Universitat
de les Illes Balears

# First Description of a Self-Reconfigurable Infrastructure for Critical Adaptive Distributed Embedded Systems

A. Ballesteros, J. Proenza, M. Barranco, L. Almeida, P. Palmer

Departament
de Ciències Matemàtiques
i Informàtica

TECHNICAL REPORT
May/2019
A-03-2019

# First Description of a Self-Reconfigurable Infrastructure for Critical Adaptive Distributed Embedded Systems

Alberto Ballesteros
DMI, Universitat Illes Balears
Palma de Mallorca, Spain
a.ballesteros@uib.es

Julián Proenza
DMI, Universitat Illes Balears
Palma de Mallorca, Spain
julian.proenza@uib.es

Manuel Barranco
DMI, Universitat Illes Balears
Palma de Mallorca, Spain
manuel.barranco@uib.es

Luís Almeida
CISTER / Instituto de Telecomunicações
Universidade de Porto, Portugal
lda@fe.up.pt

Pere Palmer
DMI, Universitat Illes Balears
Palma de Mallorca, Spain
pere.palmer@uib.es

## ABSTRACT

Adaptive systems, apart from fulfilling some functional requirements, can modify their behaviour autonomously and dynamically to cope with new operational requirements or conditions. The DFT4FTT project aims at providing a self-reconfigurable infrastructure that can support distributed applications with real-time, reliability and adaptivity requirements. This paper describes the architecture and the set of mechanisms that make it possible to: monitor the environment and the system itself, decide when a new configuration is needed, decide on a new valid configuration and apply said configuration. Finally, note that this self-reconfiguration capabilities are not only interesting from a functional perspective, but from a reliability perspective as new we can implement dynamic fault-tolerance mechanisms. In this regard, DFT4FTT implements a N-Modular Redundancy scheme with spares, that increases the system reliability if it is used from the very beginning of the mission.

## 1. INTRODUCTION

An Adaptive Distributed Embedded System (ADES) is a type of Distributed Embedded System (DES) that has the ability to modify its behaviour autonomously and dynamically in response to changing operational requirements or conditions. Some examples of potential applications of these systems are: autonomous vehicles, exploration vehicles, machinery in a smart factory and self-repairable devices.

In a distributed system the ability to adapt is an interesting feature from the point of view of the functionality, efficiency and dependability. First, ADESs can dynamically change their behaviour to cope with new operational requirements. Second, ADESs can dynamically reserve the necessary computational and communication resources at every instant, thus, avoiding the need to dimension the system for the worst case scenario. Third, ADESs can dynamically restore themselves by resetting faulty tasks, or even reallocating them, when node in which they execute fails.

To properly implement an ADES it must be provided with the appropriate architecture and mechanisms, that ensure that real-time, dependability and adaptivity requirements are fulfilled. The DFT4FTT (Dynamic Fault Tolerance for Flexible Time-Triggered Ethernet) project [6] proposes a complete infrastructure that addresses all these requirements both at the *network* and *node* level.

At the network level, DFT4FTT relies on FTTRS (Flexible Time-Triggered Replicated Star) [4], a switched-Ethernet implementation of the Flexible Time-Triggered (FTT) communication paradigm. FTT provides full flexibility in the communications. This means, on the one hand, that it supports the exchange of periodic and aperiodic traffics with different real-time requirements. On the other hand, it allows dynamically changing the real-time requirements of the traffic. High reliability is achieved by means of fault tolerance. Specifically, the network is replicated to tolerate permanent hardware faults, while messages are proactively retransmitted to tolerate transient faults. Finally, we are currently working in adding dynamic fault-tolerance capabilities to FTTRS, so that the number of proactive retransmissions is autonomously and dynamically modified in response to changing Bit Error Rates (BERs), thus, making an efficient usage of the communication resources.

At the node level, DFT4FTT proposes a centralized approach in which the so-called *Node Manager* (NM) decides autonomously and dynamically how to allocate the tasks into the available nodes of the ADES. A given allocation of tasks and messages, together with their operational attribute, such as periods or deadlines, is called *configuration* of the system. The NM guarantees that the system configuration fulfils the real-time and reliability requirements of the tasks. The former is achieved by performing a schedulability analysis, both in nodes and in the communications, while the second is achieved by means of active replication with majority voting. Specifically, each critical task is executed in parallel in several nodes and task replicas periodically vote to obtain a consensus result. Finally, note that the ability to change on-line the configuration is not only interesting from a functional, but from a reliability perspective. For instance, if a task replica fails we can restore the level of replication, thus, increasing the system reliability.

