Towards Dynamic Fault Tolerance on FTT-based Distributed Embedded Systems

Sinisa Derasevic, Julián Proenza, David Gessner
*
DMI, Universitat de les Illes Balears, Spain
sinishadj@gmail.com, julian.proenza@uib.es, davidges@gmail.com

Abstract

Distributed embedded systems (DES) have been traditionally designed to operate in static environments that do not change over time. Flexible designs are increasingly being introduced to achieve continuous and correct operation under dynamic environments. Some designs, such as the Flexible Time-Triggered communication paradigm (FTT), are focused on being able to modify the real-time operation upon changing requirements imposed by the environment. The on-going project Fault Tolerance for FTT (FT4FTT) purports to increase the reliability of a DES based on the FTT protocol by introducing static fault tolerance. In this paper we give some hints on how to go one step beyond by adding dynamic fault tolerance to the DES. This would result in new systems that would combine the qualities of flexible real-time operation and flexible and adaptive fault tolerance, much enlarging their sphere of applicability.

1. Introduction

Traditionally, distributed embedded systems (DES) have been designed to operate in static environments that do not change over time. This has led to static approaches that are inadequate for continuous and correct operation under dynamic environments. The alternative are flexible approaches. Some of these aim to modify the real-time operation upon changing requirements imposed by the environment. This is the case of the Flexible Time-Triggered communication paradigm (FTT) [7].

When faced with critical applications, real-time guarantees may not be enough and high reliability may also be required. With this idea in mind, the goal of the on-going project *Fault Tolerance for FTT* (FT4FTT) is to show that it is possible to significantly increase the reliability of a DES based on the FTT protocol by introducing static fault tolerance, e.g. by redundantly executing the tasks in several nodes and periodically voting on the replicas' results.

However, when high-reliability becomes a main concern in systems that must work in a changing environment, both the specific tasks to be executed and the probabilities of them suffering errors may also change over time. Therefore, being able to dynamically change which tasks are allocated to each node and what degree of replication, and thus fault tolerance (FT, hereafter), is granted to each task (e.g. in how many nodes it is redundantly executed) would clearly allow a more efficient use of the available resources.

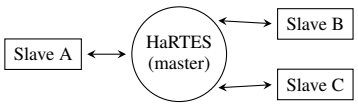

Figure 1. HaRTES architecture.

Here we propose to go beyond the goal of project FT4FTT by adding dynamic FT to the DES. This would result in new systems that combine the qualities of flexible real-time operation and flexible and adaptive FT, much enlarging their applicability.

2. The FT4FTT architecture

We take as a starting point the architecture that is being developed within the FT4FTT project. The communication subsystem of FT4FTT, currently under development, uses the Flexible Time-Triggered paradigm (FTT) [7]. Specifically, it replicates the switch of the Hard Real-Time Ethernet Switching (HaRTES) architecture, which implements FTT for Ethernet [9] providing highly reliable communication.

As shown in Figure 1, HaRTES implements a simplex, not replicated, microsegmented star topology, with the HaRTES switch as a central element that provides the most relevant functions of FTT. In particular, the switch embeds an FTT master. This master grants access to the network following a centralized master/multi-slave scheme. This means that a single message from the master triggers the transmission of messages in several slaves. Specifically, the master divides the communication time into rounds called *Elementary Cycles* (ECs), which are synchronized among and have the same constant duration in all simultaneous communications that may occur across the switch. The ECs are comprised of a *synchronous window* followed by an *asynchronous window*. The FTT master initiates each EC with the transmission of a *Trigger Message* (TM) that is flooded to all slaves. This message not only marks the beginning of a new EC, inciting the slaves to transmit their synchronous messages followed by their asynchronous messages, but it also dictates the schedule for the next synchronous window, i.e., it tells the slaves which synchronous messages they should transmit during that win-

 doi:10.1109/ETFA.2013.6648152

dow. The schedule is calculated by the master based on the contents of a *System Requirements Database* (SRDB). This database specifies the communication requirements for different message streams, which are sequences of messages related to the same entity (e.g., a sequence of readings of the same sensor) and are analogous to a task in processor scheduling. Example requirements are deadlines and periods, which are both expressed as multiples of the EC length. Slaves may request changes to the SRDB, but these requests are subject to an online admission control performed by the master. The admission control basically ensures that the SRDB is only updated with the requested change if the system will still be schedulable afterwards.

For FT4FTT to achieve its high-reliability goal, it also provides tolerance of faults in the nodes (now called slaves in FTT) of the DES by using node replication. Node replication, like any proper replication scheme, must ensure failure independence between replicas. This is achieved by preventing error propagation [6] from one node to another, i.e. by preventing an error generated in a given node from creating new errors in another node. Specifically, error propagation can be prevented by preparing non-faulty replicas to cope with the failure of another replica. The difficulty of this depends on how that failure can manifest, i.e., on the failure semantics [1] of the faulty replica.

