First Experimental Evaluation of the Consistent Replicated Voting in the Hard Real-time Ethernet Switching architecture

Sinisa Derasevic, Maties Melià, Alberto Ballesteros, Manuel Barranco and Julián Proenza
Dept. Matemàtiques i Informàtica, Universitat de les Illes Balears, Spain
{sinishadj, maties.melia.galmes}@gmail.com, {a.ballesteros, manuel.barranco, julian.proenza}@uib.es

***Abstract*—Distributed Embedded Systems (DESs) typically have dependability and real-time requirements. Moreover, when they are deployed in dynamic environments, they must be flexible enough to adapt to changes in the operation requirements. The Fault Tolerance for Flexible Time-Triggered Ethernet (FT4FTT) project aims at providing a Switched-Ethernet architecture, based on the Flexible Time-Triggered communication paradigm (FTT), that is flexible and highly reliable. In particular, FT4FTT provides node fault-tolerance by means of active replication with majority voting. In this sense, FT4FTT includes the Consistent Replicated Voting (CRV) protocol to enforce replica determinism, even in presence of faults, while maximizing the reliability that can be achieved thanks to the node redundancy and the communication subsystem itself. This papers presents a first implementation of this protocol in a real prototype, and shows the on-going experimental evaluation been carried out to asses its correctness.**

I. Introduction

Distributed Embedded Systems (DESs) typically have demanding dependability and strict real-time requirements. Moreover, nowadays they are deployed in changing environments that impose a dynamic behaviour in the nodes and in the communication subsystem. Therefore, DES must not only be highly reliable, but also flexible enough to adapt to changes.

The Flexible Time-Triggered communication paradigm (FTT) [1] is a promising communication paradigm for these so-called adaptive DESs. FTT follows a master/multi-slave polling mechanism in which a master triggers the transmission of data messages in a set of slaves.

Communication is divided into fixed-duration time slots called Elementary Cycles (ECs), each of which is in turn divided into three windows. In the first one, i.e. the Trigger Message Window (TMW), the master broadcasts the so-called Trigger Message (TM). This message synchronizes all the slaves, as it indicates the beginning of a new EC; and it contains the EC-schedule, i.e., the messages that the slaves must transmit in that EC. The following windows are the Synchronous Window (SW) and the Asynchronous Window (AW), in which slaves transmit periodic and aperiodic messages respectively, according to the EC-schedule. FTT provides communication mechanisms for the master to change the schedule on-line, upon scheduling update requests from the slaves. These features make FTT specially well suited for adaptive DES, since nodes can exchange real-time periodic and aperiodic traffic in a flexible and adaptive manner.

Unfortunately FTT has scarce mechanisms to fulfill the reliability requirements of the most demanding highly-reliable DESs. To overcome this problem we are carrying out a project called FT4FTT (Fault Tolerance for Flexible Time-Triggered Ethernet). The main aim of FT4FTT is to provide Switched Ethernet infrastructure base in the FTT paradigm that is both flexible and highly reliable.

In particular one of the FT4FTT most important challenges is to provide mechanisms to tolerate hardware and software node faults. In the following, we will use the N-Version Programming (NVP) [2] terminology to describe these mechanisms, even though we assume no design diversity. Node's hardware faults are tolerated by means of *active replication*, i.e. critical nodes are replicated and each one of such replicas executes the same code. As concerns node's software faults, they are tolerated by using a voting mechanism that compensates the errors they generate. Specifically, replicas perform partial executions of the software in parallel, called *segments*. At the end of each segment each replica produces an output called *cc-vector*. Next replicas exchange their cc-vectors and vote on them to get a *consensus cc-vector*, which then they use to compute the next segment.

For this voting mechanism to work properly it is fundamental to ensure that each replica votes with the same set of cc-vectors. Otherwise the resulting consensus cc-vector will not be identical in all the replicas. This problem of providing each replica with the same input is called *external replica determinism* [3], and it is well-known to be an important issue of communication subsystems. In order to enforce external replica determinism in FT4FTT, even in the presence of permanent and transient node and network faults, we proposed in [4] a Consistent Replicated Voting (CRV) protocol.

This papers presents both a first implementation of the CRV protocol in a real prototype that aims at demonstrating its feasibility, as well as an on-going experimental evaluation to asses the CRV behaviour in presence of faults.