In this paper we describe the on-going work we are carrying out to design DFT4FTT, the first self-reconfigurable FTT-based infrastructure, and the set of mechanisms it implements to: monitor the environment and the system itself; decide when a new configuration is needed; decide on a new valid configuration; and apply said configuration. Additionally, we describe how the these self-reconfiguration capabilities enable the use of a N-Modular Redundancy scheme with spares, and how it increases the system reliability.

## 2. TASK MODEL

The time during which a system has to operate, or *mission time*, can be divided into various *phases*. Each phase defines the sub-objectives that have to be met, as well as the operation conditions under which they have to be met. Consequently, every phase imposes different functional and non-functional requirements to the system.

Functional requirements are fulfilled thanks to the execution of *functionalities*. Some examples of car functionalities are: the throttle control, the climate control or the infotainment. Each functionality is implemented by an *application* which, in turn, is composed by a set of distributed and interconnected *tasks* that are executed in a sequential and/or parallel manner. For instance, as can be seen in Fig. 1, a basic replicated control application could be composed of one task for consulting the value of a sensor; a triplicated task for determining the actuation value from the sensor value; and a third task for voting on the replicated actuation values and, then, performing the consensus actuation. Note that the tasks communicate among them by exchanging messages through the network. Thus, an application can be seen as a sequence of task executions and message transmissions.

With what regards to the non-functional requirements, functionalities can have different real-time and reliability requirements in each of the phases of the mission.

The real-time requirements of the functionalities are fulfilled thanks to the work presented in [2], which makes it possible to determine, for a given application, a sequence of task executions and message transmissions that allows them to meet their deadlines. Moreover, this is done in a centralized, on-line and holistic manner. In the context of our problem, as shown in Exps. 1, 2 and 3, this means that, by providing the algorithm $f$ with the basic real-time attributes of the tasks ($C_i$, $T_i$ and $D_i$, from $T$), the transmission time of the messages ($C_j$, from $M$) and the dependencies between tasks and messages ($MP_i$, $MC_i$, $PT_i$ and $CTL_i$, from $T$ and $M$), we can obtain the triggering instant of each task ($Ph_i$), as well as the period, triggering instant, and deadline of each message ($T_j$, $Ph_j$ and $D_j$). Nevertheless, note that this work does not consider the distribution of tasks. Consequently, we will extend it with additional schedulers to ensure that, both, the computational and communication resources in each of the nodes and links are enough to meet the deadlines of all the tasks and messages in execution.

To ensure that the reliability requirements of the functionalities are fulfilled, we use task and message replication. Specifically, critical tasks are executed redundantly and in parallel in different nodes, while the messages of said tasks are pro-actively sent several times. With this we are able to tolerate permanent and temporary faults affecting either the nodes or the communications. Note that the work in [2] seamlessly supports all these mechanisms as it allows to execute multiple tasks in parallel and the additional time needed to transmit the message replicas can be modelled by properly increasing the worst case transmission time.

$$T = \{t_i(C_i, T_i, D_i, MP_i, MC_i), i = 1..t\} \quad (1)$$

$$M = \{m_j(C_j, PT_j, CTL_{j,i}), j = 1..m, i = 1..t\} \quad (2)$$

$$\{Ph_i, i = 1..t\} \cup \{(T_j, Ph_j, D_j), j = 1..m\} = f(T, M) \quad (3)$$

$t$: Number of tasks — $m$: Number of messages
$C_i$: Worst case exec. time — $C_j$: Worst case tx time
$T_i, T_j; Ph_i, Ph_j; D_i, D_j$: Periods, Phases and Deadlines
$MP_i$: Message produced — $MC_i$: Message consumed
$PT_j$: Producer task — $CTL_{j,i}$: Consumer task list

## 3. SYSTEM ARCHITECTURE

The DFT4FTT architecture is composed of various hardware components, as can be seen in Fig. 2: an FTTRS network, several sensors and actuators, several computational nodes and a *Node Manager*.

The central point of the architecture is the communication network, which enables the real-time communication among all the other components. Moreover, it also provides the necessary flexibility services and reliability services. As explained in the introduction, in the current design, FTTRS is selected as the underlying communication subsystem.