If the node replicas have non-byzantine failure semantics, then replica failures can be handled much more easily. Enforcing such failure semantics can be achieved by local mechanisms at each node or by enhancing the communication subsystem appropriately. The latter approach is the choice in FT4FTT and it has the advantage that it can keep the nodes simpler. Moreover, if switches are used such that all communication must pass through them, which is the case in the communication subsystem of FT4FTT, then their global view of the communication can be exploited.

2.1. Node fault tolerance in FT4FTT

To achieve tolerance to node faults in FT4FTT we follow the guidelines from [8]. Therefore, our design uses *active replication* [10] to provide node fault tolerance, i.e. several nodes execute each a replica of the same program and, after each partial computation, they exchange their results and a voting takes place. We assume the replicas are identical pieces of software, allowing to tolerate hardware faults but not software (design) ones. Yet, we will generally follow the terminology of N-Version Programming (NVP) [8], despite NVP assuming replicas with design diversity.

Each replica is partitioned into segments. Each time a replica finishes executing a segment, it issues a vector of results of this segment, called *cc-vector*. Then a decision algorithm is executed to obtain a consensus cc-vector, which is sent back to all replicas to be used in the following computations. This mechanism, called cc-point, provides masking of faults in a minority of replicas. We will use *a-replica* to refer to any application program replica.

To improve the global dependability, FT4FTT performs three additional FT operations on the a-replicas: error detection (i.e. detection of a-replica errors); fault passivation; and recovery of an a-replica when it has suffered a temporary fault. Error detection is achieved by comparing the cc-vectors with the consensus cc-vector calculated by voting. Fault passivation is done by disconnecting a node from the network when it is affected by faults. Finally, a-replica recovery is achieved by sending to a faulty a-replica all the information it needs to resume its operation, as long as it has not reached a significant number of consecutive errors (which would be taken as indicating a permanent fault).

In principle, these operations are performed by the nodes themselves, e.g. a node detects its own errors and another node sends to a faulty node the information required for recovery. However, since nodes can internally fail arbitrarily, in order to increase the chances of errors to be properly detected and recorded, and of nodes correctly recovering after failures, a device (either an additional node or a new part of the switch) called Node Replication Manager (NRM) also executes the same voting as the nodes. This should provide it with all the necessary information to detect the errors of all nodes, keep an error counter for each of them, order the disconnection of one of them and also help a node to recover, in case the severity of its failure does not allow it to recover by simply using the last result it obtained in the voting. Therefore, the NRM acts as a supervisor that provides additional support for the node's FT operations. The NRM will be internally duplicated and compared to present crash failure semantics and, although not being a single point of failure, it is advisable to replicate it to increase its availability.

3. Dynamic FT in FT4FTT

Following the work described in [2] we define *dynamic FT* as a system's ability to be *flexible* and *adaptive* from the point of view of its FT behaviours. *Flexible* fault-tolerant systems have different degrees of FT available at configuration time. An appropriate FT degree can be chosen depending on the specific system installation or depending on the applications the system executes. On the other hand, *adaptive* fault tolerant systems adapt the degrees of FT to changes during the runtime of an application.

3.1. Fault-tolerant tasks

We consider that an application is divided into tasks, where fault-tolerant tasks use active replication [10]. Voting on the outputs for each set of replicated tasks takes place at a new central element called *Node Dynamic Replication Manager* (NDRM), which replaces the NRM [8], as explained in Section 3.2. Depending on its criticality, a fault-tolerant task can require a higher or lower *degree of FT*, meaning the number of nodes it is executed upon is higher or lower. The specific tasks to be executed and their degree of FT can change during system operation and the NDRM will play a central role in managing these changes.

To have replicas of the tasks executing on different nodes they need to be distributed first. We propose two ways for tasks distribution. *a*) Software containing all

tasks possibly needed to run the application is kept only by the NDRM. The NDRM sends tasks as a whole through a communication channel and distributes them to the available nodes during normal operation. When there is a need to load a new set of tasks, that corresponding software is loaded only in the NDRM, which is the main advantage of this approach, the disadvantage being the bandwidth consumption and overhead of task distribution during normal operation. *b*) The software is distributed to all nodes by loading it into each of them during commissioning. During operation, the NDRM sends through the communication channel only the commands activating tasks already contained within each node. The advantage of this approach is its lower bandwidth consumption and overhead during operation. The main disadvantage is the need for all nodes to have enough memory for storing all the application software. The latter is our choice since a reduced overhead of task distribution during normal operation can be critical to guarantee real-time response.