The paper proceeds as follows. Section II describes the main aspects of the CRV protocol. Then, Section III explains the most important aspects related to the implementation in a real prototype. After that, Section IV presents the set of experiments carried out and the results obtained from them. Finally, Section V concludes the paper by highlighting the contribution of this work and pointing out the future work.

 doi: 10.1109/ETFA.2015.7301632.

II. Basics of the CRV protocol

The Consistent Replicated Voting (CRV) protocol is divided into the Cc-vector Exchange Protocol (CVEP) and the Voting Set-Up Algorithm (VSUA). The former specifies the way in which cc-vectors are transmitted to maximize the number of cc-vectors that are consistently exchanged among all replicas, when permanent or transient faults affecting the replicas or the network do occur. As regards the VSUA, it specifies what subset of replicas and what subset of cc-vectors should be used for voting.

A. Cc-vector Exchange Protocol (CVEP)

The Cc-vector Exchange Protocol (CVEP) is a publisher-subscriber retransmission protocol in which replicas exchange, during an *exchange round*, the cc-vectors they have to vote on. Each round consists of a number of $k \geq 1$ ECs that are used to retransmit cc-vectors as many times as needed to provide a majority of replicas with at least one majority set of consistently exchanged cc-vectors. For the replicas to know which cc-vectors have been consistently exchanged, the master builds up a matrix called the *Message Status vector* (MS-vector) during the round. This matrix indicates, for each cc-vector, both: whether or not its publisher has transmitted it, as well as which subscribers have acknowledged its reception.

The CVEP is divided into 4 phases, as shown in Fig. 1: the *Schedule*, the *Broadcast*, the *Acknowledge* and the *Accept* phases. The first three phases are carried out in each EC of the round, whereas the fourth one is only done after the end of the last EC. The *Schedule* phase takes place during the TMW and is reserved for the master to broadcast within the TM the list of cc-vectors that the publishers have to transmit in that EC. Next, in the *Broadcast* phase, each publisher transmits, within the SW, the cc-vectors indicated by the TM. After the SW, in the *Acknowledge* phase, each subscriber sends, in the AW, an ACK for each message it has correctly received.

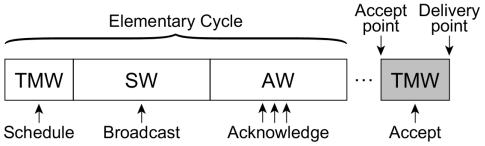

Fig. 1: Phases of an exchange round

After each EC, the master updates the MS-vector taking into account the set of cc-vectors have been successfully transmitted and acknowledged so far in the round. Moreover, for the sake of efficiency, it uses the information of the resulting MS-vector to schedule within the next EC only the cc-vectors each publisher has to retransmit.

Finally, the *Accept* phase occurs in the TMW of the first EC after the end of the last EC of the round. In the *Accept point*, i.e., at the beginning of that TMW, the master both: decides the final content of the MS-vector, and broadcasts it in the TM. In the *Delivery point* each replica executes the VSUA algorithm, on the basis of the content of the final MS-vector, to decide which replicas should vote on which cc-vectors and, then, this decision is delivered to the application.

B. Voting Set-Up Algorithm (VSUA)

As introduced previously, the Voting Set-Up Algorithm (VSUA) is executed in each replica during the Accept phase, and specifies the set of replicas and cc-vectors that will be used in the Delivery point for voting. The most important aspect of this algorithm is that it ensures both, that all replicas that can communicate are going to vote, and that at least all the cc-vectors sent by these replicas are going to be considered in the voting. In this, sense the system works as long as there is a majority of replicas that can communicate and that produce correct cc-vectors; where majority means $\lfloor X/2 \rfloor + 1$, where X is the number of replicas.

The input of the VSUA is the MS-vector, which is constructed by the master and contains the list of cc-vectors transmitted and acknowledged by each replica. As an example, let's assume the system with three replicas depicted in Fig. 2. *Replica 1* transmits its cc-vectors to replicas *2* and *3* through message *A*, *Replica 2* transmits its cc-vectors to replicas *1* and *3* through message *B*, and *Replica 3* transmits its cc-vectors to replicas *1* and *2* through message *C*. In this system, the MS-vector is the matrix shown in Fig. 3.

