Towards Extending the OMNeT++ INET Framework for Simulating Fault Injection in Ethernet-Based Flexible Time-Triggered Systems

Mladen Knezic
FEE, University of Banja Luka, Bosnia and Herzegovina
Email: mladen.knezic@etfbl.net

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

***Abstract*—Traditional distributed embedded systems are configured using static environment information and thus do not support dynamic behavior of the system. The necessary flexibility in the system may be provided by the Flexible Time-Triggered (FTT) communication paradigm. If, in addition, it is required that the system operates continuously, the suitable fault tolerance mechanisms that provide high reliability have to be developed and deployed in the system. To be able to successfully assess those mechanisms, it is reasonable to develop simulation models that support injection of various types of faults.**

This paper describes an OMNeT++ simulation model for distributed systems that are based on the Hard Real-Time Ethernet Switching (HaRTES) implementation of the FTT paradigm. The contribution of the paper is twofold. First, we provide a library of components that are required for modeling FTT networks with arbitrary number of FTT slaves connected to a HaRTES switch, and second, we used the developed components to build an FTT system that is suitable for assessing some of recently proposed mechanisms for tolerating certain transient faults in the communication channel.

I. Introduction

Traditional distributed embedded systems (DESs) have not been conceived to support both dynamic behavior of the system and highly-reliable continuous operation. The Flexible Time-Triggered (FTT) communication paradigm provides flexibility in the system by means of on-line QoS management with arbitrary scheduling policies [1]. It follows the master/multi-slave communication model, which means that the master polls several slaves using a special message called *Trigger Message* (TM). In FTT, the communication is divided into rounds of fixed duration called *Elementary Cycles* (ECs). Each EC starts when the FTT master sends a TM, which contains information about the current schedule (*EC Schedule*), i.e. which messages should be transmitted by which slaves in the current EC, and synchronizes all slaves. This way, message scheduling is always under control of the FTT master and can be adapted on-line depending on the current state of the system. The current schedule is calculated by the FTT master using information from the *System Requirements Database* (SRDB). The EC is further divided into two windows: the *synchronous window*, used for transmission of synchronous messages, and the *asynchronous window*, used for transmission of asynchronous messages.

Currently, there are several implementations of the FTT communication paradigm. Initially, the FTT was developed for CAN (*Controller Area Network*), a widely used protocol in DESs (especially for automotive applications). Afterwards, it was adapted to be used with other types of networks including Ethernet (FTT-Ethernet) and Switched Ethernet (FTT-SE). *Hard Real-Time Ethernet Switching* (HaRTES) may be considered as an evolution of the FTT-SE protocol that brings in many improvements such as simplified asynchronous traffic handling, increased system integrity, seamless integration of legacy nodes, and improved network synchronization [2].

To enable high reliability, several fault tolerance mechanisms based on HaRTES have been proposed recently [3]–[5] as a part of the *Fault Tolerance for Flexible Time-Triggered Ethernet-based systems* (FT4FTT) project, which aims at providing high reliability and flexibility to a DES. To evaluate the proposed fault tolerance mechanisms, it is faster and often more convenient to develop adequate simulation models and perform exhaustive simulations instead of implementing the real prototypes. In addition, fault injection is usually easier to enforce in a simulation environment.

OMNeT++ is a highly modular, easily extensible component-based C++ discrete event simulation library and framework, designed to support modeling of very large communication networks built from reusable model components [6]. The INET framework [7] is an open-source library for OMNeT++ that contains models for large number of wired and wireless communication protocols. It is worthy of remark that in OMNeT++, a collection of components that implements services of a protocol is referred to as *simulation model* or *simulation library*, whereas the simulation instance of some specific system is called *network module*.

This paper presents a simulation model for distributed systems that are based on the HaRTES implementation of the FTT communication paradigm. The model has been built on top of the OMNeT++ INET framework, and provides the library of components that can be used to simulate the behavior of an FTT network with an arbitrary number of FTT slaves connected to a HaRTES switch. Given that HaRTES is one of the Ethernet-based implementations of the FTT paradigm, the modules that provide the functionalities related to the Ethernet protocol can be reused directly from the INET library. Other modules were specifically developed to support specific services of the FTT protocol. To assess some of the recently proposed mechanisms for tolerating certain transient faults in the communication channel, we built an FTT system (OMNeT++ network module) using the developed components and OMNeT++ modules that enable injection of pseudo-random transient faults into the communication channel based on the specified bit error rate (BER). Finally, we present and discuss some preliminary results of the simulations.

