An OMNET++ model to asses node fault-tolerance mechanisms for FTT-Ethernet DESs

Sinisa Derasevic, Manuel Barranco, Julián Proenza
DMI, Universitat de les Illes Balears, Spain
sinishadj@gmail.com, manuel.barranco@uib.es, julian.proenza@uib.es

***Abstract*—Distributed embedded systems (DESs) that operate in dynamic environments require emerging flexibility and adaptivity communication requirements. When those DESs are deployed for critical applications, they must also employ appropriate fault-tolerance (FT) mechanisms to attain a high level of reliability. The FTT-Ethernet communication protocol supports the flexibility needed in dynamic environments, but does not provide adequate fault tolerance. In order to overcome this limitation the ongoing FT4FTT project proposes a communication architecture that includes fault-tolerance capabilities at different levels of DESs relying on FTT-Ethernet. In particular, it provides communication and execution mechanisms to tolerate node failures by means of active node replication with majority voting. This paper builds upon a previous OMNET++ model of an FTT-Ethernet-based DES in order to add, simulate and assess those mechanisms. Specifically, it models the communication mechanisms envisaged to enforce replica determinism in the voting procedure, as well as to trigger and coordinate the tasks executed in the replicas.**

I. Introduction

The Flexible Time-Triggered (FTT) Ethernet [1] communication protocol has been designed to support the changing communication and computation requirements of DES operating in dynamic environments. Nevertheless, it lacks appropriate fault-tolerance mechanisms to attain the level of reliability required by critical DESs. To overcome the reliability limitations of FTT-Ethernet, the ongoing FT4FTT project aims at providing an holistic FTT-Ethernet-based DES architecture that includes fault-tolerance mechanisms at different levels.

Particularly, FT4FTT provides tolerance to node failures by means of active node replication [2] with majority voting. Specifically, it uses multiple identical replicas (same hardware and software) of the nodes executing critical functions. In [3] we proposed a set of communication mechanisms and a voting algorithm for replicas to consistently vote in the presence of permanent and transient node and channel faults. Moreover, in [4] we explain how to trigger and dispatch the tasks and the messages of a distributed control application that relies on the FT4FTT replication and voting strategy.

In order to support the design of the FT4FTT architecture, [5] proposes an OMNET++ model for simulating the behavior of a DES relying on it. OMNET++ is an object-oriented modular discrete event network simulation framework written in C++ programming language [6]. An open-source collection of OMNET++ modules for wired and wireless network protocols called INET Framework [7] is also used by [5].

The model in [5] includes the main features of FTT-Ethernet, and allowed us to closely and quickly approach a real DES based on this technology. Specifically, although not being complete, the model has already served us to inject faults and to test and refine some FT mechanisms proposed in FT4FTT.

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

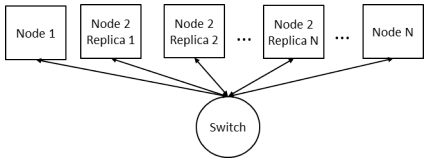

Fig. 1. System Architecture

The present paper builds upon [5] to add, simulate and test the voting and dispatching mechanisms we proposed in [3] and [4]. The results obtained with this new model allowed us to qualitatively check the correctness of those mechanisms.

Section II sketches the architecture, fault model and fault tolerance mechanisms of FT4FTT. Sections III and IV respectively describe in more detail the fault-tolerance and the coordination/dispatching mechanisms we have modeled in OMNET++. Section V describes the OMNET++ model itself with special focus on those mechanisms. Section VI summarizes our results, whereas section VII points out future work and gives some concluding remarks.

II. System architecture and organization

As seen from Figure 1, the architecture of the system we conceive in FT4FTT consists of multiple nodes, some of which are replicas of the same one. The nodes are interconnected in a star topology by a means of a full-duplex Ethernet switch.

The underlying communication protocol is FTT-Ethernet, which supports real time response over standard Ethernet. FTT is a maser/multi-slave paradigm in which a special node, called *FTT-Master*, controls the communication of multiple slaves. This paper considers a particular implementation of FTT where the Master node is integrated within the switch. This switch is called *Hard Real Time Ethernet Switch* (HaRTES) [8].

