Universitat
de les Illes Balears

DOCTORAL THESIS
2018

NODE FAULT TOLERANCE FOR DISTRIBUTED EMBEDDED SYSTEMS BASED ON FTT-ETHERNET

Sinisa Derasevic

Universitat
de les Illes Balears

DOCTORAL THESIS
2018

Doctoral Programme of Information and Communications Technology

---

NODE FAULT TOLERANCE FOR DISTRIBUTED EMBEDDED SYSTEMS BASED ON FTT-ETHERNET

Sinisa Derasevic

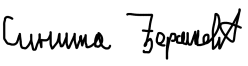
Thesis Supervisor: Dr. Manuel Barranco
Thesis Supervisor: Dr. Julián Proenza
Thesis Tutor: Dr. Manuel Barranco

Doctor by the Universitat de les Illes Balears

Declaration of Authorship

I, Sinisa Derasevic, declare that this thesis titled, "Node fault tolerance for distributed embedded systems based on FTT-Ethernet" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a doctorate degree in *Tecnologies de la Informació i les Comunicacions* at *Universitat de les Illes Balears*.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 05.10.2018

Supervisors' Agreement

Manuel Barranco Ph.D. in Computer Science and *Profesor Contratado Doctor* at the *Department of Mathematics and Computer Science, Universitat de les Illes Balears* and Julián Proenza, Ph.D. in Computer Science and *Profesor Titular de Universidad* at the same department.

DECLARE

that the thesis titled "Node fault tolerance for distributed embedded systems based on FTT-Ethernet", presented by Sinisa Derasevic, to obtain the degree of *Doctor en Tecnologías de la Información y las Comunicaciones*, has been completed under our supervision and meets the requirements to opt for a Doctorate.

For all intents and purposes, we hereby sign this document.

Signed: _____

Date: 05.10.2018

Universitat de les Illes Balears

Abstract

Doctor of Philosophy

Node fault tolerance for distributed embedded systems based on FTT-Ethernet

by Sinisa Derasevic

Distributed embedded systems are systems composed of a set of interconnected nodes working towards achieving some common goal and that form a part of a larger mechanical or electrical system. Nodes are usually interconnected by means of a communication network.

As regards communication networks, in the recent decades Ethernet has become one of the most popular technologies due to its many advantages such as simplicity, ever increasing bandwidth, and inexpensiveness, among others.

When distributed embedded systems form a part of larger systems that execute critical applications, there is often a need to provide a support for both real-time response requirements and for the achievement of a very high reliability. Ethernet's original technology does not provide any such support.

Therefore, in this dissertation we make use of the recently proposed *Flexible Time-Triggered Replicated Star* (FTTRS) communication subsystem as a means of interconnecting the nodes of distributed embedded systems executing critical applications. FTTRS takes Ethernet networking technology as basis and then additionally provides mechanisms to support both real-time response and high reliability. Real-time response is provided by the use of the *Flexible Time-Triggered* (FTT) communication paradigm implemented on top of the Ethernet protocol that, besides the provision of real-time guarantees, also supports flexibility, more specifically, the ability to modify the network behaviour at runtime while maintaining the established real-time guarantees. The high reliability in FTTRS is achieved by mechanisms that deal with the faults that could affect the communication among nodes.

However, providing fault tolerance to the communication subsystem only is not enough to satisfy the most demanding reliability requirements of critical applications. In order to attain high levels of reliability, faults in the nodes of the distributed embedded system must also be dealt with.

Consequently, we have designed various fault tolerance mechanisms to deal with faults affecting the correct functioning of the nodes. These mechanisms take advantage of the characteristics of the FTTRS communication subsystem and of the underlying FTT communication paradigm.

Concluding, in this thesis we will see how we can, with the addition of specific mechanism for tolerating the faults of the nodes of a distributed embedded system based on FTTRS, achieve very high levels of reliability for the system as a whole. In addition to designing fault tolerance mechanisms addressing the nodes' faults, we will also show how the achieved reliability can be assessed, and we will establish what is the obtained benefit by comparing said reliability with that of a non fault-tolerant version of the same system.

Summary in Spanish

Los sistemas empotrados distribuidos son sistemas compuestos por un conjunto de nodos interconectados que trabajan para lograr un objetivo común y que forman parte de un sistema mecánico o eléctrico más grande. Los nodos suelen estar interconectados por medio de una red de comunicación.

En cuanto a las redes de comunicación, en las últimas décadas Ethernet se ha convertido en una de las tecnologías más populares debido a sus muchas ventajas tales como simplicidad, anchos de banda siempre crecientes y bajo coste, entre otras.

Cuando los sistemas empotrados distribuidos forman parte de sistemas más grandes que ejecutan aplicaciones críticas, a menudo existe la necesidad de proporcionar un soporte para requisitos de respuesta en tiempo real y para la consecución de una muy elevada fiabilidad. La tecnología original de Ethernet no proporciona ningún soporte de este tipo.

Por lo tanto, en esta disertación usamos el recientemente propuesto subsistema de comunicación que recibe el nombre de *Flexible Time-Triggered Replicated Star* (FTTRS) como medio para interconectar los nodos de los sistemas empotrados distribuidos que ejecutan aplicaciones críticas. FTTRS toma la tecnología de red Ethernet como base y sobre ella proporciona mecanismos para soportar respuesta en tiempo real y elevada fiabilidad. La respuesta en tiempo real es proporcionada por el uso del paradigma de comunicación *Flexible Time-Triggered* (FTT) implementado sobre el protocolo Ethernet el cual, además de la provisión de garantías de tiempo real, también proporciona flexibilidad, en concreto, la capacidad de modificar el comportamiento de la red en tiempo de ejecución mientras se mantienen las garantías de tiempo real comprometidas. La elevada fiabilidad en FTTRS se logra mediante mecanismos que toleran los fallos que podrían afectar a la comunicación entre nodos.

Sin embargo, proporcionar tolerancia a fallos únicamente al subsistema de comunicación no es suficiente para satisfacer los requisitos de fiabilidad más exigentes de las aplicaciones críticas. Para alcanzar altos niveles de fiabilidad, los fallos en los propios nodos del sistema empotrado distribuido también deben ser tratados.

En consecuencia, hemos diseñado varios mecanismos de tolerancia a fallos para tratar los fallos que puedan afectar al correcto funcionamiento de los nodos. Estos mecanismos aprovechan las características del subsistema de comunicación FTTRS y del paradigma de comunicación FTT subyacente.

Concluyendo, en esta tesis veremos cómo podemos, con la introducción de mecanismos específicos para tolerar los fallos de los nodos de un sistema empotrado distribuido basado en FTTRS, lograr muy elevados niveles de fiabilidad para el sistema en su conjunto. Además del diseño de los mecanismos de tolerancia a fallos de los nodos, también mostraremos cómo se puede evaluar la fiabilidad resultante y estableceremos cuál es el beneficio obtenido, comparando dicha fiabilidad con la de una versión no tolerante a fallos del mismo sistema.

Summary in Catalan

Els sistemes encastats distribuïts són sistemes composts per un conjunt de nodes interconnectats que treballen per aconseguir un objectiu comú i que formen part d'un sistema mecànic o elèctric més gran. Els nodes solen estar interconnectats mitjançant una xarxa de comunicació.

Quant a les xarxes de comunicació, en les últimes dècades Ethernet s'ha convertit en una de les tecnologies més populars a causa dels seus molts avantatges tals com a simplicitat, amples de banda sempre creixents i baix cost, entre d'altres.

Quan els sistemes encastats distribuïts formen part de sistemes més grans que executen aplicacions crítiques, sovint existeix la necessitat de proporcionar un suport per a requisits de resposta en temps real i per a la consecució d'una molt elevada fiabilitat. La tecnologia original d'Ethernet no proporciona cap suport d'aquest tipus.

Per tant, en aquesta dissertació usem el recentment proposat subsistema de comunicació que rep el nom de *Flexible Time-Triggered Replicated Star* (FTTRS) com a mitjà per interconnectar els nodes dels sistemes encastats distribuïts que executen aplicacions crítiques. FTTRS pren la tecnologia de xarxa Ethernet com a base i sobre ella proporciona mecanismes per suportar resposta en temps real i elevada fiabilitat. La resposta en temps real és proporcionada per l'ús del paradigma de comunicació *Flexible Time-Triggered* (FTT) implementat sobre el protocol Ethernet el qual, a més de la provisió de garanties de temps real, també proporciona flexibilitat, en concret, la capacitat de modificar el comportament de la xarxa en temps d'execució mentre es mantenen les garanties de temps real compromeses. L'elevada fiabilitat en FTTRS s'aconsegueix mitjançant mecanismes que toleren les fallades que podrien afectar a la comunicació entre nodes.

En qualsevol cas, proporcionar tolerància a fallades únicament al subsistema de comunicació no és suficient per satisfer els requisits de fiabilitat més exigents de les aplicacions crítiques. Per aconseguir alts nivells de fiabilitat, les fallades en els propis nodes del sistema encastat distribuït també han de ser tractades.

En conseqüència, hem dissenyat diversos mecanismes de tolerància a fallades per tractar les fallades que puguin afectar al correcte funcionament dels nodes. Aquests mecanismes aprofiten les característiques del subsistema de comunicació FTTRS i del paradigma de comunicació FTT subjacent.

Concloent, en aquesta tesi veurem com podem, amb la introducció de mecanismes específics per tolerar les fallades dels nodes d'un sistema encastat distribuït basat en FTTRS, aconseguir molt elevats nivells de fiabilitat per al sistema en el seu conjunt. A més del disseny dels mecanismes de tolerància a fallades dels nodes, també mostrarem com es pot avaluar la fiabilitat resultant i establirem quin és el benefici obtingut, comparant aquesta fiabilitat amb la d'una versió no tolerant a fallades del mateix sistema.

Publications resulting from this dissertaion

In the text that follows I will give a chronological overview of all the publications that constituted a part of the work presented in this dissertation.

The first publication gives some ideas of how dynamic fault tolerance can be added to Distributed Embedded Systems as a means of increasing their reliability. Note that the ideas presented therein did not become a part of the present dissertation and are left for the future work.

- Sinisa Derasevic, Julián Proenza, and David Gessner (2013). "Towards dynamic fault tolerance on FTT-based distributed embedded systems". In: *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE, pp. 1–4

Following two publications are one of the first contributions related to this dissertation:

- Sinisa Derasevic, Julián Proenza, and Manuel Barranco (2014). "Using FTT-ethernet for the coordinated dispatching of tasks and messages for node replication". In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE

- Sinisa Derasevic, Manuel Barranco, and Julián Proenza (2014). "Appropriate consistent replicated voting for increased reliability in a node replication scheme over FTT". in: *Emerging Technology and Factory Automation (ETFA), IEEE*

The first one illustrates how to use the underlaying network protocol, the FTT-Ethernet, to coordinate all the system activities. The second one introduces the initial idea of achieving consistency among replicated nodes of Distributed Embedded Systems with regards to the locally performed majority voting. As will be explained, this idea will only be implemented and used partially.

The following set of publications is related to the simulation and implementation of the mechanisms presented in this dissertation:

- Sinisa Derasevic et al. (2015). "First experimental evaluation of the consistent replicated voting in the hard real-time ethernet switching architecture". In: *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE, pp. 1–4

- Sinisa Derasevic, Manuel Barranco, and Julián Proenza (2015). "An OMNET++ model to asses node fault-tolerance mechanisms for FTT-Ethernet DESs". In: *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE

- Alberto Ballesteros et al. (2016b). "First implementation and test of reintegration mechanisms for node replicas in the FT4FTT Architecture". In: *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE

- Alberto Ballesteros et al. (2016a). "First implementation and test of a node replication scheme on top of the flexible time-triggered replicated star for ethernet". In: *Factory Communication Systems (WFCS), 2016 IEEE World Conference on*. IEEE

Finally, the last publication presents the latest set of fault tolerance mechanisms that were devised in this dissertation as a means to diagnose faults and reintegrate the recoverable faulty node replicas:

- Sinisa Derasevic, Manuel Barranco, and Julián Proenza (2016). "Designing fault-diagnosis and reintegration to prevent node redundancy attrition in highly reliable control systems based on FTT-Ethernet". In: *Factory Communication Systems (WFCS), 2016 IEEE World Conference on*. IEEE, pp. 1–4

Acknowledgements

First of all I would like to thank my supervisors Julián Proenza and Manuel Barranco without whom it would have been impossible to complete this dissertation.

When I first came to the University of Balearic Islands, I came from a background of software development and programming of web applications which were my main fields of interest at the time being. I had no idea what research was all about.

I would like to express my gratitude to Julián Proenza who devoted a lot of time and effort to introduce me to the field of dependability which was completely new to me and to gradually get me to grow my interest in this topic.

Both Julián and Manuel welcomed me to their team and helped me countless number of times through my struggle to realize how the research should be done. They gave me their unconditional guidance, and encouragement throughout six difficult years.

Next, I would like to thank my colleagues Alberto Ballesteros, David Gessner and Inés Álvarez whom I worked with very closely. We all went through the same difficulties and struggles and helped each other out endless number of times. We belonged to a small group dedicated to dependability under the supervision of Julián and Manuel within the Systems, Robotics and Vision (SRV) Research Group.

Another important person from whom I learnt about OMNeT++ is Mladen Knezic from University of Banja Luka. I feel grateful to him for passing his knowledge and experience about network simulation.

I would also like to show my appreciation by thanking the leader of SRV group, Gabriel Oliver, for allowing me to work in his laboratory, and Alberto Ortiz, who helped me with all the administrative issues.

Finally, I would like to thank my father and my mother for all of their love and support throughout my entire life, and Marija who always stayed with me and supported me during both good and bad times.

Funding

This dissertation was supported by the Spanish *Ministerio de Economía y Competividad* through the FT4FTT project, grant DPI2011-22992 (MINECO/FEDER, UE) and also by the Spanish *Agencia Estatal de Investigación* and the *Fondo Europeo de Desarrollo Regional* through the DFT4FTT project, grant TEC2015-70313-R (AEI/FEDER, UE). My research stay at the *Universitat de les Illes Balears* was supported by the EUROWEB Project funded by the Erasmus Mundus Action II programme of the European Commission.

Contents

List of Figures

List of Tables

List of Abbreviations

A	**A**ctuate
BER	**B**it **E**rror **R**atio
C	**C**ontrol
CAN	**C**ontroller **A**rea **N**etwork
CDF	**C**umulative **D**istribution **F**unction
CEC	**C**ommunication **E**rror **C**ounter
CVEP	**C**c-**V**ector **E**xchange **P**rotocol
DCMV	**D**istributed **C**onsistent **M**ajority **V**oting
DEC	**D**iscrepancy **E**rror **C**ounter
DES	**D**istributed **E**mbedded **S**ystems
DTMC	**D**iscrete **T**ime **M**arkov **C**hains
EAV	Message **E**xchange of **A**ctuation **V**alues
EC	**E**lementary **C**ycle
ECAC	**E**xtended **C**ontrol **A**pplication **C**ycle
ESV	Message **E**xchange of **S**ensor **V**alues
FER	**F**orward **E**rror **R**ecovery
FCS	**F**rame **C**heck **S**equence
FT	**F**ault **T**olerance
FTT	**F**lexible **T**ime-**T**riggered
FTTRS	**F**lexible **T**ime-**T**riggered **R**eplicated **S**tar
FTU	**F**ault **T**olerant **U**nit
HaRTES	**H**ard **R**eal-**T**ime **E**thernet **S**witch
IC	**I**nternal **C**ounter
MFA	**M**aximum **F**ault **A**ssumption
NRDB	**N**ode **R**equirements **D**ata **B**ase
NVP	**N**-**V**ersion **P**rogramming
PG	**P**ort **G**uardian
PID	**P**roportional-**I**ntegral-**D**erivative
RT	**R**eal-**T**ime
S	**S**ense
SEU	**S**ingle **E**vent **U**pset
SIFT	**S**oftware **I**mplemented **F**ault **T**olerance
SRDB	**S**ystem **R**equirements **D**ata **B**ase
TDMA	**T**ime-**D**ivision **M**ultiple **A**ccess
TLLFL	**T**ransient **L**ong **L**asting **F**aults affecting **L**inks
TM	**T**rigger **M**essage
TMSN	**T**rigger **M**essage **S**equence **N**umber
TMW	**T**rigger **M**essage **W**indow
TNFP	**T**ransient **F**aults affecting the **N**odes manifesting as **P**ermanent ones
VCR	**V**oting **C**ommunication **R**ound
VS	**V**oting on **S**ensor values
VSUA	**V**oting **S**et-**U**p **A**lgorithm
YAA	**Y**ou **A**re **A**live

Dedicated to my parents and Marija

Chapter 1

Introduction

We shall start this chapter by defining the problem that we are going to solve with this dissertation. Next, we shall define the thesis statement which will serve as the aim of this study. After that, we will describe what are the main contributions, and finally, we shall give an overview of the rest of the thesis.

1.1 Problem statement

When *Distributed Embedded Systems* (DES) operate in evolving environments, changing requirements might be imposed on the system. The ability of the system to adjust its behaviour accordingly to the change in requirements is called *adaptivity* and requires flexibility at all system levels.

Moreover, when such systems are employed for real-time (RT) critical applications, support for both satisfying stringent RT requirements and attaining a high level of reliability must be provided.

The previously proposed *Flexible Time-Triggered* (FTT) paradigm (Pedreiras and Almeida, 2003) provides support for network adaptivity and stringent real-time requirements that RT DES require. Adaptivity is provided in the form of *operational flexibility*, i.e. the ability of the network to modify its behaviour at runtime. Moreover, FTT does this without jeopardizing the specified RT guarantees. However, it does not provide specific means to guarantee a high reliability.

A system can be provided with a high reliability by using fault prevention and/or *fault tolerance* (FT) (Laprie, 1992). Fault prevention refers to design methodologies and construction rules of entire systems including both hardware and software. Fault prevention includes best-practice software implementation techniques, rigorous design rules, component shielding and radiation hardening, etc. But, in spite of the use of fault prevention, faults may still occur. Thus, when truly high levels of reliability are needed, it is commonly accepted that it is necessary to use fault tolerance. Fault tolerance is focused on providing a correct service in the presence of active faults. In this dissertation, fault prevention is out of the scope and our focus is put on fault tolerance.

Since faults can appear in any part of a distributed embedded system, it is necessary to analyze how to tolerate them in each part. For distributed embedded systems based on the FTT paradigm there have already been some efforts to achieve tolerance to faults in the communication subsystem. More specifically, for the original implementation of FTT over the *Controller Area Network* (CAN) a number of fault tolerance mechanisms were developed that were intended to make FTT-CAN more suitable for critical applications (Ferreira et al., 2006; Barranco et al., 2006; Barranco, Proenza, and Almeida, 2009; Proenza et al., 2012).

The mechanisms for tolerating faults in communication subsystems based on FTT-CAN that are mentioned above are specially adapted to this specific technology

and thus, are not applicable to others. For later implementations of FTT, mechanisms for tolerating faults in the communication subsystem have also been proposed. Indeed, the FTT paradigm has been implemented over other protocols. In the recent years, due to its simplicity and low price-to-bandwidth ratio, Ethernet has become a ubiquitous networking technology in all domains, from household to industry. Thus, the FTT paradigm has also been applied to Ethernet and resulted in protocols such as FTT-Ethernet (Pedreiras, Almeida, and Gai, 2002), FTT-SE (Marau, 2009) and HaRTES (Santos, 2010). In the context of Ethernet, fault tolerance mechanisms which are specific for the FTT paradigm have been also designed. More specifically, the *Flexible Time-Triggered Ethernet Star* (FTTRS) (Gessner et al., 2013; Gessner, 2017) has been recently proposed to add channel fault tolerance mechanisms as a means to increase the reliability of critical RT DES that use the above mentioned HaRTES protocol (Santos, 2010).

Despite the improvement that FTTRS represents, it is known that in order to reach high levels of reliability, in addition to providing FT to the underlaying network, faults in the nodes of the DES must be tolerated as well (Barranco, Proenza, and Almeida, 2011). Therefore, node FT has to be addressed and suitable node FT mechanisms need to be designed.

1.2 Aim of the study and thesis statement

For the creation of an adaptive critical RT DES on top of an Ethernet-based implementation of the FTT paradigm, it is necessary to provide this system with a high reliability. This is done by ensuring the ability of the system to tolerate its own faults. We will assume that we count with the ability of FTTRS to tolerate faults at the communication subsystem level but we realize that the ability to tolerate node faults still needs to be added.

Therefore, the aim of this dissertation is to prove the following thesis statement.

"*It is possible to attain high levels of reliability of adaptive critical RT DES that rely on a reliable and flexible RT communication subsystem based on an FTT implementation on Ethernet by providing FT mechanisms for the nodes.*"

1.3 Main contributions

This section identifies the main contributions of the present thesis.

To fulfill the aim of the study described in the previous section we design different FT mechanisms to tolerate hardware faults in the nodes and hardware faults in the channel connecting them that may jeopardize their operation and/or communication. These mechanisms take advantage of the underlaying FTTRS communication subsystem (Gessner et al., 2013; Gessner, 2017) that was chosen due to the provision of network reliability, real-time guarantees and operational flexibility, the latter two inherited from the use of the FTT communication paradigm.

The FT mechanism that will serve as a basis for providing node FT is active node replication (Powell, 2012). Active node replication assumes that the critical nodes are identically replicated (same hardware and software), thus, if the replicas are provided with the same input, they will produce the same output, as long as their software does not include any non-deterministic programming constructs. Providing the replicas with the same input is called external replica determinism enforcement (Poledna, 2007) and will be explicitly addressed in this dissertation. Active node replication is best suited for RT systems since it requires no delay in the delivery

of the response. Moreover, this approach is fully transparent to the users in case of replica failures.

To compensate potential errors produced by the faulty node replicas we propose a solution we call *Distributed Consistent Majority Voting* (DCMV). DCMV consists of each node replica producing a result which is then reliably exchanged with the other node replicas. Once each replica has its own result and the results from the other node replicas, it locally votes on them to obtain a consensus result. All replicas then use this consensus result, so that the system compensates potential erroneous results produced by faulty replicas. Moreover, by using this consensus result instead of its own erroneous result, a transiently faulty replica may easily recover itself from the erroneous result it proposed. In some particular cases, however, transient hardware faults can manifest in a way that the affected replica cannot easily recover from. To handle these faults we provide a more sophisticated set of recovery mechanisms we call reintegration mechanisms. Moreover, some transient hardware faults may even lead a replica to behave as if it was permanently faulty as long as it is not reset. To cope with these situations, we further propose additional mechanisms for diagnosing when a replica is affected by this kind of faults and, then, to force it to resume and reintegrate. Note that without reintegration and fault diagnosis, some transient faults may unnecessarily lead to loss of node redundancy (redundancy attrition), since replicas affected by these faults may remain unable to participate in later votes.

Note that the adaptivity of the nodes themselves and the adaptivity of the fault tolerance techniques applied are not addressed and are beyond the scope of the present dissertation.

All the activities executed by our fault-tolerant system have to be synchronized, e.g. node replicas have to first execute certain application tasks in order to produce the results, then exchange the produced results by transmission/reception of messages using the underlaying network protocol, and lastly execute application tasks that vote on the locally produced result and the results received through the channel. We do this by taking as a starting point a previously proposed network-centric approach (Calha and Fonseca, 2002; Silva et al., 2005) and then modifying it by proposing our own approach for coordinated dispatching of tasks and messages using the underlaying protocol of the FTTRS.

Besides completing the development of all particularities of our system, we demonstrate the feasibility of our design on a particular case of control applications. We show how to apply all of the aforementioned mechanisms and techniques in this concrete case.

We have assessed our system using simulation, implementation and dependability evaluation.

Simulation was done using OMNeT++, an object-oriented discrete-event network simulation framework (Varga, 2001), and the INET framework (Varga, 2007) which is an open-source library for OMNeT++ that contains models for wired and wireless link layer protocols. The goal was to simulate a control system provided with our FT mechanisms and, then, inject different faults to verify via simulation that these mechanisms work as intended.

Next, some of my colleagues at the Universitat de les Illes Balears implemented a series of prototypes (Ballesteros et al., 2016a; Ballesteros et al., 2016b) that include our FT mechanisms. The goal of these implementations was twofold. One of the objectives was to experimentally demonstrate that the FT mechanisms proposed in this dissertation can be integrated with the ones FTTRS already provides. The other objective was to inject faults to further corroborate in an experimental manner that

our FT mechanisms work as expected. My contribution was to supervise that the implementation of the FT mechanisms proposed in this dissertation correspond to their design, to partially implement some of them, and to help in the design of the fault-injection experiments.

Lastly, we built up a dependability model of a DES relying on our FT mechanisms and FTTRS. We used this model to quantify the reliability that the system can achieve depending on several aspects. In this way we demonstrated that our FT mechanisms do indeed increase the system reliability. For building up the model and quantitatively measure the system reliability we used PRISM (Kwiatkowska, Norman, and Parker, 2011), a probabilistic model checker tool.

1.4 Thesis organization

The remainder of the dissertation is organized as follows. Chapter 2 introduces the basic concepts and terminology used to describe and design a dependable system. Special attention has been paid to describing the specific fault tolerance techniques used in this dissertation. After that, in Chapter 3 we present the foundations on top of which we develop our node FT mechanisms. In particular, we present the communication paradigm and protocol being used by the FTTRS, and then, the details of the FTTRS itself. Chapter 4 summarizes the previous works that use active node replication as a means to tolerate node faults and that are most similar to the design of our system.

Chapters 5 to 9 are devoted to the main contributions of this dissertation.

Specifically, Chapter 5 focuses on the node fault tolerance mechanisms. It starts with the general description of the overall system, followed by the specification of all the faults that we consider (fault model). Ensuing the fault model, we concentrate on describing the proposed FT mechanisms based on active node replication and FTTRS. Chapter 6 is dedicated to the realization of the designed FT mechanisms having particular applications in mind, more specifically, control applications.

Chapter 7 and Chapter 8 describe the simulation and real implementation of our system as a means to test and verify the correctness of our design. This is done by injecting different faults and then inspecting if the implemented mechanisms function as intended in a simulated and a real environment respectively.

Chapter 9 shows how we measure the reliability achieved by our system. Here, we specify a dependability model of our system and then demonstrate how we use it to numerically quantify the achieved reliability.

Lastly, Chapter 10 sums up this dissertation by giving a list of obtained conclusions and indications of possible future endeavors.

Part I

Background

Chapter 2

Basic dependability concepts

We begin this chapter by defining what dependability is. Then, we describe the general concepts in the field of dependability and identify the specific concepts that we use in this dissertation.

The terminology we use to refer to dependability and the concepts related to dependability is taken from the work of Jean-Claude Laprie (Laprie, 1992). In this work the definition of dependability is formulated as follows:

"*Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers*"(W. Craig, 1982)

The constitutional dependability concepts are described by the dependability tree depicted in Figure 2.1.

The *attributes* of dependability allow the different properties of the system to be expressed. The *reliability* is the ability of the system to provide its intended service continuously. The *availability* is the readiness of the system to provide its service. The *safety* is the ability of the system to not cause the effects that can have catastrophic effects on the environment and human lives. The *security* is the ability of the system to prevent an unauthorized access.

The *impairments* to dependability are "undesired circumstances causing or resulting from un-dependability" (Laprie, 1992). A *fault* is a defect in a system resulting from a deformity in its design or operation. A *fault* occurrence might provoke an *error*. An *error* is an improper state in the system deviating from the intended one, e.g. an unexpected value. An *error* can further cause a system *failure*. A system *failure* is a deviation from the intended service, i.e. an incorrect behaviour that does not conform to the one specified for the system.

As shown in Figure 2.2, the work done by (Avizienis et al., 2004) classifies all faults according to eight basic viewpoints. Each of these eight classes is called an *elementary fault class*.

Combining the presented elementary fault classes there can be 256 distinct *combined fault classes*. However, not all the combinations make sense (Avizienis et al., 2004). Therefore, the author of (Avizienis et al., 2004) identifies 31 likely combinations presented by Figure 2.3.

One of the prerequisites when designing a dependable system is to specify the *fault model*, i.e. to specify the expected classes of faults that the system can exhibit during its lifetime. The specification of the fault model usually consists of identifying from the above described combinations of faults (see Figure 2.3) which ones are to be expected to occur.

When a fault provokes an error that in turn causes a system to fail, the system and its constituting components can fail in different ways classified as *failure modes*. Failure modes are specified using the effects observed by the user of the system service. The classification of different failure modes presented next is taken from (Poledna, 2007; Proenza, 2007).

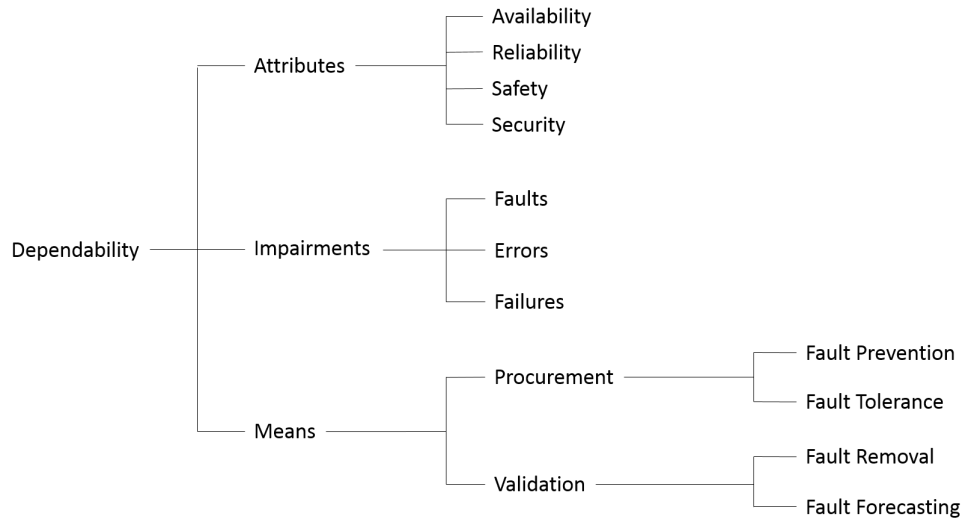


FIGURE 2.1: Dependability Tree

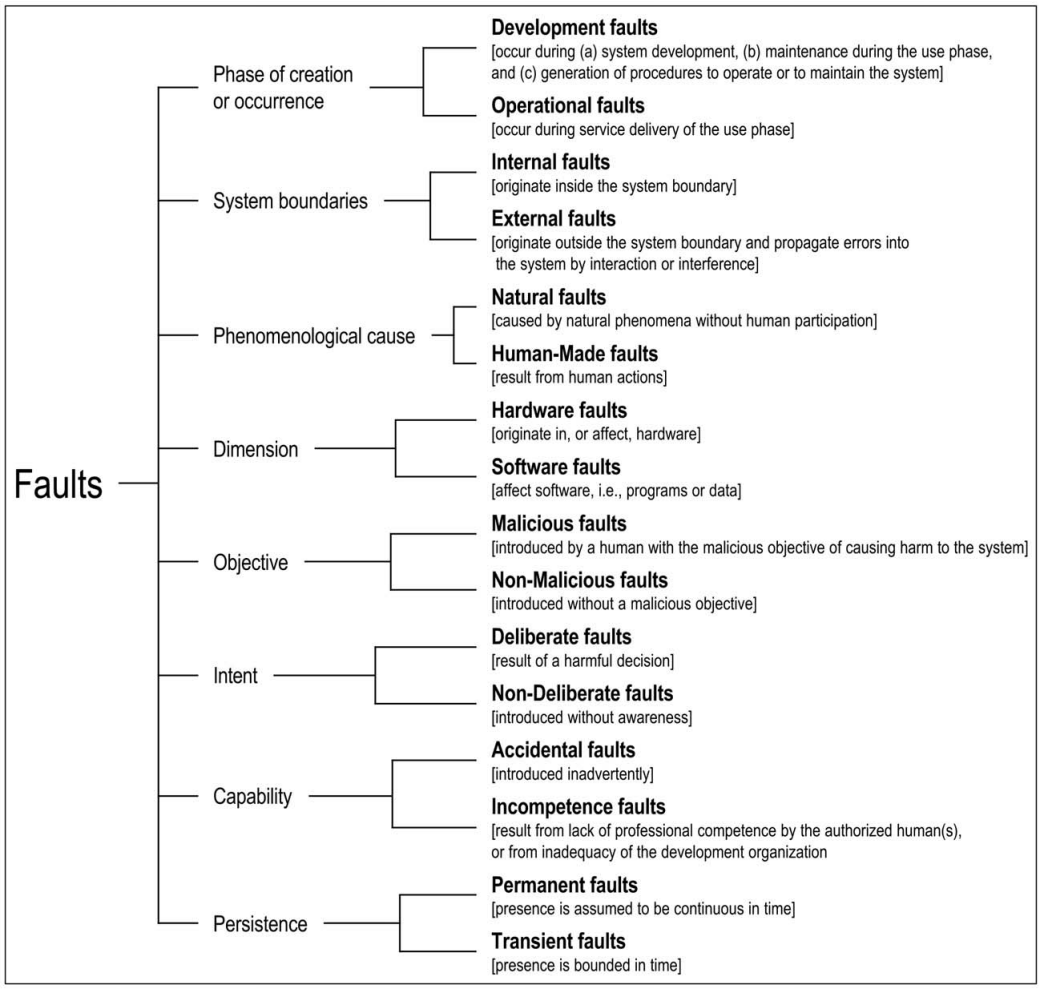


FIGURE 2.2: Fault classes (reproduced as it appears in (Avizienis et al., 2004))

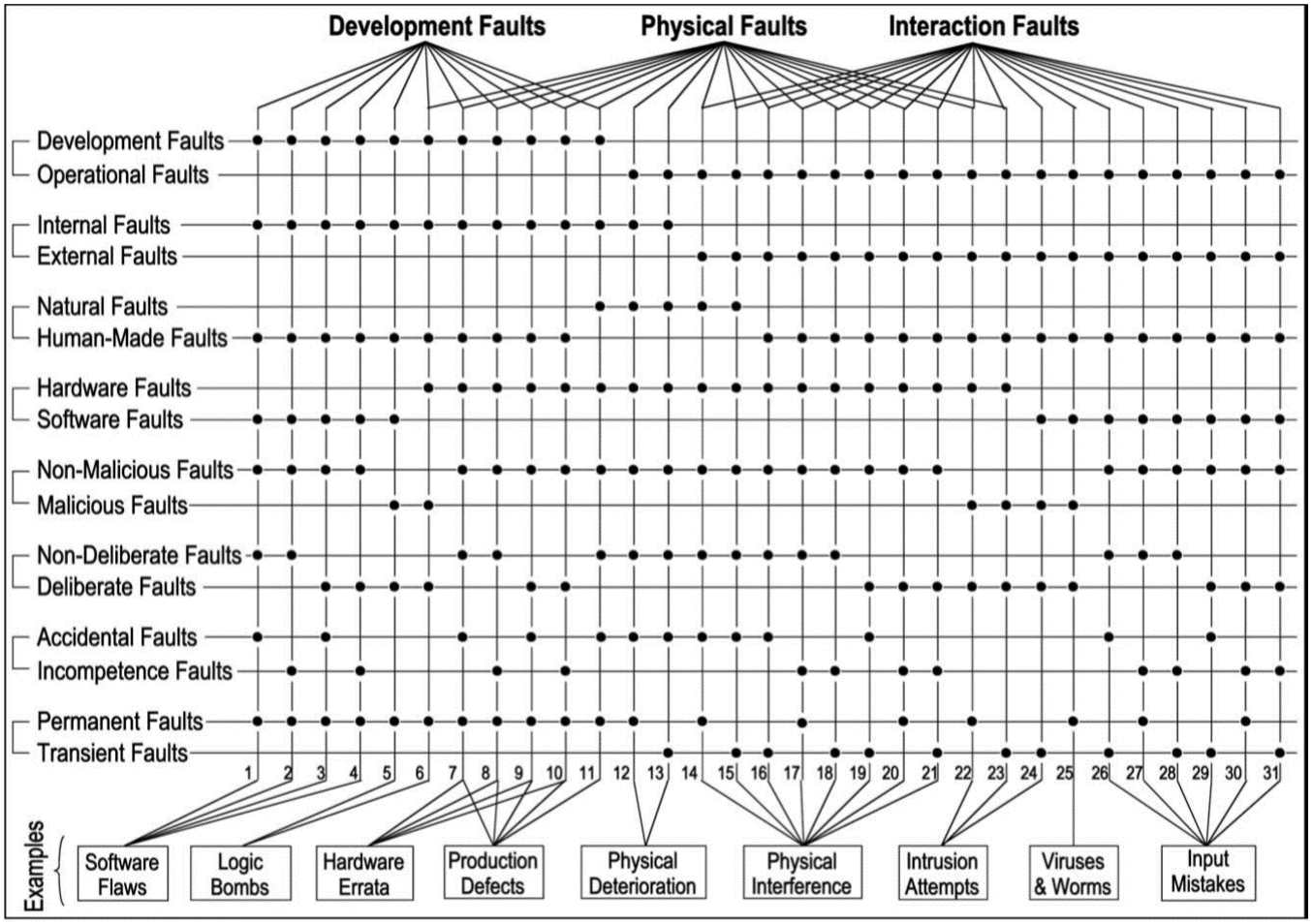


FIGURE 2.3: The matrix representation of the combined fault classes (reproduced as it appears in (Avizienis et al., 2004))

- *Byzantine or arbitrary failures* (Lamport, Shostak, and Pease, 1982). This failure mode is distinguished by the absence of any assumption of how the system might behave. The user of the system cannot have any expectation on the perceived effects on the system service that are caused by faults. Alternatively, this failure mode is referred to as *malicious* or *fail-uncontrolled*. This failure mode includes some undesired behaviours such as "two-faced" behaviour, i.e. a failed system can send a message "fact φ is true" to one user and a message "fact φ is false" to another user, and message forging, i.e. a system may falsify messages so that is appears as if the fabricated messages belong to another system.

- *Authentification detectable byzantine failures* (Dolev and Strong, 1983). This failure mode is same as the previous one with one exception, a system is unable to forge messages of another system, i.e. the failed system cannot lie about the facts that are sent by other systems. Typically, this is accomplished by authenticated messages.

- *Incorrect computation failures* (Laranjeira, Malek, and Jenevein, 1991). In this failure mode if a system fails, it can only fail by delivering incorrect results in either the value or the time domain.

- *Performance failures* (Cristian et al., 1986; Powell, 1992). This failure mode is similar to the previous one, but more benign. Particularly, if a system fails, it can fail only by delivering incorrect results in the time domain. Results may be delivered early or late.

- *Omission failures* (Perry and Toueg, 1986; Powell, 1992). Omission failures are a special case of performance failures. Specifically, results are never delivered, or in other words, they are delivered infinitely late.

- *Crash failures* (Lamport, Shostak, and Pease, 1982; Powell, 1992). In this failure mode a system fails by not responding to the current and any subsequent service request. If this is the case, it is said that the system has crashed.

- *Fail-stop failures* (Laprie, 1992). When a system presents this failure mode, it can only manifest as a crash failure, but additionally, this failure is assumed to be detectable by the other systems. This is done by assuming that the failed system maintained a stable storage which reflects the last correct service. The stored value can then be read by anyone even though the system maintaining it has crashed.

If a dependable system behaviour conforms to a specific failure mode, we can say that the system exhibits a *failure semantics* of the conformed failure mode. A more formal definition is given by (Poledna, 2007):

" A system exhibits a given failure semantic if the probability of failure modes which are not covered by the failure semantic is sufficiently low "

The restriction of failure semantics is usually done at the design stage of the system by using adequate design techniques. Depending on the targeted failure semantics to be achieved different mechanism can be used (Barborak, Dahbura, and Malek, 1993). Restricting a failure semantics of a dependable system can significantly facilitate its later design.

The *means* for dependability are methods and techniques used to ensure that reliance can be placed on the services that the system provides. These methods are grouped into four classes that are often combined to provide dependability:

- *fault prevention* methods prevent faults from occurring or being introduced in the system.
- *fault tolerance* methods focus on providing a service conforming to the specification in spite of faults.
- *fault removal* methods aim at reducing the seriousness and number of faults after their occurrence.
- *fault forecasting* methods assess the presence of faults, their frequency and the effects they might cause.

In this dissertation we devise the fault tolerance methods as a means to provide dependability. Specifically, the attribute that we provide is reliability. Therefore, the next section elaborates more on fault tolerance in general, and the specific techniques used in this dissertation in particular.

2.1 Fault Tolerance

As already explained, the goal of fault tolerance is to provide a correct system service in spite of faults. Fault tolerance is realized by means of *error processing* and/or *fault treatment*. *Error processing* purpose is to remove errors from the computational state, preferably before they can cause a failure. *Fault treatment* purpose is to prevent faults from causing any further errors.

Error processing may be realized by using the following two methods:

- *Error recovery*. It is done by restoring an error-free state starting from the erroneous state. This can be achieved using two different approaches:
  - *backward recovery* is done by restoring the system to a prior error-free state using the pre-saved points in time, called the *recovery points*, that were established before the error has occurred.
  - *forward recovery* is done by transforming an erroneous state with a new state in which the system may resume to provide its service, but possibly in a degraded mode.
- *Error compensation*. It is done by employing enough redundancy to allow the system to provide its service in spite of the erroneous internal state.

Fault treatment is accomplished by the execution of two subsequent steps. The first step is called *fault diagnosis* and it involves discovering what are the cause(s) of error(s) covering their location and nature. The next step is called *fault passivation* and its aim is to realize the prime goal of fault treatment which is to prevent faults from causing any further errors, i.e. to passivate them. This step is accomplished by excluding the identified faulty component(s) from the rest of the system execution. If this exclusion causes the system not to be able to preserve the delivery of intended service, then a *reconfiguration* of the system might be realized.

In this dissertation we make use of both error processing and fault treatment. As regards error processing, we use the error compensation approach in general and the replication approach in particular. As will be described later, we complement the node replication with error recovery in the form of forward error recovery.

When error recovery is being used, the first step is to identify the erroneous state before replacing it with an error-free one. This process is called *error detection* and always precedes error recovery.

It is important to note that if errors are compensated without using error detection, a phenomenon called *redundancy attrition* can manifest. Redundancy attrition occurs when faulty components are not detected and as a result the available redundancy is decreased without being noticed by the system.

Next, we list the different replication techniques according to the degree of synchrony among replicas in the time domain.

- *Lock-step*. All replicas execute exactly the same operation, at the same time. The same time for the replicas is provided by using a common hardware clock source. The examples of systems using this replication technique are Stratus (Taylor and Wilson, 1989) and Sequoia (Bernstein, 1988).

- *Active*. All replicas execute exactly the same operation, but without using a common hardware clock source. Some of the examples are CIRCUS (Cooper, 1984), Clouds (Ahamad et al., 1987), active replication in Delta-4 (Chérèque et al., 1992), MAFT (Keichafer et al., 1988), MARS (Kopetz et al., 1989) and SIFT (Wensley et al., 1978).

- *Passive*. Only one replica, called primary, makes all the decisions, and sends updates to the other replicas, called backups, which then apply the changes. An example of a system using passive replication is Tandem GUARDIAN (Bartlett, Gray, and Horst, 1987).

- *Semi-active*. Same as active replication concerning deterministic decisions, i.e. the decisions that have to produce exactly the same (identical) outcome in each replica, but with the exception that one of the replicas, called *leader*, can take non-deterministic decisions, i.e. the decisions that are taken independently from the other replicas and can yield an arbitrary outcome. Then, this non-deterministic decision has to be propagated to the other replicas, called *followers*. Examples of semi-active replication systems are XPA if Delta-4 (Barret et al., 1990) and ISIS (Birman et al., 1985).

The node FT mechanisms designed by this dissertation and the communication subsystem that these mechanisms are built on use active and semi-active replication. The reasons why we chose to use these two strategies are explained in the text that follows.

Active and semi-active replication are fully transparent to the users of the system if some of the replica(s) fail, which is not the case for passive replication. With passive replication, the failure of a primary replica would enlarge the response time, thus making it ineffective for real-time systems.

In lock-step replication technique the correct execution of replicas' operation is strongly dependent on the provision of common hardware clock source. This source has to provide a clock signal that has to reach all the replicas thus limiting the speed and distance among replicas as well as the number of replicas. These limitations are not in line with distributed systems in general. On the other hand, both active and semi-active replication require no common hardware clock source and the aforementioned limitations are not present.

Active and semi-active replication exhibit the so called *group failure masking semantics*. This means that if one of the replicas is faulty, the group as a whole can mask the fault quasi-instantaneously, which is in line with the critical real-time DES requirements. These techniques were successfully applied in avionics systems (Wensley et al., 1978), and dependable computer architectures such as MAFT (Keichafer et al., 1988) and Delta-4 (Barret et al., 1990).

Therefore, we can conclude that active and semi-active replication are the most appropriate for critical real-time DES.

The main drawback of both active and semi-active replication is that they require a great effort in order to enforce *replica determinism* (Poledna, 2007; Wiesmann et al., 2000), i.e. to ensure that the non-faulty replicas exhibit a consistent behavior in the absence of faults. Therefore, in the section that follows we will define what replica determinism problem is, and how to deal with it.

Once all the fault tolerance mechanisms have been designed, an evaluation of the overall fault-tolerant system is usually performed to verify the correctness and the effectiveness of its design. There are two kinds of evaluation techniques, *qualitative* and *quantitative* (Proenza, 2007).

The *qualitative evaluation* is fitting to verify the correctness of the system design that includes all the FT mechanisms which are necessary to cope with the expected fault model, i.e. the expected classes of faults the system can exhibit, which was explained at the beginning of this chapter. The work described in (Avizienis, 1995) proposes structured guidelines for qualitative evaluation.

The *quantitative evaluation*, also called *numerical evaluation*, is done to verify if the system meets its numerical requirements with regards to designated attributes of dependability. Quantitative evaluation is typically done by building a dependability model of the system which is then analyzed to obtain the numerical value for the desired dependability attribute.

A quantitative evaluation of a fault-tolerant system is usually performed to obtain the *coverage* of error processing and fault treatment mechanisms (Bouricius, Carter, and Schneider, 1969; Arnold, 1973). The coverage can be defined as a proportion of faults from which the system can recover using the designed error processing and fault treatment mechanisms. The estimation of coverage includes methods such as estimates, analyses, simulations, and experimentation with prototypes of critical elements under fault conditions. One of the most widely used method for obtaining coverages is called *fault injection* (Arlat, Crouzet, and Laprie, 1989; Gunneflo, Karlsson, and Torin, 1989) and is done by provoking different faults and then testing the designed mechanisms.

As mentioned before, the dependability attribute of interest for this dissertation is reliability. In Chapter 9 we will elaborate more of which specific quantitative evaluation techniques we use and how we use them to measure the reliability achieved by our system.

2.2 Replica Determinism

When replication is being used as a means to provide fault tolerance, even in case some of the replicas fail, assuming that no more than a given number of replicas have failed, the correct service should still be provided. However, it can happen that due to some undesirable side effects the non-failed replicas disagree and make it difficult to provide the intended service.

Consider the following example. There are two replicas that are in charge of controlling an emergency valve. In case excess pressure is detected, both replicas have to issue a command to close the emergency valve. However, the pressure is being read from an analogue sensor and it can happen that each of the replicas receive a marginally different reading as a result of limited accuracy when reading the sensor. This can lead to a scenario when one replica issues a command to close the emergency valve while the other concludes that the valve should be left open.

As a consequence, even though both replicas are non-erroneous, they disagree on the result. On the other hand, the similar scenario can happen even if both replicas read exactly the same input from the sensor. Concretely, if the decision to close the valve also depends on the duration of time in which the excess pressure lasts, it can happen that due to a small divergence in clock speeds one replica issues a command and the other one does not. Again, replicas disagree on the result even though they are non-erroneous.

Replica determinism problem deals with the aforementioned issue. Particularly, it has to ensure that disagreement among non-erroneous replicas does not occur even in the presence of undesired side effects.

Consequently, the replica determinism is intended for assuring that all the non-faulty replicas belonging to the same group behave in a deterministic manner (Poledna, 2007), i.e. that they do not disagree on the produced results. The more formal definition of the problem is as follows:

"*Correct servers show correspondence of server outputs and/or service state changes under the assumption that all servers within a group start in the same initial state, executing corresponding service requests within a given time interval*"(Poledna, 2007)

In the general scope of replication, the term server from the above definition is identical with the term replica. This definition is quite generic and is covering a wide spectrum of applications and replication strategies. Therefore, in the scope of a specific application and replication strategy the correspondence of outputs and service requests as well as time interval from the above definition would have to be defined more accurately.

Note that the replica outputs do not have to be equivalent. They need to show correspondence in the value and time domain that is specific to the application being executed. For example, as regards the correspondence of outputs in the value domain, for the applications that use floating point number values, correspondence requirement allows the outputs to differ by a given maximum deviation, whereas, for the applications that use boolean values, correspondence requirement needs the outputs to be identical.

Concluding, replica determinism needs to be defined within the scope of a specific application. When this is done, the next step is to pinpoint what are the sources that lead to non-deterministic behaviour, i.e. cause replica non-determinism, and then define techniques to prevent them.

There are many sources of replica non-determinism, e.g., when using hardware generators of random numbers, different replicas may yield different random numbers due to the difference in hardware, or if the decisions taken by the replicas depend on time, each replica uses its own clock to measure time, and, if the clocks are not synchronized, non-deterministic behaviour can easily occur.

Poledna (Poledna, 2007) made an attempt to characterize all the sources of replica non-determinism. The result of this characterization are the following three classes:

- *The real world abstraction limitation* - this source is caused by the abstraction that computer systems make when trying to quantify real world continuous processes by using a finite set of discrete numbers (discretization). It can happen that different replicas end up using different values for the quantification of the same real world continuous process.

  One example from Poledna's work (Poledna, 2007) of this source is when two temperature sensors measure the same temperature of exactly $100°$ C, and assuming that the discrete number 99 represents the temperature range $[99, 100[°$ C, and the discrete number 100 represents the temperature range $[100, 101[°$ C,

it can happen that due to the limited accuracy used by the sensors devices, one sensor outputs 99 and the other one outputs 100.

Another example of this source is the non-determinism introduced by the arithmetic operations. If replicas are not identically implemented, they can produce different outputs even if the inputs are identical, e.g. if one replica can represent up to 20 decimal points and the other one can represent up to 40, inconsistencies may emerge when these replicas are doing the same arithmetic operations and these inconsistencies can lead to replica non-determinism.

Note that in the second example the replica non-determinism can be prevented by using the identically implemented replicas, whereas in the first one the replica non-determinism cannot be avoided even when using the identical replicas. Therefore, this source can never be entirely avoided.

- *Impossibility of exact agreement* - we saw from the previous source that the replica non-determinism is always present at the interface between a computer system and the observed environment. Therefore, this source deals with the problem of impossibility to eliminate or mask the replica non-determinism in a systematic manner once it has been introduced (Poledna, 2007). It will be seen that in order to enforce replica determinism, the application semantics has to be considered. The following two approaches can be followed to cope with the previously described replica non-determinism:

  - *Application semantics analysis*: guarantee by analysis of the application semantics that non-deterministic observations have no effect on the correctness of the system's function (Poledna, 2007). Since the non-deterministic observations tend to be similar, replicas can take advantage of this similarity to make sure that the service responses of each replica correspond to each other. If this is the case, there is no need for the replicas to exchange information of their individual observations. However, only for very trivial systems it is possible to guarantee that the application semantics allows the inconsistent observations. As soon as we have more complex systems, it is very difficult to deal with replica non-determinism at the application level and this calls for quite complicated analysis and system design.
  - *Information exchange*: reach an agreement upon the observation of a real world continuous process by replicas exchanging information on their individual observations. Using this approach, the application complexity is significantly reduced and is shifted towards a protocol that has to reach an agreement among replicas, i.e. to reach a *consensus*. The first work on consensus was done by (Pease, Shostak, and Lamport, 1980).

Concluding, the first approach is an application-specific and unstructured solution than can only be applied to a limited number of applications whereas the second one is a systematic solution that can be widely applied to deal with replica non-determinism and therefore is frequently utilized.

However, it has to be noted that even with the second approach it is still impossible to completely eliminate the replica non-determinism and reach a perfect agreement among replicas. The actual kind of agreement strongly depends on the requirements of applications.

- *Intention and missing coordination* - This is the last class of replica non-determinism sources and is characterized either by the "intentional" or by the "non-intentional" introduction of replica non-determinism. A classical example of "intentional" introduction of replica non-determinism is the use of "true" random number generators where each replica can yield different random number. On the other hand, as regards the "non-intentional" introduction of replica non-determinism, it is caused by omitting the coordination between replicas with respect to non-deterministic behaviour. This usually happens due to the usage of non-deterministic language constructs and/or local information.

  Lastly, note that unlike the previous two sources that are always present to some extent, this one can be avoided by the proper design of the replicated system.

In order to prevent the aforementioned sources of replica non-determinism, replica determinism needs to be enforced. Different methods for enforcing replica determinism can be classified by asking the questions *where* and *how*.

When it comes to *where* to enforce replica determinism, according to (Poledna, 2007) there are two methods that can be used:

- *Internal* replica determinism enforcement is concerned with dealing with the intention and missing coordination source of replica non-determinism by restricting the functions used to implement a given service. This means that non-deterministic language constructs and local information should be avoided. Also, when implementing a given service, identical implementation is advised so as to avoid different replicas yielding inconsistent results.

- *External* replica determinism enforcement is concerned with dealing with the remaining two sources of replica non-determinism - the real world abstraction limitation and the impossibility of exact agreement. When obtaining observations from external resources such as a set of replicated sensors, a proper voting or adjudicating function (Di Giandomenico and Strigini, 1990) has to be selected in order to coordinate sensor inputs and thus enforce replica determinism. The selection of these functions is application-specific.

  Same as with the coordination of sensor inputs, communication services that are used to exchange observations among replicas have to be considered when enforcing replica determinism. Choosing the appropriate characteristics of the communication service minimizes the non-determinism introduced by the communication.

  When dealing with external replica determinism enforcement the following requirements are generally defined (Schneider, 1990; Cristian, 1991):

  - *Membership*: Every non-faulty server within a group has timely and consistent information on the set of functioning servers which constitutes a group.
  - *Agreement*: Every non-faulty server in a group receives the same service requests within a given time interval.
  - *Order*: Explicit service requests as well as implicit service requests, which are introduced by the passage of time, are processed by non-faulty servers of a group in the same order.

Note that these requirements can never be truly fulfilled and the replica non-determinism cannot be entirely prevented, only reduced. Because of the impossibility of exact agreement, the first two properties - membership and agreement - can never be absolutely fulfilled. These properties, including order, have to be relaxed depending on the system requirements. This relaxation allows the cheaper protocols to be used, but, it is necessary to know the semantics of service requests. For example, if service requests are commutative or independent, the order requirement can be relaxed. Also, if replicas use N-Version Programming (Avizienis, 1985), i.e. each replica uses an independently implemented program satisfying the same specifications, agreement requirement can be relaxed as well. However, note that these relaxations are application-specific solutions to fault tolerance which add complexity to the design of the system.

When it comes to *how* to enforce replica determinism, Poledna (Poledna, 2007) classifies the following two methods:

- *Centralized*: this method is also called *asymmetric* and it defines one server that can be distinguished from the others. This server is called *central* and all the other servers belonging to the same group have to be synchronized with it. The central server is the one controlling replica determinism by compelling the other servers of the group to accept its decisions and processing pace. The term that is used for the other servers depends on the correlation with the central server. When the non-central servers receive and process service requests together, but slightly delayed from the central server, the *follower* servers term (Powell, Chérèque, and Drackley, 1991) is used. When the non-central servers do not receive service requests, but instead receive from the central one checkpoint messages that contain the state of the service, the *standby* servers term (Budhiraja et al., 1991) is used. Examples of replicated systems making use of centralized method are the semi-active and the passive replication strategies of Delta-4 (Powell, Chérèque, and Drackley, 1991), the ISIS system (Birman et al., 1985) or some database-oriented systems that use checkpointing (Shin, Lin, and Lee, 1987; Koo and Toueg, 1987).

  The most evident advantage of the centralized approach is the ease with which communication protocols can achieve the order requirement. Since the communication, both internal and external, is always done through the central server, order is guaranteed implicitly. Also, non-deterministic decisions of the group can be resolved by sending them to the central server. Due to the aforementioned, the crucial disadvantage of this approach is the fact that the central server plays a critical role and its failure can jeopardizes the whole system. Thus, the central server cannot have byzantine failure semantics.

- *Distributed*: this method is also called *symmetric* because there is no central server and all the servers play the same role. The non-deterministic decisions and the processing pace have to be agreed upon, i.e. consensus has to be reached and therefore each server of the group executes the same communication algorithm. Examples of replicated systems making use of distributed method are MARS (Kopetz et al., 1989), Totem (Amir et al., 1993), Delta-4's active replication (Chérèque et al., 1992) and MAFT (Keichafer et al., 1988).

  The main advantage of distributed approach is that no server has a distinguished role, thus imposing no restrictions on the failure semantics the servers

have to exhibit. However, the communication protocols have to be more complex in order to fulfill agreement and order requirements.

Note that there can also be approaches that are in between the centralized and the distributed ones. One example is the broadcast protocol for the Amoeba system (Kaashoek and Tanenbaum, 1991) where the functionality about non-deterministic decisions is partly centralized and partly distributed. Another one is the broadcast protocol described in (Chang and Maxemchuk, 1984) where the role of the central server is performed sequentially by different servers.

Chapter 3

Foundations

Now that we have introduced all the relevant dependability concepts, in this chapter we are going to describe the foundations on top of which our node FT mechanisms are designed. In particular, we are going to focus on the part "reliable and flexible communication subsystem" from the thesis statement, c.f. Section 1.2. Being even more specific, we will start off by describing the communication paradigm and the protocol based on this paradigm that are used throughout this dissertation. Then, we will delve in the details of the communication subsystem that makes use of the described paradigm and protocol and that serves as a basis for our node FT mechanisms.

3.1 FTT paradigm and HaRTES

The *Flexible Time-Triggered* (FTT) communication paradigm (Pedreiras and Almeida, 2003) is a real-time communication paradigm that provides support for adaptivity of RT DES in the form of operational flexibility, i.e. the ability to change the traffic requirements on the fly, i.e. during the runtime, while maintaining real-time guarantees.

