Fault Injection Framework for Time-Triggered Systems

DISSERTATION
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften (Dr.-Ing.)

Dissertation vorgelegt von:

Onwuchekwa, Daniel Lucky

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen

Date of Oral Examination: 09. October 2020

Betreuer und erster Gutachter:
Prof. Dr. Roman Obermaisser, Universität Siegen

Zweiter Gutachter
Prof. Dr. Kristof Van Laerhoven, Universität Siegen

Prüfungskommission:
Prof. Dr. Roman Obermaisser
Prof. Dr. Kristof Van Laerhoven
Prof. Frank Gronwald
Prof. Malte Lochau

Acknowledgement

I wrote this thesis during my employment at the department of Embedded Systems, University of Siegen. I thank Prof. Dr Roman Obermaisser for providing the opportunity for me to be a member of his team, and also for introducing me into the field of fault-tolerant and safety-critical systems. His valuable remarks and criticism advanced my scientific experience.

I would also like to thank my colleague, friend, and football teammate, Tobias Pieper. He played a considerable role that enabled my integration in Germany. He supported my settlement, taught me a lot about the German culture and introduced me to TSG Adler Dielfen football team, a common interest for us. I would also like to thank all my colleagues at the institute. Tobias, Hongie Fang and Maryam Palehvan have been an immense source of inspiration and a pool of knowledge which was beneficial during my work. I thank my colleague also Stefan Otterbach for ensuring that I get all the necessary tools required for the success of this work. I also thank Simon Meckel, Veit Wiese, Michael Schmidt, Hamidreza Ahmadian and Setareh Majidi for the pleasant atmosphere and harmony at work.

Special thanks go to my dearest wife Jennifer, who has been very helpful in providing constructive criticisms and exhibited patience on the way to reach my goal. She also provided the required emotional support to carry out my work. I also thank my kids, Giovanni and Gianna, for providing me with the necessary daily smiles and emotional support. I also thank my sisters Eberechi and Ezinne for their encouragement. I would also like to thank my in-laws for their presence and support in much needed time while I carried out my work. Finally, I would like to thank my friends Ugochukwu Osabiku, Oghneneochuko Obie, and Raymond Webilor for their support.

Zusammenfassung

In dieser Dissertation wird eine Methodik zur Verifizierung und Validierung des Verhaltens von integrierten System vorgestellt, die auf zeitgesteuerten Ethernet-Netzwerken basieren. Der Determinismus und die ausreichende Bandbreite, die durch ein zeitgesteuertes Ethernet-Netzwerk bereitgestellt werden, ermöglichen die Konstruktion sicherheitskritischer Systeme in verschiedenen Bereichen wie Eisenbahn, Luftfahrt, Gesundheit und Automobil. Viele Anwendungen in diesen Bereichen stellen hohe Anforderungen an die Zuverlässigkeit. Deshalb sind Verifizierung und Validierung in den meisten Phasen des Entwicklungsprozesses sicherheitskritischer Systeme erforderlich.

Aufgrund der Komplexität von zeitgesteuerten Netzwerkprotokollen verwenden Entwickler meist formale Methoden und Simulationen als Verifikations- und Validierungstechniken. Allerdings verifizieren und validieren diese Methoden hauptsächlich bestimmte Funktionen der zeitgesteuerten Protokolle und nicht das Verhalten des integrierten Systems. Die Gründe dafür liegen in den Nachteilen dieser Ansätze. Die Modellierung komplexer Systeme führt bei der Benutzung formaler Methoden zu einer Explosion des Zustandsraums und Simulatoren modellieren bestimmte komplexe Funktionen nicht ausreichend. Des Weiteren erfordern Simulatoren eine zusätzliche Verifizierung durch ein physikalisches Netzwerk, um die Aussagekräftigkeit zu verbessern. Da die Evaluierung der physikalischen Realisierung von zeitgesteuerten Ethernet-Netzwerken zu den besten Ergebnissen führt, konzentriert sich diese Arbeit auf die Anwendung der Fehlerinjektion auf physikalische Geräte.

Diese Dissertation schlägt ein neuartiges und topologieunabhängiges Cut-Through-Fehlerinjektions-Framework vor, welches das integrierte Systemverhalten von zeitgesteuerten Ethernet-Netzwerken auswerten kann. Sie bietet zudem eine Lösung für die Fehlererkennung in zeitgesteuerten Netzwerken während des Synchronisationsstarts, bevor eine globale Zeit festgelegt wird. Darüber hinaus werden experimentelle Verfahren und Ergebnisse diskutiert, die die Verwendung des Fehlerinjektions-Frameworks für die Bewertung einer Auswahl verschiedener Anwendungsfälle demonstrieren. Die hier durchgeführten Experimente bestätigen, wie das neuartige Framework andere zeitgesteuerte Ethernet-

Frameworks übertrifft, indem es die kollektiven Anforderungen erfüllt. Hierzu gehören geringe Störanfälligkeit, Portabilität und die Abstraktion der Fehlerinjektionskomponente aus dem zu testenden Netzwerk.

Abstract

This thesis presents a methodology and tool for verifying and validating the integrated system behaviour of time-triggered Ethernet networks. The determinism and sufficient bandwidth provided by time-triggered Ethernet network make it appealing for building safety-critical systems in different domains such as railway, aviation, health, and automobile. Many applications in these domains impose stringent dependability requirements. Therefore, verification and validation are often required at most stages of the development process when designing these systems.

Due to the complexity of time-triggered network protocols, design engineers mostly employ formal methods and simulations as the verification and validation techniques. However, these methods mainly verify and validate only certain functions of the time-triggered protocol and not the integrated system behaviour. The reasons stem from the downsides of these approaches. The formal method suffers from a state-space explosion when modelling complex systems, and simulators do not sufficiently model certain complex functionality. Simulators also require cross-verification from a physical network to gain better confidence. Since evaluating the physical realisation of time-triggered Ethernet networks results in the best confidence levels, this work then focuses on the use of fault injection on physical devices for this purpose.

This work proposes a novel and topology independent cut-through fault injection framework that can be used to evaluate the integrated system behaviour of time-triggered Ethernet networks. This work also describes a technique that can be used for failure detection in time-triggered networks during the synchronisation startup before the establishment of global time. It furthermore presents a discussion of experimental procedure(s) and results that demonstrate the use of the fault injection framework for the evaluation of a selection of different use cases. The Experiments carried out herein confirms how the novel fault injection framework surpasses other time-triggered Ethernet frameworks by satisfying a set of collective requirements which mainly include low-intrusiveness, portability, and the abstraction of fault injection component from the network under test.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and motivation

Today's world is currently experiencing the growing demand for embedded devices for applications in different sectors such as in aviation, health, railway and the military. The placement of these embedded devices is often done in a distributed fashion. As a consequence, computations are carried out locally on embedded devices that are placed in multiple locations. For example, let us consider a railway use case, where several Electronic Control Units (ECUs) are placed in different locations inside a railway vehicle. Each ECU is responsible for handling certain applications such as the Heating, Ventilation, and Air Conditioning (HVAC) system, door system, brake control, and entertainment system. These systems communicate with a central controller which issues control commands. This kind of system is a distributed system since each node performs local computations and also interacts with a central controller.

The communication infrastructure used to coordinate the interaction between devices that are located in different parts of the railway vehicle use-case and its controller is a distributed communication system. In the railway use-case, it can be observed that there is a mix of applications with different safety and real-time requirements. For instance, the door controller has higher safety and real-time constraint compared to the entertainment system. Therefore, embedded system designers tend to consider communication systems that satisfy these requirements. The current trend is towards time-triggered communication based on Ethernet that satisfies mixed-criticality safety requirements. The TTEthernet and the Time-Sensitive Networking (TSN) are examples of time-triggered communications that are based on Ethernet. These protocols satisfy mixed-criticality requirements by providing spatial and temporal separation, at the same time efficiently utilising available bandwidth, in addition to satisfying real-time constraints. This thesis was motivated by a project titled "*SAFE architecture for Robust distributed Application Integration in roLling stock*" (SAFE4RAIL) [Safe4RAIL, 2018] which developed concepts for the distributed communication platform, based on deterministic Ethernet communication. The distributed communication platform is named "Drive-by-Data" (DBD) in the SAFE4RAIL project.

The TTEthernet communication protocol has been used in the avionic industry and was standardised in 2010 [AS6802, 2011]. While the TSN is a group of standards in which all of the sub-standard is not yet completed as of today. Generally, time-triggered Ethernet networks provide the following advantages:

- Increased bandwidth for safety applications: Time-triggered networks based on Ethernet have the potential to provide more bandwidth compared to other protocols. For example, in the railway use case, Ethernet based communication provides more bandwidth compared to existing Train Communication Network (TCN), especially for the safety functions.
- Reduced complexity: The reduced complexity provided by time-triggered Ethernet networks can be illustrated with the railway example which consists of different protocols for high criticality applications and low safety-criticality applications. Currently, the Wire Train Bus/Multifunction Vehicle Bus (WTB/MVB) are used to implement safety functions while an Ethernet-based network named the Ethernet Train Backbone/Ethernet Consist network (ETB/ECN) is used for applications with large bandwidth requirements such as the entertainment systems. There is increased complexity in the maintenance and management of the mixed network. Hence the need for TTEthernet and TSN.
- Reduced Maintenance cost: There is a high cost incurred in maintaining multiple communication platforms as opposed to a single platform. Therefore, hosting all train applications on the same network would yield reduced cost for maintenance.
- Reduced weight: Time-triggered Ethernet networks host network traffic from applications with multiple safety-criticality levels on a single link. This feature avoids the traditional use of multiple wires for applications with different safety levels, hence the overall weight of all wires is reduced.
- Low latency and Jitters for message transmissions.

These benefits of time-triggered Ethernet networks are the reasons why it is a suitable candidate for railway, avionic, and automotive applications. Most verification and validation activities of the time-triggered protocols carried out to this day focus on formal methods and simulations. These methods mainly verified and validated certain functions of the protocol and not the integrated system behaviour. Formal method suffers from a state-space explosion, and are therefore constrained to modelling only a limited set of behaviour. The approach of using simulators for verification and validation requires cross-verification from a physical network to gain better confidence. Certain complex functionality may not be sufficiently modelled in simulation, which poses a question of fidelity for the simulator. For this reason, verification of physical prototypes provides more confidence than formal methods and simulation. Dependability evaluation of a physical prototype can be achieved by fault injection. Verification and validation using physical fault injection for network protocols is not a new technique. However, a suitable framework that uses physical fault injection for time-triggered Ethernet networks such as TTEthernet and TSN to test for the variety of failures covered by major safety standards such as the IEC 61508 [IEC61508, 2010] remains unrealised. Considering the anticipated increase in the deployment of time-triggered Ethernet networks for different application, a verification and validation tool that covers a broad range of failures that can test the integrated services of time-triggered Ethernet networks including their synchronisation services is required.

In time-triggered networks, the correctness of the entire protocols depends on the synchronisation service. A major problem is that it is illogical to rely on the global time base before the completion of the startup process to implement diagnostic protocols to detect faulty network components. This is because the global time base is only established after the startup process. The early detection of failed components during startup would facilitate early activation of diagnostic services, and ensure that fault removal implementations are triggered earlier. Thus, there is the need to implement a solution to handle failures before the completion of the startup.

1.2 Objectives and contribution

This work aims at the research of fault injection techniques capable of verifying and validating the integrated system behaviour of time-triggered Ethernet networks such as TTEthernet and TSN, providing low intrusion and high confidence level. In addition, the work strives to implement a solution to failure detection during synchronisation startup before a global time base is established. In this work, a new fault injection framework named TRAITOR (cuT-thRough fAult InjecTiOn fRamework) for time-triggered Ethernet networks is proposed. TRAITOR is designed and implemented to induce the failures listed in IEC 61508. This work enriches the sparse literature in the verification and validation of time-triggered Ethernet networks using physical fault injection. More insight into the integrated behaviour of the fault-tolerant implementation of time-triggered Ethernet networks during failure scenarios is desired. TRAITOR is aimed at validating time-triggered Ethernet network protocol implementations to assist in the final validation phase of systems designed on top of these protocols. TRAITOR is aimed at satisfying the following requirements:

1. The ability to target the physical execution of the distributed time-triggered protocol encompassing the combined software and hardware implementation. Time-triggered Ethernet networks require the combined operation of a distributed software and designated hardware components. Fault injection frameworks are often aimed at validating specific hardware components or software that are considered as a single unit. TRAITOR is required to target the protocol operation as a whole and not individual hardware or software components. Therefore, the required target domain is the network protocol.
2. TRAITOR is required to target the integrated services of time-triggered Ethernet networks, and the protocols of these networks comprise multiple services. For instance, TSN comprises several sub-protocols such as protocols for synchronisation, bridges and bridged networks, and frame replication and elimination. Likewise, TTEthernet comprises sub-services such as clock synchronisation, and startup service. TRAITOR is required to target the integrated behaviour of all services provided by time-triggered Ethernet networks.
3. TRAITOR is required to target the physical implementation of time-triggered Ethernet networks. The abstraction level of TRAITOR is not aimed at simulation or formal modelling but the physical implementation.
4. TRAITOR is required to be low intrusive. The addition of TRAITOR to the Network Under Test (NUT) should not affect its operation. TRAITOR should not introduce noticeable delays to the NUT that will affect its mode of operation.
5. TRAITOR is required to abstract the fault injector component from the NUT. Most physical fault injection frameworks require modification of the network components under test to achieve error injection, but herein TRAITOR realises the concept for a physical fault injection framework that does not require such modifications.
6. It is required for TRAITOR to be portable to multiple applications utilising time-triggered Ethernet networks as the underlying communication platform. Another dimension to the portability requirement is the applicability of TRAITOR to im-

plementations by multiple vendors. TRAITOR is required to be developed on the concept that different implementations of time-triggered Ethernet networks can be tested without the need to modify vendor-specific implementations or applications.

7. Finally, TRAITOR is required to deliver a technique to diagnose failures during the startup of synchronisation under different failure scenarios. The applicability of TRAITOR in data generation for fault diagnostics is explored. Herein, the use of TRAITOR to generate training data for a neural network is desired. Since time-triggered Ethernet networks are designed to be fault-tolerant systems with Safety Integrity Level 4 (SIL4) [IEC61508, 2010], waiting for a failure to occur in a SIL4 system is not feasible since the mean time to failure is very large. Therefore, TRAITOR is required to have the capability to generate the training and test data for different failure scenarios.

1.3 Thesis structure

The structure of this thesis is shown in Figure 1.1. After the introduction presented in this chapter 1, the background theory is presented in chapter 2 to provide an understanding of the fundamental concepts used in this work. Chapter 3 describes the target networks, that is the TTEthernet and TSN, and provides the details of the characteristics of the respective networks and their prominent features. Chapter 4 provides a discussion about the related works in this field — the existing gaps in the state-of-art and how the proposed technique closes these gaps described in the requirements. Chapter 5 presents a discussion about the system model of TRAITOR. It provides a general overview of TRAITOR's design and operation. The detailed description of the implementation and algorithms of TRAITOR is presented in chapter 6. A series of experiments and evaluations are carried out in Chapter 7 to demonstrated the operation of TRAITOR, showing how it satisfies the design requirements, specifically the requirements which are outlined in Chapter 4. The sections 7.2, 7.3, 7.4, and 7.5 of Chapter 7 report the works carried out using TRAITOR in [Onwuchekwa et al., 2018], [Onwuchekwa and Obermaisser, 2018], [Onwuchekwa and Obermaisser, 2019] and [Onwuchekwa et al., 2020] respectively.The results of each test are also analysed in each of the mentioned sections. Finally, the conclusion of this work is presented in Chapter 8.

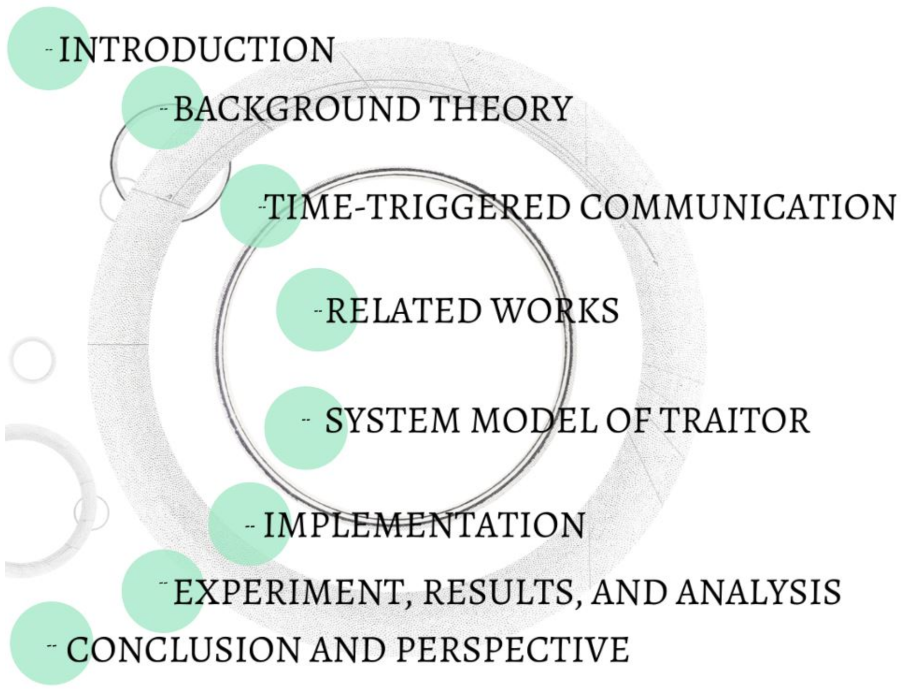

INTRODUCTION
BACKGROUND THEORY
TIME-TRIGGERED COMMUNICATION
RELATED WORKS
SYSTEM MODEL OF TRAITOR
IMPLEMENTATION
EXPERIMENT, RESULTS, AND ANALYSIS
CONCLUSION AND PERSPECTIVE

Figure 1.1: Thesis structure

Chapter 2

Background Theory

This chapter discusses the basic concepts and terminologies used in this work. Firstly, the notion of a real-time system is presented in section 2.1. Section 2.2 introduces the concept of dependability of a system with an emphasis on the following: means to attain dependability, threats to dependability, attributes of dependability, the concept of redundancy, and the methods employed in dependability evaluation. Section 2.3 discusses verification and validation to make clear the distinctions between these concepts. Next section 2.4 explains in detail the idea of fault injection, fault injection categories, and fault injection environment. Section 2.5 then discusses deep learning and the associated terminologies used in this work.

2.1 Real-time systems

A *system* can be defined as a modelling concept which describes the mapping of a set of inputs to a set of outputs [Laplante, 2004]. When only the inputs and outputs of a system are of interest, without consideration of the internal operation of the system, then is the system referred to as a black-box. Conversely, the system is referred to as "white-box" when the interactions between the system's input(s) and its internal dynamics to give a corresponding output(s) are considered. The term *system behaviour* refers to the sequence of *output in time* of a system [Kopetz, 2011]. The intended behaviour of a real-time system according to the specification is known as the *system's service.* If the behaviour violates the specification, the behaviour represents a failure. A real-time system is one where the logical correctness of the system behaviour is tied both to the accuracy of the outputs and the physical time when these outputs are produced.

It can be argued that most systems are real-time; nevertheless, to make clear of the degree of timeliness and consequence of failure, this work divides real-time systems into three classes. Depending on the severity of the consequence of not meeting a system time-constraint, a real-time system can be classified as hard, firm or soft. A hard real-time system is one in which the failure to satisfy the timing constraints can result in a catastrophe such as loss of life, severe damage to the environment and huge financial loss. The firm real-time describes a system in which missed deadlines result in the generation of unusable or inaccurate system outputs. The distinction between the hard and firm real-time system is that missed deadlines in the firm real-time system do not result in a catastrophic consequence, even though the service delivered becomes useless in the event of a missed deadline. The soft real-time is used to describe a system which still produces useful output after a missed deadline.

2.2 dependability of a system

The work in [Laprie, 1992] described the dependability of a system as the measures taken to justify the reasons for trusting the services delivered by the system. In more recent times, a more refined definition in [Avizienis et al., 2004] described dependability of a system as the ability to avoid service failures that are more frequent and more severe than acceptable. This work adopts the concepts of dependability presented in [Avizienis et al., 2004], which attempted to establish a consensus of concepts for dependability.

2.2.1 Means to attain dependability in a system

The means to attain dependability in a system can be classified into four categories namely, *fault prevention*, *fault tolerance*, *fault removal* and *fault forecasting*. A system that provides verified fault preventive mechanisms and fault-tolerance services can be considered to deliver trusted services. Fault prevention and fault tolerance aim to deliver trusted services while fault removal and fault forecasting aim to reach confidence in that ability.

Fault prevention is the means to prevent the occurrence or introduction of faults. Fault prevention is implemented during the development phase of system design. Some examples of prevention strategies for software and hardware can include the use of a modular structure for programming and information hiding, and the use of design rules.

Fault tolerance is the means to deliver correct service in the presence of faults. It is carried out in dependable systems using fault detection, fault containment, fault localisation, fault recovery and fault masking mechanisms [Barry, 1989]. Fault detection identifies whether there is a fault present in a system. Fault containment is the process of isolating the fault to prevent error propagation through the system. Fault localisation involves the determination of where a fault occurs. Fault recovery mechanisms transform a system ladened with error into a state without errors. Fault masking achieves fault tolerance by hiding faults, such that it prevents a system from introducing errors into its information structure.

Fault removal is the means to reduce the number or severity of faults. It can be considered all through a system lifecycle, from development to deployment. During the development phase, fault removal involves three steps, namely, verification, diagnosis and correction. Every embedded system must be designed to meet certain requirements, and these requirements are called *properties* or *specifications*. Verification is the process of checking whether systems adhere to the specified property. Verification is discussed in more details in section 2.3. Diagnosis and correction of faults are carried out when the system fails its verification. Fault removal for an already deployed system is usually in the form of corrective or preventive maintenance.

Fault forecasting is the technique used to predict the occurrence of a fault in a system, to isolate, remove or circumvent these faults. This is achieved by evaluating the system behaviour either qualitatively (ordinal evaluation) or quantitatively (probabilistic evaluation).

2.2.2 Threats to the dependability of a system

The threats to a dependable system are *faults*, *errors*, and *failures* [Arlat et al., 1993]. A fault is a physical defect, imperfection, or flaw that occurs in a hardware or software component of a system. Precisely, a fault can be defined as the adjudged or hypothesised cause of an error. An error is the part of a system's total state that may lead to a failure [Avizienis et al., 2004]. When a fault results in an error, it is said to be "active" otherwise it is "dormant". Errors can be propagated from one system sub-state to another, eventually resulting in the system's failure, except if it is tolerated or contained. Service failure is the event that occurs when the error is perceived by the users or external systems [Natella et al., 2016]. The deviation of a system from providing the correct service is termed *service failure*. The service failure may manifest in different ways known as *service failure modes*. The "chain of threats" shown in figure 1, as cited in [Avizienis et al., 2004] illustrates the relationship between fault, error and failure. If a fault produces an error when activated, this error can be propagated from one system component to another if not contained, which can then result in the system's failure. In the case of a complex system where the failure is localised on a certain component, the failed component has the potential to cause another fault, thereby prolonging the chain of events. The taxonomy of faults given in [Avizienis et al., 2004] classified faults into three major partially overlapping groupings, namely, development faults, physical faults, and interaction faults. The development faults are the faults that occur during the development process of a system. Physical faults are all the possible faults that affect hardware component of a system. Interaction faults are all the faults that originate externally to the system.

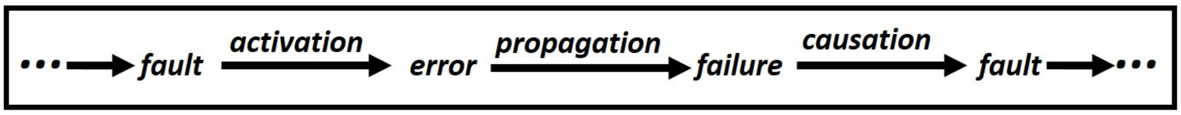


Figure 2.1: The fundamental chain of dependability and security threats [Avizienis et al., 2004]

Apart from the service failure, failures can also be classified into two additional classes; the development failures, and dependability and security failures. Development faults may be introduced into a system during the development phase by its environment, human developer, and development tools. These faults may lead to partial or development failures or may remain undetected in the development phase and manifest in the use phase. Failures that occur during the development phase are termed as development failures. Similarly, it is to be expected that various kinds of faults can also affect a system during it's use phase causing unacceptably degraded performance or even total service failure. A dependability and security specification is agreed upon to states goals for dependability attributes (see next section 2.2.3 for dependability attributes). Thus a dependability and security failure occurs in the use phase when the given system suffers service failures more frequently or severely than acceptable.

2.2.3 Attributes of the dependability of a system

The dependability concept is integrative, it encompasses the attributes: *availability*, *reliability, safety, integrity,* and *maintainability.* Availability can be defined as the readiness of a system to provide correct service. The reliability of a system is the probability that a system will continue to provide the specified service until time t, given that the system was operational at $t = t_o$ [Kopetz, 2011]. Safety is reliability regarding malign (critical) failure modes. Safety is further explained in Subsection 2.2.6. Integrity describes the absence of improper system alterations. Maintainability is used to describe the ability of a system to undergo modification and repair.

2.2.4 Redundancy

Redundancy is an essential requirement in fault tolerance [Verma et al., 2011]. Redundancy is the introduction of additional support of information, time or resources to a system, to give the system fault-tolerance capabilities and prevent failure. Redundancy can be attained by replicating hardware, software, information or by addition of extra time to perform a certain function.

Hardware redundancy

Hardware redundancy is the addition of extra hardware for the purpose of detecting or tolerating faults. Hardware redundancy can be achieved using three techniques, namely, **passive**, **active** and **hybrid** methods [Verma et al., 2011]. In *passive* technique, fault masking is achieved by replicating hardware and exploiting voting or median of the module output to determine the correct output, such as in Triple Modular Redundancy (TMR) or N-Modular redundancy. The *active* technique achieves fault tolerance by detecting the faults and then performing a function to remove the faulty hardware. Detection is achieved using a comparison function. The *hybrid* technique combines the passive and active method.

Information redundancy

Information redundancy occurs when redundancy is introduced into the transmitted data or into the memory of a system. Techniques to verify the correctness of message transmission are exploited, such as checksums, cyclic codes, and duplicate codes used to store additional information.

Time redundancy

Time redundancy is achieved by the repetition of computations targeted at detecting transient faults. A computation can be performed multiple times, or a message retransmitted multiple times to detect transient faults. However, extra time is required and thus, can pose challenges in applications with stringent timing requirements. Nevertheless, it is useful when extra time can be provided, for instance, using high-speed devices and in situations where additional hardware cannot be used for redundancy.

Software redundancy

Software redundancy is the addition of extra software for the purpose of detecting or tolerating faults. Software redundancy can be implemented in different forms. For instance, a complete replica of a program can be implemented. The downside is that software flaws are often a result of incorrect design. Therefore, given a flawed specification, different software implementations based on this specification will fail in a correlated manner. However, techniques such as *consistency checks*, *capability checks*, and *N-version programming* are used to implement software redundancy. The N-version programming uses the comparisons of designs, code and results of multiple versions of the same software to detect design flaws. Consistency checks use prior knowledge about output to detect certain errors. Capability checks are performed to verify that a system possesses the capability expected.

2.2.5 Methods of dependability evaluation

There are several methods of evaluating the dependability of a system. A classification based on the intent of these methods is given in [Benso and Prinetto, 2003]. These methods include dependability evaluation by analysis, field experience, and testing. In dependability evaluation by analysis, the analysis of a process or product can be applied at different stages of a product development lifecycle. For example, in the 'concept phase' of product design, some applied techniques include performing comparative studies, Preliminary Hazard Identification and Analysis (PHIA), risk analysis, risk evaluation, and consequence analysis. In later design phases, when the system architecture is available, other techniques applied include reliability block diagram, Fault Tree Analysis (FTA), Failure Mode and Effect Analysis (FMEA), and simulations. However, dependability evaluation by analysis can be affected by high complexities, unavailability of certain information, and invalid assumptions.

After a system has gone into service, its dependability can also be evaluated. This evaluation is known as evaluation by field experience. However, field experience is not feasible when the dependability requirement is very high. For example, the relationship between the observation time T, desired confidence (γ) and the hazardous failure rate (λ) are shown in equation 2.1. The reaching of 95% confidence for a Safety Integrity Level 4 (SIL4) system, $\lambda = 10^{-9}$, requires an infeasible observation time. This spells out the challenges with evaluation by field experience.

$$T = -\ln(\gamma)^{*}1/\lambda \tag{2.1}$$

Dependability evaluation by fault injection is deployed when direct evidence by testing is a feasible way to provide the required evidence for validation. Fault injection is discussed in section 2.4 in more detail. However, it is important to note that dependability evaluation by fault injection reduces the observation time needed to verify/validate a system.

2.2.6 Safety

There are respective boundary conditions for safety in different fields. For instance, in the automotive industry, safety is often discussed as the property of a system that describes the probability of correct operation without critical failures that could cause harm to people or the environment. Automotive producers stretch this definition to mean that the system should cause no accident by any means. This is because even if there is no harm to people, drivers could still cause an accident. Therefore, producers make efforts to prevent any possible occurrence of accidents [Leveson, 2012].

In the area of risk management, safety can be defined as freedom from unacceptable risk. In this context, the risk must be tolerable to be safe [IEC61508, 2010]. However, [Committee et al., 1999] gives a general definition of system safety as the application of engineering and management principles, criteria, and techniques to optimise all aspects of protection within the constraints of operational effectiveness, time and cost throughout all phases of the system life cycle.

2.2.7 Safety-criticality system

A safety-criticality system is a system whose failure can result to harm impacted onto the physical world (people, property or environment). Safety-criticality systems exist across several domains. In the medical field, some examples of safety-criticality systems include the insulin pump hardware, heart pacemaker, defibrillator machines, and medical imaging devices (e.g. X-ray). Similar examples of safety-criticality systems in the automotive industry include systems such as the airbag, braking, seat belt, power steering and battery management systems. In the aviation industry, similar examples include the engine control, air traffic control and radio navigation system. The vast majority of these safety-criticality applications require real-time guarantees, which describes the property of a system to deliver its services to meet a specified deadline.

The combination of real-time and safety requirements in cyber-physical systems demands a communication platform that satisfies stringent safety and real-time requirements. For example, communication protocols such as the TTEthernet, and TSN [Farkas et al., 2018] implement services to meet both real-time and reliability requirements. This is because, in many safety criticality systems, the reliability of the network is important for the safety of the system. TTEthernet and TSN are within the scope of this work and are therefore discussed extensively in chapter 3.

2.3 Verification and validation

The diverse scientific and technological areas of embedded devices have led to different definitions of the concept of Verification and Validation (V & V). Verification is described in [Babuska and Oden, 2004] as the processes of establishing if a computational model and the code implementing the computational model represent the mathematical model of the design with sufficient accuracy. The IEEE Standard Glossary of Software Engineering Terminology' defines verification as the process of determining whether or not the product of a given phase of the software development cycle fulfils the requirements established during the previous phase whereas validation is the process of evaluating software at the end of the software development process to ensure compliance with the software requirements. [Boehm, 1984] extends the definition of validation to include the activities at the beginning of a software definition process: activities that determine the fitness or worth of a software product for its operational mission. Another definition of validation is given in [Oberkampf and Roy, 2010] as the substantiation that a computerised model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model. The distinction between verification and validation is often unclear and has no universal consensus.

In [Grady, 2010], validation is considered to mean a process carried out to show that one or several requirements are clearly understood and that it is possible to satisfy these requirements through design work within the current technological state of the art, funding, and schedule whereas verification is defined as the proof process for unequivocally revealing the relationship between the content of a specification and the characteristics of the corresponding product. This work follows the definition of verification and validation in the IEEE Standard Glossary of Software Engineering Terminology, since it answers the question "Is the right system being built?" for validation, and addresses verification to answer the question "is the system being built right?". These two questions can be used to establish a clear difference between verification and validation.

2.4 Fault injection

Fault injection is a dependability evaluation technique, as mentioned in section 2.2.5. A fault injection system sets up controlled experiments for a target system into which faults are deliberately introduced. The user of the fault injection framework then observes the target system to determine or analyse the following: the impact of faults on performance, analyse fault tolerance mechanisms and test the resilience of the system against known faults. Unlike the evaluation of a system by field experience, which takes a long time, fault injection is used to accelerate the occurrence and propagation of faults into a system to evaluate the system's dependability.

[Arlat et al., 1993] defines fault injection as a technique used to evaluate Fault Tolerance Algorithms and Mechanisms (FTAM's) in line with the faults that they are intended to tolerate. When considering the use of fault injection on FTAMs, its intent can be seen in two dimensions: either it is used for fault removal or fault forecasting. If fault injection is used for fault removal, it is used to target the reduction of faults present in the design and implementation of FTAM. However, if fault injection is used for fault forecasting, it can be used to rate the efficiency of FTAM by evaluating the system.

Fault injection often aims at evaluating the match between a system's response and its specification, in the presence of faults. Nevertheless, the main objectives of fault injection remain for fault removal and fault forecasting. These two methods yield the following benefits as described in [Ziade et al., 2004].

- Provide insights into the effect of the fault on system functionality and performance.
- Enable the assessment of implemented fault tolerance mechanisms.
- Enable the forecasting of faults in a system, in particular encompassing the measurement of the coverage provided by the fault tolerance mechanisms.
- Identification of weak points in system design.
- Studying the system's behaviour in the presence of faults, for example, fault containment regions and error propagations.

Fault removal and fault forecasting are not usually used separately in practice. They are used together to help the designer improve the system.

2.4.1 Fault injection categorisation

Three major categories can be used to classify fault injection techniques [Benso and Di-Carlo, 2011]. These categories include:

- Classification based on the target domain.
- Classification based on the abstraction layer.
- Classification based on the level of intrusiveness.

Figure 2.2 illustrates these classifications with a diagram. Classification based on the target domain expresses whether the fault injection framework targets hardware systems

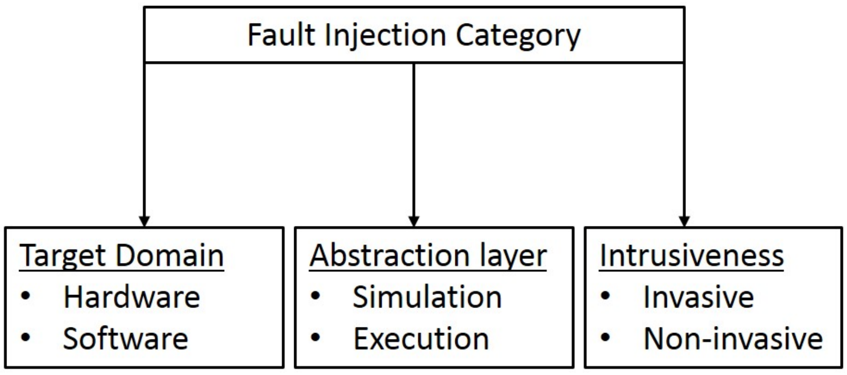


Figure 2.2: Fault Injection Category

or software. Faults can be directly injected at pin-level (internally or externally) into hardware which could either be an emulated component of the system or the actual system. In software, faults can be inserted at compile time or into running software, at the CPU registers or networks.

The classification of fault injection based on the abstraction layer is illustrated using two approaches: simulation-based and execution based. Simulation-based fault injection is employed in the early life of a system when no actual implementation is available. An abstraction/model of the system is developed in which faults are injected. In contrast, the execution based fault injection involves injecting faults into a prototype version of the system. Faults are directly injected into the real system, and the system execution is observed.

Classification of fault injection based on the intrusion level is described using the terms "invasive" and "non-invasive". In setting up a fault injection experiment, the addition of the fault injection component may introduce noticeable footprints. For example, a fault injector component may introduce delay into the system. The fault injection is said to be invasive when such a footprint is noticeable. However, when such a footprint is masked, or the fault injection has no effect on the system under test, then it is termed as non-invasive fault injection.