Each row in this matrix informs about the cc-vectors transmitted and acknowledged by a given replica. For instance, the first row specifies if *Replica 1* has transmitted message *A*, and if it has received messages *B* and *C*. Specifically, if cell [*R1*,*A*] is true, it means that the master has received the cc-vector from *Replica 1*. In contrast, if cells [*R1*,*B*] and [*R1*,*C*] are true, it means that the master has received the ACK message from *Replica 1* for messages *B* and *C*, respectively.

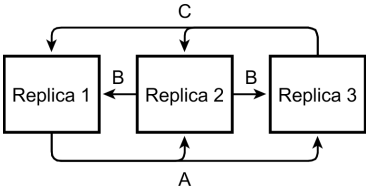

Fig. 2: Message exchange.

	A	B	C
Replica 1	T/F	T/F	T/F
Replica 2	T/F	T/F	T/F
Replica 3	T/F	T/F	T/F

Fig. 3: MS-vector.

Once the VSUA knows the set of cc-vectors transmitted/received, it generates all the possible voting combinations following the order next described. First, it fixes to the maximum the number of cc-vectors with which each replica votes. With this number of cc-vectors, VSUA generates all the combinations modifying the number of replicas. If the algorithm finds at least one combination in which a majority of replicas have consistently exchanged the cc-vectors, it takes the best one, i.e. the one that maximizes the number of replicas. Otherwise, the number of cc-vectors is fixed to the maximum minus one, and the generation of combinations is carried out again. This procedure is repeated until a solution is found or the number of cc-vectors does not represent a majority.

The VSUA finds the best solution, i.e. it maximizes the number of both cc-vectors and replicas. Moreover, the VSUA prioritizes the number of cc-vectors over the number of replicas. This criteria makes it possible to tolerate in some scenarios a number of faults greater than the one that can be strictly tolerated with X replicas, e.g. with $X = 3$ it allows tolerating not only a permanent fault affecting one replica, but also additional transient faults [4].

III. Implementation of the CRV protocol

We implemented the CRV protocol on top of a switched-Ethernet version of FTT, called Hard Real-Time Ethernet Switching architecture (HaRTES). In HaRTES the slaves connect to a custom switch which embeds the master. This yields important advantages in terms of the management of the aperiodic traffic, the implementation of fault-tolerant mechanisms and the delay of the master messages [5].

Next we describe how the regular implementation of HaRTES [1] [5] has been extended to cope with the new requirements. For simplicity, each exchange round takes one EC (see Fig. 1). This means that if a fault corrupts a cc-vector or an ACK, that cc-vector or ACK will be irreversibly lost. Certainly this decision limits the reliability with which cc-vectors and ACKs are exchanged and, ultimately, it does not prevent unnecessarily node replicas attrition. However it eases the implementation and the tests carried out to assess the fundamental features of the protocol, i.e. that non-faulty replicas correctly vote on the same set of consistently exchanged cc-vectors, and that the system provides its service as long as there is a majority of non-faulty replicas that consistently exchange a majority of cc-vectors.

In Fig. 4 we show the internals of one slave and the HaRTES switch. The slave is composed by the *FTT slave* which provides network services to the *Application*. The switch embeds the *FTT master* and the *Switching module*, which forwards messages from/to the master and the slaves. White rectangles depict the components that were not modified with respect to the prior version of HaRTES, white rectangles with a dashed border depict components that were partially modified, and grey rectangles depict the new components.

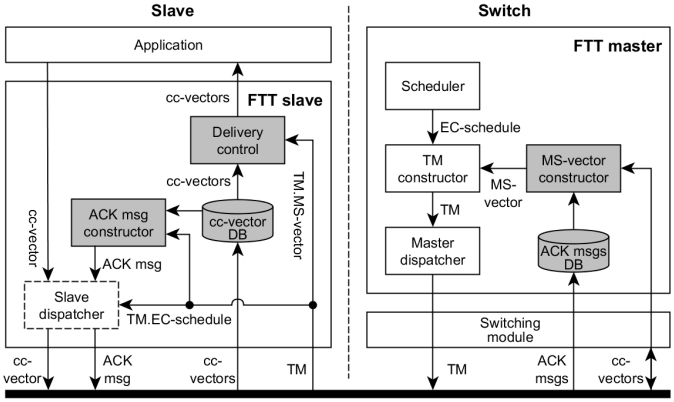

Fig. 4: Internals of one slave and the master.

During the Schedule phase, the *TM constructor* in the master gets the EC-schedule from the *Scheduler* and inserts it inside a TM. This TM is then passed to the *Master dispatcher*, which broadcasts it to the slaves during the TMW.