The FTT communication paradigm has been implemented on different network technologies resulting in different FTT protocols. The first protocol was FTT-CAN protocol (Pedreiras and Almeida, 2000) designed for *Controller Areal Network* (CAN). However, in the recent years Ethernet has been replacing the competing network technologies due to its support of ever-increasing bit rates and its simplicity. Consequently, the FTT protocols using Ethernet are: FTT-Ethernet protocol (Pedreiras, Almeida, and Gai, 2002) designed for shared Ethernet, and FTT-SE protocol (Marau, 2009) designed for COTS switched Ethernet.

Moreover, the FTT-based designs have proved to be a viable solution in the scope of adaptive systems and recent works in this area show that there is an on-going interest in continuing improving the RT-related features of the FTT protocol (Garibay-Martínez et al., 2016; Ternon, Goossens, and Dricot, 2016; Ashjaei, Behnam, and Nolte, 2016; Ashjaei et al., 2016).

The specific FTT protocol being used throughout this thesis is HaRTES (Hard Real-Time Ethernet Switch) protocol (Santos, 2010). The HaRTES protocol is a master/multi-slave publisher-subscriber protocol implemented on top of the standard Ethernet. Each switch of the FTTRS is an enhanced Ethernet switch that embeds the FTT master within and that is where the name HaRTES comes from. The HaRTES protocol data is encapsulated within the standard Ethernet frame payload. The protocol supports both real-time (divided into time-triggered, also called synchronous, and event-triggered, also called asynchronous) and non-real-time traffic classes. The protocol organizes the communication into fixed-duration time slots called *Elementary Cycles* (ECs). Each EC is further separated into three windows, c.f. Figure 3.1.

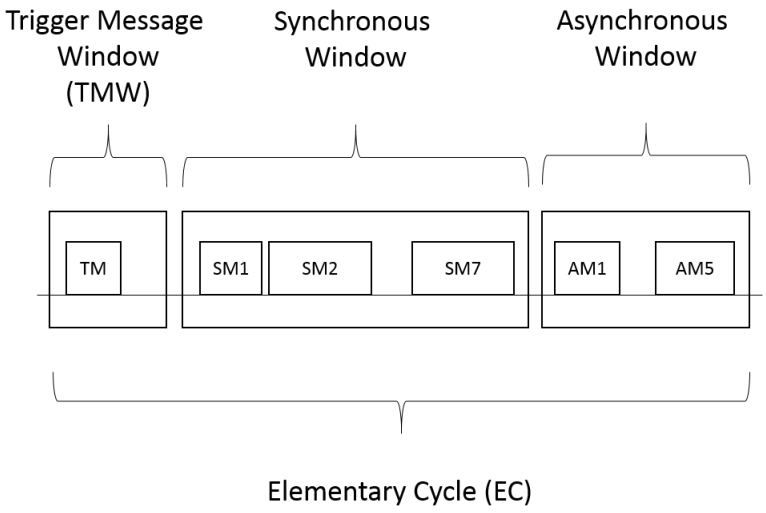


FIGURE 3.1: Elementary cycle

Trigger Message Window (TMW) is used by the FTT master node embedded in the switch to broadcast a special control message, called the *Trigger Message* (TM), to all the FTT slaves. The TM contains the schedule for the synchronous messages that are due to be transmitted in the current EC. Thus, the FTT master polls the synchronous traffic directly.

Synchronous Window is intended for the exchange of synchronous messages.

Asynchronous Window is intended for the exchange of asynchronous messages and non-real-time messages. This traffic in not polled directly by the FTT master, but queued in dedicated memory pools inside the HaRTES. HaRTES then shapes this traffic by transmitting it within this dedicated window so it does not interfere with the synchronous traffic.

The architectural components of the HaRTES protocol that include the enhanced FTT-enabled switch with the embedded FTT master (see Figure 3.2), and the FTT slave are presented next. The introduction of the architecture and the operation of the specific components of HaRTES protocol will be a basis for understanding the simulation and implementation described by Chapter 7 and Chapter 8.

The components in the gray zone belong to the FTT master. The *System Requirements Data Base* (SRDB) is the central repository for all the traffic management related information such as the message attributes for synchronous and asynchronous real-time traffic (e.g. period, priority, deadline), global configuration information(e.g. EC duration, the duration of the synchronous window) and resource allocation information. The admission control and optional QoS Manager closely collaborate with the SRDB to guarantee real-time traffic timeliness. They receive and process change requests assuring that any previously negotiated schedule will be fulfilled. At the beginning of every EC the scheduler scans the SRDB to build a list of synchronous messages that should be sent in that EC (EC-schedule). The EC-schedule is sent to dispatcher that is turn broadcast the TM conveying the received EC-schedule.

The rest of the components, non-gray (white) zone, belong to an enhanced FTT-enabled switch excluding the above described FTT master.

We shall now go in depth describing functionalities of components belonging to HaRTES input and output areas depicted in Figure 3.2, and components belonging to FTT Slaves.

HaRTES input area

When packets get received they are classified by the packet classifier in one of the following: FTT real-time packets, non-real-time packets, and FTT request packets. An FTT packet is validated and, if deemed valid, stored in the corresponding synchronous or asynchronous packet memory queue of the global memory pool. A non-real-time packet is stored in a non-real-time queue of the global memory pool. An FTT request packet is targeted for the FTT master and forwarded to the admission control/QoS Manager module. Whenever a packet is placed in one of the queues of the global memory pool, the packet forwarding process is activated. In case of FTT real-time packet, the packet forwarding module does not consult MAC address, but is based on a producer-consumer model that inspects SRDB to determine ports that have consumers attached and forward packets from global memory queues to corresponding output port queues. Non-real-time packets consult MAC address like in standard Ethernet switches. One of the main advantages of this architecture is the seamless integration of both FTT and non-FTT-compliant nodes. Note that the non-FTT-compliant nodes can send their traffic as non-real-time packets and they will conform to procedures of the standard Ethernet packet forwarding.

HaRTES output area

Output ports have the same queues with the same meaning as in the global memory pool. The forwarding process described above does not move the actual packets, but what happens is that the queues store the pointers to the packets in the global memory pool. The port dispatcher stores the information about the EC windows and handles packet transmission. Initially, the FTT real-time packets are transmitted in the appropriate windows and at the end, if there is enough time, non-real-time packet are transmitted at the end of the asynchronous window. As concerns the TM, it is transmitted directly from the FTT master dispatcher module to all the output ports.

FTT Slave architecture

The architectural components of the FTT slaves are much simpler. The FTT slaves only have to obey the commands that come from the FTT master. Thus, the most relevant components of the FTT slaves are discussed next. The *Node Requirements Data Base* (NRDB) is a counterpart of the SRDB in the FTT slave. The FTT slave memory pool stores the packets received both from the application and the network in different queues. Finally, the dispatcher module is responsible for the message transmission according to the EC-schedule conveyed by the TM received from the FTT master.

3.2 FTTRS

Flexible Time-Triggered Replicated Star (FTTRS) is the communication subsystem devised by our team as a means to add fault tolerance to the FTT communication paradigm that uses Ethernet networking technology taking HaRTES protocol as a basis while not jeopardizing the support that FTT provides for RT and operational flexibility (Gessner et al., 2013; Gessner, 2017). Overall, the FTTRS provides the necessary network requirements for adaptive real-time DES executing critical applications. On the one hand, it provides a support for network adaptivity and stringent

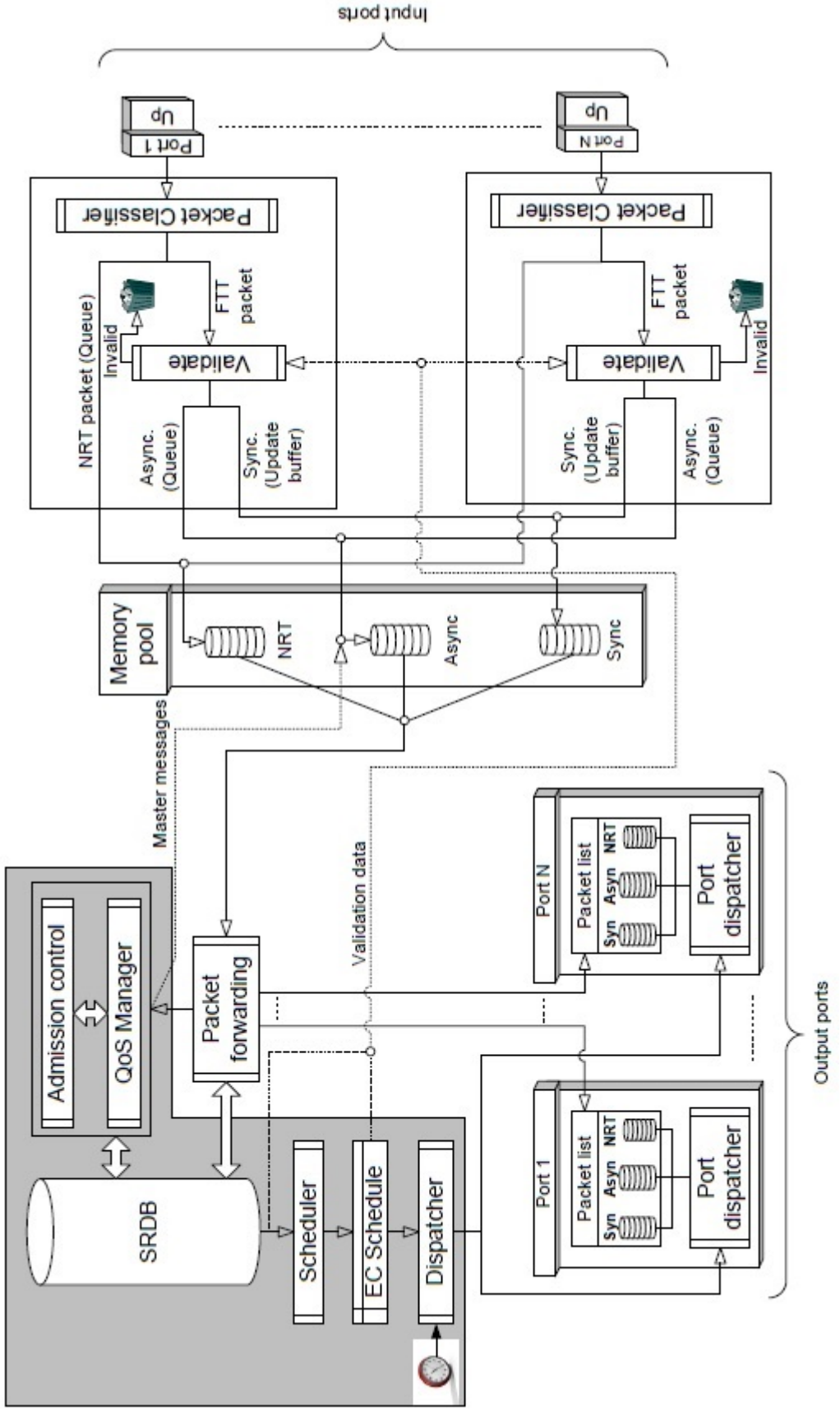


FIGURE 3.2: HaRTES (reproduced as it appears in (Santos, 2010))

RT requirements by using the above described HaRTES protocol. On the other hand, the support for the critical applications is provided by developing different fault tolerance mechanisms on the network level.

For the reasons explained above, in this dissertation we use the FTTRS as a basis on top of which we implement our node fault tolerance mechanisms to further increase the reliability achieved by the overall system. In the next sections we will first describe the fault model used by the FTTRS, followed by its architecture, and finally, the FT mechanisms designed to handle the faults considered by the introduced fault model.

3.2.1 Fault model

Fault model describes which faults and associated rate of occurrence are assumed by the system being designed. The FTTRS assumes the faults enclosed by the gray square depicted in Figure 3.3 which was described previously in Chapter 2 (Figure 2.3).

The faults considered by the FTTRS are non-malicious operational hardware faults. According to different viewpoints these faults can be: system boundaries - internal or external; phenomenological cause - natural or human-made; intent - deliberate or non-deliberate; capacity - accidental or incompetence; persistence - permanent or transient. Two examples of the faults considered by the presented fault model are physical deterioration and physical interference as seen in Figure 3.3. These faults are caused by processes such as radiation, power transients, noisy input lines, etc.

As regards the rates of faults in the FTTRS, permanent and transient faults are distinguished. No specific rates were considered for any of them in the design of the FTTRS, but as will be shown in Chapter 9, we will assume some domain specific failure rates when measuring the obtained system reliability.

All components of the FTTRS have unrestricted failure semantics, i.e. they can fail in an arbitrary manner. Exception from this are transient faults in the links. It is assumed that they are detectable by Ethernet *Frame Check Sequence* (FCS) which drops the corrupted frames and transforms these faults into frame omissions. When designing FT mechanisms, having an unrestricted failure semantics is very difficult to cope with due to the complexity needed to handle all the different effect the failed components may experience.

Therefore, the FT mechanisms of the FTTRS are divided into two subsets. First, the FTTRS includes a subset of FT mechanisms devoted to restricting the failure semantics of the components. This restriction would then facilitate the second subset of FT mechanisms that are devoted to masking the effects of now restricted component failures, and providing a non-disrupted communication service.

In the next sections we are first going to present the FTTRS architecture, and then to discuss the aforementioned subsets of FT mechanisms.

3.2.2 Architecture

The building blocks of the FTTRS are: FTT slave nodes, enhanced FTT-enabled switch and FTT Master (see Section 3.1), henceforth switch, and links.

The switch, including both FTT-enabled switch and FTT Master, is a single point of failure, i.e. if it fails, the complete system fails, due to inability to provide communication services. To eliminate such as single point of failure, FTTRS relies on two

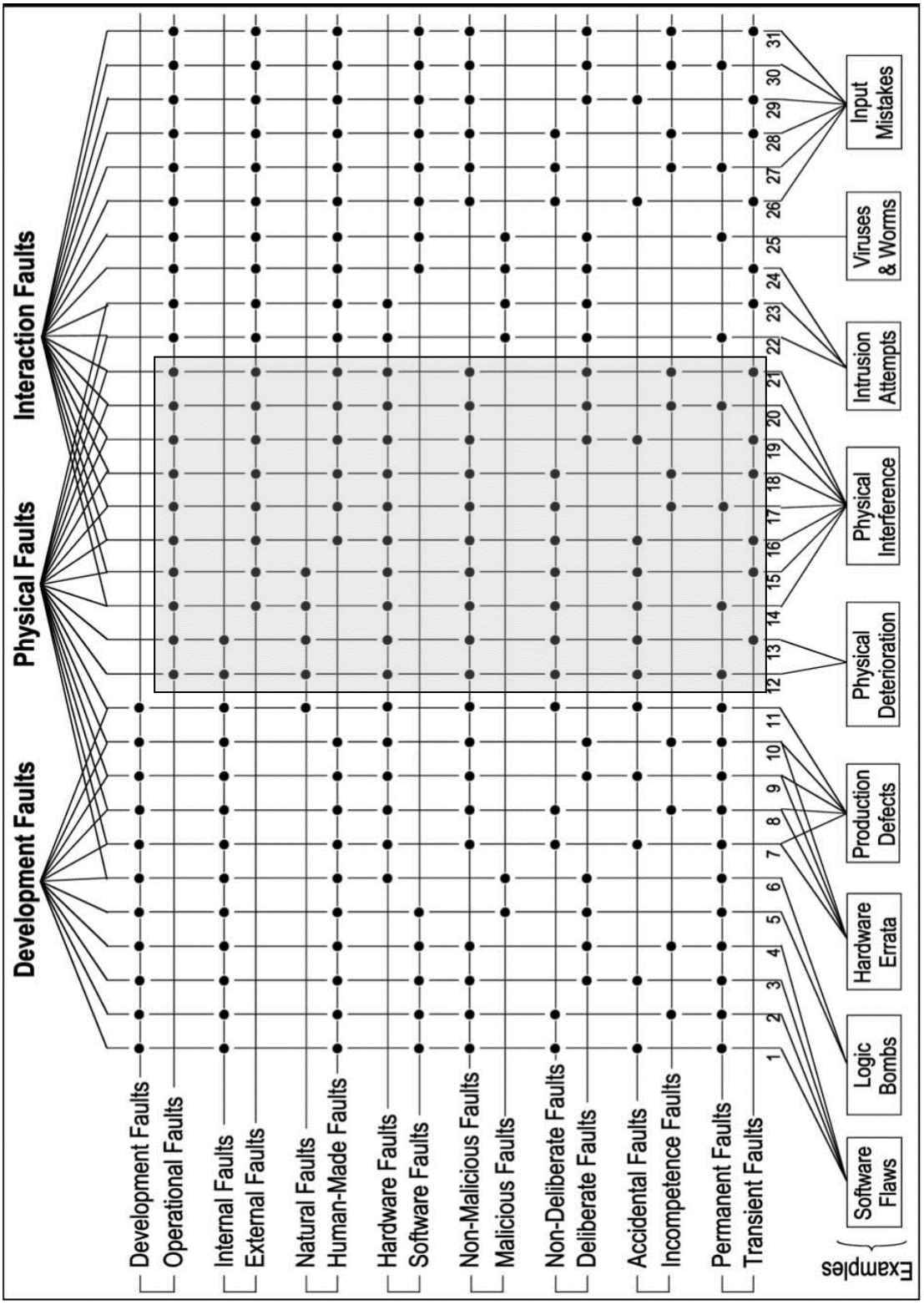


FIGURE 3.3: The classes of faults considered for FTTRS (source (Avizienis et al., 2004))

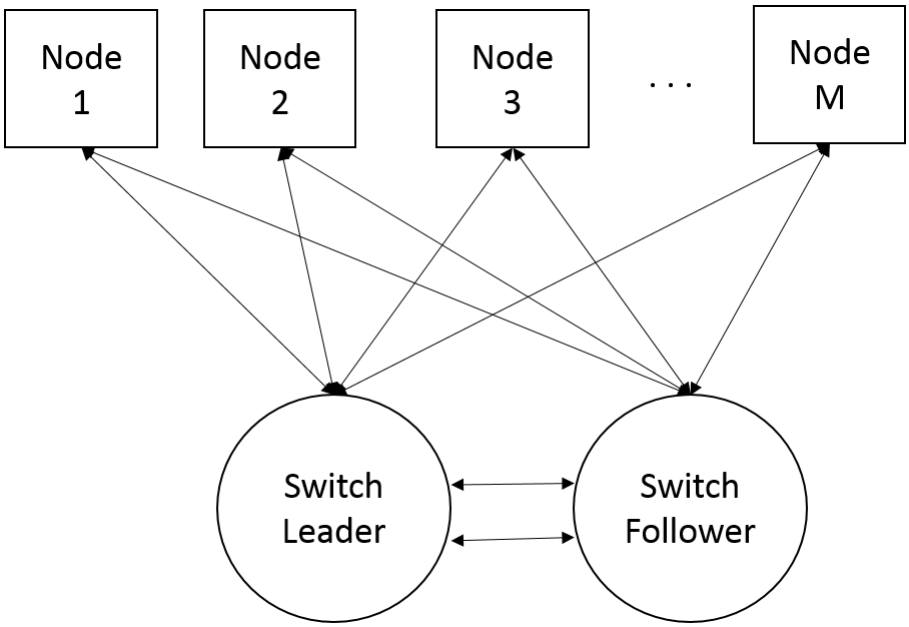


FIGURE 3.4: FTTRS architecture

independent switch active replicas; if one switch replica fails, the other one continues providing communication services.

As already said, replica determinism needs to be enforced (Poledna, 2007) so as to ensure that the surviving replica/s of a replicated group, a surviving switch in this case, provide/s a correct service. FTTRS provides several mechanisms to ensure that both switches are replica determinate (Gessner, 2017). Since some of these mechanisms need switches to exchange information with each other to reach an agreement with regards to different aspects, both of them are interconnected by at least two bidirectional (full-duplex) links called interlinks. All interlinks are used in parallel, so that no interlink represents a single point of failure.

Each FTT slave needs to be connected to both of the switches to tolerate the failure of one of them (or of the link that connects it to a given switch). In this way, if one of the switches (or a link) fails, an FTT slave can still use the other non-faulty switch (or link) for communicating.

Figure 3.4 shows the resulting architecture of FTTRS , which is composed of two switches interconnected by at least two interlinks, and several FTT slave nodes that are connected to each one of the switch replicas.

It is noteworthy to mention that each switch forwards through the interlinks the messages received from the slaves directly connected to it. This provides four redundant paths between each pair of slaves, and keeps the network connected in spite of faults affecting multiple links.

However, this provokes a phenomenon called *replica radiation* (Gessner, 2017). Specifically, this phenomenon causes the number of slave messages transmitted in the downlinks to be doubled compared with the ones in the uplinks. Figure 3.5 illustrates this phenomenon. As we can see, the slave located at the left hand transmits 2 replicas of the same message, each one through a different uplink; whereas the slave on the right hand receives two copies of this message per downlink.

This phenomenon can be desirable as it increases the reliability of the communication. However, it may unnecessarily limit the performance of the network when, in order to tolerate transient faults, the system already includes mechanisms for replicating the transmission of messages in the time domain.

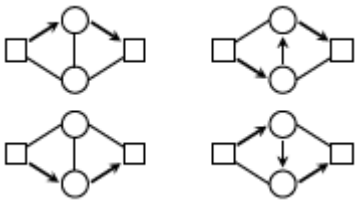


FIGURE 3.5: Replica Radiation in FTTRS (reproduced as appears in (Gessner, 2017))

This is an important aspect to take into account since, as it will be explained in the next Section, FTTRS also provides mechanisms to pro-actively retransmit critical messages. Moreover, some of the fault tolerance mechanisms proposed in this dissertation (Chapter 5) rely on or introduce pro-active retransmissions as well.

For example, in FTTRS a slave can pro-actively retransmit k copies of a given critical message through each one of its uplinks. Thus, replica radiation would provoke each receiving slave to receive $2 * k$ copies of that message per downlink.

Whether or not replica radiation represents a disadvantage from the performance point of view depends on the application. Fortunately, replica radiation can be disabled when necessary by configuring each FTTRS switch to appropriately restrict the number of copies of a given message it forwards through its downlinks. In fact, replica radiation has been disabled in some of the prototype implementations of FTTRS so far.

For the sake of simplicity, in this dissertation we consider that replica radiation does not happen. On the one hand, we will assume that replica radiation is disabled when using the FTTRS pro-active retransmission mechanisms. On the other hand, we will design new pro-active retransmission mechanisms so that they do not provoke this phenomenon.

This decision of preventing replica radiation from happening does not limit the attainable reliability. This is because the level of time redundancy of each pro-active retransmission mechanism should be chosen to attain a desired reliability, independently of whether or not replica radiation is enabled. Moreover, FTTRS can be configured and our new pro-active retransmission mechanisms modified to enable replica radiation if desired.

3.2.3 FT mechanisms

In this section we explain first how components of the FTTRS may fail. Afterwards, we describe the FT mechanisms used to restrict the failure semantics, and finally, we describe how this restriction is used to implement further FT mechanisms to deal with expected restricted component failures.

Components of the FTTRS have unrestricted failure semantics (Gessner et al., 2013; Gessner, 2017), i.e. each component may fail arbitrarily. The exception are links that due to the FCS of Ethernet transform frames corrupted by hardware link faults into omissions. Concluding, FTT slaves and switches can fail in an arbitrary fashion while the links fail by corrupting frames which are in turn dropped after Ethernet inspects the FCS.

To restrict failure semantics of the switch, both internal composition units, FTT-enabled switch and FTT master, use internal duplication and comparison. Specifically, the hardware circuitry of the switch and its including composition units are

internally duplicated and compared and if any discrepancy is detected by the comparison, the complete switch turns itself off. This failure semantics is called *crash failure semantics* and means that the switches either provide a correct service or crash, i.e. remain silent permanently.

To restrict failure semantics of the slaves, previous work (Ballesteros et al., 2013) proposes to further enhance both switches with devices called *Port Guardians* (PGs) attached to each port of the switch. PGs inspect the traffic coming to connected ports and drop all frames deemed incorrect considering a set of specified rules. Using the PGs, slaves' failure semantics is restricted to *incorrect computation failure semantics*, but only from the point of view of the other slaves. In particular, slaves still fail in arbitrary manners, but after the PGs' filtering, all untimely frames and frames exhibiting undesired behaviours, e.g. a replica trying to impersonate a frame of another replica, are dropped and a slave can only fail to deliver a correct result to another slave in value or time domain (incorrect message or no message at all(dropped by the PGs)).

No further actions are taken to restrict failure semantics of the links since it is already benign and easy to cope with.

After the first applied set of FT mechanisms described above, the failure semantics of the FTTRS components is as follows. Switches have crash failure semantics, i.e. they fail either by proving a correct service or remaining silent, slaves have incorrect computation failure semantics, i.e. they can fail only by delivering an incorrect result to another slave in value or time domain, and links failure semantics remains the same, they fail by corrupting frames which are in turn transformed into omissions.

Next, we explain the FT mechanisms used by the FTTRS to tolerate both permanent and transient hardware faults in all the components of the FTTRS. We will see how the above failure semantics restriction facilitates the FT mechanisms that follow.

First off, since slaves are not considered being part of the communication subsystem, the FTTRS does not provide mechanisms to tolerate slave permanent and transient faults. However, as already explained, mechanisms that restrict their failure semantics are devised so that slave faults cannot prevent other slaves from communicating. In particular, PGs will filter out all the undesirable behaviours of the slaves that can jeopardize the correct functioning of the communication. Nonetheless, recall that the main aim of this dissertation is devoted to providing FT mechanisms to tolerate both permanent and transient slave faults as will be described later in Chapter 5.

Permanent faults in the FTTRS components that form a part of the communication subsystem are tolerated by means of switch and link replication (Figure 3.4).

Transient faults in the switches are transformed into permanent ones due to the crash-failure semantics enforced by the duplication and comparison of the switches' internal circuitry (Gessner, Proenza, and Barranco, 2014a; Gessner, 2017). This means that the switches either provide a correct service or crash, i.e. remain silent permanently.

Lastly, transient faults in the links affect the messages being transmitted by the slaves and the switches, and are being transformed into omissions using the Ethernet FCS. To tolerate these faults and omissions they cause, the FTTRS proposes to pro-actively transmit each message k times assuming that the number k has to be sufficiently high so that the probability of a failed message transmission becomes negligible.

As regards the slave messages, the number of replicas for each message sent by the slaves is stored in the SRDBs and the NRDBs as an additional message attribute (see Section 3.1) so that both the switches and the slaves have the knowledge of redundancy message levels. Having this in mind, slaves know how many messages they should transmit and switches can drop all the message replicas that go beyond the predefined redundancy level for that message.

As regards the TMs transmitted by the switches, their redundancy level is defined in the SRDBs with other global configuration information. Note that in the HaRTES protocol a single TM is broadcast in the TMW to all connected slaves and is received by each of them approximately at the same time ensuring a synchronized start of the corresponding EC. Now, instead of broadcasting a single TM, the FTTRS proposes that each switch broadcasts multiple TM replicas as a means to tolerate transient links faults. Since some of the TM replicas may be lost due to omissions, a solution for EC synchronization has to be provided bearing this in mind.

The solution comprises in both switches *isochronously*, i.e. at quasi-simultaneous time instants, broadcasting their TM replicas. The sequence number for each TM replica in conveyed within the TM payload and can be used by receiving slaves to determine the start of synchronous window in EC by calculating how much TM replicas are left to be received. For the details of managing TM redundancy by slave node an interested reader can refer to the previous works (Gessner, Proenza, and Barranco, 2014a; Gessner, 2017).

Moreover, all TM replicas have to contain the same payload. This is important both from the point of view of tolerating permanent failure of one switch and from the point of view of the above described mechanism of isochronous broadcasting of TMs. This can be ensured by enforcing replica determinism of the switch replicas, i.e. switch replicas must, starting from the same initial state and same input, produce *corresponding* outputs.

The outputs that the switches produce are the TMs. Using the TMs, switches command the start instance of each EC and the transmission schedule of periodic messages in that EC. Thus, we can discriminate the output correspondence in the time and value domain. The time domain correspondence is achieved if both switch replicas transmit their TM quasi-simultaneously and the value domain correspondence is achieved if both switch replicas produce exactly the same TM content (same periodic messages schedule).

As regard the replica determinism enforcement in the time domain, semi-active replication strategy described in Section2.1 is used. One of the switches is the *leader* and will transmit TMs according to its own internal clock through both links and interlinks. The other switch is the *follower* and it also does the same, but it synchronizes with the leader with the reception of leader TMs. Concretely, it uses the arrival time of the received TMs to decide whether it needs to defer or advance the start of its next TM transmission (Gessner, 2017).

As regards the replica determinism enforcement in the value domain, active replication strategy described in Section2.1 is used. Since both switches start from the same initial state and exhibit crash failure semantics no internal non-determinism can be expected. However, externally switches might receive different inputs from the slaves. To overcome this, switch replicas exchange all the information relevant for TM production through interlinks so that they have the same input needed for TM production and then unambiguously determine which information has to be considered, details in (Gessner, Proenza, and Barranco, 2014b; Gessner, 2017).

How to tolerate slave faults was not considered by the FTTRS. Therefore, in the

Chapter 5 we will describe how we build on top of the just described FTTRS communication subsystem slave FT mechanisms. The incorrect computation failure semantics enforced for the slaves will be used to facilitate the development of the node FT mechanisms.

Chapter 4

Similar active node replication proposals

This chapter is devoted to the identification of the works that are similar to the one proposed by this dissertation.

We will first present active node replication works in general that are similar to our work. This is done due to the fact that we use this technique as a basis to tolerate the node faults. After listing these, we will give an overview and comparison with our approach and then we will shift our focus to the communication subsystems by identifying some of the specific solutions built on communication subsystems similar to the one used by us, FTTRS.

One of the first research done on this subject was *Software Implemented Fault Tolerance* (SIFT) (Wensley et al., 1978). This is a very important reference in the history of fault-tolerant computing because it is the first experience in the study of fault tolerance in distributed systems and focuses on the use of standard avionics computers as the nodes of the complete system. The problem of replica determinism was explicitly addressed and the *consensus* problem, which was explained in detail in Section 2.2, was first defined in the context of this system (Pease, Shostak, and Lamport, 1980).

Another software-based architecture that uses node replication is Chameleon (Kalbarczyk et al., 1999). In this case the architecture was proposed in the context of adaptive fault tolerance. This work uses specific objects, called ARMORs (Adaptive, Reconfigurable, and Mobile Objects for Reliability) to create fault-tolerant software infrastructure. However, this work does not explicitly address the replica determinism problem.

Some systems that make use of the active replication technique use very specific architectures and designs, and have been developed for very specific applications. Next, we will list some of these works that we have identified as similar to our system.

This architecture for flight control systems and incorporated fault-tolerance mechanisms are designed specifically for JAS39 Gripen aircraft (Alstrom and Torin, 2001). Active replication is applied for the aircraft actuators which use the same replicated software, and voting is done to mask the faulty replicas. Replica determinism is mentioned, but an explicit solution of how to achieve it was not provided.

The work in (Chtepen et al., 2009) describes how active replication in combination with technique called checkpointing (saving state from which the component can recover) can efficiently be deployed to provide fault tolerance in grid computing environments. Depending on the current grid load, hybrid fault tolerance approach that switches between these two techniques has been proposed. In case when enough computing resources are available, active replication is used, and in

case when the system is overloaded, checkpointing is used. This work does not deal with enforcing replica determinism.

The work in (Ductor, Guessoum, and Ziane, 2011) presents an on-line adaptive active replication technique concerning the number and location of replicas for large scale Multi-Agent Systems (MAS). This work employs active replication adaptively as a function of two main parameters: agent criticality and the amount of available resources. As in the previous work, the replica determinism problem was not addressed here as well.

Software-based fault tolerance using active replication can be integrated in JAVA ahead-of-time compiler called KESO (Thomm et al., 2011), which allows the application to be replicated transparently, i.e. the application is unaware of the existence of replication. This work proposes suggestions of how to enforce replica determinism based on analyzing the application code in order to detect indeterminism and deal with it.

The previous works were either too general or quite application specific. Hence, we will focus next on more complete works that propose entire architectures providing hardware solutions for active replication.

The Multicomputer Architecture for Fault Tolerance (MAFT) (Keichafer et al., 1988) was designed for real-time control systems requiring ultra high reliability. MAFT employs physical partitioning of the nodes into two distinct subsystems. One for performing application related functions, called application processor (AP) and other for performing system executive functions, called operation controller (OC). Active replication usage is suitable for MAFT system. The problem of replica determinism was addressed by employing interactive consistency (Pease, Shostak, and Lamport, 1980) and convergent voting algorithms (Dolev et al., 1986) to reach an agreement.

The Fault Tolerant Multiprocessor (FTMP) (AL Hopkins, Smith III, and Lala, 1978) is very important reference for fault tolerant computing and also its successor the Fault Tolerant Processor (FTP-AP) (Lala and Alger, 1988). Similar like in MAFT this architecture divides each node into two subsystems: the core FTP, which plays a similar role to MAFT's OC, and the Attached Processor (AP), which has a similar role to MAFT's application processor. The designers of FTP-AP were very concerned with the Byzantine Generals' problem which was also addressed in MAFT and followed the techniques from (Pease, Shostak, and Lamport, 1980).

Delta-4 (Chérèque et al., 1992) is an open dependable distributed computing systems architecture. Like the previous two each node is divided into two subsystems. Host subsystem is in charge of application functions while the Network Attached Controller (NAC) subsystem is in charge of system executive functions. Fault-tolerance technique used is active replication of run-time software components on host computers which are interconnected by a local area network. The replica determinism problem is explicitly addressed and consensus is achieved using proprietary *atomic* multicast [1] protocol.

The Generic Upgradable Architecture for Real-time Dependable Systems (GUARDS) (Powell et al., 1999) is a fault-tolerant computer architecture based on Computer-Off-The-Shelf (COTS) components. GUARDS can be configured along three different dimensions (channels, lanes, integrity levels) to meet the dependability requirements of a wide variety of end-user applications. The specific interactive consistency protocol used in GUARDS is based on the so-called ZA algorithm (Powell, 2001).

[1] Atomic refers to both reliability and total order, i.e. the messages have to be received reliably by all the nodes belonging to the same multicast group and in the exact same order.

The table 4.1 gives an overview of all of the aforementioned approaches. Column *System* stands for the name of the system, column *Replica Determinism* says whether a system addressed the problem of replica determinism or not, and column *Active Replication* depicts whether a systems can be used to employ active replication in hardware (hw) or in software (sw).

System	Replica Determinism	Active Replication
SIFT	yes	sw
Chameleon	no	sw
JAS39 Aircraft	no	hw
GRID	no	sw
MAS	no	sw
MAFT	yes	hw
KESO	yes	sw
FTP-AP	yes	hw
DELTA4	yes	hw
GUARDS	yes	hw

sw : software solution.
hw : hardware solution.

TABLE 4.1: Comparison of systems using active replication

As already mentioned, all but the last three approaches listed in Table 4.1 are either too general or too application specific and mostly lack details of specific FT techniques applied. Some of them not even consider replica determinism enforcement that is crucial when replication is being used.

FTP-AP uses a *penalty count* to disconnect a lane (replica) in the presence of persistent erroneous behavior. Specifically, when a lane produces an erroneous result, a penalty is accounted for said lane. When this penalty reaches a specific threshold, a faulty lane is excluded from the others. However, there is no indication of how to resynchronize a faulty lane. Oppositely, in this dissertation we define how to completely resynchronize (reintegrate) a faulty replica after the presence of persistent erroneous behavior as will be explained in detail in the later chapters.

DELTA-4 does not take advantage of low-level services provided by the communication subsystem and was designed to provide a general high-level communication protocol that can be used on standard LAN technologies, specifically token-ring, token-bus and FDDI. Our approach on the other hand does take advantage of our communication subsystem, FTTRS, and as a result the replica determinism enforcement is considerably simplified.

The main disadvantage of GUARDS is the fact that the restriction of failure semantics of the nodes is not being used. As a result, there is a need for expensive and complex network topology. In particular, in GUARDS each node requires a communication channel. Our system on the other hand relies on a communication subsystem (FTTRS) that restricts the failure semantics of the nodes, so that we can use a simpler network topology not requiring one channel per node. As was explained before, we use a replicated star to connect the nodes.

Having explained the aforementioned works that make use of active replication technique as a means to tolerate node faults we move on to the works with the most similar communication subsystems and protocols as the one used by this dissertation.

Our active node replication architecture is built on top of FTTRS, our Ethernet-based implementation of FTT communication paradigm. Next, we present the architectures that are closer to our proposal but use different real-time protocols. Those are the architectures based on TTP/C (Kopetz and Grunsteidl, 1993), FlexRay (Makowitz and Temple, 2006) and TTEthernet (Kopetz et al., 2005) protocols. We have to note that unlike FTTRS, the aforementioned protocols are not suitable for adaptivity.

The aforementioned works propose a way to deal with node failures by replicating and grouping them into what they called *Fault Tolerant Units* FTUs, but they do not address the specific details, e.g. which replication strategy to use or how to reach an agreement between replicas within an FTU depending on the replication strategy used. Although they propose reaching an agreement among nodes for data messages by using a membership service, how to use this service for node replication is not addressed in detail.

Moreover, although there have been some works on the modeling of the reliability of the group membership protocol that all of these protocols include (Rosset et al., 2012) and broadcast protocols used by FlexRay (Souto, Portugal, and Vasques, 2016), the dependability evaluation that includes all the components of the complete fault-tolerant architecture has not been addressed yet, to the best of this author's knowledge.

Part II

Main Contribution

Chapter 5

Node Fault Tolerance

We will start this chapter by giving an overview of our system architecture and organization, followed by its fault model. Fault model section will also include the description of the failures that our components may exhibit, so called failure semantics. Then, taking into account the presented fault model and failure semantics, we describe in detail the devised node FT mechanisms. Recall that the node FT mechanisms are the main aim of this dissertation, c.f. Section 1.2, and also one of the main contributions.

5.1 Overall System Description

As shown in Figure 5.1 our system architecture is composed of M nodes, of which the critical ones are replicated N times. The nodes are interconnected by the replicated switches and links of the FTTRS communication subsystem. The FT techniques designed for the FTTRS were explained in detail in Chapter 3. Therefore, in this section we focus mainly on the nodes and their FT.

Nodes are replicated by means of *active replication* (Powell, 2012). Active node replication implies that all node replicas belonging to the same group play identical role and should simultaneously provide the same service, and, in case some of them fail, the provided service should not be interrupted. Note that each node can be replicated, thus, all the replicas of a single node belong to one replication group, e.g., as seen in Figure 5.1, Node 3 is replicated N times, and Replicas 1 to N belong to the replication group of Node 3. As was explained before, we chose active node replication technique as the best suited one for RT systems when compared to the alternative replication approaches (see Section 2.1).

In order to provide fault tolerance by means of node replication we use a strategy based on the *N-Version Programming* (NVP) paradigm (Chen and Avizienis, 1977;

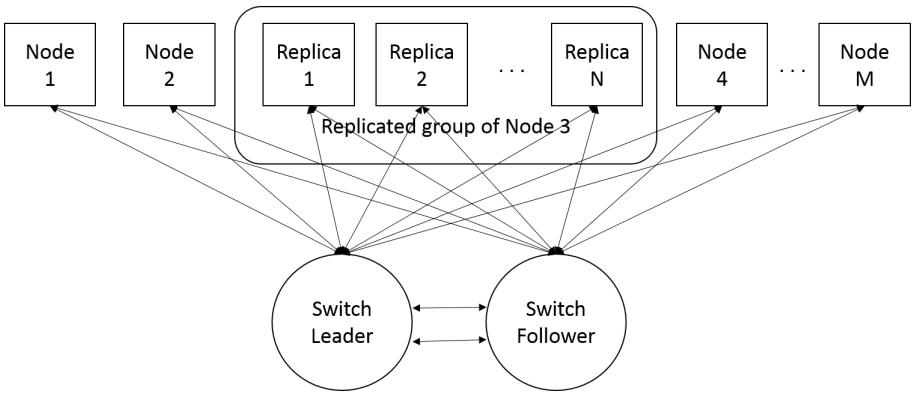


FIGURE 5.1: Complete system architecture

Avizienis, 1985), which is intended for software fault tolerance. As seen in Figure 5.2, according to NVP each replica's operation is divided into sequentially executed fragments, called segments. Each replica calculates (C) an output called *cross-check vector* (cc-vector), which is then sent to all the other replicas for comparison (E). Once each replica obtains all the cc-vectors, they use a majority voting function (V) to obtain a consensus value that then is used, thereby compensating the erroneous outputs produced by faulty replicas. Moreover, each replica uses this consensus value as the input of the next segment in order to carry out *Forward Error Recovery* (FER); where FER is generally defined as replacing an erroneous state by an error-free one. In this way, any transiently faulty replica may recover from an erroneous state that would be characterized by the fact of erroneously assuming that the output it generated was correct.

It is important to note that although we use the just mentioned fault-tolerance strategy of NVP, we are not proposing an NVP solution. First, this is because NVP is intended to tolerate software faults, while in contrast we are devoted to tolerating hardware faults. Second, because what we propose is a set of mechanisms that are not only devoted to providing error compensation and forward error recovery, but also other features that allow to further improve reliability.

More specifically, the cornerstone mechanism that we propose is called Distributed Consistent Majority Voting (DCMV). As it will be explained later on in this chapter, the novelty of DCMV is that it allows including the just-mentioned additional mechanisms to further improve reliability. For instance, recall from Section 2.2 that when active node replication is used as a means to provide fault tolerance, the replica determinism problem (Poledna, 2007) needs to be addressed. On the one hand, it is necessary to enforce internal replica determinism, so that all non-faulty replicas produce the same output as long as they are provided with the same inputs. We enforce internal replica determinism by using the identical hardware and identical software constructs. On the other hand, making sure that all replicas are provided with the same inputs is what is called external replica determinism enforcement. This is one of the aspects where DCMV plays a crucial role to attain high reliability; since it includes mechanisms to enforce external replica determinism, i.e. it makes sure with high probability that all replicas receive the same inputs for voting.

Lastly, in order to build up a RT application, it is necessary to execute its different tasks at the appropriate instants of time. This calls for different synchronization mechanisms. Therefore, in order to support real-time features at the application level in our node replicas we take advantage of the *synchronization service* that the TM already provides. Recall that the TM is already used to synchronize the start of ECs in all the nodes connected to the FTTRS communication subsystem. Thus, in this dissertation we propose a solution that takes advantage of the TMs in order to also trigger tasks in node replicas. This idea arose from the *network-centric* approach for the coordination of all system activities proposed in (Calha and Fonseca, 2002) and the work on CAMBADA robots (Silva et al., 2005) where the TM content was modified in order to, besides triggering message exchange among nodes, also trigger the execution of tasks in the nodes. In Section 5.3 we delve deeper in the details of how we implement this approach and how we make it fault-tolerant.

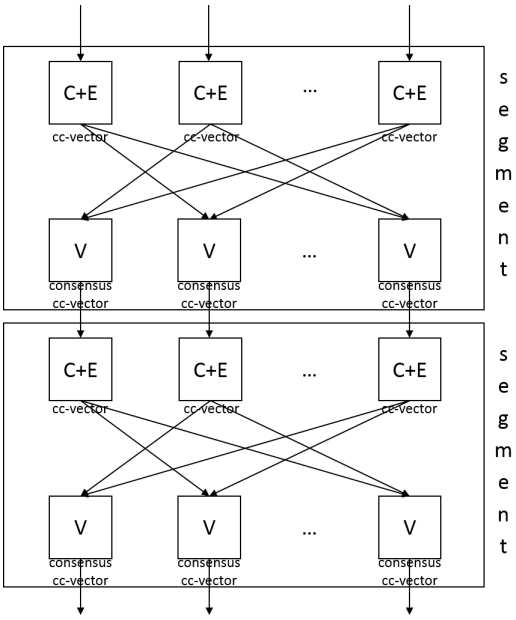


FIGURE 5.2: DCMV following the NVP strategy

5.2 Fault Model and Failure Semantics

The fault model considered for this dissertation is the same one assumed for the FTTRS described previously in Section 3.2.1 (see Figure 5.3), i.e. we consider non-malicious operational hardware faults.

However, as regards the persistence of faults we shall extend the classification by introducing two new classes of faults. Besides permanent and transient faults we shall introduce two intermediate classes describing the persistence of faults. One class will be introduced for the links and one class for the nodes of the FTTRS.

First, to tolerate transient faults in the links, the FTTRS proposes the use of pro-active retransmission of critical messages in a single EC (see Section 3.2.3), e.g. the TM is sent k times in the TMW of an EC to ensure that link transient faults affecting the TM transmission/reception are tolerated. However, in this dissertation we assume that in some cases transient faults can last longer than the pro-active retransmission mechanism can handle. We shall refer to this new type of faults as *Transient Long Lasting Faults affecting Links* (TLLFL). There are many examples in which TLLFL may occur. For instance, although the maximum length of burst can be somehow predicted, their duration is actually arbitrary. Thus, it may happen that the length of a burst produced by space radiation or lightnings in dynamic environments, e.g. a spacecraft, a space probe, or a commercial aircraft, do actually exceed the temporal redundancy provided by the FTTRS pro-active retransmission mechanism. In fact, the higher the bit rate of the communications (in Ethernet the bit rate has been increasing over the last years), the higher the probability that a burst length surpasses the temporal redundancy. The example of these type of faults was evident in in-orbit experiment (Takano et al., 1996). To deal with this new class of faults we shall provide additional FT mechanisms that are going to be one of the topics of the next section.

Second, in some cases transient faults in the node replicas can manifest in such manner that the effects caused by them cannot be distinguished from the effects caused by permanent faults and thus prevent the affected node replica from correctly communicating and/or operating as long as it is not reinitialized and recovered. The

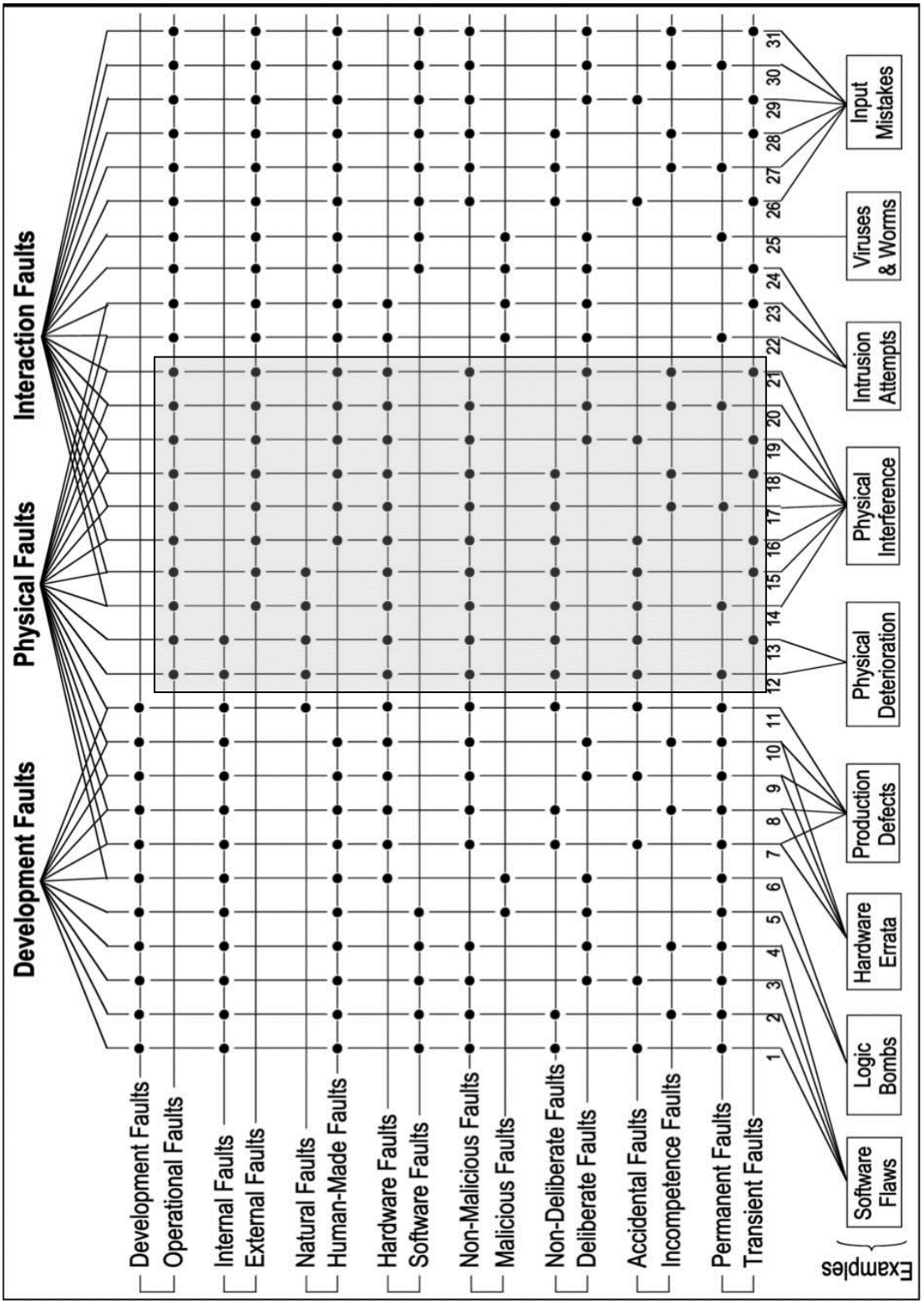


FIGURE 5.3: The classes of faults considered for the overall system (source (Avizienis et al., 2004))

TABLE 5.1: Fault classification according to persistence

	Switch	Node	Link
Permanent	✓	✓	✓
TLLFL			✓
TFNP		✓	
Transient	✓	✓	✓

name we attribute to this new class of faults is *Transient Faults affecting the Nodes manifesting as Permanent ones* (TFNP).

An example of TFNP is described next. In memory devices the most prevalent type of transient faults are *Single Event Upset* (SEU) faults (Dodd and Massengill, 2003). SEU causes the change of a bit from one stable binary to another. This bit change can leave a certain memory location in an altered state and the node can keep accessing it reading the corrupted data. This effect (reading of corrupted data) will persist unless the error (bit flip) caused by the transient fault (SEU) is corrected. The FT mechanisms that deal with these faults are described in detail in the next sections.

Concluding, according to fault persistence faults considered for the overall system of this dissertation are depicted in Table 5.1. Note that there is no equivalent of TLLFL for the nodes and TFNP for the links. First, in the case of TLLFL, they only make sense in the context of pro-active retransmission designed for tolerating transient fault in the links. For the nodes, this can be seen as multiple consecutive transient faults, and we already have FT mechanisms in place to deal with this. Second, the links cannot be reinitialized like the nodes, and therefore, in the context of the links TFNP would make little sense.

Now that we have described our fault model, we explain how the components of our system may fail (our failure model), taking into account how FTTRS already restricts their failure modes. In other words, we explain the failure semantics of each one of the components that constitute our system.

Recall that switch replicas have crash failure semantics, i.e. they fail either by proving a correct service or remaining silent, and link failure semantics is restricted to omission failures, i.e. links can fail only by omitting messages. In particular, in case of link transient faults corrupting the transmitted frames, the CRC mechanism of the Ethernet detects them and turns them into omissions. How these restrictions were achieved was already explained before in Chapter 3.

The nodes can fail in arbitrary manners, i.e. there is no assumption or restriction on their behaviour in case of a failure. However, as was already explained in detail in Section 3.2.3, due to the switches and their attached port guardians (PGs), the nodes' failure semantics is restricted to *incorrect computation failures* (Laranjeira, Malek, and Jenevein, 1991) from the point of view of the other nodes. This means that a node can fail to deliver the correct results either in the time or value domain. Further details of how the node faults manifest in terms of node communication and/or operation will be given in the next section. This restriction will, besides preventing error propagation, facilitate the design of the node FT mechanisms presented in the next sections of this chapter.

5.3 Node Fault Tolerance Mechanisms

This section describes one of the core contributions of this dissertation. Here, we explain in detail all the devised node FT mechanisms and connect them with the fault and the failure models.

5.3.1 Error compensation

As was already explained, we use active replication of nodes to tolerate permanent node faults following a strategy similar to the one proposed in NVP (Figure 5.2).

More specifically, we tolerate node faults by means of error compensation. For this, we propose what we call the Distributed Consistent Majority Voting (DCMV), which allows nodes to consistently vote in a distributed manner.

As regards the fault model, the DCMV is devoted, mainly, to cope with permanent faults that prevent node replicas form correctly operating and/or communicating, e.g. a node replica that cannot transmit and/or receive, a node replica that crashes, etc. Furthermore, the DCMV includes the time redundancy that allows tolerating transient faults and, to some extent, also TLLFL. Moreover, since DCMV also serves a basis for achieving forward error recovery, fault diagnosis and reintegration, as will be explained later, it serves a basis for tolerating and reintegrating from TLLFL and TFNP.

The design of the DCMV is facilitated by the incorrect computation failure semantics that the PGs attached to the switch replicas enforce for the node replicas. Thanks to the PGs, from the point of view of non-faulty replicas, a faulty replica manifests as either omitting (not transmitting) its cc-vector, or as transmitting a cc-vector with an incorrect data.

Now, we will briefly explain how the DCMV works. After some calculation each replica produces an output called cc-vector. Cc-vectors are then exchanged through the communication channel so that each node replica obtains all the cc-vectors. Once the cc-vectors are exchanged, each node replica locally performs a majority voting function. By voting in such a way, the system compensates the potential errors delivered by faulty node replicas. Note that, thanks to the incorrect computation failure semantics that PGs enforce, those errors can only manifest (from the perspective of node replicas) as incorrect/missing cc-vectors. The result of each voting is a consensus value, i.e. the consensus cc-vector, that is supposed to be the same in all the non-faulty node replicas. All these actions constitute one segment. The consensus cc-vector of one segment is the input for the next one, as in NVP (Figure 5.2).

Note that a non-faulty replica can produce the correct consensus cc-vector in one segment only if it votes with a majority of non-erroneous cc-vectors and if it remains non-faulty in that segment. Thus, in a previous segment at least a majority of replicas had to be non-faulty to ensure a majority of non-erroneous cc-vectors. Also, the non-faulty replicas had to be interconnected so that cc-vectors could have been exchanged. From the aforementioned we derive our *Maximum Fault Assumption* (MFA): the system provides its intended service if at any time at least a majority of non-faulty replicas are able to exchange their cc-vectors. If MFA is violated, we assume that the system fails, even though, as will be seen later, in some cases we can tolerate faults that go beyond this MFA.

As already pointed out in Section 5.1, for the majority voting to succeed, it is necessary to enforce external replica determinism, i.e. all non-faulty node replicas must be provided with a consistent set of inputs to vote on. Thus, to make the MFA to hold with a high probability, and thus attain a high system reliability, the DCMV itself

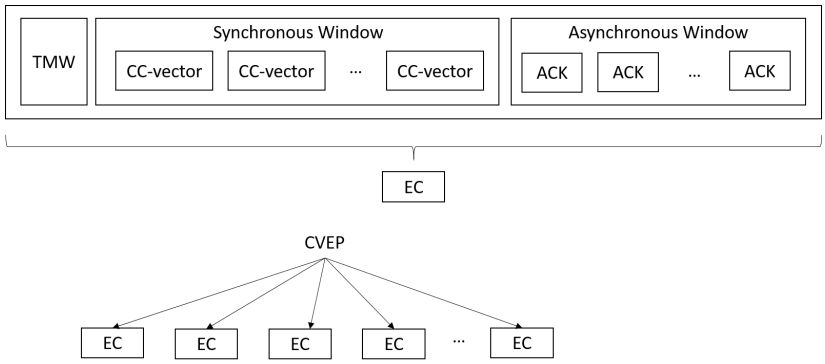


FIGURE 5.4: CVEP

includes a highly-reliable mechanism for node replicas to exchange their cc-vectors. We call this mechanism *Cc-vector Exchange Protocol* (CVEP) (Derasevic, Barranco, and Proenza, 2014).

The CVEP is a proactive retransmission protocol carried out to tolerate transient faults, affecting the nodes or the links, that prevent node replicas from either correctly transmitting their cc-vectors to at least one non-faulty switch replica, or correctly receiving the cc-vectors of the other node replicas from at least one non-faulty switch replica.

Generally speaking, on the one hand, in CVEP each cc-vector is pro-actively retransmitted multiple times within the same EC, taking advantage of the pro-active retransmission service already provided by FTTRS (Gessner, 2017). On the other hand, in CVEP this EC in which the cc-vectors are exchanged is consecutively repeated several times to further increase the chances of their successful exchange (see Figure 5.4)

The communication round that results from proactively retransmitting cc-vectors within each EC of a set of several consecutive ECs is called *Voting Communication Round* (VCR).

CVEP details

In each EC of the VCR each node replica pro-actively sends multiple copies of its cc-vector to each one of the switches during the synchronous window. As soon as a switch receives the first copy of the cc-vector from a given replica in the VCR, it keeps it and considers that copy as the *legitimated cc-vector* from that replica for the current VCR. On the one hand this means that, from then on within each EC of the current VCR, the switch pro-actively forwards a given number of copies of this legitimated cc-vector to the other node replicas. Note that the switch forwards this number of copies in the synchronous window of each EC, including the one in which the switch successfully receives the first copy of the cc-vector. On the other hand, it means that the switch will discard any further copy of the cc-vector it receives from that node replica during the rest of the VCR. Also, note that the switches do not need to be aware of the content of the cc-vector messages. They only need to store, replicate, and forward them. Otherwise, the communication channel would need to be application-aware and one of our goals is to keep the communication channel application-agnostic. We will come back to this topic again later in this section when talking about fault diagnosis.

When a node replica receives a cc-vector from another replica through any of the switches, it reacts by pro-actively retransmitting during the VCR an acknowledgment (ACK) to confirm the successful reception of that cc-vector. More specifically, the node replica pro-actively retransmits a given number of copies of the ACK during the asynchronous window of each EC of the current VCR, including the first EC in which it successfully received the cc-vector.

Note that, during the CVEP, each node replica stores the first copy of the cc-vector it receives from each one of the other node replicas; discarding the rest of the cc-vector copies. For instance, if we consider 3 node replicas called A, B and C, then the node replica A stores the first cc-vector copy it receives from node replica B and the first cc-vector copy it receives from node replica C. When the VCR ends, each node replica delivers the cc-vector copies it stored to its application, which uses them in the voting that follows as will be explained in the next section.

At this point it is important to highlight that the switches neither store nor forward the ACKs. Instead the switches use the ACKs to gather information concerning what cc-vectors are successfully confirmed by the node replicas in the current VCR. As it will be explained later in Section 5.3.4, this information regarding the ACKs is used for implementing fault-diagnosis mechanisms.