2.4.2 Fault injection environment

The fault injection environments is a term used to describe the various components used to perform a fault injection campaign. These components include the *fault injector*, *workload generator*, *fault library*, *workload library*, *controller*, *monitor*, *data collector*, and *data analyzer* [Hsueh et al., 1997]. The *fault injector* is the component that introduces faults into a target system by executing commands from a *workload generator* which generates the workload for the system input. The *fault library* stores the set of fault

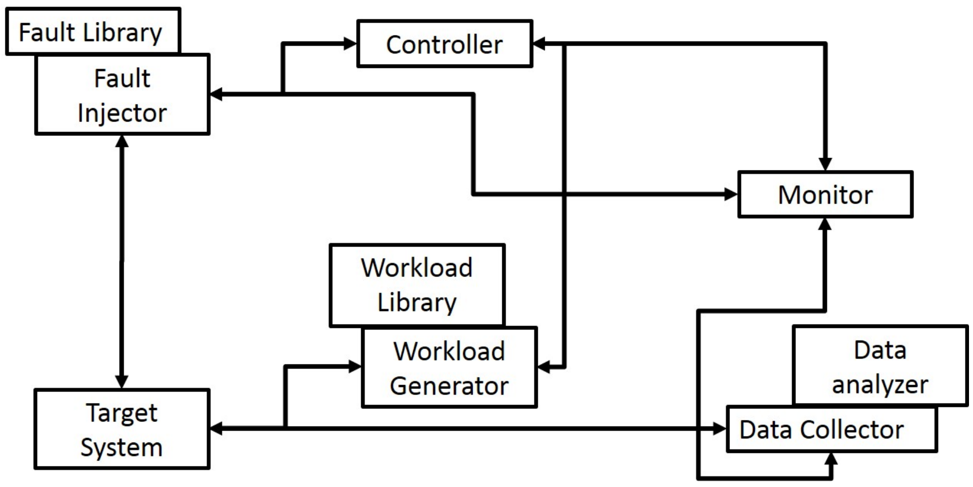


Figure 2.3: Basic components of a Fault injection framework

types, fault locations and fault times for the fault injection. The *workload library* stores the sample workloads for the target system. The controller determines the behaviour of the experiment. The *monitor* tracks the execution commands in the fault injection implementation, while the *data collector* gathers the data observed during the experiment, which is then examined by the *data analyzer*. The model of a fault injection framework illustrating the basic components and the relationship between them as described by [Hsueh et al., 1997] is shown in Figure 2.3. The controller in the diagram triggers the injection and selects the type of fault to inject.

2.4.3 Modelling a fault injection framework

[Arlat et al., 1993] proposed the Faults Activations Readouts and Measures (FARM) model to model a fault injection framework. The work proposed experimental evaluation methods that include concepts of a fault injection *test sequence*, characterised by an *input* domain and an *output* domain. The input domain is used to represent a set of injected faults F and a set A that specifies the data used for the *activation* of the target system, and consequently, of the injected faults. Fault activation is the application of an input to a component that has the potential to cause a fault to become active. Most internal faults cycle between a dormant state and an active state. The output domain represents a set of *readouts* R that are obtained to determine the behaviour of the target system in the presence of a fault, and a set of *measures* M that are derived from analysing the *FAR* sets.

[Benso and DiCarlo, 2011] used the terms *temperance, justice, prudence and courage* to discuss aspects of the FARM model concerning a fault injection environment. Temperance is described as the virtue suggested in choosing an effective fault model *F*. Justice is the

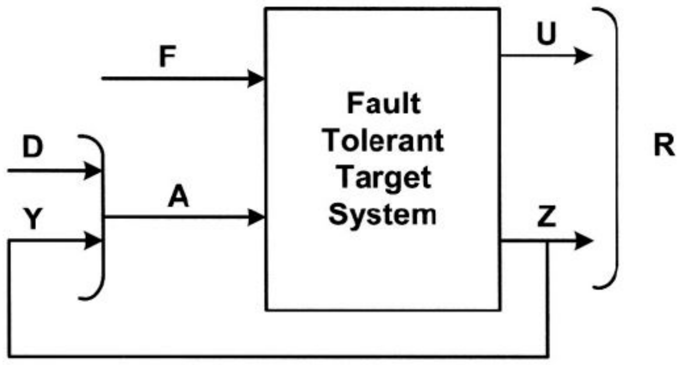


Figure 2.4: Global framework depicting the applicability of FARM model, [Benso and Prinetto, 2003].

virtue that is necessary when choosing a set of activation A that strictly reflect real working conditions, and cannot be chosen just to make an experiment work. Prudence should be applied in the choice of readouts R from a fault injection experiment. Finally, real and useful results require analysing the raw data with courage, without being afraid of having to add experiments or modify the methodology to obtain significant results.

A fault injection model is affected by the level of abstraction of the target system and the fault-tolerant requirements for the target system [Benso and Prinetto, 2003]. Three levels of abstraction are identified in [Arlat et al., 1993]: *Axiomatic models, empirical models and physical models*. The axiomatic model describes a high abstraction layer where reliability block diagrams, fault trees and Markov chain modelling can be used to build analytic models to represent the target system. The empirical model uses more detailed information about the system structure and behaviour to build a model. Lastly, the physical model is a prototype of the actual system, when used as the target system in a fault injection campaign. The experimental data gathered from the physical model has the potential to affirm or deny the hypothesis for parameter selection of axiomatic models.

[Benso and Prinetto, 2003] provided a global framework depicted in Figure 2.4 that characterises the application of fault injection for testing the FARM of a target system. The input domain is represented as $I = \{D \times Y \times F\}$, where D is a set that designates the external input data, Y is a set that defines the current internal states, and F is a set of injected faults. The output domain is represented as O = {Z × U}. Z is a set that defines the internal state of the target fault-tolerant system, and U is a set that represents the services provided by the target system.

The FARM model explicitly describes the fault space in the input domain $I = D \times Y \times F$.

It enables the target system to be described using a function f that relates the input domain to the output domain $O = Z \times U$. The target system behaviour is then described by a sequence of states that perceives the impact of the fault. This is formally described using equation 2.2.

$$\forall t, \exists d(t)\, and/or\, y(t)\, such\, that\, f(d, y, f; t) \neq f(d, y, f_o; t) \tag{2.2}$$

Where $f_o(t)$ in equation 2.2 indicates the "absence of fault". This reads that at all times t, there exists an element of the set D, d(t), and or y(t) (i.e. an element of the set Y, internal system state) such that there is a deviation from the expected output trace under a given fault. This activation corresponds to the deviation from the expected behaviour of the system, which is observed by the expected trace of the output (Z or U, or in both Z and U shown in Figure 2.4). When only the state Z is altered, an internal error is perceived. However, when such activation affects the service of a system due to failure, U also deviates from the specified service.

2.4.4 Types of fault injection

Over the past years, many fault injection techniques have been proposed and used. They can be classified into five categories: hardware-based fault injection, software-based fault injection, simulation-based fault injection, emulation-based fault injection and hybrid fault injection, as surveyed by [Ziade et al., 2004]. Hybrid fault injection is the integration of two or more techniques to exploit the advantages of each technique. The description, benefits and drawbacks of the fault injection techniques are given as follows.

- **Hardware-based fault injection**. This involves changing the parameters at a physical level, and faults are usually injected into a prototype system. Depending on the type of faults to be injected into a target system, faults can be injected either by direct contact or without contact. In chip manufacturing, an example of fault injection by direct contact is by forcing faulty analogue signal to the pins. Non-contact fault injection can be induced by changing the physical environment, that is applying techniques such as heavy-ion radiation or electromagnetic interference. The target hardware can be a prototype or the final system.

  The benefits of hardware-based fault injection are that experimental evaluation can accurately estimate the coverage and latency of actual hardware, as experiments are fast and can be run in real-time. Hardware fault injection can also inject faults with low perturbation. The drawback of hardware fault injection is the process of fault injection may introduce a high risk of damage to the target hardware.

Low portability and limited possible points to inject faults are also a significant drawback. The types of fault that can be injected are also limited. Lastly, hardware-based fault injection is affected by cases of limited observability, controllability and reproducibility.

- **Software-based fault injection**. In recent times, software faults are the leading cause of system failure. Fault injection is used to access the consequences of software faults such as hidden bugs. In software-based fault injection, the software that runs on the system under test is altered to enable the system designer to modify the system's state machine. The designer usually modifies the software of the system to inject a fault and to observe the effect. The software injection can be applied during compile-time or run-time. For the compile-time injection method, the faults are injected into the source code of the target program before the program image is loaded or executed. This method can emulate permanent faults since it is hardcoded. Software-based fault injection is considered a run-time injection method when a fault injection mechanism is triggered during the execution of the program.

  The advantage of software-based fault injection is that it can target the application and operating system. It often does not require any special purpose hardware, thus having low implementation cost. The disadvantages of this technique, however, is that it requires the modification of the source code to support fault injection. Therefore, the code that is used for the fault injection experiment is not the exact code that is deployed to the field. The consequence of such modification is that there is the possibility that the system will not behave the same and thus poses a question of fidelity for the software-based fault injection.

- **Simulation-based fault injection** requires the construction of a simulation model of the target system, including a detailed simulation model of the target system's processor. This method is favourable when the system is still under development, and a hardware prototype is not yet available. The use of Hardware Description Languages (HDL) often aids the design of simulation models. For example, the Very high speed integrated circuit - Hardware Description Language (VHDL) can be a preferable choice due to its recognition as a viable language for developing high-level models of the digital system and for driving test activities.

  The simulation-based fault injection has the advantage that it can support all system abstraction levels: axiomatic, empirical, and physical. It provides timely feedback to the system designer and is able to model both transient and permanent faults. A major drawback of this approach is that an enormous development effort is required to model an actual system. Besides the accuracy of the results depends on the quality of the developed model, i.e. how close is the model to the real system.

- **Emulation-based fault injection** exploits the use of Field Programmable Gate Arrays (FPGAs) to speed up fault occurrences inside circuit emulations. Emulation-based fault injection technique provides the designer with the ability to study the actual behaviour of the circuits in the application environment. The VHDL designs used in these techniques must be synthesisable. The benefit of this technique is the speed of injection time compared to simulation, where the drawbacks are increased development efforts and cost.

2.5 Concept of deep learning

2.5.1 Machine learning

Machine learning is a subset of a broader field known as Artificial Intelligence (AI). AI is a field that exploits ways to enable computers to mimic the cognitive capabilities of humans. Machine learning is a type of applied statistics with an increased emphasis on the use of computers to estimate complicated functions statistically. A machine learning algorithm is one that is able to learn from data. [Goodfellow et al., 2016, p. 98]. The meaning of learning as defined by [Mitchell et al., 1997], states that a computer is said to learn from experience with respect to some class of tasks and performance measures, if its performance at tasks in the class when measured by the performance measures, improves the experience.

The concept of learning defines the means to attain the ability to perform a specified task. However, the learning itself is not the task. For example, if we want a computer to learn to walk, the process of learning is not the task but walking. Machine learning is used to solve several complex tasks such as classification, classifications with missing inputs, regression, transcription, machine translation, anomaly detection, imputation of missing values, de-noising, density estimation and so on.

The ability of a machine learning algorithm is usually evaluated quantitatively. For example, the accuracy of a classification task can be measured directly from the proportion of correct classification achieved by the machine learning algorithm or by measuring the error rate.

The categorisation of machine learning can be carried out based on the kind of experience the learning algorithm has during its process execution. It can be categorised as ***supervised*** and ***unsupervised*** learning. Supervised learning algorithms experience a dataset containing features that are mapped to labels. Unsupervised learning algorithms experience datasets containing many features in which they learn useful properties of the dataset structure, and there is no labelling of datasets. In unsupervised learning, the algorithms attempt to implicitly or explicitly learn the probability distribution $\mathrm{P}(\mathbf{x})$ of observed random vector $\mathbf{x}$ or some interesting properties of $\mathrm{P}(\mathbf{x})$ of observed random vector $\mathbf{x}$. However, in supervised learning, the observed random vector $\mathbf{x}$ is associated with a value or vector $\mathbf{y}$. The vector $\mathbf{y}$ is predicted from $\mathbf{x}$ usually by estimating $\mathrm{P}(\mathbf{y}|\mathbf{x})$ [Goodfellow et al., 2016]. A key challenge in most traditional ML methods is **feature extraction**. Tasks such as object recognition require the programmer to develop algorithms for feature extraction. The success in performing such tasks are heavily reliant on the ability to extract features used for classification. However, the deep learning approach provides an automated way to accomplish feature extraction with little or no guidance from the programmer.

2.5.2 Deep learning

Deep learning is a subset of machine learning techniques that uses multiple layers on nonlinear data processing for supervised or unsupervised feature extraction and transformation [Deng et al., 2014]. Deep learning cuts across multiple research areas, including neural networks, artificial intelligence, graphical modelling, optimisation, pattern recognition, and signal processing. In this work, deep learning is classified into two major classes which reflect its usage. These classes include:

1 Deep networks for unsupervised and generative learning

2 Deep networks for supervised learning

In classification tasks, where information about the target class label is unavailable, unsupervised learning can be used to capture the high-order correlation of the observed data for pattern analysis or synthesis purposes. Networks that implement the unsupervised deep learning architecture include:

- Restricted Boltzmann Machine (RBM).
- Deep Belief networks (DBN).
- Autoencoders.
- Generative Adversarial Networks (GANs).

The description of the operation of these architectures can be found in [Wani et al., 2020]. The subsequent discussion will be limited to supervised learning since it is the applied architecture in this work. The supervised networks are also called the discriminative deep networks [Deng et al., 2014, p 214]. Supervised learning is used to provide discriminative power for pattern classification often by characterising the posterior distribution of classes conditioned for available data.

Supervised deep learning architectures have evolved over the past few years with increasing accuracy on many tasks. The Convolutional Neural Network (CNN), Deep Neural Network (DNN), and Recurrent Neural Network (RNN) are the most commonly used supervised deep learning architectures. CNN is a suitable choice for image data recognition and classification because of its high accuracy. The training of a CNN that consist of many layers is achieved using large sets of labelled data. Some examples of the supervised CNN architectures include AlexNet, GoogleNet, LeNet-5, ZFNet, VGGNet, DenseNet, CapsNet and ResNet [Wani et al., 2020]. The RNN is a suitable choice when dealing with sequential data such as text, audio and time-series. However, for tabular data, DNN is the most suitable choice.

Artificial neural network

Biological neurons inspired the modelling of an Artificial Neural Network (ANN). The model of artificial neurons is illustrated in Figure 2.5. It consists of the summation of weighted inputs that is passed through an activation function to produce an output. ANN was loosely inspired by the attempt to simulate the brain function of humans. The ANN concept stems from mimicking a neuron which has dendrites, a nucleus, axon and terminal axon as shown in figure 2.6. The neurons are usually connected via synapses; these synapses connect the dendrite to the terminal axon of another neuron as shown in the left of figure 2.6.

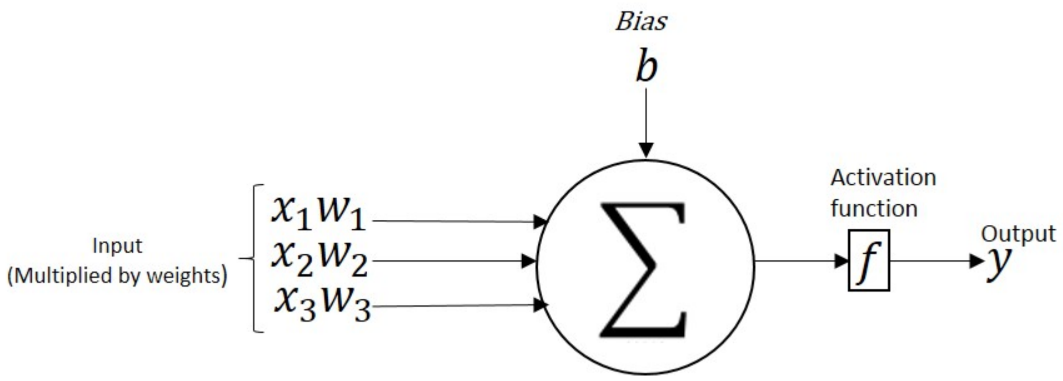


Figure 2.5: Model representation of a neural network.

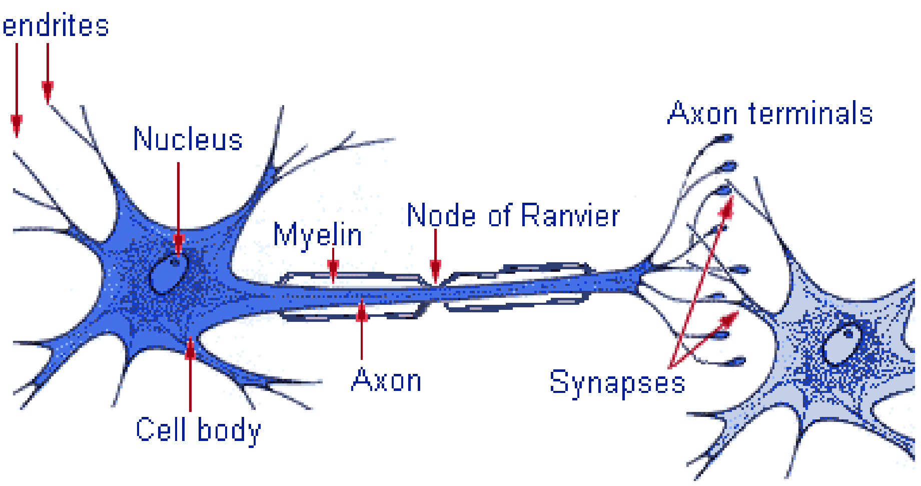


Figure 2.6: A network of two neurons [leavingbio, 2019].

Deep feed-forward network

Deep feed-forward networks are also referred to as Multilayer Perceptrons (MLPs) and are the quintessential deep learning models. MLPs have no feedback connections that feed the output of the model to itself. It is a network of functions that are described with a directed acyclic graph, in which the overall length of the chain of functions gives the depth of the model. The first function (layer) is termed the ***input layer***, while the final layer of the chain of functions is called the ***output layer***. All the layers in between the input and output layers are known as ***hidden layers***.

The feed-forward network can be described as a function approximation machine designed to achieve statistical generalisation. To achieve statistical generalisation, linear models such as logistic and linear regression are appealing. However, the model capacity of linear models is limited to linear functions. Therefore, the interactions between any two input variables cannot be exposed. Nevertheless, linear models can be extended to represent nonlinear functions by applying a nonlinear transformation to the input. The nonlinear transformation can be seen as a new representation of the input or could provide a set of features describing the input. Therefore, the linear model is not applied directly to the input, say $\boldsymbol{x}$, but a to a nonlinearly transformed input. The deep learning strategy is to learn the nonlinear transformation. Equation 2.3 illustrates the deep learning approach used to attain such a non-linear transformation.

$$y = f(x; \theta, w) = \phi(x; \theta)^T w. \tag{2.3}$$

The parameter θ in the equation is used to learn the nonlinear transformation from a broad class of functions, while $\boldsymbol{w}$ is used to map the nonlinearly transformed input to the desired output. This description is an example of a feed-forward network, where ϕ (nonlinear transformation) defines the hidden layer.

Chapter 3

Time-Triggered Ethernet communication

This chapter begins with a discussion of standard Ethernet, where the key characteristics of Ethernet are introduced, including the components of Ethernet. It then points out the desirable features for safety criticality applications such as determinism, bounded latency, and jitter of message transmission that are lacking in Ethernet. Section 3.2 introduces the time-triggered networking concepts in which clocks and the notion of a global time are discussed. Early deployment of Ethernet did not take into account determinism and safety-criticality requirements. The time-triggered networks address these requirements. This work considers two time-triggered Ethernet networks, TTEthernet and TSN, which extends classical Ethernet with additional services to meet the requirements of a fully deterministic communication and guarantee constant latency. Also, they provide both temporal and spatial isolation for its network traffic. TTEthernet and TSN are considered due to the increasing interest of embedded system designers to use these networks in applications with mixed-criticality, for both industrial and non-industrial use cases. The TTEthernet protocol is discussed in Section 3.3. Therein the TTEthernet frame structure, fault-tolerant clock synchronisation, services for startup and restart are addressed. Finally, section 3.4 discusses TSN with emphasis on the synchronisation protocol. The Precision Time Protocol (PTP) and the Generalised Precision Time Protocol (gPTP) are discussed.

3.1 Ethernet

Ethernet is the most widely used Local Area Networking (LAN) technology today. Ethernet's origin dates back to 1973 when it was introduced in a memo by Xerox Palo Alto Research Centre (PARC) [Spurgeon, 2000]. Ethernet was invented in an effort to improve the Aloha network. The Aloha network was an early experiment in the development of mechanisms for sharing common communication channels. The Aloha network suffered from message collisions.

Ethernet was thus developed to include a mechanism that detected when a collision occurred, and also to implement a "listen before talk" scheme. The "listen before talk" is described as carrier sensing, since stations listened for activity on the channel before transmitting. This is the idea behind the naming of the Ethernet channel access protocol as Carrier Sense Multiple Access with Collision Detection (CSMA/CD). More sophisticated backoff algorithms have since then been developed that allow Ethernet in combination with the CSMA/CD protocol to function more efficiently. The first version of Ethernet operated at 2.94 Mbps. However, this has been improved over the years to the following: 10 Mbit/, 100 Mbit/s, 1000Mbit/s and above.

Ethernet was first standardised in 1985 as IEEE 802.3, titled "Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification". There have been various versions over the years with major improvements and optimisations.

3.1.1 Open system interconnection layers

The Open System Interconnection (OSI) layer is a reference model developed by the International Organisation for Standardisation abbreviated as ISO. This model consists of seven layers. The OSI reference model is a method of describing how the interactions between sets of hardware and software can be organised in a communication network to work together.

There are seven OSI layers, namely, application, presentation, session, transport, network, data link, and physical layer. The physical layer describes the part of the networking standard that covers the electrical, mechanical, and functional control of data circuits which connects the physical medium [Spurgeon, 2000]. The data link layer is also known as the MAC layer. In this part of the standard, the Ethernet frame format and MAC protocols are defined. The network layer establishes higher-level functions and procedures for the exchange of data between *nodes* across multiple links. In this work, the term

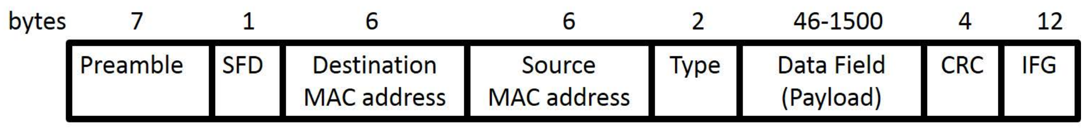


Figure 3.1: The Ethernet frame format

"**node**" is similar to the description in [Obermaisser, 2011, p. 6], it is a self-contained composite hardware/software subsystem that can interact with other nodes. The transport layer is used to provide end-to-end error recovery functions and flow control in the high-level networking software. The session layer provides mechanisms for establishing reliable communications between cooperating applications. The presentation layer provides the functions for dealing with how data is presented to the application. Finally, the application layer provides mechanisms to support end-user applications such as file transfers, mails and so on. The Ethernet standard is only applicable to the physical and data link layer.

3.1.2 Key characteristics of Ethernet

Ethernet system is made up of four building blocks that work together to provide the networking services. These building blocks include the following:

- Ethernet frame
- Media access control (MAC) protocol
- Signaling component
- Physical medium

Frame

An Ethernet frame is a standardised data unit used to exchange data between nodes. It is the data chunk which is sent over the Ethernet network. Figure 3.1 shows the frame format of the Ethernet.

The preamble is 7 bytes long and is used to align the physical signalling circuit on the sender and receiver. The preamble allows the participating nodes in a network to synchronise their receiver clocks. Each of the 7 bytes of the preamble is a repeated pattern of “10101010” bit. The preamble is followed by the Start Frame Delimiter (***SFD***) which is used to indicate the start of a frame. The SFD is 1 byte long and contains the bits: “10101011”. The following field is called the ***Destination MAC address*** field and it is 6 bytes long. It specifies the address of the receiver of a frame. The first 3 bytes in the *destination MAC address field* gives the Organisationally Unique Identifier (OUI), and the next 3 bytes specify the Network Interface Controller (NIC). The following field is the ***source address*** field which is the sender's address. The **Source MAC address** is 6 bytes long and has the same descriptions as the **destination MAC address**. The ***type*** field follows the source MAC address and it is 2 bytes long. The ***type*** field indicates the type of the client protocol or can be used to indicate the size of the Ethernet frame. The following field is the ***data*** field which carries the **payload** and ranges between 46 bytes to 1500 bytes. The actual message is encoded in this field. Payloads below 46 bytes are padded with zero bytes. The following field is called the **Cyclc Redundancy Check** (**CRC**) field or **Frame Check Sequence (FCS)** and it is 4 bytes long. This is used by the receiving node to check the integrity of frames during frame transmission. The last field is called the **Inter-Frame Gap** (**IFG**) field and consist of a minimum of 12 bytes. The **IFG** is used to establish the spacing between subsequent Ethernet frames.

Media access control (MAC) protocol

The MAC protocol defines a set of rules used to arbitrate access to the shared channel among a set of nodes connected to that channel. It allows a fair sharing of access for multiple nodes to a shared medium. There is no central controller, and each node is equipped to operate independently. An example operation delivered by the MAC is the broadcast delivery, where each frame transmitted is seen by every node on the network.

The MAC control mechanism uses the CSMA/CD protocol. The CSMA/CD protocol operates such that every node first listens to a quiet period on the channel before transmission. This action is called **carrier sense**. Every node is giving an equal chance when there is no transmitting node. This is described with the term **multiple access**. If it happens that two nodes transmit at the same time, the occurrence is detected, and both nodes stop transmitting. The detection is known as **collision detection**.

Ethernet signalling

Signalling is used to describe how bits are transmitted over the physical layer. Various encoding techniques are used for several Ethernet variants. There exist a variety of signalling techniques. For example, the 100BASE-TX uses the 4B5B MLT-3 coded signalling while the 1000BASE-T uses the PAM-5 coded signalling. Early Ethernet standards used Manchester coding to ensure self-locking signals that are not adversely affected by high pass filters.

Physical medium

The physical medium specifies the type of connectors and cables used to establish connections between nodes. It describes the electrical or optical properties of the physical connection between nodes in the network. The Ethernet links are usually a twisted-pair of cables for different network speed: 10/100 Mbps and 1000 Mbps (Gigabit) Ethernet. Ethernet can use either copper or fibre as the physical media. Some example media types include Copper:100 BaseTX-UTP Cat5 and the Fiber:100 BaseFX-Multimode/single mode.

3.1.3 Components of Ethernet

Ethernet consist of the following components: Controllers, hubs, switches and routers. The Ethernet controllers refer to the network interface card attached to a node that is used to send and receive frames. The controllers support layer 2 and other upper protocols such as the Transmission Control Protocol (TCP) [Blanton and Zimmermann, 2015], Simple Mail Transfer Protocol (SMTP) [Klensin et al., 2008], and HyperText Transfer Protocol (HTTP) [Fielding et al., 1999].

The Ethernet hub provides only layer two support. It is used to broadcast frames on all its port, and no frame content is modified. The collision domain is not reduced when using hubs as the traffic ends up in ports where it is not needed. Hubs are also known as **repeaters** because they just forward every message they get to all ports.

An Ethernet switches forwards and receives Ethernet frames. The switch supports layer two of the Ethernet. The support for layer three and upper protocols used for management are optional. Switches also do not modify frame content. However, switches learn the location of each node by reading the source address, which it uses to establish a forwarding table. Incoming frames are forwarded only to ports that connect to the correct destination node.

Routers also forward and receive frames. The frame headers are modified, and they support up to layer three and upper protocols. Routers perform similar operations with the switches except that they look at IP packets. IP packets are embedded into Ethernet frames and can be logically segmented into subnets. One major difference is that unlike switches, routers do not forward Ethernet broadcast frames. While switches reduce collision domains, routers reduce broadcast domains. The combined use of routers and switches is important for building a hierarchical, scalable network.

3.1.4 Absence of determinism in Ethernet

In bus-based Ethernet, there are no timing guarantees for the frame transmission, and the latency of frames is not bounded by the CSMA/CD mechanism. In switched Ethernet, collisions are avoided in point-to-point transmissions between switches and nodes. Nevertheless, traffic burst can cause congestion in switches and packet delays, buffer overflows and packet loss. The order in which messages are transmitted is also not maintained on reception.

Ethernet nodes have to regulate traffic to ensure determinism, operate a schedule for communicating, handle network failure and bound the latencies and jitters of messages. The switches have to support static routing and perform traffic enforcement schemes. The efforts to satisfy industrial requirement such as deterministic properties, fault tolerance, latency and jitter constraints, are the reason for the development of several protocols over Ethernet such as Ethernet/IP [Brooks, 2001], PROFINET [Guideline, 2003], MODBUS-TCP [Swales et al., 1999], SERCOS III [Bosch, 2010], EtherCAT [Park, 2002], Ethernet POWERLINK [POWERLINK, 2004], AFDX [Part, 2006], TTEthernet, and TSN. The protocols TTEthernet and TSN are within the scope of this work and are therefore discussed in section 3.3 and 3.4, respectively.

3.2 Time-triggered control

Cyber-physical systems often involve the interaction between sensors, actuators and embedded computers. The embedded computer interacts with a controlled object using sensors and actuators. Cyber-physical systems are often distributed and demand stringent real-time and safety requirements. Therefore, the underlying communication between controllers, sensors and actuators also have to satisfy these real-time and safety requirements. A time-triggered network provides promising features to satisfy the real-time and safety requirement for connecting cyber-physical nodes. The node consists of a host computer and a communication controller. The term “Cluster” refers to a set of nodes in conjunction with the interconnection medium between them. Figure 3.2 illustrates the structure of a time-triggered network. The nodes are connected by multiple **links** through **switches**. A Time-Division Multiple Access (TDMA) scheme is used to control the transmission of messages between nodes.

The TDMA scheme operates by granting nodes access to a network at **periodic** intervals. Therefore, the temporal allocation of communication resources on links to messages from individual nodes is according to a global time base. The schedule for frame transmission

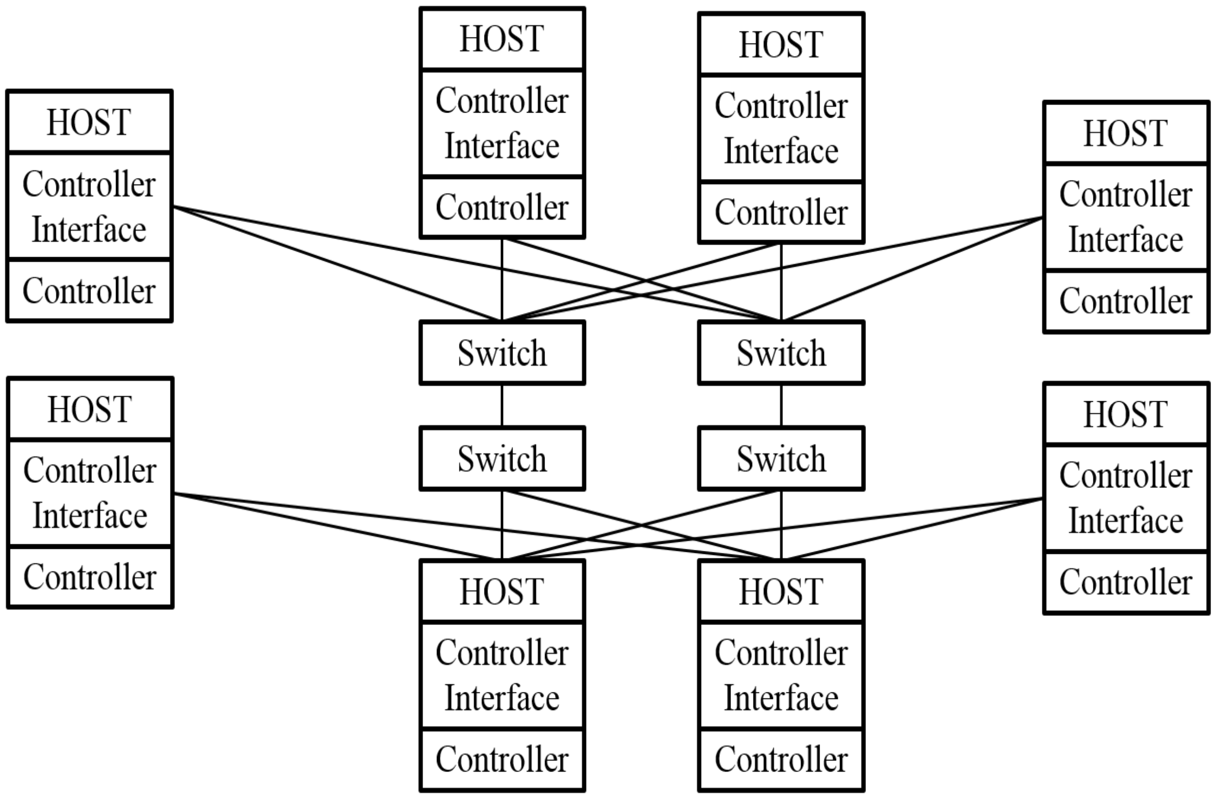


Figure 3.2: An example structure of the Time-triggered network

in a time-triggered network is carried out following a global time base.

3.2.1 Clock and global time

A digital clock is a device that measures time. A clock is realised with a counter that is increased in response to a physical oscillation mechanism that reacts periodically. The periodic event generated by the physical oscillation mechanism is called the **microtick**. The interval between two microticks is called the **granule**. The measurement of a clock's granularity can only be achieved with a clock of finer granularity. Let g^z represent the granularity of reference clock $\boldsymbol{z}$ used for measurement and g^k the granularity of clock $\boldsymbol{k}$. The granularity of the given clock $\boldsymbol{k}$ can be measured by counting the number of microticks of clock z between two microticks of clock k. Let this nominal number be represented as n^k.

The *drift* of clock k can then be derived from equation 3.1.

$$drift_i^k = z(microtick_{i+1}^k - microtick_i^k)/n^k \tag{3.1}$$

The drift rate is influenced by environmental conditions such as temperature changes, variations in voltage levels applied to a crystal oscillator or the wearing out of the crystals. Clocks that are never re-synchronised are called **free-running** clocks. Clock drift causes free-running clocks to deviate from a bounded relative time interval after a finite time. Therefore, there is a need for periodic synchronisation to ensure that clock drifts are corrected. The synchronisation of all clocks thus requires a global **global time** to serve as a reference.

3.2.2 Clock offset

The term offset denotes the time difference between the respective microticks of two given clocks [Obermaisser, 2011]. It is measured using the number of microticks of the reference clock. The offset between two clocks j and k measured by a reference clock z at its i^{th} microtick is defined in equation 3.2.

$$offset_i^{jk} = |z(microtick_i^j) - z(microtick_i^k)| \tag{3.2}$$

As mentioned earlier, the need for clock synchronisation is brought about by the drifting of clocks over time. Clock drifts have adverse effects on the performance of time-triggered networks. Other major causes of discrepancies between local clocks that communicate in a network include the following:

- Different clocks can have different starting points of microticks.
- Clocks could have different frequencies.
- Varying actual tick interval between clocks [Zhang et al., 2016].

Various services ensure that devices participating in distributed safety-criticality communication do meet their timing requirements, irrespective of the existence of clock drifts. Some of these synchronisation services include the Global positioning system (GPS), BeiDou (BD) system, Network Time Protocol (NTP), Inter-Range Instrumentation Group time code B (IRIG-B) and IEEE 1588. In [Yang et al., 2016], a comparison of these synchronisation services that considered accuracy, lock-time, cost, Ethernet support and reliability was carried out. IEEE 1588 did have dominant attractive features when it came to reliability, cost and support for Ethernet. However, due to the need for maintaining a tighter synchronisation, further development such as the TTEthernet synchronisation service has been realised afterwards. The TTEthernet synchronisation service was standardised as SAE AS6802 [AS6802, 2011]. It added merits to guarantee the precise moments for time-triggered message transmission and reception between network participants.