II. An Overview of HaRTES

The simplified functional architecture of HaRTES is illustrated in Fig. 1. The figure shows the architecture of both a HaRTES switch and an FTT slave. For simplicity, the switch in the figure has only one Ethernet port (in case of a real implementation, the number of MAC and PHY blocks

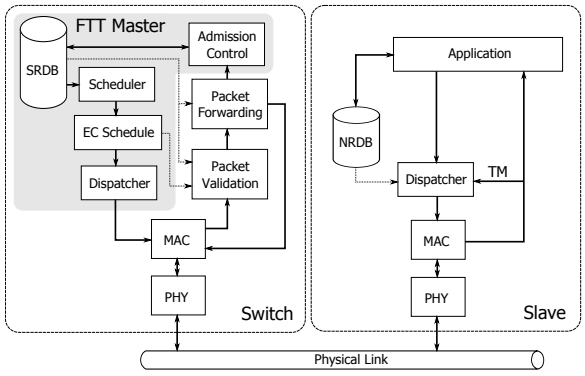

Fig. 1. Simplified functional architecture of a HaRTES switch and an FTT slave (based on [2] and [8]).

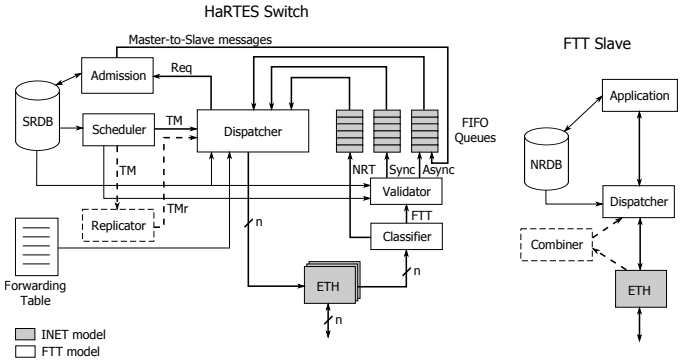

Fig. 2. HaRTES switch and FTT slave modules in OMNeT++.

would correspond to the number of ports). Moreover, the real architecture of the switch contains additional function blocks such as queues and port dispatchers that are left out for brevity.

As can be seen from Fig. 1, the HaRTES switch contains an FTT master and function blocks that are used for validation and forwarding of FTT packets according to the rules in the SRDB. Since the FTT master is integrated in the switch, FTT slaves can transmit asynchronous messages autonomously without a need for signaling mechanism used in FTT-SE, which significantly simplifies asynchronous traffic handling.

When an Ethernet frame is received, it is first checked for validity by the *Packet Validation* module against the current schedule found in the *EC Schedule* (synchronous messages) and other attributes from the SRDB (e.g. inter-arrival time in case of asynchronous messages). All messages that do not meet the system requirements are discarded, which increases the system integrity. After validation, messages are enqueued according to their type (real-time synchronous/asynchronous or non real-time) and afterwards processed in the *Packet Forwarding* module, which relays them to the corresponding ports based on the unique message identifier (*Stream Id*) within the FTT message. Non real-time (NRT) messages are enqueued and processed separately using standard MAC relaying logic under the control of the FTT master to avoid congestion, which enables seamless integration of legacy nodes. A special type of *Slave-to-Master* asynchronous messages that are used for requesting updates in the SRDB are forwarded directly to the *Admission Control* module for feasibility evaluation.

An FTT slave in the HaRTES network retains the same architecture as in FTT-SE. The *Node Requirements Database* (NRDB) in the FTT slave is the SRDB counterpart, which stores local information about the messages and their attributes for a particular FTT slave. This database is managed by the application, but its changes are triggered by the FTT master using dedicated *Master-to-Slave* asynchronous messages [8]. The *Dispatcher* module in the FTT slave is responsible for transmitting the messages according to the schedule decoded from each TM and the attributes from the NRDB.

III. Simulation Model Description

Component-based modeling is an appealing feature of OMNeT++ that enables hierarchical model design with reusable components called *modules*. Modules can be *simple* (module functionality is encapsulated in the corresponding C++ class) or *compound* (module functionality is implemented by combining a number of simple and/or compound modules). A special type of compound module that is placed at the top of the hierarchy is called *network module*. In order to run a simulation, the user must provide at least one network module.