Before continuing, let us to briefly come back again to the strategy that switches follow to forward cc-vectors during the VCR. The reason why each switch forwards multiple copies of just the legitimated cc-vector received from each node replica, and drops any further cc-vector copy it receives from the nodes replicas, is twofold. The first reason is that this strategy prevents the CVEP from causing the replica radiation phenomenon. In this sense, please recall from Section 3.2.2 that we decided to design new pro-active retransmission mechanisms in such a way that they do not provoke this phenomenon. The second reason is that this strategy can increase the reliability of the retransmissions. On the one hand, the switches can pro-actively retransmit the legitimate cc-vector of a node replica even if that node replica fails before the VCR ends. On the other hand, in such a way switches prevent any node replica from exhibiting a two-faced behaviour as regards the retransmission of its own cc-vectors. In other words, if a node replica is the one in charge of retransmitting its cc-vector, it may fail by retransmitting cc-vector versions different from the one it originally sent (its legitimate one). If this happens, different non-faulty node replicas might end up the VCR with different versions of the cc-vector of the faulty node replica; thus having an inconsistent set of cc-vectors for voting, which ultimately may lead them to lose replica determinism.

In order to better understand how the two-faced behaviours can occur and be prevented by the switches by keeping and forwarding the first received cc-vector replica, consider the following example. If the switches do not locally keep and retransmit the cc-vectors, the following scenario might occur. As shown in Figure 5.5, replicas A and B receive a correct cc-vector C_1 from replica C in the first EC of the VCR, $EC1$. In the following EC of the VCR, $EC2$, replica A fails to receive all the cc-vectors send from replicas B and C due to too many transient faults in the channel. Simultaneously, replica C becomes faulty as well and sends an incorrect cc-vector C_2 to replica B. At the end of the VCR, the non-faulty replicas A and B will have cc-vectors with different values received from replica C, and this leads to replica non-determinism. Note that even though cc-vectors from replica B, B_1 and B_2, are different in non-faulty replicas A and B, the values are the same since there were no faults in these replicas. Therefore, as was just explained, we need to use the switches when retransmitting cc-vectors. Unlike the node replicas, the switches are reliable due to the enforced crash computation failure semantics (Gessner, Proenza,

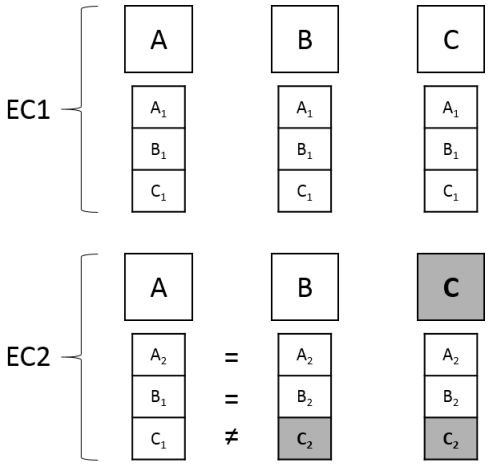


FIGURE 5.5: cc-vector exchange without switches

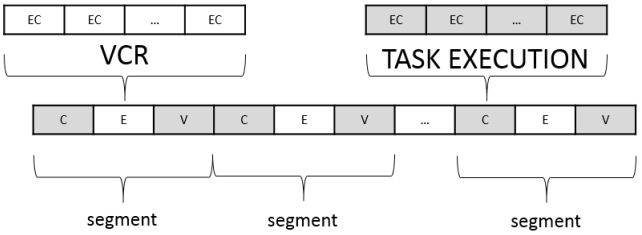


FIGURE 5.6: DCMV complemented with CVEP

and Barranco, 2014a; Gessner, 2017) and will either provide a correct service or crash. Therefore, we can rely on them to correctly perform the retransmissions.

To conclude the current Section Figure 5.6 shows how an application in our architecture is executed due to the introduction of the just explained CVEP. Each white square marked as 'E' represents a VCR round where the cc-vectors are exchanged in multiple ECs following the CVEP protocol. Gray squares, 'C' and 'V', represent the calculation and voting tasks executed by the node replicas, respectively. Each task, depending on its duration, can take arbitrary number of ECs to execute. As already explained in Section 5.1 these tasks are triggered by the TMs. Thus, what we have just presented in Figure 5.6 is a general solution that can be applied to any application. However, depending on a specific application, the number and distribution of tasks and VCRs can be arbitrary. In Chapter 6 we will show how the application tasks and VCRs are distributed in the case of control applications.

Furthermore, it can happen that the data produced by a node replica (cc-vector) is large and does not fit in a single EC. In case of large messages, the HaRTES protocol used by the FTTRS provides a way to automatically fragment them in a sequence of packets where each packet is scheduled sequentially and individually. Therefore, to accommodate for this scenario, we can first exchange all the fragments following the CVEP strategy, then, we can compose the cc-vectors and continue with the voting phase. This solution would result in the extension of the VCR by multiplying it by the number of fragments since we have to exchange all the fragments in multiple ECs, same as we did with the non-fragmented cc-vectors.

Note that by the previously defined MFA only a minority of node replicas can fail at any time. However, with the introduction of our CVEP we can tolerate more than our MFA in some specific cases, e.g., if all node replicas fail by not receiving/transmitting cc-vectors in the first EC of the VCR, this will be dealt with by resending

all the cc-vectors in the second EC of the VCR, and in this case even faults violating MFA will be tolerated.

Moreover, note that the CVEP can also tolerate TLLFL to some extent. Specifically, if TLLFL last longer than a single EC but shorter than the CVEP, the message losses will be tolerated if they are successfully resent in one of the ECs of the VCR.

5.3.2 Forward Error recovery

As already explained in Section 2.1, *Forward Error Recovery* (FER) is done by replacing an erroneous state by an error-free state. In the case of our system we consider that a given node replica can recover from an error provoked by a fault (happening in the channel or in its internal circuitry) by using FER, when that fault is transient and only corrupts data in a segment that, then, can be corrected with the consensus cc-vector node replicas obtain when voting at the end of that segment. What kind of data can be corrected in such a way depends on the application. For instance, in a control application these data can be an incorrectly acquired sensor value, an incorrectly calculated actuation value, an incorrectly calculated intermediate result, etc.

More specifically, each replica performs FER in any given segment by using the consensus cc-vector obtained by the Distributed Consensus Majority Voting Mechanism (DCMV). The following example illustrates this. One replica produces an erroneous output (consensus cc-vector) due to transient faults in a segment i. Since this is an input to the next segment (c.f. Figure 5.2), $i + 1$, the produced cc-vector of the segment $i + 1$ will also be erroneous. However, the faulty replica can seamlessly recover from the erroneous cc-vectors in a segment $i + 1$ by using the cc-vectors received and voted upon from the non-faulty replicas to correctly calculate the next output (consensus cc-vector) of the segment $i + 1$.

It is important to clarify that this FER is not a novel mechanism since using a consensus cc-vector to recover, as just explained, was already proposed in NVP.

In any case, note that the duration of transient faults and their effects can vary. Transient faults can affect one or multiple ECs, and/or even one or multiple segments. Depending on this duration, it can happen that the just-mentioned simple FER will not suffice. In those cases, a more elaborated recovery procedure is needed. From now on we will refer this more complex recovery procedure to as *Reintegration*, and we will discuss it in detail in Section 5.3.3.

Finally, note that the DCMV can be used not only to perform FER and reintegration, but also as a way to provide node replicas that suffer from faults with the basis for being able to carry out self fault diagnosis with regards to their locally produced cc-vectors. Particularly, if a node replica produces an erroneous cc-vector, this cc-vector will be different from the one obtained by the majority voting function. Such a situation can then serve as an indication that the node replica was faulty. Section 5.3.4 shows how we use this idea to provide one of the fault-diagnosis mechanisms therein presented.

5.3.3 Reintegration

As mentioned before, a fault may affect a replica in such a way that goes beyond that replica's FER capacity. When this happens, the replica affected by the fault can become desynchronized with respect to the non-faulty ones at the communication and/or the application level.

In order to cope with this problem, in the current section we propose two reintegration mechanisms. One of them allows a faulty replica to resynchronize at the communication level, whereas the other one allows it to resynchronize from the point of view of the application.

But before continuing, it is necessary to clarify what does it mean to be desynchronized at the level of the communication and to be desynchronized at the level of the application. First, a node replica is desynchronized at the communication level when it disagrees with the non-faulty ones about what is the current EC. Second, to understand when a node replica is desynchronized at the application level, it is necessary to note that when a node replica calculates its cc-vector, it has to consider certain input variables. These variables, depending on the application, apart from the consensus cc-vector from the previous segment, also include some locally calculated and stored variables not being exchanged among node replicas. Together, we will refer to all of these variables that include the whole state of the computation as the *operational state* of a replica. Having this in mind, we say that a node replica is in synchrony at the application level with the other non-faulty node replicas if it has the same operational state as them.

Coming back to the point at issue, the first reintegration mechanism we propose is called *TM resynchronization*.

To better understand how this mechanism works, please recall that at the beginning of this chapter we have introduced the idea of *network-centric* approach to coordinate the execution of all the system activities by using the TMs. This approach was based on previous works (Calha and Fonseca, 2002; Almeida, Pedreiras, and Fonseca, 2002) that modified the TM content in order to, besides triggering message exchange among nodes, also trigger the execution of tasks in the nodes.

However, the approach used by this dissertation is to use the same idea but not to modify the TM content in order to trigger tasks in the node replicas (Derasevic, Proenza, and Barranco, 2014). The main reason is to avoid the network subsystem being aware of the application executed on top of it. Since each TM produced by the switches conveys a *Trigger Message Sequence Number* (TMSN) that gets incremented for every new EC, we have decided to use this information in the node replicas. Thus, each node replica implements a local TMSN that, on the reception of each TM, gets overwritten by the one conveyed in the TM.

Having the above in mind, we have to define in advance a set of TMSN values (specific ECs) when the node replicas should start with the execution of specific tasks. By using this design, the execution times of application tasks have to be expressed in the number of ECs and each task can then be scheduled for triggering at the beginning of a certain EC. Consequently, each node maintains a lookup table where each row specifies a TMSN value and a task to be executed, which is then used by the node's application. Each time a new TM gets received by the node's application, it has to compare the TMSN value conveyed within with the TMSN values stored in the lookup table to see if there is any task to be triggered.

In our replicated group of nodes, as long as there are no faults, node replicas will receive TMs promptly and for each node replica, the locally stored TMSN and the TMSN conveyed in the received TMs will differ by 1. If this is the case, all the replicas are in synchrony from the communication point of view.

However, if, due to transient faults that last longer than the duration of the TMW of an EC, classified as TLLFL, a node replica misses to receive TMs for some period of time, it might miss to trigger some tasks and as a result it might become desynchronized with the other node replicas on both communication level and application level. As soon as this replica receives the first TM it can, by inspecting the difference

between local TMSN and the just received TMSN conveyed in the TM, determine that it was out of synchrony with others and for how long. Then, depending on the duration, it can take appropriate actions to continue with the correct execution. As regards the TMSN, replica resynchronizes with others at the communication level just by copying the TMSN value from the received TM. As regards the loss of synchrony at the application level, the faulty replica needs to take further recovery actions which are explained next.

In some cases, the FER provided by the DCMV will suffice to resynchronize a faulty replica at the application level, i.e. it will make the operational state of a faulty replica consistent with the non-faulty node replicas. However, in some situations, this will not be the case and a more sophisticated recovery mechanism is needed to restore synchrony of a faulty node replica at the application level. For these latest and less benign situations we propose our second reintegration mechanism, i.e. the *Voting Reintegration Point*.

We base the Voting Reintegration Point on the idea of *Recovery Point* proposed in (Avizienis, 1985). A recovery point consists in exchanging the whole state of the computation among all versions, equivalent to our node replicas, to ensure that each of them have a consistent state afterwards.

In order to achieve this, in the calculation of cc-vectors each replica has to consider exactly the same input variables. These variables have to include the whole operational state.

As a solution to recover the faulty replicas at the application level, we propose to include in the cc-vectors all the variables that constitute the operational state of a replica. Therefore, when a faulty replica receives cc-vectors from the non-faulty ones, it can use them to restore its operational state. It does this by extracting all the variables, voting on each of them, and using them as an input.

The Voting Reintegration Point is done pro-actively, i.e. we always include in the cc-vectors all the information needed to restore operational state of a faulty replica regardless whether there was a fault or not. Otherwise, the detection and postponed sending of operational state variables would require some time and it would prolong the time to reintegrate, which can be crucial in critical systems since more node replicas can fail in the meantime. As a result, this can increase chances of system failure and decrease the system reliability.

The Voting Reintegration Point resynchronizes the faulty replicas at application level, or in other words, it restores replica determinism of a faulty replica with regards to their operational state. We say that a replica is replica determinate with the others if, at the end of a given segment, it is able to produce the same consensus cc-vectors as the others, i.e. it reaches the same operational state.

It also has to be noted that the Voting Reintegration Point can be applied only for the applications whose operational state can be exchanged in the VCR. If the overall size of all the exchanged variables is huge, this can prolong the VCR duration to the point in which the application cannot meet its deadlines anymore.

However, since most of the RT applications are control ones and the data constituting operational state is reasonably small, the proposed solution is feasible. Moreover, since we use Ethernet, and it supports extensive bandwidths, we are able to transfer a lot of data within our EC.

For illustrative purposes we shall give an example of how much bandwidth would be used by all the messages being exchanged in each EC of the VCR in case of 3 node replicas.

We assume that the cc-vector data and the ACK data will be able to fit within the smallest-size Ethernet frame. The size of the smallest Ethernet packet (Ethernet

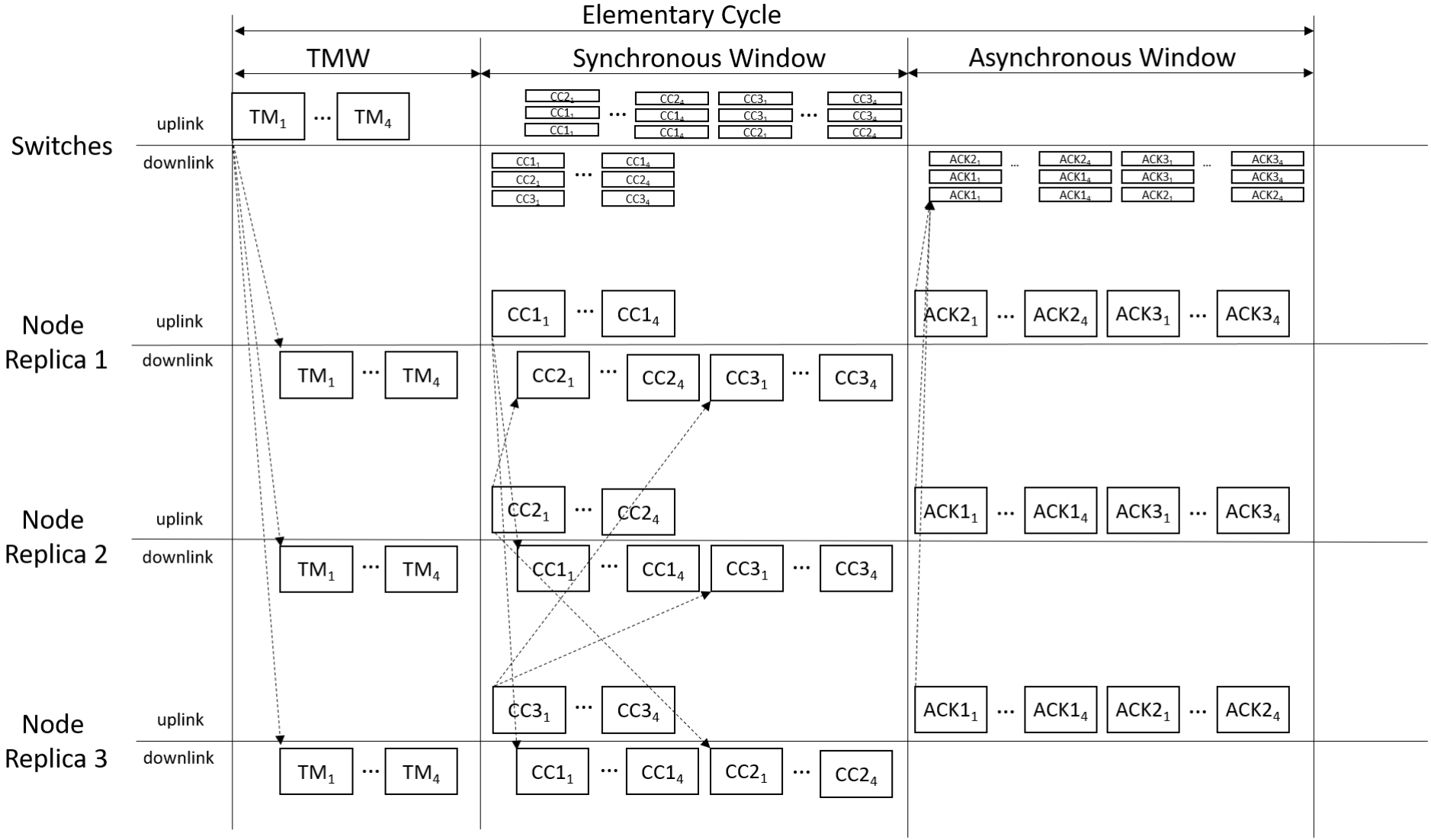


FIGURE 5.7: Message exchange illustration

frame and Inter packet gap) is 84 Bytes. The payload occupies 46 Bytes. As regards the FTT, for synchronous and asynchronous FTT messages, FTT-related fields take 9 Bytes, so there is 37 Bytes left for the actual payload. Therefore, we assume that the cc-vector and ACK data will fit in the remaining 37 Bytes, if not, either larger Ethernet frames or the FTT fragmentation can be used.

The Ethernet bandwidth assumed for this illustration is $100Mbps$. Since the duration of EC is $1ms$, the number of Bytes (B) that can be transmitted in a single EC is calculated as follows: $100Mbps * 1ms = 12500B$.

For the number of message replicas, we chose 4 TM replicas, 4 cc-vector replicas and 4 ACK replicas per each node replica in an EC (Chapter 9 gives rationale behind these numbers). Figure 5.7 depicts the traffic exchange in one uplink and downlink replica of all the components. Note that the other link replicas' traffic is equivalent. Let us observe now what happens in the downlinks and uplinks so we can estimate bandwidth consumption.

First, in the TMW, 4 TM replicas, $TM_1...TM_4$, are sent by the switches (uplink) and received by each node replica (downlink). Second, in the synchronous window, 4 cc-vector replicas are transmitted by each node replica (uplink) to other two node replicas which in turn receive 8 cc-vector replicas (downlink), 4 from each of the remaining two node replicas. In particular, node replica i transmits cc-vector replicas, $CCi_1...CCi_4$, to the switches which in turn locally keep and then retransmit the same number of copies according to the CVEP execution pattern to the other two node replicas. Note that the arrows in the synchronous window shown in Figure 5.7 depict the relations for cc-vector exchange between the node replicas excluding the switches in order to simplify the image. Last, in the asynchronous window, each node replica transmits 8 ACK replicas (uplink) to the switches (downlink) to confirm the successful reception of the cc-vectors received from the other two node replicas.

In order to facilitate the bandwidth consumption estimation for this illustration we shall make the following pessimistic assumption. The bandwidth consumption is the worst case transmission time (WCTT) of both uplinks and downlinks for each node replica link, i.e. if a messages are transmitted through uplinks and b through downlinks, and $a > b$, then, the WCTT is a.

Therefore, as seen in Figure 5.7, in the TMW, the WCTT is 4 Ethernet packets corresponding to the 4 TM replicas received. In the synchronous window, the WCTT is 8 Ethernet packets corresponding to the 8 cc-vector replicas received, and finally, in the asynchronous window, the WCTT is 8 Ethernet packets corresponding to the 8 ACK replicas transmitted.

Thus, the summed WCTT is $4+8+8 = 20$ Ethernet frames. We have to add to this number the delays and time gaps between the TM replicas. So, let us assume that they attribute for another 5 Ethernet frames. Thus, is total, we use $25 * 84B = 2100B$ which is 16.8% of the total number of Bytes available.

Remember that reintegration was introduced as a means to deal with transient faults that affect a node replica in such a way that it becomes desynchronized with the other node replicas either at communication or application level, but is still recoverable by using the aforementioned mechanisms.

Still, some transient faults in the node replicas may exhibit more severe effects and can manifest as permanent ones (TFNPs). If not dealt with, TFNPs can lead to quick attrition or redundancy achieved by the applied active replication of nodes. To recover from these faults, the affected replica needs to be reinitialized by means of a reset. Once a replica is reset, it can resynchronize with others by using the above described reintegration mechanisms.

As already mentioned, these faults cannot be distinguished from real permanent faults since they produce the same effects. Thus, when we diagnose a replica as permanently faulty, we first try to reset and reintegrate it a predefined number of times. If this does not yield any success, we permanently disconnect it from the others.

In the next section we describe the mechanisms used to diagnose a replica as permanently faulty and issue a reset.

5.3.4 Fault Diagnosis

As regards node faults it is difficult to differentiate when they are transient, permanent, or transient but causing the node to behave as permanently faulty (TFNP faults).

Fortunately, transient node faults that do not manifest as permanent are merely tolerated by means of the FT mechanisms already explained. However, when a transient node fault manifests as permanent, it cannot be tolerated by means of these mechanisms, nor it can be differentiated a priori from a permanent one.

For these two later cases our approach is thus to reinitialize a node that seems to be permanently faulty and try to recover it. If the fault is actually transient, the recovery will be successful. This is because the reintegration mechanisms will allow the replica to resynchronize with the rest of the replicas both at the level of the communication and the application. Conversely, if the fault is permanent, the recovery will not succeed and the faulty node will be reinitialized again. To prevent a truly permanently faulty node to keep on reinitializing, we shall disconnect it after a predefined number of reinitialization attempts.

In any case, to apply this strategy of resume and recovery, it is necessary to provide the system with mechanisms to diagnose replicas as permanently faulty and reinitialize them by means of a reset.

The first mechanism is *Discrepancy Error Counter* (DEC). The purpose of DEC is to detect discrepancies in the votings performed by the node replicas that occur due to internal replica faults affecting their ability to either produce the correct voting values of to perform the voting correctly.

Specifically, if faults affect a node replica's ability to produce the correct inputs for the voting function, the local values produced by a node replica will not correspond to the consensus value obtained by a majority voting. On the other hand, if locally produced values are correct but the majority voting carried out by this replica is affected by faults, then the local majority voting value (consensus) might not be obtained or might be an incorrect one. In both cases this can be diagnosed as a discrepancy.

Depending on the comparison function used by the majority voting and the values being compared, correspondence can be defined differently. One example would be that the input values for the votings are the real numbers obtained from reading the sensor values. Due to the *real world abstraction limitation* defined in Section 2.2 the input values can be declared as corresponding even if there are slight differences. A value can be considered correct if it falls in between the predefined bounds. In this case, the majority voting function can be defined as bounded average, i.e. the average is applied only to inputs which values are within the predefined bounds. Another example would be that the input values for the votings are booleans, and in this case, the correspondence is defined as an exact match between the values.

To detect a discrepancy defined above each replica maintains an error counter data structure called DEC. In case a discrepancy is detected after the votings, the

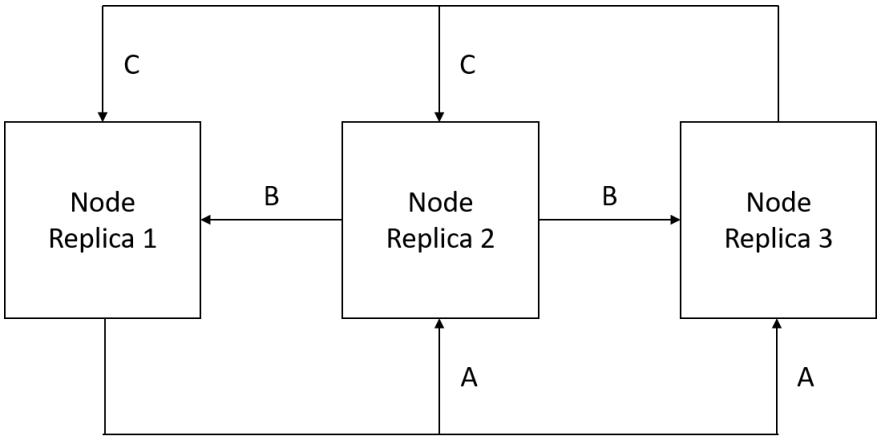


FIGURE 5.8: CC-vector exchange between 3 node replicas

DEC is incremented by a modifiable increment value, and in case no discrepancy is detected, the DEC is decremented by a fixed decrement value. When the DEC reaches a predefined threshold, a replica is diagnosed as permanently faulty, and it will issue a reset. If multiple consecutive discrepancies are detected, the increment value will be increased after each discrepancy. This is done to increase the speed of permanent fault detection since this is the expected behaviour in case there is a permanent fault.

The second mechanism is *Communication Error Counter* (CEC). The purpose of CEC is to detect communication errors caused by internal replica faults affecting its ability to transmit/receive messages.

To help replicas detect communication faults, we implement inside the switches *Messages Status* (MS) vector. MS vector is a matrix that gets populated by the switches during a VCR and gets delivered to node replicas in the EC following the last EC of the VCR. Specifically, the switches piggyback the MS vector within the TM, so as to the make the transmission of this vector highly reliable. As will be explained, this vector contains the information that help replicas decide if they suffered from communication faults.

We shall demonstrate how the MS vector looks like and how it gets populated for an example case of 3 node replicas. Figure 5.8 depicts how 3 node replicas exchange their cc-vectors. As seen in the Figure, node replica 1 produces and transmits cc-vector A and receives cc-vectors B and C, while node replicas 2 and 3 produce and transmit cc-vectors B and C and receive A and C, and A and B respectively.

The MS vector corresponding to these node replicas is depicted in Figure 5.9. There are $3X3$ cells, each having one of 2 possible values, $true(T)$ or $false(F)$. The first row corresponds to replica 1. First cell (cell 1-A) specifies if the replica 1 has successfully transmitted its own cc-vector A to the switches. If so, its value will be set to T, and if not, to F. The other two cells specify if the replica 1 has successfully received cc-vectors from replicas 2 (cell 1-B) and 3 (cell 1-C) and then transmitted the corresponding ACKs successfully. The second row corresponds to replica 2. Second cell (cell 2-B) specifies if the replica 2 has successfully transmitted its own cc-vector B to the switches. The other two cells specify it the replica 2 has successfully received and acknowledged cc-vectors from replicas 1 (cell 2-A) and 3 (cell 2-C). The meaning of the last row is analogous to the above 2.

If all the cc-vectors get successfully exchanged and acknowledged in a VCR, all the cells will be set to T. If some cell has a F value, we have to interpret it and

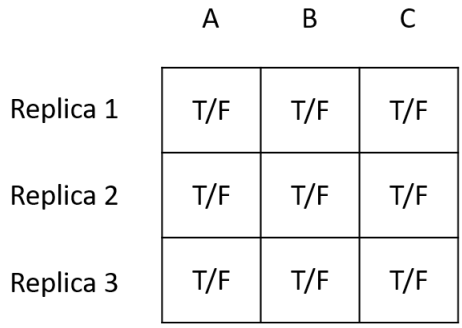


FIGURE 5.9: MS vector

decide which replica was communication faulty, i.e. which replica failed to transmit/receive.

We distinguish between two classes of cells and interpret them differently.

First, diagonal cells (1-A, 2-B, 3-C) specify if replicas managed to transmit their cc-vectors to the switches. If some value is F, the corresponding replica is said to be transmission faulty in that VCR.

Second, non-diagonal cells specify if replicas managed to receive and acknowledge cc-vectors from the others. However, if some non-diagonal value is F, we cannot immediately conclude that the replica corresponding to the row in which the value was located was communication faulty in that VCR. It could have happened that the replica transmitting the cc-vector was transmission faulty. Therefore, when a non-diagonal value is F, first we have to inspect diagonal value corresponding to that cc-vector (column). If the diagonal value is F, we can conclude that the transmitting replica was transmission faulty, since the switch did not receive the message and did not forward it to the receiving replica. But, if the diagonal value is T, we can conclude that the receiving replica was communication faulty, since the switch did receive the message and did forward it to the receiving replica which in turn failed to receive or acknowledge it.

At this point, and before continuing, it is important to note that, due to channel faults, it could be possible that by the end of the VCR both switches have not received the same ACK messages from the slaves and, thus, each master may come up with a different MS vector. Since masters must be replica determinate, it is important to provide them with a mechanism to agree on the content of their MS vectors. In this sense what we propose is to force both masters to reconcile the content of their MS vectors and then provide slaves with a consistent view of that vector. For this, at the end of the last EC of the VCR, both masters must both exchange with each other their respective MS vectors several times and, then, locally carry out a logic OR between each pair of corresponding cells of the MS vectors. In this way they will obtain a consensus MS vector that then they can broadcast to the slaves. A similar idea is proposed in (Gessner, 2017) to enforce an agreement between the FTT masters on the update requests.

It is also noteworthy that, originally, the MS vector was used by each node locally to perform *Voting Set-Up Algorithm* (VSUA) proposed in previous work (Derasevic, Barranco, and Proenza, 2014). The goal of VSUA was to decide which node replicas and which cc-vectors should be used by each majority voting in order to provide a consistent voting in each node replica. The VSUA worked by inspecting the MS vector and then finding the combination of node replicas and cc-vectors that satisfy the rule: at least a majority of cc-vectors in at least a majority of node replicas. The

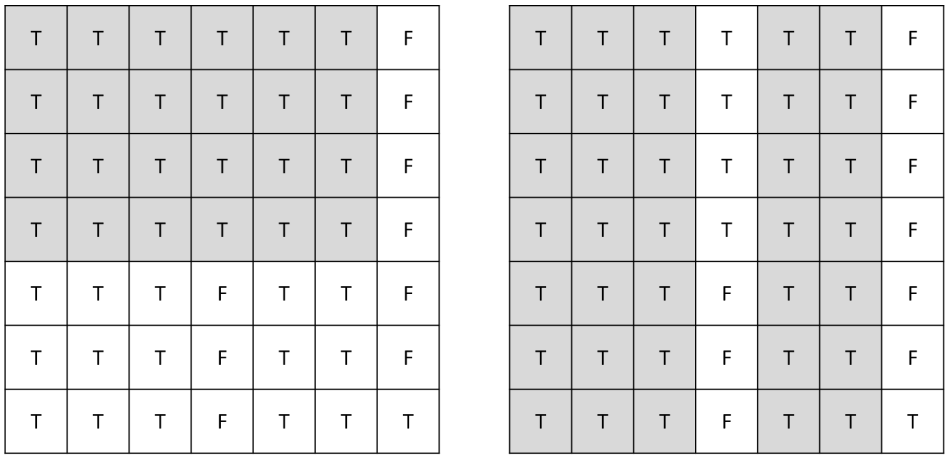


FIGURE 5.10: VSUA conflict example

VSUA always maximized the number of cc-vectors first and then the number of node replicas that have these cc-vectors. The advantage of the VSUA is its ability to make a decision even in the cases when the MFA is violated. But, this algorithm was abandoned due to inability to determine the correct combination of node replicas and cc-vectors to take into consideration for voting in some particular cases. E.g., in a case of MS vector depicted in Figure 5.10 the combination of replicas and cc-vectors that the VSUA would chose would be the gray area on the left-hand side, i.e. vote with first 4 replicas and first 6 cc-vectors because the VSUA maximizes the number of cc-vectors. However, note that there is another valid choice on the right-hand side of Figure 5.10 with only one cc-vector less that would allow all the replicas to vote.

Due to the aforementioned issues, in this dissertation we have decided that, instead of using VSUA, all the node replicas vote with whichever cc-vectors they have, and if MFA is fulfilled, voting will be successful. The MS vector is still being used, but for fault diagnosis purposes only.

When replicas receive the MS vector they can conclude if they were communication faulty in a VCR. If a replica was communication faulty in a VCR, its CEC gets incremented, and if not, its CEC gets decremented. Similarly like with DEC, we shall penalize consecutive communication faults by increasing the increment value. When the CEC reaches a predefined threshold, a replica is diagnosed as permanently faulty and it will issue a reset.

Since the nodes replicas can fail arbitrarily, there is no guarantee that the above error counter mechanisms (DEC and CEC) will work in the presence of node faults and force a reset. Therefore, the ideal scenario would be that we also implement these mechanisms in the switches since they can be relied upon due to their crash failure semantics. Then, if a faulty node fails to detect permanent fault and reset by itself, switches can force it to reset.

However, we cannot implement DEC in the switches because then they would also have to vote on the application variables contained within cc-vectors received from the node replicas in order to detect discrepancies. This would mean that the switches would have to be application-dependent and our goal is to keep the FT-TRS unaware of the application/s executed by the nodes. On the other hand, CEC can be implemented since the switches already populate MS vector and have the knowledge of node communication faults. If the CEC mechanism implemented in the switches detects a permanently faulty replica, the switches send to that replica a reset command. Note that a faulty replica would have already been reset by itself if

its CEC functioned properly and detected the fault first.

Note that the node replicas can even disobey the reset commands coming from the switches in case of more severe node failures. In this case we devise the mechanism described below.

The last mechanism implemented by the node replicas to detect permanent node faults and issue a reset is *You Are Alive* (YAA) watchdog timer. This mechanism was introduced as a final step that would guarantee that a faulty replica that is unable to issue a reset by itself, or is unable to obey the commands from the switches, gets reset. This usually happens when a replica crashes.

This device starts counting from a pre-established time until it reaches 0. When a timer reaches 0, a permanent fault is diagnosed and a hard-induced reset is issued.

The timer itself resets to a pre-established time upon each reception of YAA message conveyed within each TM and starts its countdown from the beginning. Thus, as long as the TMs and YAA messages conveyed within get received, a timer restarts. The YAA message has to traverse the path from the switches, through a replica, and to the YAA watchdog timer device. If a replica crashes, the timer will not be restarted and will expire. This will result in a hard-induced reset of a crashed replica.

Moreover, as concerns the YAA message itself, in order to prevent a faulty replica from forging it, the switches include within each YAA message a code based on the TMSN that only the switches and the YAA watchdog timer know. The YAA watchdog timer validates each YAA message according to this code, and if that code is not correct, then the YAA message is being ignored and the timer is not reset.

The architecture of the watchdog timer, as well as how it connects and communicates with the node replica are out of the scope of this work. In any case, it is important to guarantee that the watchdog should be external to the node replica and should not share common sources with it, e.g. clock sources, in order to prevent spatial-proximity and common-mode failures. As concerns its complexity, the watchdog timer should be quite simple. For instance, it can be synthesized within a small FPGA, and can connect to the node replica by means of a *General Purpose Input/Output* (GPIO) interface. This FPGA could synthesize a timer, a buffer to store the YAA message, a small ROM that stores a look-up table to calculate the code based on the TMSN, and an automaton that controls the watchdog timer actions.

5.4 Overview of the applied FT mechanisms

In this section we present an overview of all the FT mechanism applied for the classification of faults introduced previously in Section 5.2, as can be seen in Table 5.2.

Permanent and transient switch faults are tolerated by using the active and semi-active replication of switches. Remember that the transient switch faults were transformed into permanent ones due to enforced crash failure semantics. Permanent link faults are tolerated by means of replication (duplication) of links. All these mechanisms were described previously in Section 3.2.3.

Permanent node faults are handled by error compensation by means of active node replication and DCMV described in this chapter.

TLLFL are handled by different mechanisms depending on the exhibited duration and the affected ECs. TM resynchronization mechanism is always needed as we assume that these faults last longer than at least one EC in which case the affected replicas need to apply this mechanism. When exchanging cc-vectors, if these faults last less than one VCR, then all the cc-vector will be exchanged successfully by CVEP. However, if the duration exceeds the length of a VCR, depending on the cc-vectors

TABLE 5.2: Applied fault tolerance mechanisms according to persistence of faults

	Switch	Node	Link
Permanent	replication	replication & DCMV	replication
TLLFL			TM resynchronization CVEP replication & DCMV reintegration
TFNP		fault diagnosis reset reintegration	
Transient	replication	all	TM retransmission CVEP

lost in the process there can either be enough cc-vector received for successful voting or not. If there is, nothing has to be done as the applied active replication and DCMV will compensate the errors, and if there is not, the faulty replicas will have to be reintegrated after the next VCR. If TLLFLs affect ECs in which the triggering of the tasks that calculate the variables of operational state occurs, these tasks will not be triggered and as a result the operational state will not be calculated correctly. Therefore, the affected replicas will have to be reintegrated after the next VCR.

TNFP are handled by first diagnosing a permanent fault of a node replica followed by its reset and reintegration.

Transient faults in the nodes can manifest in many different ways. Depending on the manifestation, all of the FT mechanisms come into play to tolerate them. E.g. if a transient fault in a node manifests in such a way that a node replica only fails to receive TMs for a certain period of time, TM Resynchronization mechanism will handle the fault, or if the manifestation causes a node not to calculate or to miscalculate some of the operational state variables, a complete reintegration is needed, etc.

Transient links faults are handled by proactive retransmission of critical messages. TMs are retransmitted per-EC basis as a part of the FTTRS FT mechanisms and cc-vectors are retransmitted per-EC and then this EC is repeated multiple times per-VCR basis as described earlier when talking about the DCMV and CVEP.

Chapter 6

Realization of the proposed Active Node Replication for the case of control applications

In this chapter we describe how to deploy the most commonly used type of applications for Embedded Systems, control applications, in our fault-tolerant system architecture. These applications and their deployment will be used in the rest of the dissertation.

Figure 6.1 depicts the deployment of control applications. We assume that the controller nodes of the control application are the most complex ones. Therefore, these nodes use the aforementioned active node replication and DCMV and are interconnected by the FTTRS communication subsystem. On the other hand, the instrumentation of the plant, sensors and actuators, are less complex, and other, more simpler, FT mechanisms can be applied. Also, connecting these devices with controller nodes can be done via FTTRS, via other communication network, via dedicated links, etc. The FT of sensors and actuators and the way to connect them to controller nodes is beyond the scope of this dissertation. However, this issue was already covered by earlier works that describe, fist, the methodology of how to tolerate sensor values (Marzullo, 1990), and second, the *output consolidation* technique for actuators FT (Powell et al., 1999) that can easily be applied to our architecture.

A typical control application periodically repeats 3 phases: *sense*, *control*, and *actuate*. The period with which these phases are repeated is called *sampling period*. Due to the introduction of our FT mechanisms in general, and DCMV and CVEP in particular, in our fault-tolerant architecture, control applications now have 7 phases (see Figure 6.2), henceforth *Extended Control Application Phases* (ECAC):

- *Sense* (S). Each node replica obtains the value(s) measured by the sensor(s). The obtained value(s) shows the present state of the plant.
- *Message Exchange of Sensor Values* (ESV). Each node replica populates its cc-vector with the obtained sensor value(s) and all the operational state values. Node replicas then exchange their cc-vectors in a VCR.
- *Voting on Sensor values* (VS). Each node replica performs multiple voting procedures, one for each of the values conveyed in the cc-vectors. Each voting is done with the values received from the other replicas and the locally obtained values. As a result, each replica produces *consensus* values that compensate possible errors.

  Concerning the operational state values, due to the enforced replica determinism (see Section 2.2), each node replica will produce identical values. Thus, for

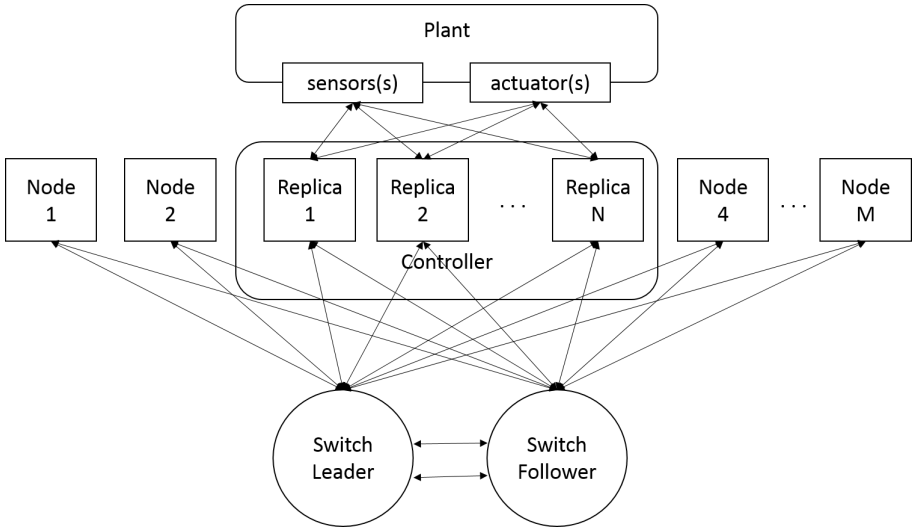


FIGURE 6.1: Control Application Architecture

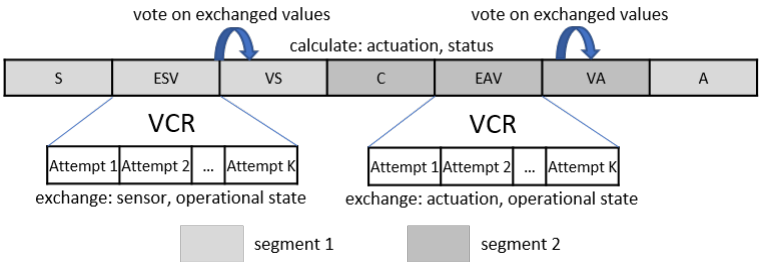


FIGURE 6.2: Control Application Phases

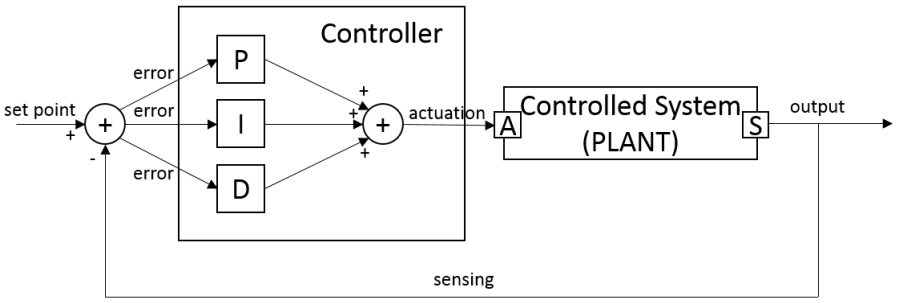


FIGURE 6.3: PID controller

voting procedures applied for these values we always uses *exact match*, where bit-by-bit identical data is expected (Makam, 1982).

As regards the voting on sensor values, multiple scenarios are possible, e.g., if we assume that each node replica is connected to one sensor replica and the measured values are decimal numbers that can have small differences, then the *numeric match* is often applied (Makam, 1982). On the other hand, if in the same scenario sensors produce boolean values, then we can apply the aforementioned exact match. Exact match is also applied if there is only one sensor replica connected to all node replicas. In this case, the value produced by a single sensor will be passed to all of the node replicas and this value is expected to be the same in the absence of faults.

Note that in the case there are multiple sensor replicas that produce decimal numbers, we use numeric match to tolerate the differences in the measured values. In this case there is an additional advantage of using our architecture. When we vote on the sensor values, we are able to compensate the potential sensor errors in addition to tolerating errors produced by the faulty replicas.

- *Control* (C). Each node replica uses the consensus sensor value and the operational state consensus values, and, by means of a control law calculates an actuation value. We consider that a *Proportional-Integral-Derivative* (PID) controller (*The Control Handbook. 1996*) is used as a control algorithm (see Figure 6.3), which is commonly used in industrial control systems.

  A sample code of a PID control algorithm is depicted bellow (Osman, Rahmat, and Ahmad, 2009):

```
previous_error = 0
integral = 0
loop:
error = setpoint - measured_value
integral = integral + error*dt
derivative = (error - previous_error)/dt
output = Kp*error + Ki*integral + Kd*derivative
previous_error = error
wait(dt)
goto loop
```

  The $previous\_error$ and $integral$ variables will be used to calculate *integral* (I) and *derivative* (D) terms of a PID controller. The control law is executed as follows. First, the $error$ is calculated by subtracting the value measured by sensor,

$measured\_value$, and the set point (set_point). Then, the integral and derivative terms are calculated using the $error$ and the precalculated $previous\_error$ and $integral$ variables. The output (actuation value) is calculated by adding *proportional* (P) term and the precalculated integral and derivative terms multiplied by their constants, Kp, Ki, and Kd, respectively. Lastly, $previous\_error$ is updated and the control law repeats.

Consequently, operational state values in the case of a PID controller will be $setpoint$, $integral$ and $previous\_error$ values. These are all the values needed for a controller to always successfully calculate the output (actuation value).

- *Message Exchange of Actuation Values* (EAV). Each node replica populates its cc-vector with the calculated actuation value(s) ($output$) and all the operational state values ($setpoint$, $integral$ and $previous\_error$). Again, node replicas then exchange their cc-vectors in a VCR.

- *Voting on Actuation values* (VA). Again, as in VS, each node replica performs multiple voting procedures, and, as a result, produces *consensus* values that compensate possible errors.

  Now, the cc-vectors contain actuation values instead of sensor values. Recurrently, due to the enforced replica determinism (see Chapter 5), each node replica will produce the identical actuation value and exact match procedure will be applied for voting.

- *Actuate* (A). Each node replica sends its actuation value to the plant's actuator device(s) and the actuator device(s) perform the actuation accordingly. How actuators deal with the multiple reception of actuation values is out of the scope of this work, but typically an *output consolidation* of the received values (Powell et al., 1999) can be done.

It should be noted that is suffices to have only one reintegration performed after the ESV phase for the faulty replicas to successfully recover/reintegrate. The reason for this is that all the operational state variables that the replica uses are used by the control law of the PID executed in the C phase.

However, we have also decided to perform the reintegration after the EAV phase. The reason is that a faulty replica can reintegrate sooner, at the end of the VA phase. As a consequence, if more replicas fail in the meantime, the ones just reintegrated at the end of the VA phase can be used.

Another important observation is that the phases VS and C, and VA and A, which are the task execution phases executed by the node replicas, are executed one after the other. Thus, these phases can be triggered by one value of the TMSN (see Chapter 5) by triggering the first phase only and the second one is handled by the application by starting immediately upon the completion of the first one. The advantage of doing so is the potential reduction of idle time that is introduced by the separate triggering. E.g., if the phase VS lasts 1.1 ECs [1] and phase C lasts 2.2 ECs, and if we trigger them separately, then we have to wait 0.9 ECs after triggering the phase VS until we can trigger the next phase C, because this is the soonest that we can receive a new TM conveying TMSN. Afterwards we would also have to wait additional 0.8 ECs to trigger the following EAV phase. In total, that is 1.7 ECs of idle time. But, if we trigger them jointly, the idle time is reduced to 0.7 ECs, i.e. the time until the start

[1] Recall that all application tasks have to be expressed in the EC duration so that we can use our network-centric approach described in Chapter 5.

of the next EAV phase following these two. Another case is when the joint duration of these tasks is quite small, e.g., 0.7 ECs. Then, if we trigger them separately we would introduce 1.3 ECs of idle time instead of 0.3 ECs.

Chapter 7

Verification and Characterization via Simulation

The objective of this chapter is to verify the correctness and characterize the behavior of the node fault tolerance (FT) mechanisms proposed in this work via simulation.

More specifically, it is important to note that the intention of this chapter is not to calculate the coverage of the node FT mechanisms, i.e. their probability of success. Instead its intention is to exhaustively inject each type of fault that may affect the nodes ability to operate and/or communicate and, then, to corroborate that all our node FT mechanisms worked as intended to tolerate and recover/reintegrate from them. As concerns the characterization of the node FT mechanisms, this chapter is devoted to clarify, for each injected fault, how it impairs the nodes, what FT mechanisms take place, and what is the time needed to recover/reintegrate.

The chapter is organized as follows. We first introduce the chosen simulation framework. Then we explain how we have used this framework to model the particularities of our system and to inject faults. Finally, we present and discuss the results of the set of fault-injection experiments we carried out with this model.

7.1 Description of the simulation model

The simulation of our system was built using OMNeT++, an object-oriented discrete event network simulation framework (Varga, 2001), and INET framework (Varga, 2007) which is an open-source library for OMNeT++ that contains models for wired and wireless link layer protocols.

OMNeT++ is a component-based modeling framework. The models are built by combining hierarchically nested construction units called *modules* programmed in C++. Modules communicate by passing messages to each other. Modules can either be *simple* or *compound*. Simple modules are the lowest level of module hierarchy that encapsulate functionalities programmed by the OMNeT++ simulation classes, i.e. simple modules are active components with associated behaviour. Compound modules are modules that contain other modules, simple and/or compound, and they are used for grouping purposes only. This means they are inactive and have no associated behaviour. In each OMNeT++ model there is only one top level module, called *system module*, that is on top of the hierarchy.

The model of our system builds on a previous model carried out by our research group and which is described in (Knezic, Ballesteros, and Proenza, 2014). Specifically, (Knezic, Ballesteros, and Proenza, 2014) proposes an OMNeT++ simulation model of a distributed embedded system based on a slightly enhanced version of HaRTES (Section 3.1) that includes the mechanisms FTTRS proposes for both, the

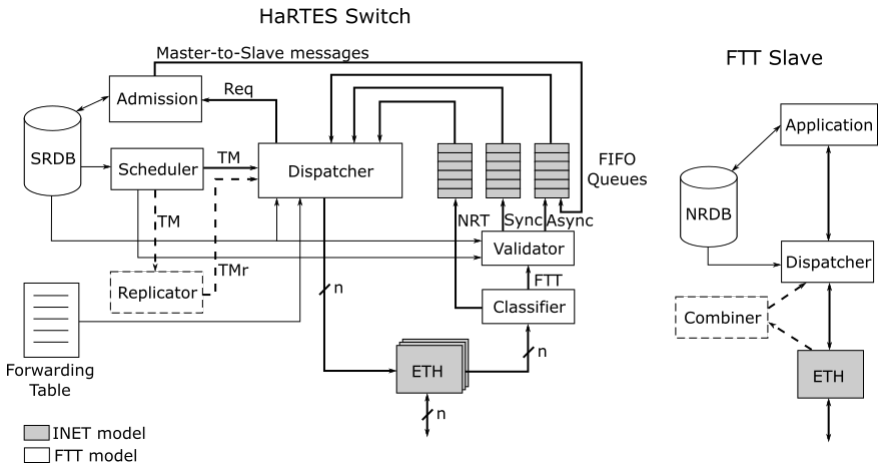


FIGURE 7.1: OMNeT++ model of enhanced HaRTES protocol (source (Knezic, Ballesteros, and Proenza, 2014))

FTT-enabled switch to pro-actively retransmit the TM and for the FTT slaves to synchronize at the EC level taking into account the different TM replicas.

As it will be explained later, we modified and extended this previous model so as to be able to analyze and verify the node FT mechanisms proposed in this dissertation. In this sense it is important to note that we have not extended that model to include any further mechanism of FTTRS. In fact, we could have even simulated our node FT mechanisms on a model of HaRTES that does not include any mechanism of FTTRS. To better understand this point please recall that the mechanisms FTTRS provides are devoted to increase the reliability of the FTT communication services themselves, e.g. it provides redundant communication paths, proactive retransmission of critical messages, reliable and consistent real-time requirement updates, etc., but it does not provide any new communication service our node fault-tolerance mechanisms need to rely on.

Moreover, this decision allowed us to simplify the model itself and to avoid simulating all the FTTRS related mechanisms, which are out of the scope of this dissertation.

Next, and prior to explaining the details of our model, we describe the previous simulation model (Knezic, Ballesteros, and Proenza, 2014). As can be seen in Figure 7.1, this previous model includes the OMNeT++ models of a HaRTES switch and of several FTT compliant nodes (FTT slaves) (please refer to Section 3.1). More specifically, there are three types of OMNeT++ modules depicted in Figure 7.1. First, the gray modules are the ones used from INET library, second, the white modules with solid border model the behaviour of HaRTES protocol described in Section 3.1, and finally, the white modules with dashed border model the TM retransmission mechanism described in Section 3.2.3.

The *HaRTES Switch* compound module is presented of the left-hand side of Figure 7.1. We describe its constituting modules next. The *SRDB* module stores all the traffic management and global configuration information and is initialized at the beginning of simulation from an XML file. The *Admission* module that models the admission control/QoS manager functions is not implemented. It is intended to be used as a part of the future work and in this version of the model it simply drops the received packets. The *Scheduler* module constructs the EC-Schedule and the corresponding TM. The *Replicator* module replicates TM simulating the TM retransmission mechanism. Both the original TM and the replicated ones are sent to the

Dispatcher module that in turn broadcasts them. The *Dispatcher* module also implements the complete packet forwarding process. For the case of producer-consumer forwarding process of the FTT real-time packets it consults the *Forwarding Table* module that stores the packet forwarding rules which are again read from an XML file at beginning of simulation. The *Classifier* module identifies the packet types, i.e. it classifies if the packet type is FTT real-time (synchronous or asynchronous) or FTT non real-time, and forwards the FTT real-time packets to the *Validator* module. The *Validator* module validates the FTT real-time packets against the information contained in the *SRDB* and in the *Scheduler* (The EC-Schedule). The packets are then stored to one of the INET FIFO queues according to their type, i.e. either to *NRT*, *Sync* or *Async* queue corresponding to FTT non real-time, synchronous, and asynchronous FTT real-time packet type, respectively. The *ETH* modules are used from INET, and model the Ethernet interfaces.

The *FTT Slave* compound module is presented of the right-hand side of Figure 7.1. Its constituting modules are described next. The *NRDB* module stores all the traffic management and global configuration information related to the FTT slave and is initialized at the beginning of simulation from an XML file. The *Combiner* module collects all the TMs, the original one and the retransmitted replicas, it decodes them, and at the end of the TMW sends the EC-Schedule to the *Dispatcher* module. The *Dispatcher* module receives the EC-Schedule and builds the synchronous messages that are scheduled to be transmitted in the current EC using the data from the *Application* module. Additionally, this module is also responsible for the delivery of the received messages targeted to this FTT slave to the *Application* module. Same as before, the *ETH* modules are used from INET, and model the Ethernet interfaces.

The goal of this model was twofold. First, in order to verify the correctness of the TM retransmission mechanism, it was simulated step by step and the *Replicator* and the *Combiner* modules were inspected in detail for corroborating their proper operation. Second, it was used to assess how many ECs are expected to be successfully processed depending on the BER and the number of TM replicas, k, that are pro-actively retransmitted.

Although the results of (Knezic, Ballesteros, and Proenza, 2014) are not fundamental for the current dissertation, we still outline them briefly for the sake of consistency. Note that these results are a contribution of the FTTRS FT mechanisms and not the FT mechanisms devised as a part of this dissertation. As shown in Figure 7.2, (Knezic, Ballesteros, and Proenza, 2014) simulates a system based on a slightly extended version of HaRTES composed of 3 FTT slaves, each connected with a single link to a single HaRTES switch. The Ethernet links were 10 meters long and the bandwidth configured was 100 Mbps.

For injecting errors in the links, the model uses the *BER parameter* that OMNeT++ provides for injecting channel bit errors. Particularly, as seen in Figure 7.2, it uses three BER values. First, for the link connecting FTT Slave1 to the HaRTES, the BER was of 10^{-3}, which is a very pessimistic assumption, and then it was decreased for one (10^{-4}) and three orders of magnitude (10^{-6}) for links connecting FTT Slave2 and FTT Slave3 to the HaRTES, respectively.

It was then shown for these different BER values and different numbers of TM replicas (k) what was the number of successfully processed ECs on a sample of one million ECs, as can be seen in Table 7.1.

The just described OMNeT++ model was the starting point for our new model, as already said. The new model adds the node FT mechanisms for a group of three actively replicated nodes (FTT Slaves) executing a control application. As can be

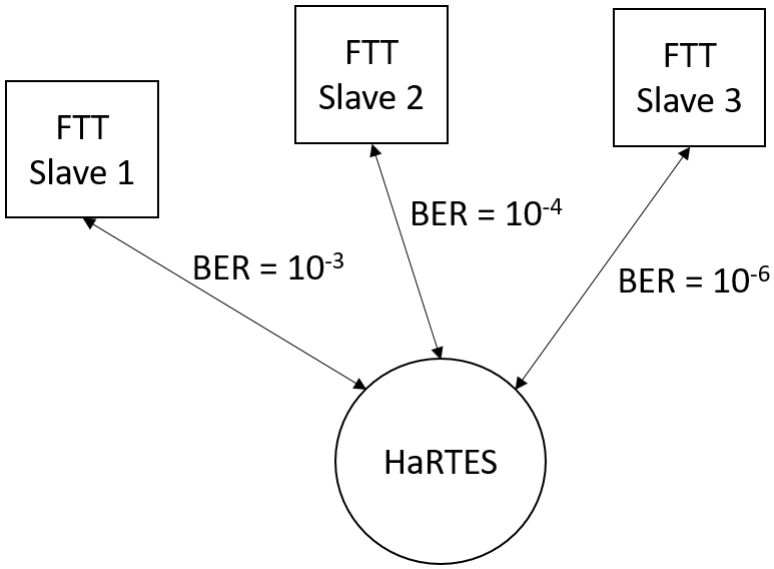


FIGURE 7.2: General architecture of the system modeled in (Knezic, Ballesteros, and Proenza, 2014)

TABLE 7.1: Processed EC results

k	Slave1 (BER = 10^{-3})	Slave2 (BER = 10^{-4})	Slave3 (BER = 10^{-6})
1	56.2075%	94.3675%	99.9398%
2	80.7965%	99.6966%	100%
4	96.329%	99.999%	100%
8	99.8675%	100%	100%
16	99.9998%	100%	100%

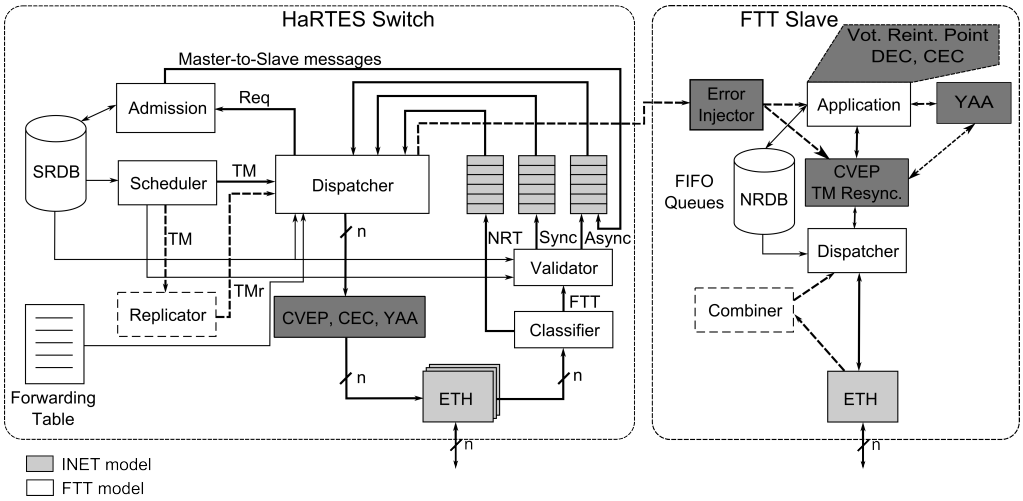


FIGURE 7.3: OMNeT++ model for node replication (source (**derasevic2015OMNeT++**))

seen in Figure 7.3, the new OMNeT++ model includes new modules (dark gray rectangles) and modifies some existing ones.