3.2.3 Time-Triggered system

All actions in a time-triggered system are triggered by the passage of time. Such actions include communication actions and application task. Optimal system performance is achieved by coordinating task and communication activities that are triggered by the passage of time. The two properties that characterise a time-triggered system are the global notion of time and the global schedule. In a distributed system, the global time is available to each node, and the interaction between nodes are in accordance with a global schedule. The global time and schedule are used to realise a system with timing guarantees, minimised jitter, and predictable real-time behaviour.

Protocol operation of a time-triggered system

In a time-triggered system, the action time for transmission and reception of frames is performed during known time slots that are reserved by the scheduler. The knowledge of the global time and scheduling scheme is resident inside the protocol controllers (MAC layer) and cannot be modified by the application CPU. During operation, each time-triggered node performs a timing check to account for clock drift and frame propagation delays.

3.3 TTEthernet system

TTEthernet is a fault-tolerant communication protocol that extends legacy Ethernet (IEEE 802.3) to provide reliability and determinism for safety criticality applications. The extension supports applications where temporal determinism and fault tolerance are required. TTEthernet delivers the platform for mixed-criticality applications such that simple data acquisition systems, multimedia systems and the safety-critical real-time system can all share the same hardware platform. TTEthernet uses a decentralised clock synchronisation mechanism to facilitate the transmission of time-triggered (TT) frames with low jitter. The TTEthernet standard shows the combined operational principle of a fault-tolerant clock synchronisation used for TT frame interaction, together with the ARINC 664p7 Rate-constrained (RC) frames [Part, 2006], and the legacy Ethernet frames which are known as best-effort (BE) frames.

TTEthernet consist of nodes, switches, and physical links that connect them, individually referred to herein as network participants. TTEthernet supports three traffic classes for frame transmission: TT, RC, and BE traffic. The term traffic class differentiates between TTEthernet communication modes. TTEthernet transmits TT traffic periodically at predefined times, referencing a synchronised global time base that is available to all network participants. The interval of silence between two TT messages is used for the transmission of RC and BE traffic. A TTEthernet distributed network controls the transmission instance of TT frames such that the transmissions occur during the active interval of a sparse time base [Kopetz et al., 2005]. Applications that require tight latency, jitter, and determinism use TT frames to fulfil such requirements.

TTEthernet provides bandwidth guarantees for RC traffic, including multicast capability. The RC messages are sent between the intervals of the silence of TT traffic in a way to realise a well-shaped dataflow. The difference between TT and RC traffic is that unlike TT traffic, TTEthernet does not transmit RC messages according to a system-wide synchronised time base. As a result, the network participants may send RC messages at the same point in time to the same receiver, consequently resulting in queueing at the network switches. Therefore, there is increased jitter compared to TT traffic, and the switches also require increased buffer space. TTEthernet switches require a conflict resolution strategy to handle contention between the different traffic classes and employ three methods, namely, preemption, timely block and shuffling.

- In preemption, if a high priority frame arrives a switch when a low priority frame is being transmitted, the low priority frame is halted. The switch then establishes the minimum silence and then transmits the high priority frame at a time slot specified a priori.
- In timely block, considering that the switch knows a priori the arrival time of every high priority frame, the switch will not forward any frame at these time slots, in order to ensure that the high priority frame is transmitted without delay.
- In shuffling, if a low priority frame is transmitted when a higher priority frame arrives, the low priority frame transmission is completed before transmitting the higher priority frame. Shuffling presents an optimal solution from a utilisation point of view since it does not truncate frames nor block the outgoing ports of low priority frames.

BE traffic is event-triggered traffic and conforms to the classic Ethernet implementation. SAE AS6802 arranges the different traffic types in order of decreasing priority as TT, RC and BE. All network participant must be synchronised in TTEthernet when configured to support TT traffic.

Depending on the involvement of each network participants, three roles are defined for each network participants, namely, synchronisation master (SM), synchronisation clients (SC) and compression master (CM). These roles describe the part played by each participant in the exchange of synchronisation frames known as protocol control frame (PCF).

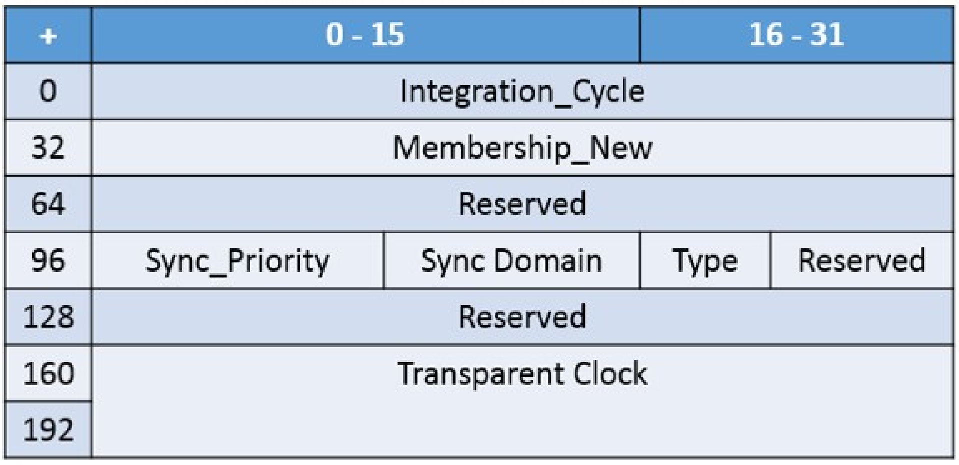
TTEthernet configures the end systems and switches as SM, SC or CM. TTEthenet performs a synchronisation algorithm in two steps [Obermaisser, 2011]. In the first step, end systems configured as SMs initiate the synchronisation algorithm by sending PCFs containing their local time to the compression masters (usually switches), and each compression master produces a new PCF called compressed PCF. In the second step, the compressed PCFs are sent by all CMs to SMs and SCs. SCs are only recipients of the compressed PCFs. SCs do not send PCF frames. TTEthernet can configure switches and end-systems as SCs. On completion of the synchronisation startup process, the system makes a global time available to all network participants. Therefore, it then creates a predetermined schedule for the transmission of TT messages.

TTEthernet implements the concept of virtual links for RC frames as described in ARINC 664p7. A virtual link is a unidirectional communication path between a sender and receiver(s). The message source is always from one sender to one or many receivers. The frame structure of legacy Ethernet is modified to support TTEthernet critical traffic. TTEthernet splits the destination address of Ethernet into two parts, namely, the constant field bits and the virtual link identifier (VLID). The first four bytes of the source address is referred to as the constant field while the last two bytes specify the VLID. The constant field bits are used to separate critical traffic (TT and RC) from the BE traffic. The VLID can be further used to separate TT from RC traffic.

3.3.1 TTEthernet frame structure

The frame structure of TTEthernet is compliant with the Ethernet frame format. However, some implementations of TTEthernet use the Ethertype field to distinguish the different traffic classes in TTEthernet. More efficient and faster implementations perform this distinction using the destination address field. TTEthernet encapsulates its synchronisation messages in the PCF frames. A PCF frame is also an Ethernet frame with a minimum payload of 46 octets. TTEthenet sets the Ethertype Field to 0x891d, to distinguish a PCF frame from other frame types. Figure 3.3 depicts the content of the PCF frame.

The TTEthernet uses the Integration_Cycle field to carry the number of the current integration cycle. TTEthernet represents time cyclically with a period called the cluster cycle. The cluster cycle is defined by the least common multiple of all time-triggered frames. The cluster cycle represents the full cycle through the schedule of TT frames [Obermaisser, 2011]. The system provides multiple synchronisation points between two TT periods. TTEthernet refers to these integration points as the integration cycles. The



+	0 - 15		16 - 31	
0	Integration_Cycle			
32	Membership_New			
64	Reserved			
96	Sync_Priority	Sync Domain	Type	Reserved
128	Reserved			
160	Transparent Clock			
192				

Figure 3.3: The PCF frame format

period for clock synchronisation is called the "integration cycle". TTEthernet tracks the participants in the synchronisation protocol using the membership_new field. TTEthernet uses the sync_priority field to provide hooks for system_of_systems through these priority settings. When there is more than one synchronisation domain, TTEthernet uses the sync_domain field to specify the domain which a participant belongs. The different types of PCF used in TTEthernet synchronisation protocols include ColdStart (CS) frame, Coldstart Acknowledgment (CA) frames and Integration (IN) frame. TTEthernet carries out its startup and restart process using the CS and CA. It then performs clock synchronisation with the IN frames.

3.3.2 Fault tolerant clock synchronization

TTEthernet utilises a transparent clock mechanism to ensure a detailed end-to-end latency computation for its synchronisation messages (PCFs). The transparent clock mechanism ensures that all devices measure and add their send delays, relay delays, and receive delays imposed on the PCF in the final delay computation. A high precision network requires several integration cycles within one cluster cycle. A TTEthernet device uses two variables to maintain its local view of the synchronised global time-base. These variable are called the local_clock and local_integration_cycle. TTEthernet uses the local_clock variable to count cyclically within an integration cycle and uses the local_integration_cycle to count the number of integration cycles. The synchronisation service executes at the beginning of each integration cycle using a two-step implementation as earlier mentioned.

In the first step, the network participants configured as SMs send PCFs to the participants configured as CMs. The CMs perform a convergence function to produce a compressed PCF. The compressed PCF is sent to all participants configured as SMs and SCs. The second step executes a function such that the SMs and SCs collect the compressed PCF from different CMs and performs a second convergence function.

First convergence function

It is the CMs that execute the first step convergence. The compression function runs unsynchronised to the synchronised global time. The SMs dispatch PCFs according to their local clocks, which is then received by a compression function that runs in the CMs. The compression function produces a new PCF which it sends to SMs as a response. This dispatch point in time from the CM is known as the compressed point in time [Obermaisser, 2011, p. 197]. The compression function starts to run at the reception of a PCF, and since the senders (SMs) of this PCF are unsynchronised, it is possible to have SMs that send their PCF too early or too late. If the PCFs arrive in the compression function too early or too late, the convergence computation could be performed for only a subset of PCFs in a given integration cycle. The CM derives the permanence point in time of the PCFs using the permanence function.

The permanence function is a method that uses the transparent clock mechanism to transform the dynamic network delay into a constant maximum delay. TTEtherent computes the permanence delay using equation 3.3.

$$permanence\_delay = max\_transmission\_delay - pcf\_transparent\_clock \tag{3.3}$$

The maximum transmission delay is a parameter computed offline. It defines the maximum one-way transmission delay of any PCF transmitted between any synchronisation participants. The pcf_transparent_clock is the value of a field in the PCF frame payload which stores the accumulated send, relay and receiving delays of the PCF. For PCFs in the same integration cycle, the CM uses the first recorded permanence point in time as a reference for successive PCFs. The CM records the relative offset of the successive PCFs to the first permanent point in time. The TTEthernet protocol accommodates situations where multiple PCFs having different integration cycles become permanent. When this occurs, a parallel compression function is executed. The collection phase of PCFs by the CM is regulated by the Observation Window (OW). The OW is the maximum deviation of two correct local clocks in the system as measured by a clock within the network [Obermaisser, 2011, p. 199]. The OW enables the computation of the compression function in a fault-tolerant manner. To tolerate a certain number of faulty synchronisation masters *f*, Equation 1 denotes the computation for the Maximum Observation Window (MaxOW).

$$MaxOW = (f + 1) * OW \tag{3.4}$$

.

The TTEthernet protocol computes the clock correction value using a variant of the fault-tolerant median. The use of this approach is to mitigate the impact of faulty SMs. Where $p_i, i \geq 1$ represents the permanence point in time. The variable i is an integer which represents the order for the permanence point in time with p_1 being the first permanence point in time. Table 1 illustrates the computation of the fault-tolerant median implemented in TTEthernet.

Permanence Point in time (p_i)	Correction Value
1	0
2	$\frac{p_2 - p_1}{2}$
3	$p_2 - p_1$
4	$\frac{((p_2 - p_1)(p_3 - p_1))}{2}$
5	$p_3 - p_1$
> 5	Average of $(k + 1)^{th}$ largest and $(k + 1)^{th}$ smallest inputs. Where K is the number of faulty SMs that has to be tolerated

Table 3.1: Variant fault tolerant median for computing correction value

After the collection phase, the CM waits for a certain time before dispatching the compressed PCF. This delay between the collection phase and the dispatching of the compressed PCF is known as the **delay phase**. The delay phase is computed from equation 3.5

$$delay\_phase\_duration = Cval + (k + 1) * OW - Cphase \tag{3.5}$$

Where ***Cval*** is the correction value, ***OW*** is the observation window, and ***Cphase*** is the collection phase duration. The variable ***k*** is the number of faulty SMs that have to be tolerated. At the compressed point in time, the CM generates the compressed new PCF with an integration cycle that conforms with the current integration cycle, assuming there is no parallel compression execution. The compressed PCF is sent to all SMs. When a central guardian function is employed, the compressed PCF can be delayed to support such a service.

Second convergence function

At the SM, the expected receive time of the compressed PCF is known as the scheduled_receive_pit, where ***pit*** refers to point in time. The scheduled_receive_pit is decided offline and is computed from equation 3.6

$$scheduled\_receive\_pit = dispatch\_pit + 2 \times MTD + CMD. \tag{3.6}$$

where MTD is the maximum_transmission_delay, and CMD is the compression master delay. The MTD represents the interval between the dispatch point in time from an SM when the local clock counter is equal to zero to the time when the PCF becomes permanent in the CM. It takes another MTD until the compressed PCF becomes permanent in the SMs. An interval around the scheduled_receive_pit is defined, to tolerate faulty components, digitalisation errors, and timing errors introduced by crossing clock domains. This interval to accommodate this variance is known as the acceptance window.

The SMs evaluate all PCFs that are within the schedule of the defined acceptance window. It evaluates multiple PCFs given, according to the number of CM per channel. The evaluation is done in three steps [Obermaisser, 2011, p. 201]:

- Per-channel selection
- Low-membership exclusion
- Clock-correction calculation

In per-channel selection, the SM selects one PCF per channel. It is the PCF with the highest number of bits set in the PCF membership new field that is selected. In case of an equal maximum number of bits set by multiple PCFs, the latest PCF with the maximum number of bits set in the PCF membership_new is selected. The SM removes PCFs with a relatively low number of bits in the PCF membership_new field. Finally, the SM calculates the clock correction. In the case of one best-channel PCF, the clock correction is the scheduled receive point in time minus the permanence point in time of the best-channel PCF. However, if two-best channel PCFs remain, then the clock correction value is the arithmetic mean of the difference between the schedule receive point in time and the permanence point in time of both values. [Obermaisser, 2011, p. 201].

3.3.3 TTEthernet startup and restart service

The startup and restart services define the algorithm to initially synchronise the TTEthernet network participants. Before synchronisation, the SM triggers a fault-tolerant handshake. The fault-tolerant handshake is a sequence of negotiation where each SM sends

CS frames to all CMs, which relays the frames back to all SMs on the network; The SMs subsequently respond by sending a CA frame. The SAE AS6802 standard specifies different state machines for implementing synchronisation on CM, SM or SC depending on the failure hypothesis considered. An SM can be configured to have a failure mode restricted to inconsistent-omission failure. In a case where the TTEthernet network tolerates the fault of a CM and an SM at the same time, it is considered as a high integrity configuration. When the TTEthernet network is configured to tolerate the failure of only SMs or CMs, then the term standard integrity is used. The difference is that for high integrity configurations, the SMs accepts all CS frames except the ones that originate from itself while in standard integrity configuration, the SMs accepts all CS frames. This is because, in the event of an arbitrary failure of an SM, the CS frame are not acknowledged in the high integrity configuration. Under normal operating conditions, the synchronisation participants are expected to be synchronised after a successful fault-tolerant handshake.

A detailed formal description of the state machines of TTEthernet startup is described in the TTEthernet standard AS6802. The TTEthernet synchronization protocol uses five variables: *local_timer*, *local_clock*, *local_integration_cycle*, *local_sync_membership* and *local_async_membership*. The *local_timer* is an unsynchronised timer and is used prior to synchronization to measure timeouts. The *local_clock* is a synchronized timer used to measure time instants relative to the current integration cycle [AS6802, 2011]. The *local_integration_cycle* is used to count the number of integration cycles. The *local_sync_membership* and *local_async_membership variable* are used mainly in the synchronous/asynchronous and relative clique detection functions. The *clique detection* together with the *integration*, *coldstart* and *restart* function are the four functions realized by the startup/restart service of the TTEthernet protocol.

Integration

TTEthernet devices differentiate between an unsynchronised and synchronised state. Figure 3.4 illustrates the state machine for an SM. A TTEthernet device first enters the integration state when powered-on. In Figure 3.4, it is illustrated as SM_INTEGRATE for an SM. The device first attempts to integrate into an already existing synchronisation if available. Usually, the newly connected SM waits for a duration of two integration cycles to be certain whether synchronisation already exists in the network. It will receive an integration frame from a CM if a synchronisation service is running. On reception of an integration frame, the integrating SM examines the number of bits set on the membership_new field of the frame. The weight of the value determines what mode the newly connected component would enter next. A sufficiently high number of bits in the membership_new field would cause the integrating component to join and start the execution

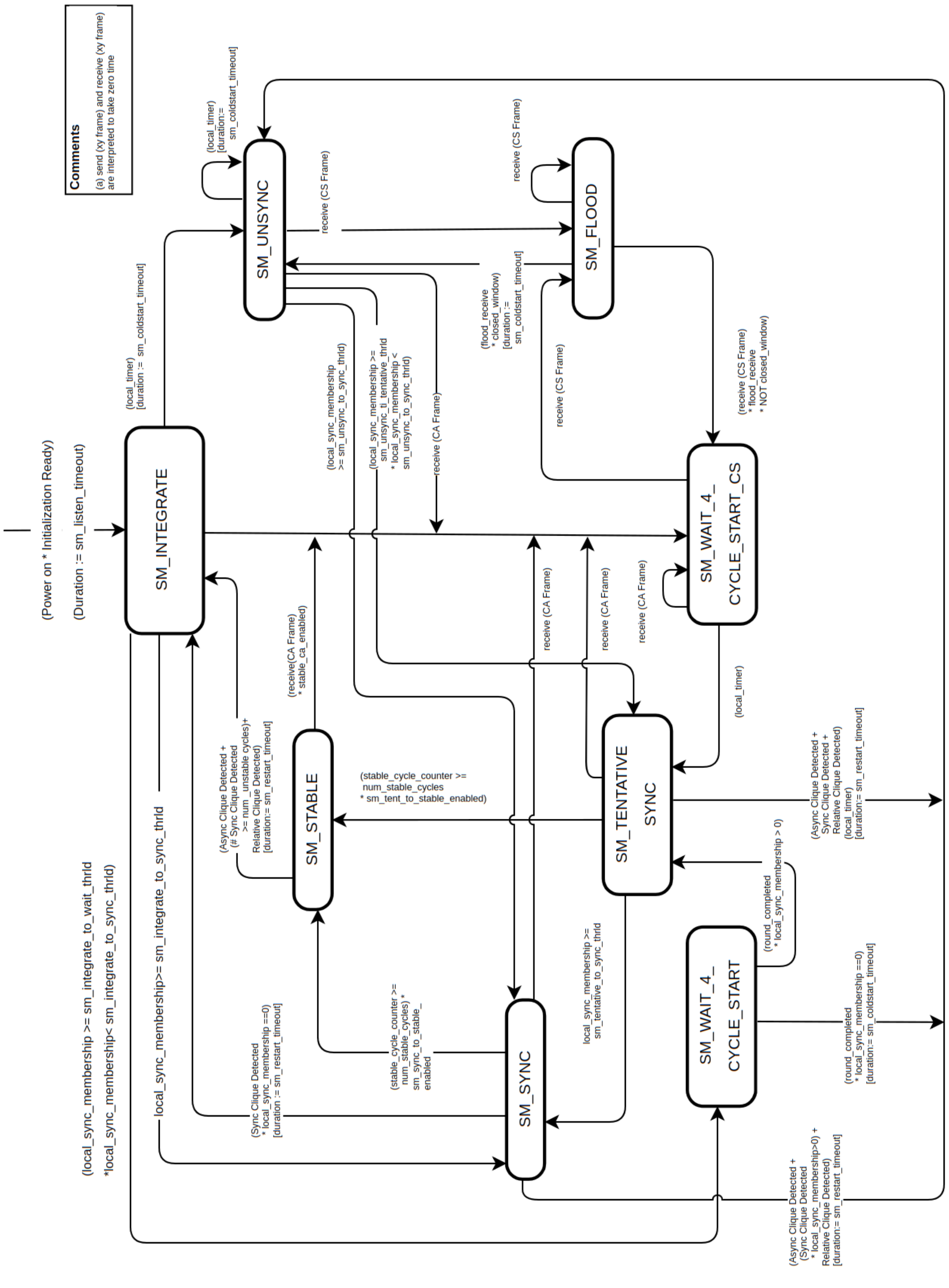

Comments
(a) send (xy frame) and receive (xy frame) are interpreted to take zero time
(Power on * Initialization Ready)
(Duration := sm_listen_timeout)
SM_INTEGRATE
SM_UNSYNC
SM_FLOOD
SM_WAIT_4_CYCLE_START_CS
SM_TENTATIVE SYNC
SM_STABLE
SM_SYNC
SM_WAIT_4_CYCLE_START
(local_timer) [duration:= sm_coldstart_timeout]
(local_timer) [duration := sm_coldstart_timeout]
receive (CS Frame)
(flood_receive * closed_window) [duration := sm_coldstart_timeout]
receive (CS Frame)
(receive (CS Frame) * flood_receive * NOT closed_window)
(local_sync_membership >= sm_unsync_to_sync_thrld)
(local_sync_membership >= sm_unsync_ti_tentative_thrld * local_sync_membership < sm_unsync_to_sync_thrld)
receive (CA Frame)
receive (CA Frame)
receive (CA Frame)
receive (CA Frame)
(receive(CA Frame) * stable_ca_enabled)
(local_timer)
(Async Clique Detected + (# Sync Clique Detected >= num_unstable cycles)+ Relative Clique Detected) [duration:= sm_restart_timeout]
(stable_cycle_counter >= num_stable_cycles * sm_tent_to_stable_enabled)
(Async Clique Detected + Sync Clique Detected + Relative Clique Detected) (local_timer) [duration:= sm_restart_timeout]
(round_completed * local_sync_membership > 0)
local_sync_membership >= sm_tentative_to_sync_thrld
(round_completed * local_sync_membership ==0) [duration:= sm_coldstart_timeout]
(stable_cycle_counter >= num_stable_cycles) * sm_sync_to_stable_enabled
(Sync Clique Detected * local_sync_membership ==0) [duration := sm_restart_timeout]
local_sync_membership>= sm_integrate_to_sync_thrld
(local_sync_membership >= sm_integrate_to_wait_thrld *local_sync_membership< sm_integrate_to_sync_thrld)
(Async Clique Detected + (Sync Clique Detected * local_sync_membership>0) + Relative Clique Detected) [duration:= sm_restart_timeout]

Figure 3.4: Synchronization Master state machine.

of the synchronisation service immediately. Otherwise, the integrating component enters the SM_UNSYNC state and executes the coldstart procedure.

Coldstart

The coldstart procedure is executed in the SM_UNSYNC and SM_FLOOD state. The SMs would send CS frames to all CMs. The CMs will then spread these CS frames to all SMs. As explained earlier in this section, high integrity configuration would only process CS frames from other SMs but not the ones sent by itself. The SM acknowledges the received CS frames by sending a CA frame. Finally, the CM again relays the CA to all SMs to complete the coldstart procedure. The SMs then transit to the SM_WAIT_4_CYCLE_START_CS state. After a defined offset allowing the received CA to become permanent, the SM_TENTATIVE_SYNC state is entered to test whether the normal operation mode can be entered. In this state, the SMs send IN frames to the CM; they are compressed and sent back to the SMs. When the SM receives back the IN frames, they examine the number of membership bits for sufficiency. A strong-synchronised operation mode is assumed when the number of bits is sufficient.

Clique detection

It is a strong requirement for startup algorithms to guarantee timely and safe startup [Steiner and Kopetz, 2006]. The safety property is focused on eliminating cliques during startup. Cliques are scenarios that emerge when a subset of integrating devices seems to communicate synchronously within the set but not with others. During synchronised operation, the TTEthernet synchronisation participant can be in either one of the three states, namely TENTATIVE_SYNC, SYNC and STABLE. A clique detection algorithm is utilised by the TTEthernet to detect all clique scenarios.

Restart

Restart is triggered when a component loses synchronisation. The clique detection mechanism is used to detect the synchronisation state of a TTEthernet component. When synchronisation is lost, the component attempts to regain synchronisation by re-executing the restart or coldstart mechanism which depends on the current state of the system.

3.4 Time sensitive networking

Time-Sensitive Networking (TSN) belongs to a group of IEEE 802.1 standards, which cover bridged networks and network management. TSN uses a Stream Reservation Protocol (SRP) for Audio Video Bridge (AVB) traffic bandwidth reservation. TSN can reserve up to 75% of the total bandwidth for AVB traffic in a port [Zhao et al., 2018]. TSN transmits the regular best effort (BE) frames within the unreserved bandwidth.

Furthermore, similar to the TTEthernet network, TSN uses a TDMA scheme that provides temporal partitioning on an Ethernet network. TSN utilises preconfigured time slots to ensure the deterministic transmission of messages awaiting transmission in IEEE 802.1Q [IEEE802.1Q, 2014] priority queues. TSN synchronises all its nodes which exchange TT traffic following a global time base. The synchronisation mechanism is covered in IEEE 802.1AS-rev, which is an ongoing amendment of IEEE 802.1AS-2011 [IEEE802.1AS, 2011].

In TSN, each switch port consists of 8 priority queues as specified in the IEEE 802.1Q standard. Priorities are assigned to TT, AVB and BE traffic. However, the time-aware shaper defined in IEEE 802.1Qbv [IEEE, 2016] provides the temporal partitioning support required for TT traffic. Table 3.2 shows the list of standards covered by the TSN task group. The updates to the TSN standard can be tracked in [Farkas et al., 2017]. The IEEE 802.1Qbv and IEEE 802.1AS-rev lay the foundation for the time-triggered communication paradigm, such that TSN uses to attain temporal and spatial isolation [Oliver et al., 2018].

IEEE 802.1Qbv provides the enhancement for scheduled traffic using the Time aware shaper (TAS). TAS is a mechanism that controls the opening and closing of gates that are associated with each priority queue defined in the IEEE 802.1Q. It uses a preconfigured cyclic schedule known as the gate control list (GCL) to control the state of each queues' gate, a gate of which can only be in one of two states (Open or Close). Frames are only available for forwarding when the gate associated with its queue is open. Figure 3.5 illustrates an example of the time-aware shaper. The time-aware is basically a gate mechanism that dynamically enables or disables the selection of frames that are queued up at the egress of a TSN device. The gate control list shown in Figure 3.5 has a predefined schedule given by the gate control entry (GCE) from *T00* to *T06*. Each GCE is opened for a time that corresponds to a pre-scheduled time window with a desired transmission selection algorithm (TSA) once the queue gate is opened. For instance, T00 uses an 8-bit binary system to hold the schedule "01001000" which translates to the time slot for the schedule T00, the queue for *traffic class number 6* and *traffic class number 3* are open, while other gates are closed.

However, in the example scenario shown in Figure 3.5, VLAN 60 and 70 are configured to convey TT messages and assigned priority class 6 and 7, respectively. AVB frames are routed to queues assigned priority 3, 4 and 5 on VLAN 30, VLAN 40 and VLAN 50 respectively. The GCL holds the status of the state for each queue that is assigned

IEEE Standard	Function Title
802.1Qbu-2016	Frame Preemption
802.1Qbv-2015	Enhancements for Scheduled Traffic
802.1Qca-2015	Path Control and Reservation
802.1AS-2011	Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks
802.1Qat-2010	Stream Reservation Protocol (SRP)
802.1Qav-2009	Forwarding and Queueing Enhancements for Time-Sensitive Streams
802.1BA-2009	Audio Video Bridging (AVB) Systems

Table 3.2: Time-Sensitive Networking Standards

priority 0 to 7 according to IEEE 802.1Q. The GCL synchronously changes according to predefined time intervals. The GCL status determines which queue is allowed to forward frames. Figure 3.5 shows the moment the gate of priority class 7 (TT) is open. In the current GCL state, all other gates are closed.

Since the TSN consist of several standards, all of which cannot be covered within the scope of this work, this work focuses on the standard which provides synchronisation for the distributed network.

3.4.1 Background on precision time protocol and profile

Precision time protocol

IEEE 1588-2008 Precision time protocol (PTP) synchronises the clocks of a distributed system. TSN makes it possible to synchronise the clocks of a sensor's local node, actuators, and other devices that operate in a distributed fashion using the Ethernet network. A primary objective of the PTP is to achieve a sub-microsecond synchronisation accuracy. A delay request-response mechanism is the bases of the PTP clock synchronisation. PTP measures the offset and delay between a master and a slave clock. It operates on packet-based networks that support multicast communications [Eidson, 2006]. There are five types of devices in the PTP; the *ordinary clock*, *boundary clock*, *end-to-end transparent clock*, *peer-to-peer transparent clock* and *management node*.

- The *Ordinary clocks* are end nodes that support PTP interactions. They are characterised by a single port which is used to communicate with the network via two logical interfaces for event messages and general messages. PTP defines two message classes, event messages, and general messages. Event messages require accurate timestamping at both transmission and reception, while the general messages do not require accurate timestamps.

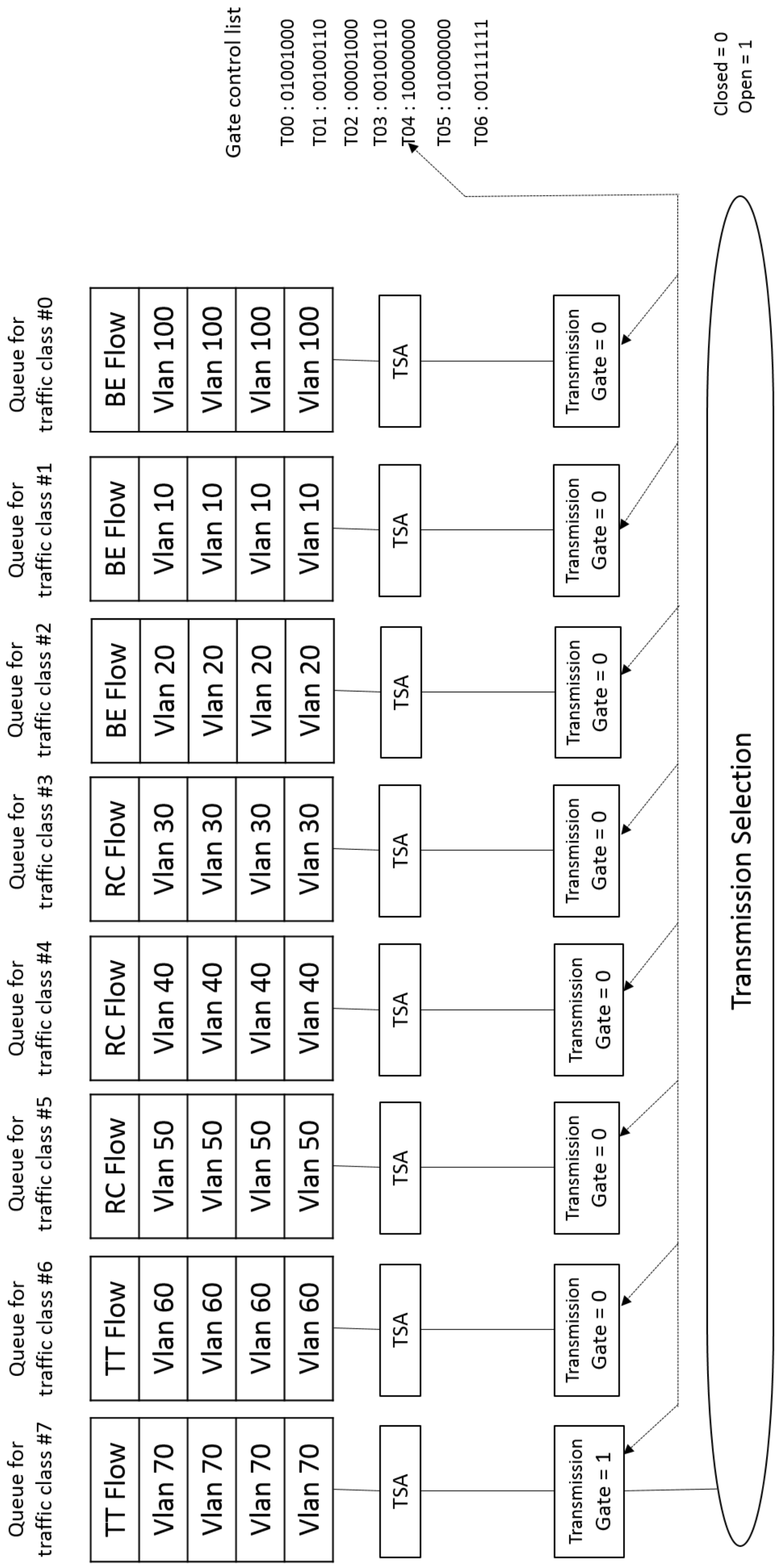


Figure 3.5: Illustration of IEEE 802.1Qbv transmission selection

- *Network devices* such as switches and routers that are equipped with PTP capabilities are usually used as boundary clocks. The boundary clock has multiple ports, and each of the ports is similar to that of an ordinary clock. The exceptions to these similarities are:

    1. The data sets and local clock are common to all ports of the boundary clock.

    2. The protocol engine in each of the ports resolves the state of all ports in the device to determine which port's reference time will be used to synchronise the common local clock.

- *The end-to-end transparent clock* is used to measure the residence time of event messages. The residence time is the time it takes the event messages to traverse the transparent clock.

- *The peer-to-peer transparent clock* is similar to the end-to-end transparent clock except that it corrects and handles event messages. It computes delays between link peers by exchanging Pdelay_Req, Pdelay_Resp, and possibly Pdelay_Resp_Follow _Up messages.

- *The management node* is used to configure and monitor the clocks during PTP operation.

PTP uses a two-step execution in its operation. Firstly it establishes a master-slave hierarchy, and secondly, it synchronises the local clocks. PTP consist of event messages and general messages.

The event messages consist of four message types, namely *Sync, Delay_Req, pDelay_Req, and pDelay_Resp.* The general messages consist of six types of messages, namely *Announce, Follow_Up, Delay_Resp, Pdelay_Resp_Follow_Up, Management, and Signaling.* IEEE 1588-2008 (PTP) uses two mechanisms to measure propagation delay, the *delay request-response* and the *peer delay mechanism.* The *Sync, Delay_Req, Follow_Up (optional), and Delay_Resp* messages are used by the delay request-response mechanism to synchronize an *ordinary* and *boundary* clock. While the *pDelay_Req, Pdelay_Resp and Pdelay_Resp_Follow_Up* are used by the peer delay mechanism to measure the link delay between two clock ports. PTP uses the *Announce* message to establish the synchronisation hierarchy. It uses the *management* messages to query and update the clock data set. For other purposes such as rate negotiation of unicast messages between clock nodes, PTP uses the *signaling* messages.

PTP permits a given communication network to host multiple independent synchronisation systems known as "subdomains". It logically configures the network into a tree structure to support the master-slave hierarchy before it starts operating. The "grandmaster clock" is at the root of this hierarchy. Every PTP node utilises the same timescale with the grandmaster clock. The ports of an ordinary clock or boundary clock execute an independent copy of the PTP protocol state machine. The states of each port are determined using an algorithm termed "best master clock algorithm" (BMCA). During the execution of the BMCA, the protocol analyses the contents of the announce message and the data sets associated with the PTP devices to resolve the state of each port. The possible port states are MASTER, SLAVE, and PASSIVE. A port in the MASTER state provides the time in which a port in the SLAVE state uses for synchronisation. A port in the PASSIVE state serves neither as a master nor a slave. The BMCA consists of two subset algorithms, namely the data set comparison and state decision algorithm. PTP uses the data set comparison algorithm to compare the attributes of clock pairs, after which it selects the best clock. The state decision algorithm is used to determine the next port state. It is only after the implementation of the BMCA that the clock synchronisation procedure is started.