Our FTT simulation model corresponds to the HaRTES architecture presented in Fig. 1 and is designed using a modular approach. Given that the fault tolerance mechanisms proposed so far within the FT4FTT project provide no guarantees for the delivery of asynchronous messages, the current implementation of our model is mainly focused on synchronous traffic, and therefore offers only limited support for asynchronous messages. In particular, handling the *Slave-to-Master* and *Master-to-Slave* asynchronous messages is not currently supported by the model. We must also note that NRT traffic is not supported by the model at this time.

A. HaRTES Switch

The structure of the *n*-port HaRTES switch module in OMNeT++ is shown in Fig. 2. The module and connections represented with dashed lines are not part of the normal HaRTES switch, but rather provide the functionalities related to some of the FT4FTT fault tolerance mechanisms in the switch, which will be described later. Moreover, the functions for handling special asynchronous messages (*Slave-to-Master* and *Master-to-Slave*) and NRT traffic are not currently implemented. The modules responsible for these functions (the *Admission* and the *Dispatcher* module) are currently designed to simply discard these messages upon reception.

As can be seen from Fig. 2, some modules (such as the Ethernet interface ETH, which contains MAC and PHY components, and FIFO queues) are reused directly from the INET library. Other modules are specifically developed to simulate the behavior of the different function blocks described in Section II. We will first describe the modules that are not directly used for packet processing, but instead provide global information related to the system configuration.

The *SRDB* module stores the system attributes that are necessary for scheduling and validating the packets (message deadline, period, size, inter-arrival time, etc.). It is initialized at the beginning of the simulation using information from a specifically defined XML file.

The *Forwarding Table* module is used by the *Dispatcher* to forward data packets to the corresponding ports of the switch. The forwarding rule is based on *Stream Id* following the publisher/subscriber communication model rather than using MAC address information. The content of the table is initialized using information from a specifically defined XML file on simulation startup.

In FTT, slaves are instructed by the FTT master to transmit the synchronous messages according to the current schedule conveyed by the TM in each EC. When an Ethernet frame from an FTT slave is received by the switch, it is first examined inside the ETH in order to detect if the frame was corrupted during transmission, and, if correct, it is delivered to the *Classifier*, which is responsible for the identification of the packet type. Non real-time packets are enqueued directly in the NRT FIFO queue, whereas FTT real-time packets are passed to the *Validator* module. The *Validator* module validates FTT frames received from the *Classifier* using information from the *SRDB* and the current schedule. Depending on the message type, valid FTT packets are enqueued either in the Sync (synchronous messages) or the Async (asynchronous messages) FIFO queue.

Each EC is initiated by the *Scheduler* module. Periodically, it constructs a TM with the current schedule and sends it to the *Dispatcher* module. The period of the EC is defined during the simulation startup. Clearly, the *Scheduler* takes over some responsibilities from the dispatcher (such as defining the time instant when each EC is initiated). The reason for this is reducing the number of message objects, which results in a more efficient simulation model. We must note that, at this point, the *Scheduler* only checks if each synchronous message should be sent in the current EC based on the offset and period attributes in the *SRDB*. However, its C++ class could be easily extended in order to enforce any scheduling policy.

Upon reception of the TM, the *Dispatcher* module broadcasts it to all ports. The ETH in each port encapsulates the TM into an Ethernet frame and transmits it over the communication channel. After broadcasting the TM to all ports, the *Dispatcher* gets the packets from the Sync FIFO queue and forwards them to the corresponding ports based on the information obtained from the *SRDB* and the *Forwarding Table*. Afterwards, the *Dispatcher* transmits the packets from the Async FIFO queue. Since the functions for relaying NRT traffic are not yet implemented in the model, messages retrieved by the *Dispatcher* from the NRT FIFO queue are simply discarded upon reception. As in the case of the *Scheduler*, in order to attain a more efficient model, we combined packet forwarding and dispatching functions in one module (the *Dispatcher*).

As explained in section II, the *Admission* module is responsible for managing the changes to the *SRDB* that are requested by FTT slaves using *Slave-to-Master* asynchronous messages called *slave update requests* (*Req*). Given that asynchronous messages are not yet fully supported by the model, the *Slave-to-Master* messages, which are forwarded by the *Dispatcher*, are discarded upon reception by the *Admission* module.

B. FTT Slave

The structure of the FTT slave module in OMNeT++ is shown in Fig. 2. Similar to the HaRTES switch, some modules (ETH) are reused from the INET library, whereas others (marked with dashed lines) provide the functionalities related to the fault tolerance mechanisms which will be described later.

The *Dispatcher* module receives and decodes each TM and builds the synchronous messages that should be transmitted in the current EC, using data provided by the *Application* module. It also delivers to the *Application* module the synchronous messages that are targeted to this specific FTT slave.