We have modeled CVEP, CEC and YAA in the switch. In particular, there is a new module between the *Dispatcher* module and the *ETH* mode that includes the aforementioned mechanisms, which are the contributions of this dissertation.

The CVEP is simulated as described in Section 5.3. There are multiple ECs in each VCR devoted to both original and retransmission *cc-vector and ACK exchange rounds*.

First, upon a reception of a cc-vector packet, the corresponding MS vector diagonal value gets populated (set to $true$) and the packet is locally saved in the array data structure maintained by the new module called *retransmissionVector*. In the next ECs of a VCR, if there is a locally stored cc-vector packet in the *retransmissionVector*, it will be forwarded to the output port/s and the one received from the *Dispatcher* module will be dropped. Note that in each retransmission EC the node replicas send the same packets again, even though the cc-vector packets will be dropped by the switch if the new module already has them. The reasons for this are manifold. The EC-schedule is already negotiated for these packets and the bandwidth is reserved. Therefore, if they are not sent, the bandwidth would be left unutilized. Moreover, the resending of cc-vector packets can serve as a signal that the node replicas are still operational. Finally, it would be too complex to change the EC-schedule from one retransmission to another in each EC of the VCR.

Second, upon a reception of the ACK packet, the corresponding MS vector non-diagonal values get populated (set to $true$).

Last, the new module piggybacks the constructed MS vector to the TM replicas following the last EC of the VCR. Furthermore, at this point, the CEC implemented in the switch also inspects the MS vector values to determine if there were any communication errors, and if so, the counter is updated accordingly. If the threshold value is reached, a command is sent to the node replica telling it to reset. The command is sent by piggybacking it to the TM.

Moreover, in this new module we also implement the YAA logic. Specifically, this module adds to each TM the corresponding YAA message that will be used by the YAA watchdog timer implemented in the node replicas.

In the FTT Slaves there are 3 new modules and also a set of new functionalities modeled in the existing *Application* module. We describe them next.

The *Application* module executes a simple PID control application according to the execution pattern described in Chapter 6. This module implements the Voting Reintegration Point, DEC, and CEC mechanisms. Voting Reintegration Point is modeled by including in the cc-vector messages all the variables constituting the operational state of the PID control application and voting on each of them. After each voting, if there is a discrepancy, DEC is increased accordingly. If the threshold is reached, the replica resets itself. Lastly, CEC is modeled by inspecting the MS vector data contained in the TMs received from the switch. If there are communication errors, the counter is updated, and if the threshold value is reached, the replica performs a reset.

The new *CVEP and TM Resynchronization* module located in each FTT Slave simulates the corresponding FT mechanisms. Particularly, as regards the CVEP, this module stores the first received cc-vector from the other two node replicas and forwards it to the *Application* module for voting in the last EC of each VCR. And, as regards the TM Resynchronization, each replica maintains a local TMSN which is populated by copying it from the received TMs, as described in Chapter 5.

The new *YAA* module added to FTT slaves models the YAA watchdog timer device connected to each node replica. In particular, this module simulates the timer which gets reset every time a new TM conveying YAA message is received. If consecutive TMs are not received by this device, enough for the implemented timer to expire, the node replica will be reset by this module.

Finally, the *Error Injector* module is used to inject errors in the *Application* module and the *CVEP and TM Resynchronization* module in order to check the correctness of all the simulated mechanisms. This module is connected to the *Dispatcher* module of the switch in order to allow it to be synchronized with the state of the EC, thus allowing us to inject errors at certain time points in the EC.

7.2 Fault-injection experiments

In order to characterize the behavior of the node FT mechanisms and demonstrate their correctness for a control application, we used this model to carry out a series of fault-injection experiments.

We exhaustively injected each type of temporary fault in a given node replica, at every phase in which it has sense to do so. We left the injection of permanent link and node faults for the real prototype implementation that will be described later in Chapter 6.

In this sense we injected via simulation Transient Long Lasting Faults affecting Links (TLLFL), transient node faults, and Transient Faults affecting the Nodes manifesting as Permanent ones (TFNP). We did not inject transient link faults. Note that transient link faults are tolerated by the proactive retransmission mechanisms provided by FTTRS and the CVEP. Since these mechanisms are simple, transient link faults are expected to be transparently tolerated by them and, thus, there is no real need to verify their correctness.

In order to simplify the fault-injection procedure what we actually injected were the manifestations of faults in terms of the node inability to correctly communicate and/or operate. The way in which we injected faults is summarized in Figure 7.4. Each column of the table specifies the ECAC phase in which the fault is injected.

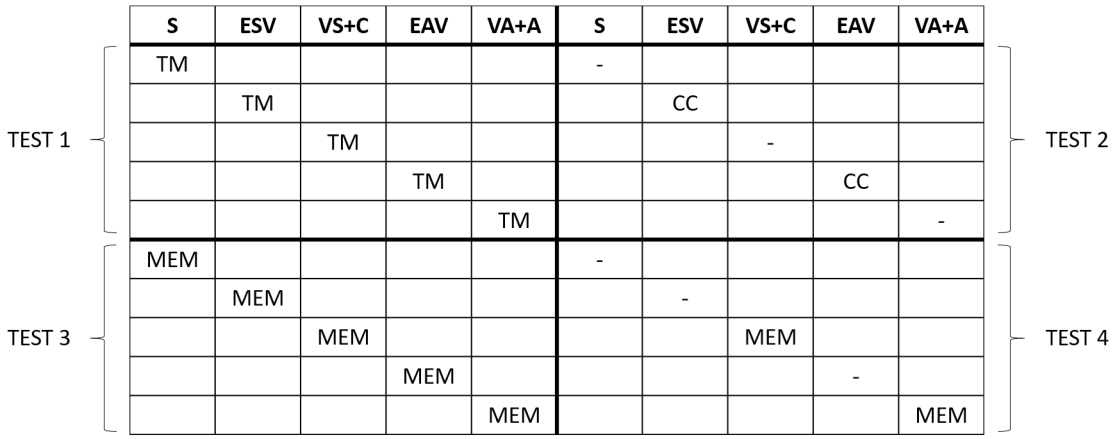


FIGURE 7.4: Error Injection Tests

For the sake of succinctness, Table depicted in Figure 7.4 groups the injected faults into 5 sets of tests. Test set 1 is depicted in the top-left part, test set 2 in the top-right part, test set 3 in the bottom-left part, and test set 4 in the bottom-right one. Test set 5 is somehow depicted together with test set 1, as we will explain later.

In test set 1 we injected TLLFLs preventing a node replica from receiving all the TMs during a given EC and, thus, from triggering the corresponding phase. These injections are indicated by the labels TM in Figure 7.4. In test set 2 we injected TLLFLs that prevent a node replica from successfully receiving, transmitting, or receiving and transmitting all cc-vectors. These injections are labeled as CC. As Figure 7.4 shows, we injected them at phases ESV or EAV, i.e. at the phases in which the node replicas exchange their cc-vectors. Among other, this test set was used to test CEC mechanism by provoking sequential cc-vector reception failures, enough to make the CEC to reach its threshold value and reset a replica. In test set 3 we injected transient node faults that corrupt the operational state of a node replica. These injections are labeled as MEM in Figure 7.4, which means that we corrupted the value of the variables where the node replica stores the operational state. In test set 4 we injected TFNPs that prevent the replica from correctly voting until its DEC reaches the corresponding threshold and resumes the node replica. For this purpose as Figure 7.4 shows, we corrupted during several consecutive ECACs the value of the variable in which the node replica stores every voting result. Finally, in test set 5 we force the replica to not receive several consecutive TMs until its YAA watchdog diagnoses the node replica as crashed and resumes it. Note that the way in which we injected these faults is similar to the one in which we injected the faults of test set 1 (see the top-left part of Figure 7.4), but preventing the reception of all the TMs during several consecutive ECs.

As a result of injecting all the above-described faults, we corroborated that our node fault-tolerance (FT) mechanisms always worked as intended to tolerate and recover/reintegrate from them.

More specifically, the experiments allowed us to characterize the node FT mechanisms. Figure 7.5 and Figure 7.6 summarize this characterization. In particular, the 3rd and 5th column respectively indicate the way in which faults impaired the node(s) and the FT mechanisms that were involved to tolerate (and reintegrate from) them. Moreover, the 4th column quantitatively characterizes the time (in number of ECs) that the affected node needed to recover/reintegrate.

To understand the exhaustiveness of the test sets, let us differentiate between

TM Reception Failure by the Node Replica				
Phase	EC	Consequences	Recovery Time	Recovery Actions
Sense	0	• Sense phase was not triggered • Sensor value was not read from the plant • YAA timer message missed	5 ECs 2 Phases	• Compensated by voting on sensor values in VS + C phase (reintegration) • EC counter resynched from 0 to 1 (TM resynch)
	1	• Watchdog timer message missed	1 EC 1 Phase	• EC counter resynched from 1 to 2 (TM resynch)
ESV	2	• YAA timer message missed • Cc-vector not sent by the replica 1 • Replicas 2 and 3 did not receive cc-vector from replica 1 in the 1st EC of ESV VCR	1 EC 0 Phases	• EC counter resynced from 2 to 3 (TM resynch) • Dealt with using retransmissions that follow, i.e. by the CVEP
	3	• YAA timer message missed • Cc-vector not sent by the replica 1	1 EC 0 Phases	• EC counter resynced from 3 to 4 (TM resynch) • Dealt with using the cc-vectors exchanged in the previous EC, i.e. by the CVEP
	4	• YAA timer message missed • Cc-vector not sent by the replica 1	1 EC 0 Phases	• EC counter resynced from 4 to 5 (TM resynch) • Dealt with using the cc-vectors exchanged in the previous EC, i.e. by the CVEP
VS + C	5	• YAA timer message missed • VS + C phase was not triggered • Operational state values were not calculated	4 ECs 2 Phases	• EC counter resynced from 5 to 6 (TM resynch) • Compensated by voting on all the operational state values in VA + A phase (reintegration)
EAV	6	• YAA timer message missed • Cc-vector not sent by the replica 1 • Replicas 2 and 3 did not receive cc-vector from replica 1 in the 1st EC of ESV VCR	1 EC 0 Phases	• EC counter resynced from 6 to 7 (TM resynch) • Dealt with using retransmissions that follow, i.e. by the CVEP
	7	• YAA timer message missed • Cc-vector not sent by the replica 1	1 EC 0 Phases	• EC counter resynced from 7 to 8 (TM resynch) • Dealt with using the cc-vectors exchanged in the previous EC, i.e. by the CVEP
	8	• YAA timer message missed • Cc-vector not sent by the replica 1	1 EC 0 Phases	• EC counter resynced from 8 to 9 (TM resynch) • Dealt with using the cc-vectors exchanged in the previous EC, i.e. by the CVEP
VA + A	9	• YAA timer message missed • VA + A phase was not triggered • Consensus A value not sent to the actuator	1 EC 0 Phases	• EC counter resynced from 9 to 10 (TM resynch) • Compensated by the output consolidation phase performed by the actuator device
	10	• YAA timer message missed	1 EC 0/1 Phases	• EC counter resynced from i to (i+1) % 20 (TM resynch)
	11			
IDLE	12			
	13			
	14			
	15			
	16			
	17			
	18			
	19			

FIGURE 7.5: OMNeT++ results for TM reception failures

CC-Vector Reception Failure by the Node Replica				
Phase	EC	Consequences	Recovery Time	Recovery Actions
Sense	0 - 1			
ESV	2	• Cc-vector from the node replicas 2 and 3 not received by the node replica 1 in each EC of the ESV VCR • Replica unable to obtain consensus values for any operational state variables in the voting that follows	5 ECs 3 Phases	• Compensated by voting on all the operational state values in VA + A phase (reintegration)
	3			
	4			
VS + C	5			
EAV	6	• Cc-vector from the node replicas 2 and 3 not received by the node replica 1 in each EC of the EAV VCR • Replica unable to obtain consensus values for any operational state variables in the voting that follows	17 ECs 5 Phases	• Compensated by voting on all the operational state values in VS + C phase (reintegration)
	7			
	8			
VA + A	9 - 11			
IDLE	12 - 19			
CC-Vector Transmission Failure by the Node Replica				
Phase	EC	Consequences	Recovery Time	Recovery Actions
Sense	0 - 1			
ESV	2	• Cc-vector from the node replica 1 was not transmitted to the node replica 2 and 3 in each EC of ESV VCS	1 EC 1 Phase	• Compensated by voting on all the operational state values in VS + C phase (reintegration)
	3			
	4			
VS + C	5			
EAV	6	• Cc-vector from the node replica 1 was not transmitted to the node replica 2 and 3 in each EC of EAV VCS	1 EC 1 Phase	• Compensated by voting on all the operational state values in VA + A phase (reintegration)
	7			
	8			
VA + A	9 - 11			
IDLE	12 - 19			
Sensor Value Corrupted				
Phase	EC	Consequences	Recovery Time	Recovery Actions
Sense	0 - 1	• Incorrect sensor value • Discarded by the majority voting procedure	4 ECs 2 Phases	• Compensated by voting on sensor values in VS + C phase (reintegration)
Actuation Value Corrupted				
Phase	EC	Consequences	Recovery Time	Recovery Actions
VS+C	5	• Incorrect actuation value • Discarded by the majority voting procedure	4 ECs 2 Phases	• Compensated by voting on actuation values in VA + A phase (reintegration)

FIGURE 7.6: OMNeT++ results for Cc-vector transmission/reception failures and sense/actuation value corruption

communication and node faults. As regards communication faults, it is important to recall that due to communication faults, a node replica may either not receive TMs, not receive the cc-vectors from the other node replicas, or not transmit its own cc-vector. Figure 7.5 shows the results obtained when preventing the node from receiving all the TMs of a given EC; whereas first two subtables of Figure 7.6 specify the results when preventing the node from receiving the cc-vectors from the other node replicas or from transmitting its cc-vector. As explained before, this inability for receiving/transmitting a given kind of message was injected in each possible EC in which the node replica is expected to receive/transmit that kind of message. Thus, we exhaustively injected communication faults.

Concerning node faults, please recall that they exhibit a byzantine failure semantic. Moreover, at this point we have to recall that these faults are perceived differently, depending on whether we analyze the point of view of the other node replicas or the point of view of the faulty node replica itself.

On the one hand, as already explained, from the other nodes point of view, the Port Guardians (PGs) force a faulty node replica to exhibit an incorrect computation failure semantic. In this sense, a faulty node either does not transmit its cc-vector (because its PG discards it) or proposes an incorrect sensor or actuation value (which is then transmitted in its cc-vector). The first one of these manifestations is covered by the experiments in which the node is prevented from sending its cc-vector (test set 2 and the second half of Figure 7.6). For the second type of these manifestations, we force the node replica to produce an incorrect sensor value in the sense phase, or an incorrect actuation value in the VS+C phase. This is done in test set 4, and the results are shown in the last two parts of Figure 7.6.

On the other hand, node faults are perceived by the affected node replica itself (from its local point of view) as exhibiting an unrestricted (byzantine) semantic. For instance, the node could execute the phases untimely or in an arbitrary order, propose incorrect sensor and actuation values, incorrectly vote thereby obtaining an incorrect sensor and/or actuation value, etc. We believe that all these unrestricted fault manifestations are indirectly reflected in the fault-injection test sets 1, 2 and 3, when they are transient. Note that a node that transiently fails in an arbitrary manner becomes desynchronized from the point of view of the communication and / or the application. As explained in Section 5.3, this means that the faulty node disagrees with the non-faulty ones about what is the current EC and/or what is the current operational state. As we demonstrated by means of test set 1, when the faulty node becomes desynchronized at the communication level, it eventually resynchronizes by means of the TM resynchronization mechanism. Certainly, what Figure 7.5 shows is that the TM resynchronization mechanism successfully resynchronizes the faulty node at the communication level when the difference between the correct Trigger Message Sequence Number (TMSN) and the TMSN considered by the fault node is just of one unit. However, the way in which the TM resynchronization mechanism works is independent from this difference. Thus, a faulty node that is desynchronized at the communication level will resynchronize at that level when eventually receiving the TM. Similarly, by means of test sets 2 (Figure 7.6) and 3 we demonstrated that when the faulty node becomes desynchronized at the application level (either because it could not vote correctly after nor receiving the other replica's cc-vectors, or because its operational got corrupted), it successfully resynchronizes at the application level when voting, independently from the variables of its operational state that have an incorrect value. Finally, test sets 4 and 5 cover the cases in which a transiently byzantine fault makes a node replica to become permanently desynchronized. If the node replica is still able to adequately manage its CEC

and DEC, it will eventually reset. If not, it still exits the possibility that the YAA watchdog resumes the node. Of course, there may be scenarios in which the node replica makes everything wrong, except forwarding the TMN to the YAA watchdog. In this case the node will not be able to reintegrate, but at least its failure will be compensated since from the point of view of the other nodes replicas, the faulty one exhibits an incorrect computation semantics.

By means of the experiments and results presented in this section we have analyzed the behavior of the FT mechanisms proposed in this dissertation and verified their correctness to a large extent. In this sense note that, as said before, here we have not simulated permanent faults. However, permanent faults manifest as transient ones, but in a permanent manner. Thus, since our simulations demonstrate that the errors produced by transient faults are compensated, then they indirectly demonstrate that our FT mechanisms also compensate these errors when they are produced by permanent ones. In any case, as already said, next chapter checks that permanent faults are compensated in a real prototype implementation.

It is important to highlight that the results of the current chapter must not be interpreted as the coverage of our node FT mechanisms; but as the corroboration of their correctness. Note that in order to quantify the coverage of the node FT mechanisms, it would be necessary to carry out extensive fault-injection campaigns, with the appropriate statistical properties.

Finally, it is noteworthy that OMNET++ is not the most adequate tool for measuring the reliability of a system relying on our node FT mechanisms and FTTRS. On the one hand, OMNET++ does not allow to simulate the failure of hardware components following a given time-to-failure distribution - for instance, it does not allow simulating that a node fails following an exponential distribution - or the probability of success of deterministic actions, .e.g. the probability of success of a given fault-diagnosis action. On the other hand, OMNET++ will require more computation time than other reliability quantification techniques; like the one we use in Chapter 9, in which the system reliability is modeled by means of stochastic processes that are analytically solved in a more reasonable amount of time.

Chapter 8

Prototype implementation and Fault-Injection experiments

In this chapter we describe two real prototype implementations of a distributed control system relying on FTTRS and the node FT mechanisms proposed in this dissertation, as well as a set of fault-injection tests we conducted with these prototypes.

The objective of this chapter is twofold. First, it is devoted to demonstrating that it is possible to integrate our node FT mechanisms with the FT mechanisms already provided by FTTRS. Moreover, note that FTTRS is based on the HaRTES protocol, which allows mixing traffic of different criticality levels. Thus, this chapter also aims at demonstrating that a critical control application executed by a set of actively replicated nodes that rely on our FT mechanisms can be executed alongside non-critical applications executed by non-critical nodes.

The second objective of this chapter is to conduct a set of fault-injection tests to demonstrate the correctness of our FT mechanisms. In this sense please note that the second objective of the current chapter is analogous to the one of Chapter 7. However, the difference is that in the present chapter we further want to check the correctness of the mechanisms in a real prototype. In any case, it is important to highlight that, like in Chapter 7, the present chapter is not intended to quantify the coverage of our node FT mechanisms.

As we will explain, the first one of the two prototypes presented in the current chapter was devoted to checking that our FT mechanisms correctly compensate permanent faults. Therefore, of all our node FT mechanisms, it only includes the ones that make it possible for nodes to vote and compensate the failure of a minority of node replicas (it does not include our forward error recover, the reintegration and the fault diagnosis mechanisms). In contrast, the second prototype does include all our FT mechanisms. Thus, we used it to inject the transient faults we already injected via simulation in Chapter 7.

8.1 Description of the first prototype

A first prototype implementation of a control application relying on FTTRS and our node FT mechanisms was presented in (Ballesteros et al., 2016a). My particular contribution to this work was to help in the implementation of the task triggering mechanisms executed by the node replicas (see Chapter 5), as well as in the implementation of the Distributed Consistent Majority Voting (DCMV) mechanism (which in this prototype does not include the CVEP), which allows nodes to consistently vote in a distributed manner.

As depicted in Figure 8.1, the architecture of this prototype includes two interconnected switch replicas and a set of node replicas, one of which is sketched, that

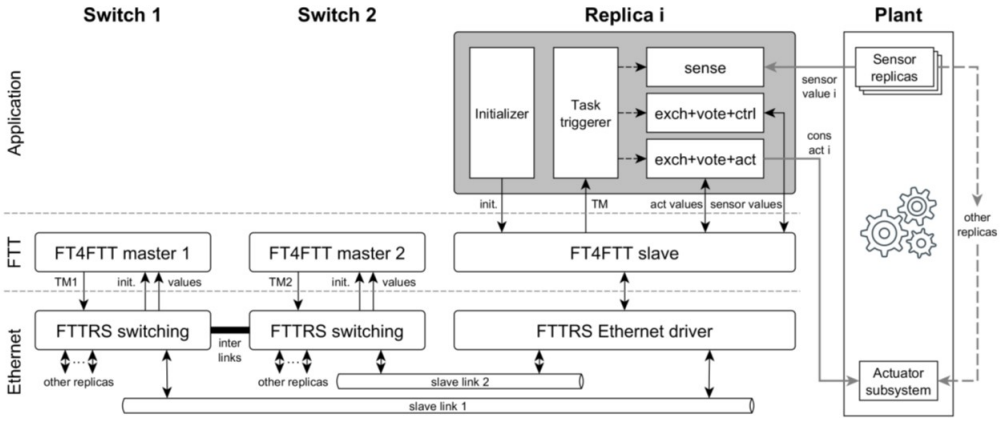


FIGURE 8.1: Implementation Architecture (source (Ballesteros et al., 2016a))

are connected to a plant and its instrumentation (a set of sensor replicas and an actuator subsystem).

We can distinguish between three layers. The first layer, Ethernet, is in charge of Ethernet frame transmission between the node and the switch replicas. The second layer, FTT, is in charge of providing the FTT services, i.e. switches have FT4FTT masters embedded within them while the node replicas have FT4FTT slaves. The last layer, Application, is in charge of the execution of the control application (see Chapter 6).

Now, we briefly describe the operation of all the components. The switches start by broadcasting the TMs. As soon as the node replicas start receiving the TMs they know that the communication subsystem is available. The *Initializer* of each node replica notifies the FT4FTT masters about its communication requirements, i.e. the number of the messages, period, deadline, etc. The *Initializer* of each node replica performs the initialization in a certain time instant and in the certain order to make sure the initialization is done in one EC, after which the regular operation can start. The *Task triggerer* in each node replica receives the TMs and implements our network-centric approach described in Chapter 5, i.e. it handles the TMSN. Note that the phases executed by the control application are merged even further than what we proposed at the end of Chapter 6. Namely, the phases where the cc-vectors are exchanged are triggered together with the phases that represent the application tasks where the voting upon the received cc-vectors takes place and the information obtained by the voting is used.

The prototype executes two applications. One of them is an application that controls an inverted pendulum. It is implemented using a *hardware in the loop* technique, i.e. the inverted pendulum is simulated using Simulink and the control system is implemented in hardware. The other application consists of two nodes; one of them records a video and sends the resulting stream to the other node that, then, displays it.

The hardware of the prototype is depicted in Figure 8.2. There are two PCs that implement the FTTRS switches and one of them implements inverted pendulum in Simulink. There are 5 nodes; 3 of them implement the node replicas executing the control application, whereas the other 2 ones record, exchange and display the video stream as explained above.

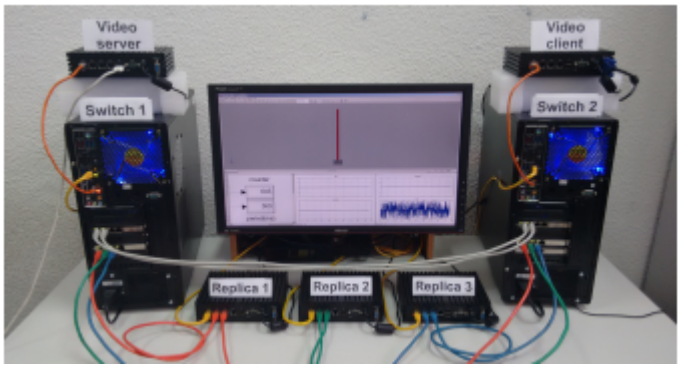


FIGURE 8.2: Prototype (source (Ballesteros et al., 2016a))

8.2 Fault-injection experiments with the first prototype

We used this prototype to inject permanent faults in the channel. All the faults were injected by manually unplugging the links connecting components while the system was operating.

First, it was tested what happens if any of the switches crashes by disconnecting the links connecting them to the rest of the system. It was observed that all the applications were executed normally with no disturbances. Second, it was tested what happens if some of the links become disconnected by unplugging them. The result was that as long as at least a majority of replicas were connected to one switch or two interconnected switches (MFA), the control application operates correctly.

It is important to note that we also injected scenarios in which a node replica becomes permanently disconnected from both switches, again, by disconnecting the links connecting it to the rest of the system. These scenarios in which a node is completely disconnected from the network are an indirect way to inject permanent node faults. This is because from the non-faulty nodes perspective, a permanently faulty node either transmits cc-vectors with incorrect values or transmits no cc-vector at all. In both cases the permanently-faulty node is tolerated by the DCMV, which succeeds independently of whether the non-faulty nodes vote using an incorrect cc-vector from the faulty node or vote using no cc-vector from that node.

8.3 Description of the second prototype

After conducting all these experiments, the prototype of (Ballesteros et al., 2016a) was extended so as to include the rest of our node FT mechanisms. The new resulting prototype is described in (Ballesteros et al., 2016b). Specifically, the new mechanisms included in the resulting prototype are the CVEP; the reintegration mechanisms, namely the TM resynchronization and Voting Reintegration Point; and the fault-diagnosis mechanisms, i.e. the DEC, CEC, and YAA watchdog timer. For a detailed description of these mechanisms, please refer to Chapter 5. My particular contribution to this work was to help in the implementation of all these mechanisms.

Figure 8.3 depicts the architecture of the new resulting prototype. The gray squares are the new/modified implementation modules. In this implementation we perform the execution of phases as was described at the end of Chapter 6 and therefore modify the *Action triggerer* and the phases accordingly. *Communication Error Counter* (CEC) mechanism was implemented at the second layer, FTT, because it belongs to the communication subsystem and *Discrepancy Error Counter* (DEC) and

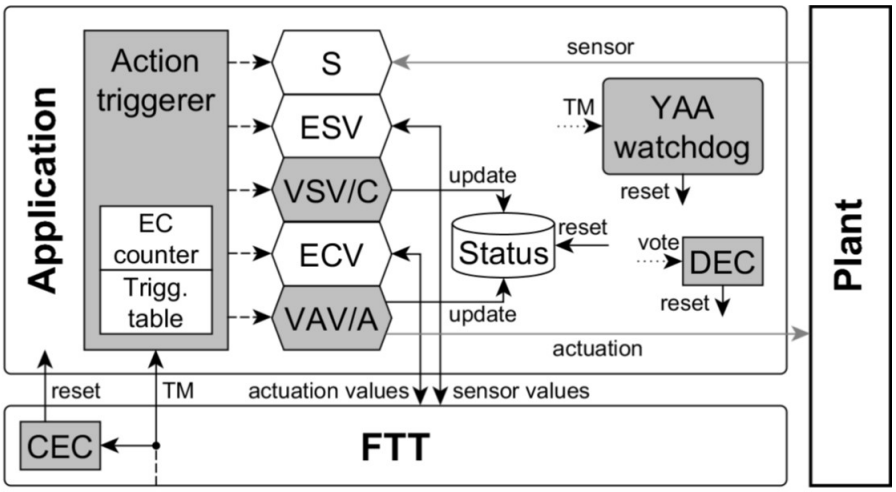


FIGURE 8.3: Implementation Architecture (source (Ballesteros et al., 2016b))

You Are Alive (YAA) watchdog were implemented at the third later, Application, for the analogous reasons.

8.4 Fault-injection experiments with the second prototype

We used this new prototype to inject, at one of the three node replicas that execute the inverted pendulum control application, the same types of faults we injected using the simulation model (Section 7.2), namely Transient Long Lasting Faults affecting Links (TLLFL), transient node faults, and Transient Faults affecting the Nodes manifesting as Permanent ones (TFNP).

Note that we used the same strategy as in Section 7.2 to exhaustively inject these faults while reducing the complexity of the fault-injection procedure. On the one hand, this means that we classified the fault-injection tests into the same 5 test sets of Section 7.2. On the other hand, this implies that we injected the faults in the same way as in the simulation, i.e. we injected faults by preventing one of the nodes of the prototype from correctly communicating and/or operating.

Just for the sake of clarity, Figure 8.4 repeats the Table depicted in Figure 7.4 of Section 7.2, which specifies what kind of faults we injected in each test set, as well as the way in which we did so.

Please note again that each cell represents the phase in which a given type of fault is injected. Label TM specifies that the node was forced to lose all the TM replicas of the ECs that compose the phase. Label CC specifies the phases in which the node was prevented from receiving all the cc-vectors sent by the other node replicas and/or prevented from transmitting its own cc-vector to both switches. Label MEM indicates the phases in which the value of the variables that compose the operational state that is locally stored in the replica are corrupted.

By means of conducting the fault-injection test sets in the real prototype we corroborated what we already observed when conducting them via simulation. First, we successfully checked that all of our FT mechanisms worked as intended in presence of the injected faults. Second, we characterized the way in which faults impair the nodes, the FT mechanisms involved in each case, and the recovery/reintegration time.

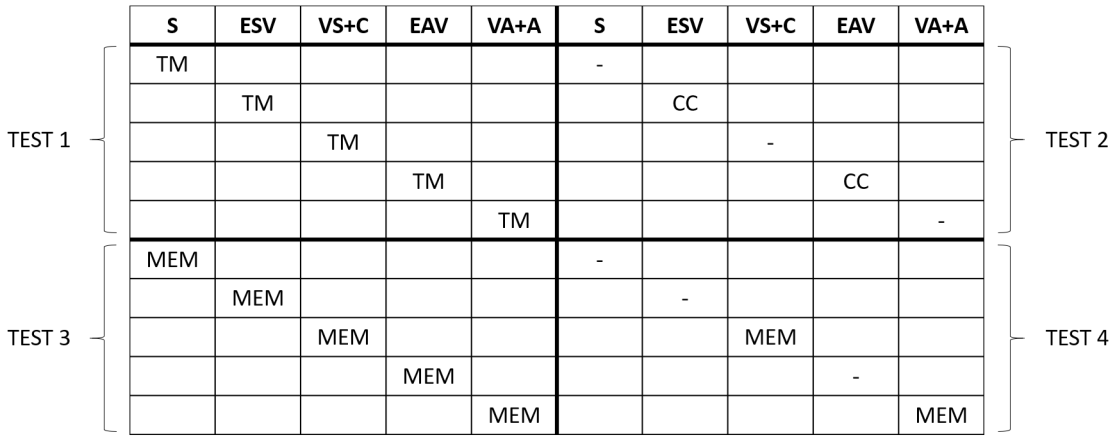


FIGURE 8.4: Error Injection Tests

Next, we summarize the characterization regarding the faults' impairments and the involved FT mechanisms; please refer to Figure 7.5 and Figure 7.6 of Section 7.2 for further details about it.

Test set 1 is devoted to injecting TLLFLs that prevent the node replica from receiving all the TMs sent at the ECs that compose a given phase. As a result, the replica does not execute the corresponding control application phase. To resynchronize at the communication level the faulty node replica always uses the TM resynchronization mechanism. However, some of the missed phases cause the fault replica to not obtain the same operational state as the others, e.g. when the voting is not performed, and thus the Voting Reintegration Point mechanism is needed as well to resynchronize the replica at the application level.

Test set 2 aims at injecting TLLFLs that prevent the replica from receiving and/or transmitting cc-vectors. In these set of tests we test all the combinations, namely the inability of a faulty node replica to receive cc-vectors, the inability to transmit cc-vectors, and both. The manifestation of these faults causes a faulty replica to not obtain the same operational state as the others, and thus, the Voting Reintegration Point mechanism is needed. Note that it makes sense to inject these faults only in the phases in which the cc-vectors must be exchanged.

Test set 3 injects transient node faults that corrupt the operational state of the node replica. This was done by corrupting the operational state values the node replica generates and uses. Same as before Voting Reintegration Point is needed.

Test set 4 injects a TFNP that prevents the node replica from correctly voting. In particular, the values obtained by the node replica when locally voting are corrupted for several consecutive voting phases; enough to make the node replica DEC to reach its threshold and, thus, to compel that node replica to reset. After the reset the node replica uses both the TM resynchronization and the Voting Reintegration Point mechanisms to reintegrate.

Test set 5 injects a TFNP that leads the node replica to completely crash, i.e. the replica is unable to execute any action. This is done by preventing the node replica from receiving TMs, enough to cause its YAA watchdog timer to expire and reset the replica. Like with the previous test set, the node replica uses both the TM resynchronization and the Voting Reintegration Point mechanisms to reintegrate.

Concerning the characterization of the time need for the node replica to recover/reintegrate, the 4th column of Tables depicted in Figure 7.5 and Figure 7.6 already summarize the results. In addition, Figure 8.5 provides some extra details. Each row

	Last phase affected by the fault				
	S	ES	VS/C	EA	VA/A
1	2	3	2	0	0
2	-	3	-	0	-
3	2	3	2	0	0
4	-	-	2	-	3
5	2	3	2	4	3

Statistics	
max	avg
3	1
3	1.5
3	1.4
3	2.5
4	2.8

FIGURE 8.5: Time to recover/reintegrate (reproduced from the source (Ballesteros et al., 2016b))

of each table refers to a given test set. As concerns the left-hand table, each one of its columns indicates the phase in which a fault being injected ceases; whereas the number specified in each one of the cells corresponding to a column indicates the number of ECs the replica needs to recover/reintegrate once the fault ceases. The right-hand table specifies, for each test set, the maximum and the average number of ECs needed for the node replica to recover/reintegrate when the injected fault ceases.

As can be seen in those tables, the faulty replica recovers/reintegrates as soon as the first successful voting takes place; so that the time needed to recover/reintegrate is the time that elapses since the fault ceases until the node replicas votes.

8.5 Conclusion

The results obtained by means of the fault-injection tests carried out with the real prototypes described above fulfill the two objectives of the current chapter. On the one hand they demonstrate that the mechanisms proposed in this dissertation can be integrated with those already proposed by FTTRS. In this way, we prove that it is possible to provide critical nodes with mechanisms to tolerate faults affecting the network and/or the nodes themselves, while being able to share a reliable and flexible real-time network with non-critical nodes. On the other hand, the results further corroborate in a real prototype what we already show in Chapter 7 as regards the correctness and the characterization of our node FT mechanisms.

Certainly, as happens with the OMNeT++ simulation model, it would be quite complex and time-consuming to use a real prototype to quantify the reliability benefits of our FT mechanisms. Hence, as already pointed out, Chapter 9 is devoted to clarifying this issue by means of a more adequate and efficient reliability evaluation technique.

Chapter 9

Dependability Evaluation

This chapter is devoted to describing the dependability evaluation of our system. The goal of this evaluation is to obtain numerical results of our system reliability.

The goal of dependability evaluation, which is an equivalent to quantitative evaluation technique described in Section 2.1, presented in this section is to first model our system, and then to analyze it and quantify the reliability achieved by it. When building dependability models, we consider the system architecture that executes the control applications presented in Chapter 6.

To model our system we follow the approach proposed in the work of Valério Rosset et al. (Rosset et al., 2012). This work uses *Discrete-Time Markov Chains* (DTMC) formalism to model *Group Membership Protocol* (GMP) for *Time-Division Multiple Access* TDMA networks.

Specifically, the GMP is executed repeatedly on a TDMA communication round basis and its faults assumptions are expressed on a per-execution basis. Thus, the DTMC models of the GMP use a time step that is closely related to the GMP execution. The use of DTMC allows the association of the state transitions of the models with the passing of chosen time intervals.

Similarly, in this thesis, we use DTMC formalism to model the behaviour of our system within ECAC that is periodically repeated. The time step of DTMC models will be closely related to the discrete phases that constitute each ECAC.

The built models are analyzed and the reliability achieved is measured by a technique called *reliability prediction* (Blischke and Murthy, 2011). The goal of reliability prediction is to predict the failure rates of components and overall system reliability. Particularly, within the scope of reliability prediction, the reliability can be defined as the ability of the system to perform its intended service without failure under stated conditions and for a declared period of time. The reliability is quantified in the terms of probabilities. The quantification is defined as follows: it is the survival probability of the system over a specific period of time during its life, when only one failure can occur.

To build DTMC models that describe our system and to measure the reliability achieved using reliability prediction we use the PRISM tool (Kwiatkowska, Norman, and Parker, 2011). In the next sections we first describe the PRISM tool, then we describe our dependability models and how we analyze them using the PRISM language, and finally, we discuss the obtained reliability results.

Note that all the PRISM queries and variables specified in this chapter are just for the illustration purposes. The detailed code and all the used variables will be explained in Appendix A.

9.1 PRISM model checker

PRISM is a probabilistic model checker tool that supports the following probabilistic models (Kwiatkowska, Norman, and Parker, 2011):

- *Discrete-Time Markov Chains* (DTMCs)
- *Continuous-Time Markov Chains* (CTMCs)
- *Markov Decision Processes* (MDPs)
- *Probabilistic Automata* (PAs)
- *Probabilistic Timed Automata* (PTAs)

To develop and analyze models in PRISM, the PRISM language has to be used. Following are the three main components of the PRISM language:

- *Modules* are the main composition units which can cooperate with each other describing the whole model. They are specified as:

  ```
  module name ... endmodule
  ```

- *Variables* are defined within each module and they represent the *local state* of the encircling module. The *global state* of the model is determined by the *local state* of the included modules. A variable is declared as:

  ```
  x : [0..2] init 0;
  ```

  In the example above the variable x is declared as an integer ranging from 0 to 2 with an initial value of 0.

- *Commands* can be defined within a module and specify its bahaviour. They have the following declaration:

  ```
  [sync] guard -> p1 : u1 + ... + pn : un;
  ```

  The *guard* represents a predicate over the variables of all the modules in one model. If the condition of the guard is fulfilled, an *update*, the transition that modifies the local variables, $u1$ to un can be taken with the corresponding *probability* $p1$ to pn. The optional $sync$ label defines the synchronization with the other modules. If commands from two or more different modules have the same label defined in the square brackets, then these two commands are synchronized, i.e. the transition specified by them are taken simultaneously. If no label is defined, or if the defined label does not match any of the labels from the other module commands, there is no synchronization.

PRISM property specification language is used to analyze the constructed probabilistic models. The operators that we shall use in our models are the regular and the bounded *P* operators that are used to reason about the probability of an event occurrence, in a bounded case within a specified time bound. The path property that we shall use in combination with P operator is *F* property, usually called *eventually*, and it means that the property being checked eventually becomes true at some point along the path.

A simple example of these two operators are illustrated below:

```
P=? [ F messageLoss = true ]
P=? [ F<=100 systemFail = true ]
```

This first query is an example of a regular P operator with F property that specifies what is the probability of a message eventually being lost (variable $messageLoss$ being $true$). The second query is an example of a bounded P operator with F property that specifies what is the probability of a system failure event eventually occurring (variable $systemFail$ being $true$) within 100 time units.

In PRISM models we can also define *rewards*, i.e. the real values associated with certain states or transitions of the model, that can then be analyzed to obtain the *expected values* of the defined rewards. The particular reward properties that we use in our models are *cumulative reward* properties. They function by associating a reward with each path of a model and then accumulating it along a path, but only up to a given time bound. An example of cumulative reward property is:

```
R=? [ C <= mission_time ]
```

This property returns the expected accumulated reward within $mission\_time$ time units of operation.

PRISM tool also supports *transient probabilities calculation*. This feature allows us to calculate the probability of each possible global state of the model (combination of the values of the local variable of all the modules) for a defined number of time units. As will be shown in Section 9.3 we will use this feature of PRISM to obtain reliability results.

9.2 Dependability Models

In the first modeling attempt we have modeled all the functionalities in a single complex PRISM model, but due to state space explosion, the PRISM tool was unable to build and execute the model. After couple of iterations by applying different modeling optimization techniques described in (Boyd and Lau, 1998) our final version of the model consists of 3 PRISM models. There is one *main* model and two auxiliary ones. The idea was to reduce the complexity (state space) of the main model by disentangling some of the functionalities to the auxiliary models. The auxiliary models are then executed separately and the results obtained by them are used by the main model.

Moreover, to further reduce the state space, these models use a set of values that are calculated beforehand, and that characterize the probabilities of node replicas and switches failing to transmit or receive certain key messages.

Our dependability models are used to model the execution of 3 node replicas interconnected by the FTTRS. We choose to model 3 node replicas because this is the typical level of node redundancy used in fault-tolerant architectures based on majority voting.

Each model will be composed of a couple of modules. In each model the module representing a node replica will be the one dictating the execution of discrete time steps and the other modules will synchronize with it. A node replica module will always be instantiated 3 times to represent 3 node replicas and they will execute in a lock-step.

All models follow the same execution pattern. A set of sequentially executed discrete time steps, henceforth steps, is periodically repeated until a termination number of time steps is reached.

A step is a set of mutually exclusive synchronized PRISM commands modeling some specific behaviour. An example of a module executing two steps is presented below.

```
module example

...

step : [1..n] init 1;
systemFailed : bool init true;
failedSwitches : [0..2] init 0;

...

[sync] step = 5 & systemFailed = true -> ...;
[sync] step = 5 & systemFailed = false -> ...;

[sync] step = 6 & failedSwitches = 0 -> ...;
[sync] step = 6 & failedSwitches = 1 -> ...;
[sync] step = 6 & failedSwitches = 2 -> ...;

...

endmodule
```

As shown above, for illustration purposes we define a module called $example$ with three local variables, $step$, $systemFailed$ and $failedSwitches$, representing the sequentially executed steps of the module, the system failure, and the number of failed switches respectively. Note that if there is another module that also executes commands with $[sync]$, it will be executed simultaneously with this one.

In the first step, $step = 5$, if the system has failed, $systemFailed = true$, execute the body of the first command of step 5, and if not, $systemFailed = false$, execute the body of the second command of step 5.

In the second step, $step = 6$, if no switches have failed, $failedSwitches = 0$, execute the body of the first command of step 6, if 1 switch has failed, $failedSwitches = 1$, execute the body of the second command of step 6, and if both switches have failed, $failedSwitches = 2$, execute the body of the third command of step 6.

Figure 9.1 depicts the graphical representation that we use to represent our models. Circles represent the steps and arrows the set of mutually exclusive transitions. Furthermore, we distinguish between 4 types of steps in our models: *regular*, *failure*, *evaluation* and *synchronization* steps. The first three types are executed by the modules representing the node replica instances only, while the last one involves the execution of the other modules as well.

Next, we explain each step type.

- *Regular* steps model the consequences of faults on different system components during certain system operations and also the actions taken by these components to deal with them, e.g. the effect of cc-vector losses on the execution of EAV ECAC phase. They are represented by single border white circles.
- *Failure* steps model the failures of different components of our system due to transient and permanent faults. In all the models these steps are executed at the

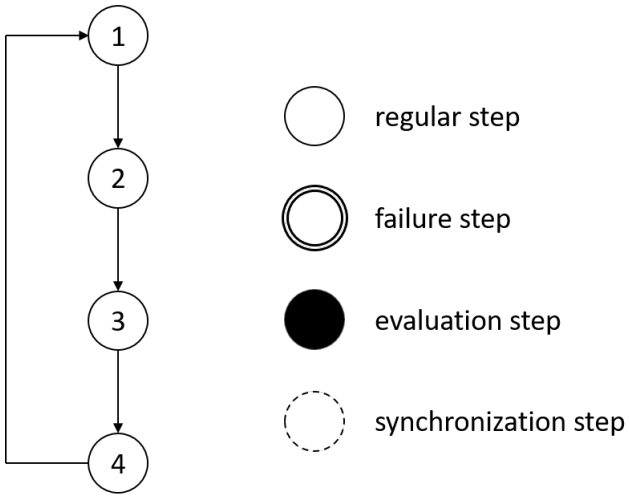


FIGURE 9.1: PRISM steps

beginning of the periodic execution of the model. By using this approach, we assume that the failed component is useless immediately after the evaluation of the failure step. The approach is pessimistic because of two reasons. First, a failure might have happened anywhere within the periodic execution of the model and the component might have still been operational, and second, in reality the failure can manifest after some time and the component might have still been used until the manifestation.

Note that in the case of permanent failures, the effects will last throughout the entire execution of the model, and in the case of transient failures, only throughout one period of model execution.

These steps modeling failures are represented by double border white circles.

- *Evaluation* steps model the evaluation of the system state in order to decide in a deterministic manner if the system has failed or not. If it has, the model evolves to a global state by each node replica module modifying its the local variables correspondingly to represent the system failure. This being done prevents any local variable from being modified. As a result, the introduction of these steps has a positive effect on the modeling by reducing the number of possible transitions and speeding up the model. These steps are represented by black circles.

- *Synchronization* steps model the simultaneous execution of different modules of the model. Recall that the modules representing multiple instances of node replicas are always executed in lock-step. Therefore, these steps refer to the synchronization of different modules with multiple instances of node replica modules. They are represented by dashed border white circles.

As regards our 3 models, the main model is used to model the execution of our system in an ECAC. The organization of ECAC phases used in the one described at the end of Chapter 6, i.e. the phases VS and C and the phases VA and A are both merged. The auxiliary VCR model is used to model the execution of a single VCR and the auxiliary reset model is used to model if the failure of our system occurs when one replica is being reset. All three models are periodically executed until a termination time expressed by a predefined number of discrete time steps is reached. However, note that the period and the termination time for each model is different.

The period for the main and the reset model is an ECAC, and the period for the VCR model is an EC. The termination time for the main model is the number of discrete time steps in the duration of the mission time for a specific domain (this will be the topic of Section 9.4), for the reset model the duration of a reset of a node replica, and for the VCR model the duration of a VCR.

Now that we have described the basis of the used execution pattern and model separation, in the following sections we describe, first, how we calculate probabilities used by the models, and then, each of the PRISM models in detail.

9.2.1 Probabilities calculation

Prior to the execution of the modules of different models, we calculate a set of probabilities that are then used by the module commands.

The first set of precalculated probabilities refers to node replicas or switch replicas losing, i.e. failing to transmit or receive, different key messages in a given EC. These probabilities and their corresponding acronyms are the following ones:

- The probability of a node replica losing (failing to receive) all the TM replicas sent by the switches in a given EC (TMRF)
- The probability of a node replica failing to transmit its cc-vector to the switches in a given EC (CCTF)
- The probability of a node replica failing to receive any cc-vector from the switches sent by one of the other node replicas in a given EC (CCRF)
- The probability of one switch replica failing to receive from the other switch replica any message needed to synchronize and to be replica determinate with the other switch replica in a given EC (SWRF).

To understand why it is important to provide the modules with these set of precalculated probabilities, first note that if TMRF occurs, then the affected replica cannot execute the corresponding EC, which may lead that replica to fail. Similarly, CCTF and CCRF affect replicas' ability to vote, recover and/or reintegrate after a VCR. Finally, if SWRF happens the whole system fails, as switches may lose synchronism or replica determinism with respect to each other.

Next, we will explain how each of these probabilities needs to be calculated depending on the possible network configurations, i.e. the combination of non-failed network components.

First, as seen by Figure 9.2 there are 4 possible network configuration combinations, 3 of which are relevant, for the calculation of TM loss probabilities (TMRF).

In the case 1, all of the network components are operational, i.e. a replica is connected by two links to two interconnected switches. In this scenario a replica receives k TM replicas 4 times: first switch sends TM replicas to the node replica directly through the link connecting them, and through the other link via the interlinks with the other switch; second switch does the same.

In the case 2, there are two operational switches that are not interconnected. This case leads to the system failure due to inability of the switch replicas to communicate and to establish replica determinism.

In the case 3, a replica is connected to one of the two interconnected switches. In this scenario a replica receives k TM replicas 2 times, once from each of the switches.

In the case 4, a replica is connected to one of the surviving switches. In this scenario a replica received k TM replicas only once.

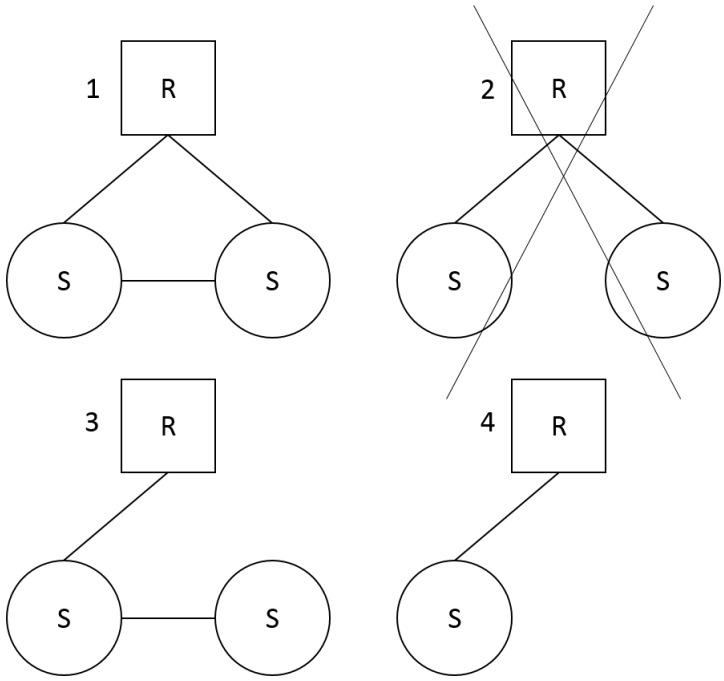


FIGURE 9.2: Possible configuration for calculating TM probability loss

Second, the probability of transmitting cc-vector replicas (CCTF) depends only on the number of links with which a replica is connected to the switches. If it is connected with only one link (cases 3 and 4 from Figure 9.2), a replica transmits k copies, and if it is connected with two (note that this assumes two interconnected switches - case 1 from Figure 9.2), a replica transmits $2k$ copies.

Third, as seen by Figure 9.3 there are also 4 possible network configuration combinations for the calculation of the failure to receive cc-vectors (CCRF).

In the case 1, all of the network components are operational, i.e. both transmitting node replica R_{tx} and receiving node replica R_{rx} are connected by two links to two interconnected switches. In this scenario a receiving replica receives k cc-vector replicas 4 times since there are 4 distinct paths from R_{tx} to R_{rx}.

In the case 2, everything is the same apart from R_{tx} that now has only one link.

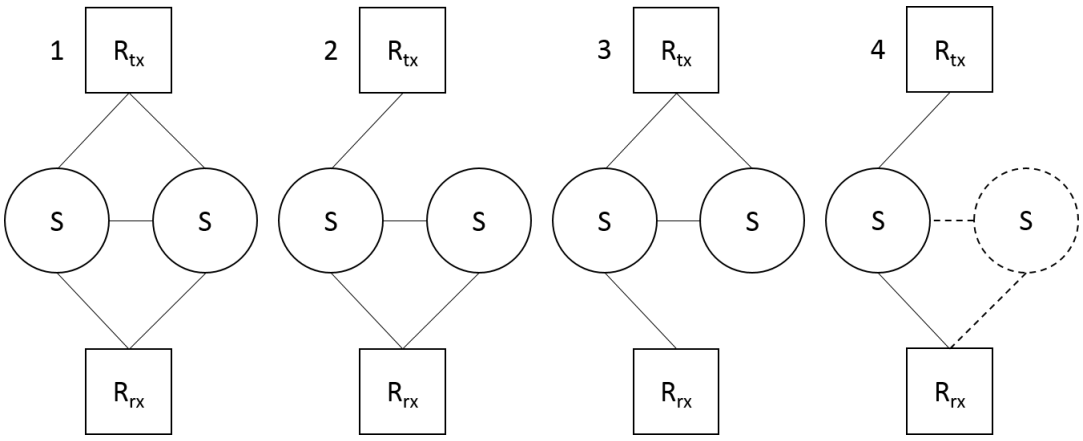


FIGURE 9.3: Possible configuration for calculating cc-vector probability loss

In this scenario a receiving replica receives k cc-vector replicas 2 times since there are 2 paths.

In the case 3, everything is the same apart from R_{rx} that now has only one link. In this scenario a receiving replica receives k cc-vector replicas 2 times since there are 2 paths.

In the case 4, R_{rx} and R_{tx} are connected with only one link to the switch/es. In this scenario a receiving replica receives k cc-vector replicas only once due to the existence of only one path.

Last, the probability of one switch replica failing to receive from the other switch replica any message needed to synchronize and to be replica determinate with the other switch replica in a given EC (SWRF) is taken into account only when both switches are operational and interconnected. In that case each switch receives from the other a predefined number of control message replicas.

Once a configuration is determined, corresponding message loss probability (cc-vector, TM) needs to be calculated. Acknowledgments (ACK) message loss probability will not be considered in the model because these messages cannot lead to system failure, as will be explained later. This probability is calculated as follows:

$$\begin{aligned} ProbSingleMessageLoss &= 1 - (1 - BER)^{FrameSizeInBytes*8} \\ ProbAllMessageLoss &= ProbSingleMessage^{num\_repetitions} \end{aligned} \tag{9.1}$$

First, we calculate the probability of losing a single message, $ProbSingleMessageLoss$, using *Bit Error Ratio* (BER) parameter. BER is the ratio between bit errors and total number of transferred bits. Then, we can calculate the probability of losing all messages, $ProbAllMessageLoss$, by multiplying $ProbSingleMessageLoss$ as much times as the message is repeated, $num\_repetitions$.

The second calculated set of probabilities refers to permanent and transient component failures and for their calculation we use the failure rates. Failure rate of a component is a frequency with which a component fails and is expressed in failures per unit of time.

For failure rates we assume the exponential distribution. With the assumed exponential distribution we can calculate *Cumulative Distribution Function* (CDF), i.e. the probability that a component fails within t time units, where T is a random variable representing a time to failure of a component, and λ is the failure rate of the component.

$$CDF = P(T <= t) = 1 - e^{-\lambda t} \tag{9.2}$$

9.2.2 Auxiliary VCR Model

It models all the particularities of the VCR of the CVEP (See Chapter 5). This model extracts the VCR behaviour and calculates the probabilities that will be reused by the main model.

The input parameter for this model is a network configuration, i.e. the number of operational (non-failed) switches, interlinks and links. The output of the model is a list of probabilities of cc-vector reception failures for each node replica after the execution of a VCR. The goal of this model is to calculate cc-vector reception failures for each replica and for each possible combination of input parameters. Moreover, this model takes into account the redundancy level used for the messages and the

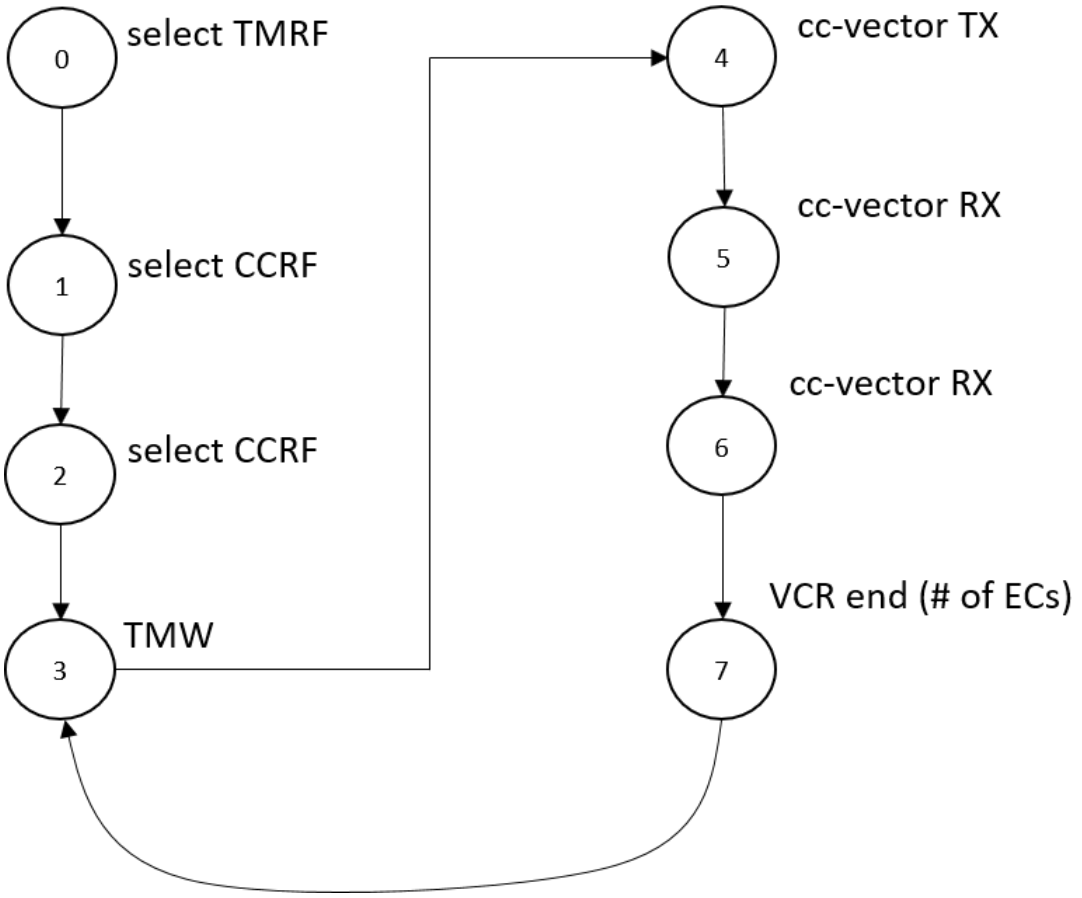


FIGURE 9.4: VCR PRISM model

ECs of the VCR. Then, we reuse these results by the main and the reset model whenever an outcome of a VCR (cc-vector reception failures) has to be consulted. These probabilities (outcome) can be used to update in the main model whether or not cc-vectors have been successfully exchanged, which was the aim of the VCR.