Figure 3.6 shows a timing diagram of a PTP synchronization mechanism, illustrating the timeline of a *master* and *slave* clock. The master node sends a multicast *Sync* message at egress time *t1* recorded by the master clock. The slave clock receives the *Sync* message and assigns an ingress timestamp at time *t2*. Afterwards, the master node sends another multicast message (*Follow_Up*) to all slaves. The *Follow_Up* message is an optional configuration in PTP synchronization when one step-clock mode is not available [Liu and Yang, 2011]. If the *Follow_Up* message is configured, then the egress timestamp *t1* of the initial sync message is conveyed to the slave node through the *Follow_Up* message's data field. The slave nodes is then prompted to send a *Delay_Req* message to its master clock. The slave nodes records the *Delay_Req's* egress time as *t3*. The master nodes receives the *Delay_Req* message and records it at time *t4*. The timestamp *t4* is sent by the master immediately in the payload of a *Delay_Resp* message to the slave node. The slave node finally calculates the offset using *t1, t2, t3 and t4* with equation 3.7.

$$Offset = \frac{(T_2 - T_1) - (T_4 - T_3)}{2} \tag{3.7}$$

The slave node adjusts its local clock by adding the offset to its current node time using equation 3.8.

$$SynchronizedTime = CurrentNodetime + Offset \tag{3.8}$$

PTP operates under the assumption that the propagation delay in the network is sym-

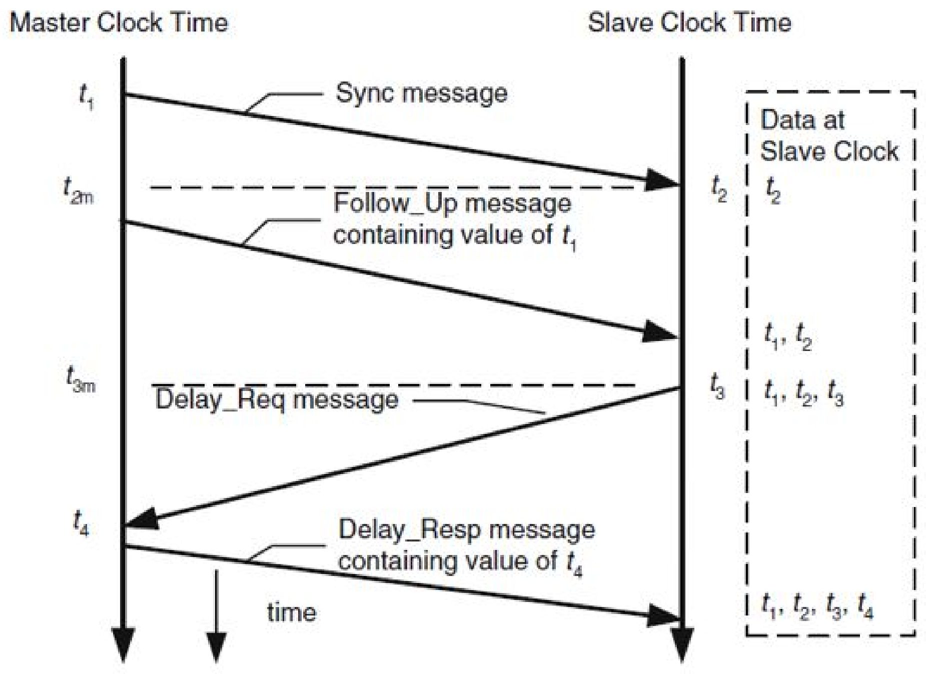


Figure 3.6: PTP timing diagram for synchronization message exchange

metrical. This assumption means that the message propagation delay from the master to slave is the same when perceived from slave to master. Figure 3.7 illustrates the link delay in clocks that support P2P path correction. This mechanism operates on port peers. The connected ports that share the same link carry out delay measurements for the link. The delay measurement allows clock corrections irrespective of the direction taken by *Sync* messages. In figure 3.7, port-1 sends a *Pdelay_Req* message noting the egress time stamp as *t1*. The *Pdelay_Req* message when received by port-2 is timestamped at *t2*. Afterwards, port-2 sends a *Pdelay_Resp* message, noting the egress timestamp as *t3*. The difference between the timestamp *t2* and *t3* is returned immediately by port-2 either in either a *Pdelay_Resp* message or the *Pdelay_Resp_Follow_Up* message or in both. Port-1 then records the ingress message at *t4*. The timestamps *t1, t2, t3 and t4* are then used to compute the mean link delay.

3.4.2 Generalized precision time protocol

A profile of PTP known as generalised PTP (gPTP) is standardised as IEEE 802.1AS-2011. The gPTP consist of two types of devices referred to as time-aware systems. These devices are the *time-aware end station* and *time-aware bridge*. Delay mechanism is defined in gPTP based on the media. However, this work focuses only on the IEEE 802.3 Ethernet full-duplex point-to-point links. Similar to PTP, the gPTP uses the BMCA in the selection

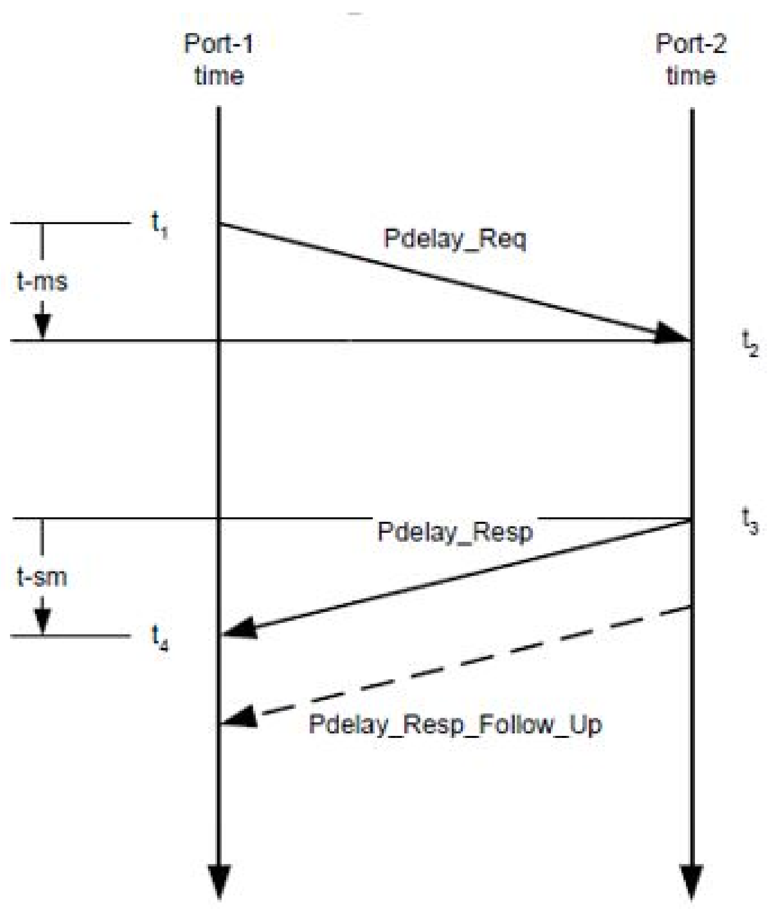


Figure 3.7: PTP link delay measurement

of the master clock. However, the BMCA implementation in gPTP is simplified. The major differences between gPTP and PTP are summarised in clause 7.5 of the IEEE 802.1AS-2011 standard.

Delay measurements take into account the *resident time* and *communication path delay.* The *resident time* describes the time taken by a time-aware bridge to forward a received message. The computation of*resident time* is performed internally by the time-aware bridges and are straightforward. However, The link delay is measured using the peer delay algorithm defined in PTP. The time synchronisation correction is dependent on the accuracy of the *link delay* and *resident time* measurements.

3.4.3 Start-up time

One of the objectives of PTP is the establishment of an automated mechanism and distributed approach that handles power-up, clock appearance, and disappearance, and change in clock functionality and system topology [Eidson, 2006]. The BMCA is used to achieve

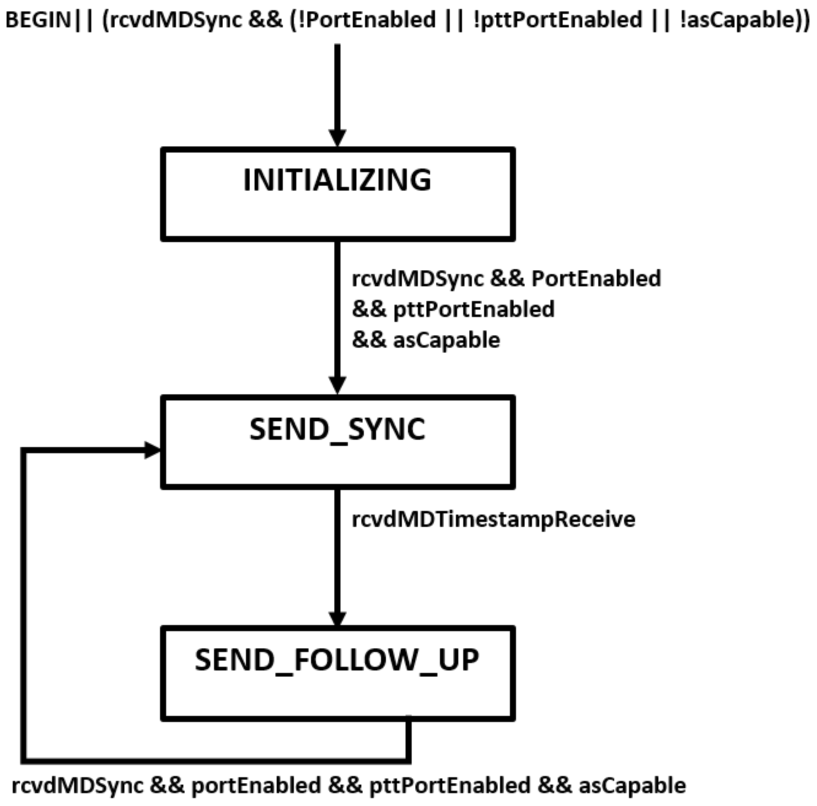

BEGIN|| (rcvdMDSync && (!PortEnabled || !pttPortEnabled || !asCapable))
INITIALIZING
rcvdMDSync && PortEnabled && pttPortEnabled && asCapable
SEND_SYNC
rcvdMDTimestampReceive
SEND_FOLLOW_UP
rcvdMDSync && portEnabled && pttPortEnabled && asCapable

Figure 3.8: MDSyncSendSM state machine

the PTP startup and reconfiguration. Each ordinary or boundary clock first listens for an announce message for a configurable time interval. The announce message contains information about the status and characterisation information of the transmitting node and its reference grandmaster clock. The clocks assume the role of a master when no announce message is received until a better clock appears.

However, on startup, several state machines are executed by each port of a gPTP device. Out of these state machines, only the state machines that impact the startup time and behaviour are depicted herein. These include the BMCA state machines, time-synchronisation state machines, and the state machine for peer-delay state mechanism. These state machines are described in clause 10 to 11 of the gPTP standard for a full-duplex point-to-point link [IEEE802.1AS, 2011]. Figure 3.8 to 3.10 presents an abstraction from these state machines and is used to illustrate how they impact the startup mechanism of gPTP. The variables affected by the startup process of the state machines are the rcvdSync, rcvdMDsync, portEnabled, pttPortEnabled and asCapable. The details of all other variables are explained in the gPTP standard. However, some variables illustrated in the figure 3.8 to 3.10 provides more clarity to the operation of gPTP.

The MDSyncSendSM state machine is illustrated in Figure 3.8. The MDSyncSendSM

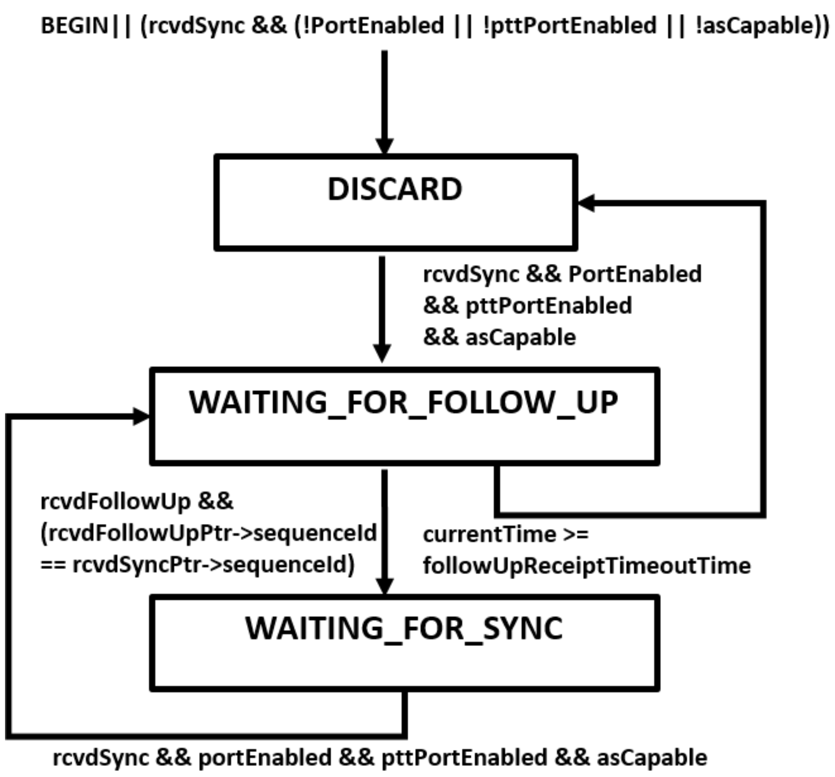

BEGIN|| (rcvdSync && (!PortEnabled || !pttPortEnabled || !asCapable))
DISCARD
rcvdSync && PortEnabled && pttPortEnabled && asCapable
WAITING_FOR_FOLLOW_UP
rcvdFollowUp && (rcvdFollowUpPtr->sequenceId == rcvdSyncPtr->sequenceId)
currentTime >= followUpReceiptTimeoutTime
WAITING_FOR_SYNC
rcvdSync && portEnabled && pttPortEnabled && asCapable

Figure 3.9: MDSyncReceiveSM state machine

receives an MDSyncSend structure from the *PortSync entity* of the same port, transmits synch messages and also computes the information needed for the Follow_Up message before transmission. PortSync entity computes port-specific delays needed for the time synchronisation correction. The global variables *portEnabled*, *pttPortEnabled rcvdMD-Sync* and *asCapable* are required during the INITILIAZING state to enable the system transition into the SEND_SYNC state and SEND_FOLLOW_UP state. The *portEnabled* is a Boolean that is set if the time-aware system's MAC Relay Entity and Spanning Tree Protocol (STP) entity can use the MAC Service provided by the Port's MAC entity to transmit and receive frames to and from the attached Local Area Network (LAN). The *pttPortEnabled* is also a boolean variable that is set to TRUE if the BMCA and the time-synchronisation function of the port are enabled. The *rcvdMDSync* is a boolean variable that notifies the current state machine when an MDSyncSend structure is received. The *rcvdMDSync* is reset by the current state machine. The *asCapable* is a Boolean that is set to TRUE if and only if it has been established that the ports of two connected time-aware bridges can inter-operate with each other via the IEEE 802.1AS protocol.

In Figure 3.9, the MDSyncReceiveSM receives Sync and Follow_Up messages, and sends information about time-synchronisation contained in the message to the PortSync entity of the same port. The rcvdSync, portEnabled, pttPortEnabled and asCapable are also

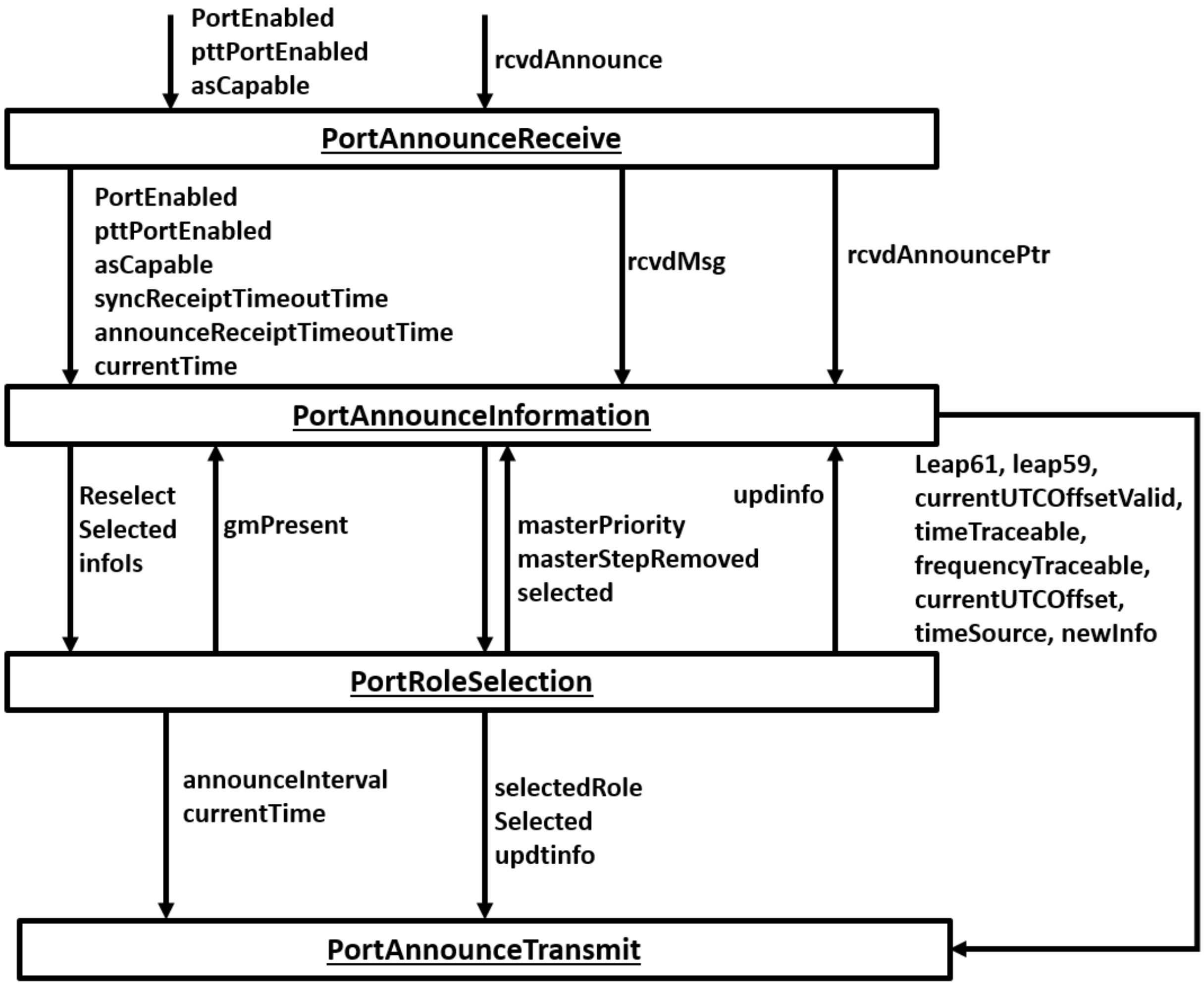


Figure 3.10: Best master clock algorithm state machines interelationships

required for a port that receives a synchronisation message to change into the WAITING_FOR_FOLLOW_UP state. Else the *WAITING_FOR_FOLLOW_UP* message will not be entered until the conditions are met. On gPTP startup, the variable portEnabled is TRUE only when conditions defined in clause 10, 10.2.4.11 of IEEE 802.1AS-2011 are met. When the time-synchronisation and BMCA functions are enabled, only then is the pttPortEnabled set. The rcdMDSync is a variable that notifies the current state machine when an MDSyncSend structure is received, while rcvdSync notifies the current state when a sync message is received. The time required to determine if a link is gPTP capable has a notable impact on the startup time of gPTP. The variable *asCapable* is used to indicate gPTP capable link. The variable*asCapable* is used to determine if each port connecting multiple time-aware systems can interoperate with each other via the IEEE 802.1AS protocol. It is noteworthy that the variable *asCapable* is used by the *PortSync* entity (shown in IEEE 802.1AS -2011 Clause 10, Figure 10-1). However, *asCapable* is set in the MDPdelayReq state machine. Therefore, the startup procedure of the other dependent state machines, including the BMCA state machine shown in Figure 3.10 are only completed when the *asCapable* variable is determined. Therefore, in some implementations, the variable *asCapable* is first determined before an announce and SYNC messages can appear on any link. The PortAnnounceReceive state shown in Figure 3.10 is invoked by the PortSync entity. The state receives and checks if announce messages are qualified. The PortAnnounceInformation announce state is used to compare if a newly received announce information is better than the current best master information. The PortRoleSelection determines and updates the port role for each port, while the PortAnnounceTransmit transmits announce information at specified announce interval.

Therefore, contrary to the startup time definition in popular opinions such as in [Diarra et al., 2015] which defines startup-time as the time between the transmission of the first Sync message by a grandmaster and the last arrival of a Pdelay_Resp message at an end device. This work defines the startup time as the elapsed time between the transmission of the first PDelay_Req message by any node and the arrival of the Follow Up message at the end device which received it last. In this way, the time taken for the determination of asCapable and other global variables (e.g. portEnabled, pttPortEnabled) is included. Another source of delay during startup is the spanning tree protocol (STP) protocol, but this work focuses only on the IEEE 802.1AS startup mechanism and does not include the STP protocol.

Chapter 4

Related Work

This chapter discusses existing fault injection frameworks and methods used in evaluating the dependability of a system. Firstly, the requirements of the proposed validation framework for TTEthernet and TSN are discussed in section 4.1. Afterwards, different fault injection tools are discussed in section 4.2, to portray how the existing state-of-art do not satisfy the requirements for the intended framework of this work. In the following section 4.3, the verification carried out for different MAC layer protocols is discussed. The discussions include the different methods of network verification, and practices such as peer review, simulation, formal methods and prototype testing. Section 4.4 discusses the application of techniques such as formal methods, simulation and prototype testing for the verification of different network protocols. The related works discussed include different network protocols such as Ethernet, Controller Area Network (CAN), FlexRay, Time-Triggered Protocol (TTP/C), and Time triggered CAN (TTCAN). Finally, section 4.5 summarises the related works, emphasises the gaps, and outlines the contribution of this work.

4.1 Requirement

The use of fault injection in evaluating the dependability of a system, be it hardware systems or software systems is not a new technique. The design of a fault injection framework is constrained by certain factors such as the target domain (hardware or software), level of intrusiveness, abstraction layer, and type of faults to be injected. Several fault injection frameworks and tools have been designed over the years targeting different platform, and these tools are discussed in section 4.2. Nevertheless, due to the diversity in target applications (both in software and hardware), the fault injection framework is usually designed for a specific target domain. Therefore, a fault injection framework that is designed to validate the dependability of a hardware target by injecting hardware faults may not be suitable for validating software target. The requirement of the desired fault injection framework in this work is that the fault injector must match the target domain, which is a network protocol that comprises the joint operation of both hardware and software devices. The mentioned domain requirement is considered in this work as the *requirement 1*, such that the failure modes listed in IEC 61508 are covered, and the fault injection framework is aimed at validating a distributed communication protocol (TTEthernet and TSN in focus).

This work is based on two time-triggered communication protocols, TTEthernet and TSN. These two protocols are appealing to applications with high dependability requirements. This works aims at developing a fault injection framework in which the target domain is the time-triggered communication protocol. Time-triggered protocols require specialised hardware devices and software implementation to operate. Therefore, verification and validation exercises for these networks using some existing state-of-art are not suitable for the target domain. The integrated system behaviour of the time-triggered protocols cannot be captured by evaluating only the hardware implementing implementation or only software implementation. Several fault tolerance mechanisms are often implemented for time-triggered protocols. For example, the fault-tolerance startup and restart service discussed in section 3.3.3 of the previous chapter 3. A fault injection framework that targets all the integrated behaviour of both hardware and software implementation of the fault-tolerant mechanism and behaviour under failure for TTEthernet/TSN physical setup remains insufficiently researched. As a consequence, no framework gives total coverage of the protocol state machine paths for all physical network participants. The requirement that refers to the integrated behaviour of all implemented TTEthernet and TSN is referred herein as *requirement 2*.

Existing industrial practices rely on verification and validation techniques carried out in simulation and mathematical models, and fault injection is often implemented in simulation. A significant challenge with these techniques is the gap that exists between abstraction layers. Notably, if models are classified in increasing order to close out missing abstraction details, one would see it as follows “Mathematical model - Simulation - Emulation - Prototype”. The verification and validation of prototypes provide a higher confidence level since it is the closest to the actual system to be deployed. Verification and validation of fault-tolerant properties and system behaviour under failure for TTEthernet and TSN physical setup fall within the scope of this work. The choice of hardware, operating system, and coding techniques used in the realisation of a prototype or final version plays a critical role in determining the behaviour of the systems. Therefore, it can be argued that it is not enough to conclude validation exercises in simulation or formal methods, thus the abstraction level requirement. This abstraction requirement is referred to herein as *requirement 3*.

As discussed in section 2.4.1, the introduction of a fault injection component into a NUT may leave a footprint that affects the system's functions, which should not be the case. Fault injection frameworks often present some level of intrusion where the framework itself affects the operation of the system under test, predominantly by adding delays in target domains that require message exchange. Low invasiveness is essential for time-triggered communication, where the correctness of the system depends on the timely transmission and reception of messages. The effect of the fault injection component must be limited to the faults it is intended to inject. It is necessary to avoid all side effects, for instance, unaccounted and inconsistent delay footprint by the fault injector. Therefore, overcoming this challenge is a critical requirement for designing a fault injection framework for TTEthernet and TSN. Low-intrusion and a masked delay footprint requirement of the fault injector component are referred herein as *requirement 4*.

This chapter aims to evaluate existing fault injection framework based on its suitability to cover the validation of the correctness of the fault-tolerance services in TTEthernet and TSN protocols, and the reliability evaluation of applications based on these protocols. Further requirements such as the abstraction of the fault injector from the target system (*requirement 5*), the portability of the fault injection framework across TTEthernet and TSN (*requirement 6*), form the basis of analysis for the state-of-art.

Abstraction here means the separation of the fault injection framework from the system being tested (network). Several fault injection framework that will be discussed do modify the components of the systems under test or NUT. For example, the works carried out by [Rodriguez-Navas et al., 2003] and [Ziermann et al., 2012], as will be discussed in this chapter performed such modification. The TTEthernet protocol and TSN protocols present auspicious safety features and are currently being implemented by several vendors. Therefore, such an abstraction feature is needed to enable the fault injection framework to evaluate multiple implementation and applications of the different vendors. The sfiCAN framework in [Gessner et al., 2014] is an example of where such abstraction is attained. The abstraction of the testing framework from the NUT would make it possible to test the fault injection implementation across the different applications that use TTEthernet and TSN. Such an application-independent framework makes it possible to inject faults into a variety of applications for dependability evaluation. Since the component being tested is a given network, the placement of the fault injection framework should be such that the components of the network (switches and nodes) are not modified. In regards to the portability requirement (*requirement 6*), the fault injection framework should not be vendor-specific. The view of the portability requirement is that the fault injection framework should be portable to different time-triggered protocols such as TTEthernet and TSN, as well as across different vendors.

Apart from the use of fault injection frameworks to carry out verification and validation exercises for fault-tolerant mechanisms, the data acquired from a given framework can also be learned to avoid failures. The adoption of machine learning techniques for learning large data statistics is currently a popular field. The application of deep learning to learn the profile of certain failure modes such as corruption and omission failures provide a promising diagnostic solution for detecting failures before the establishment of a global time-base in time-triggered networks. To detect failures earlier before the completion of the synchronisation startup process is quite a challenge since such a fault-tolerance mechanism would most likely rely on missed deadlines to identify omission or corruption failure. Deep learning can be used to train a model using the results of the fault injection framework at development time, and then deploying the model in the real system afterwards. Hence, the trained model would be useful at run-time. Therefore, the fault injection framework must be capable of generating data that can be used to train a neural network. The ability of the fault injection framework to generate data that can be used for training the neural network is referred to herein as *requirement 7*.

4.2 Fault injection tools

A detailed survey on fault injection tools was carried out in [Ziade et al., 2004]. Several fault injection tools are described for software, hardware, simulation, and hybrid based methods. These frameworks include the following below, arranged according to different target domains:

1. FERRARI (Fault and Error Automatic Real-Time Injection) [Kanawati et al., 1995], FTAPE (Fault Tolerance and Performance Evaluator) [Tsai and Iyer, 1995], FIAT (Fault Injection-based Automated Testing) [Segall et al., 1995], XCEPTION [Carreira et al., 1998], DOCTOR [Han et al., 1995], EXFI [Benso et al., 1998], NFTAPE [Stott et al., 2000], and GOOFI [Aidemark et al., 2001] used for software-based fault injection.

2. RIFLE [Madeira et al., 1994], FOCUS [Choi and Iyer, 1992], MESSALINE [Arlat et al., 1990], FIST (Fault Injection System for Study of Transient Fault Effect) [Gunneflo et al., 1989], and MARS(Maintainable Real-time System) [Karlsson et al., 1998] used for hardware-based fault injection.

3. VERIFY (VHDL-based Evaluation of Reliability by Injection Faults Efficiently) [Sieh et al., 1997], MEFISTO-C [Arlat et al., 2003], HEARTLESS [Rousselle et al., 2001], VFIT (VHDL simulation-based Fault Injection Tool) developed by the GSTF (Fault Tolerant Systems Group — Polytechnic University of Valencia) to run on a PC platform [Gil et al., 2003], and FTI (Fault Tolerance Insertion) [Entrena et al., 2001] used for simulation fault injection.

4. LIVE (Low-Intrusion Validation Environment) [Impagliazzo and Fabiomassimo, 2003] used in the hybrid fault injection

Ferrari uses a software trap and trap handling routine to inject faults into registers and memory locations to emulate data corruption. The execution state of a program is altered by a fault/error injection process running concurrently. The two processes run concurrently on the same machine; therefore, the fault injector component is not abstracted from the target device. In as much as Ferrari is a software-based fault injector, it does not cover the failure modes defined in IEC 61508; hence *requirement 1* is not met. The integrated behaviour of hardware and software is not within the scope of Ferrari and consequently does not satisfy *requirement 2*. Finally, there is no abstraction of the fault injection component from the target device; therefore, *requirement 5* is also not met.

FTAPE is a tool that integrates the injection of faults and the generation of the workload necessary to propagate those faults. The tool is composed of three main parts: FI (the fault injector), MEASURE, and WG (the workload generator). The FI performs fault injection, measurements of the current workload activity is performed by the MEASURE, while the WG creates workloads to propagate the injected faults. FTAPE targets CPU, local memory, and mirrored disk system [Tsai and Iyer, 1995]. Ferrari and FTAPE target the software running on a given hardware device. The context for controlling the injection of faults in a distributed system was not within the scope. FTAPE is not aimed for distributed network protocols. Thus, the FTAPE and Ferrari are not suitable for validating the total distributed nature of fault-tolerance mechanism for TTEthernet and TSN. FTAPE does not satisfy *requirement 1, requirement 2, requirement 5, requirement 6, and requirement 7.*

FIAT is designed for distributed systems and implements hardware structures composing of; Fault Injection REceptacles (FIRE) and the Fault Injection Manager (FIM). The FIM provides run-time control for the experiment and supports the data collection/analysis. The FIRE provides the execution platform for the distributed system under test in the experiment. Therefore, the framework is not abstracted from the target device, and hence *requirement 5* is not met. Requirement 1 is also not met since it does not cover the communication failure modes defined in IEC 61508.

The XCEPTION tool is a software-based fault injection tool consisting of three modules, namely the kernel, fault setup, and Experiment Manager Module (EMM). The scope of XCEPTION is for a target system composed by the processor, system buses and memory. XCEPTION is abstracted from the target application, but the implementation is on the same target system. Although XCEPTION is portable across different processors, it is not designed to target distributed communication systems. XCEPTION does not satisfy *requirement 1, requirement 5, requirement 6, and requirement 7.*

DOCTOR is a software fault injection environment which is capable of generating synthetic workload, injecting various types of faults including communication fault, and collecting performance and dependability data. DOCTOR supports three kinds of faults: memory faults, CPU faults, and communication faults. The fault injector component of DOCTOR consists of three modules: Experimental Generation Module (EGM), Experiment Control Module (ECM), and Fault Injection Agent (FIA). The FIA injects faults or causes a workload to wait/start/stop by receiving commands from the ECM via Ethernet. The EGM generate executable images of workloads that are downloaded to the target system. DOCTOR is designed for a distributed system, and its design separates the components of the fault injector from the target system, thereby, minimising the delay footprint that can be caused by the fault injector component. The communication failure realised in DOCTOR are classified in [Han et al., 1995] as message loss (omission), altered message (corruption), duplicated message, and delayed message. Although the failure modes compatible with DOCTOR are covered in the IEC 61508, the operating dynamics of TSN and TTEthernet communication protocol presents a different challenge. Both networks protocols consist of implementations of multiple complex state machines which can be impacted by the fault location. For example, as mentioned earlier, all the traffic classes in TTEthernet include, PCF, TT, RC, and BE. Validating the fault-tolerant startup protocol of TTEthernet would require targeting the PCF traffic class. The fault injection framework also needs to understand the dynamics of the target system to have the ability to satisfy *requirement 1* completely. DOCTOR does not satisfy *requirement 1, requirement 2, and requirement 6.*

EXFI is a software fault injection based on the trace exception mode available in most microprocessors. EXFI is divided into three modules, namely the Fault List Manager (FLM), Fault Injection Manager (FIM) and result analyser. The FLM generates the fault list to be injected into the system, the FIM injects the faults into the system, and the result analyser collects the results and produces a report concerning the whole fault injection experiment. EXFI is not designed for distributed systems and hence can not be used to validate the integrated system behaviour of TTEthernet and TSN. EXFI does not satisfy *requirement 1, requirement 2, and requirement 6.*

NFTAPE is a software fault injection tool for composing automated fault injection experiments from available lightweight fault injectors, triggers, monitors and other components. In NFTAPE, the fault injection component is replaced by a *LightWeight Fault Injector* (LWFI). The NFTAPE also handles logging, configuration, and communication functions. NFTAPE operates in a distributed environment and targets CPU registers, memory, applications and specific operating system functions. The framework is not suitable for TTEthernet and TSN protocol based on the same argument provided for DOCTOR in regards to the operating dynamics of the target system (TTEthernet and TSN protocol). Therefore, *requirement 1, requirement 2, and requirement 6* are not met.

GOOFI is a fault injection tool highly portable to different host platforms. GOOFI supports pre-runtime Software Implemented Fault Injection (SWIFI) and Scan-Chain Implemented Fault Injection (SCIFI). The SCIFI is used to inject faults through a built-in boundary scan-chain and internal scan-chain that is present in many modern VLSI circuits. In the pre-runtime SWIFI, faults are injected into the program and data areas of the target system before the program starts. Again, GOOFI is not sufficient to target distributed communication platform. Even though the fault injector component is abstracted from the target system, requirement 1 is not satisfied.

As mentioned in section 4.1, the two time-triggered communication protocols, TTEthernet and TSN require both hardware and software support. Unlike the above-listed frameworks which target either software or hardware, the target system in this work is the distributed network. Therefore, RIFLE, FOCUS, MESSALINE, FIST and MARS do not satisfy requirement 1. Similarly, VERIFY, VFIT, and FTI are simulation-based framework; the target domain requirement is also not met. The failure modes of the hybrid fault injector LIVE do not cover the failure modes given in IEC 61508. LIVE does not satisfy *requirement 1*, and *requirement 6*, and *requirement 7*.

4.3 Network verification methods

There exist several verification techniques to investigate the correctness of systems. These techniques include peer-reviewing, analysis using formal methods and testing on simulators, emulators and prototypes. Peer review involves the analysis of a system by an experienced professional. Most often, the reviewer checks the system (e.g. software code) manually for mistakes. Peer review is usually followed by an automated analysis which is extensively used in industry [Geilen, 2002].