In the Broadcast phase, the *Slave dispatcher* uses the EC-schedule contained in the TM to determine the cc-vectors to be transmitted in the current EC and retrieves them from the application. After that, it transmits said messages within the SW. Moreover, when a slave receives a cc-vector from another slave, the *cc-vector DB* keeps it until the delivery point, i.e. until it is decided whether or not to deliver it to the application. Finally, also in this phase, the master also needs to receive the cc-vectors to fill the diagonal of the MS-vector.

In the Acknowledge phase, the *ACK msg constructor* of each slave builds an ACK message containing the list of received and non-received cc-vectors. This behaviour is different from the one described in Section II-A, in which each slave transmits one ACK message per cc-vector. This simplification eases the implementation and allows us to better diagnose the origin of the errors, as we can distinguish when a replica did not receive a message and when it was not able to transmit the NACK. Moreover, this modification does not affect the results of the evaluation since we can cover all the error scenarios by injecting faults in the cc-vectors. In order to construct the ACK message, the *ACK msg constructor* compares the list of received cc-vectors, maintained in the *cc-vectors DB*, with the list of cc-vectors that should have been received, contained in the EC-schedule. Once created, the ACK message is delivered to the *Slave dispatcher*, which transmits it inside the AW.

Note that, to completely acknowledge all the cc-vectors, the slaves must have enough time to receive them before constructing the ACK message. That is, it must be ensured that all the slaves start the AW and, thus, the Acknowledge phase, at the same time. However, FTT does not provide this intra-EC synchronization, i.e., when a slave finishes the transmission of its periodic messages (like a cc-vector), it immediately considers the SW as finished and starts transmitting the aperiodic ones. To solve this issue, we modified the *Slave dispatcher* in all the slaves to make the size of the SW static. In this way, if a slave finishes the transmission of its cc-vectors before the end of the SW, it waits until the beginning of the AW to trigger the construction of the ACK message.

Transmitted ACK messages are stored by the master in the *ACK msgs DB*. Later on, in the Accept phase, the *MS-vector constructor* consults this DB, builds up the MS-vector and inserts it into the TM together with the EC-schedule. As in the Schedule phase, the TM is transmitted, but now the MS-vector is passed to the *Delivery control*. This component uses the MS-vector to execute the VSUA and, then, delivers to the application the appropriate cc-vectors from the *Data msgs DB*.

IV. Experimental evaluation of the CRV protocol in HaRTES

The original paper of the CRV protocol [4] assessed the correctness of the VSUA by implementing it in Java. Next we present an extended evaluation performed over a real prototype that implements not only the VSUA but also the CVEP.

A. The HaRTES prototype

The prototype in which we have carried out the experiments is constructed in commercial-of-the-shelf (COTS) personal computers (PCs), and it is composed of three slaves connected among them through a custom switch. Note that the three slaves perform the same task in parallel, i.e., they are replicas.

The switch is implemented in a regular multi-core PC with two Intel 350-T4 quad-port Ethernet server adapters. This configuration has several advantages, such as the amount of ports, as well as the software and hardware flexibility. In contrast, each of the slaves is implemented in a Jetway JBC373F38-525-B barebones, i.e., a specific hardware for network embedded devices that includes an Atom processor and four standard Ethernet network interface cards (NICs).

From the point of view of the software, the switch and the slaves run a regular GNU/Linux OS. On top of it, the switch executes the Switching module and the FTT master (see Fig. 4). Conversely, each slave executes an FTT slave, which provides the FTT services to an Application. This application is a simplified version of a replicated control application that repeatedly executes a 3-phase scheme: sense, exchange sensor values and vote on them [6]. In order to inject channel errors, we modify the behaviour of some replica components during the exchange phase.

B. Fault-tolerance tests

The test campaign we have carried out is devoted to assessing the capacity of the CRV protocol to provide a consistent majority voting in the presence of faults that prevent replicas from communicating. For this, we injected faults that forced replicas to not transmit or receive cc-vectors.

It should be noted that, since the communication round takes just one EC, each injected fault is considered as a permanent one. However, this is not a limitation for this evaluation. Moreover, we do not force omissions of individual ACKs, nor we inject omissions in the transmission/reception of any ACK message (which would then force an omission of all the ACKs therein included). This is because we indirectly inject errors in individual ACKs by forcing an omission in the transmission/reception of the corresponding cc-vectors.

