supervised by A. Ballesteros and M. Barranco.

Additionally, the final-year project related to the construction of the simulated plant (see Sec. 4.1) is not listed here as it still is an ongoing work.

Acknowledgements

This work was supported by projects DPI2011-22992 and TEC2015-70313-R funded by the Spanish Ministerio de Economia y Competitividad (MINECO) and by the Fondo Europeo de Desarrollo Regional (FEDER).

References

[1] FT4FTT project. http://srv.uib.es/ft4ftt/.
[2] FT4FTT prototype demo. http://srv.uib.es/ft4ftt/.
[3] Hardware-in-the-loop. wikipedia.org/wiki/Hardware-in-the-loop_simulation.
[4] Inverted pendulum. wikipedia.org/wiki/Inverted_pendulum.

[5] PID controller. wikipedia.org/wiki/PID_controller.

[6] Alberto Ballesteros, Sinisa Derasevic, Manuel Barranco, and Julián 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.

[7] Alberto Ballesteros, Sinisa Derasevic, David Gessner, Francisca Font, Inés Álvarez, Manuel Barranco, and Julián Proenza. First Implementation and Test of a Node Replication Scheme on top of the Flexible Time-Triggered Replicated Star for Ethernet. In *Proc. 12th IEEE World Conf. on Factory Comm. Systems (WFCS)*, Aveiro, 2016.

[8] Sinisa Derasevic, Manuel Barranco, and Julián Proenza. Designing fault-diagnosis and reintegration to prevent node redundancy attrition in highly reliable control systems based on FTT-Ethernet. In *Proc. 12th IEEE World Conf. on Factory Comm. Systems (WFCS)*, Aveiro, 2016.

[9] Sinisa Derasevic, Maties Melia, Alberto Ballesteros, Manuel Barranco, and Julián Proenza. First experimental evaluation of the consistent replicated voting in the hard real-time ethernet switching architecture. In *Proc. 20th IEEE Int. Conf. on Emerging Tech. and Factory Autom. (ETFA)*, Luxemburg, 2015.

[10] Sinisa Derasevic, Julián Proenza, and Manuel Barranco. Using FTT-ethernet for the coordinated dispatching of tasks and messages for node replication. In *Proc. 19th IEEE Int. Conf. on Emerging Tech. and Factory Autom. (ETFA)*, Barcelona, 2014.

[11] David Gessner, Julián Proenza, and Manuel Barranco. A Proposal for Master Replica Control in the Flexible Time-Triggered Replicated Star for Ethernet. In *Proc. 10th IEEE Int. Workshop on Factory Comm. Systems (WFCS)*, Toulouse, 2014.

[12] David Gessner, Julián Proenza, Manuel Barranco, and Luis Almeida. Towards a Flexible Time-Triggered Replicated Star for Ethernet. In *Proc. 18th IEEE Int. Conf. on Emerging Tech. and Factory Autom. (ETFA)*, Cagliary, 2013.

[13] Paulo Pedreiras and Luis Almeida. The Flexible Time-Triggered (FTT) Paradigm: An Approach to QoS Management in Distributed Real-Time Systems. *Proc. Int. Parallel and Distributed Processing Symp.*, 2003.

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