3.2. NDRM features

The central component supporting dynamic FT is the NDRM. It orchestrates the execution of the application, which consists of tasks requiring different and changing degrees of FT. This goes beyond voting on the results of each FT task, and includes deciding and notifying the allocation of specific tasks or task replicas to the different nodes.

The *Mode Database* (MDB) keeps all the modes (see Section 4) of an application.

The NDRM includes the *Fault Tolerance Requirements Database* (FTRDB), which keeps all the information regarding FT degrees of different tasks. For each task, multiple degrees of FT can be defined. This means that the user can define the minimum number of replicas below which a task cannot be executed, and other desired FT degrees it would be desirable to have for that task.

The NDRM also hosts the *Resource Database* (RDB) that keeps track of the nodes' available resources, allowing for the NDRM to perform the allocation of tasks to nodes.

As with the NRM [8], the NDRM outputs are crucial to the normal system operation. If it fails and starts sending arbitrarily erroneous messages, it could cause a global failure. To avoid error propagation it is necessary to restrict the failure semantics of the NDRM such that it is only allowed to fail by crashing. For this, its circuits are duplicated and compared. On the other hand, the NDRM represents a single point of failure and therefore needs to be replicated. Both aspects will be discussed in later works.

4. System operation in the absence of faults: FT modes

To allow a fault-tolerant application to operate in different environments, or under different conditions, our system defines *FT modes* kept by the NDRM's MDB. Each mode defines tasks to be executed, their FT degrees and a traffic schedule for those tasks.

A change of the current FT mode could be triggered by the user or by the system itself when it senses environmental changes or detects other changes in the system. Examples of reasons for changing the FT mode are workload changes in a radar system [3], an airplane changing from taxing to flight mode [5], or a spacecraft entering a high-radiation area [4].

To trigger a mode change we propose two solutions. *a*) Nodes sensing the environment are in charge of triggering mode changes. This operation should be executed by multiple nodes for increased reliability, e.g. active replication could be used, and the NDRM votes on mode change requests sent by replicas. *b*) The NDRM directly senses the environment and triggers the changes. This is our prefered option since the NDRM can be trusted, i.e. it has restricted failure semantics and it will not be a single point of failure, and, unlike the previous solution, it doesn't consume scarce resources of the nodes. This solution could have the disadvantage of adding more responsibility to the NDRM, which can decrease its dependability since the software it executes becomes more complex and therefore less reliable.

When a mode change has to occur, in line with the work in [2], there are two solutions for the tasks currently in execution: *a*) Wait for all, or only the critical ones, to finish execution. *b*) Perform the change immediately not leaving the application in an undefined state. A complete description of how to deal with this problem is left as future work. However, it is obvious that both the system dependability and meeting the application deadlines must not be compromised during the mode change operation.

Primarily, when changing modes, schedulability analysis of tasks inside each node and of the corresponding messages in the channel is required. This will tell whether there is a specific allocation of tasks to nodes that is schedulable.

After the schedulability analyses have found a suitable allocation, each node needs to be notified of its allotted tasks. The NDRM is in charge of notifying the nodes and order the task activations at the right time.

4.1. Adaptivity within an FT mode

While an application runs in a specific mode, a task may require an increase of FT degree, i.e. an increase of the number of active replicas. This can be done by allocating new task replicas to nodes, if there are nodes with enough available resources and if the increase of traffic is schedulable. If not, tuning of existing task allocation and FT degrees would require consulting the RDB and FTRDB of the NDRM, respectively. Then, less critical tasks could be deallocated by decreasing the number of their active replicas and/or non-critical tasks could be aborted and deallocated, thus freeing resources. In any case, the number of active replicas cannot be decreased below the minimum FT degree defined in the FTRDB. Newly allocated replicas of tasks would have to be synchronized with the existing ones.

4.2. Algorithm for application execution

Application execution is divided in two phases: *initialization phase* and *normal operation phase*.

The initialization phase is described by the following

steps: *a*) The Software to be executed is loaded into all nodes and the NDRM. *b*) The MDB is populated based on specific application and its operating modes. *c*) The RDB is populated based upon connected nodes and their resources. *d*) An FT mode is selected, in this case the initial mode. *e*) The FTRDB is populated with the application tasks' FT requirements. This population is done depending on the selected mode. *f*) The schedulability analysis of node resources is performed, and a resource schedule is created. *g*) The schedulability analysis of traffic is done. The FTT master is in charge of creating a traffic schedule. The *System Requirements Database* (SRDB) contains all the information required for the traffic management. Before the schedule is created the FTT master goes thorough admission control and if this phase is completed, the SRDB is updated [9]. *h*) The FTT master creates a schedule based on information from the SRDB and that schedule is included in the trigger message. *i*) The NDRM notifies the nodes of the allocation of the tasks and waits for an acknowledgment from each one of the nodes. *j*) After the allocation notification is confirmed, a "start signal" is generated telling that application execution can be initiated. This "start signal" is part of the next TM. Also, the RDB will be updated since some of the nodes' resources are now utilized by the allocated tasks. *k*) After the TM with the "start signal" has been received by each node, execution starts at the time instant defined by the application.