In order to study the effects of faults in the transmission, the *Slave dispatcher* component has been modified in all the replicas. Specifically, we force the omission of some cc-vectors before being transmitted. This fault injection was applied to one, two and three replicas in all the combinations. The result of this experiment is that, as expected, the protocol can tolerate no more that one transmission error at a time.

Reception faults were injected by forcing the *cc-vector DB* to not store cc-vectors, which in turn prevents the transmission of the corresponding ACKs. This was done for all the combinations of received cc-vectors in one, two and three replicas.

The results obtained from this experiment are shown in Table I. Each row specifies the fault-tolerance capacity of the CRV protocol, when injecting a given number of faults (#faults) in a communication round. Specifically, the second column (#scens) indicates the number of scenarios that were generated by injecting the fault/s; whereas each one of the rest specifies the proportion of these scenarios in which the VSUA allowed the system to vote, using a given number of replicas and cc-vectors (*#replicas / #cc-vectors*).

For instance, injecting 2 faults leads to 15 different scenarios. In the 20% of them the system could consistently vote using 2 replicas and 3 cc-vectors and, also, 2 replicas and 3 cc-vectors. Moreover, in the 60% of them the system could consistently vote using 2 replicas and 2 cc-vectors. As expected, the higher the number of faults, the lower the number of replicas and cc-vectors that can be used to consistently vote. However, results demonstrate that in some cases the CRV protocol allows to consistently vote while tolerating up to 4 faults, which is higher than the number of faults that could be strictly tolerated when using 3 node replicas. This happens when two replicas vote with the same two messages: the one they transmitted and one received from the other replica.

		#replicas / #cc-vectors				
#faults	**#scens**	**3/3**	**2/3**	**3/2**	**2/2**	**0/0**
0	1	100%	—	—	—	—
1	6	—	100%	—	—	—
2	15	—	20%	20%	60%	—
3	20	—	—	—	90%	10%
4	15	—	—	—	20%	80%
5	6	—	—	—	—	100%
6	1	—	—	—	—	100%

TABLE I: Tolerance of errors in the reception.

V. Conclusions and future work

We presented a first implementation of the the Consistent Replicated Voting, a voting protocol for enforcing node replica determinism in FTT-Ethernet networks. The presented real prototype is intended to demonstrate the protocol feasibility and to experimentally evaluate it. We tested some faults scenarios that allowed us to measure the fault-tolerance capacity of the protocol in the presence of faults that prevent replicas from receiving information from each other. In particular, we showed that in some cases the protocol can tolerate a number of receiving faults higher than the number of faults that could be strictly tolerated when using 3 node replicas.

The next steps involve a evaluation with more replicas, while taking into account faults in the transmission and in the reception at the same time. Moreover, we will also measure the efficiency of the implementation: delay introduced by the additional logic and accuracy of the synchronization needed to determine the beginning of the AW. Furthermore, it is also a pending issue to implement the CRV in the final architecture of FT4FTT in which the single point of failure that the switch represents is replaced by two interconnected switches. Finally, we are currently working in an OMNeT++ model to assess the CRV protocol via simulation, and in the formal validation of the CRV protocol by means of model checking.

Acknowledgments

This work was supported by project DPI2011-22992 (Spanish *Ministerio de economía y competividad*), by FEDER funding, and by the Portuguese Government through the FCT in the scope of project Serv-CPS-PTDC/EEA-AUT/122362/2010. Sinisa Derasevic was supported by a scholarship of the EUROWEB Project, which is funded by the Erasmus Mundus Action II programme of the European Commission.

References

[1] P. Pedreiras and A. Luis, "The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems," *Proc. Int. Parallel and Distributed Processing Symp.*

[2] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. on Soft. Engineering*, no. 12, pp. 1491–1501, Dec.

[3] S. Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers.

[4] S. Derasevic, M. Barranco, and J. Proenza, "Appropriate consistent replicated voting for increased reliability in a node replication scheme over FTT," in *Proc. 19th IEEE Int. Conf. on Emerging Tech. and Factory Automation (ETFA)*, Barcelona.

[5] R. Marau, P. Pedreiras, and L. Almeida, "Enhanced ethernet switching for flexible hard real-time communication."

[6] S. Derasevic, J. Proenza, and M. 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 Automation (ETFA)*, Barcelona.