The Computational Nodes (CNs) are the components responsible for the execution the tasks. However, CNs do not decide which tasks do they execute, as will be explained later, it is the Node Manager the one that determines on-line which tasks have to be executed in each of the CNs.

The sensors and the actuators (SAs) are the components responsible for interacting with the environment. Note that, contrary to many DES where SAs are connected to the nodes doing the processing, in the DFT4FTT architecture they are connected directly to the network. By doing this, SAs are independent from the CNs, which eases the allocation of and makes the architecture more fault tolerant. On the one hand, there is no restriction when allocating the tasks into the CNs, since SAs can be accessed from any CN. On the other hand, SAs are failure independent from the CNs.

The Node Manager (NM) is the component responsible for dynamically allocating the tasks into the CNs. The NM constantly monitors the environment and the system itself, identifies the situations where a new configuration is needed, decides which tasks have to be started and/or stopped in each of the CNs and carries out said actions. All this operation will be further described in the next sections. Finally, note that, although the NM is represented as an independent physical component, it is desirable that it is integrated inside the network, specifically, inside a switch, similarly as we have done previously for other software components [5]. By doing this the NM can take advantage of its privileged position to better monitor the system.

As concerns the software architecture of the NM and the CNs, Fig. 3 shows their internals. As can be seen, at the lowest level the *Communication Enabler* allows to interface with the network. Above that, several modules give support to the monitoring and configuration change processes. The services provided by these modules can be accessed by means of the *Task Allocation Scheme (TAS) Service Interface*. Finally, at the top, we find the application layer, where the *Knowledge Entity* in the NM and the tasks in the CNs rely on the TAS Service Interface to instruct the changes in the configuration of the system, when the requirements are not fulfilled. It is noteworthy that, although the NM must be fed, among other things, with the list of applications and their operational attributes to properly operate, none module in the NM depends on the applications.

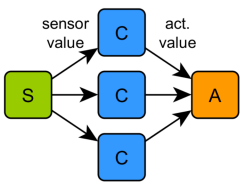


Figure 1: App example.

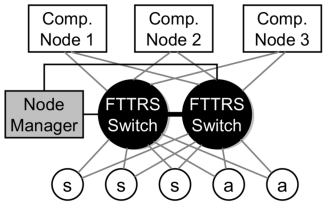


Figure 2: System architecture.

# 4. MONITORING PROCESS

The first step in the self-reconfiguration process is monitoring the environment and the system itself to obtain the *system state*. The system state is constituted by: the *status of the architecture*, i.e., the list of hardware components, how they are interconnected among them and the fault state of said components and their links; the *failure/bit error rates*, i.e., the quantity of errors per time unit that each hardware component and link is suffering; the *status of the execution*, i.e., the list of tasks executed in each of the CNs and the fault state of said tasks; and the *status of the resources*, i.e., the amount of computational and communication resources being used in each of the CNs and links.

The *Monitoring Manager* (MM) in the NM is the component responsible for obtaining the system state. Note that we assume that CNs can fail in an uncontrollable manner and, thus, the information they can provide about their status is not reliable. Instead, we infer their status from their outputs, i.e., the messages they generate. Next we explain which data is gathered and and how it is processed.

To obtain the *status of the architecture* the MM is fed with the initial architecture of the system, which is then updated upon the detection of any hardware fault. Note that the status of CNs and SAs, as well as of their links can be gathered from the *Port Guardians* (PGs) of the FTTRS switches [1]. The main purpose of the PGs is to filter some types of communication errors so that they cannot propagate to the other network components. However, in DFT4FTT PGs are extended to include diagnostic features.

The *failure rate* of the hardware components can change dynamically depending on the operational conditions. However, it can be determined by providing the MM with a failure rate model of said components, like the one proposed in the MIL-HDBK-217 handbook [3] and, then, by feeding this model with the attributes of the environment at each instant. The attributes of the environment can be obtained from the sensors already available in the system, or by placing new ones. Similarly, the *bit error rate* of the links can also be determined by sensing the environment. Moreover, we can also infer the bit error rate by consulting the PGs, which keep track of the messages that are lost.

As concerns the *status of the execution*, we force tasks to periodically send an *I am alive message* so that the MM can determine which applications are running in each of the CNs. Moreover, the MM can also detect if a task replica is producing wrong results, i.e., it is faulty. For this, the MM collects the results from the replicas, vote on them and determine if any result deviates significantly. Note that, for diagnostic purposes, tasks send their messages to the network, even if the receiving task is in the same node.