The communication is divided into fixed-size rounds called *Elementary Cycles* (ECs). The FTT-Master starts each EC by broadcasting a special control message, called *Trigger Message* (TM), that serves two purposes. First, it synchronizes the network by being broadcast in accurate points in time with low jitter. Second, it conveys the schedule for that EC. Each EC is further divided into two windows, synchronous and asynchronous, that handle the transmission of periodic and aperiodic messages respectively.

The fault model of FT4FTT includes permanent and transient node, link and switch faults. Nodes have byzantine failure semantics, i.e. they fail in an arbitrary manner, while the switch has crash failure semantics, i.e. it fails only by crashing.

We tolerate both transient and permanent switch faults by using two actively replicated switches connected by two interlinks. Furthermore, each switch is enforced to exhibit crash failure semantics by duplicating and comparing its internal circuitry. More details of switch replication are found in [9].

Transient link faults are tolerated by using both temporal and spacial redundancy, i.e. critical messages are sent multiple

times through multiple different paths/links. In particular, the work in [10] provides tolerance to transient faults affecting the TMs by employing the aforementioned redundancy.

Permanent link faults are tolerated by spacial redundancy only. Since each node is connected to the switch by means of a dedicated link, when the switch is replicated these links become replicated as well.

Node faults are tolerated by actively replicating them. More specifically, each critical node is replicated as a set of identical hardware units executing the same software. To mask the values produced by faulty node replicas, we employ distributed majority voting. In particular, before using some value (input/output), each replica exchanges it with other replicas. Once the values are exchanged, each replica locally votes on all the obtained values. We shall use the terminology from NVP [11] and call the values that replicas vote on *cross-check vectors* or *cc-vectors* for short.

In [4] we proposed a dispatching strategy, based on FTT-Ethernet, to coordinate the execution of the tasks performed by the replicas and the transmission of the messages they need to exchange. This strategy is explained in details in section IV.

When replication is used, one of the main concerns is how to ensure that non-faulty replicas behave the same. This problem is know as *replica determinism* [2] and has to be solved. Since we use active replication, i.e. same hardware and software, we can assume that non-faulty replicas will vote the same provided with the same input information. Therefore, we have proposed a consistent voting protocol for tackling the problem of providing same voting input to all correct replicas [3] while tolerating node and channel faults.

To prevent unnecessary attrition of node redundancy due to node transient faults, we proposed the mechanisms in [12]. These mechanisms are orthogonal to the ones we describe and model in this work. They are used to recover the nodes being affected by transient faults, and also to disconnect the nodes that are permanently faulty and beyond recovery.

The previous work [5] models the mechanism proposed in [10] to tolerate transient faults affecting the TM. As already said, in this paper we extend this previous work by modeling the mechanisms proposed in [3] and [4]. Next two sections give more details about those new modeled mechanisms.

III. Node FT mechanisms

As mentioned above, we use active replication and majority voting to tolerate node faults. To enforce replica determinism in this voting while also tolerating channel faults, we proposed the consistent replicated voting protocol of [3]. This protocol is divided into the following two parts:

1) *Cc-vector exchange protocol* (CVEP). Replicas first exchange their cc-vectors. Upon the reception of a cc-vector, each replica sends the acknowledgment (ACK) that the cc-vector has been successfully received. The FTT-Master keeps track of which messages have been exchanged and forms a matrix called *messages status vector*, MS-vector for short. As can be seen in Figure 2, this matrix can take one of two values (0/1) representing received or lost message.

In the MS-vector we can discern two different cells having two different meanings: a) diagonal cell [i,i] that indicates whether the switch has received the cc-vector produced and published by the node replica i.

	M_1	M_2	...	M_n
R_1	0/1	0/1	...	0/1
R_2	0/1	0/1	...	0/1
...	...	...	...	0/1
R_n	0/1	0/1	0/1	0/1

Fig. 2. MS-vector

b) non-diagonal cell [i,j] that indicates whether the switch has received an ACK from node replica i. This ACK signifies whether the node replica i has received the cc-vector previously published by the replica j.

If some cell value is 0, it means that some message has been lost and the CVEP will then order the retransmission of the appropriate message. The number of retransmissions is envisioned to ensure that channel transient faults are tolerated.