The model consists of only one module representing a node replica that is instantiated 3 times. The node replica modules are executed in lock-step and the execution of the model is depicted in Figure 9.4.

Steps 0, 1 and 2 are used to deterministically select the probabilities of messages losses depending on input network configuration that were calculated beforehand (see Section 9.2.1). Specifically, Step 0 selects TMRF, and Steps 1 and 2 select CCRFs corresponding to the reception of cc-vector from the other two node replicas.

Steps 3-6 model the execution of one EC of a VCR. These steps are executed periodically until the VCR ends. Step 3 models the TMW in which a replica receives TMs from the switches taking into account the preselected TMRF. At least one TM replica needs to be received for the node replica to be able to transmit/receive its cc-vectors. If no TM passes through to the node replica, Steps 4, 5 and 6 will take this into account and simply move on to the next subsequent step until Step 7 is reached. Step 4 models the transmission of cc-vector replicas from the node replicas to the switches taking into account the CCTF. At least one cc-vector replica from each node replica needs to be successfully transmitted for the switches to be able to forward it to the receiving replicas. Steps 5 and 6 model the reception of cc-vectors from the switches that were transmitted by the other two replicas in the previous step taking into account the preselected CCRFs. Finally, step 7 deterministically decides

TABLE 9.1: The output of the VCR model

Switches	Interlinks	Links 1	Links 2	Links 3	cc 12	cc 13	cc 21	cc 23	cc 31	cc 32
0	0	0	0	0	p12	p13	p21	p23	p31	p32
0	0	0	0	1	p12	p13	p21	p23	p31	p32
0	0	0	1	1	p12	p13	p21	p23	p31	p32
0	0	0	1	0	p12	p13	p21	p23	p31	p32
0	0	1	0	0	p12	p13	p21	p23	p31	p32
...										
2	1	2	1	2	p12	p13	p21	p23	p31	p32
2	1	2	2	0	p12	p13	p21	p23	p31	p32
2	1	2	2	1	p12	p13	p21	p23	p31	p32
2	1	2	2	2	p12	p13	p21	p23	p31	p32

whether or not all the ECs of the VCR have been executed, taking into account how many ECs compose the VCR.

The specific output that is calculated by the VCR model is presented by Table 9.1. We obtain this table in PRISM by defining an experiment in which we vary the input parameters.

First 5 columns are the input parameters. Specifically, columns $Switches$ and $Interlinks$ specify the surviving switches and interlinks of the FTTRS, and columns $Links\, i$ specify links of replica $i, i \in [1, 3]$. Note that for the interlinks we assume that the maximum value is 1 and this value represents that at least one interlink is operational. On the contrary, the value 0 represents the failure of both interlinks. Why we model interlinks like this will be explained in the next section, Section 9.2.3.

Last 6 columns are the output of the VCR model for each combination of input parameters. The output is specified by the following probabilities: $pij|ij \in [12, 13, 21, 23, 31, 32]$, where each column specifies a probability of a replica i failing to receive at least one cc-vector from replica j.

9.2.3 Auxiliary Reset Model

It models if the system fails during a reset of one replica. Note that if more than one replica is reset, the system fails due to violation of MFA and this will be considered by the main model. Therefore, we model a reset of one replica only by this model and the behaviour of the other two non-reseting replicas.

Similarly like in the VCR model, the input parameters for this model are a subset of network configurations and the output is the probability of the system failure for each case. The goal of this model is to reuse it output (the set of calculated probabilities) by the main model whenever an outcome of the system failure during a reset of one replica has to be consulted.

The model consists of three modules. The first module, *Node Replica* module, models a node replica behaviour in an ECAC and it is instantiated 3 times. One instance will model a reseting replica while the other two instances will model the non-reseting replicas. All three node replica modules are executed in lock-step and they dictate the execution of the entire model. The second module, *Switches* modules, models the permanent failures of the switches in an ECAC, and the last module, *Interlinks* module, models interlinks permanent failures in an ECAC. The last two modules only execute some of the steps in synchrony with the node replica modules.

It has to be noted that we have modeled interlinks failures as follows: either both of them fail simultaneously or both of them are operational. This was done in order

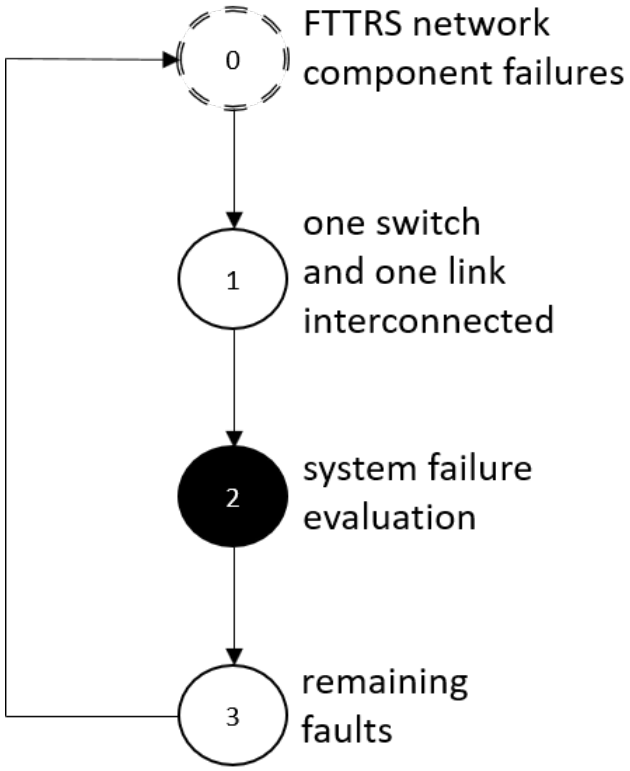


FIGURE 9.5: reset PRISM model

to further reduce the complexity of the model and the state space. This approach is again pessimistic since we could tolerate the failure of one interlink.

As seen by Figure 9.5 there are only 4 steps that model the periodic execution of ECAC.

Step 0 is a failure synchronization step. It models the permanent failures of FT-TRS components: links, switches and interlinks in a current ECAC. All three modules, namely, three *Node Replica* module instances and one instance of each *Switches* and *Interlinks* modules, execute this step in synchrony to model the failures of components that each belong to a different module.

Step 1 is used to detect a scenario where there is only one surviving node replica link and one surviving switch replica left. If this is the case, we model that with 50% probability this surviving link is connected to the surviving switch and with 50% probability it is not (equiprobable chances). This is done since we do not distinguish between switch replicas and link replicas in our model. So, this specific scenario had to be covered for.

Step 2 is an evaluation step. It evaluates the global system state to determine if the system has failed. If so, all the relevant variables are set to fitting values to signify the system failure.

Step 3 models the occurrence of all the faults that can lead to the system failure that were not modeled by the previous steps. These faults include both permanent and transient node replica failures along with transient failures affecting both the TMs and the cc-vectors. After the execution of step 3 the model evolves back to step 0 and starts the next ECAC period.

Similarly like in the VCR model, the particular output produced by this model, portrayed in Table 9.2, is obtained by defining a PRISM experiment in which the

TABLE 9.2: The output of the reset model

Switches	Interlinks	Links 2	Links 3	System Failed
1	0	1	1	pSysFailed
1	0	1	2	pSysFailed
1	0	2	1	pSysFailed
1	0	2	2	pSysFailed
1	1	1	1	pSysFailed
. . .				
2	1	1	1	pSysFailed
2	1	1	2	pSysFailed
2	1	2	1	pSysFailed
2	1	2	2	pSysFailed

input parameters (the first 4 columns) representing the relevant network configurations are varied to obtain the output (the last column) that represents the probability of the system failure.

Since one replica is being reset, and cannot transmit or receive messages, its links ($Links1$) are not of importance for the evaluation of this model output. Also, since the system failures due to component failures are going to be evaluated by the main model, in this model we only take into consideration network configurations in which the system has not failed yet.

9.2.4 Main Model

This model is the most complex one. It models how the system evolves in a per ECAC basis including the occurrence of faults, the consequences caused by them, the actions carried out by each of the components, and finally, whether the whole system fails. The last action of the model will be used to measure the system reliability.

The main model is composed of 4 modules. Same as the previous reset model, there are 3 instances of *Node Replica* module and 1 instance of each *Switches* and *Interlinks* modules. However, the main model has one more module, *System Failure Evaluation* module, that maintains a variable that indicates if the system has failed and that is going to be used when quantifying the achieved reliability, as will be seen later in Section 9.3.

The periodic execution of the main model is illustrated in Figure 9.6.

Steps 0 to 5 model the occurrence of permanent and transient faults in the current ECAC, excluding messages omissions due to bit errors affecting the frames.

Same as in reset model, step 0 models the permanent failures of the FTTRS components in current ECAC: links, switches and interlinks. These components belong to *Node Replica*, *Switches* and *Interlinks* modules respectively.

Step 1 is an evaluation step in which it is assessed if the system fails due to the fault occurrences modeled in step 0, if any. Additionally, this step also evaluates if node replicas are permanently faulty due to their inability to communicate with the rest, i.e. analogous to step1 from the previously described reset model there can be

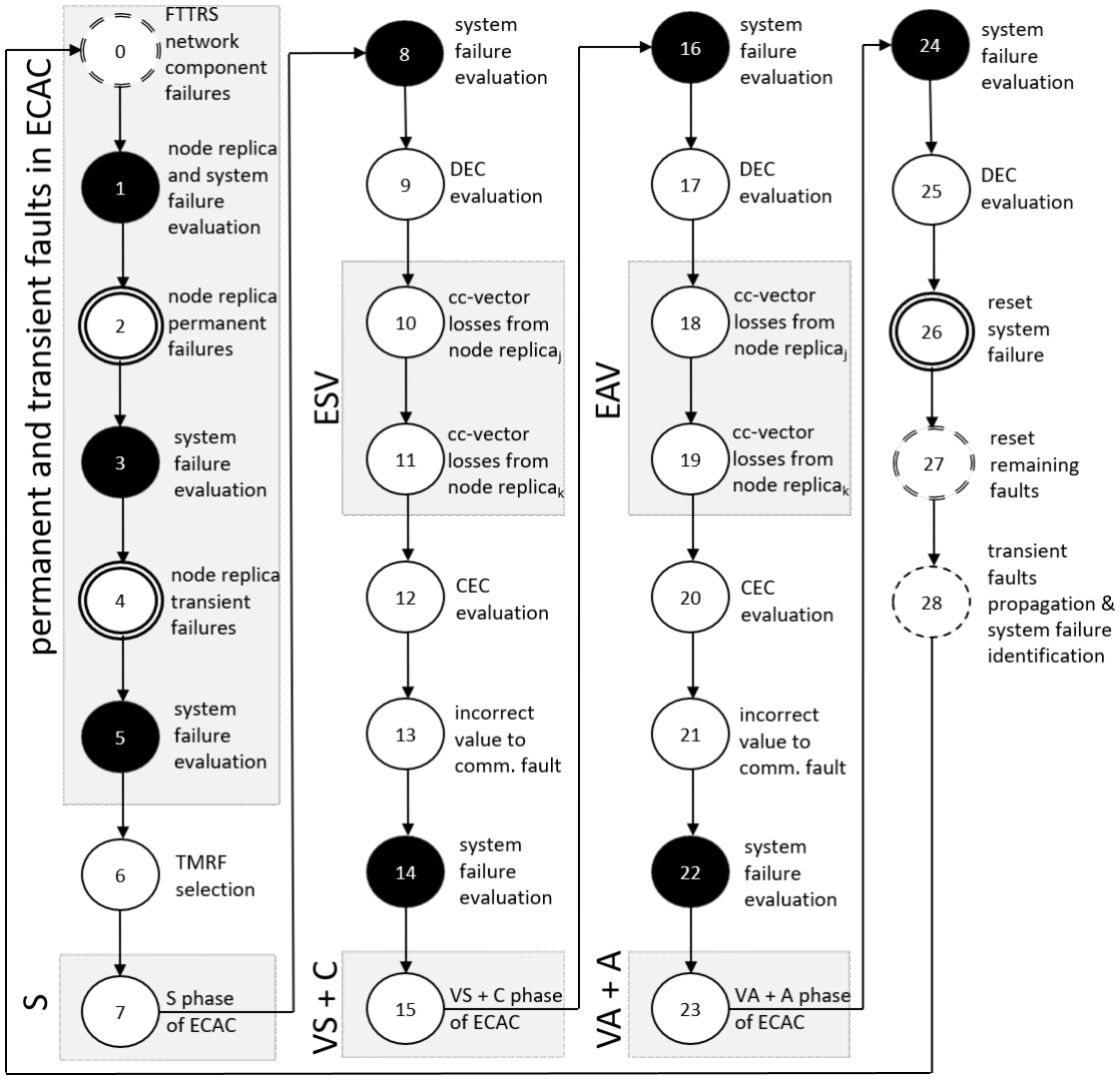


FIGURE 9.6: main PRISM model

only 1 link and 1 switch replica left but are not connected. Also, both links might have failed permanently.

Steps 2 and 4 model permanent and transient node replica failures respectively. Recall that these are the failure steps explained in the beginning of Section 9.2.

Like the first part of Step 1, Steps 3 and 5 are also evaluation steps that assess if the system fails due to all the fault occurrences so far. In case the system has failed, all the relevant local variables are set to certain values that disable the paths in the model taken in case of no failure. As explained before, this reduces the number of possible transitions and speeds up the model execution. It has to be noted that steps 8, 14, 22 and 24 do exactly the same thing, but include the evaluation of more scenarios that can lead to the system failure, e.g. failure to meet the MFA due to message omissions and/or unsuccessful majority voting.

Step 6 is used by the *Replica Node* module instances to select one of the precalucated probabilities of TM message losses (TMRF) depending on the current network configuration (see Section 9.2.1), which results from the faults modeled throughout steps 0 to 5, if any. This probability is used in the steps that model ECAC phases that are executed by the node replicas (S, VS + C, VA + A). These phases might not be triggered if a replica does not receive at least one TM from the switches and as a result no output can be produced.

Step 7 models the outcome of the *Node Replica* module after performing the S phase of the ECAC. Analogously, Steps 15 and 23 model the outcome of the *Node Replica* module after performing the VS + C and VA + A phases of the ECAC respectively. As previously explained, each of these steps uses the TMRF selected in Step 6.

Steps 9, 17, and 25 model the permanent fault detection due to the DEC reaching its threshold because of too many transient faults. As a result the replica is reset. These steps are located after Step 7 (S), Step 15 (VS + C) and Step 23 (VA + A). If the output of these ECAC phases is incorrect due to detected discrepancies in the votings, the DEC increases and it is evaluated if its threshold has been reached. In particular, discrepancy in the voting is detected if the locally produced cc-vector from a node replica differs from the consensus cc-vector obtained after the majority voting, or if the majority voting was unsuccessful, i.e. there was no majority. After Step 7 (S), we can immediately know that the locally produced sensor value is wrong, i.e. there will be a discrepancy. After Step 15 (VS + C), we know that the consensus value was wrong (unsuccessful voting). Note that this is also the input to the next voting. Thus, there was a discrepancy in the current voting and will also exist in the next voting. Lastly, after Step 23(VA + A), we have to check if there was a discrepancy due to the unsuccessful voting.

Steps 10 and 11, and 18 and 19 model the cc-vector reception losses (CCRFs) occurred in the 1st and the 2nd VCR (ESV, EAV) respectively. For the sake of clarity, we will refer the node replica modeled by each instance as rep_i, rep_j and rep_k. By default we describe the behavior of the *Node Replica* module instance that models rep_i. Specifically, after a VCR, a replica rep_i can fail to receive all the cc-vector copies from the other two replicas rep_j and rep_k as a result of faults affecting the messages being transmitted/received. This is represented by the variables, $ccVEctRX_i, j$ and $ccVEctRX_i, k$, which respectively indicate whether or not rep_i successfully receives at least one copy of the cc-vector of rep_j and rep_k. Recall, that this was already modeled by the VCR model described in Section 9.2.2. Therefore, to make these to decisions each *Node Replica* module instance takes into account the current network configuration and the corresponding probabilities precalculated by the VCR model.

Steps 12 and 20 model the permanent fault detection due to the CEC reaching its threshold because of too many transient faults. Similarly like with DEC, as a result the replica will be reset. These steps are located after each of the VCR steps. If the output of these steps is incorrect, i.e. cc-vector messages were lost due to communication faults, the CEC increases and it is evaluated if its threshold has been reached.

Steps 13 and 21 evaluate the values of the cc-vectors prior to the VCR round and transform the incorrect ones into the failure to receive a corresponding message conveying that cc-vector by updating the variables $ccVEctRX_i, j$ and $ccVEctRX_i, k$. If a cc-vector value was incorrect before the VCR, this wrong value would have been propagated to the rest of the replicas and the implication is the same as if the message was not received, i.e. the value will be incorrect for the voting procedure that follows. This simplifies the model since, otherwise, we would need to distinguish between replicas not receiving a cc-vector and replicas receiving a cc-vector that carries an incorrect value.

Step 26 and 27 evaluate the faults that occurred during a reset of a replica and if these faults caused the system failure. First, in Step 26 we model the occurrence of a system failure during a reset of one replica by using one of the probabilities calculated by the reset model described in Section 9.2.3 depending on the current network configuration. Next, in Step 27 we evaluate the remaining faults that were not covered by the reset model, e.g. if during a reset of one replica, the system has not failed but one of two links connecting a node replica to the switch has failed.

Note that a rest occurs either after the TNFP occurrence modeled in Step 4 or after DEC or CEC evaluation modeled in Steps 9, 17, and 25, or Steps 12 and 20, respectively. The first reset includes diagnosis, reset and reintegration time while the last two include only reset and reintegration time. However, since the reset time is orders of magnitude higher that the diagnosis and reintegration times combined, this is not an issue for the model.

Step 28 models the propagation of the effects of transient node faults to the next ECAC round. In particular, it can happen that transient node faults occur after the VA phase. If this is the case, a node replica can have its operational state corrupted and it can only be recovered in the next ECAC, in the first voting round, VS. Moreover, since this is the last step, we use it to execute in synchrony the module that evaluates if the system has failed, the *System Failure Evaluation* module. If it has, its local variable $sysFail$ of the *System Failure Evaluation* module will be updated to signify that the system has failed. This variable will be used to quantify the achieved system reliability described in the next Section.

9.3 Property verification

In this section we describe first how we use PRISM property specification language to obtain the properties that represent the output of the auxiliary VCR and reset models and then we demonstrate how to use the PRISM feature of transient probability calculation to measure the reliability achieved by our system modeled with the main model.

In the VCR model we obtain the outputs for each network configuration by calculating what are the probabilities of each node replica failing to receive cc-vectors from the other node replica at the end of the VCR. Each instance of the modeled node replica module has 2 local variables that represent if the cc-vectors from other 2 node replicas have been received. Also, each instance of node replica module has

variables that specify if the VCR has ended. To calculate one of the aforementioned probabilities we specify the query as follows:

```
P=? [ F vcrEnd = true & ccVectorReceive12 = 0 ]
```

The query specifies what is the probability of replica 1 eventually at the end of the VCR ($vcrEnd = true$) failing to receive cc-vector from replica 2 ($ccVectorReceive12 = 0$).

In the reset model we obtain the outputs for each network configuration by calculating the probability of the system failure within a predefined number of discrete time units.

The reset time has to be converted in the number of discrete time units and it is done as follows:

```
reset_time = X s
duration_ECAC = Y s
steps_ECAC = 4
discrete_time_units = floor(reset_time/duration_ECAC) * steps_ECAC
```

Reset time and the duration of ECAC are both expressed in the number of seconds. The reset model executes 4 discrete time steps (0-3) in each ECAC phase. First, dividing the reset time by the duration of ECAC and taking the floor value, we obtain how many ECAC phases does the model execute. Then, multiplying the number of ECAC phases with the number of discrete time steps executed by each ECAC phase, we obtain the number of discrete time units that the model executes.

The instances of node replica modules have variables that define if the system failure has occurred. Using these variables, the reset model calculates the system failure probability using the query below:

```
P=? [ F<=discrete_time_units sysFail1|sysFail2|sysFail3 ]
```

It is specified what is the probability of the system failure event detected by any instance of the node replica module eventually occurring ($sysFail1|sysFail2|sysFail3$) within a specified number of discrete time units ($discrete\_time\_units$).

Now that we have explained how we use PRISM property language to obtain the properties that represent the output of the auxiliary models we move on to describing how to measure our system reliability.

To obtain the reliability we use the PRISM feature of transient probability calculation. In particular, we execute the main model and obtain the probabilities for each possible global state of the model (combination of the values of the local variables of the included modules) every $step$ discrete time units starting from the $init$ discrete time units until a specified number of discrete time units $time$ is reached by executing the command bellow:

```
prism main_model.pm -tr init:step:time
```

The conversion from real to discrete time for the different times ($init$, $step$, $time$) used by the transient probability calculation can be done by applying the same logic as for the reset model. The only difference now is that the $steps\_ECAC$ variable is 29 since there are 29 steps executed by the main model in each ECAC phase.

Recall that in the main model we had a module that at the end of each ECAC phase evaluates if the systems has failed and sets a corresponding variable $sysFail$ to 1 if this was the case. The initial value of the $sysFail$ variable is 0.

Once the transient probabilities are obtained, the next step is to detect and sum up the probabilities where the local variable of the evaluation module $sysFail$ has a value of 1. This number represents the unreliability of our system, i.e. the probability that the system failure has occurred. Then, the reliability is simply calculated by subtracting from 1 the obtained unreliability.

Additionally, the main model does not consider the reset time. The steps executed by the main model including resets are considered to last one ECAC phase. However, whenever a replica is reset, a time in the duration of a predefined number of seconds has passed. Therefore, in order to precisely determine a total time of the model execution, we need to add the accumulated reset time to the specified model execution time.

We do this by defining a reward that associates a value 1 each time a replica is reset. Then by using cumulative reward properties we define a query following query:

```
R=? [ C<=execution_time ]
```

which gives us the expected number of resets within $execution\_time$ discrete time units. To determine total model execution time we add to the execution time the expected number of resets multiplied by the specified reset duration.

9.4 Results

This section presents the reliability results we have obtained. As will be seen, we shall show what is the measured value of the reliability throughout the pre-specified time, i.e. the value of the reliability function $R(t)$. We have considered both permanent and transient faults.

Note that the reliability results depend on the values given to the different parameters used by our model, e.g. failure rates, fault tolerance coverages, number and size of different messages, etc. Therefore, first, it is necessary to establish a *case of reference* where for each parameter we decide upon a reasonable/realistic value that should be used. Then, starting from this reference case we perform sensitivity analysis with respect to different parameters to see the effects they have on the measured reliability.

Ethernet is recently being introduced in many different industry branches replacing older network technologies. In particular, automotive industry is receiving a wide attention nowadays and therefore in the scope of automotive domain Ethernet can play a key role in substituting or coexisting with more traditional network technologies such as CAN. In this sense, we have decided to set up the case of reference parameters for automotive domain. In particular, we are interested in quantifying what would be the reliability benefits of our proposal when using components of commercial quality. Since those components do not strive to achieve high reliability goals, in our analyses we consider the reliability requirements for the least demanding automotive x-by-wire applications, such as throttle-by-wire (Morris and Koopman, 2005). Throttle-by-wire applications have a reliability requirement of ≥ 0.99999 during a mission time of 10 hours.

Next, we present the parameters that our model uses and for each parameter give a case of reference value.

The failure rates and Bit Error Ratio (BER) describing both permanent and transient failures of the components of our system in the scope of automotive domain are shown in Table 9.3.

TABLE 9.3: Automotive applications failure rates and BER

Rate Name:	Rate Value:
Perm. Node Fault Rate (FR) (Defense, 1995)	1E-5 faults/hr
Perm. Switch FR (Defense, 1995)	1E-6 faults/hr
Perm. Link and Inter-link FR (Defense, 1995)	1E-7 faults/hr
Power Supply FR (Defense, 1995)	1E-5 faults/hr
Transient Switch and Node FR (Peti et al., 2005)	1E-4 faults/hr
Bit Error Ratio (BER) (Stephens, 2004)	1E-6 err/bit

TABLE 9.4: The coverages used by the model of our system

Coverage Description:
Coverage of tolerating the occurrence of a replica reseting due to a DEC reaching its threshold caused by transient faults
Coverage of tolerating the occurrence of a replica reseting due to a CEC reaching its threshold caused by transient faults
Coverage of tolerating a permanent failure of a switch replica
Coverage of tolerating a permanent failure of a node replica
Coverage of tolerating a transient failure of a node replica
Coverage of tolerating switches synchronization when exchanging control messages
Coverage of a node replica tolerating a TNFP

Next, the parameters shown in Table 9.4 present the different coverages that we use in our system. They represent the probabilities with which certain FT mechanisms successfully tolerate intended faults. The first two coverages are are used for a specific scenario in which a replica resets after the detection of a "permanent fault" by either the discrepancy error counter (DEC) or communication error counter (CEC). However, this detection is triggered not by a permanent fault, but due to the occurrence of too many transient faults in the node replicas or in the links, excluding the transient faults in the nodes manifesting as permanent ones (TNFPs). These transient faults will trigger the unnecessary reset after being falsely diagnosed as a permanent ones by either DEC or CEC. The rest of the coverages are self-explanatory.

In our experiments we assume the same vales for all the presented coverages, and the value assumed is 99.9% even though it is a pessimistic assumption according to the following. In a typical highly-reliable system such as the Self-Repairing Flight Control System of a military aircraft the fault-tolerance coverages are of the order of 99,99% and 99,9992% (Wu, 2002).

Finally, in Table 9.5 we present the values of different parameters used by the PRISM model of our system. For the detailed list of models and modules that make use of these please refer to Appendix A.

For the size of the TMs and the Switch Sync messages we assume the lowest Ethernet frame size of 64 bytes. Since both of these message types do not convey a lot of information, we can safely assume that the Ethernet payload of 46 bytes within the chosen Ethernet frame size of 64 bytes will suffice to convey all the data. Switch

TABLE 9.5: The parameter used by the model of our system

Parameter Name:	Parameter Value:
TM frame size expressed in bytes	64
Switch Sync frame size expressed in bytes	64
CC-vector frame size expressed in bytes	512
# of TM replicas sent in one EC	4
# of Switch Sync replicas sent in one EC	4
# of cc-vector replicas sent in one EC	4
# of ECs constituting a VCR	3
EC duration expressed in seconds	0.001
ECAC duration expressed in the # of ECs	20
reset duration expressed in seconds	10
ratio of transient node fault effects propagating to the next ECAC	0.2
ratio of transient node faults manifesting as permanent ones	0.1
probability that a sensor device transiently fails in one ECAC	0

Sync messages are the messages exchanged among switch replicas as a means to enforce replica determinism in the value domain, as was described previously in Section 3.2.3. On the other hand, for cc-vectors we assume much larger frame size of 512 bytes. We have chosen this value due to the fact the cc-vectors have to convey the complete operational state of a node replica and we assume that 512 bytes should be more than enough to store this data.

For the number of TM replicas, Switch Sync replicas and cc-vector replicas sent in one EC we chose a value of 4 replicas. For the number of ECs that constitute a VCR we consider a value of 3 ECs. We assumed these values as intuitively reasonable ones, but, as will be seen later, we affirm this choice when doing sensitivity analysis.

The duration of the EC is chosen to be $1ms$ because the current technology used for implementing the switches and the HaRTES protocol that our underlaying network uses (see Section 3.1) confines the minimum size for the EC to this value.

For the duration of ECAC we decide to use the value of a sampling period of Ball-On-Plate balancing system (Awtar et al., 2002). This system was was used as an example when first designing our network-centric approach for coordination of task and message scheduling (Derasevic, Proenza, and Barranco, 2014). We kept using the values of this system thought our system design. This value was $20ms$, or 20 elementary cycles.

The duration of a node replica reset is 10 seconds. This was the average value of a reset measured in node replicas used in our implementation described in Chapter 7.

The ratio with which the effect of transient fault in the node replicas propagate to the next ECAC is 0.2 because of the following. There are 5 phases in ECAC when we use the merging of phases executed by controller nodes explained at the end of Chapter 6: S, ESV, VS + C, EAV, VA + A. If transient faults in the node replicas occur in the last phase (VA + A), the operational state of that replica might become corrupted and as a result this error can be compensated only in the next ECAC, in the first phase with voting (VS + C). Assuming that all phases are of equal duration and that transient fault distribution in node replicas is uniform, the proportion of their propagation is equal to the duration of one phase only (one fifth of the entire ECAC), which is 0.2 (See Figure 9.7).

The ratio of transient node faults manifesting as permanent ones (TNFPs) is assumed to be 0.1. We do not have any real knowledge of this parameter and think

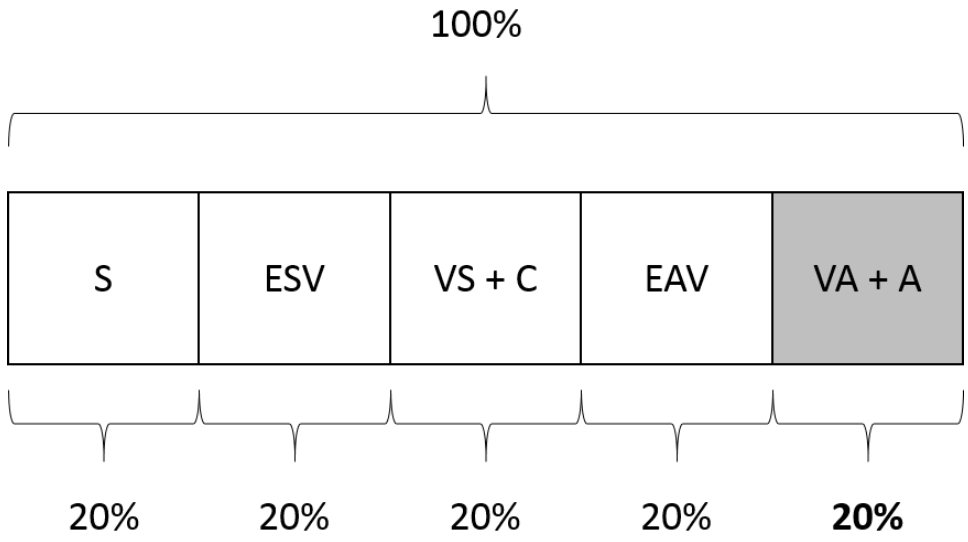


FIGURE 9.7: Merged phases of ECAC

that 10% is a realistic value. We will show later in this section when doing sensitivity analysis that this parameter has no significant impact on the achieved reliability and thus the precise decision on its value is not of considerable importance.

Lastly, we have a parameter that represents the probability that a sensor device transiently fails in an ECAC. This probability is set to 0 and is not currently being used by the model. This parameter was introduced as a means to model sensor transient failures. This is done since we have detected that our system can also deal with sensor transient faults, e.g. in case there is a single sensor connected to our system and it fails in a way that some of the node replicas manage to receive a correct value and some of the node replicas do not, if a majority of node replicas did receive the correct values, all replicas can, assuming there are no more faults, obtain the correct sensor values by means of DCMV, i.e. by receiving and voting on all sensor values. However, since we make no assumption of how sensor FT is achieved, i.e. by replication or some other technique, we left this feature unused.

Next step is to carry out the quantification of reliability for the values of the parameters presented above. As previously explained these values will be our case of reference. The aim of performed quantification is to, first, see if we meet the reliability goals for the chosen throttle-by-wire application of automotive industry, and second, to see what is the reliability gain when we compare our fault-tolerant design based on active node replication and FTTRS to a simple non-replicated design.

The non-replicated system that we use for comparison is quite simple and is modeled as a single node connected to a plant. Since there is no need for the communication infrastructure (links and switches of the FTTRS), only the node faults have to be considered for this model. Therefore, the non-replicated system fails on occurrence of any node fault: permanent or transient. For this simplex system we did not have to build any model. The reliability can be calculated with a simple expression: $R(t) = 1 - P(t) = 1 - (1 - e^{-\lambda t})$, which evaluates to $e^{-\lambda t}$, where λ is a sum of permanent and transient node failure rates.

As illustrated in Figure 9.8, the first measurement we performed shows the reliability results obtained for our fault-tolerant system with previously described case of reference parameter values and the reliability results of a simple non-replicated system. The y axis shows the measured reliability, i.e. the probability that the system

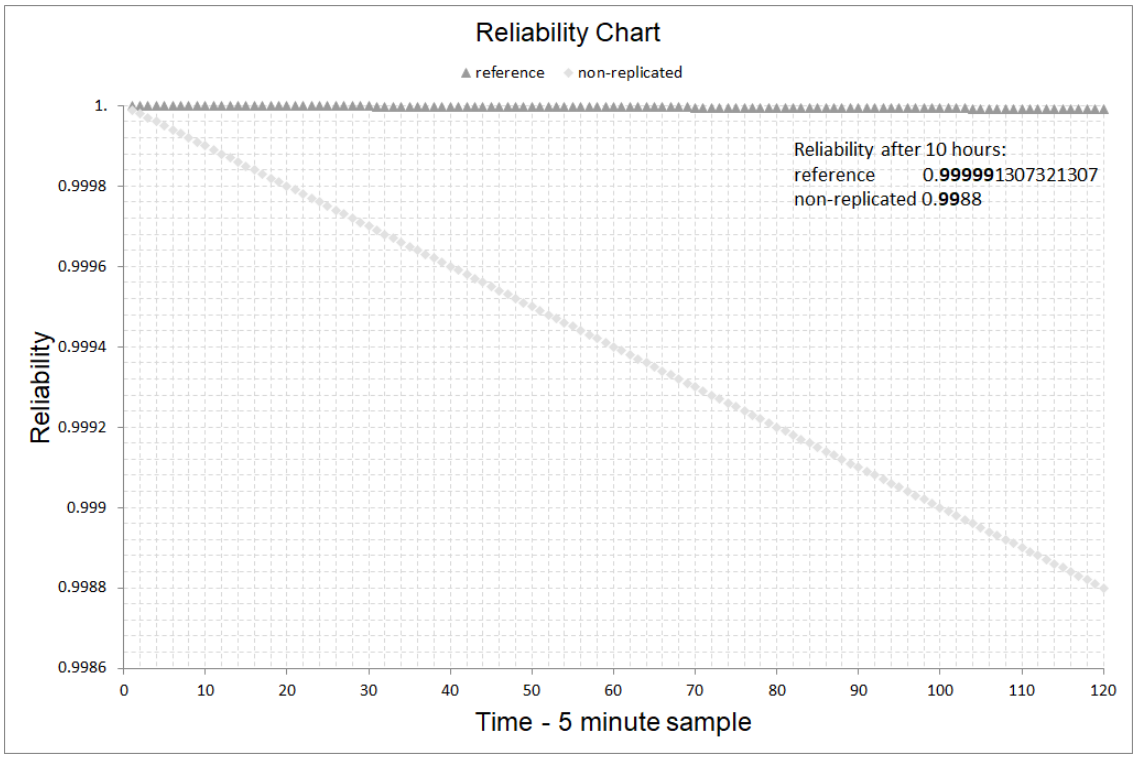


FIGURE 9.8: Reliability comparison between proposed fault-tolerant and non-replicated system

will not fail, and the x axis shows the time samples at which the reliability is measured. The scale for x axis is 5 minutes, e.g. the number 30 represents 150 minutes $(30 * 5)$.

The results show that with our system it is possible to meet the specified reliability goals for considered throttle-by-wire target applications. The reliability achieved by our system is higher than the required 0.99999 after a mission time of 10 hours. On the other hand, the non-replicated system's reliability after 10 hours is only 0.9988, which is quite low when compared to our system. Moreover, Figure 9.8 shows that the reliability of our system decreases slowly throughout 10 hour period whereas the non-replicated system's reliability decreases quite rapidly. Even after 5 minutes, which is the first measured sample, the measured non-replicated system's reliability is 0.999915, which is still less than the required 0.99999. With these results we can conclude that it is justifiable to have a fault-tolerant system design, and that with our design in particular it is possible to meet the reliability goals of throttle-by-wire automotive applications.

Furthermore, apart from measuring reliability for the pre-specified parameter values of our reference case, we also perform a sensitivity analysis, i.e. we define experiments varying the values of different parameters of our model to determine the impact they have on the achieved reliability. For the sensitivity analysis performed in this dissertation we chose to vary the values of parameters we were the least certain about. The list of different experiments and the goals of each are listed below:

- Experiment 1 - Varying the TM redundancy level. The goal of this experiment is to see how the reliability changes by changing the number or pro-actively transmitted TM replicas.

- Experiment 2 - Varying the cc-vector redundancy level. In this experiment we want to see how the reliability changes by modifying the total number or cc-vector replicas transmitted. This is done by simultaneously changing the number of cc-vectors transmitted in one EC and the number of ECs of a VCR.
- Experiment 3 - Varying the coverage values. By observing different set of values for all the specified coverages we want to examine how they affect reliability.
- Experiment 4 - Varying the ratio with which transient faults manifest as permanent ones. Since we do not have any concrete knowledge about this value, in this experiment we want to see its significance in the reliability results.
- Experiment 5 - Varying the component transient failure rate. Within the scope of automotive applications this value can actually range from $1E-3$ to $1E-4$ according to (Kopetz, 2004), so we want to see how these two different values influence the measured reliability.
- Experiment 6 - Partial disabling of reintegration. The goal of this experiment is to illustrate the importance of reintegration mechanism. However, due to the complexity of our model it was very difficult to add additional logic that would differentiate when is the reintegration needed and when simple voting suffices to tolerate transient faults in the links and nodes, i.e. it was impossible to extract reintegration mechanism only, due to state space explosion. Therefore, we have found an intermediate solution, and this solution was to modify the existing model to partially disable reintegration. Specifically, we have disabled reintegration only after the reset of a replica that can occur either due to TNFP or due to DEC or CEC reaching their thresholds. Thus, when reintegration is disabled, the replica that resets will not reintegrate and will stay permanently faulty. We then observe how this affects the achieved reliability.

Next, we present the results obtained by each of the above listed experiments and discuss them in more details.

The Figure 9.9 shows the results of Experiment 1 in which we vary the number of TM replicas being sent by the switches. Remember that for our reference case we chose a value of 4. In this experiment we are going to consider from 1 to 5 TM replicas in order to analyze the impact of this parameter on the reliability results and show why we chose 4 as the most adequate value.

First, the results show a significant difference between sending 1 and 2 TM replicas. With only 1 TM replica the results show that the reliability requirements for our test application cannot be satisfied, i.e. the reliability is much lower than 0.99999 after 10 hours of mission time. Moreover, note that the reliability measured when only 1 TM replica is being sent is even worse that the reliability achieved by the non-replicated system. However, in the case of 1 TM replica in the first 10 minutes (first two samples) the reliability is bigger than the required 0.99999 and then starting from the third sample it becomes lower, which was not the case with the non-replicated system that never achieved the reliability requirement, as was discussed in the first measurement.

Note that this observation is normal since the rates of transient faults affecting the messages are much higher than the rates affecting the hardware of the components. Recall that in the non-replicated system we did not have any communication. Moreover, in our fault-tolerant system the TM is crucial for providing both communication services and correct system operation. Therefore, if we do not tolerate

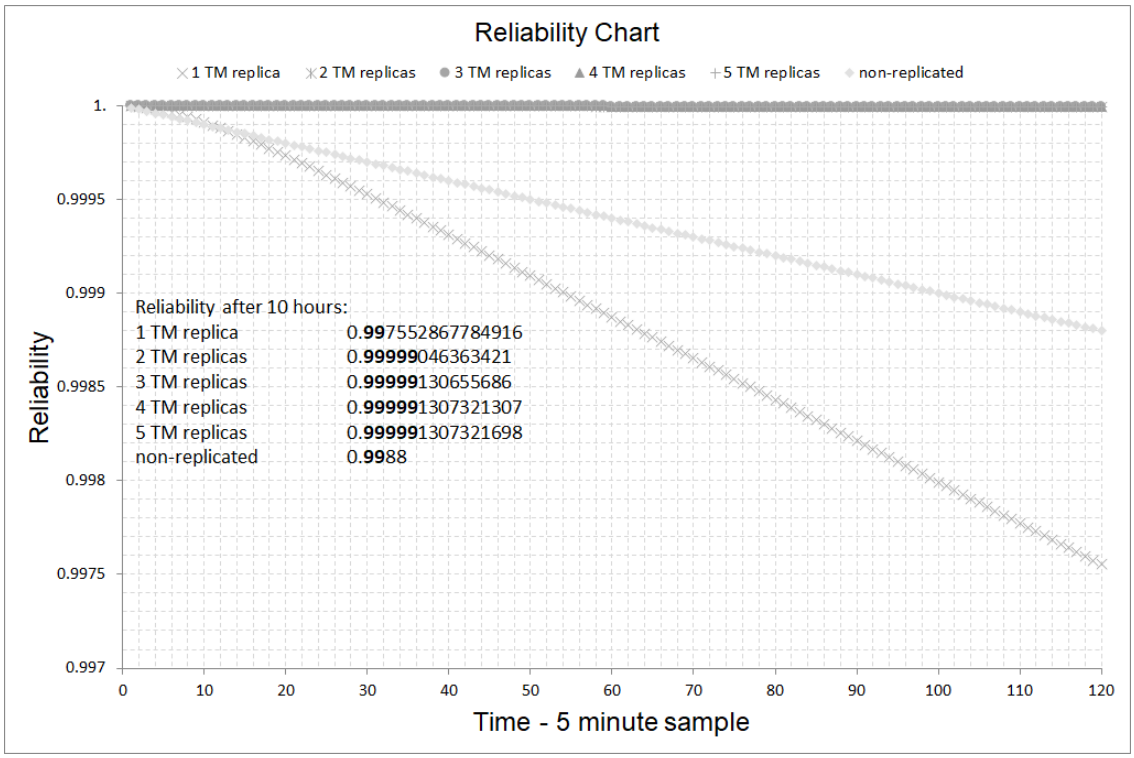


FIGURE 9.9: Experiment 1 - Varying the TM redundancy level

transient link faults affecting the TM transmission, we cannot achieve high reliability.

From then on, we further increase the number of TM replicas until and including the value 5. We can see that for each value larger than 1 TM replica the reliability requirement is satisfied. In particular, we can see that between 2 and 3 TM replicas the difference starts to appear in the 6th digit after the decimal, and between 3 and 4 TM replicas the difference start to be visible in the 9th digit. Lastly, the difference between 4 and 5 TM replica is in the 13th decimal digit. We chose the value of 4 TM replicas as our reference case value since the effect on reliability starts to be negligible at that point as compared to 3 TM replicas (9th digit).

In the next experiment, Experiment 2, we vary the number of cc-vector replicas together with the number of ECs that constitute a VCR to see how the overall cc-vector redundancy level affects the final reliability.

Note that the by increasing/decreasing overall cc-vector redundancy level we also increase/decrease the overall acknowledgments (ACKs) redundancy level due to the fact that for each transmitted cc-vector two acknowledgments are transmitted by the receiving replicas (see Chapter 5). However, since the only purpose of ACKs is to populate the MS vector non-diagonal cells that are used for diagnosing purposes by the switches, they are of little relevance for the reliability model and are not considered.

The results obtained by this experiment are illustrated in Figure 9.10. The values of our reference case were 4 cc-vector replicas sent in every EC of the VCR and the number of ECs of the VCR was chosen to be 3. In this experiment we vary both parameters starting from 1 cc-vector replica sent in a single EC during the VCR that has only 1 EC, then 2 cc-vector replicas sent in each EC of the VCR that has 2 ECs, then our reference case of 4 cc-vector replicas sent in each EC of the VCR that has 3 ECs, and finally 5 cc-vector replicas sent in each EC of the VCR that has 4 ECs.

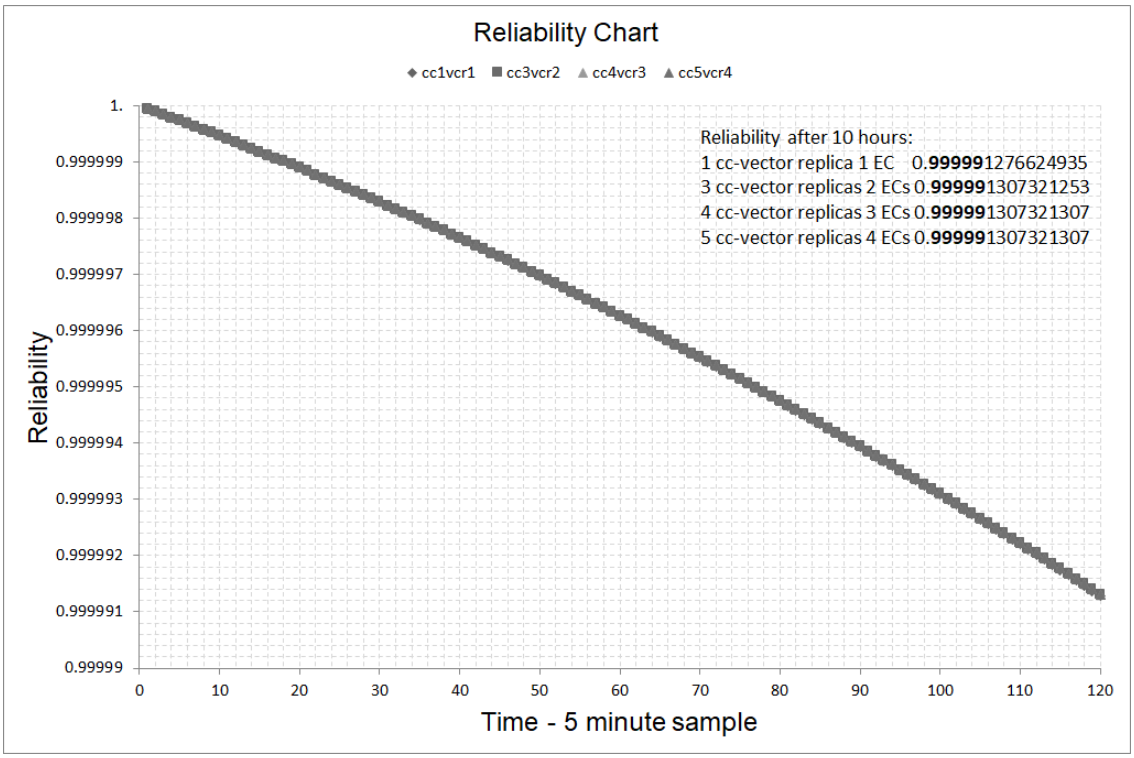


FIGURE 9.10: Experiment 2 - Varying the cc-vector redundancy level

The results show that for each of the chosen cc-vector redundancy level value the reliability requirement after a mission time of 10 hours is met. As regards the difference in reliability numbers we can see that for the first two measurements, 1 cc-vector replica sent in a single EC and 3 cc-vector replicas sent in each of the 2 ECs of the VCR, the difference is in the 8th decimal digit of reliability. The difference between the next two measurements, 3 cc-vector replicas with 2 ECs VCR and our reference case, 4 cc-vector replicas with 3 ECs VCR, shows in the 13th digit, and lastly, the difference between the last two measurements, our reference case and 5 cc-vector replicas with 4 ECs VCR, is not obtainable because the PRISM tool cannot display more than 16 decimal digits. Similarly like with the previous experiment with TM redundancy level we have decided that the difference in 13th digit is negligible.

Observing the results, we can conclude that the cc-vector redundancy level is not as important as TM redundancy level, e.g. if we consider the extreme values for both experiments, 1 TM replica and 1 cc-vector replica, we can see a huge difference in the reliability results. With 1 cc-vector replica we can achieve our reliability goal, whereas with 1 TM replica we cannot.

This observation is due to different facts. First, node replicas can recover from cc-vector losses using DCMV. Particularly, node replicas can, by exchanging and voting on cc-vectors, tolerate cc-vector losses as long as at the end of the VCR each node replica has at least a majority of non-faulty cc-vectors to vote with. Note that there is no equivalent mechanism for tolerating the loss of TMs. Second, note that TM replicas are sent in every single EC of ECAC as opposed to cc-vectors that are sent only in the ECs of VCRs. Therefore, node replicas are much more likely to become faulty due to TM losses then due to cc-vector losses. The above stated facts justify the results obtained for the cc-vector experiment as compared to the previous experiment with TMs.

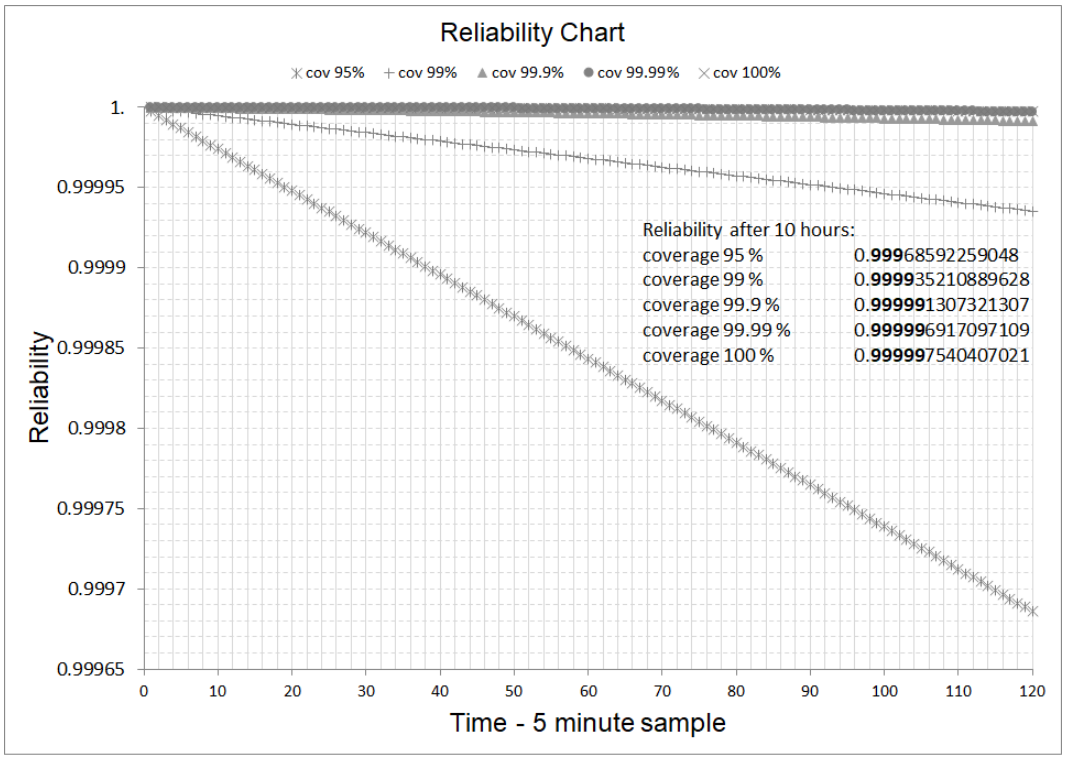


FIGURE 9.11: Experiment 3 - Varying the coverage values

Following is the Experiment 3 presented in Figure 9.11 in which we vary the coverage values. The results demonstrate that for the coverage values of 99.9% (reference case), 99.99% and 100% our system meets the reliability requirements for automotive throttle-by-wire applications and for the values of 99% and 95% it does not. As demonstrated the coverage values have a significant impact on the achieved reliability and it can be seen that even the small variations in these values can have a major impact on measured reliability.

Note that for choosing the reference value for this parameter we do not take the same approach as we did with TM replicas and cc-vector replicas. This is because for the previous two experiments the parameter values are a matter of the system configuration and for the parameter of this experiment, the coverage values probability, it is much more complex. Attaining a high coverage implies carrying out a lot of testing and refining aspects such as the accuracy and the quality of the error-detection mechanisms which is a demanding and time-consuming task that was not done by this dissertation.

Ensuing experiment, Experiment 4, is used to vary the ratio with which transient node faults manifest as permanent ones (TNFP). For a reference case value we chose a value of 10 %.

We have decided to vary this parameter starting with the value of 0% and increasing it every 10 % until 100 %. As illustrated in Figure 9.12 this parameter has no major effect on the reliability. We can see that for all the value variations the reliability goal is met. Therefore, our decision to choose 10 % as a reference case value will not have any major impact on the obtained results.

As regards the changes in reliability values, note that the 10th decimal digit is where the differences start to appear between different ratio values. Even in the extreme case of 100 % where every transient fault manifests as a permanent one, a reliability is not decreased by much.

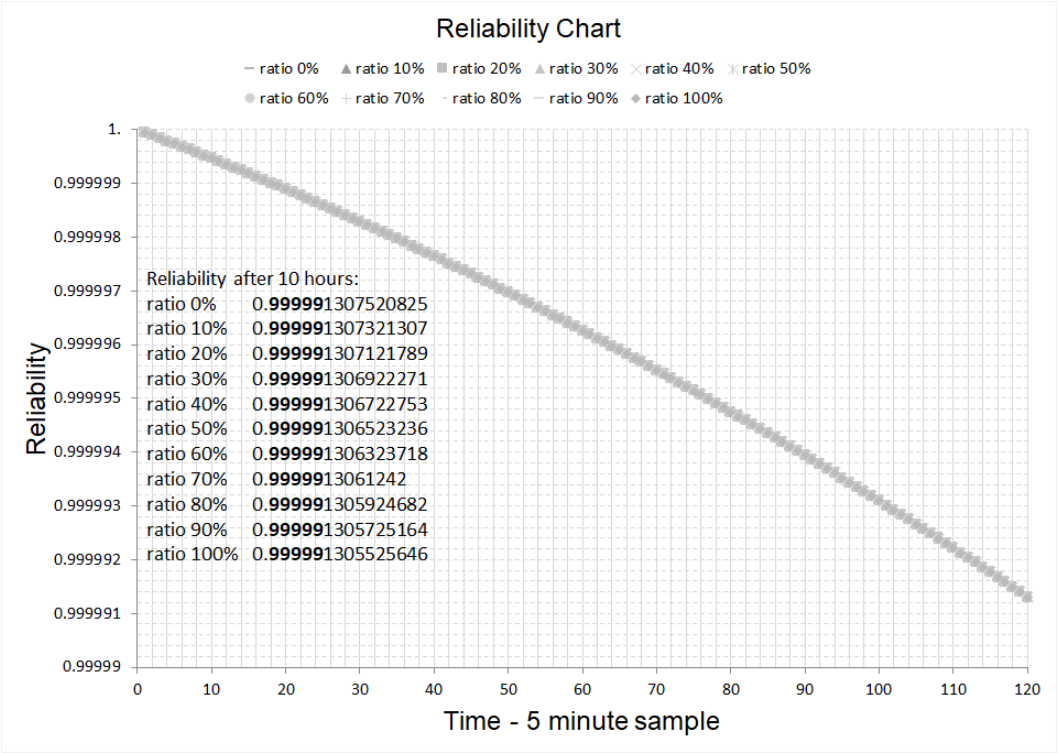


FIGURE 9.12: Experiment 4 - Varying the ratio with which transient faults manifest as permanent ones

These observations can be explained by the existence of reset and reintegration mechanisms which promptly recover node replica being affected by TNFPs. The reset and reintegration of a faulty replica happens fast enough to avoid more replicas from being affected by TNFPs thus preventing the system from failing.

Experiment 5 shows what happens when we decrease the failure rate of transient faults in the hardware of switches and node replicas by one order of magnitude. As displayed on Figure 9.13 we compare our reference case where this rate is $1E-4$ to the failure rate decreased by one order of magnitude, $1E-3$. As expected the value of this rate has a notable impact on the achieved reliability. Note that with the decreased failure rate our system would not be able to meet the reliability requirements of throttle-by wire automotive applications.

Conclusion is that the components used, nodes and switches in our case, have to be designed in such a way to achieve a specific failure rates by e.g. using techniques such as component shielding and radiation hardening.

In our last experiment, Experiment 6, we disable reintegration after a reset of a replica and compare the reliability results with a non-disabled model for different values of ratios with which transient faults manifest as permanent ones (TNFP), since these faults are one of the major causes of replica resets.

We chose the values of 0 %, 10 %, 50 % and 100 % for this experiment and for each of these values we compare the results of the model with partially disabled reintegration and the model with reintegration included.

As seen by Figure 9.14, for the first two values of TNFP, 0 % and 10 %, we can see that we still meet the reliability requirements both with and without reintegration. However, for the last two values, 50 % and 100 %, we can see that without reintegration we cannot meet our reliability requirement of 0.99999 after 10 hours. We can also observe that the greatest reliability difference between models with and

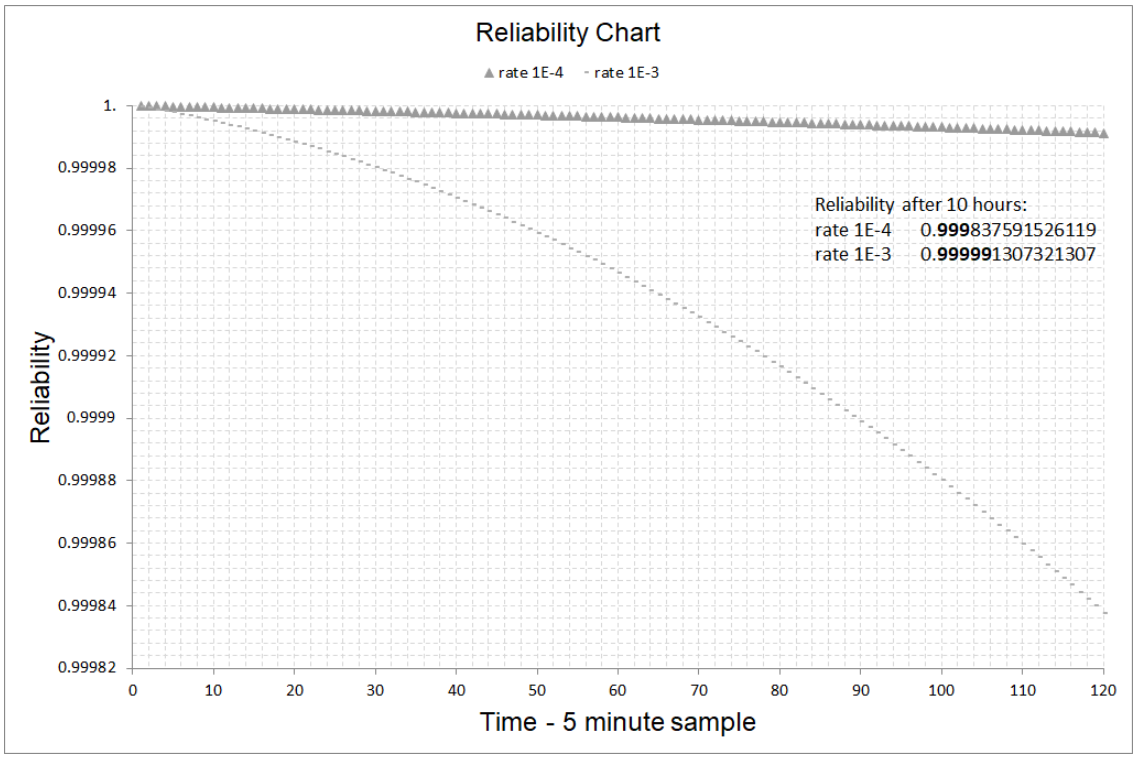


FIGURE 9.13: Experiment 5 - Varying the component transient failure rate

without reintegration is for the case that favors the occurrence of reset the most, i.e. for the value of TNFP ratio of 100 %.