To determine the *status of the resources*, each application specifies in advance the maximum amount of computational and communication resources that are required to execute each of its associated tasks. With this information and the distribution of tasks, the MM can infer the amount of resources used in every CN and its links.

Finally, regarding the time needed to obtain the system status, the time elapsed between the occurrence of a relevant event, in the environment or in the system itself, to the detection of said event is not negligible. However, for the events we can detect, we know how do they propagate and, thus, we can specify an upper bound for this elapsed time.

# 5. DECISION PROCESS

As introduced previously, the modules responsible for taking the decisions on when and how to switch to a new configuration are the *Knowledge Entity* (KE) in the NM and the tasks in the CNs. On the one hand, the KE performs automatic configuration changes, i.e., it constantly checks that the *system state* fulfils the *system requirements* and, if not, it forces a new configuration. On the other hand, tasks can also request for changes by modifying the *system requirements*, which will then be applied by the KE. Next we describe in more detail what are the system requirements, as well as how the KE and the tasks take all these decisions.

## 5.1 System Requirements

The system requirements are the list of applications, together with their real-time and reliability requirements, that have to be executed. This list is divided in two parts: the sub-list of applications related to the phase of the mission and the one related to the on-demand functionalities.

The phase-related applications are those indispensable applications needed to fulfil the functional requirements of a given phase of the mission. Each phase starts as a result of the fulfilment of a specific condition. For instance, in a commercial flight, when the plane reaches a certain altitude it is considered that the *climb* phase has finished and that the *cruise* phase has started. The KE is the one that determines when a new phase starts, by inspecting the system state, and by updating the system requirements accordingly.

The on-demand-related applications are those indispensable and non-indispensable applications that are started as a result of a new functional requirement not related to the phase of the mission. For instance, the application responsible for the infotainment system of a vehicle can start an additional application to provide multimedia service to a passenger requesting it. Another example are those critical applications started to support some overruling command send by the driver of said vehicle. In both cases the request is issued by the tasks in th CNs.

## 5.2 Knowledge Entity

As already said, the decision process in the KE has two steps. First, the KE constantly checks that the system state fulfils the system requirements. For this, the KE extracts from the system state the set of applications that are being executed and compares it against the list of tasks from the system requirements. Moreover, the KE also determines if the real-time and reliability requirements are fulfilled. While real-time requirement violations can be easily detected thanks to the PGs, to determine if the reliability requirements of an application are being violated, it is necessary to determine the reliability of each of its associated tasks. This implies compiling and processing the operational attributes of the tasks: the fault state and the failure rate of the nodes executing the tasks, the bit error rate and the number of proactive retransmissions of the links conveying data from the tasks and the number of replicas for each task.

If the KE determines that the system requirements are not fulfilled, then, it comes the second step, where the KE decides on the new configuration to apply. Finding a new configuration that meets all the system requirements can take a significant amount of time due to the number of parameters involved. Consequently, we propose a three-stage search approach. First, the KE provides, as soon as possi-

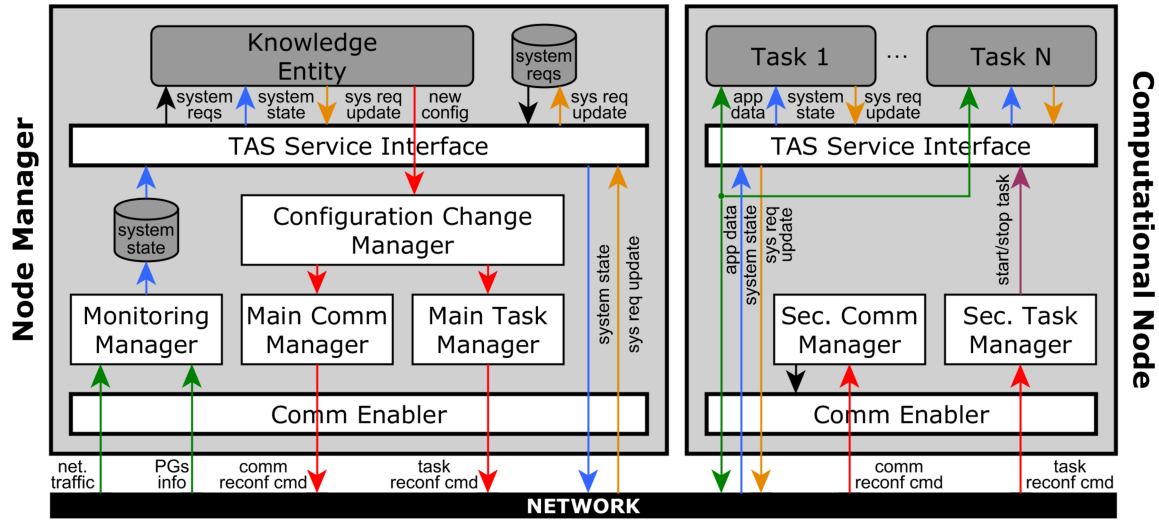


Figure 3: Internals of the Node manager and a Computational Node.