The switch keeps the received messages. If the switch has the message it is the one responsible for retransmissions, if not, the replica retransmits. Once the switch has the message it must have forwarded it to the other replicas. If we allowed the replica to retransmit the messages that the switch has, it may happen that the replica becomes faulty and sends the erroneous messages and these messages might be different from the ones received before by the other replicas. This cannot happen with the switch though since it has crash failure semantics.

2) *Voting Set-Up Algorithm* (VSUA). Once the messages are exchanged, the FTT-Master piggybacks the MS-vector on the TM and each replica receives it. This is when VSUA comes in place. This algorithm is performed locally by replicas and it inspects the exchanged messages information contained in the MS-vector. The result of the algorithm is a decision of which replicas and messages are to be used by the majority voting.

More details about CVEP and VSUA are found in [3].

IV. Coordinated dispatching of tasks and messages

This section summarizes the strategy we proposed in [4] for coordinating the dispatch of tasks and messages executed/transmitted by node replicas.

This strategy is thought for general control applications. As it is well known, control applications are generally divided into three phases: sense (acquisition of data), control (processing of acquired data) and actuation (perform the action set by the control phase). These phases are repeated cyclically with a period called *sampling period* (T_s).

FT4FTT uses active replication and distributed majority voting. Therefore in FT4FTT the phases of a control application have to be designed so that replicas can exchange and vote on both, the information obtained from the sensors and the information calculated for the actuation.

Specifically, the following five phases are proposed in FT4FTT: 1) Sense; 2) Exchange of sensor values; 3) Voting on sensor values + Control; 4) Exchange of actuation values; 5) Voting on actuation values + Actuate; Worst case execution time of each phase (C_i) has to be calculated and sum of all phases has to be less or equal to the sampling period. Note

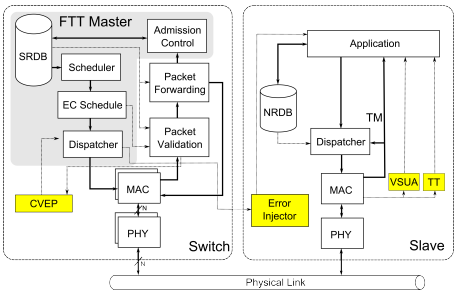

Fig. 3. Modeled general architecture of an FT4FTT-based DES

that the sum of each one of these phases (C_i) has to be less or equal to the sampling period, i.e. $\sum_{i=1}^{5} Ci < T_s$.

To trigger the tasks of phases 1, 3 and 5 the TM is used implicitly. More specifically, each replica has an EC counter that increases with the reception of TM, as well as a table that specifies which tasks have to be triggered for each EC value.

As regards how to trigger the transmission of messages of phases 2 and 4, FT4FTT uses the polling mechanism already provided by the TM in FTT. In this sense recall that the FTT Master keeps a table, called Synchronous Requirements Table (SRT), which includes the data about the message scheduling.

More information about the dispatching is found in [4].

V. OMNET++ SIMULATION SYSTEM MODEL

Figure 3 sketches the general modeled functional architecture of a FT4FTT-based DES. It differs from the previous model [5] by *CVEP*, *VSUA*, *TT* and *Error Injector* blocks.

On the left side is the switch. It receives Ethernet frames via all the ports (*PHY* and *MAC*), checks each of them for validity (*Packet Validation*) and forwards them (*Packet Forwarder*). The *Scheduler* constructs TMs and the *Dispatcher* dispatches them to the rest of the slaves. For more details about the functioning of the aforementioned blocks the reader can refer to [5]. The added block *CVEP* is responsible for incorporating homonymous protocol to the switch.

On the right side there is one of the Slaves, i.e. one of the replicas. Its structure is similar to the corresponding subset of blocks of the switch. The main difference is that the slave includes the following blocks: the *VSUA* block responsible for incorporating homonymous algorithm; the *TT* block responsible for the task triggering mechanisms; and the *Error Injector* block intended to inject errors.

The detailed simulation OMNET++ model of the FT4FTT-based system is depicted by Figure 4. The modules colored in gray are INET framework modules. Both switch and slave have INET modules for Ethernet interfaces *ETH*.