Concluding, we can see from this experiment the importance that reintegration has in the reliability results even though we have only disabled it partially. Thus, the need for introduction of this FT mechanism is justified.

Examining all the experiments it has to be noted that transient faults in the channel are prevailing and are most important ones to deal with comparing to other rates. This is why we can see that the results of experiments focusing on varying the parameters that are more closely related to the rates affecting the system components, experiments 2, 4 and 6, do not show big fluctuation in reliability results.

Lastly, note the expected reset time was not included in the results since the values obtained were negligible compared to the mission time of 10 hours, i.e. the expected number of resets was never more than 0.5, even in the worst case reset experiments, i.e. the experiments that favor the probability of node replicas being reset. Recall that for the reset time of a node replica we assumed a value of 10 seconds.

Overall, we show in this section that it is important to provide tolerance to node faults if we want to achieve high reliability goals and we can see that some parameters of the system are more important than others.

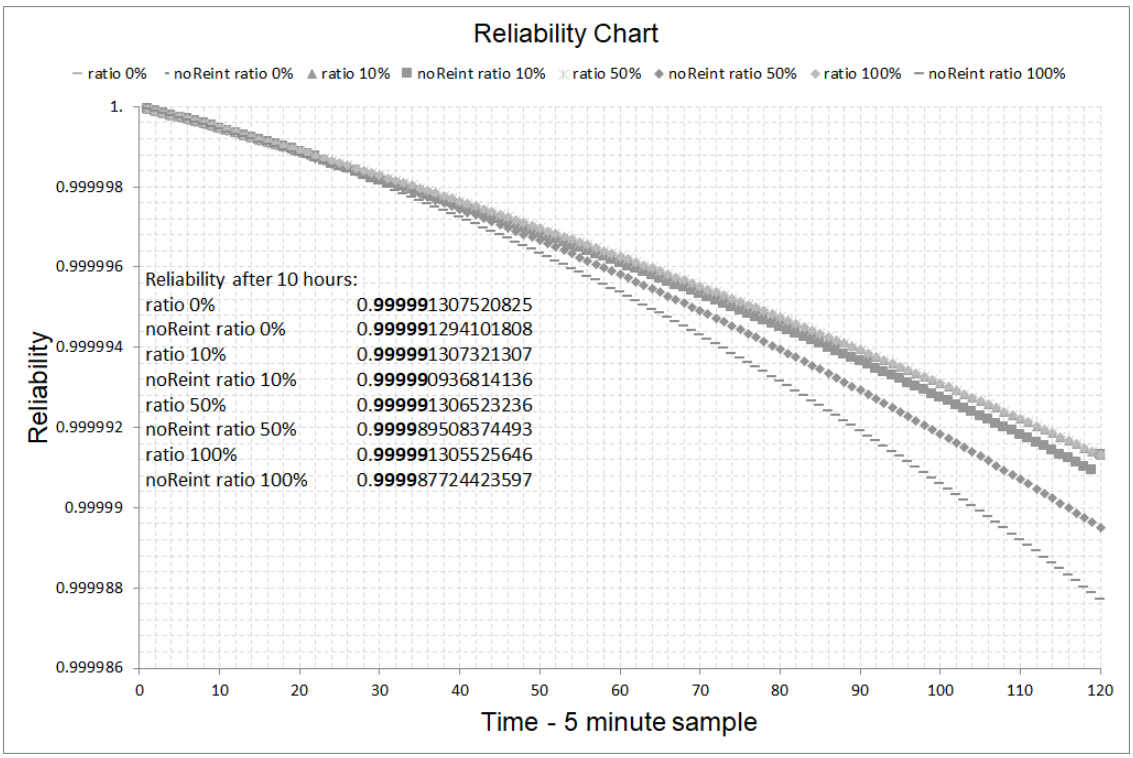


FIGURE 9.14: Experiment 6 - partial disabling of reintegration

Part III

Conclusions and Future Work

Chapter 10

Conclusions and future work

In this chapter we conclude the presented dissertation and give an overview of possible future developments.

10.1 Thesis validation, contributions and conclusions

The thesis of the present dissertation states that "*it is possible to attain high levels of reliability of adaptive critical RT DES that rely on a reliable and flexible RT communication subsystem based on an FTT implementation on Ethernet by providing FT mechanisms for the nodes.*".

By means of the work described in the current document we have proven this thesis and we have provided a series of novel contributions. Next, we both briefly argue how this work validates the thesis, and highlight its main contributions while focusing on the main conclusions that can be drawn from it.

Generally speaking, to prove the thesis this dissertation first proposes a node architecture that provides mechanisms to tolerate hardware faults affecting the nodes, and that relies on an existing fault-tolerant and flexible RT communication subsystem based on Ethernet. Second, the dissertation verifies, both via simulation and experimentally, the correct operation of these architecture's FT mechanisms, as well as their integration with the underlying communication subsystem. Finally, it proposes a dependability model of a DES relying on this node architecture that quantitatively demonstrates that the proposed FT mechanisms do increase the reliability of the DES.

In this sense, the first step to validate the thesis was to select an appropriate Ethernet implementation of the FTT paradigm. For this we chose the FTTRS communication subsystem due to the provision of the following features:

- FTTRS is based on HaRTES, which is a specific realization of the FTT paradigm on top of full-duplex Ethernet. Thanks to HaRTES, FTTRS provides flexible RT communication on top of Ethernet. In this way, on the one hand, FTTRS provides RT guarantees for traffics with different RT requirements (hard and soft). On the other hand, FTTRS supports operational flexibility by allowing the change, at runtime, of the communications RT requirements while keeping all RT constraints. This last property of HaRTES and thus of FTTRS is specially well-suited for supporting network adaptivity.
- FTTRS is the most reliable implementation of the FTT paradigm on top of Ethernet. FTTRS provides reliable FTT communication services based on the following FT mechanisms:
  - FTTRS relies on a duplicated star topology (two switches) to tolerate transient/permanent faults affecting any of the switches.

- Each node can connect to each one of the switches by means of an independent full-duplex link, so as to tolerate permanent hardware faults affecting its links.
- Both FTTRS switches are interconnected by means of more than one full-duplex link. This allows switches to exchange appropriate information to coordinate with each other and, also, provides redundant communication paths among the nodes.
- Each FTTRS switch is internally duplicated and compared. In this way each switch exhibits a crash failure semantic, which facilitates the design and integration of further FT mechanisms as the ones proposed here.
- Both FTTRS switches are replica determinate and, thus, provide reliable and consistent communication services. This is a property that any redundant network chosen for supporting our nodes and FT mechanisms must provide.
- Each FTTRS switch includes a Port Guardian (PG) for each one of the nodes connected to it. PGs filter out incorrect messages to enforce that, from the point of view of the non-faulty nodes, faulty ones exhibit an incorrect computation failure semantic. Again, this facilitates the design and integration of further FT mechanisms like the ones herein proposed.
- FTTRS provides a proactive retransmission service for critical messages. This allows tolerating transient hardware link faults that corrupt the messages being exchanged. In this sense, FTTRS further paves the way for our mechanisms towards achieving a high system reliability.
- Both switches of FTTRS pro-actively retransmit isochronous Trigger Messages (TMs) to reliably indicate the start of each communication cycle (Elementary Cycle, EC). On the one hand, this provides a highly reliable mechanism for nodes to synchronize with the start of each EC. On the other hand, it served us as a reliable time basis to both timely trigger the tasks to be executed at the nodes, and implement recovery (reintegration) mechanisms.

However, FTTRS does not provide any mechanism to tolerate faults that prevent a node from correctly communicating and/or operating. This is an important limitation for attaining a high system reliability. On the one hand, this is because nodes are normally the most unreliable elements of a DES. On the other hand, this is because a critical DES normally includes several nodes such that each one of them is fundamental for the DES to provide its intended service.

Therefore, the next step necessary to validate the thesis was to devise a set of mechanisms to tolerate these faults that prevent nodes from correctly communicating and/or operating.

Our first decision towards this purpose was to use an active node replicated architecture. We took this decision because by means of active node replication node failures can be timely and transparently masked so as to prevent them from jeopardizing the RT response of the system.

Following this approach and, in oder to make node replicas being able to compensate errors, it is necessary to provide FT mechanisms that allow them to be kept replica determinate and to reliably vote. For doing so, we first needed to define a fault model, to specify the failure semantics of the system elements and, finally, to exhaustively analyze to which extent the FT mechanisms of FTTRS already deal with faults.

As explained, we adopted the fault model of FTTRS, i.e. we aim at tolerating non-malicious operational hardware faults. Faults that then can manifest arbitrarily; except in the links, which are the only elements of the system that exhibit an omission failure semantic. However, we showed that on the one hand, since each FTTRS switch is internally duplicated and compared, we can rely on switches that exhibit a crash failure semantic. On the other hand, we also recalled that thanks to the port guardians of the FTTRS switches, faulty nodes exhibit an incorrect computation failure semantic from the point of view of non-faulty nodes, i.e. non-faulty nodes perceive faulty ones as either omitting messages or sending messages that carry incorrect data (payload).

To analyze the extent of the FT mechanisms of FTTRS, we thoroughly studied the effects of faults depending on their persistence, the elements they affect, and the way in which they manifest.

First, we showed that permanent hardware faults occurring in a switch and/or in the links may lead a node to not be able to communicate with the rest of the nodes any further. Similarly, permanent faults affecting the hardware of a node may permanently prevent that node from communicating; either because it cannot transmit any message or because it transmits incorrect messages that are then discarded by the corresponding PGs. Moreover, a node may not be able to correctly operate, e.g. compute, any longer. In any case, independently of whether the fault occurs in the channel or in the node hardware, in the worst case a permanently faulty node will be perceived by the non-faulty ones as a node that sends messages that carry incorrect data (payload) from the application point of view, e.g. an erroneous sensor/actuation value. FTTRS provides no mechanism to tolerate any of these situations.

Second, as regards temporary hardware faults, the current proposal of FTTRS transforms temporary hardware faults occurring at a switch into a permanent crash failure of that switch. Thus, we considered those faults as permanent ones, since proposing mechanisms for a switch to recover from a temporary hardware fault is out of the scope of the current work. Concerning temporary hardware faults occurring in the links or in the nodes themselves, we found out that the persistence of these faults may have different impacts on the nodes ability to operate and/or communicate. Thus, in order to appropriately identify what are the necessary FT mechanisms in each case, we further classified temporary faults as *transient* (affecting the links or the nodes hardware), *Transient Long Lasting Faults affecting Links* (TLLFL), and *Transient Faults affecting the Nodes manifesting as Permanent ones* (TFNP).

Transient link faults may transiently affect the capacity of a node for transmitting/receiving, but they are transparently tolerated by using the pro-active retransmission mechanism already provided by FTTRS. Thus, we did not need to propose any further mechanism to cope with those faults.

Conversely, TLLFLs are transient faults in the links whose duration exceed the FT capacity provided by the pro-active retransmission mechanisms of FTTRS. Due to TLLFLs, a node may not be able to send messages to the non-faulty ones. Also, a node may not receive messages that are necessary for it to be kept synchronized with the non-faulty nodes from the point of view of the communication and/or the application. This means that a node may become unable to correctly operate from then on. In any case, again, in the worst case a node affected by a TLLFL will be perceived by the non-faulty ones as if it were affected by a permanent fault, i.e. as omitting messages or as sending messages with incorrect data. FTTRS does not provide any mechanism to cope with TLLFLs.

Transient (hardware) node faults are those that may transiently prevent a node from communicating, or compel a node to transiently carry out incorrect operations.

Certainly, thanks to the FTTRS pro-active retransmission mechanisms, an affected node may tolerate these faults and correctly transmit/receive. However, the pro-active retransmission mechanisms of FTTRS are originally thought to tolerate transient faults in the links. Thus, the level of redundancy of these mechanisms, i.e. the number of pro-active retransmissions, should be calculated taking into account the rate with which messages may be corrupted at the links, rather than the rate with which nodes suffer from transient faults. Furthermore, in any case, these pro-active retransmission mechanisms are useless to tolerate transient node faults that lead nodes to perform incorrect operations. This is especially important to solve, because a node that carries out even a single incorrect operation may not only send messages with incorrect data, but may also become desynchronized at the communication and/or application levels and, thus, may also become unable to correctly operate any further.

Finally, TFNP faults are transient faults affecting the hardware of a node itself such that the node manifests as permanently faulty, unless the node is reset. In other words, the node needs to be reset prior to be able to appropriately re-synchronize, with the non-faulty nodes, from the communication and application points of view. FTTRS does not propose any mechanisms for dealing with TFNPs.

Taking into account this analysis, we proposed a voting mechanism for node replicas we call the *Distributed Consistent Majority Voting* (DCMV). The main advantage of the DCMV is its simplicity, since in principle each node replica only needs to locally vote (majority voting) on the result it obtains locally and the ones it receives from the other node replicas. This simplicity is achieved since, thanks to FTTRS, non-faulty nodes perceive faulty ones as proposing either no or an incorrect value for voting.

In any case, for DCMV to succeed, it is necessary to enforce that node replicas are replica determinate (internally and externally). On the one hand, internal replica determinism is enforced by using the same hardware and software constructs by each node replica. This is not a novel idea of this dissertation, but a well-known strategy. On the other hand, for enforcing that node replicas are externally replica determinate, we included within the DCMV a mechanism to guarantee, with high probability, that non-faulty replicas successfully exchange a consistent majority set of correct values to vote on. This mechanism is essentially a pro-active retransmission mechanism, called *Cc-vector Exchange Protocol* (CVEP), that further increases the redundancy given by the pro-active retransmission mechanisms provided by FTTRS. One of the advantages of CVEP is that it performs pro-active retransmissions in each one of multiple ECs, so as to avoid that TLLFLs, .e.g. bursts produced by electromagnetic disturbances, may prevent non-faulty replicas to successfully exchange the values they use to vote on. The other advantage of the CVEP is that it served as a basis to implement fault-diagnosis mechanism, as we will recall later on.

At this point, it is important to note that the replicated node architecture and the DCMV itself would be enough to tolerate faults that prevent a node from communicating/operating. Nevertheless, as pointed out above, temporary faults in the channel or in a node replica itself can actually lead that replica to irremediably fail, even though it is not affected by a permanent fault. This may lead to a node redundancy attrition problem that ultimately limits system reliability and, thus, the benefits of the redundancy investment.

To overcome this limitation, in this dissertation we proposed a series of mechanisms to thoroughly prevent temporary faults from making node replicas to be perceived as permanently faulty.

In this sense, the first aspect that is worth to highlight is that the DCMV itself already provides a simple and well-known form of *Forward Error Recovery* (FER). Specifically, a node replica can use the result of the voting to correct the value it proposed for voting in case that proposed value was not correct.

However, faults may lead a node replica to become desynchronized at the communication and/or the application level beyond the error recovery capacity of the just mentioned FER. Thus, we realized that it was necessary to propose more sophisticated recovery mechanisms to prevent undesirable fault attrition. We refer these advanced recovery mechanisms to as *reintegration mechanisms*.

The first one of these mechanisms is called *TM resynchronization*, and it allows node replicas to resynchronize at the level of the communication. The TM resynchronization is based on the fact that we propose to divide the application, being executed at the node replicas, into different phases or tasks that are triggered synchronously in every node replica. More specifically, in order to force this synchronism among node replicas we decided to use a network-centric approach similar to the one presented in (Calha and Fonseca, 2002; Silva et al., 2005). In this sense, the idea of dividing the application in phases triggered by the Trigger Message (TM) the FTT provides is not a contribution of this dissertation. However, what is novel is the way we proposed for node replicas to use the TM to trigger the phases. Basically, what our idea proposes is that node replicas use the *Trigger Message Sequence Number* (TMSN) to appropriately trigger the different phases. In this way, conversely to what happens in (Calha and Fonseca, 2002; Silva et al., 2005), the use of the TM we propose for triggering the phases does not require the network subsystem from being aware of the application executed on top of it. Given this strategy for triggering the application phases, what we propose by means of the TM resynchronization is a way for a node replica to use the TMSN to re-resynchronize at the level of the communications and, then, to regain its ability to trigger the application phases in synchrony with the non-faulty replicas.

The second reintegration mechanism we proposed is called the *Voting Reintegration Point*. This mechanism takes advantage of the DCMV to exchange and vote on not only specific intermediate results, but also to exchange and vote on all the variables that compose the state of their computation, i.e. all the variables of their operational state. When to exchange the variables and vote on them is application dependant. In this sense, it is not mandatory that node replicas exchange and vote on their operational state every time they exchange and vote on an intermediate result. Doing so could be feasible for control applications, since their operational state is usually small. To demonstrate this suitability for control applications, in this dissertation we have proposed different examples of how the control cycle can be divided into phases to benefit from the use of the Voting Reintegration Point mechanism. For other applications that have greater operational states, the CVEP could be extended so that replicas can use many communication rounds (even non consecutive) to exchange all the values needed to vote on; or replicas may exchange and vote on at least the variables that are strictly necessary to keep themselves replica determinate as concerns their most critical functions.

Note that the TM resynchronization and the Voting Reintegration Point mechanisms allow reintegrating a node replica that suffered from any temporary fault, except from a TFNP. This is because, as explained above, a TNFP may lead a replica to behave as if it was permanently faulty as long as it is not reset. In other words, as said above, a replica that suffers from a TFNP needs to reset prior to be able to undertake any reintegration mechanism.

In order to treat TFNPs we proposed a set of *fault-diagnosis* mechanisms for detecting when a replica is affected by this kind of faults and, then, to force it to resume and reintegrate.

First, we propose each node replica to locally manage what we call a *Discrepancy Error Counter* (DEC) to diagnose when, after having accumulated too many computation errors, itself seems to be affected by a TFNP. Basically, the node replica increases its DEC when it detects a discrepancy between the value/s it proposes for a given voting and the value/s that result from that voting. Whenever the DEC reaches a given threshold, the node resets itself and carries out the necessary reintegration procedures.

Second, we propose each node replica to locally manage a *Communication Error Counter* (CEC) to diagnose when, after having accumulated too many communication errors, itself seems to be affected by a TFNP. The management of these counters builds upon the CVEP, which compels node replicas to pro-actively retransmit acknowledge messages (ACKs) for reliably confirming the correct reception of the messages sent from the other node replicas. The FTT switches use these ACKs to consistently fill up what we call the *Messages Status* (MS) vector, .i.e. a matrix that specifies for each node replica whether or not that replica correctly sent its message to any of the switches and whether or not each one of the other node replicas acknowledged it. Switches send the MS-vector to all node replicas every time replicas execute the CVEP. In this way, each node replica can use the MS-vector to update its local CEC and reset itself if that CEC reaches a specific threshold.

There may be situations in which a node replica fails to correctly manage its local CEC or to correctly reset when necessary. To cope with these situations, both FTT switches manage their own set of CECs, using the MS-vector they consistently fill up. When a given CEC within the switches reaches a threshold, the switches themselves diagnose the corresponding node replica as being affected by a TFNP and, then, they reliably send a reset command to that replica.

Note that we decided not to include DECs within the switches. This is because doing so would require switches, i.e. the communication subsystem, to be aware of details that belong to the application itself, .e.g. the variables that compose the operational state.

In any case, a node replica may still fail to reset either when its local DEC/CEC reaches a threshold or when the switches instruct it to do so. To overcome this limitation we proposed a mechanism called *You Are Alive* (YAA) watchdog timer. The idea basically consists in the switches periodically sending a You Are Alive (YAA) message to every node replica they have not diagnosed as faulty. The node replica must forward this message to a dedicated YAA watchdog timer attached to (but independent from) it. This timer resets the node replica after not receiving any YAA message during a given interval of time. In order to prevent a node replica from forging the YAA message, the switches include within each YAA message a code that is dynamically updated in a way that only the switches and the YAA watchdog timer know.

Given all these mechanisms we proposed, the next step to validate the thesis was to demonstrate both that they are correct and that these mechanisms can be integrated with those already proposed by FTTRS.

We demonstrated the first one of these aspects by means of a simulation model and by means of two real prototypes.

The simulation model was primarily intended to verify the correctness of our mechanisms when considering all the temporary faults described above. For this purpose, we used as a basis an OMNET++ simulation model proposed prior to

the present work and that models an enhanced version of HaRTES. We extended this model to include all our mechanisms and, then, to carry out a series of simulated fault-injecting experiments that thoroughly verify the correctness of the mechanisms. As we explained in the corresponding chapter, the fact of having simulated our mechanisms on an enhanced-version of HaRTES does not limit the validity of the verification results. Moreover, it further shows that although our mechanisms can take advantage of FTTRS to rely on a highly-reliable communication subsystem, they can also be used on top of other FTT realizations. In fact, some of our mechanisms, e.g. the YAA watchdog timer mechanism, could be used in other communication paradigms as well.

Concerning the verification by means of the real prototypes, we first supervised the implementation of our mechanisms on them. Then, we also supervised the design and execution of the fault-injection tests we conducted with those prototypes to experimentally verify the correctness of the mechanisms. Those tests were basically the same as the ones conducted via simulation, but including also permanent faults. Additionally, these tests allowed us to acquire statistics about the time a faulty node replica needs to recover/reintegrate. These statistics further demonstrate that this time should be low enough to consider as negligible the probability of extra temporary faults affecting a node replica while, at the same time, another replica is temporary unavailable due to previous near-coincident temporary faults.

It is important to highlight that these tests were conducted on a set of triplicated critical hard RT nodes that share the FTTRS network with non-critical soft RT nodes. Thus, the experiments carried out with the real prototypes also allowed us to experimentally demonstrate that our mechanisms can be integrated with the ones of FTTRS. In this way we proved that it is possible to provide node FT mechanism in a mix-critical DES in which critical nodes co-exist with non-critical ones, and where all of them communicate on a reliable and flexible RT network.

The last step to validate our thesis was to quantitatively demonstrate that our mechanisms allow attaining a high reliability level for a DES that rely on FTTRS.

For this purpose we used a model checker tool, called PRISM, to model and quantify the reliability of a DES that executes a control application while relying on our FT mechanisms and FTTRS. We build this model using the *Discrete Time Markov Chain* (DTMC) formalism, which is well suited for periodic execution of control applications. The characterization of the system aspects that may influence the reliability, e.g. the components failure rates, as well as the targeted reliability requirements were chosen so as to reflect the reliability properties and needs of throttle-by wire applications in the context of the commercial automotive industry. In any case, we also performed a set of sensitivity analyses with respect to these properties and, also, with respect to aspects related to our FT mechanisms (e.g. the probability of success of certain reintegration mechanisms) and to the FT mechanisms of FTTRS (e.g. the number or pro-actively retransmitted TMs). On the one hand, the reliability figures we obtained proved that the reliability requirements of these applications can be met when they rely on our FT mechanisms and FTTRS. On the other hand, we also proved by comparing our system with a non-replicated one, that without our node FT mechanisms the reliability would significantly decrease thus justifying the need for the implementation of our mechanisms.

To conclude we can ascertain that the work presented throughout the present dissertation proves our thesis statement by means of several analyses and constructs; tests carried out via simulation and on real prototype implementations; and a model that quantifies the achievable system reliability.

10.2 Future Work

Next, we outline a series of potential developments we identified in this dissertation, which can be considered by future work.

- Note that adaptivity, i.e. the ability of the system to adapt to the change of system requirements, was only supported at the network level by means of operational flexibility. In other words, our system only supports adaptivity in terms of the flexibility that FTTRS already provides for mixing different kinds of RT traffic, as well as for modifying at run time the traffic scheduling without jeopardizing the desired real-time constraints. Adaptivity of the nodes' application execution was not addressed by this dissertation and is one of directions to take in order to extend the work herein presented.

- Instinctively, the concept of adaptivity can be considered even for the applied FT mechanisms. FT can be dynamically changed depending on the system requirements, e.g. the number of node replicas, the replication strategy applied, the number of pro-actively sent messages can all be dynamically adjusted as the system changes the environment and requires more or less FT. We have already considered this partially in one of our previous works, (Derasevic, Proenza, and Gessner, 2013), but this dissertation does not address this topic.

- One future task can be devoted to exploring the possibility of adding redundancy preservation, i.e. trying to maintain the same pre-established redundancy level, as a means to further increase the reliability achieved. This idea would be applied to actively replicated nodes, when one node replica permanently fails and eventually gets disconnected either by using another node in the system or by replacing the failed one to preserve the redundancy.

- Another potential point for further development would be to broaden the fault model and consider more faults. Then, this would open possibility for implementing further FT mechanism to deal with newly considered faults and their effects.

- The idea of VSUA presented in Section 5.3.4 was abandoned due to inability to determine which node replicas and cc-vectors should be taken into consideration for majority voting in some particular cases. This opens room for future work by trying to enhance the original algorithm of VSUA to make an optimal decision about which node replicas should vote and with which cc-vectors to maximize the reliability achieved and weaken the current MFA considered, which might be too strict.

- Recall that the nodes can fail arbitrarily and the FTTRS was used to restrict their behaviour by attached PG filtering. This facilitated the design of node FT mechanisms. Inheriting from this, future work can consider how to restrict node failure semantics by the nodes themselves and this would then lead to further simplification in the design of FT mechanisms.

- We can try to reduce reintegration time by doing reintegration in every single EC. As regards the TM resynchronization there would be no overhead since TM is received anyways in every EC and TMSN is copied by each replica. However, as regards the Voting Reintegration Point, this approach would require messages to be exchanged in each EC. This would have a negative effect

of bandwidth consumption, but, if these messages are small enough, the effect would be negligible, especially because of the fact that we use Ethernet communication network that allows for high bandwidths thus allowing huge quantities of data to be transmitted. Additional voting would also have to be performed in every EC, but this operation is rather simple and would not impose too much complexity on the node replicas' application. This approach would require additional modeling and experimentation and the gain in reliability would be questionable.

- It is possible to closely inspect the operation of the application executed by the node replicas and identify different failures modes of the replica. Then, if a node replica fails permanently, but can still contribute by performing only a subset of operations, we can find a way to detect this and allow the replica to function in a degraded mode without permanently disconnecting it. This would require the provision of more extensive fault diagnosis mechanisms that would be able to detect this.

- Although we provide mechanisms for nodes to tolerate faults in the sensors, in this dissertation we do not explicitly address sensor nor actuator fault tolerance. Thus, one future objective could be to take into account using the oversampling and prediction based techniques as a means to tolerate sensor and actuator faults even further. Moreover, different sensor replication and actuator replication techniques can also be regarded.

- Future efforts could be taken to perform more inclusive sensitivity analysis of the presented PRISM models. All the parameters presented in Chapter A can be varied and for every single parameter an optimum value can be determined. However, all these parameters depend greatly on the application executed.

- Note that the coverage values that we used in our dependability evaluation model are based on different assumptions. A potential line of research would be to use the presented simulation and implementation to obtain the concrete value of each one of these coverages.

Part IV

Appendices

Appendix A

PRISM source code

In this chapter we describe our three PRISM models in details. All the modeling details including the source code of each prism module are encompassed here.

A.1 Main model

The parameters listed in the Table A.1 are constants defined for the main model. The values of the constants presented are taken from the case of reference experiment for automotive industry throttle-by-wire applications. The parameters $pCCVectRx_i$ are the output of the auxiliary VCR model and the parameters $pResetSysFail_i$ are the output of the auxiliary reset model which are used by the main model in the steps modeling the VCR and reset respectively.

TABLE A.1: Main model parameters

Name	Type	Value	Description
numReplicas	int	3	Number of node replicas
FSTM	int	64	Frame size of the TM expressed in bytes
numTM	int	4	Number of the TMs sent in one EC
FSCtrlMsg	int	64	Frame size of the control messages exchanged among the switches expressed in bytes
numCtrlMsg	int	4	Number of control messages exchanged among switches
BER	double	1E-6	Bit-error ratio
ecDuration	double	1/1000	EC duration in seconds
missionDuration	double	36000	Mission duration in seconds
resetDuration	double	10	Reset duration in seconds
hypercycleDuration	int	20	The duration of the ECAC expressed in the number of elementary cycles
disThshFalsePosProb	double	0.001	Probability that a replica resets due to a discrepancy error counter reaching its threshold due to to many transient faults causing a reset

commThshFalsePosProb	double	0.001	Probability that a replica resets due to a communication error counter reaching its threshold due to to many transient faults causing a reset
switchFailSysFailCov	double	0.001	Coverage of tolerating a permanent failure of one switch
nodeFailSysFailCov	double	0.001	Coverage of tolerating a permanent failure of a node
nodeTransFaultSysFailCov	double	0.001	Coverage of tolerating a transient failure of a node
switchesSyncSysFailCov	double	0.001	Coverage of tolerating losses of synchronization messages between switches
TNFPSysFailCov	double	0.001	Coverage of tolerating a TNFP
transFailPropRatio	double	0.2	Ratio with which the transient fault propagates to the next ECAC
transFailPermFailManifRatio	double	0.1	Ratio with which the transient fault manifests as a permanent one
senTransFailProb	double	0	Probability that a sensor device transiently fails in one ECAC
PRr	double	1E-6	Permanent replica failure rate
PLr	double	1E-7	Permanent link failure rate
TRr	double	1E-4	Transient replica failure rate
PSWr	double	1E-6	Permanent switch failure rate
TSWr	double	1E-4	Transient switch failure rate
PIr	double	1e-7	Permanent interlink failure rate
PSr	double	1e-5	Power supply failure rate
pCCVectRx1	double	8.9202 980775 29113 E-29	Probability that all the cc-vectors from the sending replica are lost in a VCR when there is 1 switch and any number of receiving and transmitting links (1 useful), or 1 of each links and 2 switches
pCCVectRx2	double	2.2300 745204 80771 E-29	Probability that all the cc-vectors from the sending replica are lost in a VCR when there are 2 switches, 2 receiving links and 1 transmitting link

pCCVectRx3	double	4.9732 323640 97859 E-58	Probability that all the cc-vectors from the sending replica are lost in a VCR when there are 2 switches, 2 receiving links and 2 transmitting links
pCCVectRx4	double	1.9892 931234 958 E-57	Probability that all the cc-vectors from the sending replica are lost in a VCR when there are 2 switches, 1 receiving links and 2 transmitting links
pResetSysFail1	double	6.6739 994639 49503 E-7	Probability that the system fails during a reset S = 2, I = 1, L2 = 2, L3 = 2
pResetSysFail2	double	1.0339 772780 1204 E-6	Probability that the system fails during a reset S = 2, I = 1, L2 = 2, L3 = 1
pResetSysFail3	double	1.0339 772780 1204 E-6	Probability that the system fails during a reset S = 2, I = 1, L2 = 1, L3 = 2
pResetSysFail4	double	1.2176 109906 7524 E-6	Probability that the system fails during a reset S = 2, I = 1, L2 = 1, L3 = 2
pResetSysFail5	double	1.0340 945129 998447 E-6	Probability that the system fails during a reset S = 1, L2 = 2, L3 = 2
pResetSysFail6	double	1.0340 945129 998461 E-6	Probability that the system fails during a reset S = 1, L2 = 2, L3 = 1
pResetSysFail7	double	1.0340 945129 998461 E-6	Probability that the system fails during a reset S = 1, L2 = 1, L3 = 2
pResetSysFail8	double	1.0340 945129 998461 E-6	Probability that the system fails during a reset S = 1, L2 = 1, L3 = 1

The probabilities used by the main model are calculated as shown in Table A.2. As was explained before, there are two set of calculated probabilities. One set refers to message loss probabilities and uses BER parameter and the other set refers to

component failure probabilities and uses failure rates. As regards the message loss probabilities, the main model only uses TM loss probabilities that were explained in Section 9.2.1.

TABLE A.2: Main model probabilities calculation

Name	Calculation Expression	Description
pSingleTMLost	1-pow(1-BER, FSTM * 8)	Probability of losing one TM
pAllTMLost	pow(pSingleTMLost, numTM)	Probability of loosing all the TM replicas sent in a single TMW
pTMLost1	pAllTMLost * pAllTM-Lost * pAllTMLost * pAllTMLost	Probability of losing all TMs: 2S 2L 1I (2 links, 2 TMs = 4 copies)
pTMLost2	pAllTMLost * pAllTM-Lost	Probability of losing all TMs: 2S 1L 1I (1 link, 2 TMs = 2 copies)
pTMLost3	pAllTMLost	Probability of losing all TMs: other (1 link, 1 TM = 1 copy)
pSingleCtrlMsgLost	1-pow(1-BER,FSCtrlMsg * 8)	Probability of losing one control message exchanged between switches
pAllCtrlMsgLost	pow(pSingleCtrlMsgLost, numCtrlMsg)	Probability of losing all the control messages exchanged between switches
pRepFailHyperCycle	1- pow(2.718281828459, -hypercycleDuration * ecDuration * (PRr+PSr) / 3600)	Probability that a replica permanently failed during a hyper-cycle
pLinkFailHyperCycle	1- pow(2.718281828459, -hypercycleDuration * ecDuration * PLr / 3600)	Probability that a link permanently fails dur-ing a hypercycle
pRepFailReset	1- pow(2.718281828459, -resetDuration * (PRr+PSr) / 3600)	Probability that a replica permanently failed during a reset
pLinkFailReset	1- pow(2.718281828459, -resetDuration * PLr / 3600)	Probability that a link permanently fails dur-ing a reset
pRepTransientFailHyperCycle	1- pow(2.718281828459, -hypercycleDuration * ecDuration * TRr / 3600)	Probability of transient failure occurring dur-ing a hypercycle
pRepTransientFailReset	1- pow(2.718281828459, -resetDuration * TRr / 3600)	Probability of transient failure occurring dur-ing a reset

pSwitchHyperCycle	1- pow(2.718281828459, -hypercycleDuration * ecDuration * (2*(PSWr+PRr)+TSWr+PSr) / 3600)	Probability of switch permanently failing during a hypercycle
pInterlinkHyperCycle	1- pow(2.718281828459, -hypercycleDuration * ecDuration * PIr / 3600)	Probability of an interlink permanently failing during a hypercycle
pSwitchReset	1- pow(2.718281828459, -resetDuration * (2*(PSWr+PRr)+TSWr + PSr) / 3600)	Probability of switch permanently failing during a reset
pTNFPReset	transFailPermFailManifRatio * pRepTransientFailReset	Probability of TNFP occurring during a reset
pRepTransientReset	(1 - transFailPermFailManifRatio) * pRepTransientFailReset	Probability of non-TNFP occurring during a reset

The main model incorporates the four prism modules that are described in the following sections.

A.1.1 Node Replica module

This module will be instantiated 3 times representing the 3 node replicas that we have in our system. The node replica module is the most complex one. These modules execute all the 29 steps sequentially and dictate the execution of the main model. Steps are defined as constants as shown in Table A.3.

TABLE A.3: Node replica module sequential step constants

Name	Type	Value	Description
NetworkComponentFail	int	0	Permanent network topology component failure step : switches, interlinks, links
EvalNetCompSysFail	int	1	Evaluate the system failure after the network components permanent failures, deem replicas unable to communicate as permanently faulty
ReplicaFailiure	int	2	Permanent replica failure step
EvalRepSysFail	int	3	Evaluate the system failure after the permanent failures of the replicas

ReplicaTransientFail	int	4	Transient replica failure step : sensor, actuation and consensus actuation values wrong
EvalRepTransSysFail	int	5	Evaluate the system failure after the transient failures
TMProbCalculcation	int	6	Determine the TM probability to be used in the phases triggered by the TM depending on the surviving network components
Sense	int	7	Sense step : sensor value wrong due to lost TMs
EvalSenseSysFail	int	8	Evaluate the system failure after the sense step
DiscrepancySense	int	9	Discrepancy threshold reached after the sense step due to too many transient faults
VCR1CCVectRx1Lost	int	10	Determine the probability of cc-vector 1 (received from the 1st replica) being lost depending on the current network configuration
VCR1CCVectRx2Lost	int	11	Determine the probability of cc-vector 2 (received from the 2nd replica) being lost depending on the current network configuration
CommErrVCR1	int	12	Communication error threshold reached after the 1st VCR due to too many transient faults
UpdateRXVCR1	int	13	Update the received cc-vectors variables due to wrong values of the corresponding cc-vectors from the other replicas after the 1st VCR to facilitate the next evaluation step
EvalVCR1SysFail	int	14	Evaluate the system failure after the 1st VCR step
VoteSenControl	int	15	Vote on the sensor values and control step : actuation value wrong due to lost TMs
EvalVoteSenControlSysFail	int	16	Evaluate the system failure after the vote on the sensor values and control step

DiscrepancyVoteSensControl	int	17	Discrepancy threshold reached after vote on the sensor values and control step due to too many transient faults
VCR2CCVectRx1Lost	int	18	Determine the probability of cc-vector 1 (received from the 1st replica) being lost depending on the current network configuration
VCR2CCVectRx2Lost	int	19	Determine the probability of cc-vector 2 (received from the 2nd replica) being lost depending on the current network configuration
CommErrVCR2	int	20	Communication error threshold reached after the 2nd VCR due to too many transient faults
UpdateRXVCR2	int	21	Update the received cc-vectors variables due to wrong values of the corresponding cc-vectors from the other replicas after the 2nd VCR to facilitate the next evaluation step
EvalVCR2SysFail	int	22	Evaluate the system failure after the 2nd VCR step
VoteActActuate	int	23	Vote on the actuation values and actuation step : consensus actuation value wrong due to lost TMs
EvalVoteActActuateSysFail	int	24	Evaluate system failure after the vote on the actuation values and actuation step
DiscrepancyVoteActActuate	int	25	Discrepancy threshold reached after the vote on the actuation values and actuation step due to too many transient faults
EvalResetSysFail	int	26	Evaluate if the system failed during the reset step of one replica
EvalResetFaults	int	27	Evaluate all the faults of replicas that could have happened during the reset of one replica

TransFaultProp	int	28	The effects of transient faults might propagate to the next ECAC, evaluate global system failure

The variables presented in Table A.4 represent the local state of the node replica module.

TABLE A.4: Node replica module local variables

Name	Type	Initial Value	Description
step1	0..28	0	sequential steps of the ECAC
sensTempWrong1	0..1	0	obtained/calculated sensor value temporarily wrong
actuationTempWrong1	0..1	0	obtained/calculated actuation (consensus sensor) value temporarily wrong
consActuationTempWrong1	0..1	0	obtained/calculated consensus actuation value temporarily wrong
replicaFailed1	0..1	0	replica permanently failed
links1	0..2	2	operating (non-failed) links
oneSwitchOneLinkInterconnected1	bool	false	flag indicating whether a switch and a replica link are interconnected when there is one left of each
resetActive1	0..1	0	replica is being reset
tmProbability1	1..3	1	indication of the all TM loss probability (pTMLost1 - pTMLost3)
ccVectRx11	0..1	0	cc-vector 1 received by the replica
ccVectRx21	0..1	0	cc-vector 2 received by the replica
sysFail1	bool	false	a flag indicating the system failure

When a replica is permanently faulty, it will move through all the steps, modify $steps1$ variable only and skip the execution of step logic that updates the other local variables. This is done in order to avoid deadlocks since all the replicas are synchronized by each step and each replica has to execute all the steps in sequence. If the permanently faulty replica did not execute some step, it would block the other replicas synchronized with that step and would prevent the model execution to proceed.

The two additional instances of node replica module are modeled by using the concept of *module renaming* defined by PRSIM language as depicted in Listing A.1.

The renaming is done on a textual level and it allows us to rename the existing identifiers and introduce the new ones for the newly introduced instances of node replicas. We are able to do so since all the node replicas are identical executing identical set of transitions.

LISTING A.1: Node replica module renaming

```
module NodeReplica2 = NodeReplica1 [
step1 = step2,
sensTempWrong1 = sensTempWrong2, sensTempWrong2 = sensTempWrong1,
actuationTempWrong1 = actuationTempWrong2, actuationTempWrong2 =
    actuationTempWrong1,
consActuationTempWrong1 = consActuationTempWrong2,
    consActuationTempWrong2 = consActuationTempWrong1,
replicaFailed1 = replicaFailed2, replicaFailed2 = replicaFailed1,
resetActive1 = resetActive2, resetActive2 = resetActive1,
links1 = links2, links2 = links1,
oneSwitchOneLinkInterconnected1 = oneSwitchOneLinkInterconnected2,
tmProbability1 = tmProbability2,
ccVectRx11=ccVectRx12, ccVectRx21=ccVectRx22, ccVectRx12=ccVectRx11
    , ccVectRx22=ccVectRx21, ccVectRx13=ccVectRx23, ccVectRx23=
    ccVectRx13,
sysFail1 = sysFail2, sysFail2 = sysFail1
]
endmodule

module NodeReplica3 = NodeReplica1 [
step1 = step3,
sensTempWrong1 = sensTempWrong3, sensTempWrong2 = sensTempWrong1,
    sensTempWrong3 = sensTempWrong2,
actuationTempWrong1 = actuationTempWrong3, actuationTempWrong2 =
    actuationTempWrong1, actuationTempWrong3 = actuationTempWrong2,
consActuationTempWrong1 = consActuationTempWrong3,
    consActuationTempWrong2 = consActuationTempWrong1,
    consActuationTempWrong3 = consActuationTempWrong2,
replicaFailed1 = replicaFailed3, replicaFailed2 = replicaFailed1,
    replicaFailed3 = replicaFailed2,
resetActive1 = resetActive3, resetActive2 = resetActive1,
    resetActive3 = resetActive2,
links1 = links3, links2 = links1, links3 = links2,
oneSwitchOneLinkInterconnected1 = oneSwitchOneLinkInterconnected3,
tmProbability1 = tmProbability3,
ccVectRx11=ccVectRx13, ccVectRx21=ccVectRx23, ccVectRx12=ccVectRx21
    , ccVectRx22=ccVectRx11, ccVectRx13=ccVectRx22, ccVectRx23=
    ccVectRx12,
sysFail1 = sysFail3, sysFail2 = sysFail1, sysFail3 = sysFail2
] endmodule
```

Prism language defines a concept of *formula*. Formulas help to avoid code duplication and can be used anywhere an expression is expected. The next formulas count the number of occurred errors and are used by this module in the system failure evaluation steps.

Formula A.2 counts the number of permanently failed replicas.

LISTING A.2: Formula counting the number of peramenntly failure replicas

```
formula errorCounterRep = replicaFailed1 + replicaFailed2 +
    replicaFailed3;
```

Formula A.3 counts the number of replicas' sensor value errors due to replica permanent or transient failures.

LISTING A.3: Formula counting the number of replicas' sensor value errors

```
formula errorCounterSen =
ceil((replicaFailed1 + sensTempWrong1)/2) +
ceil((replicaFailed2 + sensTempWrong2)/2) +
ceil((replicaFailed3 + sensTempWrong3)/2);
```

Formula A.4 counts the number of replicas' consensus sensor (actuation) value errors due to replica permanent or transient failures.

LISTING A.4: Formula conting the number of replicas' consensus sensor (actuation) value errors

```
formula errorCounterCtrl =
ceil((replicaFailed1 + actuationTempWrong1)/2) +
ceil((replicaFailed2 + actuationTempWrong2)/2) +
ceil((replicaFailed3 + actuationTempWrong3)/2);
```

Formula A.5 counts the number of replicas' consensus actuation value errors due to replica permanent or transient failures.

LISTING A.5: Formula counting the number of replicas' consensus actuation value errors

```
formula errorCounterAct =
ceil((replicaFailed1 + consActuationTempWrong1)/2) +
ceil((replicaFailed2 + consActuationTempWrong2)/2) +
ceil((replicaFailed3 + consActuationTempWrong3)/2);
```

Formula A.6 counts the number of replicas' lost cc-vectors. Only if both of the cc-vectors are lost, this counts as an error.

LISTING A.6: Formula counting the number of replicas' lost cc-vectors

```
formula errorCounterComm =
floor((1-ccVectRx11 + 1-ccVectRx21)/2) +
floor((1-ccVectRx12 + 1-ccVectRx22)/2) +
floor((1-ccVectRx13 + 1-ccVectRx23)/2);
```

Now, we shall give a detailed description and PRISM command source code of all the 29 sequentially executed steps.

The first step *NetworkComponentFail* is depicted in Listing A.7. This step models the permanent failures of network components in a hypercycle. Particularly, in this node replica module, any of the replica's links may permanently fail with a probability $pLinkFailHyperCycle$.

LISTING A.7: Network components failures

```
//If there are two links and the replica has not failed -> any of
    the links can fail
[netCompFail] step1 = NetworkComponentFail & links1 = 2 &
    replicaFailed1 = 0 ->
        2*pLinkFailHyperCycle*(1-pLinkFailHyperCycle) : (links1'=1)
            & (step1'=EvalNetCompSysFail)
        + pLinkFailHyperCycle*pLinkFailHyperCycle : (links1'=0) & (
            step1'=EvalNetCompSysFail)
        + (1-pLinkFailHyperCycle)*(1-pLinkFailHyperCycle) :(links1
            '=2) & (step1'=EvalNetCompSysFail);
```

```
//If there is one link and the replica has not failed -> the link
   can fail
[netCompFail] step1 = NetworkComponentFail & links1 = 1 &
   replicaFailed1 = 0 ->
        pLinkFailHyperCycle : (links1'=0) & (step1'=
           EvalNetCompSysFail)
        + 1-pLinkFailHyperCycle : (links1'=1) & (step1'=
           EvalNetCompSysFail);
//If there no links or if the replica has failed -> continue
[netCompFail] step1 = NetworkComponentFail & (links1 = 0 |
   replicaFailed1 = 1) ->
        (step1'=EvalNetCompSysFail);
```

The step *EvalNetCompSysFail* depicted in Listing A.8 evaluates if the replica is deemed as permanently faulty or if the system has failed depending on the surviving network components.

The first command can be interpreted as follows. On the one hand, if the replica is no longer connected with the rest of the replicas, $links1 = 0$, it is deemed as permanently faulty by setting all the local variables to the values as seen below in Listing A.8. On the other hand, if the communication subsystem has failed, either by both switches failing or by the connection between the switches failing while they are both operational thus not allowing them to synchronize, $switches = 0|switches = 2\&interlinks = 0$, each replica module will be deemed as permanently faulty.

The second command detects the case when there is one switch and link left due to permanent faults of the former, $switches = 1\&links1 = 1$. If this is the first evaluation of this case, the variable $oneSwitchOneLinkInterconnected1$ is $false$ (initial value). Then, the replica and the switch might be interconnected or not with a 50% chance. If they are not, the replica is deemed as permanently faulty.

The third command just moves to the next step since none of the above guards were fulfilled.

LISTING A.8: Evaluation of replica and system permanent failures depending on which components of network topology failed

```
//If there are no more links or switches or if there are 2
   disconnected switches -> replica is deemed as permanently faulty
[evalSysFail] step1 = EvalNetCompSysFail & (links1 = 0 | switches =
    0 | switches = 2 & interlinks = 0) ->
        (step1'=ReplicaFailiure)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
           consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
           (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0);
//If there is one link and one switch and they are not
   interconnected -> they will be interconnected or not(replica is
   deemed as permantly faulty)
[evalSysFail] step1 = EvalNetCompSysFail & switches = 1 & links1 =
   1 & oneSwitchOneLinkInterconnected1 = false ->
        0.5 : (step1'=ReplicaFailiure) & (
           oneSwitchOneLinkInterconnected1'=true)
        + 0.5 : (step1'=ReplicaFailiure)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
           consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
           (resetActive1'=0)
```

```
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0);
        //If not of the above is the case -> continue
[evalSysFail] step1 = EvalNetCompSysFail & !((links1 = 0 | switches
    = 0 | switches = 2 & interlinks = 0)) & !(switches = 1 & links1
    = 1 & oneSwitchOneLinkInterconnected1 = false) ->
        (step1'=ReplicaFailiure);
```

The step *ReplicaFailiure* (Listing A.9) models the permanent failure of a replica in a hypercycle. The replica may permanently fail with a probability $pRepFailHyperCycle$. This failure may further cause the system failure with a coverage $nodeFailSysFailCov$ by setting the variable $sysFail1$ to $true$. This variable will be evaluated by the next step.

LISTING A.9: Replica permanent failure

```
//If the replica has not failed -> it can permanently fail which
   may cause the system failure with nodeFailSysFailCov
[repFailure] step1 = ReplicaFailiure & replicaFailed1 = 0 ->
        pRepFailHyperCycle * (1-nodeFailSysFailCov) : (step1'=
            EvalRepSysFail)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
            (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
        + pRepFailHyperCycle * nodeFailSysFailCov: (step1'=
            EvalRepSysFail) & (sysFail1'=true)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
            (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
        + 1 - pRepFailHyperCycle : (step1'=EvalRepSysFail);
//If the replica has failed -> continue
[repFailure] step1 = ReplicaFailiure & replicaFailed1 = 1 ->
        (step1'=EvalRepSysFail);
```

The step *EvalRepSysFail* (Listing A.10) evaluates if the system has failed either by more than a majority of replica failing permanently, $errorCounterRep >= majority$, or by a single replica permanent failure propagating and causing the system failure $sysFail1|sysFail2|sysFail3$.

LISTING A.10: Evaluate the system failure

```
//If there are more than a majority of permanent replica failures
   or if the system has failed previously -> replica is deemed as
   permanently faulty
[evalSysFail] step1 = EvalRepSysFail & (errorCounterRep >= majority
    | sysFail1 | sysFail2 | sysFail3) ->
        (step1'=ReplicaTransientFail)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
            (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
```

```
        & (sysFail1'=true);
//If not -> continue
[evalSysFail] step1 = EvalRepSysFail & !(errorCounterRep >=
    majority | sysFail1 | sysFail2 | sysFail3) ->
        (step1'=ReplicaTransientFail);
```

The step *ReplicaTransientFail* (Listing A.11) models the transient failure of a replica in a hypercycle.

On the one hand, a replica may transiently fail with a probability $pRepTransientFailHyperCycle$. If this is the case, we take the most pessimistic approach and assume that the outputs of all the computational control cycle phases (S, VS+C and VA+A) are wrong: sense value, actuation value and consensus actuation value. This transient failure may further cause the system failure with a coverage $nodeTransFaultSysFailCov$ by setting the variable $sysFail1$ to $true$.

On the other hand, this transient failure may be manifesting as permanent one with a ratio $transFailPermFailManifRatio$. If this is the case, we do the same updates as for permanent failure of a replica additionally setting the variable $resetActive1$ to 1 to differentiate it from permanent one. This failure may also further cause the system failure with a coverage $TNFPSysFailCov$ by setting the variable $sysFail1$ to $true$.

Lastly, we added a support for modeling the failure of sensor devices only, i.e. with probability $pRepTransientFailHyperCycle$ a sensor device may transiently fail and set $sensTempWrong1$ to 1.s

The system failure will be evaluated by the next evaluation step as was the case with the replica permanent failures.

LISTING A.11: Replica transient failure

```
//If the replica has not failed -> it can transietly fail (put all
    temp variables to 0) which may cause the system failure with a
    nodeTransFaultSysFailCov or it can manifest as a TNFP which may
    further cause the system failure with a TNFPSysFailCov
[repTransFail] step1 = ReplicaTransientFail & replicaFailed1 = 0 ->
        (1-transFailPermFailManifRatio)*pRepTransientFailHyperCycle
            *(1-nodeTransFaultSysFailCov) : step1'=
            EvalRepTransSysFail)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1)
        + (1-transFailPermFailManifRatio)*
            pRepTransientFailHyperCycle*nodeTransFaultSysFailCov : (
            step1'=EvalRepTransSysFail) & (sysFail1'=true)

        + transFailPermFailManifRatio*pRepTransientFailHyperCycle
            *(1-TNFPSysFailCov) : (step1'=EvalRepTransSysFail)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
        & (resetActive1'=1)
        + transFailPermFailManifRatio*pRepTransientFailHyperCycle*
            TNFPSysFailCov : (step1'=EvalRepTransSysFail) & (
            sysFail1'=true)

        + (1 - pRepTransientFailHyperCycle)*(1-senTransFailProb) :
            (step1'=EvalRepTransSysFail)
        + (1 - pRepTransientFailHyperCycle)*senTransFailProb : (
            step1'=EvalRepTransSysFail) & (sensTempWrong1'=1) ;
```

```
//If the replica has failed -> continue
[repTransFail] step1 = ReplicaTransientFail & replicaFailed1 = 1 ->
        (step1'=EvalRepTransSysFail);
```

The step *EvalRepTransSysFail* (Listing A.12) evaluates if the system has failed either by more than a majority of replica failing transiently, $errorCounterSen >= majority$ or by a single replica transient failure propagating and causing the system failure $sysFail1|sysFail2|sysFail3$. Note that by evaluation of the formula $errorCounterSen$ we only check one variable $sensTempWrong(i)$. This is enough since we know that all others ($actuationTempWrong(i)$, $consActuationTempWrong(i)$) will be set to 1 as well in a case of a transient failure.

LISTING A.12: Evaluate the system failure

```
//If there are more than a majority of permanent replica failures
   or if the system has failed previously -> replica is deemed as
   permanently faulty
[evalSysFail] step1 = EvalRepTransSysFail & (errorCounterSen >=
   majority | sysFail1 | sysFail2 | sysFail3) ->
        (step1'=TMProbCalculcation)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
            (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0);
//If not -> continue
[evalSysFail] step1 = EvalRepTransSysFail & !(errorCounterSen >=
   majority | sysFail1 | sysFail2 | sysFail3) ->
        (step1'=TMProbCalculcation);
```

The step *TMProbCalculcation* (Listing A.13) determines which of the precalculated TM probabilities $pTMLost1...pTMLost3$ will be used by the subsequent steps depending on the surviving network components.

LISTING A.13: Determining the probability of losing all TMs depending on network topologty failures

```
[tmProb] step1 = TMProbCalculcation & replicaFailed1 = 0 & switches
    = 2 & links1 = 2 & interlinks = 1 ->
        (tmProbability1'=1) & (step1'=Sense);
[tmProb] step1 = TMProbCalculcation & replicaFailed1 = 0 & switches
    = 2 & links1 = 1 & interlinks = 1 ->
        (tmProbability1'=2) & (step1'=Sense);
[tmProb] step1 = TMProbCalculcation & replicaFailed1 = 0 & !(
   switches = 2 & links1 = 2 & interlinks = 1) & !(switches = 2 &
   links1 = 1 & interlinks = 1) ->
        (tmProbability1'=3) & (step1'=Sense);
[tmProb] step1 = TMProbCalculcation & replicaFailed1 = 1 ->
        (step1'=Sense);
```

The step *Sense* (Listing A.14) models the sense phase of the extended control application cycle. Since we have already modeled the transient faults in the previous *ReplicaTransientFail* step, the only thing that can affect the output of this step, sensor values produced by the replicas, is the reception of the TMs. If no TM is received, this phase will not be activated and no output will be produced as a result. Therefore, depending on network topology configuration and predetermined TM loss probability $tmProbability1$, all TM replicas can be lost with a corresponding probability

$pTMLost1...pTMLost3$ and as a result the variable $sensTempWrong1$ can be set to 1.

LISTING A.14: Extended control application cycle Sense phase

```
// If the replica has not failed permanently or transiently ->
   sensor value can be wrong due to lost TMs
[sense] step1 = Sense & replicaFailed1 = 0 & sensTempWrong1 = 0 &
   tmProbability1 = 1 ->
        pTMLost1 : (sensTempWrong1'=1) & (step1'=EvalSenseSysFail)
        + 1 - pTMLost1 : (step1'=EvalSenseSysFail);
[sense] step1 = Sense & replicaFailed1 = 0 & sensTempWrong1 = 0 &
   tmProbability1 = 2 ->
        pTMLost2 : (sensTempWrong1'=1) & (step1'=EvalSenseSysFail)
        + 1 - pTMLost2 : (step1'=EvalSenseSysFail);
[sense] step1 = Sense & replicaFailed1 = 0 & sensTempWrong1 = 0 &
   tmProbability1 = 3 ->
        pTMLost3 : (sensTempWrong1'=1) & (step1'=EvalSenseSysFail)
        + 1 - pTMLost3 : (step1'=EvalSenseSysFail);
// If the replica has failed permanently or transiently -> continue
[sense] step1 = Sense & (replicaFailed1 = 1 | sensTempWrong1 = 1)
   ->
        (step1'=EvalSenseSysFail);
```

The step *EvalSenseSysFail* (Listing A.15) evaluates if the system will fail after the *Sense* step by checking if more than a majority of of sensor values are wrong, $errorCounterSen >= majority$.

LISTING A.15: Evaluate the system failure

```
//If there are more than a majority of permanent replica failures
   or sensor value errors or if the system has failed previously ->
    replica is deemed as permanently faulty
[evalSysFail] step1 = EvalSenseSysFail & errorCounterSen >=
   majority ->
        (step1'=DiscrepancySense)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
            (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0);
//If not -> continue
[evalSysFail] step1 = EvalSenseSysFail & errorCounterSen < majority
    ->
        (step1'=DiscrepancySense);
```

The step *DiscrepancySense* (Listing A.16) models the occurrence of permanent fault detection due to too many transient faults. If the sensor value was wrong due to transient faults, there is a probability $disThshFalsePosProb$ that due to too many previously occurred consecutive transient faults discrepancy error counter (DEC) reaches its threshold and needlessly resets a replica.