ble, a good new configuration for the critical applications, i.e., a configuration that meets all the system requirements of the critical applications. Second, we do the same but for non-critical applications. Third, the KE provides a better new configuration, i.e., a configuration that not only meets all the system requirements but that is optimal, according to some specific policy(ies), while the system is running. By doing this, the service is restored as soon as possible, taking into account the reliability requirements, and, after some time, the system can operate in a more optimal manner.

To find a new configuration, in any of the three search stages, we use the *branch and bound* technique together with a *greedy algorithm*. The root of the search tree is the current configuration and each branch represents a change, like starting a task in a CN, thus generating new configurations. After that, we use the greedy algorithm to determine which new configuration use for the next iteration. For this, it includes a heuristic that selects the best promising branch. This procedure is repeated until a valid configuration is found. Specifically, that it fulfils the functional requirements, i.e., contains the list of tasks that have to be executed, and that it fulfils the non-functional requirements, i.e., fulfils the real-time and reliability requirements.

Regarding the validation of the non-functional requirements, we follow the algorithm depicted in Fig. 4. This algorithms checks in parallel if the configuration fulfils the real-time requirements (top) and the reliability requirements (bottom). On the one hand, we obtain the triggering parameters as described in Sec. 2, which is then used by two on-line schedulability analysis algorithms that check if the computational and the communication resources are enough to meet all the deadlines. On the other hand, we carry out an on-line reliability analysis to check that the level of reliability is the one required for the critical tasks. Note that, if any of these validations fail, feedback can be obtained to improve the generation of configurations and the branch selection.

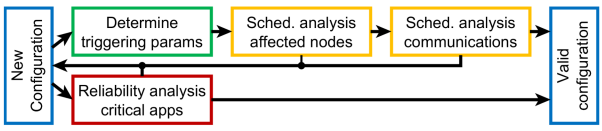


Figure 4: Decision process.

Finally, the third search stage allows to optimize the configuration for some specific policy(ies), for instance: energy consumption, reliability, performance of the network or QoS and QoC. System designers decide which are the relevant policies for the system, and their level of importance by giving a weight to each of them. Additionally, we provide a function that scores configurations, according to the selection of policies and their weights. The higher the score, the better the configuration. This function is then used together with the searching algorithm to find a better configuration.

## 5.3 Tasks

The tasks executed in the CNs carry out the operation related to the purpose of the system, i.e., they contain the semantics of the applications. Consequently, they are the only system modules that know what are the dynamic operational requirements derived from the human commands or the tasks themselves. Specifically, system designers can create specific tasks that request system changes when a user sends a command through an input device or when the tasks themselves decide that it is necessary. The available requests that a task is allowed to issue are to start and stop applications, as well as to modify the real-time and reliability requirements of the already running applications (both phase-related and on-demand). Note, in this regard, that modify the phase-related applications allow to do overruling. Finally, as explained previously, these requests are transformed into *system requirement updates*, i.e., these requests are enforced modifying the system requirements directly.

These available requests are very powerful as they allow to completely modify the operation of the system. However, since we consider that CNs can fail in an uncontrollable manner, requests could contain wrong information which would result in a wrong system modification and, thus, grim for the system. To solve this issue we restrict the scope of the task requests, i.e., requests affecting critical applications can only be issued in a reliable manner. For this, we propose two different approaches. On the one hand, critical requests can be issued by specific highly-reliable CNs. On the one hand, when a highly-reliable CN cannot be provided, the task issuing the critical request must be triplicated and said request must be agreed among the three replicas.