We will now explain the functioning of all the modules with a special focus on the modules related to the node fault tolerance and coordinated dispatching mechanisms, as they are the main contributions of this work. Next, the functions of all modules are summarized, with special focus on the ones that are new with respect to the previous model in [5], i.e. the node fault tolerance and coordinated dispatching mechanisms.

The *System Requirements Database (SRDB)* and *Node Requirements Database (NRDB)* modules keep the FTT protocol and scheduling information for the switch and the node respectively. This information is used for packet validation and scheduling. In particular, module *SRDB* includes the information about *SRT*. These modules are populated on system initialization by defining the proper xml files to read from.

The *Forwarding Table* module keeps the information of which node is connected to which port and is used by the *Dispatcher* to forward packets.

The *NRT, Sync* and *Async* modules are queues storing non real-time, synchronous and asynchronous packets respectively.

Next follows a step-by-step overview of how the simulation model works.

Each message exchange phase (2 and 4) is constituted by several ECs. The Switch starts each EC by constructing the TM in the *Scheduler* module. The TM is then sent to the *Dispatcher* module which broadcasts it to all outgoing ports.

In the initial EC of phases 2 and 4, each Slave receives the TM containing the list of synchronous messages that should be sent in that EC. In each Slave the *Dispatcher* module decodes the TM and instructs the *Application* module to send the synchronous messages the Slave produced (cc-vectors).

These cc-vectors are received, validated, and stored in the *Sync* queue by the Switch. Therein, the Switch *Dispatcher* fetches the packets from the *Sync* queue and forwards them to the *CVEP* module. Upon the reception of each synchronous packet, *CVEP* populates the appropriate diagonal value in the MS-vector. This module also saves all these synchronous packets in a vector that has one slot per slave and is called *retransmissionVector*. Finally, *CVEP* forwards all the packets to the corresponding outgoing ports.

The forwarded cc-vectors are received by the Slaves. When a Slave receives one of these packets, it passes it to its *Application*, which in turn sends the ACK packet in the asynchronous window of the EC. The Switch eventually stores all Slaves ACKs in the *Async* queue and, then, its *Dispatcher* forwards the Async packets (ACKs) to the *CVEP*. This last module populates the appropriate non-diagonal MS-vector values.

When the initial EC ends, what follows is a set of ECs devoted for retransmissions. Each one of these ECs contains the same schedule as the initial one. In principle this means that the communication pattern is repeated. But there is a subtle difference: if the Switch receives a cc-vector it has already stored in the *retransmissionVector* in a previous EC, it drops the just received cc-vector and forwards the already stored one.

Note that in each retransmission EC the Slaves send the same packets again, even though some of them will be dropped if the Switch already has them. The reason for this behavior is twofold. First, the bandwidth reserved for these packets would otherwise be non-utilized, and the packets can be used as a signal that the slaves are still operational. Second, it makes the model of the FTT-Master simpler, as the Master does not need to calculate a new schedule for each retransmission EC.

Once the initial and all the retransmissions ECs end, the *CVEP* module piggybacks the MS-vector on the next TM. When the Slaves receive it, their *VSUA* modules extract the MS-vector and perform the algorithm upon it deciding which messages and replicas are to be used by the voting.

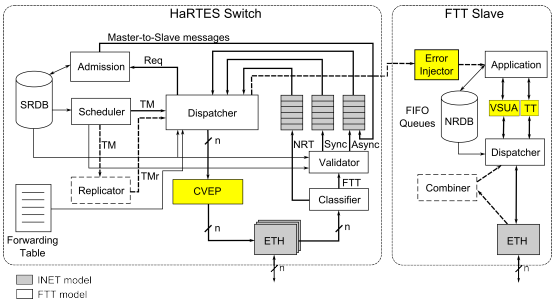

Fig. 4. OMNET++ simulation model

As concerns how the dispatching of tasks is modeled, note that the *TT* module is used for triggering the tasks in the Application module, e.g. the tasks that perform the sense phase. For this purpose, the *TT* module contains an EC counter it increases with the reception of each TM, as well as a table that specifies the correspondence between the value of this counter and the tasks to be triggered.

The *Application* module simulates the control application. In particular, we used it to simulate the majority voting on the received cc-vectors and a PID controller [13].