Simulation can be useful in exposing the erroneous behaviour of a system. Simulation is scalable and can be used for testing, evaluation, and the initial validation of a protocol stack. A simulator can be designed for general-purpose or specific applications. The accuracy of the underlying model used in creating a simulator determines its fidelity. The more accurate a model attempts to represent a physical phenomenon, the more the complexity of the simulator increases — this increased complexity results in low execution speed of the simulator [Barnes, 2017].

Formal methods are computer techniques based on mathematical logic that is used to prove that a system complies with a set of properties [Qadir and Hasan, 2015]. Usually, the system to be verified and its interactions are modelled in a mathematical language to verify a set of properties on the model. The following are needed to accomplish formal verification [Mouradian, 2013]:

- Verification method
- Formal modelling language
- Modelling tool

A verification method is used to establish proof that the system functions correctly. The two main formal verification methods for network protocols are model-checking and network calculus [Barnes, 2017]. Model-checking is used to verify that a system (usually modelled as a finite state machine (FSM)) satisfies its specification. Formal methods sometimes express specifications as temporal logic which uses a set of defined formal algorithms to explore the states of the FSM exhaustively. Some example properties include accessibility, safety and deadlock. The main limitation of model checking is the state-space explosion problem. Consider a system that has M components and each of these components having N different internal states. The total number of possible system states is N^M. The exponential growth of the number of states with the number of system components and the size of the specification is called the state-space explosion effect [Geilen, 2002]. Network calculus is a theoretical environment that provides deep insight into flow problems encountered in networking. Network calculus provides the basis to analyse the fundamental properties of flow control, scheduling and buffers or delay dimensioning [Le Boudec and Thiran, 2001]. However, the limitation is the restriction of the calculus to performance bounds for a given topology.

The formal modelling language describes the behavioural rules of the system. Some examples of formal modelling language include petri nets [Murata, 1989], process algebra [Baeten, 2005] and automata [Fisher and Raney, 1969]. The modelling tool provides the platform to model the system in a given formal language.

A prototype is an early sample of a product built to test a concept. Test runs on prototypes are highly effective and result in more confidence level than other analysis techniques. However, the major setback is that it is time-consuming and expensive to build a prototype. Before a prototype can be realised, the design is almost ready. In particular, for safety-critical systems where the cost of failure or maintenance is very high, prototype evaluation is justifiable.

4.4 Related works on the verification of network protocols

Depending on the stage of a system development lifecycle, the techniques discussed in section 4.3 are adopted to verify network communication protocols. Other factors that could affect the choice of the verification technique include financial requirements, available expertise, and certification requirements.

[Revsbech et al., 2012] designed a testbed to validate the performance of Ethernet for an in-car network. The testbed comprises a specially designed field-programmable gate array (FPGA) based networking card, the NT4E 4-port adapter from Napatech [Napatech, 2017] for measurements. The purpose of developing the testbed was to obtain precise model parameters useful for designing an in-car network. However, the work did not perform any detailed analysis of the in-car network, since it was not within the scope.

Similarly, [Ziermann et al., 2012] evaluated the timing behaviour of the Controller Area Network (CAN) with the use of a testbed. The testbed approach was adopted to reveal problems that are not visible at the simulation level. The work used a Virtex-5 OpenSPARC evaluation platform to implement each CAN node and a standard PC for two purposes. The first purpose was to allow for debugging during development time. The second purpose was to gather performance measurements. The work used the FPGA for message generation, message transmission and performance evaluation. If different vendors would like to test their CAN implementations on this platform, it would require redesigning the applications running on the FPGA to conform to their requirement. It does not abstract the evaluation strategy from the components of the network under test (NUT). Therefore, *requirement 5* is not met.

Verification can ensure that specific dependability properties are met. In safety criticality systems, fault-tolerant mechanisms are used to ensure that a system continues to operate safely when an error occurs. The verification of fault-tolerance mechanisms is often used to provide arguments towards the level of confidence placed on a safety-criticality system.

An architecture for physical fault injection in CAN networks was presented in [Rodriguez-

Navas et al., 2003]. The work designed a physical fault injection tool that is capable of testing the fault tolerance mechanisms of the CAN protocol. The work considered two approaches to selecting the fault location. The fault location specifies where to inject physical faults. The first consideration was in the transmission medium, and the second was between a CAN controller and its transceiver. [Rodriguez-Navas et al., 2003] implemented the latter, which meant that each node was modified with an individual fault injector (IFI). The challenge with this approach is that node modification was required to achieve physical fault injection. If this approach is to be adopted, it would require modifying each node to contain the IFI before performing validation exercises. Again, *requirement 5* is not met as there is no abstraction of the fault injection framework from the NUT.

[Lanigan et al., 2010] employed a software-based method to inject faults into AUTOSAR applications. By leveraging the CANoe simulation environment, the framework used two types of hooks (suppression and manipulation), inserted into the AUTOSAR codebase. A CANoe is a software tool used to develop, test and analyse electronic control units (ECUs). It is developed by Vector Informatik [Vector-Informatik, 1996]. The principal drawback of this approach as described in [Lanigan et al., 2010], is that certain faults such as modification of data buffers or attempting to cause timing violations causes a simulation-wide probe effect crashing the entire simulator. Besides, simulators are modelled to mimic certain functionality of the real-world system, and the fidelity of the simulator is determined by the underlying model used for the simulated design. Therefore, certain complex systems or complex functionality of some systems may not be sufficiently represented by a simulator. The *requirement 3* is not met since the target domain is not based on the actual system but simulation.

A Star-Based Physical Fault-Injection Infrastructure for CAN Networks (sfiCAN) was developed in [Gessner et al., 2014]. The work relied on a CAN-compliant hub, which is connected in a star-based topology to several CAN nodes. The hub described is equivalent to a CAN bus, and the framework achieves fault injection without the need to make any modification on the CAN node. It allows testing the behaviour of the software under failure without the need to modify the software being tested or the host computer of the software. The framework has the limitation that it requires the inclusion of extra COTS transceivers per node. However, its attractive feature is that it abstracts the testing component from the NUT similar to the fault injection framework implemented in this work, even though the framework is restricted to the CAN network. The sfiCAN is not portable to TTEthernet and TSN. Therefore, *requirement 6* is not satisfied.

[Kim et al., 2008] developed a system model and verification for FlexRay communication using systemC. The analysis employed the FlexRay Specification and Description Language (SDL) description. Every module was tested by test signals which are expected to be entered as the state of the communication controller changes. The drawback is that the verification is carried out on abstract models of the real systems. Therefore, the high complexity of certain network topology can be difficult to model; *requirement 3* is not satisfied. Secondly, the work did not consider verification under faulty scenarios.

Verification of FlexRay communication was also carried out through behavioural simulation in [Muller and Valle, 2010]. The work compared measurements from a modelled FlexRay physical layer transceiver, which was developed in VHDL-AMS hardware description language [Christen and Bakalar, 1999], to measurements from actual devices. The aim was to improve the confidence of a tuned transceiver model. Apart from simulation-based approaches for FlexRay, verification by analysis was also carried out in [d. Souto et al., 2016]. The work exploited an analytical model based on discrete-time Markov chains to evaluate the broadcast protocol for FlexRay. The work considered permanent, transient, omission, and asymmetric faults that affect both nodes and channels.

Furthermore, dependability evaluation of Time-Triggered Protocol (TTP/C) was carried out in [Racek et al., 2012]. Fault injection was used to evaluate the different fault-tolerant hypothesis of the TTP/C protocol. The work used a generic C-language to design the simulation models. *Requirement 3* is not satisfied with these approaches.

Several works verifying the startup algorithms in other time-triggered platforms such as TTA, FlexRay and TTCAN using formal methods were surveyed in [Saha et al., 2016]. The work gives a detailed overview on the verification approach adopted, based on the works carried out in [Dutertre and Sorea, 2004], [Steiner et al., 2004], [Saha et al., 2007a] for TTA, in [Steiner, 2005] for FlexRay, and in [Saha et al., 2007b] for TTCAN. The Symbolic Analysis Laboratory (SAL) toolset [de Moura et al., 2004], a model checking tool was predominantly used for the verification of the startup algorithms in these protocols. Nevertheless, *requirement 3* is not satisfied since the target domain was not based on the physical setup of the networks.

4.4.1 Verification and validation of TTEthernet

Formal methods

[Steiner and Dutertre, 2010] verified the TTEthernet's compression function using a SAL model checker known as the ***sal-inf-bmc***. The work formally verified several properties of different characteristics (e.g *membership and clock synchronisation*) and discussed their computational overhead. The assessment carried out in this work, allowed the addition of a configurable number of faulty dispatch processes. [Steiner and Dutertre, 2010] added a dedicated error-state to investigate the termination property of the compression function.

The SAL model checker was later extended by the same authors to present an automated proof of the full TTEthernet clock-synchronisation algorithm [Steiner and Dutertre, 2011]. The major highlights of this work are the use of a model of continuous uninterpreted time and the proof that the fault-tolerant clock synchronisation can be fully automatised.

[Ammar and Mohamed, 2011] used another formal model known as the PRISM model checker to verify the time-triggered Ethernet. PRISM is a probabilistic model checker, a tool for formal modelling and analysis of systems that exhibit probabilistic behaviour [Kwiatkowska et al., 2002]. [Ammar and Mohamed, 2011] introduced hardware faults in a probabilistic manner, and the PRISM was able to detect a state in which some TTEthernet properties are violated. The violated properties were the properties which stated the following:

- At any given point in time, there's only one node sending a message.
- A node only sends on its time partition.

The faulty states were reached when a switch and a node fail at the same time. [Ammar and Mohamed, 2011] describe the scenario in which this violation occurs in detail.

The verification activities mentioned above did not target the integrated system behaviour of all TTEthernet components, and thus *requirement 2* is not satisfied. In an attempt to capture the integrated behaviour, [Dutertre et al., 2012] used an automated test-generation tool to drive high-coverage testing of prototype TTEthernet hardware, based on a state-machine model of the TTEthernet protocol. Again the work was built on the existing SAL formalisation discussed in [Steiner and Dutertre, 2010] and [Steiner and Dutertre, 2011]. [Steiner and Dutertre, 2013] summarises the formal analysis and verification activities of the TTEthernet synchronisation protocol based on the SAL. *Requirement 3* is not satisfied with formal methods.

Simulation approach

[Abuteir and Obermaisser, 2013] developed a simulation framework for TTEthernet that supports the validation of TTEtherent-based applications in its early development stage. The simulation framework was implemented using the OPNET tool suit for discrete event simulations of communication networks [Riverbed-Technology, 2019]. Generic building blocks for TTEthernet switches, end systems (nodes) and fault injectors were developed. The building blocks can be instantiated in OPNET and configured to represent distributed embedded applications. The simulation framework extended the standard Ethernet switch and end nodes by adding deterministic frame transfer capabilities while retaining full compatibility with the requirements of IEEE 802.3. The end nodes can exchange the three traffic classes of TTEthernet frame: TT, RC and BE. The implemented fault injection block covers the following failure modes: Omission, corruption, link, crash, delay, stuck-at, babbling idiot, and masquerading failure.

Further work on simulation-based fault injection was carried in out [Fejoz et al., 2018]. Therein, a combination of a simulation model and experimental platform were used to gain quantified insights on Time-Triggered Ethernet's operation. Confidence in the resulting simulation model was gained by cross-verification with traces monitored on a real network. [Fejoz et al., 2018] used the CPAL [Navet and Fejoz, 2016] modelling and simulation environment. The authors performed cross-verification for the model validation with the following.

- A simulation model.
- Measurements taken on a physical experimental platform.
- results obtained from Formal methods.

A simulation assessment of TTEthernet startup time was carried out in [Sandic et al., 2018] using the OMNeT++ [Varga, 2010] simulator. [Sandic et al., 2018] simulated a single-fault tolerant configuration of TTEthernet, where a single switch was faulty. Several anomalies of the PCF frames were simulated, in which coldstart frames (CS), coldstart acknowledgement frames (CA) and Integration frames (IN) were sent at critical moments from the faulty switch. The experiment was carried out to analyse several anomalies that can negatively impact the startup time of the TTEthernet network.

[Li et al., 2018] modelled the TTEthernet Startup service using **SystemC**. The work focused on verifying the TTEthernet's fault-tolerant protocol under fail-omission scenarios. SystemC was used to establish an executable model of the synchronisation master and compression master, and also to design the corresponding testbench. The testbench was set to form different external stimuli for the executable models.

Simulators are often modelled to mimic certain functionality of a real-world system. For example, the simulation works in [Sandic et al., 2018], and [Li et al., 2018] focused on the synchronisation service of TTEthernet. Nevertheless, the fidelity of the simulator (OMNeT++, OPNET, SystemC, and CPAL) is always determined by the underlying model. Certain complex functionality may not be sufficiently represented. Confidence in the resulting simulation model can only be indeed gained by cross-verification with traces monitored on the physical counterpart network. Hence, *requirement 3* is again not satisfied with simulators.

Prototype testing

Performance analysis of Time-triggered Ethernet-Networks was carried out in [Bartols et al., 2011]. A low cost and lightweight approach were proposed to measure the end-to-end latency of time-triggered Ethernet traffic using COTS components. The work further presented a validation activity with an Ethernet performance analyser and a mathematical framework to check results obtained. The contribution of the work was a lightweight analyser that supports time measurements in the range of microseconds. This was accomplished on an embedded PC and a Linux OS with RT Kernel patch. The analyser supports specifically the Linux TTEthernet protocol stack. Although TTEthernet can be implemented in software, the realisation of the TTEthernet on layer 2, provides by far less end-to-end latency and jitter when compared to a software implementation. For this reason, most safety-critical applications opt to implement TTEthernet services in the MAC layer. The performance measurement approach proposed in [Bartols et al., 2011] is only supported for protocol-specific components residing between the application and the network driver.

The model-based analysis of TTEthernet mentioned in section 4.4.1 [Dutertre et al., 2012] was used to drive high-coverage testing of prototype TTEthernet hardware, based on the state-machine model of the TTEthernet protocol. The work explored the integrated system behaviour of connected TTEthernet components using a network integration laboratory (NIL). The NIL testbed consisted of more than 25 end systems and 17 switches that are instrumented for fault injection. The NIL-based testing did not target protocol branch coverage but emphasised high-level system properties. The scope of the work did not define the portability to TSN, and the sufficiency of the data to train a neural network for startup failure diagnostics was not investigated. Hence, *requirement 6 and requirement 7* are not satisfied with this method.

4.4.2 Verification and validation of TSN

[Pahlevan and Obermaisser, 2018a] designed an OPNET simulation framework for simulating TSN time-based features. The framework implemented the ingress time-based policing and enhancements for scheduled traffic as an extension of the Ethernet standard. The framework provided a modular implementation of the time-aware shaper and policer. The TSN model used the standard MAC unit for switching messages but adds the necessary functionality to support strict temporal requirements. The implementation was designed in such a way that it can be integrated into different vendor-specific network elements. A fault injection mechanism was later developed to evaluate the reliability of TT communication using the designed OPNET simulation platform [Pahlevan and Obermaisser, 2018b]. The requirement for a higher abstraction layer than simulation (*requirement 3*) is not satisfied.

4.5 Summary of related works

The argument presented herein is that when simulation and mathematical methods are used for dependability evaluation, there is a gap between the designed models and the actual system. It is a challenge to model all the properties of a real system. Usually, only certain properties of the whole system are modelled, and how accurate these models are, determines the authenticity of the simulation/mathematical approach. Therefore, for formal methods, the syntactic and semantic gap between the verified model and the real system pose a significant disadvantage. As for simulation methods, the fidelity of an experiment is much dependent on the method used in generating input parameters for the test, whether it is done randomly or with the use of some form of theory.

This work presents TRAITOR, a cut-through fault injection framework suitable for the dependability evaluation of the TSN and TTEthernet protocol by physical fault injection. TRAITOR is designed to target real systems or prototypes that implement these protocols. TRAITOR has the potential to reveal problems that are not directly visible in the simulation/analytical methods. Testing on a real system or prototype provides more accurate and trusted results than simulation and analytical methods. Similar to the sfiCAN framework described, the TRAITOR abstracts the fault injector component from the network participants (e.g. Nodes and Switches). A difference is that sfiCAN is restricted to CAN networks while the fault injection framework herein is developed for TTEthernet and TSN. The sfiCAN reference its applicability to a star topology, whereas the framework designed in this work, is topology independent.

[Revsbech et al., 2012] and [Ziermann et al., 2012] did not consider validations under failure scenarios. In the design of safety criticality systems, it is unarguable needful and a strong practice to consider failure scenarios when validating a system — the work in [Ziermann et al., 2012] modified the network under test. The node modification has an impact in the portability of the framework. The need for node modification was also done in the work of [Rodriguez-Navas et al., 2003]. In [Lanigan et al., 2010] due to the simulation approach, some complex functionality are not sufficiently represented, which is the same case in [Kim et al., 2008].

[ISO, 2011] and [IEC61508, 2010] recommend fault injection as a validation technique.

Until this day, there is no known physical fault injection framework designed for the dependability evaluation of both TSN and TTEthernet, such that the fault injector is abstracted from the components of the NUT. In addition, most works that analyse network protocols rely on observing three test criteria, namely latency, jitter, and checks for drop packets. However, efforts to observe and make visible the behaviour of a system under failure scenarios using deep learning techniques, particularly during the startup process, have not been fully exploited in evaluating the TTEthernet and TSN. During the startup process of TTEthernet and TSN, before the global time is established, it is ineffective to implement a fault tolerance mechanism that is based on the global schedule. Deep learning provides a promising way to automatically classify the behaviour of the synchronisation transaction during startup under given failure scenarios. Techniques that employ deep learning to capture the behaviour of TSN and TTEthernet during startup have not been extensively researched.

This work presents the design, implementation and testing of the physical fault injection framework (TRAITOR) for TSN and TTEthernet network equipped with deep learning capabilities during analysis.

The gaps covered by the fault injection framework proposed herein are summarised as follows:

1. The development of a framework that is able to abstract the fault injection component from the NUT.
2. The development of a framework that is portable to different time-triggered protocols such as TSN and TTEthernet.
3. The development of a framework that is able to inject double faults in a distributed network.
4. The development of a framework with a masked delay footprint; a framework with low intrusiveness.
5. A fault injection framework designed to capture the integrated behaviour of all implemented TSN or TTEthernet protocols.
6. A fault injection framework in which the analysis component applies deep learning to evaluate startup protocols before the establishment of the global time.

In summary, TRAITOR provides novel techniques beyond other methods by posing the ability to target individual traffic classes of the TTEthernet and TSN, including their synchronisation frames. TRAITOR is also designed to be portable in the sense that it can be tested on TTEthernet and TSN devices that are implemented by multiple vendors since no modification of the network participants is required. TRAITOR provides high controllability, as faults can be injected into specified traffic classes and also on particular bytes when desired. TRAITOR provides high observability, as it captures all injected faults and frames over the network. TRAITOR can capture faulty frames even before it is dropped by the switch/end system. TRAITOR has low intrusiveness; it utilises passive network test access points (TAPs) to monitor frames over the network, to avoid delay footprints. The delay on the fault injector component is masked.

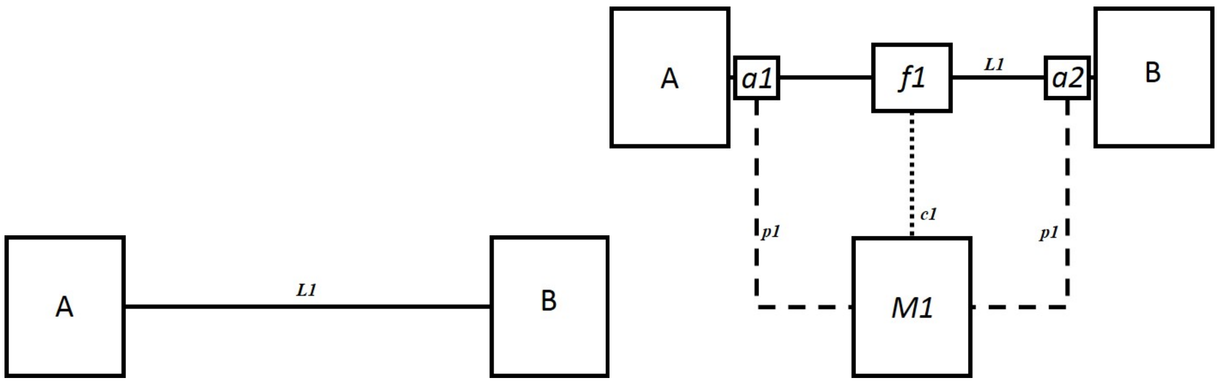
Chapter 5

System Model of Fault Injection Framework

This chapter begins with the description of the system model for TRAITOR in section 5.1. It further provides the description of the parts of TRAITOR that applies to TTEthernet, and its design in section 5.2. The TSN aspects of TRAITOR are described in section 5.3. The FPGA block diagram is then presented in section 5.4. The controller software design of the fault injection framework is described in section 5.5. Finally, a summary of how to use TRAITOR is given in section 5.6. In-depth implementation details of the fault injection framework are discussed in the next chapter.

5.1 System model

TRAITOR is modelled using an architecture similar to Figure 2.3 with the exception that the system under test is a network. TRAITOR is subdivided into the following components: target system, fault injector, controller, workload, monitor, and data collector. The target system is the network whose protocol is to be validated by TRAITOR. TRAITOR is integrated into the target system according to a cut-through paradigm. Consider Figure 5.1a, which depicts two network component A and B, connected by a link *L1*. Figure 5.1b depicts the same network with TRAITOR connected according to a cut-through paradigm. TRAITOR is placed between component A and B, as both components continue to interact without the participation of devices belonging to TRAITOR. The components data collector (*a1 and a2*), fault injector (*f1*), and the Monitor station (*M1*) are the parts belonging to TRAITOR. The controller and monitor are implemented in *M1*. The workload generator is all part of the system configuration for the NUT. The dotted links (*p1 and P2*) in Figure 5.1b connecting *a1* and *a2* to *M1* are parallel network different from the NUT, and belonging to TRAITOR. The parallel network serves two purposes. Firstly, the parallel network measures end-to-end latency between components A and B. Secondly, it sniffs all the traffic over the network. The link *c1* connecting *M1* and *f1* is used to set configuration parameters for TRAITOR.



(a) NUT without TRAITOR (b) NUT connected to TRAITOR

Figure 5.1: Overview of System Model

5.1.1 Fault hypothesis

Each node of the target network is considered to be a Fault Containment Region (FCR). The FCR is a subsystem that continues to function correctly, even in the presence of faults that occur outside the region. This work aims at emulating physical faults in FCRs. TRAITOR is designed to inject several communication failures that are defined in the IEC61508 standard with the addition of time synchronisation failure. The supported failure modes include:

- The corruption failure emulates changes to the original message, in which it alters the content of a frame. An example of a possible source of corruption failure can be due to Electromagnetic Interference (EMI) disturbances. TRAITOR emulates corruption failure by altering the content of the frame.
- Omission failure: An omission failure occurs when a transmitting node fails to send a message or when a receiving node fails to receive a message.
- Delay failure: TRAITOR emulates delay failures that originate from faulty nodes. A transmitted message is delayed for a configurable specified time.
- Babbling Idiot: A babbling idiot failure occurs when a transmitting node violates its temporal specification, thereby causing transmitted messages to monopolise the network.

- Masquerading: A masquerading failure occurs when a node assumes the identity of another node in the network without authority.
- Link failure: TRAITOR emulates a crash failure of a link.
- Crash failure: The crash failure describes a node which does not produce any output, that is a node that becomes inactive in the network.
- Time synchronisation failure: The time synchronisation failure is used to describe a scenario where the network does not meet its temporal specification. The messages from the nodes are delivered either too early or too late [Obermaisser, 2011].

TRAITOR executes the above failure modes according to a configurable time interval and emulates the behaviour of a failed node by altering the transmitted messages or the timing behaviour of the transmitted message.

5.2 TRAITOR in TTEthernet

TRAITOR injects error directly into the communication link of the TTEtherent network setup. The fault injector component of TRAITOR is placed inside the TTEthernet network according to a cut-through paradigm. The placement of the fault injector is such that TTEthernet network traffic traverses through the fault injector component. The cut-through approach which features that the fault injector is not implemented in the components of the TTEthernet network participant satisfies the abstraction requirement, mentioned in the chapter 4. The fault injector is also implemented on an FPGA to provide low invasiveness, high reachability, and portability for the framework. Utilising the programmable logic of the FPGA in implementing the network traffic pass-through, provides a faster computational capability for the framework. Thereby introducing minimal processing delay footprint. The little processing delay introduced by the cut-through is to satisfy the low invasiveness requirement. This processing delay is also constant. Therefore, the delay footprint can be removed during post-processing of results obtained from an experiment. The fault injector can target all the inputs/outputs of a node, using the programmable logic of the FPGA. All the states of the nodes can thus be impacted by the consequences of the fault, demonstrating high reachability. Since the cut-through approach utilises a fault injection component abstracted from the components of the NUT, it is thus portable from one testbed to another. Controllability in terms of time and space is also made possible by the implementation of the fault injection in the programmable logic of an FPGA. In the FPGA's programmable logic, the traffic signals can be read and manipulated at the MAC layer of the TTEthernet network protocol. The fault injector

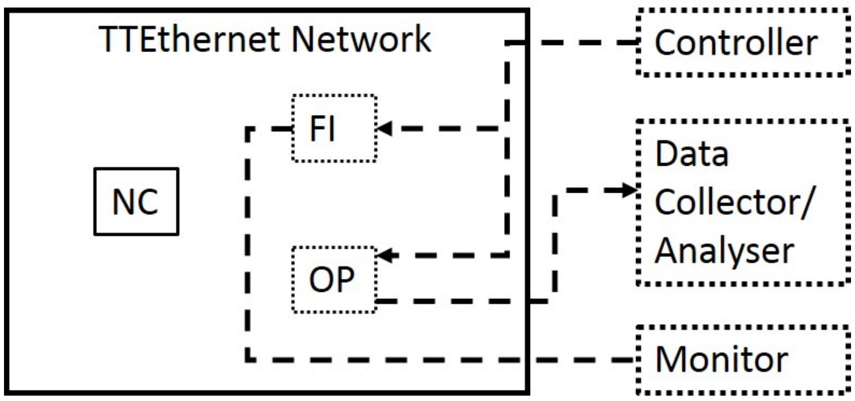


Figure 5.2: System Architecture of the fault injection framework

can target individual bytes. TRAITOR shows its application for high fidelity by targetting the physical implementation of the Time-triggered Ethernet systems. Finally, high observability is attained by utilising hardware-implemented network sniffers placed by the user at desirable points of the TTEthernet network.

5.2.1 Architectural overview of TRAITOR

TRAITOR is designed such that it can be placed at any point of the TTEthernet network. Figure 5.2 illustrates the system architecture of the TRAITOR.

The dotted components from the figure 5.2 are the components of the TRAITOR. The TTEthernet network consists of switches and end systems. The switches and end systems play different roles, as explained in section 3.3. The switches and end systems are all referred to as Network Components (*NCs*) in the diagram. The components of TRAITOR shown in the system architecture include the *Fault Injector* (*FI*), *Observation Probe* (*OP*), *Controller*, *Data Collector/Analyser*, and *Monitor*. The *FI* and *OP* can be placed at any point in the network irrespective of the network topology. The controller component is used to command and direct the operation of the FI and OP component. The FI injects a pre-configured fault into the network, while the OP collate message traces and feeds them to the data collector. The monitor is used to observe the operation of the FI, to track the activation of the injected fault.

Figure 5.3 and 5.4 illustrate how the system architecture can be mapped to a given physical setup. Figure 5.3 shows a typical TTEthernet redundant setup consisting of four end systems and four switches. Two switches operate as compression masters, and two other switches operate as synchronisation client. All the end systems operate as synchronisation masters.

Whereas, in figure 5.4, a single channel is used to describe the possible placement of

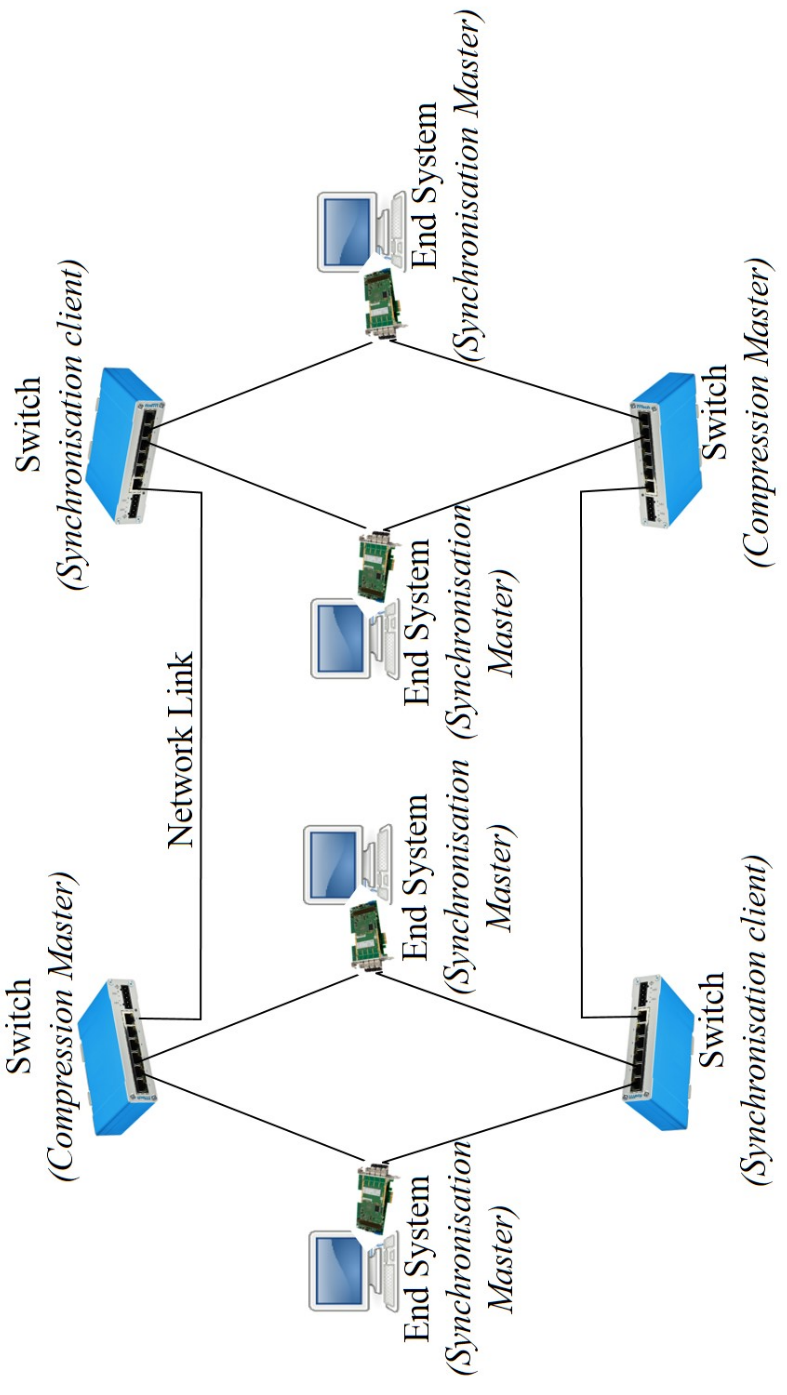


Figure 5.3: A typical redundant TTEthernet setup

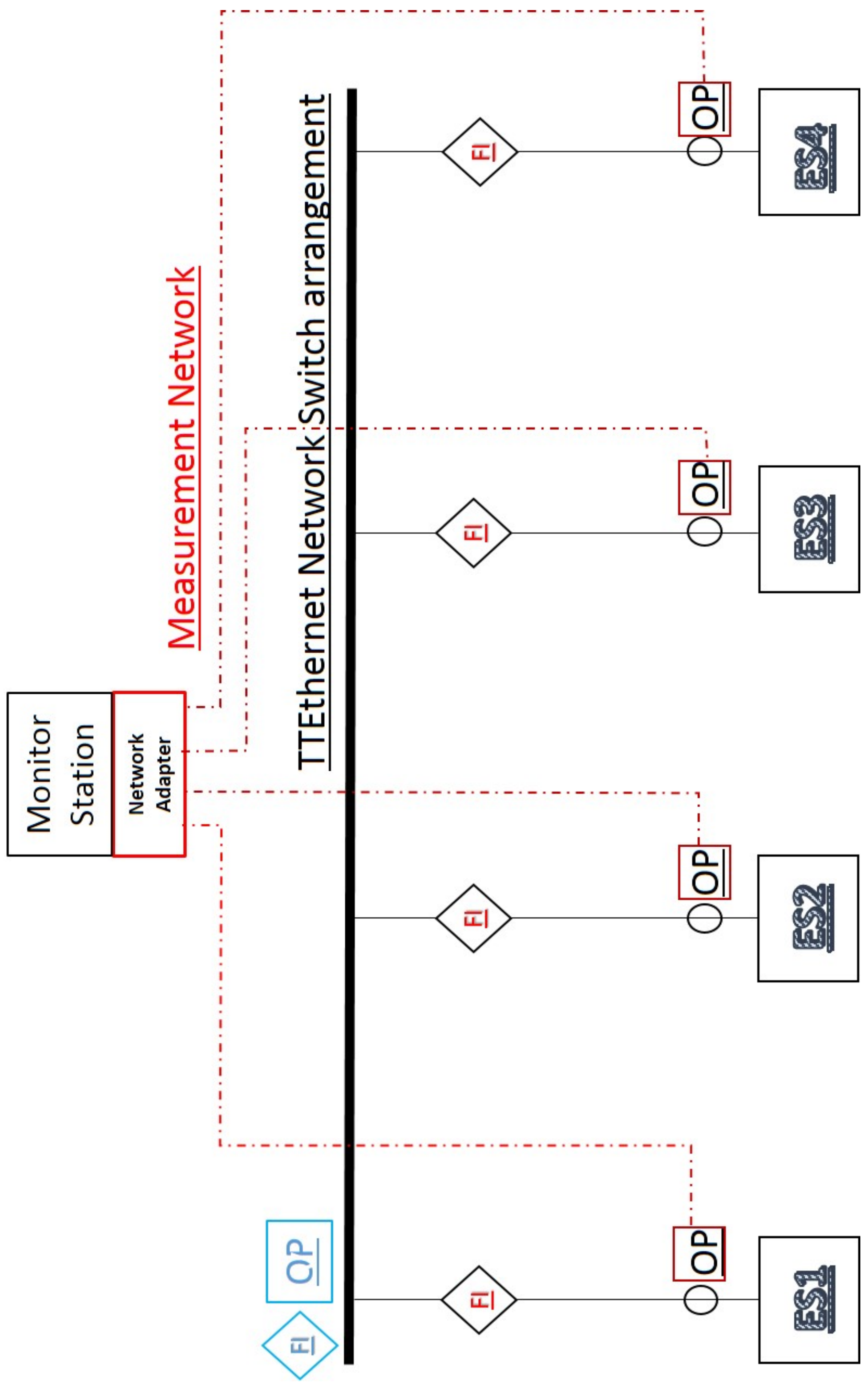


Figure 5.4: Simplified illustration of the fault injection framework on one channel.

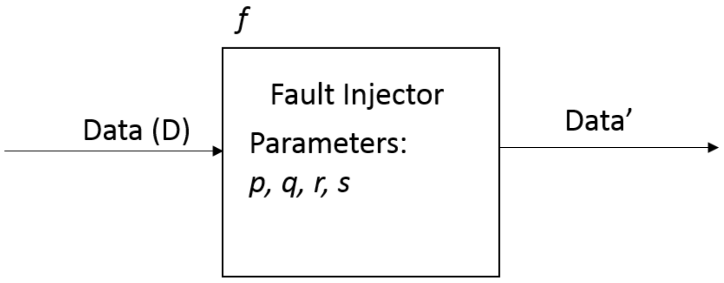


Figure 5.5: Fault Injector Model.