## 6. CONFIGURATION CHANGE PROCESS

As already explained, when the KE determines that a configuration change is required, due to a phase change or a task request, a new configuration is proposed. This new configuration is then delivered to the *Configuration Change Manager* (CCM) which orchestrates the low-level changes. To carry out the changes related to the communications, the CCM relies on the *Main Communication Manager*. This module sends communication reconfiguration commands to the *Secondary Communication Manager* in the CNs to create, remove, or modify the attributes of the communications. Similarly, the CCM relies on the *Main Task Manager* to carry out the changes related to the tasks. Specifically, this module sends task reconfiguration commands to the *Secondary Task Manager* in the CNs to start and stop tasks.

The Configuration Change process starts by liberating all the computational and communication resources that are no longer required, according to the new configuration. It is noteworthy that this is a critical procedure since stopping tasks abruptly can leave the system in a unsafe state. Two aspects to consider in this regard are the false errors that can be detected and the state of the actuators.

On the one hand, stopping tasks without any specific order can provoke scenarios that can be interpreted as errors by the Monitoring Manager. For instance, if the communication resources are removed before the associated tasks are stopped, it will happen that a task will try to use these resources provoking, thus, an error, although this task is no longer needed. Consequently, the tasks and their communication resources are stopped taking into account their interdependencies and, in some cases, specific monitoring features are disabled for the affected applications.

On the other hand, stopping tasks without any knowledge about the semantics of the applications can cause the state of the associated actuators to be wrong. For instance, when a semi-automatic vehicle switches from automatic to manual mode, it can be necessary to leave some of the actuators in a specific state, so that the manual mode starts properly. The state in which the actuators associated to an application have to be left, in order to perform a safe termination, we call it *termination condition*. In some applications the termination condition consists in leaving the associated actuators in a predefined state, while in others consists in finishing the application cycle, so that the state of the associated actuators is the last one as calculated.

When all the computational and communication resources that are no longer required have been released, the CCM instructs the reservation of the new resources. To deal with the dependency issues, this is done in the opposite order of the liberation. First, the new communications are created and, then, the associated tasks are started. Note that neither the tasks nor the communications start to operate right after being created. As explained in Sec. 2, there is a triggering plan that has to be fulfilled. More precisely, when all the computational and communication resources have been reserved, the NM triggers the execution of the tasks and the transmission of messages in the appropriate order.

Finally, as concerns the time needed to change the configuration, note that the set of steps required to change from one configuration to another can be determined in advance. Moreover, we can determine the time of each individual change. Consequently, we can establish an upper bound for the time necessary to change the configuration.

## 7. RECONFIGURATION FOR RELIABILITY

The self-reconfiguration capabilities of this infrastructure make it possible to change the set of applications being executed in the system in response to changes in the system state or in the system requirements. However, self-reconfiguration is also interesting in the scope of the reliability of the tasks. On the one hand, tasks can be dynamically started, which avoids the need to overprovision resources according to the worst-case assumptions. This is particularly interesting for critical tasks which are typically triplicated and, thus, require three times the amount of resources. On the other hand, the NM manager automatically restarts faulty tasks. For instance, this is useful when a tasks suffers a temporary error that corrupts its internal state, thus, preventing it from operating correctly from now on.

Another scenario in which this feature is useful, is when a task cannot continue its execution due to a permanent failure affecting the CN in which it is being executed and, thus, it is reallocated. For non-critical (non-replicated) tasks, this means that we can continue delivering the service after some downtime, the reconfiguration time. For critical (replicated) tasks, this means that we have *redundancy preservation*. Specifically, we are able to implement a N-Modular Redundancy (NMR) scheme with several spares. In this scheme each critical task is replicated, typically triplicated, and executed in a different CN. If any of the task replicas fails, the NM can automatically start a new replica for substituting it. Moreover, since we have *software spares* we can continuously allocate new replicas on-line, as long as there are enough communication and computational resources.

Regarding the time needed to restore the level of replication of a critical task, note that we provide cold spares, i.e., the procedure is not instantaneous. We have to reserve the required communication and computational resources, provide the new replica with an updated internal state of the task and, then, trigger its execution at the right instant.

In Fig. 5 we show the level of reliability that we can obtain if we use from the very beginning of the mission Triple Modular Redundancy (TMR) with 0, 1, 2 and 3 spares. These values were calculated using a Stochastic Activity Network that models a basic application with three task replicas, each of which having a failure rate of $1e-5$ per hour. When a replica fails it is automatically substituted by one of the spares, which also have a failure rate of $1e-5$ per hour although they are idle. Note, however, that this model contains various simplifications. On the one hand, we consider that we can always detect the failure of a task replica. On the other hand, we consider that the detection and the reconfiguration time is negligible, and this is not true.