Finally the *Error Injector* module allows injecting errors in the *Application* module, in order to check the correctness of the simulated FT and dispatching mechanisms. Note that it is connected with the switch *Dispatcher* module. This allows the injector to be synchronized with the EC state kept by the switch, e.g. to know if the synchronous window has started.

VI. Results

We have used the OMNET++ model herein proposed to carry out a series of tests to assess the correctness of the CVEP, VSUA, majority voting and the dispatching mechanisms.

First, as was done in the previous model [5], we have injected errors in the links by setting specific Bit Error Rates (BER). This was done to asses the correctness of the mechanisms when transient faults in the links do occur.

Second, we used the Error injection module to inject different errors in the slaves, e.g. to drop a packet going to/from the application, to change the value of the packet etc. We modified only the FT mechanisms related packets: cc-vectors and ACKs. This was done to simulate permanent and transient faults affecting the slaves (node replicas).

Once the errors in the links and in the slaves were injected we have tested whether the mechanisms that we have introduced worked as planned. We have done this by printing specific variables from different modules, e.g. the voting values and the voting result from the *Application* module, the input and the result of the VSUA algorithm from the *VSUA* module, the MS-vector values form *CVEP* module, etc.

The simulation model allowed us to approach the real system as close as possible and to test whether our mechanisms worked as intended. The process of designing and simulating guided us to discover some unexpected scenarios and further refine our mechanisms.

VII. Conclusions and Future Work

The contribution of this work was to extend a previous model of the FT4FTT architecture in order to be able to simulate and test a subset of FT mechanisms related to the node fault tolerance and the mechanisms for the dispatching the task and messages performed and transmitted by node replicas.

In the future, we plan to simulate the rest of the FT4FTT project mechanisms to further evaluate their correctness.

Although the model has proven to be a valuable tool to assess and refine the mentioned mechanisms, we are aware about the limitations of simulation to inject all possible error scenarios. Moreover, we are also interested in quantifying the dependability of these mechanisms and of the whole FT4FTT architecture. For these reasons we plan to carry out a formal verification of the CVEP and VSUA as well as a dependability evaluation by means of other formalisms.

VIII. Acknowledgments

Supported by project DPI2011-22992, by the Portuguese Government through FCT - Fundação para a Ciência e a Tecnologia in the scope of project Serv-CPS -PTDC/EEA-AUT/122362/2010 and by FEDER funding.

References

[1] P. Pedreiras, L. Almeida, and P. Gai, "The FTT-ethernet protocol: Merging flexibility, timeliness and efficiency," in *24th Euromicro Conf. on Real-Time Systems*. IEEE Computer Society, 2002, pp. 152–152.

[2] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Distributed Computing Systems, 2000. Proc. 20th Int. Conf. on*. IEEE.

[3] S. Derasevic, M. Barranco, and J. Proenza, "Appropriate consistent replicated voting for increased reliability in a node replication scheme over FTT," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*.

[4] S. Derasevic, J. Proenza, and M. Barranco, "Using FTT-ethernet for the coordinated dispatching of tasks and messages for node replication," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*.

[5] M. Knezic, A. Ballesteros, and J. Proenza, "Towards extending the OMNeT++ INET framework for simulating fault injection in ethernet-based Flexible Time-Triggered systems," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE, 2014, pp. 1–4.

[6] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in *Proceedings of the European simulation multiconference (ESM2001)*, vol. 9, no. S 185. sn, 2001, p. 65.

[7] T. Steinbach, H. D. Kenfack, F. Korf, and T. C. Schmidt, "An extension of the OMNeT++ INET framework for simulating real-time ethernet with high accuracy," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pp. 375–382.

[8] R. G. V. d. Santos, "Enhanced ethernet switching technology for adaptive hard real-time applications: Tecnologia de comutação ethernet melhorada para aplicações adaptativas e críticas de tempo-real," 2011.

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

[10] D. Gessner, J. Proenza, and M. Barranco, "A proposal for master replica control in the flexible time-triggered replicated star for Ethernet," in *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop*.

[11] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, 1978, pp. 3–9.

[12] J. Proenza, M. Barranco, J. Llodra, and L. Almeida, "Using FTT and stars to simplify node replication in CAN-based systems." in *ETFA*, 2012.

[13] K. J. Aström and T. Hägglund, "PID controllers: theory, design and tuning," 1995.