LISTING A.16: Discrepancy threshold reached after the sense phase

```
//If the replica has not failed and the the sensor value was wrong
   -> replica may reset due to discrepancy error counter reaching
   its threshold with a disThshFalsePosProb
[disSense] step1 = DiscrepancySense & replicaFailed1 = 0 &
   sensTempWrong1 = 1 ->
```

```
        disThshFalsePosProb : (step1'=VCR1CCVectRx1Lost)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
        & (resetActive1'=1)
        + 1 - disThshFalsePosProb : (step1'=VCR1CCVectRx1Lost);
//If the replica has not failed and the the sensor value was not
   wrong -> continue
[disSense] step1 = DiscrepancySense & replicaFailed1 = 0 &
   sensTempWrong1 = 0 ->
        (step1'=VCR1CCVectRx1Lost);
//If the replica has failed -> continue
[disSense] step1 = DiscrepancySense & replicaFailed1 = 1 ->
        (step1'=VCR1CCVectRx1Lost);
```

The steps *VCR1CCVectRx1Lost* and *VCR1CCVectRx2Lost* (Listing A.17) model the first VCR step of the extended control application cycle. Specifically, they model the loss of cc-vectors received from the other two replicas. Depending on the surviving network component configurations there are 20 possible scenarios with specific combinations of probabilities $pCCVectRx1$ to $pCCVectRx4$ that determine what are the probabilities of losing these cc-vectors. How these 20 scenarios are detected will be explained in Section A.2 when talking about the auxiliary VCR model.

LISTING A.17: VCR1

```
//Scenario 1
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & (replicaFailed1 = 1 |
   errorCounterRep > 1) ->
(step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & (replicaFailed1 = 1 |
   errorCounterRep > 1) ->
(step1'=CommErrVCR1);

//Scenario 2
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 1 & links2 = 0 & links3 > 0 ->
(step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 1 & links2 = 0 & links3 > 0 ->
pCCVectRx1 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx1 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 3
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 1 & links2 > 0 & links3 = 0 ->
pCCVectRx1 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx1 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 1 & links2 > 0 & links3 = 0 ->
(step1'=CommErrVCR1);

//Scenario 4
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 1 & links2 > 0 & links3 > 0 ->
pCCVectRx1 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx1 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
```

```
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 1 & links2 > 0 & links3 > 0 ->
pCCVectRx1 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx1 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 5
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 0 & links3 = 1 ->
(step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 0 & links3 = 1 ->
pCCVectRx1 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx1 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 6
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 0 & links3 = 2 ->
(step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 0 & links3 = 2 ->
pCCVectRx4 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx4 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 7
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 0 & links3 = 1 ->
(step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 0 & links3 = 1 ->
pCCVectRx2 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx2 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 8
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 0 & links3 = 2 ->
(step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 0 & links3 = 2 ->
pCCVectRx3 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx3 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 9
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 1 & links3 = 0 ->
pCCVectRx1 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx1 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
```

```
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 1 & links3 = 0 ->
(step1'=CommErrVCR1);

//Scenario 10
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 2 & links3 = 0 ->
pCCVectRx4 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx4 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 2 & links3 = 0 ->
(step1'=CommErrVCR1);

//Scenario 11
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 1 & links3 = 0 ->
pCCVectRx2 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx2 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 1 & links3 = 0 ->
(step1'=CommErrVCR1);

//Scenario 12
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 2 & links3 = 0 ->
pCCVectRx3 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx3 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 2 & links3 = 0 ->
(step1'=CommErrVCR1);

//Scenario 13
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 1 & links3 = 1 ->
pCCVectRx1 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx1 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 1 & links3 = 1 ->
pCCVectRx1 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx1 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 14
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   1 & links2 = 1 & links3 = 2 ->
pCCVectRx1 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx1 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
```

```
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    1 & links2 = 1 & links3 = 2 ->
pCCVectRx4 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx4 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 15
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    1 & links2 = 2 & links3 = 1 ->
pCCVectRx4 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx4 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    1 & links2 = 2 & links3 = 1 ->
pCCVectRx1 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx1 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 16
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    1 & links2 = 2 & links3 = 2 ->
pCCVectRx4 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx4 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    1 & links2 = 2 & links3 = 2 ->
pCCVectRx4 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx4 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 17
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    2 & links2 = 1 & links3 = 1 ->
pCCVectRx2 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx2 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    2 & links2 = 1 & links3 = 1 ->
pCCVectRx2 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx2 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 18
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    2 & links2 = 1 & links3 = 2 ->
pCCVectRx2 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx2 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
    errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
    2 & links2 = 1 & links3 = 2 ->
pCCVectRx3 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx3 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 19
```

```
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 2 & links3 = 1 ->
pCCVectRx3 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx3 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 2 & links3 = 1 ->
pCCVectRx2 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx2 : (ccVectRx21'=1) & (step1'=CommErrVCR1);

//Scenario 20
[vcrProbRX1] step1 = VCR1CCVectRx1Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 2 & links3 = 2 ->
pCCVectRx3 : (step1'=VCR1CCVectRx2Lost)
+ 1 - pCCVectRx3 : (ccVectRx11'=1) & (step1'=VCR1CCVectRx2Lost);
[vcrProbRX2] step1 = VCR1CCVectRx2Lost & !(replicaFailed1 = 1 |
   errorCounterRep > 1) & switches = 2 & interlinks = 1 & links1 =
   2 & links2 = 2 & links3 = 2 ->
pCCVectRx3 : (step1'=CommErrVCR1)
+ 1 - pCCVectRx3 : (ccVectRx21'=1) & (step1'=CommErrVCR1);
```

The step *CommErrVCR1* (Listing A.18) models the occurrence of permanent fault detection ad reset after the first VCR due to too many transient faults. If one of the cc-vector values was wrong due to transient faults, there is a probability $commThshFalsePosProb$ that due to too many previously occurred consecutive transient faults communication error counter (CEC) reaches its threshold and needlessly resets a replica.

LISTING A.18: Communication error threshold reached after the 1st VCR phase

```
//If the replica has not failed and there were at least one
   communication error(lost cc-vector) -> replica may reset due to
   communication error counter reaching its threshold with a
   commThshFalsePosProb
[disVCR] step1 = CommErrVCR1 & replicaFailed1 = 0 & ((
   replicaFailed2 = 0 & ccVectRx11 = 0) | (replicaFailed3 = 0 &
   ccVectRx21 = 0)) ->
        commThshFalsePosProb : (step1'=UpdateRXVCR1)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
        & (resetActive1'=1)
        + 1 - commThshFalsePosProb : (step1'=UpdateRXVCR1);
//If the replica has not failed and there were no communication
   errors -> continue
[disVCR] step1 = CommErrVCR1 & replicaFailed1 = 0 & !((
   replicaFailed2 = 0 & ccVectRx11 = 0) | (replicaFailed3 = 0 &
   ccVectRx21 = 0)) ->
        (step1'=UpdateRXVCR1);
//If the replica has failed -> continue
[disVCR] step1 = CommErrVCR1 & replicaFailed1 = 1 ->
        (step1'=UpdateRXVCR1);
```

The step *UpdateRXVCR1* (Listing A.19) facilitates the execution of the following steps by transforming the wrong cc-vectors of the other replicas into a failure to receive these cc-vectors. Since a wrong cc-vector propagates to all the replicas, it can

be assumed that the cc-vector has not been received in the first place. If the cc-vector is wrong, it is useless. As a result, this will reduce the predicate condition complexity of the following steps.

LISTING A.19: Update the received cc-vectors due to wrong value of the other node replicas after the 1st VCR step

```
//If any of the sensor values is wrong -> update the correspodning
   received cc-vector variables
[updateRxVCRSen] step1 = UpdateRXVCR1 & sensTempWrong1 = 0 &
   sensTempWrong2 = 1 & sensTempWrong3 = 0 ->
        (step1'=EvalVCR1SysFail) & (ccVectRx11'=0);
[updateRxVCRSen] step1 = UpdateRXVCR1 & sensTempWrong1 = 0 &
   sensTempWrong2 = 0 & sensTempWrong3 = 1 ->
        (step1'=EvalVCR1SysFail) & (ccVectRx21'=0);
[updateRxVCRSen] step1 = UpdateRXVCR1 & sensTempWrong1 = 0 &
   sensTempWrong2 = 1 & sensTempWrong3 = 1 ->
        (step1'=EvalVCR1SysFail) & (ccVectRx11'=0) & (ccVectRx21
           '=0);
//If not -> continue
[updateRxVCRSen] step1 = UpdateRXVCR1 & (sensTempWrong1 = 1 |
   sensTempWrong2 = 0 & sensTempWrong3 = 0) ->
        (step1'=EvalVCR1SysFail);
```

The step *EvalVCR1SysFail* (Listing A.20) evaluates if the system will fail after the first VCR step.

The values being exchanged are the sensor values. Each of the obtained values might differ. The voting that is performed on these values afterwards includes inexact matching. This means that a voting comparison must include these differences. Usually a function of bounded average is being used. In order for all the non-faulty replicas to obtain the same voting result, they have to use the exact same set of values.

To explain this step we use the MS vector defined in Section 5.3.4 with substitute values $true/false$ with $1/0$. As can be seen by Figure A.1 there are 3 cases that correspond to the first 3 group of commands from Listing A.20. They are represented by different values in the matrix that represents the MS vector.

The last matrix (bottom-rigth) shows the connection with prism variables. The numbers represent the i and j from $ccVectRxij$ variables. Note that diagonal values are always fixed. This is the case because we assume that a permanently failed replica does not have a value that can be used and the non-failed replicas always have their own value whether it is wrong or not. This matrix is interpreted a bit differently for this step, since now the diagonal values represent if the node replicas have their own cc-vectors or not, not the switch replicas.

Following are the explanation of the three cases and the corresponding group of commands:

First, if a replica has failed, we can see that the corresponding row and the column have fixed values, all 0. This means that other replicas cannot use the values produced by this one and the replica itself is useless (it has no correct values). In this scenario, if at least one cc-vector is lost, that replica will be unable to vote and as a result only one voting capable replica will remain. This leads to the system failure.

Second, if one replica has produced a wrong sensor value, this value will propagate and all the replicas will use the same wrong value. This is equivalent as not having a cc-vector received. Remember that this optimization was done by the previous step *UpdateRXVCR1*. Now, if any two values from different replicas (rows) are

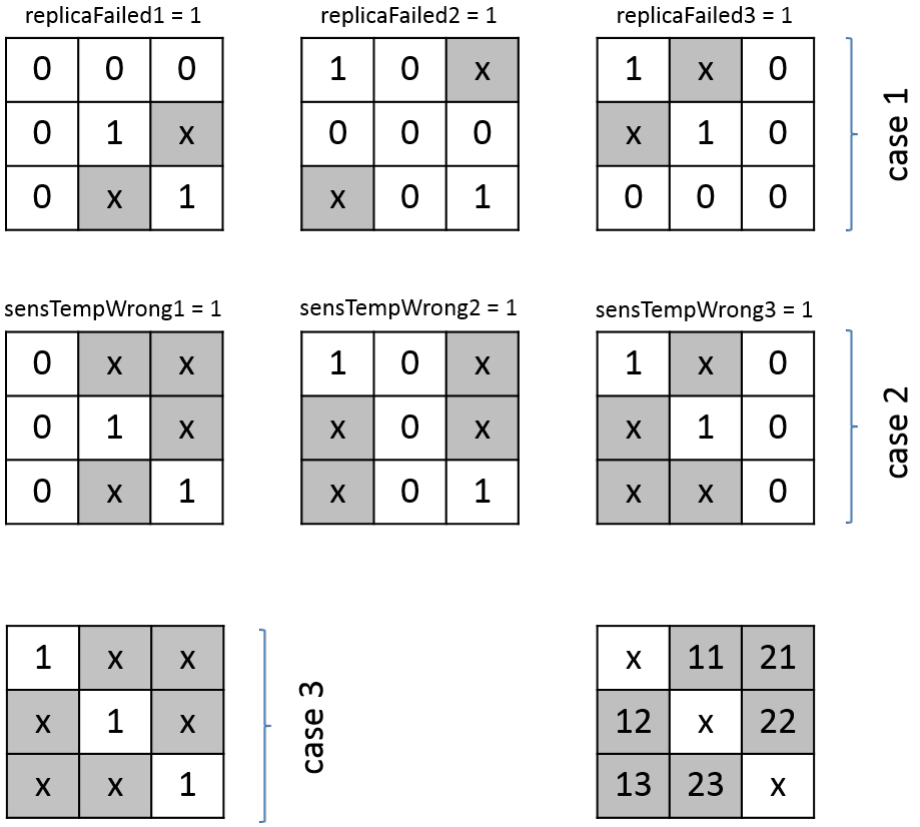


FIGURE A.1: Evaluate messages lost in the VCR

lost, there is no majority of voting capable replicas and as a result the system will fail.

Last, we evaluate the case where no replica has failed and no wrong sensor values were produced. If any two values from different replicas (rows) are lost that do not correspond to the same cc-vector (different columns), the system will fail. Note that now all the replicas are voting capable. However, since the matching used by the voting is inexact, all the replicas have to vote with the exact same set of values in order to yield the same voting results. If we have two cc-vector losses by two different replicas not corresponding to the same cc-vector, this is not the case, i.e. if $ccVectRx11 = 0$ and $ccVectRx22 = 0$, this means that each replica will have a different set of values: replica 1:1,3; replica 2:1,2; replica 3:1,2,3;. Each replica might yield a different voting result and this eventually leads to the system failure.

LISTING A.20: Evaluate if the system will fail after the 1st VCR phase

```
//If the replica has failed and any of the surviving replicas lost
    a cc-vector -> replica is deemed as permanently faulty (2
    replicas unable to vote)
[evalVCRSen] step1 = EvalVCR1SysFail & replicaFailed1 = 1 &
    replicaFailed2 = 0 & replicaFailed3 = 0 & (ccVectRx22 = 0 |
    ccVectRx23 = 0) ->
(step1'=VoteSenControl)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
```

```
& (ccVectRx11'=0) & (ccVectRx21'=0);
[evalVCRSen] step1 = EvalVCR1SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 1 & replicaFailed3 = 0 & (ccVectRx21 = 0 |
    ccVectRx13 = 0) ->
(step1'=VoteSenControl)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
[evalVCRSen] step1 = EvalVCR1SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 1 & (ccVectRx11 = 0 |
    ccVectRx12 = 0) ->
(step1'=VoteSenControl)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);

//If no replica has failed and a sensor value is wrong (entire
    column) and any of the other cc-vecors are lost(not
    correspodning to wrong sensor value) -> replica is deemed as
    permanently faulty (2 replicas unable to vote)
[evalVCRSen] step1 = EvalVCR1SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 1 & sensTempWrong2 = 0 & sensTempWrong3 = 0
& ( (ccVectRx11 = 0 & ccVectRx22 = 0) | (ccVectRx11 = 0 &
    ccVectRx23 = 0) | (ccVectRx21 = 0 & ccVectRx22 = 0) | (
    ccVectRx21 = 0 & ccVectRx23 = 0)
  | (ccVectRx22 = 0 & ccVectRx23 = 0) ) ->
(step1'=VoteSenControl)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
[evalVCRSen] step1 = EvalVCR1SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 0 & sensTempWrong2 = 1 & sensTempWrong3 = 0
& ( (ccVectRx21 = 0 & ccVectRx12 = 0) | (ccVectRx21 = 0 &
    ccVectRx22 = 0) | (ccVectRx21 = 0 & ccVectRx13 = 0)
  | (ccVectRx11 = 0 & ccVectRx13 = 0) | (ccVectRx22 = 0 &
      ccVectRx13 = 0) ) ->
(step1'=VoteSenControl)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
[evalVCRSen] step1 = EvalVCR1SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 1
```

```
& ( (ccVectRx11 = 0 & ccVectRx12 = 0) | (ccVectRx11 = 0 &
    ccVectRx13 = 0) | (ccVectRx11 = 0 & ccVectRx23 = 0)
  | (ccVectRx12 = 0 & ccVectRx13 = 0) | (ccVectRx12 = 0 &
      ccVectRx23 = 0) ) ->
(step1'=VoteSenControl)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);

//If no replica has failed and no sensor value is wrong and two
    replica lost two cc-vecors or two asymmetric cc-vectors are lost
    (sensor voting consitency not guaranteed) -> replica is deemed
    as permanently faulty (2 replicas unable to vote)
[evalVCRSen] step1 = EvalVCR1SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 0
& ( (errorCounterComm >= majority)
  | (ccVectRx11 = 0 & ccVectRx12 = 0) | (ccVectRx11 = 0 &
      ccVectRx22 = 0) | (ccVectRx11 = 0 & ccVectRx13 = 0)
  | (ccVectRx21 = 0 & ccVectRx12 = 0) | (ccVectRx21 = 0 &
      ccVectRx13 = 0) | (ccVectRx21 = 0 & ccVectRx23 = 0)
  | (ccVectRx12 = 0 & ccVectRx23 = 0) | (ccVectRx22 = 0 &
      ccVectRx13 = 0) | (ccVectRx22 = 0 & ccVectRx23 = 0) ) ->
(step1'=VoteSenControl)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);

//If none of the above ocurred -> continue
[evalVCRSen] step1 = EvalVCR1SysFail
& !(replicaFailed1 = 1 & replicaFailed2 = 0 & replicaFailed3 = 0 &
    (ccVectRx22 = 0 | ccVectRx23 = 0))
& !(replicaFailed1 = 0 & replicaFailed2 = 1 & replicaFailed3 = 0 &
    (ccVectRx21 = 0 | ccVectRx13 = 0))
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 1 &
    (ccVectRx11 = 0 | ccVectRx12 = 0))
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0
    & sensTempWrong1 = 1 & sensTempWrong2 = 0 & sensTempWrong3 = 0
    & ( (ccVectRx11 = 0 & ccVectRx22 = 0) | (ccVectRx11 = 0 &
        ccVectRx23 = 0) | (ccVectRx21 = 0 & ccVectRx22 = 0) | (
        ccVectRx21 = 0 & ccVectRx23 = 0) | (ccVectRx22 = 0 &
        ccVectRx23 = 0) ) )
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0
    & sensTempWrong1 = 0 & sensTempWrong2 = 1 & sensTempWrong3 = 0
       & ( (ccVectRx21 = 0 & ccVectRx12 = 0) | (ccVectRx21 = 0 &
           ccVectRx22 = 0) | (ccVectRx21 = 0 & ccVectRx13 = 0) | (
           ccVectRx11 = 0 & ccVectRx13 = 0) | (ccVectRx22 = 0 &
           ccVectRx13 = 0) ) )
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0
    & sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 1
```

```
    & ( (ccVectRx11 = 0 & ccVectRx12 = 0) | (ccVectRx11 = 0 &
        ccVectRx13 = 0) | (ccVectRx11 = 0 & ccVectRx23 = 0) | (
        ccVectRx12 = 0 & ccVectRx13 = 0) | (ccVectRx12 = 0 &
        ccVectRx23 = 0) ) )
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0 &
    sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 0
  & ( (errorCounterComm >= majority)
    | (ccVectRx11 = 0 & ccVectRx12 = 0) | (ccVectRx11 = 0 &
        ccVectRx22 = 0) | (ccVectRx11 = 0 & ccVectRx13 = 0)
    | (ccVectRx21 = 0 & ccVectRx12 = 0) | (ccVectRx21 = 0 &
        ccVectRx13 = 0) | (ccVectRx21 = 0 & ccVectRx23 = 0)
    | (ccVectRx12 = 0 & ccVectRx23 = 0) | (ccVectRx22 = 0 &
        ccVectRx13 = 0) | (ccVectRx22 = 0 & ccVectRx23 = 0) ) ) ->
(step1'=VoteSenControl);
```

The step *VoteSenControl* (Listing A.21) models the merged vote on sensor values and control phases of the extended control application cycle.

The first command evaluates if there are at least a majority of sensor values. If there are not, the output of this phase will be wrong, $actuationTempWrong1 = 1$.

The next three commands model the loss of TMs, same as in step *Sense*. Depending on network topology configuration and predetermined TM loss probability $tmProbability1$, all TM replicas can be lost with a corresponding probability $pTMLost1...pTMLost3$ and as a result the variable $actuationTempWrong1$ can be set to 1.

LISTING A.21: Vote on sensor values and control phase of the extended control application cycle

```
//If the replica has not failed permanently or transiently and
    there is not enough messages for voting -> actuation value will
    be wrong
[voteSenControl] step1 = VoteSenControl & replicaFailed1 = 0 &
    actuationTempWrong1 = 0 & (ccVectRx11 + ccVectRx21 = 0 |
    ccVectRx11 + ccVectRx21 = 1 & sensTempWrong1 = 1) ->
(actuationTempWrong1'=1) & (step1'=EvalVoteSenControlSysFail);
//If the replica has not failed permanently or transiently and
    there is enough messages for voting -> actuation value may be
    wrong due to lost TMs
[voteSenControl] step1 = VoteSenControl & replicaFailed1 = 0 &
    actuationTempWrong1 = 0 & tmProbability1 = 1 & !(ccVectRx11 +
    ccVectRx21 = 0 | ccVectRx11 + ccVectRx21 = 1 & sensTempWrong1 =
    1) ->
pTMLost1 : (actuationTempWrong1'=1) & (step1'=
    EvalVoteSenControlSysFail)
+ 1 - pTMLost1 : (step1'=EvalVoteSenControlSysFail);
[voteSenControl] step1 = VoteSenControl & replicaFailed1 = 0 &
    actuationTempWrong1 = 0 & tmProbability1 = 2 & !(ccVectRx11 +
    ccVectRx21 = 0 | ccVectRx11 + ccVectRx21 = 1 & sensTempWrong1 =
    1) ->
pTMLost2 : (actuationTempWrong1'=1) & (step1'=
    EvalVoteSenControlSysFail)
+ 1 - pTMLost2 : (step1'=EvalVoteSenControlSysFail);
[voteSenControl] step1 = VoteSenControl & replicaFailed1 = 0 &
    actuationTempWrong1 = 0 & tmProbability1 = 3 & !(ccVectRx11 +
    ccVectRx21 = 0 | ccVectRx11 + ccVectRx21 = 1 & sensTempWrong1 =
    1) ->
pTMLost3 : (actuationTempWrong1'=1) & (step1'=
    EvalVoteSenControlSysFail)
```

```
+ 1 - pTMLost3 : (step1'=EvalVoteSenControlSysFail);
//If the replica has failed permanently or transiently -> continue
[voteSenControl] step1 = VoteSenControl & (replicaFailed1 = 1 |
   actuationTempWrong1 = 1) ->
(step1'=EvalVoteSenControlSysFail);
```

The step *EvalVoteSenControlSysFail* (Listing A.22) evaluates if the system will fail after the *VoteSenControl* step by checking if more than a majority of of actuation (consensus sensor) values are wrong, $errorCounterCtrl >= majority$.

LISTING A.22: Evaluate the system failure

```
//If there are more than a majority of actuation or replica
   failures -> replica is deemed as permanently faulty
[evalSysFail] step1 = EvalVoteSenControlSysFail & errorCounterCtrl
   >= majority ->
        (step1'=DiscrepancyVoteSensControl)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
            (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0);
//If not -> continue
[evalSysFail] step1 = EvalVoteSenControlSysFail & errorCounterCtrl
   < majority ->
        (step1'=DiscrepancyVoteSensControl) & (ccVectRx11'=0) & (
            ccVectRx21'=0);
```

The step *DiscrepancyVoteSensControl* (Listing A.23) models the occurrence of permanent fault detection due to too many transient faults. If the actuation (consensus sensor) value was wrong due to transient faults, there is a probability $disThshFalsePosProb$ that due to too many previously occurred consecutive transient faults discrepancy error counter (DEC) reaches its threshold and needlessly resets a replica.

LISTING A.23: Discrepancy threshold reached after the vote on sensor and control phase

```
//If the replica has not failed and the the actuation value was
   wrong -> replica may reset due to discrepancy error counter
   reaching its threshold with a disThshFalsePosProb
[disCtrl] step1 = DiscrepancyVoteSensControl & replicaFailed1 = 0 &
    actuationTempWrong1 = 1 ->
        disThshFalsePosProb : (step1'=VCR2CCVectRx1Lost)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
        & (resetActive1'=1)
        + 1 - disThshFalsePosProb : (step1'=VCR2CCVectRx1Lost);
//If the replica has not failed and the the actuation value was not
    wrong -> continue
[disCtrl] step1 = DiscrepancyVoteSensControl & replicaFailed1 = 0 &
    actuationTempWrong1 = 0 ->
        (step1'=VCR2CCVectRx1Lost);
//If the replica has failed -> continue
[disCtrl] step1 = DiscrepancyVoteSensControl & replicaFailed1 = 1
   ->
        (step1'=VCR2CCVectRx1Lost);
```

The step *VCR2CCVectRx1Lost* and *VCR2CCVectRx2Lost* are completely identical to the steps *VCR1CCVectRx1Lost* and *VCR1CCVectRx2Lost* from Listing A.17 and they model the second VCR phase of the extended control application cycle.

The step *CommErrVCR2* is identical to the step *CommErrVCR1* from Listing A.18. It models the occurrence of false positive detection of permanent fault after the second VCR.

The step *UpdateRXVCR2* is identical to the step *UpdateRXVCR1* from Listing A.19 and the modeling logic is completely the same, i.e. the transformation of the wrong cc-vectors of the other replicas into the failed reception of those cc-vector is performed.

The step *EvalVCR2SysFail* (Listing A.24) evaluates if the system will fail after the second VCR phase.

The values being exchanged now are the actuation values. Since replica determinism is being enforced, each replica will produce exactly the same actuation value. The voting that is performed on these values afterwards includes exact matching. The voting comparison is much simpler since the values are expected to be identical. Now, this means that the set of values does not have to be exactly the same for all the replicas. It is sufficient that at least a majority of replicas (2) vote with at least a majority of values (2).

As can be seen by Figure A.1 there are 3 cases that correspond to the first 3 group of commands from Listing A.24.

The first two set of commands are equivalent to the step *EvalVCR1SysFail*. The last set of commands is now different since replicas do not need to use the same set of values for the successful voting. Now, the condition that leads to the system failure is that at least a majority of replicas will be unable to vote due to too many cc-vectors lost, $errorCounterComm >= majority$.

LISTING A.24: Evaluate if the system will fail after the 2nd VCR phase

```
//If the replica has failed and any of the surviving replicas lost
    a cc-vector -> replica is deemed as permanently faulty (2
    replicas unable to vote)
[evalVCRAct] step1 = EvalVCR2SysFail & replicaFailed1 = 1 &
    replicaFailed2 = 0 & replicaFailed3 = 0 & (ccVectRx22 = 0 |
    ccVectRx23 = 0) ->
(step1'=VoteActActuate)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
[evalVCRAct] step1 = EvalVCR2SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 1 & replicaFailed3 = 0 & (ccVectRx21 = 0 |
    ccVectRx13 = 0) ->
(step1'=VoteActActuate)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
```

```
[evalVCRAct] step1 = EvalVCR2SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 1 & (ccVectRx11 = 0 |
    ccVectRx12 = 0) ->
(step1'=VoteActActuate)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);

//If no replica has failed and a sensor value is wrong (entire
    column) and any of the other cc-vecors are lost(not
    correspodning to wrong sensor value) -> replica is deemed as
    permanently faulty (2 replicas unable to vote)
[evalVCRAct] step1 = EvalVCR2SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 1 & sensTempWrong2 = 0 & sensTempWrong3 = 0
& ( (ccVectRx11 = 0 & ccVectRx22 = 0) | (ccVectRx11 = 0 &
    ccVectRx23 = 0) | (ccVectRx21 = 0 & ccVectRx22 = 0) | (
    ccVectRx21 = 0 & ccVectRx23 = 0)
  | (ccVectRx22 = 0 & ccVectRx23 = 0) ) ->
(step1'=VoteActActuate)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
[evalVCRAct] step1 = EvalVCR2SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 0 & sensTempWrong2 = 1 & sensTempWrong3 = 0
& ( (ccVectRx21 = 0 & ccVectRx12 = 0) | (ccVectRx21 = 0 &
    ccVectRx22 = 0) | (ccVectRx21 = 0 & ccVectRx13 = 0)
  | (ccVectRx11 = 0 & ccVectRx13 = 0) | (ccVectRx22 = 0 &
      ccVectRx13 = 0) ) ->
(step1'=VoteActActuate)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
[evalVCRAct] step1 = EvalVCR2SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 1
& ( (ccVectRx11 = 0 & ccVectRx12 = 0) | (ccVectRx11 = 0 &
    ccVectRx13 = 0) | (ccVectRx11 = 0 & ccVectRx23 = 0)
  | (ccVectRx12 = 0 & ccVectRx13 = 0) | (ccVectRx12 = 0 &
      ccVectRx23 = 0) ) ->
(step1'=VoteActActuate)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);
```

```
//If no replica has failed and no sensor value is wrong and two
    replica lost two cc-vecors -> replica is deemed as permanently
    faulty (2 replicas unable to vote)
[evalVCRAct] step1 = EvalVCR2SysFail & replicaFailed1 = 0 &
    replicaFailed2 = 0 & replicaFailed3 = 0
& sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 0
& errorCounterComm >= majority ->
(step1'=VoteActActuate)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0);

//If none of the above ocurred -> continue
[evalVCRAct] step1 = EvalVCR2SysFail
& !(replicaFailed1 = 1 & replicaFailed2 = 0 & replicaFailed3 = 0 &
    (ccVectRx22 = 0 | ccVectRx23 = 0))
& !(replicaFailed1 = 0 & replicaFailed2 = 1 & replicaFailed3 = 0 &
    (ccVectRx21 = 0 | ccVectRx13 = 0))
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 1 &
    (ccVectRx11 = 0 | ccVectRx12 = 0))
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0
    & sensTempWrong1 = 1 & sensTempWrong2 = 0 & sensTempWrong3 = 0
    & ( (ccVectRx11 = 0 & ccVectRx22 = 0) | (ccVectRx11 = 0 &
        ccVectRx23 = 0) | (ccVectRx21 = 0 & ccVectRx22 = 0) | (
        ccVectRx21 = 0 & ccVectRx23 = 0) | (ccVectRx22 = 0 &
        ccVectRx23 = 0) ) )
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0
    & sensTempWrong1 = 0 & sensTempWrong2 = 1 & sensTempWrong3 = 0
       & ( (ccVectRx21 = 0 & ccVectRx12 = 0) | (ccVectRx21 = 0 &
           ccVectRx22 = 0) | (ccVectRx21 = 0 & ccVectRx13 = 0) | (
           ccVectRx11 = 0 & ccVectRx13 = 0) | (ccVectRx22 = 0 &
           ccVectRx13 = 0) ) )
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0
    & sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 1
    & ( (ccVectRx11 = 0 & ccVectRx12 = 0) | (ccVectRx11 = 0 &
        ccVectRx13 = 0) | (ccVectRx11 = 0 & ccVectRx23 = 0) | (
        ccVectRx12 = 0 & ccVectRx13 = 0) | (ccVectRx12 = 0 &
        ccVectRx23 = 0) ) )
& !(replicaFailed1 = 0 & replicaFailed2 = 0 & replicaFailed3 = 0 &
    sensTempWrong1 = 0 & sensTempWrong2 = 0 & sensTempWrong3 = 0 &
    errorCounterComm >= majority ) ->
(step1'=VoteActActuate);
```

The step *VoteActActuate* (Listing A.25) models the merged vote on actuation values and actuate phases of the extended control application cycle.

The first command evaluates if there are at least a majority of actuation values. If there are not, the output of this phase will be wrong, $consActuationTempWrong1 = 1$.

The next three commands model the loss of TMs, same as in step *Sense*. Depending on network topology configuration and predetermined TM loss probability $tmProbability1$, all TM replicas can be lost with a corresponding probability $pTMLost1...pTMLost3$ and as a result the variable $consActuationTempWrong1$ can be set to 1.

LISTING A.25: Vote on actuation values and actuate phase of the extended control application cycle

```
//If the replica has not failed permanently or transiently and
   there is not enough messages for voting -> consensus actuation
   value will be wrong
[voteActActuate] step1 = VoteActActuate & replicaFailed1 = 0 &
   consActuationTempWrong1 = 0 & (ccVectRx11 + ccVectRx21 = 0 |
   ccVectRx11 + ccVectRx21 = 1 & actuationTempWrong1 = 1) ->
(consActuationTempWrong1'=1) & (step1'=EvalVoteActActuateSysFail);
//If the replica has not failed permanently or transiently and
   there is enough messages for voting -> consensus actuation value
    may be wrong due to lost TMs
[voteActActuate] step1 = VoteActActuate & replicaFailed1 = 0 &
   consActuationTempWrong1 = 0 & tmProbability1 = 1 & !(ccVectRx11
   + ccVectRx21 = 0 | ccVectRx11 + ccVectRx21 = 1 &
   actuationTempWrong1 = 1) ->
pTMLost1 : (consActuationTempWrong1'=1) & (step1'=
   EvalVoteActActuateSysFail)
+ 1 - pTMLost1 : (step1'=EvalVoteActActuateSysFail);
[voteActActuate] step1 = VoteActActuate & replicaFailed1 = 0 &
   consActuationTempWrong1 = 0 & tmProbability1 = 2 & !(ccVectRx11
   + ccVectRx21 = 0 | ccVectRx11 + ccVectRx21 = 1 &
   actuationTempWrong1 = 1) ->
pTMLost2 : (consActuationTempWrong1'=1) & (step1'=
   EvalVoteActActuateSysFail)
+ 1 - pTMLost2 : (step1'=EvalVoteActActuateSysFail);
[voteActActuate] step1 = VoteActActuate & replicaFailed1 = 0 &
   consActuationTempWrong1 = 0 & tmProbability1 = 3 & !(ccVectRx11
   + ccVectRx21 = 0 | ccVectRx11 + ccVectRx21 = 1 &
   actuationTempWrong1 = 1) ->
pTMLost3 : (consActuationTempWrong1'=1) & (step1'=
   EvalVoteActActuateSysFail)
+ 1 - pTMLost3 : (step1'=EvalVoteActActuateSysFail);
//If the replica has failed permanently or transiently -> continue
[voteActActuate] step1 = VoteActActuate & (replicaFailed1 = 1 |
   consActuationTempWrong1 = 1) ->
(step1'=EvalVoteActActuateSysFail);
```

The step *EvalVoteActActuateSysFail* (Listing A.26) evaluates if the system will fail after the *VoteActActuate* step by checking if more than a majority of of consensus actuation values are wrong, $errorCounterAct >= majority$.

LISTING A.26: Evaluate the system failure

```
//If there are more than a majority of consensus actuation or
   replica failures -> replica is deemed as permanently faulty
[evalAct] step1 = EvalVoteActActuateSysFail & errorCounterAct >=
   majority ->
        (step1'=DiscrepancyVoteActActuate)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) &
            (resetActive1'=0)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0);
//If not -> continue
[evalAct] step1 = EvalVoteActActuateSysFail & errorCounterAct <
   majority ->
```

```
        (step1'=DiscrepancyVoteActActuate) & (ccVectRx11'=0) & (
            ccVectRx21'=0);
```

The step *DiscrepancyVoteActActuate* (Listing A.27) models the occurrence of permanent fault detection due to too many transient faults. If the consensus actuation value was wrong due to transient faults, there is a probability $disThshFalsePosProb$ that due to too many previously occurred consecutive transient faults discrepancy error counter (DEC) reaches its threshold and needlessly resets a replica.

LISTING A.27: Discrepancy threshold reached after the vote on actuation and actuate phase

```
//If the replica has not failed and the the consensus actuation
   value was wrong -> replica may reset due to discrepancy error
   counter reaching its threshold with a disThshFalsePosProb
[disActuate] step1 = DiscrepancyVoteActActuate & replicaFailed1 = 0
    & consActuationTempWrong1 = 1 ->
        disThshFalsePosProb : (step1'=EvalResetSysFail)
        & (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
            consActuationTempWrong1'=1) & (replicaFailed1'=1)
        & (tmProbability1'=1)
        & (ccVectRx11'=0) & (ccVectRx21'=0)
        & (resetActive1'=1)
        + 1 - disThshFalsePosProb : (step1'=EvalResetSysFail);
//If the replica has not failed and the the actuation value was not
    wrong -> continue
[disActuate] step1 = DiscrepancyVoteActActuate & replicaFailed1 = 0
    & consActuationTempWrong1 = 0 ->
        (step1'=EvalResetSysFail);
//If the replica has failed -> continue
[disActuate] step1 = DiscrepancyVoteActActuate & replicaFailed1 = 1
    ->
        (step1'=EvalResetSysFail);
```

The step *EvalResetSysFail* (Listing A.28) evaluates if the system will fail due to transient and permanent failure occurrence of all the components during the reset of one replica. Note that if two or more replicas are reset, the system will fail and this was already evaluated by the step *EvalRepTransSysFail*. This condition has to be evaluated again in this step because of all the guards of a step have to form a universal set, i.e. they have to be mutually exclusive. There are 8 possible network configurations detected by the auxiliary reset model and thus 8 probabilities $pResetSysFail1...pResetSysFail8$ for each of these configurations in which the system fails. How these probabilities are obtained will be clarified by Section A.2 when talking about the auxiliary reset model.

LISTING A.28: Evaluate the system failure during the reset

```
//If the combination of faults that can lead to the system failure
   during the reset happened with one of the configurations -> the
   system has failed
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 2 & interlinks = 1
   & links2 = 2 & links3 = 2 ->
pResetSysFail1 : (step1'=EvalResetFaults) & (sysFail1'=true)
+ 1 -pResetSysFail1 : (step1'=EvalResetFaults);
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 2 & interlinks = 1
   & links2 = 2 & links3 = 1 ->
pResetSysFail2 : (step1'=EvalResetFaults) & (sysFail1'=true)
```

```
+ 1 -pResetSysFail2 : (step1'=EvalResetFaults);
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 2 & interlinks = 1
   & links2 = 1 & links3 = 2 ->
pResetSysFail3 : (step1'=EvalResetFaults) & (sysFail1'=true)
+ 1 -pResetSysFail3 : (step1'=EvalResetFaults);
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 2 & interlinks = 1
   & links2 = 1 & links3 = 1 ->
pResetSysFail4 : (step1'=EvalResetFaults) & (sysFail1'=true)
+ 1 -pResetSysFail4 : (step1'=EvalResetFaults);
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 1 & links2 = 2 &
   links3 = 2 ->
pResetSysFail5 : (step1'=EvalResetFaults) & (sysFail1'=true)
+ 1 -pResetSysFail5 : (step1'=EvalResetFaults);
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 1 & links2 = 2 &
   links3 = 1 ->
pResetSysFail6 : (step1'=EvalResetFaults) & (sysFail1'=true)
+ 1 -pResetSysFail6 : (step1'=EvalResetFaults);
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 1 & links2 = 1 &
   links3 = 2 ->
pResetSysFail7 : (step1'=EvalResetFaults) & (sysFail1'=true)
+ 1 -pResetSysFail7 : (step1'=EvalResetFaults);
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 = 1 &
   resetActive2 + resetActive3 = 0 & switches = 1 & links2 = 1 &
   links3 = 1 ->
pResetSysFail8 : (step1'=EvalResetFaults) & (sysFail1'=true)
+ 1 -pResetSysFail8 : (step1'=EvalResetFaults);
//If not -> contiune
[evalResetSysFail] step1 = EvalResetSysFail
& resetActive1 = 1 & resetActive2 + resetActive3 = 0
& !(switches = 2 & interlinks = 1 & links2 = 2 & links3 = 2)
& !(switches = 2 & interlinks = 1 & links2 = 2 & links3 = 1)
& !(switches = 2 & interlinks = 1 & links2 = 1 & links3 = 2)
& !(switches = 2 & interlinks = 1 & links2 = 1 & links3 = 1)
& !(switches = 1 & links2 = 2 & links3 = 2)
& !(switches = 1 & links2 = 2 & links3 = 1)
& !(switches = 1 & links2 = 1 & links3 = 2)
& !(switches = 1 & links2 = 1 & links3 = 1) ->
(step1'=EvalResetFaults);
//If two or more resets happened -> the system has failed
[evalResetSysFail] step1 = EvalResetSysFail & resetActive1 +
   resetActive2 + resetActive3 > 1 ->
(step1'=EvalResetFaults) & (sysFail1'=true);
//If only one or only one reset happened -> continue
[evalResetSysFail] step1 = EvalResetSysFail & !(resetActive1 = 1 &
   resetActive2 + resetActive3 = 0) & !(resetActive1 + resetActive2
    + resetActive3 > 1) ->
(step1'=EvalResetFaults);
```

The step *EvalResetFaults* (Listing A.29) evaluates the failures that could have happened during the reset of one replica and updates the corresponding local variables. Depending on the number of the surviving links all the faults have to be evaluated for both the replica being reset and the replicas not being reset.

LISTING A.29: Evaluate faults that could have happened during the reset for each of the network components and for reseting and non-reseting replicas

```
//If the system has failed in the previous step -> replica is
    deemed as permanently faulty
[evalResetFaults] step1 = EvalResetFaults & (sysFail1 | sysFail2 |
    sysFail3) ->
(step1'=TransFaultProp)
& (sensTempWrong1'=1) & (actuationTempWrong1'=1) & (
    consActuationTempWrong1'=1) & (replicaFailed1'=1)
& (links1'=0) & (oneSwitchOneLinkInterconnected1'=false) & (
    resetActive1'=0)
& (tmProbability1'=1)
& (ccVectRx11'=0) & (ccVectRx21'=0)
& (sysFail1'=true);

//If the system has not failed in the previous step and this
    replica was reset for different network configurations ->
    evaluate faults that could have happened during the reset
[evalResetFaults] step1 = EvalResetFaults & !(sysFail1 | sysFail2 |
     sysFail3) & resetActive1 = 1 & links1 = 2 ->
2*pLinkFailReset*(1-pLinkFailReset)*pRepFailReset*
    nodeFailSysFailCov : (step1'=TransFaultProp) & (resetActive1'=0)
     & (sysFail1'=true)
+ 2*pLinkFailReset*(1-pLinkFailReset)*pRepFailReset*(1-
    nodeFailSysFailCov) : (step1'=TransFaultProp) & (resetActive1
    '=0)

+ 2*pLinkFailReset*(1-pLinkFailReset)*pTNFPReset*TNFPSysFailCov : (
    step1'=TransFaultProp) & (resetActive1'=0) & (sysFail1'=true)
+ 2*pLinkFailReset*(1-pLinkFailReset)*pTNFPReset*(1-TNFPSysFailCov)
     : (step1'=TransFaultProp) & (resetActive1'=1)

+ 2*pLinkFailReset*(1-pLinkFailReset)*pRepTransientReset*
    nodeTransFaultSysFailCov : (step1'=TransFaultProp) & (
    resetActive1'=0) & (sysFail1'=true)

+ 2*pLinkFailReset*(1-pLinkFailReset)*(1-pRepFailReset-pTNFPReset-
    pRepTransientReset*nodeTransFaultSysFailCov) : (links1'=1) & (
    step1'=TransFaultProp)
& (sensTempWrong1'=0) & (actuationTempWrong1'=0) & (
    consActuationTempWrong1'=0) & (replicaFailed1'=0) & (ccVectRx11
    '=0) & (ccVectRx21'=0)
& (resetActive1'=0)

+ pLinkFailReset*pLinkFailReset : (step1'=TransFaultProp) & (links1
    '=0) & (resetActive1'=0)

+ (1-pLinkFailReset)*(1-pLinkFailReset)*pRepFailReset*
    nodeFailSysFailCov : (step1'=TransFaultProp) & (resetActive1'=0)
     & (sysFail1'=true)
+ (1-pLinkFailReset)*(1-pLinkFailReset)*pRepFailReset*(1-
    nodeFailSysFailCov) : (step1'=TransFaultProp) & (resetActive1
    '=0)

+ (1-pLinkFailReset)*(1-pLinkFailReset)*pTNFPReset*TNFPSysFailCov :
     (step1'=TransFaultProp) & (resetActive1'=0) & (sysFail1'=true)
```

```
+ (1-pLinkFailReset)*(1-pLinkFailReset)*pTNFPReset*(1-
    TNFPSysFailCov) : (step1'=TransFaultProp) & (resetActive1'=1)

+ (1-pLinkFailReset)*(1-pLinkFailReset)*pRepTransientReset*
    nodeTransFaultSysFailCov : (step1'=TransFaultProp) & (
    resetActive1'=0) & (sysFail1'=true)

+ (1-pLinkFailReset)*(1-pLinkFailReset)*(1-pRepFailReset-pTNFPReset
    -pRepTransientReset*nodeTransFaultSysFailCov) : (links1'=2) & (
    step1'=TransFaultProp)
& (sensTempWrong1'=0) & (actuationTempWrong1'=0) & (
    consActuationTempWrong1'=0) & (replicaFailed1'=0) & (ccVectRx11
    '=0) & (ccVectRx21'=0)
& (resetActive1'=0);

[evalResetFaults] step1 = EvalResetFaults & !(sysFail1 | sysFail2 |
     sysFail3) & resetActive1 = 1 & links1 = 1 ->
pLinkFailReset : (step1'=TransFaultProp) & (links1'=0) & (
    resetActive1'=0)

+ (1-pLinkFailReset)*pRepFailReset*nodeFailSysFailCov : (step1'=
    TransFaultProp) & (resetActive1'=0) & (sysFail1'=true)
+ (1-pLinkFailReset)*pRepFailReset*(1-nodeFailSysFailCov) : (step1
    '=TransFaultProp) & (resetActive1'=0)

+ (1-pLinkFailReset)*pTNFPReset*TNFPSysFailCov : (step1'=
    TransFaultProp) & (resetActive1'=0) & (sysFail1'=true)
+ (1-pLinkFailReset)*pTNFPReset*(1-TNFPSysFailCov) : (step1'=
    TransFaultProp) & (resetActive1'=1)

+ (1-pLinkFailReset)*pRepTransientReset*nodeTransFaultSysFailCov :
    (step1'=TransFaultProp) & (resetActive1'=0) & (sysFail1'=true)

+ (1-pLinkFailReset)*(1-pRepFailReset-pTNFPReset-pRepTransientReset
    *nodeTransFaultSysFailCov) : (links1'=1) & (step1'=
    TransFaultProp)
& (sensTempWrong1'=0) & (actuationTempWrong1'=0) & (
    consActuationTempWrong1'=0) & (replicaFailed1'=0) & (ccVectRx11
    '=0) & (ccVectRx21'=0)
& (resetActive1'=0);

[evalResetFaults] step1 = EvalResetFaults & !(sysFail1 | sysFail2 |
     sysFail3) & resetActive1 = 1 & links1 = 0 ->
(step1'=TransFaultProp) & (resetActive1'=0);

//If the system has not failed in the previous step and the replica
     has not failed and this replica was not reset but the other one
     was -> evaluate faults that could have happened during the
    reset
[evalResetFaults] step1 = EvalResetFaults & !(sysFail1 | sysFail2 |
     sysFail3) & replicaFailed1 = 0 & resetActive1 = 0 &
    resetActive2 + resetActive3 = 1 & links1 = 2 ->
2*pLinkFailReset*(1-pLinkFailReset) : (links1'=1) & (step1'=
    TransFaultProp)
+ pLinkFailReset*pLinkFailReset : (links1'=2) & (step1'=
    TransFaultProp) //updated - already included in the
    systemFailure
```

```
+ (1-pLinkFailReset)*(1-pLinkFailReset) :(links1'=2) & (step1'=
    TransFaultProp); //updated - already included in the
    systemFailure

//If the system has not failed in the previous step and the replica
     has not failed and this replica was not reset but the other one
     was but with less than 2 links(evaluated in the prevouse step)
    or no replica was reset -> reset the temporarily wrong variables
[evalResetFaults] step1 = EvalResetFaults & !(sysFail1 | sysFail2 |
     sysFail3) & (
(replicaFailed1 = 0 & resetActive1 = 0 & resetActive2 +
    resetActive3 = 1 & links1 != 2)
| (replicaFailed1 = 0 & resetActive1 = 0 & resetActive2 +
    resetActive3 = 0)) ->
(step1'=TransFaultProp);

//If the system has not failed in the previous step and the replica
     has failed or if there were two or more resets of othere
    replicas -> continue
[evalResetFaults] step1 = EvalResetFaults & !(sysFail1 | sysFail2 |
     sysFail3) & (
(replicaFailed1 = 1 & resetActive1 = 0)
| (replicaFailed1 = 0 & resetActive1 = 0 & resetActive2 +
    resetActive3 = 2)) ->
(step1'=TransFaultProp);
```

The step *TransFaultProp* (Listing A.30) models the propagation of the transient faults effects to the next ECAC.

We assume that by voting the effects of transient faults not causing the reset of a replica are fixed and all the replicas have the same operational state. Then, by executing the next extended control application cycle the replica will yield a correct output.

If due to transient faults actuation and consensus actuation values were wrong, either the transient replica failure occurred or in the both of the steps *VoteSenControl* and *VoteActActuate* all the TM replicas were lost and the steps were not activated. In the earlier case, if the fault occurs after the last vote phase it might propagate to the next ECAC. In the latter case, since both voting phases were not executed the replica has a different operational state from the other replica.

This is models as follows. A replica that has these values wrong will with a certain ratio $transFailPro$ cause the transient fault effects to propagate to the next ECAC.

LISTING A.30: Transient faults effects propagation to the next ECAC

```
//If the replica has not failed permanently and there was a
    transient fault (multiple transient faults affecting the TM
    reception in two votings) -> the effects of this fault might
    propagate to the next ECAC
[transFailureProp] step1 = TransFaultProp & replicaFailed1 = 0 &
    actuationTempWrong1 = 1 & consActuationTempWrong1 = 1 ->
         transFailPropRatio : (step1'=NetworkComponentFail)
         + 1 - transFailPropRatio : (step1'=NetworkComponentFail) &
             (sensTempWrong1'=0) & (actuationTempWrong1'=0) & (
             consActuationTempWrong1'=0);
//If the replica has failed permanently or there was a transient
    fault
```

```
[transFailureProp] step1 = TransFaultProp & replicaFailed1 = 0 & (
   actuationTempWrong1 = 0 | consActuationTempWrong1 = 0) ->
       (step1'=NetworkComponentFail) & (sensTempWrong1'=0) & (
          actuationTempWrong1'=0) & (consActuationTempWrong1'=0);
[transFailureProp] step1 = TransFaultProp & replicaFailed1 = 1 ->
       (step1'=NetworkComponentFail);
```

A.1.2 Switches module

This module will be instantiated only once and it represents the operational (non-failed) switches. The variables presented in Table A.4 represent the local state of the switches module. Note that there is only one variable and it represents the number of operational (non-failed) switches.

TABLE A.5: Switches module local variables

Name	Type	Initial Value	Description
switches	0..2	2	non-failed switches

This module is synchronized with node replica module instances and it executes the following two steps: *NetCompFail* and *EvalResetFaults*. The commands of this module belonging to these steps are executed simultaneously with the corresponding node replica steps.

The first step *NetCompFail* (Listing A.31) models permanent failures of network components in a ECAC. Particularly, in this switches module, any of the switches may permanently fail with a probability $pSwitchHyperCycle$.

LISTING A.31: Network components failure step

```
//If there are two switches -> any of them can fail, one switch
   failure may cause the system failure with a switchFailSysFailCov
    and the exchange of control messages among swithces may cause
   the system failure with a pAllCtrlMsgLost*hypercycleDuration*
   switchesSyncSysFailCov
[netCompFail] switches = 2 ->
       2*pSwitchHyperCycle*(1-pSwitchHyperCycle) * (1-
          switchFailSysFailCov) : (switches'=1)
       + 2*pSwitchHyperCycle*(1-pSwitchHyperCycle) *
          switchFailSysFailCov : (switches'=0)
       + pSwitchHyperCycle*pSwitchHyperCycle : (switches'=0)
       + (1-pSwitchHyperCycle)*(1-pSwitchHyperCycle)*
          pAllCtrlMsgLost*hypercycleDuration*
          switchesSyncSysFailCov : (switches'=0)
       + (1-pSwitchHyperCycle)*(1-pSwitchHyperCycle)*(1-
          pAllCtrlMsgLost*hypercycleDuration*
          switchesSyncSysFailCov) : (switches'=2);
//If there is one switch -> it can fail
[netCompFail] switches = 1 ->
       pSwitchHyperCycle : (switches'=0)
       + 1-pSwitchHyperCycle : (switches'=1);
//If there are no switches -> contiune
[netCompFail] switches = 0 ->
       true;
```

The second step *EvalResetFaults* (Listing A.32) evaluates the failures that could have happened during the reset of one replica and updates the corresponding local variables related to the operational switches. Note that now the failure of two switches is not evaluated, $pSwitchReset * pSwitchReset$, since this evaluation was already included by the auxiliary reset model.

LISTING A.32: Evaluate faults that could have happened during the reset for network components and for reseting and non-reseting replicas

```
//If there are two switches and reset occurred -> evaluate failure
   of the switches not leading to system failure during the reset
[evalResetFaults] switches = 2 & (resetActive1 = 1 | resetActive2 =
    1 | resetActive3 = 1) ->
        2*pSwitchReset*(1-pSwitchReset) : (switches'=1)
        + pSwitchReset*pSwitchReset : (switches'=2)
                                          //updated - already
           included in the systemFailure
        + (1-pSwitchReset)*(1-pSwitchReset) : (switches'=2);
//If not -> continue
[evalResetFaults] !(switches = 2 & (resetActive1 = 1 | resetActive2
    = 1 | resetActive3 = 1)) ->
        true;
```

A.1.3 Interlinks module

This module will be instantiated only once and it represents the operational (non-failed) interlinks. The variables presented in Table A.6 represent the local state of the interlinks module. There is only one variable and it represents the number of both non-failed operational interlinks.

TABLE A.6: Interlinks module local variables

Name	Type	Initial Value	Description
interlinks	0..1	1	non-failed interlinks

This module is synchronized with node replica module instances and it executes the step *NetCompFail*. The commands of this module belonging to this step are executed simultaneously with the corresponding node replica step.

The step *NetCompFail* (Listing A.34) models permanent failures of network components in a ECAC. Particularly, in this interlinks module, both interlinks may permanently fail with a probability $pInterlinkHyperCycle$.

The synchronization with the other step, *EvalResetFaults*, is not necessary in the case of this module since it was already evaluated by the auxiliary reset model.

LISTING A.33: Network components failure step

```
[netCompFail] interlinks = 1 ->
        pInterlinkHyperCycle*pInterlinkHyperCycle : (interlinks'=0)
        + 1-pInterlinkHyperCycle*pInterlinkHyperCycle : (interlinks
           '=1);
[netCompFail] interlinks = 0 ->
        true;
```

A.1.4 Evaluate system failure module

There is a single instance of this module. This module is used to evaluate the system failure and to set its only variable $sysFail$ to $true$ in case it is detected (Table A.7).

TABLE A.7: Evaluate system failure module local variables

Name	Type	Initial Value	Description
sysFail	bool	false	system failure indication

This module is synchronized with node replica module instances and it executes the step *TransFaultProp*. Particularly, this module assesses if more than a majority of node replicas has permanently failed by evaluating the expression $replicaFailed1 + replicaFailed2 + replicaFailed3 > 1$. In case one of the previous evaluation steps detected the system failure all the variables $replicaFailed1$, $replicaFailed2$ and $replicaFailed3$ will be 1. If the system failure is detected the variable $sysFail$ will be set to $true$ and this variable is used by measuring reliability of the system by using the transient probabilities calculation feature of prism describe in Section 9.3.

LISTING A.34: Network components failure step

```
[transFailureProp] replicaFailed1+replicaFailed2+replicaFailed3>1
   ->
        (sysFail'=true);
[transFailureProp] replicaFailed1+replicaFailed2+replicaFailed3<=1
   ->
        true;
```

A.2 Auxiliary models

The auxiliary models are introduced in order to reduce the state space of the main model. This is done by extracting a particular logic from the main model and then executing these models separately. Then, the results obtained by these models are used by the main model. The two auxiliary models are analyzed next.

A.2.1 Auxiliary VCR model

It is used for modeling the VCR details. Recall that the VCR includes multiple retransmissions of cc-vectors. First, cc-vectors are sent multiple times in the same EC. Second, the complete EC round is repeated multiple times.

There are 8 sequential discrete time steps modeling the behaviour of a single EC round listed in Table A.8.

TABLE A.8: VCR module sequential step constants

Name	Type	Value	Description

TMProbCalculcation	int	0	Determine the TM probability to be used in the first window of the first EC of the VCR (TMW) depending on the surviving network components
CCVector1ProblCalculation	int	1	Determine the CC-vector 1 probability to be used in each EC of the VCR depending on the surviving network components
CCVector2ProblCalculation	int	2	Determine the CC-vector 2 probability to be used in each EC of the VCR depending on the surviving network components
TMW	int	3	Trigger Message Window (TMW) step
TX	int	4	Tranmission of cc-vector (TX) step
RX1	int	5	Reception of cc-vector 1 (RX1) step
RX2	int	6	Reception of cc-vector 2 (RX2) step
EndVCR	int	7	Determining the end of the VCR step

The parameters listed in the Table A.9 are the case of reference constants for the VCR model. Some of the parameters are the same as in the main model, thus the values are also the same. The exception are the parameters $switches$, $interlinks$ and $links1...links3$ that are the input for this model. The combination of these parameters represent all the possible network configurations.

TABLE A.9: VCR model parameters

Name	Type	Value	Description
ecNumberVCR	int	3	Number of elementary cycles in the VCR (first attempt + retransmissions)
FSTM	int	64	Frame size of the TM expressed in bytes
numTM	int	4	Number of TMs sent in one EC

FSCCVect	int	512	Frame size of the cc-vector expressed in bytes, which is includes the following: sensing/actiation value + prev_err + integral + set_point
numCCVect	int	4	Number of cc-vectors sent in one EC
BER	double	1E-6	Bit-error ratio
switches	int	par	the number of operating switches
interlinks	int	par	the number of operating switches
links1	int	par	the number of operating links of replica 1
links2	int	par	the number of operating links of replica 2
links3	int	par	the number of operating links of replica 3

The probabilities used by the VCR model are calculated as shown in Table A.10. Note that the $pTMLost2$ is an impossible scenario in which there are 2 switches that are not interconnected and thus it will never be used in the model. But, for the sake of consistency we kept its definition. The logic for determining different TM probabilities based on the surviving network configuration is the same as in the main model, c.f. 9.2.

The network configurations for calculating the probabilities $pCCVectorLost1$ ... $pCCVectorLost4$ are shown in Figure 9.3.