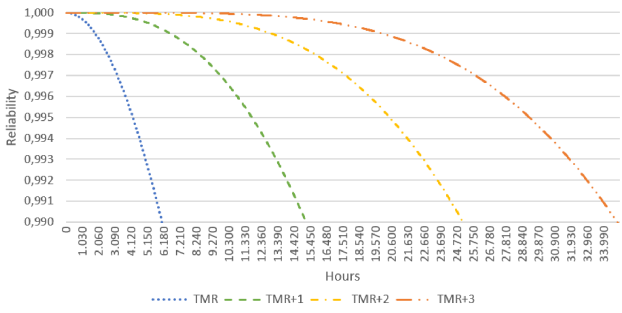


Figure 5: Reliability analysis.

## 8. CONCLUSIONS

In this paper we describe the on-going work we are carrying out to construct a self-reconfigurable infrastructure for systems with real-time, reliability and adaptivity requirements. At the network level, this infrastructure relies on FTTRS, a network that provides the necessary communication services to fulfil all these requirements. Specifically, FTTRS not only provides a means to exchange real-time traffic, while it allows to change on-line the attributes of the communications. Moreover, it provides high-reliability by means of channel and message replication.

At the node level, this infrastructure allows to execute the functionalities of the system. Each these functionalities has specific real-time and reliability requirements and is implemented by means of an application, which is composed of tasks executed in a sequential and/or parallel manner. We have designed a set of mechanisms that make it possible to dynamically allocate tasks in the nodes performing the computation. A given allocation of tasks and messages, together with their operational attributes, is called configuration, and a change in the configuration can be triggered by a change in the system requirements or in the system state.

To ensure that the real-time requirements of the functionalities are always fulfilled, every time a new configuration is needed, the system searches for a configuration that makes all the tasks and messages meet their deadlines. As concerns the reliability requirements, critical tasks are replicated using the N-Modular Redundancy with spares technique. Thus, when searching for the new configuration, we ensure that critical tasks have enough replicas.

## 9. FUTURE WORK

One of the most important aspects to address as future work is the single point of failure that the Node Manager represents. To solve this issue we plan to duplicate the Node Manager and introduce mechanisms to make both replicas replica determinate, so that they can operate in parallel and seamlessly tolerate the failure of one of them.

Another aspect we are working on is the characterization of the self-reconfiguration time. This is a very important system attribute, from a real-time perspective, since it determines how fast the the system can react to changes. As explained, the self-reconfiguration is done in three steps: detect the need for a new configuration, determine a new valid configuration and apply said configuration. While the first and the third steps are deterministic and, thus, we can find an upper bound, we do not have yet any mechanism to determine how much time it takes find a valid configuration using the searching algorithm proposed.

## 10. ACKNOWLEDGMENTS

This work was supported by project TEC2015-70313-R (Spanish *Ministerio de economía y competividad*), by FEDER funding and by the Portuguese Government through FCT grant UID/EEA/50008/2013 - Instituto de Telecomunicações.

## 11. REFERENCES

[1] A. Ballesteros, D. Gessner, J. Proenza, M. Barranco, and P. Pedreiras. Towards preventing error propagation in a real-time Ethernet switch. In *Proc. 18th IEEE Int. Conf. on Emerging Tech. and Factory Autom. (ETFA)*, Cagliary, 2013.

[2] M. J. B. Calha. *A Holistic Approach Towards Flexible Distributed Systems*. PhD thesis, Universidade de Aveiro, 2006.

[3] DOD. *MIL-HDK-217F-2 Military Handbook, Reliability Prediction Of Electronic Equipment*. Department of Defense Washington DC, 1995.

[4] D. Gessner. *Adding Fault Tolerance To a Flexible Real-Time Ethernet Network for Embedded Systems*. PhD thesis, University of the Balearic Islands, 2017.

[5] D. Gessner, J. Proenza, M. Barranco, and L. Almeida. Towards a Flexible Time-Triggered Replicated Star for Ethernet. In *Proc. 18th IEEE Int. Conf. on Emerging Tech. and Factory Autom. (ETFA)*, Cagliary, 2013.

[6] J. Proenza, M. Barranco, A. Ballesteros, I. Álvarez, D. Gessner, S. Derasevic, and G. Rodríguez-Navas. DFT4FTT Project.