The *NRDB* module is almost equal to the *SRDB* in the switch. The only difference is that it contains only local information for a particular FTT slave.

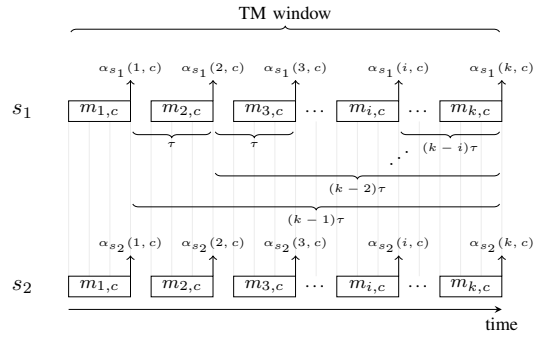

Fig. 3. Synchronization mechanism in two slaves (reproduced from [5] with the permission of the authors).

C. Fault Tolerance Modules

The correct reception of the TM of each EC is critical for the operation of FTT. Given that this message is used to convey the current schedule and to synchronize all slaves, failing to receive it would lead to a complete omission of the EC in a slave. In order to prevent this, a fault tolerance mechanism based on the temporal replication of the TM was proposed in [5]. In particular, the HaRTES switch transmits the TM of each EC multiple (k) times within a time slot called *TM window* (TMW). The value *k* depends on the BER in the channel, and if well calculated ensures that each slave receives at least one TM even in presence of transient faults in the links. In each TMW, TM replica transmissions are spaced uniformly in time using a constant period (τ) named *TM inter-arrival time*. Given that a slave might receive all or only a subset of k replicas (due to transient link faults), a synchronization mechanism must be adopted to ensure that the start of the *synchronous window* is aligned properly in each slave. To achieve it, the TM must convey some additional information: the value of k; a sequence number for each TM replica, which takes values from 1 to k; and the value of τ.

We will briefly describe the synchronization mechanism proposed in [5] using Fig. 3. Let us assume that $m_{i,c}$ and $m_{j,c}$ are replicas of the TM with sequence numbers i and j ($i \neq j$) received by the slaves s_1 and s_2, respectively, in EC c. Also, let $\alpha_{s_1}(i,c)$ and $\alpha_{s_2}(j,c)$ be the time instants when $m_{i,c}$ and $m_{j,c}$ are received by each slave. Both s_1 and s_2 will start the *synchronous window* at the same time if we set a timer in each slave to expire after $(k-i)\tau$ and $(k-j)\tau$ time units after the reception of $m_{i,c}$ and $m_{j,c}$, respectively. The timer expiration time should be recalculated every time a new TM is received by the slave.

The functionality of the described synchronization mechanism in OMNeT++ is provided by the modules represented by the dashed lines in Fig. 2. In the model of the switch that uses this synchronization mechanism, a TM from the *Scheduler* is sent to the *Replicator* instead of the *Dispatcher*. After reception of the TM, the *Replicator* creates *k* replicas (TMr), appends additional information necessary for the synchronization, and sends them to the *Dispatcher* for broadcasting.

In the model of the slave that uses this synchronization mechanism, the *Combiner* module collects the TMr messages, decodes them, and forwards the current schedule to the *Dispatcher* at the beginning of the *synchronous window*.

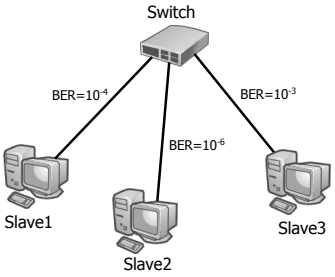

Fig. 4. Simulated network.

As explained above, the start of the *synchronous window* is determined with $(k-i)\tau$, where i is the sequence number of the TMr. This value is used to set a timer in the *Combiner*, which is rescheduled every time a new TMr is received. When the timer expires, the current schedule is sent to the *Dispatcher*, so it may start transmitting the scheduled synchronous messages.

IV. Simulation Results

To demonstrate the suitability of the proposed modeling technique for the evaluation of FTT-based fault-tolerant architectures by means of fault injection, we have assessed the described fault tolerance mechanism in presence of transient faults in the channel. More specifically, we performed a number of simulations on the FTT system shown in Fig. 4.