Normal operation starts after the initialization phase. When in normal operation, the system can change its FT mode. *a*) The command of task deallocation is sent to all the nodes. *b*) The RDB is cleared. *c*) The rest of the algorithm continues from step "c)" of the initialization phase.

5. System operation in the presence of faults

The NDRM does the error detection. Active replication of the tasks and voting, besides being used to compensate errors in a minority of replicas, are also used for error detection. Comparing the voting results with the results of the replica allows detecting errors in a node that executes a replicated task. If some node does not execute replicated tasks, meaning the error cannot be detected by voting, the omission of frames coming from non-replicated tasks executed on that node is used for error detection.

There are three ways to deal with errors based on their seriousness: *recovery*, *reintegration* and *reconfiguration*. The *recovery* of a faulty task replica is done using the consensus value reached by the voting procedure of the NDRM. Each time an error is detected in a node, an error counter(ErC) in the NDRM is increased for that node. For each error-free task segment executed by the node, the ErC is decreased until it reaches it's initial value. The *reintegration* of a node is done when the node's ErC reaches a threshold. Node resets, reset counter(ReC), also kept by the NDRM, increases, a node's ErC is reset and reintegration information is sent by the NDRM in order to reintegrate the node. This information allows all the tasks belonging to a reseted node to be resynchronized. If the ReC reaches a predefined threshold, a node is permanently disconnected. *Reconfiguration* is done when some node fails permanently. All the tasks belonging to that node need to be reallocated to other nodes. The same procedure as described in Section 4.1 needs to be carried out.

6. Conclusions

We have explored how to design a system that can adapt both its real-time response and its level of fault tolerance to the changing requirements imposed by the environment and to the natural operation phases of the application. More specifically, we have taken as starting point a previous architecture that allows fault-tolerant operation on a distributed embedded system based on the FTT paradigm, and presented a first discussion on how to extend the functionality of the architecture to support dynamic fault tolerance.

There is plenty of future work to complete our architecture. This includes the refinement of the algorithms presented, the building of models of the system to simulate the resulting operation and to quantify the reliability values achieved. Moreover, it is also planned to solve some open problems, such as the replication of critical components such as the NDRM.

7. Acknowledgement

This work was supported by project DPI2011-22992 and grant BES-2012-052040 (Spanish *Ministerio de Economía y Competividad*), by FEDER funding, and by the Portuguese government through FCT grant Serv-CPS PTDC/EEA-AUT/122362/2010. Sinisa Derasevic was funded by a Erasmus Mundus EUROWEB scholarship.

References

[1] F. Cristian. Questions to ask when designing or attempting to understand a fault-tolerant distributed system. In *Proc. 3rd Brazilian Conference on Fault-Tolerant Computing*, Rio de Janeiro, Brazil, 1989.

[2] J. Goldberg, I. Greenberg, and T. Lawrence. Adaptive fault tolerance. In *Advances in Parallel and Distributed Systems, Proc. of the IEEE Workshop on*, pages 127–132, 1993.

[3] O. González, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Real-Time Systems Symp., Proc., The 18th IEEE*, pages 79–89, 1997.

[4] M. Hecht, H. Hecht, and E. Shokri. In proc. of ieee adaptive fault tolerance for spacecraft. In *Aerospace Conference Proceedings*, pages 521–533 vol.5, 2000.

[5] H. Kopetz, R. Nossal, R. Hexel, A. Krüger, D. Millinger, R. Pallierer, C. Temple, and M. Krug. Mode handling in the time-triggered architecture. *Control Engineering Practice*, 6(1):61–66, 1998.

[6] J.-C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Wien New York, 1992.

[7] P. Pedreiras and L. Almeida. The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems. In *Proc. Int. Parallel and Distributed Processing Symposium*. IEEE Comput. Soc.

[8] J. Proenza, M. Barranco, J. Llodra, and L. Almeida. Using FTT and stars to simplify node replication in CAN-based systems. In *Emerging Tech. Factory Automation (ETFA), IEEE 17th Conf. on*, pages 1–4, 2012.

[9] R. Santos. *Enhanced Ethernet Switching Tecnology for Adaptibe Hard Real-Time Applications*. PhD thesis, Universidade de Aveiro, 2010.

[10] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, Proc. 20th International Conf. on*, pages 464–474, 2000.