- pCCVectorLost1 is the probability of losing all the cc-vectors sent by the replica R_{tx} to the replica R_{rx} when all the network components are operational. In this case 4 copies of the cc-vector are received by the receiving replica R_{rx} as there are 4 disjunct paths.

- pCCVectorLost2 is the probability of losing all the cc-vectors sent by the replica R_{tx} to the replica R_{rx} when all the network components are operational but one link of the receiving replica R_{rx}. In this case 2 copies of the cc-vector are received by the receiving replica R_{rx} as there are 2 disjunct paths.

- pCCVectorLost3 is the probability of losing all the cc-vectors sent by the replica R_{tx} to the replica R_{rx} when all the network components are operational but one link of the transmitting replica R_{tx}. In this case 2 copies of the cc-vector are received by the receiving replica R_{rx} as there are 2 disjunct paths.

- pCCVectorLost4 is the probability of losing all the cc-vectors sent by the replica R_{tx} to the replica R_{rx} when there are one link of the transmitting replica R_{tx} and one link of the receiving replica R_{rx} connected to the same operational switch. In this case only 1 copy of the cc-vector is received by the receiving replica R_{rx} as there is only one existing path.

TABLE A.10: VCR model probabilities calculation

Name	Calculation Expression	Description
pSingleTMLost	1 - pow(1-BER,FSTM * 8)	Probability of losing one TM
pAllTMLost	pow(pSingleTMLost, numTM)	Probability of loosing all the TM replicas sent in a single TMW
pTMLost1	pAllTMLost * pAllTMLost * pAllTMLost	Probability of losing all TMs: 2S 2L 1I (2 links, 2 TMs)
pTMLost2	1	Probability of losing all TMs: 2S 2L 0I (2 links, 1 TM) - Impossible
pTMLost3	pAllTMLost * pAllTMLost	Probability of losing all TMs: 2S 1L 1I (1 link, 2 TMs)
pTMLost4	pAllTMLost	Probability of losing all TMs: other (1 link, 1 TM)
pSingleCCVectLost	1- pow(1-BER,FSCCVect * 8)	Probability of losing one cc-vector
pAllCCVectLost	pow(pSingleCCVectLost, numCCVect)	Probability of losing all cc-vectors sent in a singe EC (temporal redundancy)
pCCVectorLost1	pAllCCVectLost * pAllCCVectLost * pAllCCVectLost * pAllCCVectLost	Probability of losing all cc-vectors: 2T 2S 1I 2R (2 links, 2 cc-vectors)
pCCVectorLost2	pAllCCVectLost * pAllCCVectLost	Probability of losing all cc-vectors: 1T 2S 1I 2R (2 links, 1 cc-vectors)
pCCVectorLost3	pAllCCVectLost * pAllCCVectLost	Probability of losing all cc-vectors: 2T 2S 1I 1R (1 link, 2 cc-vectors)
pCCVectorLost4	pAllCCVectLost	Probability of losing all cc-vectors: other (1 link, 1 cc-vectors)

This model has only one module. This module represents a single replica and is instantiated 3 times. The variables presented in Table A.11 represent the local state of the node replica module of the VCR model.

TABLE A.11: Node replica module of the VCR model local variables

Name	Type	Initial Value	Description
step1	0..7	0	sequential steps of the VCR

tmRx1	0..1	0	TM received by the replica
ccVectTx1	0..1	0	cc-vector transmitted by the replica
ccVectRx11	0..1	0	cc-vector 1 received by the replica
ccVectRx21	0..1	0	cc-vector 2 received by the replica
ecNum1	0..ecNumberVCR-1	0	the number of ECs in a VCR
tmProbability1	1..3	1	probability of losing all the TMs
ccVectProb11	1..4	1	probability of losing all the cc-vector received from replica 1
ccVectProb21	1..4	1	probability of losing all the cc-vector received from replica 2

Again, the two additional replica modules are modeled by using the concept of *module renaming* as depicted in Listing A.35.

LISTING A.35: Node replica module renaming of the VCR model

```
module NodeReplica2 = NodeReplica1 [
step1=step2,
tmProbability1 = tmProbability2,
ccVectProb11 = ccVectProb12,
ccVectProb21 = ccVectProb22,
tmRx1=tmRx2,
ccVectTx1=ccVectTx2, ccVectTx2=ccVectTx1,
ccVectRx11=ccVectRx12, ccVectRx21=ccVectRx22,
ecNum1=ecNum2,
links1 = links2, links2 = links1
] endmodule

module NodeReplica3 = NodeReplica1 [
step1=step3,
tmProbability1 = tmProbability3,
ccVectProb11 = ccVectProb13,
ccVectProb21 = ccVectProb23,
tmRx1=tmRx3,
ccVectTx1=ccVectTx3, ccVectTx2=ccVectTx1, ccVectTx3=ccVectTx2,
ccVectRx11=ccVectRx13, ccVectRx21=ccVectRx23 ,
ecNum1=ecNum3,
links1 = links3, links2 = links1, links3 = links2
] endmodule
```

The step *TMProbCalculcation* (Listing A.36) determines which of the precalculated TM probabilities $pTMLost1...pTMLost4$ will be used by the subsequent steps depending on the surviving network components that are defined as an input to this model.

LISTING A.36: Determining the probability of losing all TMs depending on network topologty failures

```
[tmProb] step1 = TMProbCalculcation & switches = 2 & links1 = 2 &
   interlinks = 1 ->
       (tmProbability1'=1) & (step1'=CCVector1ProblCalculation);
[tmProb] step1 = TMProbCalculcation & switches = 2 & links1 = 1 &
   interlinks = 1 ->
       (tmProbability1'=2) & (step1'=CCVector1ProblCalculation);
[tmProb] step1 = TMProbCalculcation & !(switches = 2 & links1 = 2 &
    interlinks = 1) & !(switches = 2 & links1 = 1 & interlinks = 1)
    ->
       (tmProbability1'=3) & (step1'=CCVector1ProblCalculation);
```

The step *CCVector1ProblCalculation* (Listing A.37) determines which of the precalculated CC-vector probabilities $pCCVectorLost1$ ... $pCCVectorLost4$ will be used by the subsequent steps depending on the surviving network components. These probabilities apply to the cc-vectors received from the first node replica, i.e. the first of the remaining two instances of node replica modules.

LISTING A.37: Determining the probability of losing all CC-vector sent from the 1st replica depending on network topologty failures

```
[ccVectProb] step1  = CCVector1ProblCalculation & links2 = 2 &
   switches = 2 & links1 = 2 & interlinks = 1 ->
       (ccVectProb11'=1) & (step1'=CCVector2ProblCalculation);
[ccVectProb] step1  = CCVector1ProblCalculation & links2 = 1 &
   switches = 2 & links1 = 2 & interlinks = 1 ->
       (ccVectProb11'=2) & (step1'=CCVector2ProblCalculation);
[ccVectProb] step1  = CCVector1ProblCalculation & links2 = 2 &
   switches = 2 & links1 = 1 & interlinks = 1 ->
       (ccVectProb11'=3) & (step1'=CCVector2ProblCalculation);
[ccVectProb] step1  = CCVector1ProblCalculation & !(links2 = 2 &
   switches = 2 & links1 = 2 & interlinks = 1) & !(links2 = 1 &
   switches = 2 & links1 = 2 & interlinks = 1) & !(links2 = 2 &
   switches = 2 & links1 = 1 & interlinks = 1) ->
       (ccVectProb11'=4) & (step1'=CCVector2ProblCalculation);
```

The step *CCVector2ProblCalculation* (Listing A.38) determines which of the precalculated CC-vector probabilities $pCCVectorLost1$ ... $pCCVectorLost4$ will be used by the subsequent steps depending on the surviving network components. These probabilities apply to the cc-vectors received from the second node replica, i.e. the second of the remaining two instances of node replica modules

LISTING A.38: Determining the probability of losing all CC-vector sent from the 1st replica depending on network topologty failures

```
[ccVectProb] step1 = CCVector2ProblCalculation & links3 = 2 &
   switches = 2 & links1 = 2 & interlinks = 1 ->
       (ccVectProb21'=1) & (step1'=TMW);
[ccVectProb] step1 = CCVector2ProblCalculation & links3 = 1 &
   switches = 2 & links1 = 2 & interlinks = 1 ->
       (ccVectProb21'=2) & (step1'=TMW);
[ccVectProb] step1 = CCVector2ProblCalculation & links3 = 2 &
   switches = 2 & links1 = 1 & interlinks = 1 ->
       (ccVectProb21'=3) & (step1'=TMW);
[ccVectProb] step1 = CCVector2ProblCalculation & !(links3 = 2 &
   switches = 2 & links1 = 2 & interlinks = 1) & !(links3 = 1 &
   switches = 2 & links1 = 2 & interlinks = 1) & !(links3 = 2 &
   switches = 2 & links1 = 1 & interlinks = 1) ->
       (ccVectProb21'=4) & (step1'=TMW);
```

The step *TMW* (Listing A.39) models the loss of all the TM replica depending on the predetermined network configuration stored in variable $tmProbability1$.

LISTING A.39: TMW step - reception of TM replicas

```
[tmw] step1 = TMW & tmProbability1 = 1 ->
        pTMLost1 : (tmRx1'=0) & (step1'=TX)
        + 1-pTMLost1 : (tmRx1'=1) & (step1'=TX);
[tmw] step1 = TMW & tmProbability1 = 2 ->
        pTMLost2 : (tmRx1'=0) & (step1'=TX)
        + 1-pTMLost2 : (tmRx1'=1) & (step1'=TX);
[tmw] step1 = TMW & tmProbability1 = 3 & links1 > 0 & switches > 0
   ->
        pTMLost3 : (tmRx1'=0) & (step1'=TX)
        + 1-pTMLost3 : (tmRx1'=1) & (step1'=TX);
// If there are not links nor switches -> continue (TM will not be
   received)
[tmw] step1 = TMW & (links1 = 0 | switches = 0) ->
        (step1'=TX);
```

The step *TX* (Listing A.40) models the transmission of the cc-vectors to the switch. If the replica is connected with 2 links to 2 interconnected switches, $links1 = 2 \& switches = 2$, then 2 cc-vectors are transmitted and can be lost with probability $pCCVectorLost2$. If the replica is connected with one link to one switch, $links1 >= 1 \& switches >= 1 \& !(links1 = 2 \& switches = 2)$, then a single cc-vector is transmitted and can be lost with probability $pCCVectorLost4$. Lastly, if the cc-vector has already been transmitted(received by the switches) or if no TM has been received, $ccVectTx1 = 1 \| tmRx1 = 0$, the loss of cc-vector during the transmission does not need to be modeled.

LISTING A.40: TX step - trnsmission of cc-vector to other replicas

```
// If the cc-vector has not been transmitted(received by the switch
   ) already and the TM has been received -> send the cc-vector
   successfully or not
[tx] step1 = TX & ccVectTx1 = 0 & tmRx1 = 1 & links1 = 2 & switches
    = 2 ->
        pCCVectorLost2 : (ccVectTx1'=0) & (step1'=RX1)
        + 1-pCCVectorLost2 : (ccVectTx1'=1) & (step1'=RX1);
[tx] step1 = TX & ccVectTx1 = 0 & tmRx1 = 1 & links1 >= 1 &
   switches >= 1 & !(links1 = 2 & switches = 2) ->
        pCCVectorLost4 : (ccVectTx1'=0) & (step1'=RX1)
        + 1-pCCVectorLost4 : (ccVectTx1'=1) & (step1'=RX1);
// If the cc-vector has already been transmitted(received by the
   switch) or if the TM has not been received -> continue (cc-
   vector will not be transmitted)
[tx] step1 = TX & (ccVectTx1 = 1 | tmRx1 = 0) ->
        (step1'=RX1);
```

The steps *RX1* and *RX2* (Listing A.41) model the reception of the cc-vectors from the other replicas. The first three commands model the loss of the cc-vector during the reception depending on the predetermined network configuration $ccVectProb11$. The received cc-vector can be lost with a corresponding probability $pCCVectorLost1$ ... $pCCVectorLost4$ If the cc-vector has not been transmitted, or it has been received already, or the TM has not been received, then the loss of cc-vector during the reception does not need to be modeled.

LISTING A.41: RX step - reception of cc-vectors from other replicas

```
// If the cc-vector 1 has been transmitted and has not been receved
    already and the TM has been received depedning on the
   determined probability(network component confiruration) ->
   receive it or not
[rx1] step1 = RX1 & ccVectTx2 = 1 & ccVectRx11 = 0 & tmRx1 = 1 &
   ccVectProb11 = 1 ->
        pCCVectorLost1 : (ccVectRx11'=0) & (step1'=RX2)
        + 1-pCCVectorLost1 : (ccVectRx11'=1) & (step1'=RX2);
[rx1] step1 = RX1 & ccVectTx2 = 1 & ccVectRx11 = 0 & tmRx1 = 1 & (
   ccVectProb11 = 2 | ccVectProb11 = 3) ->
        pCCVectorLost2 : (ccVectRx11'=0) & (step1'=RX2)
        + 1-pCCVectorLost2 : (ccVectRx11'=1) & (step1'=RX2);
[rx1] step1 = RX1 & ccVectTx2 = 1 & ccVectRx11 = 0 & tmRx1 = 1 &
   ccVectProb11 = 4 ->
        pCCVectorLost4 : (ccVectRx11'=0) & (step1'=RX2)
        + 1-pCCVectorLost4 : (ccVectRx11'=1) & (step1'=RX2);
// If the cc-vector 1 has not been transmitted or has been receved
   already or the TM has not been received -> continue (no
   reception)
[rx1] step1 = RX1 & (ccVectTx2 = 0 | ccVectRx11 = 1 | tmRx1 = 0 )
   ->
        (step1'=RX2);

// If the cc-vector 2 has been transmitted and has not been receved
    already and the TM has been received depedning on the
   determined probability(network component confiruration) ->
   receive it or not
[rx2] step1 = RX2 & ccVectTx3 = 1 & ccVectRx21 = 0 & tmRx1 = 1 &
   ccVectProb21 = 1 ->
        pCCVectorLost1 : (ccVectRx21'=0) & (step1'=EndVCR)
        + 1-pCCVectorLost1 : (ccVectRx21'=1) & (step1'=EndVCR);
[rx2] step1 = RX2 & ccVectTx3 = 1 & ccVectRx21 = 0 & tmRx1 = 1 & (
   ccVectProb21 = 2 | ccVectProb21 = 3) ->
        pCCVectorLost2 : (ccVectRx21'=0) & (step1'=EndVCR)
        + 1-pCCVectorLost2 : (ccVectRx21'=1) & (step1'=EndVCR);
[rx2] step1 = RX2 & ccVectTx3 = 1 & ccVectRx21 = 0 & tmRx1 = 1 &
   ccVectProb21 = 4 ->
        pCCVectorLost4 : (ccVectRx21'=0) & (step1'=EndVCR)
        + 1-pCCVectorLost4 : (ccVectRx21'=1) & (step1'=EndVCR);
// If the cc-vector 2 has not been transmitted or has been receved
   already or the TM has not been received -> continue (no
   reception)
[rx2] step1 = RX2 & (ccVectTx3 = 0 | ccVectRx21 = 1 | tmRx1 = 0) ->
        (step1'=EndVCR);
```

The step *EndVCR* (Listing A.42) models the retransmission logic. Depending on the parameter $ecNum1$ that counts the number of repeated ECs, the whole EC round cycle is repeated or not.

LISTING A.42: VCR end step - determining the last EC of the VCR

```
// If the end of the VCR has been reached -> end
[ecUpdate] step1 = EndVCR & ecNum1=ecNumberVCR-1 ->
        true;
// If the end of the VCR has not been reached -> repeat the EC
[ecUpdate] step1 = EndVCR & ecNum1!=ecNumberVCR-1 ->
        (ecNum1'=ecNum1 +1) & (step1'=TMW);
```

A.2.2 The output of the auxiliary VCR model

The output of this model is defined by the properties from Listing A.43. We define what is the probability that a cc-vector being received by a node replica is lost when the VCR round ends. We do this for each cc-vector being received by each node replica. The ending of the VCR is defined by the label $vcrEnd$. For evaluation VCR ending it is sufficient to use the variables of only one node replica module instance since all instances execute in a lock-step.

LISTING A.43: VCR property verification

```
label "vcrEnd" = step1 = EndVCR & ecNum1 = ecNumberVCR-1;

P=? [ F "vcrEnd"&ccVectRx11=0 ]

P=? [ F "vcrEnd"&ccVectRx21=0 ]

P=? [ F "vcrEnd"&ccVectRx12=0 ]

P=? [ F "vcrEnd"&ccVectRx22=0 ]

P=? [ F "vcrEnd"&ccVectRx13=0 ]

P=? [ F "vcrEnd"&ccVectRx23=0 ]
```

Using these properties we define a prism experiment for all the possible input values of the network components. Remember that these values are defined by the parameters $switches$, $interlinks$, $links1$, $links2$ and $links3$.

The result of the experiment is depicted in Figure A.2. We have removed all the cases that lead to the system failure and also redundant cases, e.g. if there is only one switch left, it does not matter if interlinks exist or not (we have kept one case with $interlinks$ set to 0). Since all the replicas are identical, the results of the experiment are symmetric. Specifically, the columns $ccVectRx11$ and $ccVectRx21$ belonging to the cc-vectors received by the replica 1 are identical to the results of the columns $ccVectRx12$, $ccVectRx22$ and columns $ccVectRx13$, $ccVectRx23$ belonging to the replicas 2 and 3 respectively. This is a proof that the modeling is correct.

Therefore, it is enough that the results of one replica only are evaluated. So, the results of interest are depicted in Figure A.3. Here we detect the specific network configuration scenarios and the corresponding probabilities.

Scenario 1 - when there are no links that connect the replicas to the switches $links1 = 0$ the probability of losing the cc-vectors will be 1.

Scenarios 2-4 - when there is only one switch it does not matter if there are 1 or 2 links, the probabilities are the same since there is only one path from the transmitting to the receiving replica. This means that only one copy of the message is sent. The only difference appears when there are no links in which case the probability or losing all the cc-vectors is 1.

Scenarios 5-20 - evaluate all the combination of links when there are two interconnected switches.

Checking these scenarios and corresponding probabilities is another proof that the modeling logic is correct since e.g. the probability of failing to receive cc-vectors when there are less switch and link replicas is greater than the the probability of failing to receive cc-vectors when there are more switch and link replicas.

From all of these values we have detected that there are only 4 different ones. These will be used as an input to the main model and will be stored in the variables

switches	interlinks	links1	links2	links3	Comm1	Comm2	Comm3	ccVectRx11	ccVectRx21	ccVectRx12	ccVectRx22	ccVectRx13	ccVectRx23
1	0	0	1	1	0	1.30E-21	1.30E-21	1	1	1	1.30E-21	1	1.30E-21
1	0	0	1	2	0	1.30E-21	1.30E-21	1	1	1	1.30E-21	1	1.30E-21
1	0	0	2	1	0	1.30E-21	1.30E-21	1	1	1	1.30E-21	1	1.30E-21
1	0	0	2	2	0	1.30E-21	1.30E-21	1	1	1	1.30E-21	1	1.30E-21
1	0	1	0	1	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1.30E-21	1
1	0	1	0	2	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1.30E-21	1
1	0	1	1	0	1.30E-21	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1
1	0	1	1	1	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
1	0	1	1	2	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
1	0	1	2	0	1.30E-21	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1
1	0	1	2	1	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
1	0	1	2	2	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
1	0	2	0	1	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1.30E-21	1
1	0	2	0	2	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1.30E-21	1
1	0	2	1	0	1.30E-21	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1
1	0	2	1	1	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
1	0	2	1	2	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
1	0	2	2	0	1.30E-21	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1
1	0	2	2	1	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
1	0	2	2	2	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
2	1	0	1	1	0	1.30E-21	1.30E-21	1	1	1	1.30E-21	1	1.30E-21
2	1	0	1	2	0	3.25E-22	2.23E-29	1	1	1	4.21E-43	1	3.25E-22
2	1	0	2	1	0	2.23E-29	3.25E-22	1	1	1	3.25E-22	1	4.21E-43
2	1	0	2	2	0	1.05E-43	1.05E-43	1	1	1	1.05E-43	1	1.05E-43
2	1	1	0	1	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1.30E-21	1
2	1	1	0	2	3.25E-22	0	2.23E-29	1	4.21E-43	1	1	3.25E-22	1
2	1	1	1	0	1.30E-21	1.30E-21	0	1.30E-21	1	1.30E-21	1	1	1
2	1	1	1	1	2.27E-21	2.27E-21	2.27E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21	1.30E-21
2	1	1	1	2	1.30E-21	1.30E-21	4.46E-29	1.30E-21	4.21E-43	1.30E-21	4.21E-43	3.25E-22	3.25E-22
2	1	1	2	0	3.25E-22	2.23E-29	0	4.21E-43	1	3.25E-22	1	1	1
2	1	1	2	1	1.30E-21	4.46E-29	1.30E-21	4.21E-43	1.30E-21	3.25E-22	3.25E-22	1.30E-21	4.21E-43
2	1	1	2	2	3.25E-22	2.23E-29	2.23E-29	4.21E-43	4.21E-43	3.25E-22	1.05E-43	3.25E-22	1.05E-43
2	1	2	0	1	2.23E-29	0	3.25E-22	1	3.25E-22	1	1	4.21E-43	1
2	1	2	0	2	1.05E-43	0	1.05E-43	1	1.05E-43	1	1	1.05E-43	1
2	1	2	1	0	2.23E-29	3.25E-22	0	3.25E-22	1	4.21E-43	1	1	1
2	1	2	1	1	4.46E-29	1.30E-21	1.30E-21	3.25E-22	3.25E-22	4.21E-43	1.30E-21	4.21E-43	1.30E-21
2	1	2	1	2	2.23E-29	3.25E-22	2.23E-29	3.25E-22	1.05E-43	4.21E-43	4.21E-43	1.05E-43	3.25E-22
2	1	2	2	0	1.05E-43	1.05E-43	0	1.05E-43	1	1.05E-43	1	1	1
2	1	2	2	1	2.23E-29	2.23E-29	3.25E-22	1.05E-43	3.25E-22	1.05E-43	3.25E-22	4.21E-43	4.21E-43
2	1	2	2	2	1.05E-43	1.05E-43	1.05E-43	1.05E-43	1.05E-43	1.05E-43	1.05E-43	1.05E-43	1.05E-43

FIGURE A.2: The output of the VCR experiment

switches	interlinks	links1	links2	links3	Comm1	ccVectRx11	ccVectRx21	Scenarios
1	0	0	1	1	0	1	1	1
1	0	0	1	2	0	1	1	
1	0	0	2	1	0	1	1	
1	0	0	2	2	0	1	1	
2	1	0	1	1	0	1	1	
2	1	0	1	2	0	1	1	
2	1	0	2	1	0	1	1	
2	1	0	2	2	0	1	1	
1	0	1	0	1	1.30E-21	1	1.30E-21	2
1	0	1	0	2	1.30E-21	1	1.30E-21	
1	0	2	0	1	1.30E-21	1	1.30E-21	
1	0	2	0	2	1.30E-21	1	1.30E-21	
1	0	1	1	0	1.30E-21	1.30E-21	1	3
1	0	1	2	0	1.30E-21	1.30E-21	1	
1	0	2	1	0	1.30E-21	1.30E-21	1	
1	0	2	2	0	1.30E-21	1.30E-21	1	
1	0	1	1	1	2.27E-21	1.30E-21	1.30E-21	4
1	0	1	1	2	2.27E-21	1.30E-21	1.30E-21	
1	0	1	2	1	2.27E-21	1.30E-21	1.30E-21	
1	0	1	2	2	2.27E-21	1.30E-21	1.30E-21	
1	0	2	1	1	2.27E-21	1.30E-21	1.30E-21	
1	0	2	1	2	2.27E-21	1.30E-21	1.30E-21	
1	0	2	2	1	2.27E-21	1.30E-21	1.30E-21	
1	0	2	2	2	2.27E-21	1.30E-21	1.30E-21	
2	1	1	0	1	1.30E-21	1	1.30E-21	5
2	1	1	0	2	3.25E-22	1	4.21E-43	6
2	1	2	0	1	2.23E-29	1	3.25E-22	7
2	1	2	0	2	1.05E-43	1	1.05E-43	8
2	1	1	1	0	1.30E-21	1.30E-21	1	9
2	1	1	2	0	3.25E-22	4.21E-43	1	10
2	1	2	1	0	2.23E-29	3.25E-22	1	11
2	1	2	2	0	1.05E-43	1.05E-43	1	12
2	1	1	1	1	2.27E-21	1.30E-21	1.30E-21	13
2	1	1	1	2	1.30E-21	1.30E-21	4.21E-43	14
2	1	1	2	1	1.30E-21	4.21E-43	1.30E-21	15
2	1	1	2	2	3.25E-22	4.21E-43	4.21E-43	16
2	1	2	1	1	4.46E-29	3.25E-22	3.25E-22	17
2	1	2	1	2	2.23E-29	3.25E-22	1.05E-43	18
2	1	2	2	1	2.23E-29	1.05E-43	3.25E-22	19
2	1	2	2	2	1.05E-43	1.05E-43	1.05E-43	20

FIGURE A.3: The output of the VCR experiment for a single replica

$pCCVectRx1...4$. These variables will then be used by the main model for a specific network configuration whenever VCR round is modeled.

A.2.3 Auxiliary reset model

It is used for modeling the faults that happen during the rest of one replica. Particularly, this model evaluates if the faults that occur during the reset of one replica will provoke the system failure.

This model is quite similar to the main model considering it includes the same three node replica modules. The switches and the interlinks modules are exactly the same as in the main model and they model the permanent failure of the switches and the interlinks respectively.

There are 4 sequential discrete time steps modeling the behaviour of a replica in a hypercycle round. The steps are defined as prism constants and are listed in Table A.12.

TABLE A.12: Reset module sequential step constants

Name	Type	Value	Description
NetworkComponentFailure	int	0	Permanent network toplology component failure step : switches, interlinks, links
LinkSwitchInter	int	1	Determining if the switch and link are interconnected step
EvalSysFail	int	2	Evaluate if the system failed due to the previous network component failures
FaultsOccurence	int	3	Modeling the system failure occurrence during the reset

The parameters listed in the Table A.13 are the case of reference constants for the reset model. Some of the parameters are the same as in the main model, thus the values are also the same. The exception are the parameters $init\_switches$, $init\_links2$ and $init\_links3$ that are the input for this model. The different combinations of these parameters will represent all the relevant network configurations. Note that, like in the main model, the last 4 values are the parameters $pCCVectRx_i$ that are the output of the auxiliary VCR model.

TABLE A.13: Reset model parameters

Name	Type	Value	Description
init_links1	int	0	initial number of links for the reseting replica
replicaID1	int	1	replica identifier - reseting replica
replicaID2	int	2	replica identifier - non-reseting replica

replicaID3	int	3	replica identifier - non-reseting replica
FSTM	int	64	Frame size of the TM expressed in bytes
numTM	int	4	Number of TMs sent in one EC
FSCtrlMsg	int	64	Frame size of the control messages exchanged among the switches expressed in bytes
numCtrlMsg	int	4	Number of control messages exchanged among switches
BER	double	1e-12	Bit-error rate
resetDuration	double	10	Reset duration in seconds
hypercycleDuration	double	0.02	ECAC duration in seconds
switchFailSysFailCov	double	0.001	Coverage of tolerating a permanent failure of one switch
switchesSyncSysFailCov	double	0.001	Coverage of tolerating a loss of synchronization messages between switches
init_switches	int	par	intital number of switches
init_links2	int	par	intital number of links for the non-reseting replica
init_links3	int	par	intital number of links for the non-reseting replica
PRr	double	1e-5	Permanent replica failure rate
PLr	double	1e-7	Permanent link failure rate
TRr	double	1e-4	Transient replica failure rate
PSWr	double	1e-6	Permanent switch failure rate
TSWr	double	1e-4	Transient switch failure rate
PIr	double	1e-7	Permanent interlink failure rate
PSr	double	1e-5	Power supply failure rate
pCCVectRx1	double	8.9202 980775 29113 E-29	Probability that all the cc-vectors from the sending replica are lost in a VCR when there is 1 switch and any number of receiving and transmitting links (1 useful), or 1 of each links and 2 switches
pCCVectRx2	double	2.2300 745204 80771 E-29	Probability that all the cc-vectors from the sending replica are lost in a VCR when there are 2 switches, 2 receiving links and 1 transmitting link

pCCVectRx3	double	4.9732 323640 97859 E-58	Probability that all the cc-vectors from the sending replica are lost in a VCR when there are 2 switches, 2 receiving links and 2 transmitting links
pCCVectRx4	double	1.9892 931234 958 E-57	Probability that all the cc-vectors from the sending replica are lost in a VCR when there are 2 switches, 1 receiving links and 2 transmitting links

The probabilities used by the reset model are calculated as shown in Table A.14. The same logic as in the previous two models was applied for the calculation of these probabilities.

TABLE A.14: VCR model probabilities calculation

Name	Calculation Expression	Description
pSingleTMLost	1-pow(1-BER,FSTM * 8)	Probability of losing one TM
pAllTMLost	pow(pSingleTMLost, numTM)	Probability of loosing all the TM replicas sent in a single TMW
pTMLost1	pAllTMLost * pAllTMLost * pAllTMLost * pAllTMLost	Probability of losing all TMs: 2S 2L 1I (2 links, 2 TMs)
pTMLost2	1	Probability of losing all TMs: 2S 2L 0I (2 links, 1 TM) - Impossible
pTMLost3	pAllTMLost * pAllTMLost	Probability of losing all TMs: 2S 1L 1I (1 link, 2 TMs)
pTMLost4	pAllTMLost	Probability of losing all TMs: other (1 link, 1 TM)
pSingleCtrlMsgLost	1-pow(1-BER,FSCtrlMsg * 8)	Probability of losing one control message exchanged between switches
pAllCtrlMsgLost	pow(pSingleCtrlMsgLost, numCtrlMsg)	Probability of losing all the control messages exchanged between switches
pRepFailHyperCycle	1- pow(2.718281828459, -hypercycleDuration * (PRr+PSr) / 3600)	Probability of a replica permanently failing during a hypercycle
pSwitchHyperCycle	1- pow(2.718281828459, -hypercycleDuration * (2*(PSWr+PRr)+TSWr + PSr) / 3600)	Probability of switch permanently failing during a hypercycle
pInterlinkHyperCycle	1- pow(2.718281828459, -hypercycleDuration * PIr / 3600)	Probability of an interlink permanently failing during a hypercycle

pFaultOccHyperCycle	1- pow(2.718281828459, -hypercycleDuration * (PRr+PSr+TRr)) / 3600	Probability that any fault causing a system failure occurs during a hypercycle

The replica module is the only one that differs from the main model. This module represents a single replica and is instantiated 3 times. The variables presented in Table A.15 represent the local state of the node replica module of the reset model.

TABLE A.15: Node replica module of the reset model local variables

Name	Type	Initial Value	Description
step1	0..7	0	sequential steps of the VCR
links1	0..2	init_links1	operating (non-failed) links
oneSwitchOneLinkInterconnected1	bool	false	a flag indicating whether a switch and a replica link are interconnected when there is one left of each
sysFail1	bool	false	a flag indicating the system failure

Again, the two additional replica modules are modeled by using the concept of *module renaming* as depicted in Listing A.44.

LISTING A.44: Node replica module renaming of the reset model

```
module NodeReplica2 = NodeReplica1 [
step1 = step2,
links1 = links2, links2 = links1,
oneSwitchOneLinkInterconnected1 = oneSwitchOneLinkInterconnected2,
sysFail1 = sysFail2, sysFail2 = sysFail1,
init_links1 = init_links2,
replicaID1 = replicaID2
] endmodule

module NodeReplica3 = NodeReplica1 [
step1 = step3,
links1 = links3, links2 = links1, links3 = links2,
oneSwitchOneLinkInterconnected1 = oneSwitchOneLinkInterconnected3,
sysFail1 = sysFail3, sysFail2 = sysFail1, sysFail3 = sysFail2,
init_links1 = init_links3,
replicaID1 = replicaID3
] endmodule
```

The first step *NetworkComponentFailure* is depicted in Listing A.45 and models the permanent failures of network components in a ECAC. Particularly, in this node replica module, any of the replica's links may permanently fail with a probability $pLinkFailHyperCycle$. This logic is identical as in the node replica module of the main model.

LISTING A.45: Network components failures

```
//If there are two links and the replica has not failed -> any of
   the links can fail
[netCompFail] step1 = NetworkComponentFailure & links1 = 2 ->
        2*pRepFailHyperCycle*(1-pRepFailHyperCycle) : (links1'=1) &
            (step1'=LinkSwitchInter)
        + pRepFailHyperCycle*pRepFailHyperCycle : (links1'=0) & (
            step1'=LinkSwitchInter)
        + (1-pRepFailHyperCycle)*(1-pRepFailHyperCycle) :(links1
            '=2) & (step1'=LinkSwitchInter);
//If there is one link and the replica has not failed -> the link
   can fail
[netCompFail] step1 = NetworkComponentFailure & links1 = 1 ->
        pRepFailHyperCycle : (links1'=0) & (step1'=LinkSwitchInter)
        + 1-pRepFailHyperCycle : (links1'=1) & (step1'=
            LinkSwitchInter);
//If there no links or if the replica has failed -> continue
[netCompFail] step1 = NetworkComponentFailure & links1 = 0 ->
        (step1'=LinkSwitchInter);
```

The step *LinkSwitchInter* (Listing A.46) models a case when one switch and link are left due to permanent failures of the former, $switches = 1 \& links1 = 1$. If this is the first evaluation of this case, the variable $oneSwitchOneLinkInterconnected1$ is $false$ (initial value). Then, the replica and the switch might be interconnected or not with a 50% chance. If they are not, the replica is deemed as permanently faulty. Again, the logic of this step is identical to the middle part of the step *EvalNetCompSysFail* of the node replica module of the main model.

LISTING A.46: Evaluation of replica and system permanent failures depending on which components of network topology failed

```
//If there is one link and one switch and initially there were 2
   links or 2 switches and they are not interconnected -> they will
    be interconnected or not
[linkSwitchInter] step1 = LinkSwitchInter & switches = 1 & links1 =
    1 & (init_links1 = 2 | init_switches = 2) &
   oneSwitchOneLinkInterconnected1 = false ->
        0.5 : (step1'=EvalSysFail) & (
            oneSwitchOneLinkInterconnected1'=true)
        + 0.5 : (step1'=EvalSysFail) & (links1'=0) & (
            oneSwitchOneLinkInterconnected1'=false);
//If not -> continue
[linkSwitchInter] step1 = LinkSwitchInter & !(switches = 1 & links1
    = 1 & (init_links1 = 2 | init_switches = 2) &
   oneSwitchOneLinkInterconnected1 = false) ->
        (step1'=EvalSysFail);
```

The step *EvalSysFail* (Listing A.47) evaluates if the network topology failures will lead to the system failure.

LISTING A.47: Evaluate if the system failed due to network topology failures

```
//For reseting replica, if there are no more non-reseting replica
   links or switches or if there are 2 disconnected switches -> the
    system failed
[evalSysFail] step1 = EvalSysFail & replicaID1 = 1 & (links2 = 0 |
   links3 = 0 | switches = 0 | switches = 2 & interlinks = 0) ->
        (step1'=FaultsOccurence) & (sysFail1'=true);
```

```
//For reseting replica, if the predicate above is not fullfilled ->
    continue
[evalSysFail] step1 = EvalSysFail & replicaID1 = 1 & !(links2 = 0 |
    links3 = 0 | switches = 0 | switches = 2 & interlinks = 0) ->
        (step1'=FaultsOccurence);
//For non-reseting replica -> continue
[evalSysFail] step1 = EvalSysFail & replicaID1 != 1 ->
        (step1'=FaultsOccurence);
```

The step *FaultsOccurence* (Listing A.48) evaluates if the fault occurrence will lead to the system failure. Depending on different network configurations a system might fail if there is an error that leads to the system failure. If either a fault leading to the system failure occurs and causes an error with probability $pFaultOccHyperCycle$, or if all TMs are lost $pTMLos_i$, or if all cc-vectors are lost $pCCVectRx_i$, the system fails.

LISTING A.48: System failure due to fault occurence

```
//For non-reseting replica, if any of the faults causing the system
    failure occurs -> the system failes
[faultOccurence] phase1 = FaultsOccurence & replicaID1 != 1 &
   switches = 2 & interlinks = 1 & links1 = 2 & links3 = 2 ->
        pFaultOccHyperCycle + 3*pTMLost1 + 2*pCCVectRx3 : (phase1'=
            NetworkComponentFailure) & (sysFail1'=true)
        + 1 - pFaultOccHyperCycle - 3*pTMLost1 - 2*pCCVectRx3 : (
            phase1'=NetworkComponentFailure);
[faultOccurence] phase1 = FaultsOccurence & replicaID1 != 1 &
   switches = 2 & interlinks = 1 & links1 = 2 & links3 = 1 ->
        pFaultOccHyperCycle + 3*pTMLost1 + 2*pCCVectRx2 : (phase1'=
            NetworkComponentFailure) & (sysFail1'=true)
        + 1 - pFaultOccHyperCycle - 3*pTMLost1 - 2*pCCVectRx2 : (
            phase1'=NetworkComponentFailure);
[faultOccurence] phase1 = FaultsOccurence & replicaID1 != 1 &
   switches = 2 & interlinks = 1 & links1 = 1 & links3 = 2 ->
        pFaultOccHyperCycle  + 3*pTMLost3 + 2*pCCVectRx4 : (phase1
            '=NetworkComponentFailure) & (sysFail1'=true)
        + 1 - pFaultOccHyperCycle - 3*pTMLost3 - 2*pCCVectRx4 : (
            phase1'=NetworkComponentFailure);
[faultOccurence] phase1 = FaultsOccurence & replicaID1 != 1
        & !(switches = 2 & interlinks = 1 & links1 = 2 & links3 =
            2)
        & !(switches = 2 & interlinks = 1 & links1 = 2 & links3 =
            1)
        & !(switches = 2 & interlinks = 1 & links1 = 1 & links3 =
            2) ->
        pFaultOccHyperCycle + 3*pTMLost4 + 2*pCCVectRx1 : (phase1'=
            NetworkComponentFailure) & (sysFail1'=true)
        + 1 - pFaultOccHyperCycle - 3*pTMLost4 - 2*pCCVectRx1 : (
            phase1'=NetworkComponentFailure);
//For reseting replica -> continue
[faultOccurence] phase1 = FaultsOccurence & replicaID1 = 1 ->
        (phase1'=NetworkComponentFailure);
```

A.2.4 The output of the auxiliary reset model

The goal of the model is to obtain the probability that the system fails during the reset of one replica. The system will fail if any of the variables $sysFail1...sysFail3$ is true.

switches	links2	links3	pSysFail
2	2	2	1.66666098605149E-05
2	2	1	1.66666376378304E-05
2	1	2	1.66666376378295E-05
2	1	1	1.66666654151450E-05
1	2	2	1.66668529120194E-05
1	2	1	1.66668529120202E-05
1	1	2	1.66668529120192E-05
1	1	1	1.66668529120200E-05

FIGURE A.4: Reset model experiments

Next, we define the number of time units of the simulation. The time units are calculated as displayed by Listing A.49.

LISTING A.49: The calculation of time units for the reset simulation

```
const int timeSteps = floor((resetDuration/hypercycleDuration)*4);
```

The reset duration $resetDuration$ is divided by ECAC duration $hypercycleDuration$ to obtain the number of ECACs. Each ECAC in further divided into 4 sequential steps, i.e. discrete time steps executed one after the other. Therefore, in order to determine the number of time steps in the duration of the predefined reset duration we multiply the number of ECACs with the number of steps in each ECAC.

The output of this model is defined by the property from Listing A.50. We define what is the probability that the system fails during the reset that lasts $timeSteps$ time units.

LISTING A.50: Reset property verification

```
P=? [ F<=timeSteps sysFail1|sysFail2|sysFail3 ]
```

Based upon this property we define an experiment with network configuration values that do not lead to the system failure as seen by Figure A.4. The output of this experiment are the 8 properties that are the input values for the main model $pResetSysFail1...pResetSysFail8$.

Bibliography

Ahamad, Mustaque et al. (1987). "Fault Tolerant Computing in Object Based Distributed Operating Systems." In: *Proceedings-Symposium on Reliability in Distributed Software and Database Systems*. IEEE.

AL Hopkins, Jr, T Basil Smith III, and Jaynarayan H Lala (1978). "FTMP—A highly reliable fault-tolerant multiprocess for aircraft". In: *Proceedings of the IEEE* 66.10, pp. 1221–1239.

Almeida, Luis, Paulo Pedreiras, and José Alberto G Fonseca (2002). "The FTT-CAN protocol: Why and how". In: *IEEE transactions on industrial electronics* 49.6, pp. 1189–1201.

Alstrom, K and Jan Torin (2001). "Future architecture for flight control systems". In: *Digital Avionics Systems, 2001. DASC. 20th Conference*. Vol. 1. IEEE, 1B5–1.

Amir, Yair et al. (1993). "Fast message ordering and membership using a logical token-passing ring". In: *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*. IEEE, pp. 551–560.

Arlat, Jean, Yves Crouzet, and J-C Laprie (1989). "Fault injection for dependability validation of fault-tolerant computing systems". In: *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*. IEEE, pp. 348–355.

Arnold, Thomas F (1973). "The concept of coverage and its effect on the reliability model of a repairable system". In: *IEEE Transactions on Computers* 100.3, pp. 251–254.

Ashjaei, Mohammad, Moris Behnam, and Thomas Nolte (2016). "SEtSim: A modular simulation tool for switched Ethernet networks". In: *Journal of Systems Architecture* 65, pp. 1–14.

Ashjaei, Mohammad et al. (2016). "Improved message forwarding for multi-hop HaRTES real-time ethernet networks". In: *Journal of Signal Processing Systems* 84.1, pp. 47–67.

Avizienis, Algirdas (1985). "The N-version approach to fault-tolerant software". In: *IEEE Transactions on software engineering* 12, pp. 1491–1501.

— (1995). "Building dependable systems: how to keep up with complexity". In: *Proc. IEEE*, pp. 4–14.

Avizienis, Algirdas et al. (2004). "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1, pp. 11–33.

Awtar, Shorya et al. (2002). "Mechatronic design of a ball-on-plate balancing system". In: *Mechatronics* 12.2, pp. 217–228.

Ballesteros, Alberto et al. (2013). "Towards preventing error propagation in a real-time ethernet switch". In: *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE.

Ballesteros, Alberto et al. (2016a). "First implementation and test of a node replication scheme on top of the flexible time-triggered replicated star for ethernet". In: *Factory Communication Systems (WFCS), 2016 IEEE World Conference on*. IEEE.

Ballesteros, Alberto et al. (2016b). "First implementation and test of reintegration mechanisms for node replicas in the FT4FTT Architecture". In: *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE.

Barborak, Michael, Anton Dahbura, and Miroslaw Malek (1993). "The consensus problem in fault-tolerant computing". In: *ACM Computing Surveys (CSur)* 25.2, pp. 171–220.

Barranco, Manuel, Julián Proenza, and Luís Almeida (2009). "Boosting the robustness of controller area networks: CANcentrate and ReCANcentrate". In: *Computer* 42.5.

— (2011). "Quantitative comparison of the error-containment capabilities of a bus and a star topology in CAN networks". In: *IEEE Transactions on Industrial Electronics* 58.3, pp. 802–813.

Barranco, Manuel et al. (2006). "An active star topology for improving fault confinement in CAN networks". In: *IEEE transactions on industrial informatics* 2.2, pp. 78–85.

Barret, PA et al. (1990). "The Delta-4 extra performance architecture (XPA)". In: *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*. IEEE, pp. 481–488.

Bartlett, Joel, Jim Gray, and Bob Horst (1987). "Fault tolerance in tandem computer systems". In: *The Evolution of Fault-Tolerant Computing*. Springer, pp. 55–76.

Bernstein, Philip A. (1988). "Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing". In: *Computer* 21.2, pp. 37–45.

Birman, Kenneth P et al. (1985). "Implementing fault-tolerant distributed objects". In: *IEEE Transactions on Software Engineering* 6, pp. 502–508.

Blischke, Wallace R and DN Prabhakar Murthy (2011). *Reliability: modeling, prediction, and optimization*. Vol. 767. John Wiley & Sons.

Borres, Mark S, Efren O Barabat, and Joy Panduyos (2013). "On Fractional Derivatives and Application". In: *Recoletos Multidisciplinary Research Journal* 1.2, pp. 1–1.

Bouricius, WG, W Ct Carter, and PR Schneider (1969). "Reliability modeling techniques for self-repairing computer systems". In: *Proceedings of the 1969 24th national conference*. ACM, pp. 295–309.

Boyd, Mark A and Sonie Lau (1998). "An Introduction to Markov Modeling: Concepts and Uses". In:

Budhiraja, Navin et al. (1991). "Lower bounds for primary-backup implementations of bofo services". In: *Proceedings of the 2nd Annual Workshop on Ultradependable Multicomputers and Electronic Systems*, pp. 81–86.

Calha, Mkio J and JA Fonseca (2002). "Adapting FTT-CAN for the joint dispatching of tasks and messages". In: *Factory Communication Systems, 2002. 4th IEEE International Workshop on*. IEEE, pp. 117–124.

Chang, Jo-Mei and Nicholas F. Maxemchuk (1984). "Reliable broadcast protocols". In: *ACM Transactions on Computer Systems (TOCS)* 2.3, pp. 251–273.

Chen, L and A Avizienis (1977). "On the implementation of n-version programming for software fault tolerance during program execution". In: *International Computer Software and Applications Conference (COMPSAC)*.

Chérèque, Marc et al. (1992). "Active replication in Delta-4". In: *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*. IEEE, pp. 28–37.

Chtepen, M. et al. (2009). "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids". In: *IEEE Transactions on Parallel and Distributed Systems* 20.2, pp. 180–190. ISSN: 1045-9219. DOI: 10.1109/TPDS.2008.93.

Cooper, Eric C (1984). *Circus: A replicated procedure call facility*. Tech. rep. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING and COMPUTER SCIENCES.

Cristian, Flavin (1991). "Understanding fault-tolerant distributed systems". In: *Communications of the ACM* 34.2, pp. 56–78.

Cristian, Flaviu et al. (1986). *Atomic broadcast: From simple message diffusion to Byzantine agreement*. International Business Machines Incorporated, Thomas J. Watson Research Center.

Defense, UDo (1995). "MIL-HDBK-217F reliability prediction of electronic equipment". In: *Defense, US Department of* 28.

Derasevic, Sinisa, Manuel Barranco, and Julián Proenza (2014). "Appropriate consistent replicated voting for increased reliability in a node replication scheme over FTT". In: *Emerging Technology and Factory Automation (ETFA), IEEE*.

— (2015). "An OMNET++ model to asses node fault-tolerance mechanisms for FTT-Ethernet DESs". In: *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE.

— (2016). "Designing fault-diagnosis and reintegration to prevent node redundancy attrition in highly reliable control systems based on FTT-Ethernet". In: *Factory Communication Systems (WFCS), 2016 IEEE World Conference on*. IEEE, pp. 1–4.

Derasevic, Sinisa, Julián Proenza, and Manuel Barranco (2014). "Using FTT-ethernet for the coordinated dispatching of tasks and messages for node replication". In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE.

Derasevic, Sinisa, Julián Proenza, and David Gessner (2013). "Towards dynamic fault tolerance on FTT-based distributed embedded systems". In: *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE, pp. 1–4.

Derasevic, Sinisa et al. (2015). "First experimental evaluation of the consistent replicated voting in the hard real-time ethernet switching architecture". In: *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE, pp. 1–4.

Di Giandomenico, Felicita and Lorenzo Strigini (1990). "Adjudicators for diverse-redundant components". In: *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*. IEEE, pp. 114–123.

Dodd, Paul E and Lloyd W Massengill (2003). "Basic mechanisms and modeling of single-event upset in digital microelectronics". In: *IEEE Transactions on nuclear Science* 50.3, pp. 583–602.

Dolev, Danny and H. Raymond Strong (1983). "Authenticated algorithms for Byzantine agreement". In: *SIAM Journal on Computing* 12.4, pp. 656–666.

Dolev, Danny et al. (1986). "Reaching approximate agreement in the presence of faults". In: *Journal of the ACM (JACM)* 33.3, pp. 499–516.

Ductor, Sylvain, Zahia Guessoum, and Mikal Ziane (2011). "Adaptive replication in fault-tolerant multi-agent systems". In: *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Volume 02*. IEEE Computer Society, pp. 304–307.

Ferreira, Joaquim et al. (2006). "Combining operational flexibility and dependability in FTT-CAN". In: *IEEE Transactions on Industrial Informatics* 2.2, pp. 95–102.

Garibay-Martínez, Ricardo et al. (2016). "Improved Holistic Analysis for Fork–Join Distributed Real-Time Tasks Supported by the FTT-SE Protocol". In: *IEEE Transactions on Industrial Informatics* 12.5, pp. 1865–1876.

Gessner, D., J. Proenza, and M. Barranco (2014a). "A proposal for managing the redundancy provided by the flexible time-triggered replicated star for Ethernet". In: *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*. DOI: 10.1109/WFCS.2014.6837600.

Gessner, David (2017). "Adding Fault Tolerance to a Flexible Real-Time Ethernet Network for Embedded Systems". PhD thesis. PhD Thesis from University of Balearic Islands, Spain.

Gessner, David, Julian Proenza, and Manuel Barranco (2014b). "A proposal for master replica control in the flexible time-triggered replicated star for Ethernet". In: *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*. IEEE.

Gessner, David et al. (2013). "Towards a flexible time-triggered replicated star for Ethernet". In: *Emerging Technologies & Factory Automation (ETFA), IEEE 18th Conference*.

Gunneflo, Ulf, Johan Karlsson, and Jan Torin (1989). "Evaluation of error detection schemes using fault injection by heavy-ion radiation". In: *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*. IEEE, pp. 340–347.

Kaashoek, M Frans and Andrew S Tanenbaum (1991). "Group communication in the Amoeba distributed operating system". In: *Distributed Computing Systems, 1991., 11th International Conference on*. IEEE, pp. 222–230.

Kalbarczyk, Z.T. et al. (1999). "Chameleon: a software infrastructure for adaptive fault tolerance". In: *IEEE Transactions on Parallel and Distributed Systems* 10.6, pp. 560–579. ISSN: 10459219. DOI: 10.1109/71.774907.

Keichafer, RM et al. (1988). "The MAFT architecture for distributed fault tolerance". In: *Computers, IEEE Transactions on* 37.4, pp. 398–404.

Knezic, Mladen, Alberto Ballesteros, and Julián Proenza (2014). "Towards extending the OMNeT++ INET framework for simulating fault injection in Ethernet-based Flexible Time-Triggered systems". In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE.

Koo, Richard and Sam Toueg (1987). "Checkpointing and rollback-recovery for distributed systems". In: *IEEE Transactions on software Engineering* 1, pp. 23–31.

Kopetz, Hermann (2004). *From a federated to an integrated architecture for dependable embedded systems*. Tech. rep. TECHNISCHE UNIV VIENNA (AUSTRIA).

Kopetz, Hermann and Gunter Grunsteidl (1993). "TTP-A time-triggered protocol for fault-tolerant real-time systems". In: *Fault-Tolerant Computing, FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*. IEEE, pp. 524–533.

Kopetz, Hermann et al. (1989). "Distributed fault-tolerant real-time systems: The Mars approach". In: *IEEE Micro* 9.1, pp. 25–40.

Kopetz, Hermann et al. (2005). "The time-triggered ethernet (TTE) design". In: *Object-Oriented Real-Time Distributed Computing, ISORC. Eighth IEEE International Symposium on*. IEEE, pp. 22–33.

Kwiatkowska, M., G. Norman, and D. Parker (2011). "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, pp. 585–591.

Lala, Jaynarayan H and Linda S Alger (1988). "Hardware and software fault tolerance: A unified architectural approach". In: *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*. IEEE, pp. 240–245.

Lamport, Leslie, Robert Shostak, and Marshall Pease (1982). "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3, pp. 382–401.

Laprie, Jean-Claude (1992). "Dependability: Basic concepts and terminology". In: *Dependability: Basic Concepts and Terminology*. Springer, pp. 3–245.

Laranjeira, Luiz A, Miroslaw Malek, and Roy Jenevein (1991). "On tolerating faults in naturally redundant algorithms". In: *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*. IEEE, pp. 118–127.

Levine, WS. *The Control Handbook. 1996*.

Makam, Srinivas V (1982). *Design study of a fault-tolerant computer system to execute N-version software*.

Makowitz, Rainer and Christopher Temple (2006). "FlexRay - A communication network for automotive control systems". In: *IEEE International Workshop on Factory Communication Systems*, pp. 207–212.

Marau, Ricardo Roberto Duarte (2009). "Real-time communications over switched Ethernet supporting dynamic QoS management". In:

Marzullo, Keith (1990). "Tolerating failures of continuous-valued sensors". In: *ACM Transactions on Computer Systems (TOCS)* 8.4, pp. 284–304.

Morris, Jennifer and Philip Koopman (2005). "Representing design tradeoffs in safety-critical systems". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 4. ACM, pp. 1–5.

Osman, Khairuddin, Mohd Fuaad Rahmat, and Mohd Ashraf Ahmad (2009). "Modelling and controller design for a cruise control system". In: *Signal Processing & Its Applications, 2009. CSPA 2009. 5th International Colloquium on*. IEEE, pp. 254–258.

Pease, Marshall, Robert Shostak, and Leslie Lamport (1980). "Reaching agreement in the presence of faults". In: *Journal of the ACM (JACM)* 27.2, pp. 228–234.

Pedreiras, Paulo and Luis Almeida (2000). "Combining event-triggered and time-triggered traffic in FTT-CAN: Analysis of the asynchronous messaging system". In: *Factory Communication Systems, 2000. Proceedings. 2000 IEEE International Workshop on*. IEEE, pp. 67–75.

— (2003). "The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems". In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 9–pp.

Pedreiras, Paulo, Luis Almeida, and Paolo Gai (2002). "The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency". In: *Proceedings of the 14th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, p. 152.

Perry, Kenneth J and Sam Toueg (1986). "Distributed agreement in the presence of processor and communication faults". In: *IEEE Transactions on Software Engineering* 3, pp. 477–482.

Peti, Philipp et al. (2005). "A maintenance-oriented fault model for the DECOS integrated diagnostic architecture". In: *Parallel and Distributed Processing Symposium, Proceedings. 19th IEEE International*. IEEE.

Poledna, Stefan (2007). *Fault-tolerant real-time systems: The problem of replica determinism*. Vol. 345. Springer Science & Business Media.

Powell, David (1992). "Failure mode assumptions and assumption coverage." In: *FTCS*. Vol. 92, pp. 386–395.

— (2012). *Delta-4: a generic architecture for dependable distributed computing*. Vol. 1. Springer Science & Business Media.

Powell, David, Marc Chérèque, and David Drackley (1991). "Fault-tolerance in Delta-4". In: *ACM SIGOPS Operating Systems Review* 25.2, pp. 122–125.

Powell, David et al. (1999). "GUARDS: A generic upgradable architecture for real-time dependable systems". In: *IEEE Transactions on Parallel and Distributed Systems* 10.6, pp. 580–599.

Powell, David et al. (2001). *A generic fault-tolerant architecture for real-time dependable systems*. Springer.

Proenza, Julián (2007). "RCMBnet: A distributed hardware and firmware support for software fault tolerance". PhD thesis. Ph. D. thesis, Department of Mathematics and Informatics. Universitat de les Illes Balears (UIB).

Proenza, Julián et al. (2012). "The design of the CANbids architecture". In: *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on*. IEEE, pp. 1–8.

Rosset, Valério et al. (2012). "Modeling the reliability of a group membership protocol for dual-scheduled time division multiple access networks". In: *Computer Standards & Interfaces* 34.3, pp. 281–291.

Santos, Rui (2010). "Enhanced Ethernet switching technology for adaptive hard real-time applications". PhD thesis. PhD Thesis from University of Aveiro in Aveiro, Portugal.

Schneider, Fred B (1990). "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4, pp. 299–319.

Shin, Kang G, Tein-Hsiang Lin, and Yann-Hang Lee (1987). "Optimal checkpointing of real-time tasks". In: *IEEE Transactions on computers* 100.11, pp. 1328–1341.

Silva, Valter et al. (2005). "Implementing a distributed sensing and actuation system: The CAMBADA robots case study". In: *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*. Vol. 2. IEEE, 8–pp.

Souto, Pedro Ferreira do, Paulo Portugal, and Francisco Vasques (2016). "Reliability Evaluation of Broadcast Protocols for FlexRay". In: *IEEE Transactions on Vehicular Technology* 65.2, pp. 525–541.

Stephens, Ransom (2004). "Analyzing jitter at high data rates". In: *IEEE Communications Magazine* 42.2, S6–10.

Takano, Tadashi et al. (1996). "In-orbit experiment on the fault-tolerant space computer aboard the satellite Hiten". In: *IEEE transactions on reliability* 45.4, pp. 624–631.

Taylor, D and G Wilson (1989). *The Stratus system architecture*.

Ternon, Cédric, Joël Goossens, and Jean-Michel Dricot (2016). "FTT-openFlow, on the way towards real-time SDN". In: *ACM SIGBED Review* 13.4, pp. 49–54.

Thomm, Isabella et al. (2011). "Automated application of fault tolerance mechanisms in a component-based system". In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM, pp. 87–95.

Varga, András (2001). "OMNET++ Discrete event simulation system". In: *Proc. of the European Simulation Multiconference (ESM'2001)*.

Varga, Andras et al. (2007). *INET framework*. URL: `https://inet.omnetpp.org/`.

W. Craig, Carter (1982). "A time for reflection". In: *Proc. 12th IEEE Int. Symp. Fault-Tolerant Computing. FTCS-12. Santa Monica, California*, p. 41.

Wensley, J.H. et al. (1978). "SIFT: Design and analysis of a fault-tolerant computer for aircraft control". In: *Proceedings of the IEEE* 66.10, pp. 1240–1255. ISSN: 0018-9219. DOI: `10.1109/PROC.1978.11114`.

Wiesmann, Matthias et al. (2000). "Understanding replication in databases and distributed systems". In: *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*. IEEE, pp. 464–474.

Wu, N Eva (2002). "Reliability analysis for AFTI-F16 SRFCS using ASSIST and SURE". In: *American Control Conference, 2002. Proceedings of the 2002*. Vol. 6. IEEE, pp. 4795–4800.

Alphabetical Index