TRAITOR components in the network. The FI can be placed between an end system and a switch or between two switches. Likewise, the OP can be placed in any of these locations to observe the NUT. A parallel network is set up to probe the NUT. The reason for the parallel network is that when a fault is injected into the NUT, the NUT has the likelihood to become ladened with errors. Therefore, confidence cannot be placed on the measurements acquired by the OP if it is in the same network as the NUT. The second reason for using a parallel network is because of the need for a network asynchronous to the NUT. If data acquired by the OP is triggered by the same clock and the synchronisation mechanism of the NUT, then it means the OP is in the same failure domain as the NUT. Therefore, a different network (measurement network shown illustrated in 5.4) is used to observe the NUT. The monitor station comprises the controller software which includes monitoring task of the FI and a network capture software. The subsequent subsections provide more details on the different TRAITOR components.

5.2.2 Fault injector component

The fault injector is implemented using the Zedboard which is a low-cost FPGA development board for the Xilinx Zynq©-7000 SoC. The FPGA communicates with a host-PC through a JTAG connection and a UART serial cable. The parameter settings: *experiment duration*, *type of fault to inject*, *rate of fault injection*, *and number of experiments* are consequently transferred through the JTAG cable to the FPGA. The model of the fault injector component is shown Figure 5.5. The parameters *experiment duration*, *type of fault to inject*, *rate of fault injection*, *and number of experiments* are represented in Figure 5.5 as p, q, r, s respectively.

The FPGA is connected according to a cut-through paradigm. The TTEthernet network traffic flows into the FPGA, and then a fault is injected depending on the failure mode activated, after which the data flow out of the FPGA. The Xilinx Zedboard is equipped with only one Ethernet port. However, an Ethernet FMC (FPGA Mezzanine Card) from Opsero Electronic Design Incorporated is added to the FMC connector of the FPGA.

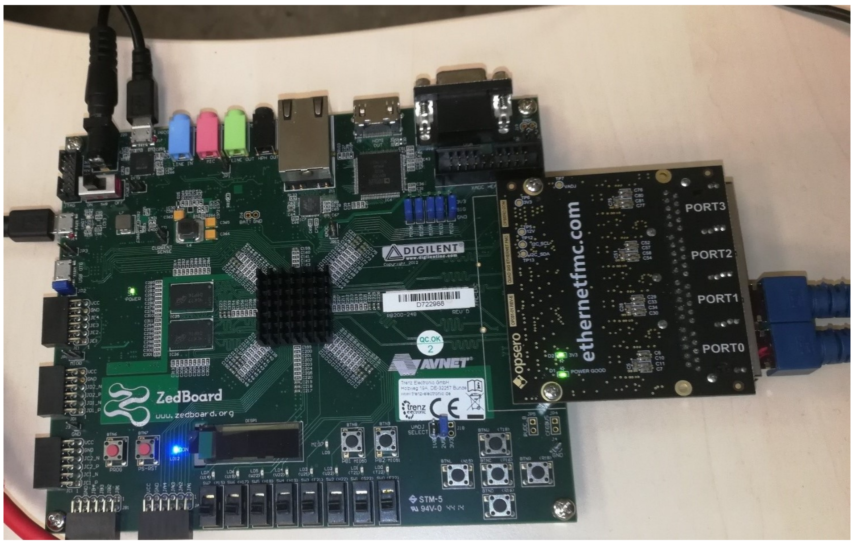


Figure 5.6: Zedboard with an Ethernet FMC addon.

The Ethernet FMC utilises 4 Gigabit Ethernet ports. Figure 5.6 shows a picture of the Xilinx Zedboard connected with an Ethernet FMC card on the FPGA's FMC port. The fault injector uses two ports of the Ethernet FMC port (i.e. PORT0 and PORT1 from the figure). The fault injector acts on an Ethernet full-duplex traffic. Depending on the direction of traffic, one port is used as ingress port and the other as the egress port.

Since the FPGA is connected as a cut-through in the TTEthernet network, it has access to all the network traffic for the link in which it is connected.

The failure mode triggered by TRAITOR to emulate failures in TTEthernet network is attained using either one or a combination of the following techniques listed below.

1. Nibble/Byte Insertion: The FPGA inserts or corrupts the nibbles in a frame that is transmitted through the link.
2. Frame Delay: The FPGA implements a delay function for configurable target frames to cause predetermined delays on the network traffic.
3. Frame Jamming: This implements frame jamming, which is used to prevent one or several end systems from receiving or sending frames. The FPGA jams specified frame(s) to emulate frame drop.

Table 5.1 summarises how the combination of the techniques mentioned above is used by TRAITOR to trigger different failure modes. The possible fault locations are summarised in Table 5.2. Table 5.2 shows where the failure modes can occur. The three possible locations are the switches, end system and links.

Failure	**Strategy**	**Insert.**	**Delay**	**Jam**
Omission	*This is implemented by jamming target frames with the FI module.*			x
Corruption	*The FI module inject nibbles/bytes to replace correct bytes/nibbles.*	x		
Link	*The FI module jams both incoming and outgoing frames on a link to implement the link failure.*			x
Crash	*Node crash is implemented by jamming all frames to and fro the target node.*			x
Delay	*This is realised by applying delay function using the FI module network traffic.*		x	
Stuck-at	*The stuck at failure is implemented by first jamming the correct message frame, then injecting the target specified frame in a continuous manner.*	x		x
Babbling Idiot	*The FI module generates and injects sporadic messages over the network*	x		
Masquerading	*This failure is realised by changing the ID's for network traffic using frame injection (e.g. the Virtual Link ID for TT frames).*	x		
Time Synch	*The FI in this case applies frame injection, frame delay, and frame jamming for PCF frames. This is done in a manner to omit, corrupt, and delay PCF frames.*	x	x	x

Table 5.1: Illustration on how the frame injection, frame Delay and Jamming can be used to implement the different failure modes.

5.2.3 Observation probe

The observation probe is realised in TRAITOR using COTS components. The Neox Networks P100CCA network Test Access Point (TAP) is used in this work to ensure that minimal delay footprint is introduced during measurement. The probes can be placed at locations desirable by the tester. The probe sniffs network traffic and transmits a copy of the frames to the **data collector**.

	Failure Mode	**End node**	**Switches**	**Links**
1	Omission	x	x	
2	Corruption	x	x	x
3	Link			x
4	Crash Failure	x	x	x
5	Delay Failure	x	x	
6	Stuck-at-	x	x	
7	Babbling Idiot	x	x	
8	Masquerading	x		
9	Time Sync	x	x	

Table 5.2: Possible Fault locations

5.2.4 Data collector/analyser

TRAITOR also uses COTS devices to collate data. It uses high-resolution Network Accelerator Cards (NACs). The NAC is manufactured by Napatech, and the model used for TRAITOR is the Napatech NT4E2-4-PTP. These NACs are used to obtain readouts with nanosecond accuracy for timestamps on frames. The NAC is used to determine the latency of the transmitted frames over the NUT. The OP (i.e. the network TAPs) extracts a copy of the traversing frame from the NUT in a passive manner and forwards it to the NAC which is used to provide a hardware timestamp just before the frames are received in the monitoring station. The original transmitted frame travels completely through the NUT; only the copy is sent to the NAC. The *Tshark* software [Combs, 2006] is used to analyse the data obtained. *Tshark* runs on a host PC that is connected to NACs. Analysis such as the computations of latency and jitter of different virtual links are handled offline via a custom program, designed specifically for analysing time-triggered Ethernet traffic. *Tshark* is operated with a library provided by Napatech to identify the interfaces of the NAC. The captured traffic is exported as Comma Separated Values (CSV) files for post-processing. Apart from computing the latency, jitter and the number of dropped packets from the captured data, a novel approach to observe the startup behaviour of synchronisation service is implemented in TRAITOR.

5.2.5 Controller

A software program is designed using C++ to control the entire experiment conducted using TRAITOR. The software controls the operation of TRAITOR through a compact and functional Graphical User Interface (GUI). The controller software can configure TRAITOR, and adjust configured settings during run-time as well as the initiation and termination of a fault injection experiment. The controller directs the operation of setting

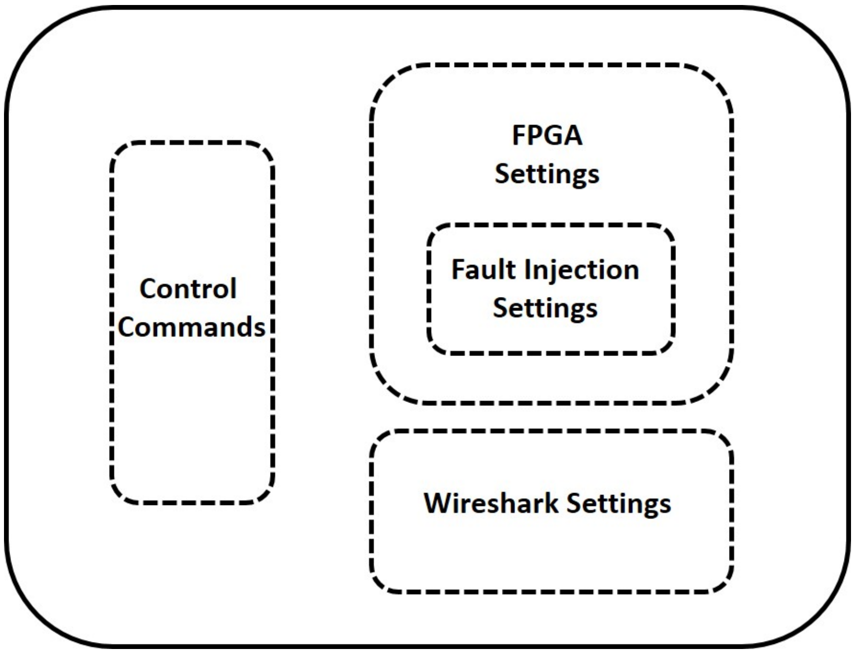


Figure 5.7: High-level Architecture of a Graphical User Interface.

up several subprograms. Figure 5.7 shows a high-level representation of the controller GUI.

The component *Control Commands* contains the instructions, which tell the program when to start and stop the experiment, load settings, and choose the number of experiments. It also contains the input command for the duration of each experiment. The *Fault Injection settings* incorporate all the information regarding the external insertion of the faults, such as the type of the inserted fault, frequencies, delays or the byte selection. The next component is *FPGA Settings*, which includes all the set of registers that determine the behaviour of the FPGA; these registers also contain the information introduced in the *Fault Injection settings*. The *Wireshark settings* is the network monitor software used to analyse the captured data. It uses the *Tshark* tool to capture the traffic in the network and save the experimental data in ordered files. The content of experimental data consists of the timestamps for the reception and transmission of frames, source address, a destination address, interface connection, and message content. The controller also directs *Tshark* when to stop capturing data from the network.

5.2.6 Monitor

The **Monitor** component implements a series of programmable logic output ports that produce HIGH and LOW signals. These signals are used to track and monitor a series of events that occur in the **FI**. Flags are set for events, such as when the **FI** detects the frame header, or when the **FI** injects a fault, or when the **FI** changes from one state to another (the **FI** states are discussed in the next chapter). The monitor also detects when an error occurs during the execution of TRAITOR.

5.3 TSN fault injection framework

The design of TRAITOR to function on TSN is similar to that of the TTEthernet. There exist only one major difference which is the implementation of the **FI** component described above. Similar architecture with the TTEthernet is employed for the **controller** component, **OP** component, **Monitor** component, and **data collector/analyser**. The FI can switch between two modes of operation, one for the TTEthernet and the other for the TSN. The frame structure of TSN is similar to TTEthernet (i.e. both are based on the IEEE 802.3), but the content and mode of operation are different. Therefore, the detection of TSN frame by the **FI** is based on the different logic block from the TTEthernet. The **FI** detects TSN frame to decide its handling based on the configuration inputted. If TSN mode is configured for TRAITOR, the **FI** targets TSN frames.

5.4 FPGA block diagram design

The FI is implemented on an FPGA and designed using the Xilinx Vivado Design Suite. Figure 5.8 illustrates the design of the FI. Herein, the network tap concept introduced by [Johnson, 2015], is used to implement the FI. The ZYNQ processing system block, AXI (Advanced eXtensible Interface) interconnection block, FIFO block, and GMII-RGMII block are Xilinx's intellectual properties (IPs). The ZYNQ Processing system and AXI interconnection block are both used for configuration purposes. The ZYNQ block is used to configure the network speed of the GMII-RGMII block through its GEM1 (Gigabit Ethernet MAC 1) port. The configuration parameters for TRAITOR are given through the AXI interconnection block using the ZYNQ M_AXI GPIO port. The dotted arrows in Figure 5.8 illustrates the path of a network cut-through via the programmable logic of the FPGA.

TRAITOR realises a network pass-through using two PHY ports form the Ethernet FMC which supports Reduced Gigabit Media-Independent Interface (RGMII) [Packard et al., 2000]. The RGMII is intended to be an alternative to the IEEE802.3u MII standard, it is developed to reduce the number of signals required to connect a PHY to a MAC and supports 10/100/1000 and Mbit/s data transmission. Table 5.3 summarises the number of bits transmitted between the PHY and MAC for each clock cycle based on the different

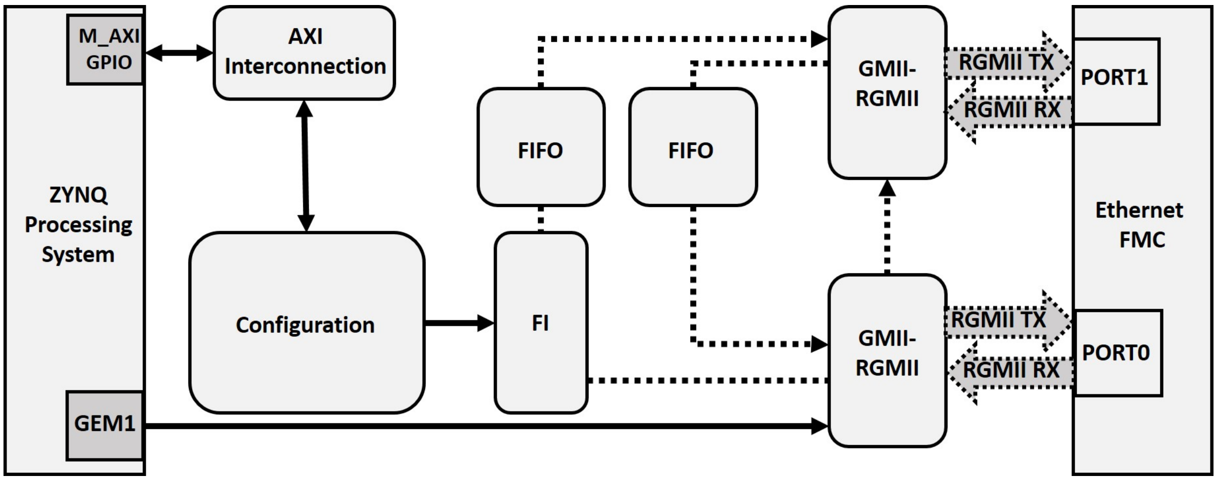


Figure 5.8: Block Diagram of FI design on FPGA.

Mbit/s	MHz	Bits/Clock Cycle
10	2.5	4
100	25	4
1000	125	8

Table 5.3: Relationship between number of bits supported by RGMII and network speed

network speeds.

RGMII transmits data on both rising and falling edges of the clock. The transmission of data at both clock edges is known as Double Data Rate (DDR). The GMII-RGMII block from Figure 5.8 converts the DDR signals to a single-data-rate. Each Ethernet FMC port is a full-duplex port (i.e. the port can send and receive data at the same time on a signal carrier). When connecting a GMII-RGMII block to another; separate clocks are observed between the transmit interface and receive interface. Since both ports are independently clocking to transmit and receive data, there is a need for a cross-clocking domain. The cross clocking domain is implemented with FIFO (First In, First Out) buffers. However, the use of an elastic buffer is proposed in [Johnson, 2015], because if momentarily the FIFO is being read slightly faster than it is being written to, there will be occasions where the FIFO is empty for one clock cycle and forced to de-assert the "valid" signal.

5.5 Software design

The controller software is used to control the operation of TRAITOR. Fault injection experiments require multiple runs to increase the confidence placed on the results obtained. Therefore, the controller software is also necessary to automate TRAITOR. The controller software is mainly to issue the commands required to configure and program the FPGA,

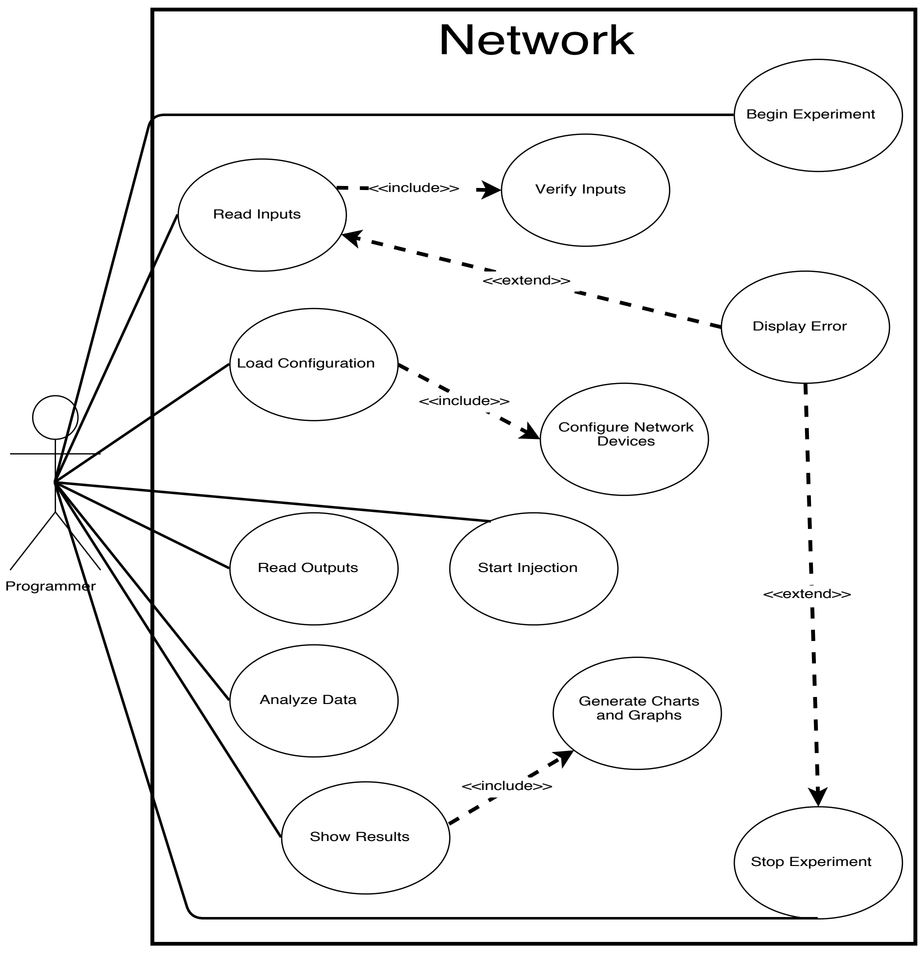


Figure 5.9: Use case diagram of TRAITOR controller.

control the capture interface, and control the execution and number of experiments. Figure 5.9 illustrates the use-case diagram of the controller software.

The programmer can perform operations such as read inputs, input TRAITOR configurations (e.g. type of fault, FER, duration of experiments e.t.c.), read outputs, analyse data, show results, start and stop the experiment. The controller is designed in modules, to achieve the functions displayed in the use case diagram. Each module is used to establish a program that provides control capability for the main *controllerModule*. Figure 5.10 shows the different modules. The *captureModule* is used to provide control capability for the *controllerModule* to control the Tshark software. The *loadConfigurationModule* provides the control capability for the *controllerModule* to manage the C-based program used to program the FPGA. The *faultDetectionModule* provides a control capability for the *controllerModule* to handle fault detection when establishing connections between

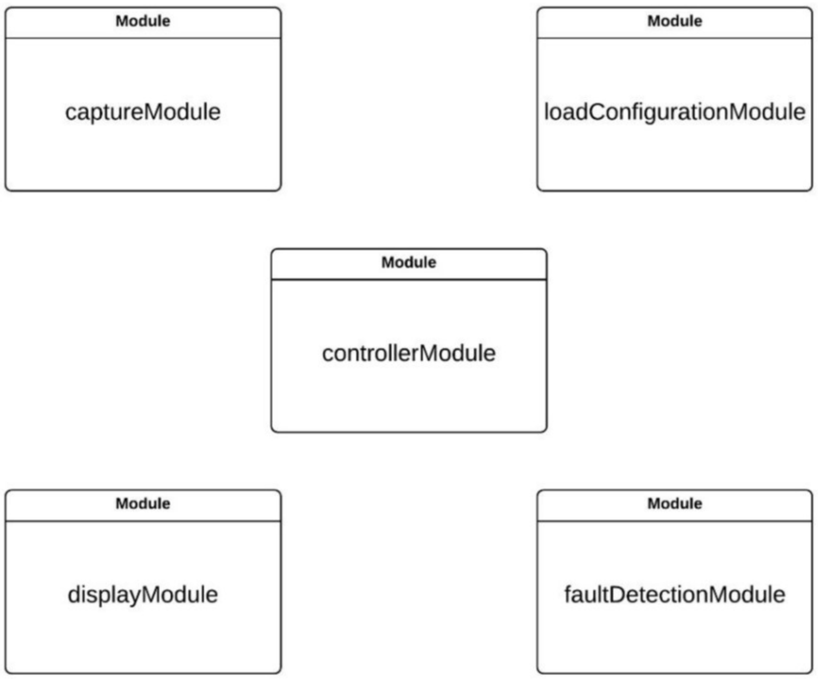


Figure 5.10: Modules of the controller software.

TRAITOR software and the FPGA. Finally, the *displayModule* is used to implement the GUI for TRAITOR.

Figure 5.11 shows the classs diagram to illustrate the methods executed in managing the network interface (i.e controlling Tshark) and methods used to control the entire experiment. In the Interface class, the software uses the *getName()* and *getID()* to obtain the interface name and ID, and uses the *setName()* and *setID()* to set the interface name and ID respectively. In the CaptureControl class, the software uses the *start()*, *stop()*, *loadConfig(..)* method to start the experiment, stop the experiment, and load configuration for the experiment.

5.6 TRAITOR operation summary

The usage of the framework can be summarised as follows. The user connects network TAPs to the desired points of the network links. Likewise, the user connects the FI in a cut-through manner to the links where fault injection is intended. TRAITOR software is then started to pop-up the GUI. Using the GUI, the user then sets the configuration mode (e.g. TSN or TTEthernet), after which the user inputs all the corresponding parameters and configurations. For example, the configuration parameters include the type of fault to inject, frequency of injection, network speed (100Mbit/s or 1000Mbit/s), target frame, and

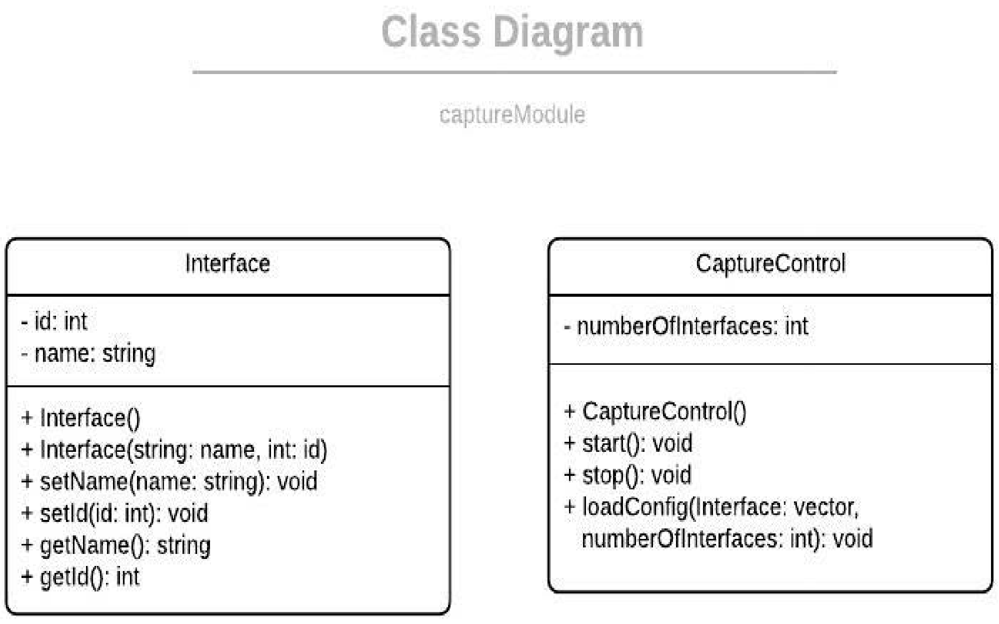


Figure 5.11: Class diagram of the captureModule.

duration of the experiment. After that, the user inputs configuration commands to set up Tshark for capturing. The user starts the test with a command issued by a button click in the GUI. As the experiment runs, TRAITOR saves the results of individual experiments in an ordered fashion. The analysis of the results obtained is carried out offline using C++ software designed for the purpose.

Chapter 6

Implementation

This chapter explains the implementation of TRAITOR's fault injection component. Detailed insight into the overall implementation strategy of the fault injection component is given in section 6.1. Section 6.2 describes the receiver logic implementation of TRAITOR. Subsection 6.2.1 describes the retriever logic component of TRAITOR. In section 6.3, the modifications needed for TSN and its implementation is described.

6.1 Fault injection component

As mentioned in the previous chapter, the abstraction of the fault injection component using an FPGA for cut-through is chosen to avoid significant delay footprint caused by the FI component. However, the only delay introduced is the processing delay of the FPGA (FI component). The processing delay Δt_c is equal to 850 nanoseconds for 1000 Mbit/s network and four microseconds for 100 Mbit/s network. This processing delay is unavoidable because to make decisions on what to do with the frame that enters the FI component requires reading the frame's MAC header. Examples of the decision referred to herein include the decision on where to inject faults and on which frame should TRAITOR inject faults. However, this delay is fixed once the FPGA is programmed, and is resolved during analysis. The delay can also be factored into the configuration of the NUT since it is a fixed delay.

The FI is modelled as a Finite State Machine (FSM) and is made up of a serial composition of two state machines. The state machines include:

- Receiver Logic (RecL)
- Retriever Logic (RetL)

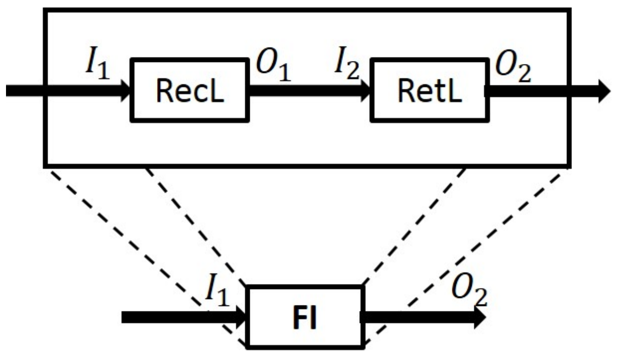


Figure 6.1: Cascade composition of RecL and RetL.

The FI serial composition is illustrated in Figure 6.1.

The output port O_1 from **RecL** feeds into the input port I_2 of **RetL**. Both O_1 and I_2 are of the same data type (byte array) V_1 and V_2 respectively, therefore

$$V_1 \subseteq V_2.$$

This asserts that the output produced by **RecL** on port O_1 is an acceptable input to **RetL** on port I_2. The reaction of **RecL** and RetL is asynchronous, therefore, a machinery for buffering the data that is sent from **RecL** to **RetL** is implemented. The RecL first passes the data to a shift register from which it is forwarded to the RetL. It is in the shift register that the MAC header is read for decision making. The delay Δt_s imposed by the shift register is a composite of the processing delay Δt_c mentioned above. The Δt_s is also a fixed delay. The size of the shift register is tied to the size of the MAC header. The shift register enables the **RetL** to determine the beginning of a frame, and the frame type and size. The content of the payload is not of interest to TRAITOR for TTEthernet. The information concerning the MAC header retrieved from the shift register is sufficient to inject faults.

6.2 Receiver logic

The **RecL** is implemented in the ingress port of the FI. Its main function is to receive and write incoming frames into a shift register. The RecL also sets the value of a variable *"Fire"*, which is used to control state changes in the RetL. The RecL implements the input instructions from the user. Algorithm 1 summarises the operation of the receiver logic. The process that executes the RecL reacts to parameters in the sensitivity list which include the *current_state*, *trigger*, *clk*, *isPreamble*, and *data_out_reg*. The *current_state* is maintained by TRAITOR to hold the current state. The *trigger* is the input command received from the user to indicate to TRAITOR the type of failure to execute. The *clk* is TRAITOR's reference clock. The *isPreamble* is a boolean variable to indicate the start of a frame, and the *data_out_reg* signals changes in the shift register. Whenever any of these variables changes, the RecL reacts. The state operates such that it passes the user input value *trigger* to cause state changes in the RetL state. These reactions are also dependent on the current state of TRAITOR, which is also given as user input. The algorithm is executed at the rising edge of the clock; $clk =' 1'$.

Algorithm 1 The RecL Operation

```
1: Process RecL($current\_state, trigger, clk, isPreamble, data\_out\_reg$)
2:     if $clk =' 1'$ then
         {CASE} $fault\_config.faultType$
         {when} $x"1" =>$
3:         $Fire <= x"1"; ---CorruptionfailureState$
         {when} $x"2" =>$
4:         $fire <= x"2"; ---OmissionfailureState$
         {when} $x"3" =>$
5:         $fire <= x"3"; ---Linkfailurestate$
         {when} $x"4" =>$
6:         $fire <= x"4"; ---DelayfailureState$
7:         $delayFaultFlag <=' 1';$
8:         $res <=' 0';$
         {when} $x"5" =>$
9:         $fire <= x"5"; ---MasqueradingFailureState$
         {when} $x"6" =>$
10:        $fire <= x"6"; ---BabblingIdiotFailureState$
         {when} $x"7" =>$
11:        $fire <= x"7"; ---TimesynchfailureState$
         {when} $x"8" =>$
12:        $fire <= x"8"; ---CrashfailureState$
         {when} $x"9" =>$
13:        $fire <= x"9"; ---Stuck-AtFailureState$ {when} others=>
         {ENDCASE}
14:    end if
15: end Process
```

6.2.1 Retriever logic

The shift register is implemented in the RetL. The Algorithm 2 shows the implementation of the shift register.

Two major functions are implemented in the shift register, one of which is carried out

Algorithm 2 The RetL Shift register algorithm

```
Process SHIFTREG($clk, signalEnable$)
    while signalEnable = true  do
        if clk = 1 then
            Obtain and update current shift register position counter
            Execute function (LocateHeader)
            if LocateHeader returns true then
                Execute Function (checkFrameContent) to identify Target Frame
                if $checkFrameContent$ returns true then
                    Update target Flag
                end if
            end if
        else
            Do nothing
        end if
    end while
end Process
function $locateHeader(mem : register0)$
    if shiftRegContent = EthernetPreamble then
        return true;
    end if
end function
function $checkFrameContent(mem : register0)$
    if shiftRegContent = TargetFrame then
        return true;
    end if
end function
```

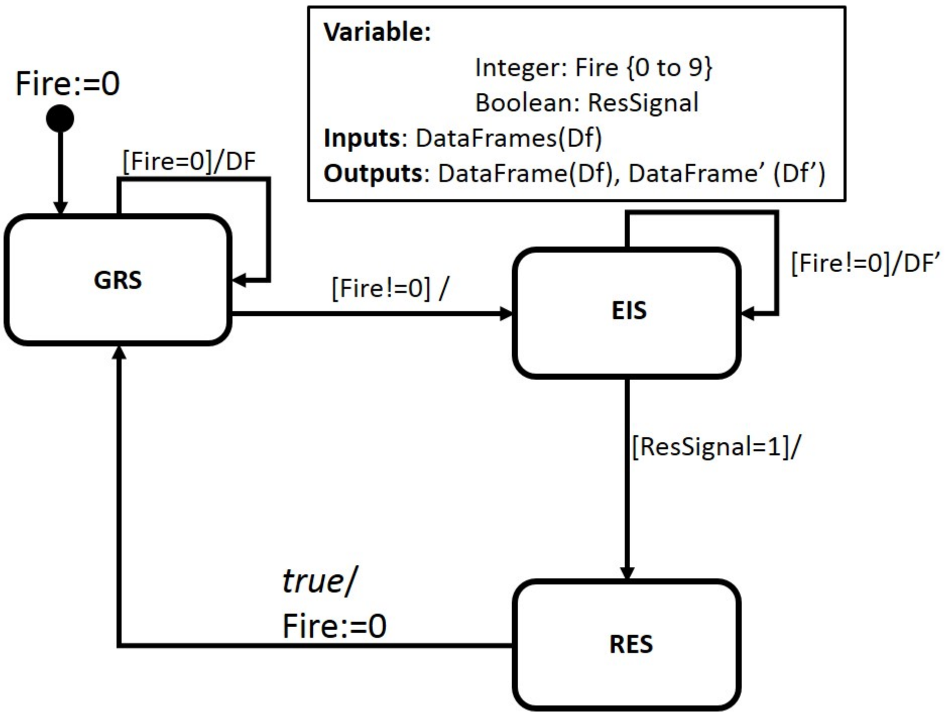


Figure 6.2: RetL state machine.

to detect the beginning of a frame and the other to identify the configured target frame to inject an error. The shift register process operates on every rising edge of the input clock. It maintains a counter that tracks the position of the bytes, starting from when it receives the frame header. The *locateHeader* function locates the frame header. When the header is identified, it then executes a function *checkFrameContent* to read the frame content as the bytes move through the shift register. The shift register updates a flag when it identifies the frame content.

The **RetL** is modelled as a complex state consisting of three major sub-states namely, Golden Run State (**GRS**), Error Insertion State (**EIS**), and the Reset State (**RES**). The RetL state machine is shown in Figure 6.2.

The RetL state initialises by entering the GRS. In the GRS state, the input data is allowed to pass through the RetL without any error insertion. The input data frame is shown as "**Df**" in the state machine diagram. The GRS represents the state of the system when no fault is applied, and it is assumed that the FPGA is in a cut-through mode with the **Df** passing through it without any modification. TRAITOR maintains a positive integer variable named *"Fire"* to transition between states. When TRAITOR is started, *"Fire"* is set to "0". TRAITOR remains in the GRS state as long *"Fire"* remains "0". When the value of *"Fire"* is changed to a non-zero value, the system moves to the **EIS** state. The **EIS** state is responsible for the insertion of error into the input data frames. Data frames that are affected by errors are shown in the state machine as ***Df'***.

The **EIS** is a complex state consisting of the following state: Corruption, Omission, Delay, Masquerading, Babbling idiot, Link, Crash, TimeSynch, Stuck-At WaitOnFault, InjectionFinished. Transition into the **EIS** is made to the sub-state that has a value set for *"Fire"* corresponding to it. The **EIS** sub-states are illustrated in Figure 6.3. The EIS reads data on the shift register and then triggers the injection of the pre-configured fault.

6.2.2 Corruption state

The corruption state would corrupt the bytes in the frame as specified by the user. The corruption state has the potential to modify any byte in the TTEthernet frame structure on the fly, as the frame passes through the shift register. This state can target specific traffic classes, and it filters the data on the shift register then uses the MAC header to specify the frame type. The inputs to this state are the data input from the shift register, trigger signal to commence fault injection, and configuration parameters. This state is entered if the value of the trigger signals corresponds to the value set for its sensitivity. The sensitivity value for corruption is given when [*"Fire"* = 1] is inputted. The implementation makes it possible to configure a Frame Error Rate (FER). The FER is the frequency at which it is intended to inject faults. The behaviour of the corruption state is such that when a user specifies which byte to corrupt in a given frame and the intended FER, the state reads the data from the shift register and then performs a NOT-gate operation on the target byte. Algorithm 3 illustrates the implementation of corruption. The description of parameters that appear on the sensitivity list is similar to Algorithm 1. The *enable_in* and *error_in* are signals that move along with the **Df**. The signals are the Reduced Gigabit Media Independent Interface (RGMII) enable and error signals. The RGMII is the physical connection between the Ethernet physical layer and the Ethernet MAC. It is a dual data rate (DDR) interface that consists of a transmit path, from FPGA to the physical layer, and a receive path, from the physical layer to the FPGA. Details of RGMII can be found in [Packard et al., 2000]. The $targetFlag$ and the $faultflag$ are indicators for showing when the target frame is acquired, and when the time to inject a fault as specified by the user is reached, respectively. The corruption state maintains a counter "*corrLocation_counter*" to aim at the target byte.