The system consists of three FTT slaves connected to a HaRTES switch using 10 m long 100 Mbps Ethernet links with different BER for each slave link as given in Fig. 4. In OMNeT++, transient link faults are injected into the communication channel in a pseudo-random fashion based on the packet error probability $p_p = 1 - (1 - BER)^N$, where N is the bit length of the packet. Each message has a special field called *bit error flag* that is set during transmission according to the probability p_p. This flag is examined in the ETH of each node in order to discover if it was corrupted during transmission.

Since our main goal was to assess the robustness of the proposed fault tolerance mechanism, the system was configured to send replicated TMs without scheduling data (i.e. the slaves do not transmit synchronous messages). We also adjusted some modules in the switch (the *Scheduler*) and slaves (the *Dispatcher*) to collect certain statistical information of the system. In particular, we count the number of ECs initiated by the *Scheduler* and the number of successfully processed ECs in each slave. The EC is considered successful if the start of the *synchronous window* in the slave matches the expected reference time that coincides with the end of the TMW. Given that these time instants could differ due to the rounding errors in OMNeT++ caused by the floating-point arithmetic, we consider that the slave is properly synchronized if their difference is less than 1 ns. To differentiate EC omissions caused by the transient faults in the channel from those originated in the simulation model (due to potential modeling errors), we also counted the number of ECs in which the slave successfully received at least one TM.

In order to obtain statistically confident results, we configured the simulation environment to run one million ECs with 1 ms period and $\tau = 10\mu s$. We repeated the simulation using different k values. The obtained results are presented in Table I. The table shows the number of successfully processed ECs. The same values were obtained in case of the number of successfully received TMs, which means that the behavior of the simulation model is correct (i.e. if slave receives at least one TM, it will be properly synchronized).

TABLE I. Percentage of Successfully Processed ECs

k	Slave1	Slave2	Slave3
1	94.3675%	99.9398%	56.2075%
2	99.6966%	100%	80.7965%
4	99.999%	100%	96.329%
8	100%	100%	99.8675%
16	100%	100%	99.9998%

From the table, we can see that all ECs are successfully processed if $k \geq 2$ in case of $BER = 10^{-6}$ (the link of Slave2) and if $k \geq 8$ in case of $BER = 10^{-4}$ (the link of Slave1). In case of $BER = 10^{-3}$ (the link of Slave3), it is evident that 100% percentage cannot be achieved even with $k = 16$. The percentage of successfully processed ECs in this case is equal to 99.9998%, which means that only two out of one million ECs are missed by the slave.

V. Conclusions and Future Work

In this paper we presented an OMNeT++ simulation model for distributed systems based on the HaRTES implementation of the FTT paradigm. Moreover, we showed how the proposed modeling technique could be used for the evaluation of FTT-based fault-tolerant architectures by means of fault injection, and presented some preliminary results obtained by simulating an FTT system that was built using the model's components.

In our future work, we will extend the model in order to support and assess via fault injection other fault tolerance mechanisms. Also, we will consider how the developed model could be utilized for addressing some performance aspects of distributed systems based on HaRTES.

Acknowledgment

This work was supported by project DPI2011-22992 (Spanish *Ministerio de Economía y Competividad*) and by FEDER funding. Mladen Knezic was supported by a scholarship of the EUROWEB Project (http://www.mrtc.mdh.se/euroweb), which is funded by the Erasmus Mundus Action II programme of the European Commission.

References

[1] P. Pedreiras and L. Almeida, *The Flexible Time-Triggered (FTT) Paradigm: an Approach to QoS Management in Distributed Real-Time Systems*, Proc. Int. Parallel and Distributed Process. Symp., 2003.

[2] R. G. V. dos Santos, *Enhanced Ethernet switching technology for adaptive hard real-time applications*, PhD thesis, Univ. de Aveiro, 2010.

[3] G. Rodriguez-Navas and J. Proenza, *A proposal for flexible, real-time and consistent multicast in FTT/HaRTES Switched Ethernet*, 18th IEEE Conf. on Emerging Technologies and Factory Automation, 2013.

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

[5] D. Gessner, J. Proenza, and M. Barranco, *A Proposal for Managing the Redundancy Provided by the Flexible Time-Triggered Replicated Star for Ethernet*, 10th IEEE Int. Workshop on Factory Commun. Syst., 2014.

[6] A. Varga, *The OMNeT++ Discrete Event Simulation System*, Proc. of the European Simulation Multiconference, 2001.

[7] The INET Framework. An open-source communication networks simulation package for the OMNeT++ simulation environment. Online: http://inet.omnetpp.org/

[8] A. Ballesteros and J. Proenza, *A description of the FTT-SE protocol*, Technical report, DMI, Universitat de les Illes Balears, 2013.