For example, the corruption state can be pre-configured to corrupt the 3rd byte in the payload and to do this for every 5th frame. When the corruption state executes the first corrupted byte, it will set a counter equals a value that corresponds to the user-specified FER (in this instance FER = 5). It will then move to wait for the injection time in the *WaitOnFault* state. If the counter condition is met (i.e. $targetFlag =' 1'$

Variable:

Integer: **Fire {0 to 9}, Counter, setVal**

Boolean: ***CorruptBit, CorruptBitAck, OmmitBit, OmmitBitAck, FakeID, FakeIDAck, stopBI, stopD, stopLF, stopCF, stopTS, stopSA, ResetSig,* Exit**

Inputs: DataFrames(Df)

Outputs: DataFrame(Df), DataFrame’ (Df’)

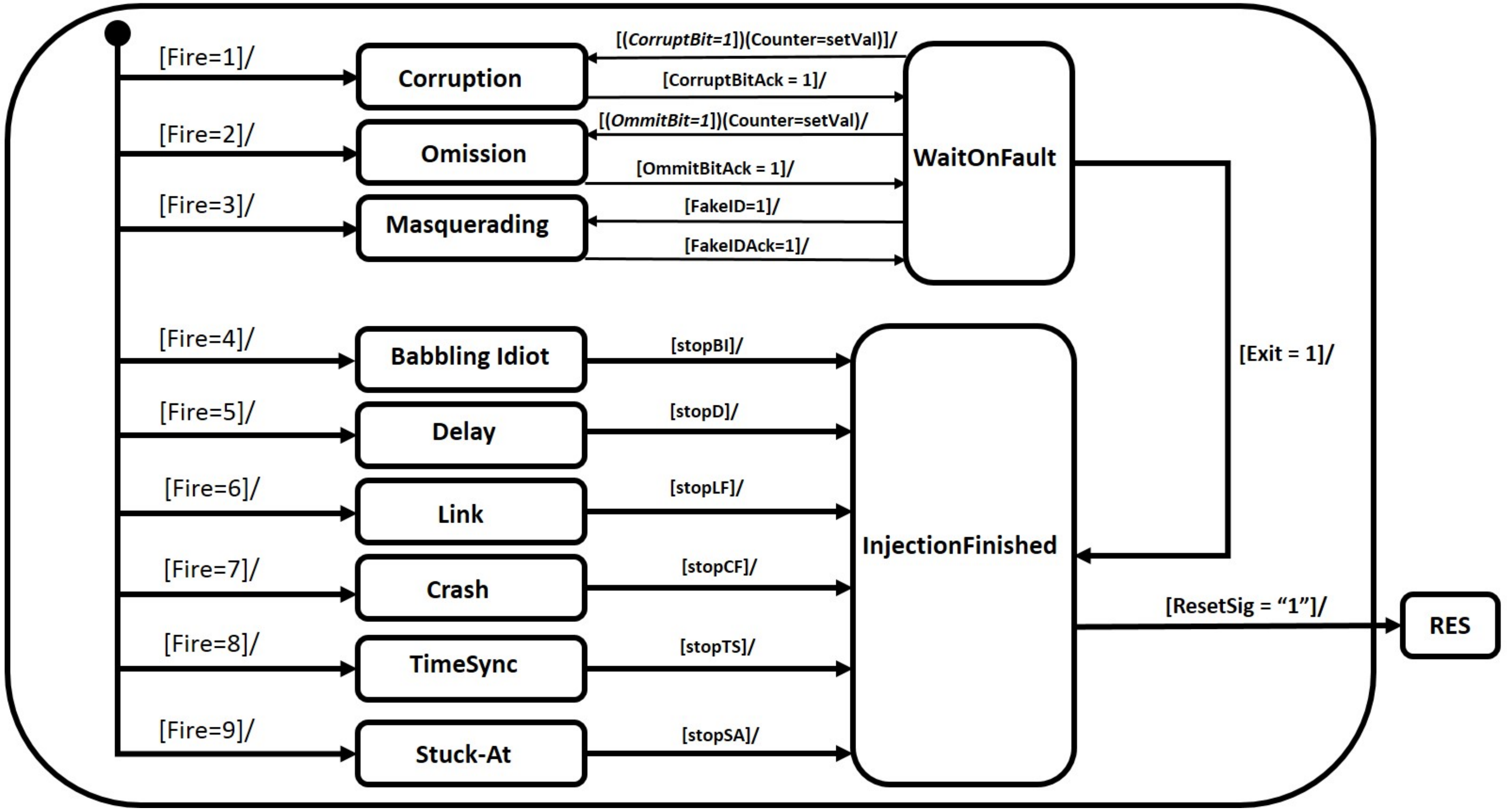


Figure 6.3: EIS state machine.

Algorithm 3 The RetL Corruption failure Algorithm

```
1: Process $RetrieverLogic(clk, data\_in, enable\_in, error\_in, data\_out_reg)$
2:     if $targetFlag =' 1'\&faultFlag =' 1'$ then
3:         if $byte\_corrupt = corrLocation\_counter$ then
4:             $data\_out\_value <= not\_operation\ (data\_out\_value)$
5:         end if
6:     end if
7: end Process
```

and $faultFlag =' 1'$), then the system transits from *WaitOnFault* state to the corruption state to execute another corrupt-byte procedure.

6.2.3 Omission state

This state causes network frames to be omitted according to the user specification, i.e. the frame types to be omitted are pre-selected by the user. In Omission state, an entire frame is omitted. The state implements a function to read the shift register, and based on specific user configurations; it omits the specified frames. An omission is achieved by not producing the output for the targeted frames. The Omission state blocks and discards the frame choice marked by the user. It resumes producing output once the frame blocking function is disabled. The inputs to this state include the traversing frame data input (contained in the shift register), and the trigger signals. The trigger signals are presented in the configuration parameters which are given in the sensitivity list shown in the RetL Algorithm. The omission state is entered when the value of [*"Fire"* = 2]. The FER is also configured for the Omission state. The state transitions to the *WaitOnFault* state when the target is identified and waits until the FER condition is met before omitting the target frame. The Algorithm 4 illustrates the operation of the omission state. When the omission state is done with the fault injection, it transitions first to the *WaitOnFault* state and then to the *Injection-Finished* state.

6.2.4 Masquerading state

The action of this state ensures that an end system assumes the identity of another end system. TRAITOR implements this by changing the source field of the MAC header, giving it the address of another end system. By so doing a masquerading failure is emulated where a system falsely assumes the identity of another system · Since the identity of an end system is stored in the frame, this information is available to the shift register. The source field is modified by replacing the content of this field with the address of another end system whose address is known apriori. When the state is activated, it modifies the

Algorithm 4 The Omission failure Algorithm

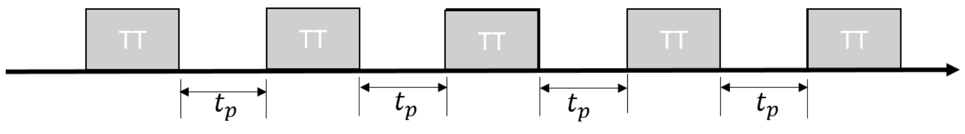
```
1: Process $RetrieverLogic(clk, data\_in, enable\_in, error\_in, data\_out\_reg)$
2:     if $targetFlag =' 1'$ and $faultFlag =' 1'$ then
3:         $data\_out <= x"00";$
4:         $enable\_out <=' 0';$
5:         $error\_out <=' 0';$
6:     else
7:         $data\_out <= push\_data\_out;$
8:         $enable\_out <= enable\_out\_reg;$
9:         $error\_out <= error\_out\_reg;$
10:    end if
11: end Process
```

source address field of the target frame it receives and transits to the *WaitOnFault* state, and then it waits for a period corresponding to the value set for the FER. When a counter condition is met, indicating that it is due to perform another masquerading fault, it will transit back to the masquerading state and cause another masquerading failure. The masquerading state is entered when [*"Fire"* = 3]. The algorithm 5 shows how masquerading failure is accomplished.

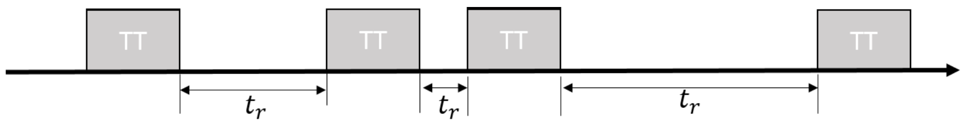
Algorithm 5 The Masquerading failure Algorithm

```
1: Process $RetrieverLogic(clk, data\_in, enable\_in, error\_in, data\_out_reg)$
2:     if $targetFlag =' 1'$ and $faultFlag =' 1'$ then
3:         if $counterMasq < startofDestinationaddress$ then
4:             $Pass\_input\_data\_to\_output\_data\_without\_modification$
5:         else if $(counterMasq >= SODA) and (counterMasq < EODA)$ then
6:             $Pass\_input\_data\_to\_output\_without\_modification$
7:             $CRC\_re - computation$
8:         else if $(counterMasq >= SOSA) and (counterMasq < EOSA)$ then
9:             $Replace\_source\_address$ AND $compute\_new\_CRC$
10:            $_push\_out\_masked\_source\_Address$
11:        else if $counterMasq > EOSA$ then
12:            $Push\_out\_the\_remaining\_frame$ AND $compute\_CRC$
13:        end if
14:    else
15:        $Pass\_input\_data\_to\_output\_data\_without\_modification$
16:    end if
17: end Process
```

The input to the state includes the trigger signal for the state, the data from the shift register, and the configuration parameters. As shown in Algorithm 5, a counter $counterMasq$ is maintained. It is used to locate and track the position of every byte in a frame. If fault injection is activated, the start and the end of the source address (i.e. the SOSA and



(a) TT frame periodic interval under normal operation.



(b) Inter frame gap illustration between TT frames.

Figure 6.4: Inter frame gap illustration between TT frames.

EOSA shown in the Algorithm 5 respectively) are located by the *counterMasq*. Since the source address is modified, the state also computes a new CRC for the frame. The CRC computation is achieved on the fly with no additional delay. The CRC computation begins with the start of the destination address field (SODA) and ends when the last byte of the payload is added.

6.2.5 Babbling Idiot state

The babbling idiot is characterised by the transmission of arbitrary messages at random points in time. TRAITOR emulates a babbling idiot failure mode by implementing a random interval between two consecutive frames. The time interval between two consecutive frames is known as the inter-frame gap (IFG). Therefore by making the IFG a random variable, frame transmissions are performed in an untimely manner. The babbling idiot state generates TTEthernet frames and implements a random function for the IFG.

Figure 6.4 illustrates the interval between two consecutive TT traffic. In the GRS state, TT frames are transmitted at a constant periodic interval (t_p). Therefore, the IFG between two TT frame is constant, as shown in Figure 6.4a. However, the IFG becomes a random variable when BIF is activated. Figure 6.4b illustrates the behaviour of TT frames with random IFG. It is seen that the IFG t_p shown in figure 6.4a assumes a random interval and is shown as t_r in figure 6.4b. In this way, the untimely generation of network traffic is realised. The babbling idiot failure state is entered when [*"Fire"* = 4]. Algorithm 6 illustrates the implementation of the babbling idiot failure. The frame description used for the generation process is provided during configuration by the user. Examples of frame description provided include source address, a destination address, constant field, and payload. The traffic is generated with random intervals using the case statements specified by the user. CRC is also recomputed by TRAITOR in the babbling idiot state.

Algorithm 6 The Babbling Idiot failure Algorithm

```
1: Process $RetrieverLogic(clk, data\_in, enable\_in, error\_in, data\_out_reg)$
       {CASE} $FrameType$
         {When} $frameType = TTconfig$
           {Generate} $TTmessages$    {Compute} $CRC$
         {When} $frameType = RCconfig$
           {Generate} $RCmessages$    {Compute} $CRC$
         {When} $frameType = BEconfig$
           {Generate} $BEmessages$    {Compute} $CRC$
       {ENDCASE}
2: end Process
```

6.2.6 Delay failure state

In delay state, target frames that enter the FPGA are delayed. The user configures the duration by which the frames are delayed. Transition is made into the delay state when the value of [*"Fire"* = 5], as shown in Figure 6.3. When the trigger *"Fire"* is set to 5, the state causes TRAITOR to delay every frame by a duration configured by the user. The frames that enter the FI leave the FPFA without any error injection except that they are delayed. Transition is made to *Injection-Finished* state when the action is completed. The Algorithm 7 is used to illustrate the implementation of the delay state. The delay state implements delay by buffering the frames.

Algorithm 7 The Delay failure Algorithm

```
1: Process $RetrieverLogic(clk, data\_in, enable\_in, error\_in, data\_out_reg)$
2:     if $EnableIn =' 1'$ and $faultFlag =' 1'$ then
         {SAVE} Data in Buffer if message available
         {WAIT for} $DelayValue = configDelay$
         $EnableOut =' 1'$
         {READ} Data from Buffer and Push to egress port
3:     else{Do Nothing}
4:     end if
5: end Process
```

If the *EnableIn* is high and fault injection is activated, frames are saved in the buffer. The *EnableIn* signal indicates when a frame is received in the ingress of the FI. TRAITOR takes control of the enable signal during re-transmission, this means that TRAITOR generates a new enable signal *EnableOut* when it needs to re-transmit the frame saved in the buffer. The *EnableIn* indicates when there is a frame available to save in the buffer. TRAITOR also checks to know how long to keep the frame in buffer. The duration in which the frame must remain in the buffer is pre-configured by the user. The time to re-transmit the frame is determined by Algorithm 7 when $DelayValue = configDelay$. The $DelayValue$ is used to keep the current delay of the messages in the buffer, while the $configDelay$ is the desired duration configured by the user.

6.2.7 Link failure State

The Link Failure state shuts out a link from the network. This is accomplished by an FPGA that breaks the link on the DDR channel that connects two network participant. This is implemented by blocking all outgoing and incoming frames on the desired link. In this state, the frames read from the shift register are discarded and not forwarded. The inputs to this state are the frame data input and the trigger signal. This state produces no output. Transition is made into the link failure state when the value of [*"Fire"* = 6], as shown in Figure 6.3. Algorithm 8 is used to illustrate the implementation of the Link failure state. The implementation is such that once the value of *"Fire"* is set to 6, TRAITOR produces no output.

Algorithm 8 The Link failure Algorithm

```
1: Process $RetrieverLogic(clk, data\_in, enable\_in, error\_in, data\_out_reg)$
       {When} [Fire = 6] Block link
2: end Process
```

6.2.8 Crash failure State

In this state, TRAITOR implements crash failure for both switches and end system. It can be viewed as a complete outage of a network participant. This framework implements the crash failure by totally blocking the DDR channel of the target participant (breaking every link connected to the participant). For example, in Figure 6.5 consider the setup of 2 switches (SW_1 and SW_2) and 2 end systems (ES_1 and ES_2). If the switch SW_1 is targeted to fail, all the links connected to it is cut off, as illustrated with the points on the link marked x. The points marked x are FPGAs connected according to a cut-through paradigm. When a crash failure is triggered, the DDR channels are simultaneously cut.

The implementation is similar to the link failure, except that crash failures are cooperating link failures. For example, to emulate a crashed end system, all the links connected to the end system are shut out. By so doing, there is no participation of the affected system in the network. The crash failure is achieved by having the controlling FPGA send out a block signal to other FPGAs connected on the links attached to the target node.

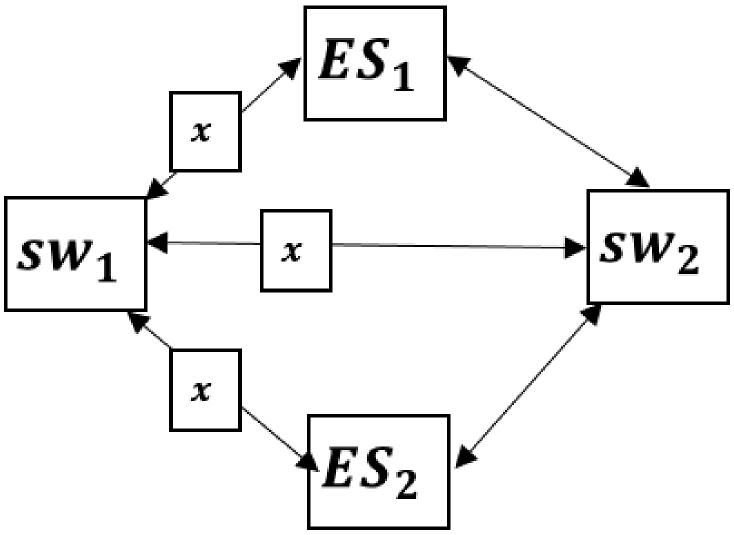


Figure 6.5: Example illustrating crash failure

6.2.9 Time synchronisation failure state

The time synchronisation failure is achieved by configuring other failure modes to target PCF frames. TRAITOR is equipped with the ability to target the individual traffic classes of TTEthernet. Therefore, in this state, the user targets the PCF frames for corruption, omission, delay, masquerading failure, and babbling idiot. The PCF frames are injected with errors to produces the corresponding failures mentioned so as to observe the operation of the synchronisation mechanism of TTEThernet under the presence of failures.

6.2.10 Stuck-At failure

Although stuck-At failures refer to when a system fails such that a bit is stuck at "high" or "low", the implementation of babbling idiot failure is modified in this work for stuck-At failure, where frame repetition still follows the network schedule but repeats the transmission of a specific frame. Therefore, stuck-at failure is implemented in the context of being stuck at a frame rather than a bit.

6.3 TSN implementation

The same state machines and algorithms used in the implementation of TTEthernet was used for the TSN. Nevertheless, although TSN and TTEThernet have the same frame structure extracted from standard Ethernet, both networks translate and handle the Ethernet frame fields in different ways.

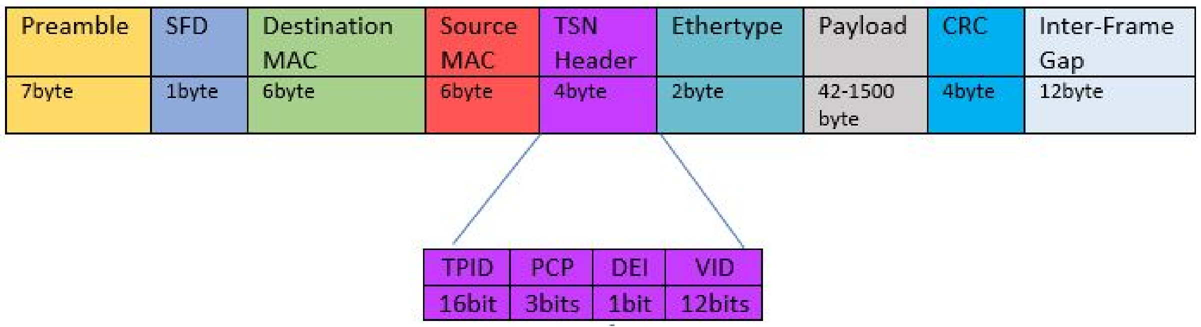


Figure 6.6: 802.1Q frame structure

The implementation of TTEthernet divides the destination address field into two parts; *constant field* and the Virtual Link Identification (VLID) field. The destination address is composed of 6 bytes (i.e. 48 bits). The first 32 bits of the destination address are used for the *constant field* while the next 16 bits are used for the VLID. The *constant field* is utilised by the TTEthernet to differentiate between BE effort frames from both TT and RC traffic. The participating devices in the TTEthernet network use the *constant field* bits to classify a frame as time-triggered or rate-constrained when the *constant field* in the Destination Address is set to a configured value. All other traffic that does not have the configured value is seen as BE frames.

In TSN, the destination address does not serve any purpose different from its usage in standard Ethernet. However, a significant difference in the whole frame is the usage of the Ethernet's 802.1Q optional tag in the TSN network. The IEEE 802.1Q frame consists of a starting 8-byte signal (7-bytes for preamble and 1-byte for Start Frame Delimiter), 6-byte for destination MAC address, 6-byte for source MAC address, 4 bytes of 802.1Q optional tag, 2-byte for Ethernet Type/Size, 46 to 1500-bytes for payload, 4-byte for Cyclic Redundancy Check (CRC) or Frame Check Sequence (FCS). The 802.1Q frames use a 32-bit optional header that is located between the source MAC address and the Ethernet Type/Size headers. It consists of a 16-bit for Tag Protocol Identifier (TPID), 3 bits Priority code point (PCP), 1-bit Drop eligible Indicator (DEI), and 12 bits VLAN identifier (VID). The 802.1Q frames structure is shown in Figure 6.6.

The FI designed in this work reads the frame contents with a nibble resolution. In the TSN mode, the TSN header that is shown in Figure 6.6 is used to identify the TSN frames for fault injection. In the TTEthernet mode, the Destination address is used to identify the TTEthernet frames for fault injection. The above explanation addresses the operational difference of how TTEthernet and TSN fault injection are implemented in TRAITOR.

Chapter 7

Experiments, Results and Analysis

This chapter discusses and demonstrates the applicability of TRAITOR to various use-cases, showing how TRAITOR satisfies its requirements. Four sets of experiments are carried out, each satisfying a given set of experimental goals described in section 7.1. Section 7.2 demonstrates the use of TRAITOR in TTEthernet fault injection, showing the realisation of the fundamental techniques discussed in Table 5.1 of chapter 5. Section 7.3, shows the use of TRAITOR in evaluating the fault containment of TTEthernet against babbling idiot failure. Section 7.4 demonstrates the use of TRAITOR in evaluating a deterministic protocol in the railway domain and compares the performance with a non-deterministic protocol. Section 7.5 demonstrates the potential of TRAITOR in generating data that can be used in training a neural network to facilitate the early detection of faults, especially during the synchronisation startup phase of time-triggered Ethernet systems. Finally, the summary of the experiments is given in section 7.6.

7.1 Experiment goals

The first goal of the experiments conducted herein is to test the implementation of the foundational techniques for TRAITOR, including the control of the fault injection instant and its ability to target individual traffic classes. Several requirements of TRAITOR were discussed in chapter 4, these were labelled *requirement 1* to *requirement 7*. The second goal of the experiments conducted in this work is to evaluate these requirements. The experiments will demonstrate how TRAITOR satisfies the following:

Requirement 1 - The applicability of TRAITOR in evaluating Time-triggered network protocols.

Requirement 2 - The use of TRAITOR to evaluate the integrated behaviour of time-triggered network protocols such as TTEthernet and TSN.

Requirement 3 - The use of TRAITOR in evaluating a physical prototype or final implementation of time-triggered network components.

Requirement 4 - To demonstrate the low intrusiveness of TRAITOR.

Requirement 5 - To demonstrate the separation of TRAITOR from the NUT.

Requirement 6 - To demonstrate the portability of TRAITOR to different time-triggered protocols or Ethernet-based protocols such as TTEthernet and TSN.

Requirement 7 - Evaluate the use of TRAITOR in generating data to train a neural network for failure detection.

Scalability - To demonstrate the scalability of TRAITOR.

Applicability to different topologies - To show that TRAITOR can be applied to validating applications and protocols irrespective of the network topology. Thereby establishing that TRAITOR is topology independent.

7.2 TTEthernet fault injection

Most fault injection techniques for network protocols integrate the fault injection component into the NUT. As earlier discussed in Chapter 4, this requires the modification of the hardware/software of the NUT component. These modifications restrict the FI implementation to specific application setup, such that for every use-case one would require to modify the NUT component to perform fault injections. However, since the method developed in this work abstracts the FI from both the end systems and switches, its advantage is the provision of the generic capability to TRAITOR to target different use cases. Also, the possibility of introducing unintended errors into the switches or end system due to modifications is eliminated. As reported in Chapter 5, the abstraction is attained by placing the FI component following a cut-through paradigm, where network faults are injected on the link between the end systems and switches.

This section reports the result obtained from the development of TRAITOR. Herein, the fundamental techniques (i.e. byte injection, frame delay, and frame jamming) are tested to observe the operation of TRAITOR. The correctness of TRAITOR's operation is manually observed in this first version of experiments. The aim of the experiment is to demonstrate the capability of the framework in achieving the following:

1. To test and demonstrate TRAITOR's foundational techniques: byte injection, frame

delay, and frame jamming.

2. To test and control the injection instant.
3. To ensure that the framework can target the individual traffic classes of a time-triggered Ethernet network protocol. This is to show the ability of TRAITOR to impact individual protocol algorithms. For instance, to impact on the synchronisation mechanism of TTEthernet, TRAITOR must be able to inject faults on the PCF frames. However, this first experiment demonstrates the ability of TRAITOR to target TT, RC and BE frames.

7.2.1 Usecase description and experiment setup

The experiment is performed on the single-hop non-redundant configuration shown in Figure 7.1. In the figure, two end systems (ES_1 and ES_2) are connected via a TTEthernet switch (SW). The SW is assigned the role of a CM, while ES_1 and ES_2 are assigned the role of an SM. Mixed-criticality traffic (i.e. the mix of TT, RC, and BE messages) is configured for transmission between ES_1 and ES_2 on a 100 Mbits/s link. A parallel network consisting of three passive network taps is set up to measure latencies, jitter and dropped packets in the network. The network taps are represented as NT_1, NT_2 and NT_3 in Figure 7.1. All TTEthernet network traffic in this setup is sniffed by the network taps and received at the monitoring station. The fault injection component of TRAITOR is represented as FI in the figure, and it implements the algorithms and state machines described in Chapter 6.

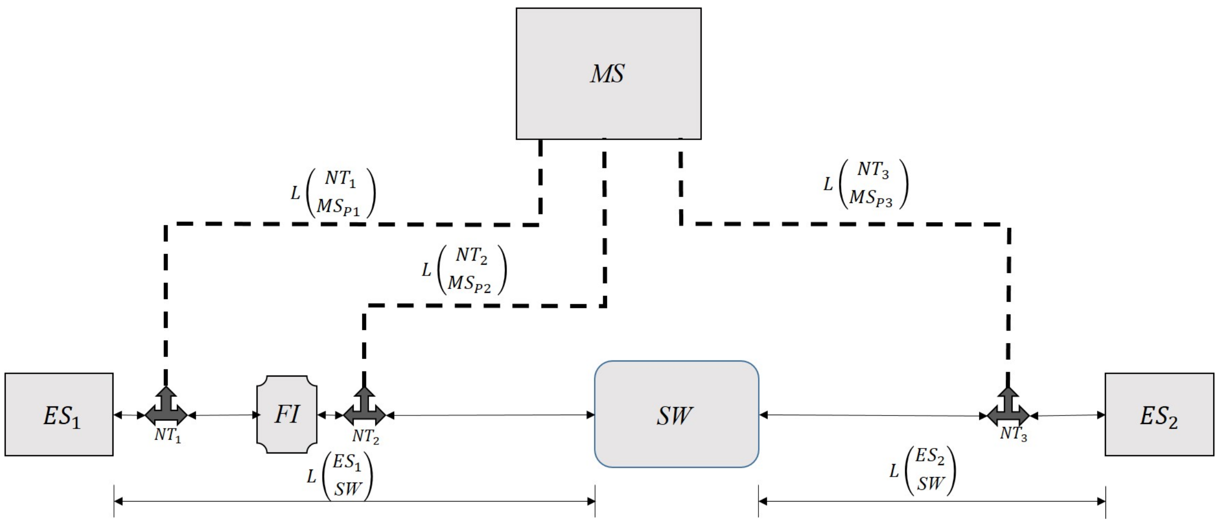


Figure 7.1: Single Hop Non-redundant network connected to a monitoring station

Delay, corruption and omission failure were injected individually into the TTEthernet network targeting TT, RC, and BE traffic classes respectively. The FI component is responsible for the injection. The impact of the FI can be observed using the three network TAPS NT_1, NT_2 and NT_3. Since there are TAPs located before and after the FI component, the action of the FI can be observed directly.

7.2.2 Results and discussions

Corruption failure is first injected, and the results are illustrated in Table 7.1. The table illustrates the results of injecting corruption fault with a FER (Frame error rate) of 3. The FER indicates the frequency at which corruption fault was applied. FER of 3 corrupts every 3rd frame. Therefore, it is expected that the switch drops every corrupted frame indicated by the CRC computation. The experiment was conducted for TT, RC, and BE traffic, to demonstrate the frameworks capability to target individual traffic classes. Table 7.1 shows that out of a 110 observed TT traffic, a FER = 3 resulted in 35 injections which are approximately one-third of the total observed frame. In the case of RC traffic, the number of injected fault is equal to 34, which is exactly one-third of the total number of observed frames. The total number of observed frames in BE is equal to 105, and the number of injected faults obtained is 36, which is also approximately one-third of the total number of frames. The values obtained can be confirmed from the number of received packets registered by the network Tap NT_3. The transmission of frames sent by ES_1 is registered by NT_1, and the reception is registered by NT_3. NT_1 and NT_3 provides a way to obtain the number of drop packets. Latency and jitter can also be computed from the timestamps obtained from NT_1 and NT_3.

The frames sniffed by NT_1, NT_2 and NT_3 are fed into the high resolution NAC card described in Chapter 5. The results are further analysed using Tshark. The monitoring network introduces no significant delay to the NUT since passive network TAPs are utilised.

Frame Type	Total no. of observed frames	No. of Injected Faults	No. of Frame Received
TT	110	35	75
RC	102	34	68
BE	105	36	69

Table 7.1: Results of Corruption Failure (FER = 3)

The results for omission failure injection show that all omitted frames are not perceived by the switch. Table 7.2 illustrates the impact of omission failure. The results indicate that the omitted frames were not forwarded to the switch and receiving end system. A FER = 3 was also configured for omission failure. Out of 120 transmitted TT message injected with omission failure, 40 omission failures were observed, and 80 frames were received. Out of 117 RC messages injected with omission failure, 39 frames were omitted, and 78 frames received, and out of 114 BE messages injected with omission failure, 39 frames are omitted, and 75 frames received. Approximately one-third of all frames are omitted. The reason for the approximate value stems from the time when the observation is started. However, the sum of the number of injected faults and the number of received frame produces accurately the total number of frames observed. Therefore, the injection is consistent for both corruption failure and omission failure.

Frame Type	Total no. of observed frames	No. of Injected Faults	No. of Frame Received
TT	120	40	80
RC	117	39	78
BE	114	39	75

Table 7.2: Results of Omission Failure (FER = 3)

Xilinx provides ways to probe the operation of custom IPs developed using the Vivado platform. One of such ways is the use of the Integrated Logic Analyser (ILA). The ILA can be used to probe the operation of a custom IP at run-time. The ILA is used herein to illustrate the outcome of delay failure on the TTEthernet network. A snippet of the ILA is shown in Figure 7.2 and Figure 7.3 to illustrate the injection of delay failure.

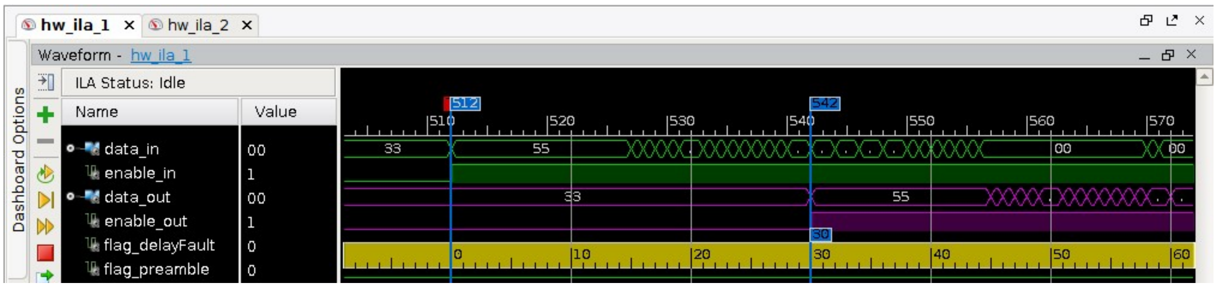


Figure 7.2: ILA illustration of Golden run

The ILA is configured to capture frames with a frequency of 25 MHz. 25 MHz corresponds to the 100 Mbit/s clock setting required for the GMII-RGMII block. The ILA captured the inverted nibble format of transmission of the 100 Mbit/s network, as shown in Figure 7.2 and Figure 7.3. Every byte transmitted on the 100 Mbit/s settings through the GMII-RGMII block is split into two nibbles, and the less significant nibble is transmitted

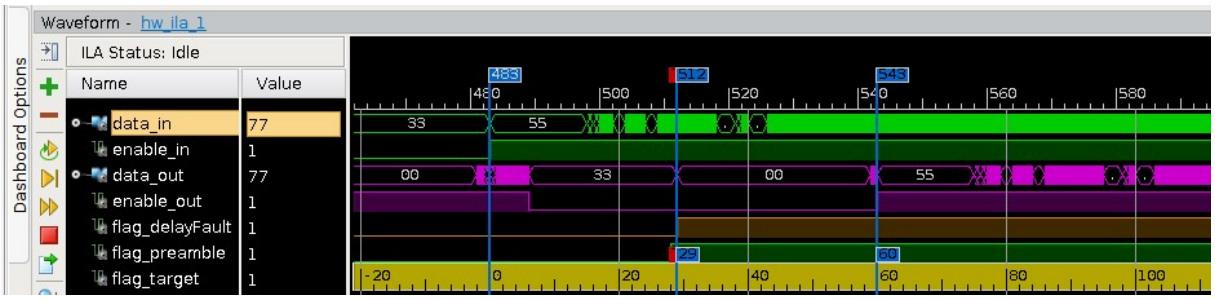


Figure 7.3: ILA illustration for delay fault of 1µs

before the higher significant nibble. Figure 7.2 captures the golden run scenario when no fault is activated. The field *data_in* and *data_out* shows the nibble signals received via the ingress port of the FI, and the nibbles signals that leave the FI via the egress, respectively. The *enable_in* signal is an RGMII enable signals which travel in parallel with the frames to indicate the start and end of a frame. The *enable_out* shows when and how the *enable_in* signal leaves the FI. $Flag\_delayFault$ is a flag that shows when delay failure is triggered, that is the instant when the value of FIRE = 5 (see Figure 6.3). The $flag\_preamble$ is a flag that shows when the FI detects the start of a frame. The $flag\_trarget$ is a flag that shows the instant when the target frame is acquired to begin fault injection.

In the early version of TRAITOR, the cut-through paradigm introduced a fixed 30 clock cycle computational delay, as shown in Figure 7.2. The delay is the processing delay of the FI, which accounts for the time taken to read the TTEthernet frame header. In figure 7.2, two blue flags carrying the value "512", and "542" are shown. The flags are used to illustrate the shift register delay in the golden run scenario. It shows the beginning of the frame and the point when the frame leaves the FI. It takes 30 clock cycles for this configuration. The inverted nibbles of the preamble and SFD make up the first 16 clock cycle. The inverted nibbles of the destination address make up the next 12 clock cycle. After which it takes the FI one clock cycle to analyse the shift register content and another clock cycle to output the frame.

Figure 7.3 is used to illustrate when a $1\mu s$ delay is injected into the TT frame. In the figure, three blue flags carrying the value "483", "512" and "543" are shown. The first value "483" shows the entry point of the frame into the FI. The value "512" indicates the instance the delay fault is activated. The value "543" indicates the point when the frame exits the egress port of the FI. The difference between the point of entry of frames and the time delay fault was activated is 29 clock cycles. The difference between the point of injection and the time the frame exits the FPGA is 31 clock cycle. The 31 clock cycle is how long it takes to implement the configured delay value. The injected 1µs delay is equivalent to a 25 clock cycles delay (Note: each nibble is clocked every 40 ns). Five clock cycles are taken by the buffer to store the frames for a delay operation. The sum of the five clock cycle buffer operation, 25 clock cycle delay and one clock cycle instance for outputting the data are equal to the 31 clock cycle calculated from the ILA in Figure 7.3.

7.2.3 Conclusion

The first overall experimental goal of this work is met in this section 7.2 by demonstrating TRAITOR's capability in targeting the individual traffic classes of the TTEthernet network. It also demonstrated the use of the framework and the correct operation of the FER. Corruption, omission and delay failure produced the expected impact when injected into the TTEthernet network. TRAITOR was also able to inject specific delay faults targeting TTEthernet's traffic classes. *Requirement 1* and *requirement 2* are also met in that TRAITOR is applied to the TTEthernet network such that it captures the integrated behaviour of all its services (e.g. synchronisation service). *Requirement 3* is met in that the validation was performed on physical devices. The experiment also shows that the fault injection component is abstracted from the components of the TTEthernet, thus satisfying *requirement 5*.

7.3 Fault containment against babbling idiot failure

Reliability measurement relies on controlled fault injection experiments that can observe the behaviour of a system under the effects of faults. Fault injection thereby provides the platform for assessing fault containment, testing error handling and tolerance of a system and assessing solutions to improve dependability. Injecting babbling idiot failures (BIFs) can be used to aid the evaluation of the fault containment of a network. A babbling idiot is characterised by the transmission of messages at an arbitrary timing. A faulty node that monopolises the common channel by sending messages at erroneous points in time is perceived to have a babbling idiot failure [Wang et al., 2009].

The babbling idiot failure mode has the potential to cause a complete system failure by disrupting communication between end systems operating correctly. A babbling idiot phenomenon can impede the fulfilment of real-time and safety criticality requirements by delaying or causing the loss of other messages. In safety criticality systems BIFs could result in catastrophic consequences.

BIFs are caused primarily by the node and can originate either from a node's hardware or software. A hardware babbling-idiot occurs when the failure is caused by the direct consequence of a hardware fault. A software babbling-idiot takes place when the fault originates from the application software (e.g. a bug in the code or human factor such as a malicious attack). The TTEthernet protocol takes advantage of channel redundancy capability to overcome a babbling idiot failure. In channel redundancy, the deterministic transmission of messages on the replica is used to override the hardware babbling-idiot fault [Buja et al., 2005]. TTEthernet also uses the bus guardian system in which it uses prior knowledge about the permitted temporal behaviour to block untimely messages. Nevertheless, in this section, the effect that a babbling idiot failure of a link has on other links is observed. This section focuses on evaluating the fault containment of TTEthernet against BIF.

7.3.1 Usecase description and experiment Setup

The experiment observed the behaviour of a TTEthernet switch network connected with four end systems using a 100Mbits/s link. The TTTECH A664 LAB devices are used for this experiment. Figure 7.4 illustrates the network under test, which is configured with five virtual links (VL1-VL5). VL1 and VL2 are configured to transmit TT traffic while VL3 to VL5 are configured to transmit RC traffic.

In Figure 7.4, the direction and type of traffic on the VLs are illustrated with the dotted arrows. The arrows associated with each virtual link point in the direction of the traffic. Messages originate from a single sender but can be received by multiple receivers, as shown in the case of VL1 and VL2. Note that the VLs are logical connections; the physical connections are shown in the figure with non-dotted blue lines. Four end systems configured as synchronisation masters are connected to one TTEthernet switch which is configured as the compression master. Passive network taps are placed on all the physical links in positions A, B, C and D of Figure 7.4. The network taps are connected to a central monitoring station, similar to the setup in Figure 7.1. All traffic through the links is sniffed in a passive manner using these network taps.

Similar to the experiment in the previous section, the monitoring station is equipped with COTS NACs that receive all traffic from the network taps. Tshark tool is used to collect all the captured packets. Computations of latency and jitter of different virtual links are handled offline via a custom C++ program, designed specifically for analysing TTEthernet traffic.

The FPGA is used to inject untimely messages on VL1 (TT) and VL4 (RC) with a total frame size of 165 bytes. Best effort messages are also injected in the background according to a random interval. The scenario in Figure 7.4 is such that the participation of ES1 and its message transmission is emulated to generate a BIF. On startup, the FPGA acts

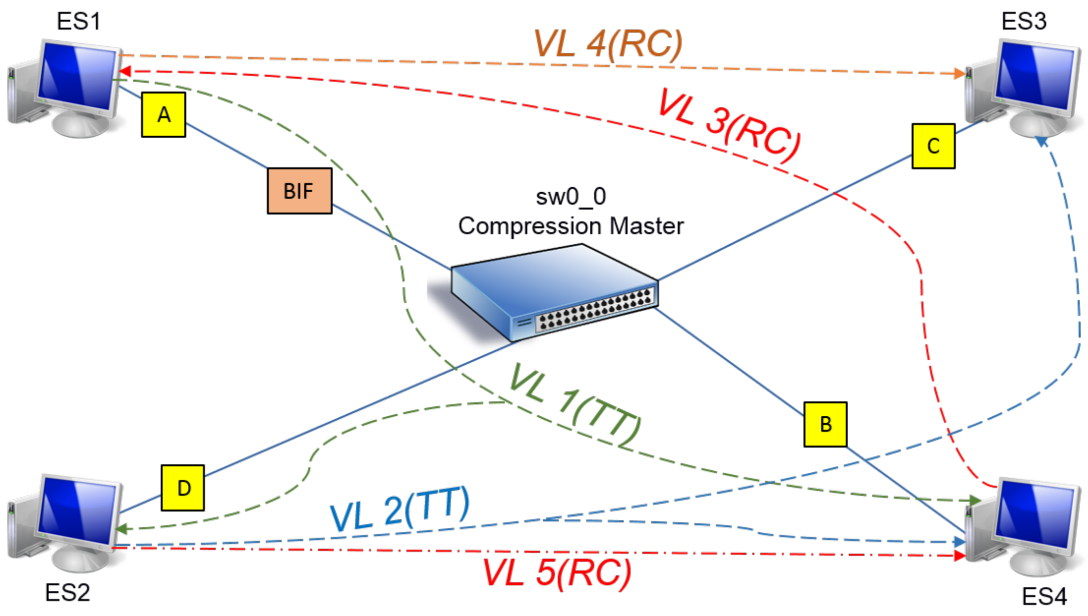


Figure 7.4: Experimental setup for Babbling Idiot Fault injection...

completely as a cut-through interface. This cut-through approach introduces a constant delay of 2 μs to the usual traffic over the channel. When the BIF is activated, ES1 is cutoff and messages from it are emulated on the FPGA, except that these messages take an untimely interval, emulating the babbling idiot failure. Information regarding the virtual links of ES1 is pre-loaded into the FPGA configuration before the start of the experiment.

The frame size and Bandwidth allocation gap (BAG) for each application running on the virtual links are configured for the experiments;

- with a payload size of 100 bytes.
- with a BAG of 15000 μs.

The above-listed configurations are set on each VL. Firstly we obtained latency, jitter and percentage of frame loss values for a Golden Run (GR) scenario. These initially measured GR values are used to establish a baseline that makes it possible to ascertain the effect of faults on the network when BIF is injected. The effect of BIF on VL2, VL3 and VL5 are observed by measuring the latency, jitter and frame loss on these VLs.

7.3.2 Results and discussions

The experimental observation for frame loss, average latency and Jitter on VL2, VL3, and VL5 are shown in Table 7.3. The average latency jitter and frame loss values for the

	VL	**Avg.Latency**	**max. Jitter**	**Frame Loss**
GR	VL2(ES2-ES3)	204.626 μs	1 μs	0
BIF	VL2(ES2-ES3)	210.96 μs	89 μs	2
GR	VL2(ES2-ES4)	204.659 μs	1 μs	0
BIF	VL2(ES2-ES4)	210.498 μs	88 μs	3
GR	VL3(ES4-ES1)	19.7907 μs	8 μs	0
BIF	VL3(ES4-ES1)	19.9522 μs	17 μs	2
GR	VL5(ES2-ES4)	19.7551 μs	7 μs	0
BIF	VL5(ES2-ES4)	28.9759 μs	181 μs	1

Table 7.3: Experimental Measurement - Golden run scenario and BIF injected with a payload size(100B) and BAG (0.015s)

GR scenario are first recorded before the activation of BIF on VL1 and VL4.

Average latency

The experiments considered latency in the context of end to end delay between two nodes. It expresses the duration taken by frames to get from one end system to another end system. The network schedule was configured with a period of 20 ms for TT traffic, and a BAG of 8 ms for RC traffic. The setup operates on a full-duplex communication, the VL arrows in Figure 7.4 illustrates clearly the direction of traffic. VL1 and VL4 are not observed since it is linked to the source of failure occurrence. The idea behind the experiment is to ascertain if the babbling idiot failure is contained when the messages arrive at the switch. The VLs outside the containment region include VL2, VL3, and VL5. To observe end to end latency on VL2, the network TAP combination (D and C), and (D and B) are used. The network tap combination is because, on VL2, ES2 transmits TT messages to both ES3 and ES4. Therefore, Table 7.3 captures two measurement path for VL2. VL3 and VL5 are both configured to transmit RC traffic. RC messages are transmitted on VL3 from ES4 to ES1, and on VL5 from ES2 to ES4.

When BIF is activated, the average latency of frames on VL2 (TT) is slightly increased. For the RC VLs, the increase in the average latency of frames when BIF is triggered is more pronounced compared to TT VL. For example, these differences in average latencies are 6.334 μs and 9.2208 μs for TT VL(ES2-ES3) and RC VL(VL2-ES4) respectively. VL3 is not significantly affected by the BIF because there is no contention with any message in the transmission direction (see VL3 (RC) in Figure 7.4).

Jitter

Increased jitter was observed across all VLs when BIF is activated. Under GR scenario, the jitter observed across TT VLs was bounded at 1μs. It is important to know what conflict resolution (between TT and other traffic) method is implemented on the switch, to accomplish a detailed evaluation of the effect of BIF on TTEthernet network. The conflict resolution strategy has an impact on the handling of traffic priority by the switch. The switch in this experiment utilised the shuffling mechanism [AS6802, 2011] for conflict resolution. Due to the store-and-forward principle and non-fragmented frame transmission implemented by the shuffling mechanism, TT traffic can be delayed by the RC traffic resulting in a pronounced jitter. However, this jitter is bounded as it does not exceed the transmission duration for a single frame.

BIF was injected on VL1 (TT) and VL4 (RC) with a frame size of 165 bytes. In a 100Mbit/s link, a byte is sampled in a bridge every 80 ns. Therefore the switch takes at least 13.2μs in this experiment to relay a complete BIF message. Since shuffling is implemented in the switch, if there is ongoing transmission of an RC message by the switch when a TT frame arrives, the TT frame is delayed by 13.2μs plus the IFG and switch processing time. The shuffling method is a trade-off between optimal bandwidth utilisation and real-time quality. The observed bounded jitter degrades Real-time quality. However, bandwidth is utilised efficiently, as there are no truncated frames. Compared to TT VL, the jitter on RC VL5 is much more significant. The buffer size specification of the switch plays a major role in how RC messages are handled. Flooding the switch with BIF has more significant consequences as the buffer limit is exceeded, resulting in more pronounced jitter for RC traffic.

Figure 7.5 and Figure 7.6 illustrate a plot on the number of frame samples against latencies observed in the experiment for VL2 (ES2-ES3) and VL2 (ES2-ES4), respectively. The plot for the GR (i.e. the observation when no fault is injected) is represented in orange, and the plot for the BIF is represented in blue. It can be observed that the jitter is not more than 1μs for VL2. However, notice the spread on both Figures (7.5 and 7.6) when BIF is triggered. The jitter on VL2 becomes 89μs in direction ES2 to ES3 and 88μs in the direction ES2 to ES4.

The deduced jitter from the plot is bounded within a maximum of 89 μs for ES2-ES3 and 88 μs for ES2-ES4. The RC VLs illustrated in Figure 7.7 direction ES4 to ES1, and Figure 7.8 in direction ES2-ES4 are not only affected by the frame size of the BIF message. They are also affected by the frequency of frame generation and the size of the configured switch buffer. As a result, the jitter observed for the RC VL was disproportionate. It can be observed from Figure 7.7 that the jitter experienced by VL3 in direction ES4 to ES1 when BIF is triggered results to 17 μs, and this is because there is no competition between VL3 and other VLs in this direction. However, the jitter on VL5 is significantly affected by BIF failure. VL5 has a jitter value up to 181μs when BIF is triggered. The significant increase in jitter is due to the buffer operation of the switch in handling RC

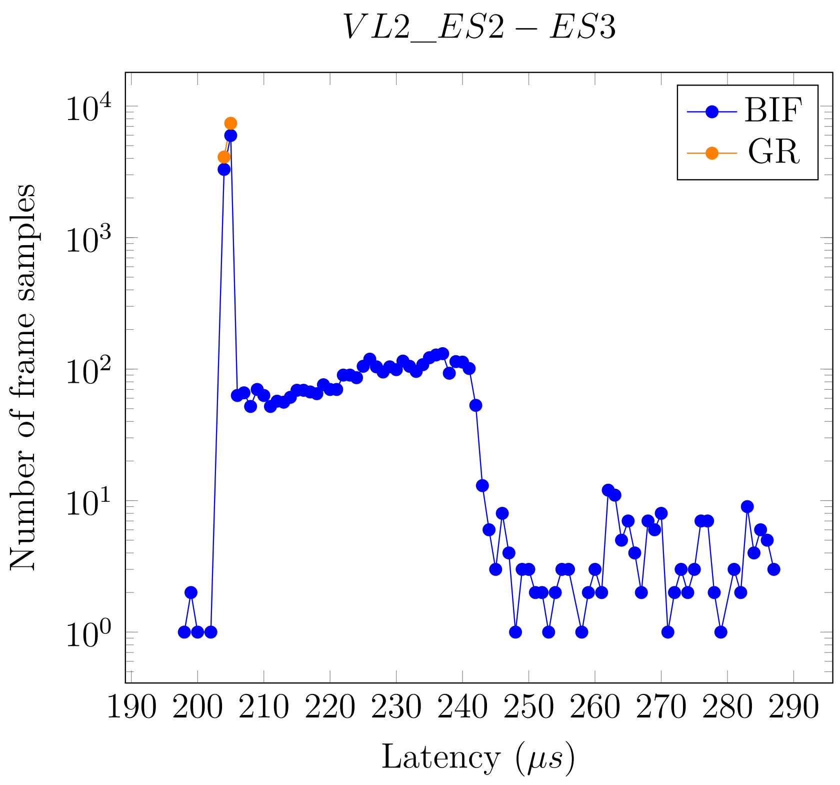


Figure 7.5: Virtual Link 2 (End system 2 to End system 3) observation for Jitter.

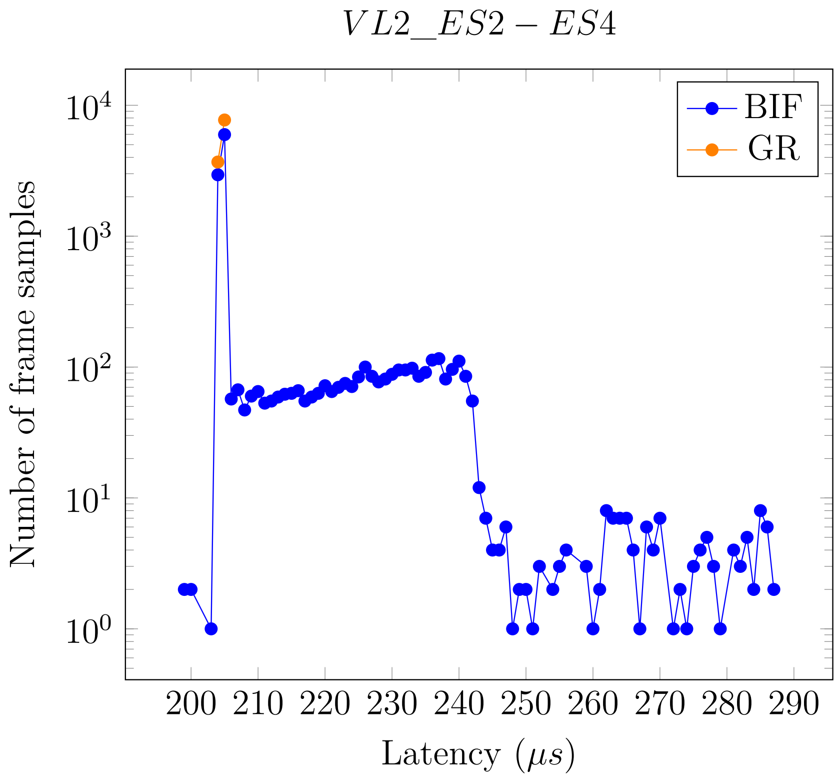


Figure 7.6: Virtual Link 2 (End system 2 to End system 4) observation for Jitter.

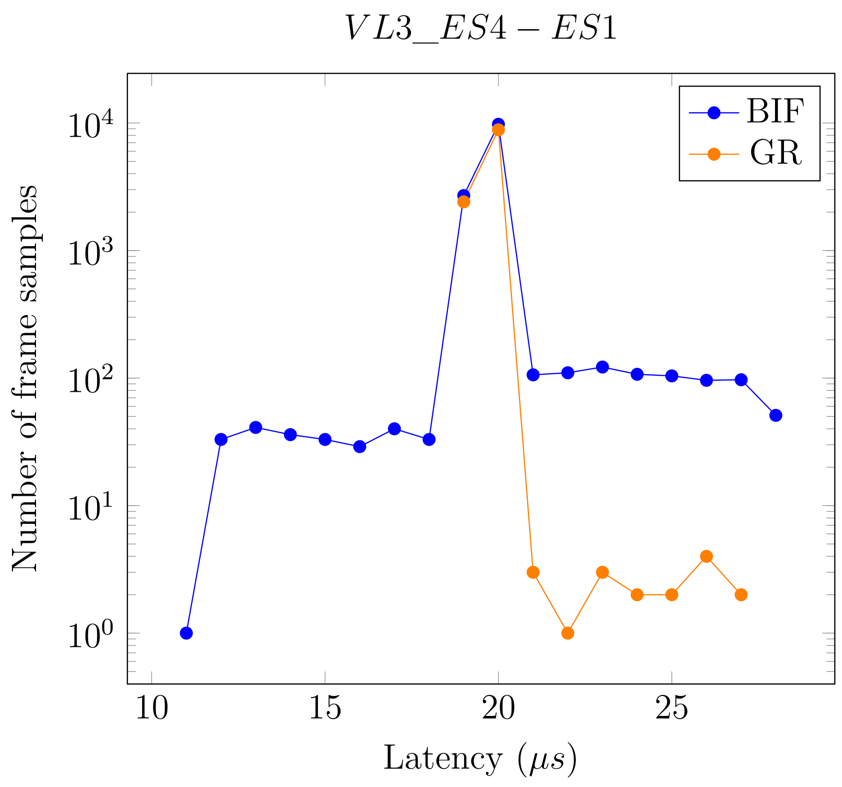


Figure 7.7: Virtual Link 3 (End system 4 to End system 1) observation for Jitter.

traffic.

Frame loss

Frame loss was recorded for both TT and RC traffic under BIF. The arrival of frames to a receiving end system outside an acceptance window that was specified a priori are discarded. Also, frames that did not meet the bandwidth allocation gap (BAG) requirement of ARINC 664 were discarded. Both RC VLs and TT VLs were prone to frame loss under BIF. The frame loss recorded from the experiment is recorded in Table 7.3.

7.3.3 Conclusion

TRAITOR proved significant in evaluating the fault containment of TTEthernet implementations against babbling idiot failures. Experimental results obtained demonstrated the use of TRAITOR in providing statistically useful data for the evaluation of TTEthernet fault containment against babbling idiot failures.

TRAITOR successfully provided a platform to observe the impact of BIF on RC and TT communication. It was observed that an end system which generates babbling idiot messages on a given virtual link could have an effect of on jitter (i.e. a jitter up to

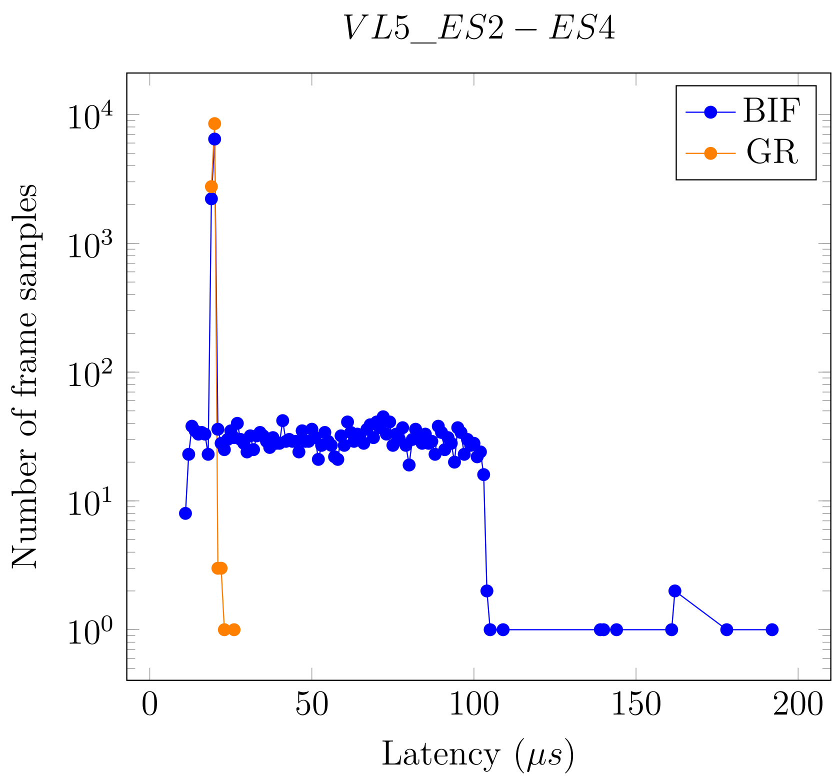


Figure 7.8: Virtual Link 5 (End system 2 to End system 4) observation for Jitter.

123 μs for maximum frame size) on a TT virtual link sharing the same switch due to shuffling. Shuffling thus affects the fault containment of TT communication under the presence of BIF. If fault containment is considered without the need for trade-offs, the timely block mechanism [Obermaisser, 2011, pp 192] is a recommended option to decrease the jitter for TT communication. TRAITOR thus provides the platform to apply direct measurement necessary for quantification of availability and reliability of TTEthernet implementation from different vendors. *Requirement 1* and *requirement 2* of the overall goal are shown in that TRAITOR is used here to evaluate the fault tolerance mechanism of TTEthernet's protection against BIF which encompasses the integrated behaviour of the connected components. *Requirement 3* is also shown since the experiment is performed on the physical implementation of TTEthernet. TRAITOR is also abstracted from the TTEthernet network, showing the fulfilment of *requirement 5*. TRAITOR was applied to a star topology in this section 7.3, and consisted of more TTEthernet network participants than the experiment carried out in section 7.2. The experiment, therefore, demonstrated the scalability and applicability of TRAITOR to different topologies.

7.4 Deterministic communication in the railway domain

The current electronic systems found in trains consists of applications running at different safety-criticality levels. Such applications include propulsion control, door controls, diagnostics, seat reservations, and so on. These applications host multiple embedded devices situated in a distributed manner in the train vehicles. A real-time communication system is necessary to ensure efficient communication between these embedded devices, hence the development of the Train Communication Network (TCN) standard [Kirrmann and Zuber, 2001], which provides the guideline for exchanging information between devices on railway vehicles. The establishment of a common standard for the electronic coupling of the vehicle's electronic equipment was the motivating factor behind the TCN development [Kirrmann and Zuber, 2001]. Before the development of this standard, the railway communication system was mostly a proprietary solution and tailored to meet specific requirements of railway operators and vendors. TCN functions on a hierarchical model consisting of two network buses, the Wire Train Bus (WTB) and the Multifunction vehicle bus (MVB). The initial version of the TCN standard (IEC 61375-1, IEC 61375-2-1ff) did not take care of applications having large bandwidth, such as surveillance cameras and on-board entertainment. The data rate of this earlier TCN version tops at 1 Mbit/s between the train vehicle (Train bus) and 1.5Mbit/s within a vehicle (Vehicle bus). Therefore, this resulted in the development of additional parts to the IEC 61375-family. The improved standard defined a faster TCN (IEC61375-2-5) capable of handling over 100 Mbits/s, and conformant to IEEE 802.3. This standard is referred herein as the ETB/ECN and is based on a hierarchical model, comprising the Ethernet Train Backbone (ETB) and the Ethernet consist network (ECN).

However, the railway industry still uses the WTB/MVB for safety functions, such as traction control, brakes, and doors. While the ETB/ECN bus is used for high bandwidth functions such as for passenger's comfort (entertainment) and passenger information (e.g. audio message) [Jakovljevic et al., 2017], this presents a mixed network with two backbones, one for ETB/ECN and the other WTB/MVB. Apart from the increase in complexity brought by this solution, there is an extended cable length and an increase in maintenance cost resulting from two different network topologies. Recently, a European railway project titled "SAFE4RAIL" [SAFE4RAIL-1, 2019] began to address these problems. SAFE4RAIL proposed the use of a deterministic platform for the train communication network. The determinism and bounded jitter feature of certain networks termed "deterministic" provide a way to support mixed-criticality traffic. It is expected that future communication system in the railway industry will utilise this mixed-criticality feature of deterministic networks to reduce wiring/weight, complexity, and maintenance cost. Deterministic platforms such as TSN [Farkas et al., 2018] and TTEthernet [AS6802, 2011] were considered during the SAFE4RAIL project. This is due to the anticipated worldwide acceptance, integrability and its cost-effectiveness, which are critical considerations for the railway vendors. The proposed integrated modular platform for the train control and management system (TCMS) in SAFE4RAIL was based on such a deterministic communication platform.

This section evaluates the application performance of TCN-TTEThernet (i.e. TCN based on a TTEthernet profile). The results of the evaluation herein provide a means to compare the performance of TTEthernet to other deterministic and non-deterministic platforms quantitatively. The work herein evaluates the application behaviour in the presence of network faults, to aid the implementation of adequate fault coverage. Both TCN-ETB/ECN (TCN based on ETB/ECN profile) and TCN-TTEthernet are observed under the BIF. TRAITOR provides the platform to evaluate the actual hardware behaviour in the presence of faults as opposed to evaluation using models provided in mathematical or formal methods. For this reason, TRAITOR is used to evaluate the performance of the implemented train communication system. The framework can thus be used, to test hardware implementation of TCN based on several deterministic platforms, to ensure conformance to standards, and to verify proper functionality.

This section aims at providing the following contributions:

- Provides the baseline for future comparison of TCN-TTEThernet with other deterministic networks.
- Exposes the impact of failure scenarios on the safety and performance of applications over TCN.
- Proposes and illustrates the use of TRAITOR for performance evaluation of TCN.
- Exposes the efficacy of TTEthernet as a suitable underlying network for TCN, specifically at layer 2 of the OSI stack.

The terminologies used to describe the train system in this section are illustrated in Figure 7.9. The *car* is the smallest unit from the physical composition. However, from a network perspective, multiple *cars* can constitute a *consist*. A train fleet is made up of multiple *consists*.

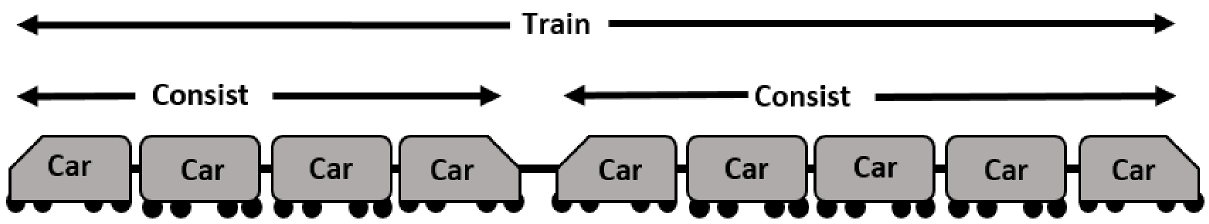


Figure 7.9: Train composition

7.4.1 Fault assumption

TTEtherent provides a means to configure the end nodes and switches with high integrity. The high-integrity design uses a commander/monitor (COM/MON) system to realize a fail-silent node [Obermaisser, 2011]. It is assumed that any message sent by a fail-silent node can only be a correct message. If we consider a TTEtherent switch in line with this assumption, it will certainly forward only the correct messages. In the case of an end system, which is the origin and sink of the message, failures can occur before it is fed into the COM/MON, for example, failures that occur in software. The COM/MON only aims for error-containment that occurs within a device. The TTEthernet uses a traffic policing mechanism and a central guardian system to protect itself against BIFs. However, we evaluate the implication of these mechanisms when hosting railway applications.

7.4.2 Usecase description

The communication infrastructure in the railway systems consists of multiple end devices (ECUs) that perform several operations, as illustrated in Figure 7.10. An ECN forms a unit that connects applications within a car. Although not captured in Figure 7.10, an ECN can consist of multiple cars that form a consist network. However, the diagram illustrates the hierarchical architecture of TCN, which is made up of the ETB and ECN network. The ETB connects multiple ECNs. The operator's cabin is positioned at the leading consist. The applications captured in the use-case have different levels of safety criticality classification. The applications considered include the following.

- Surveillance: The CCTV (closed-circuit television) cameras. This application has a high bandwidth requirement.
- The train braking system, considered in this work as safety-critical, with a higher priority than the surveillance system.
- Heating, Ventilation and Air Conditioning (HVAC) are considered in this work as safety-critical.

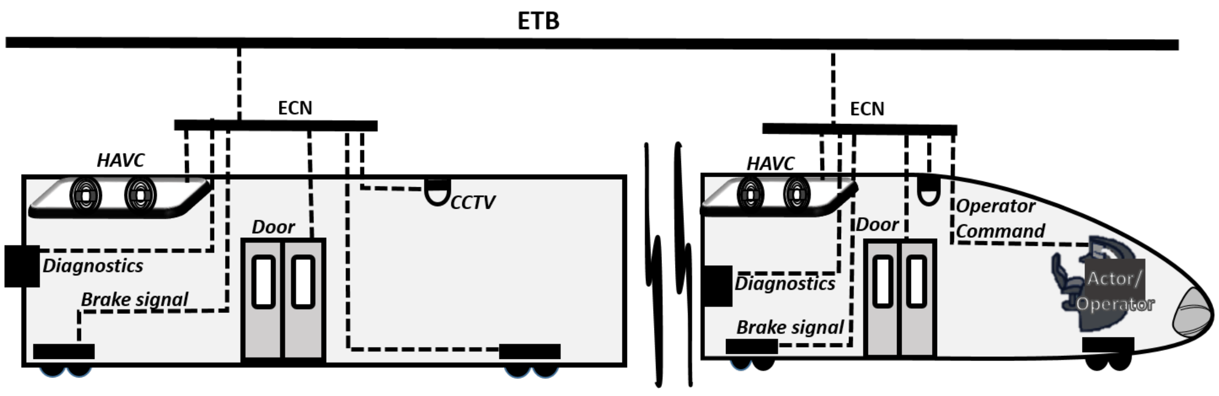


Figure 7.10: Train Consist network

- Diagnostic services, in this use case, are provided by each consist. The results are accumulated in the operator's cabin.
- Door Control: The use case includes ECUs (Electronic control units) for doors that are controlled by the operator's cabin.

This section shows the evaluation of the performance of these applications when hosted on the same network. Herein, observations are carried out to ascertain the added value of the temporal and spatial partitioning of the different application messages provided by TTEthernet. The TCN-ETB/ECN is also set up and evaluated as a baseline to understand the improvements given by TTEthernet.

7.4.3 Experiment setup

The analysis of a train topology comprising four *consist* networks connected via a redundant ETB line is carried out. Firstly, we set up a communication network between the *consists* using the existing TCN-ETB/ECN profile. Figure 7.11 shows the connections between the four *consists* network. The ETBN is a setup of switches implemented in an FPGA based board; NetLeap Base Board by novtech. The ECNs are also implemented on the Netleap boards to exchange traffic between each other. Traffic traverses the ECN nodes via redundant channels. The VLANs map to two different categories of network traffic, to provide the partitioning required for the different message types. Table 7.4 illustrates these mappings.

The VLANs are configured on different broadcast domains. Based on the IEEE 802.1Q, priorities were allotted as follows — priority level 7 for break and door control, and priority level 5 for all others. The column *message directon* illustrates the direction of messages on the ETBN channel; this is also reflected in Table 7.4. For instance, on VLAN1, the network ECN_4 transmits HVAC control commands to ECN_1, ECN_2, and ECN_3.

VLAN Name	**Message Direction**	**Application map**
VLAN1	ECN_4 → ECN_1,ECN_2,ECN_3	HVAC control
VLAN2	ECN_4 → ECN_1,ECN_2,ECN_3	Brake control
VLAN3	ECN_4 → ECN_1,ECN_2,ECN_3	Door control
VLAN4	ECN_1 → ECN_4	CCTV
VLAN5	ECN_2 → ECN_4	CCTV
VLAN6	ECN_3 → ECN_4	CCTV
VLAN7	ECN_3 → ECN_4	Diagnostic

Table 7.4: VLAN profile for TCN based on ETB/ECN

Secondly, a similar profile was further set up for TTEthernet. One major difference is the replacement of VLANs with Virtual Links. Virtual links are used to partition the different applications running on the network. A virtual link is of ARINC 664-part 7 [Part, 2006] origin and it is integrated as a part in the TTEthernet standard. The virtual link establishes a static path between one transmitter and one or several receivers. It specifies a logical communication path between one source application to one or more destination applications (*situtated in one or multiple nodes*). Table 7.5 illustrates the senders and receivers assigned following the virtual link concept. The TTE-Switch A664 Lab switches and end-systems by TTTECH were used for the ETBNs and ECNs, respectively. The four ECN nodes exchange messages between each other using these virtual links. The safety (e.g. Brakes), non-safety criticality (e.g. Diagnostics) messages and applications that require high QoS (e.g. CCTV) are assigned different virtual links. The assignments are based on the level of criticality. In TTEthernet, three message classes (TT, RC and BE) can be assigned to provide temporal and spatial partitioning of the messages exchanged between these applications. The setup demonstrates how rolling stock applications that have mixed-criticality levels could be hosted on the same network infrastructure and yet share a common link. Safety-critical applications such as the brake and door control were assigned higher priorities than lower criticality applications such as diagnostics and CCTV.

To observe the performance of TCN-ETB/ECN and TCN-TTEthernet under failure, ECN_1 was conditioned to fail in a manner to flood the network (i.e. BIF). The flooding was generated by the FI component implemented on the FPGA and connected on link L1 and L2 in Figure 7.11. Measurement probes were placed on both ports of ECN_2, ECN_3, and ECN_4 to obtain the end-to-end latencies. The entire network traffic was monitored using network accelerator cards (*NT4E2-4-PTP by Napatech*).

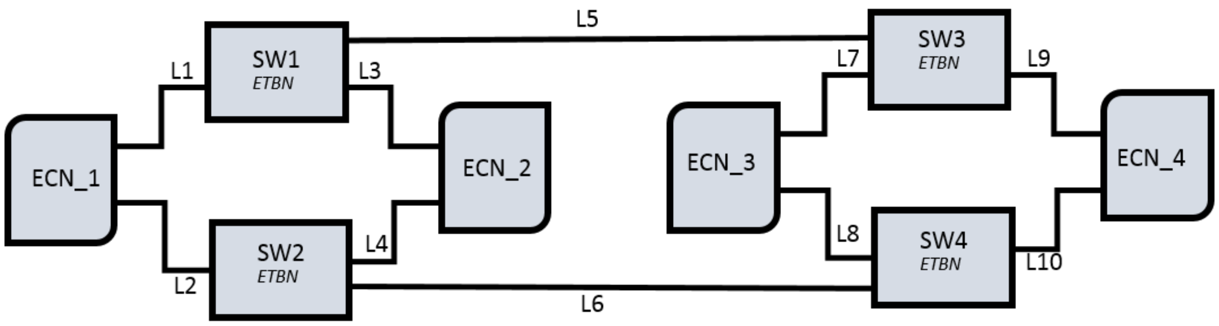


Figure 7.11: Train network setup based on TTEthernet

VL Name	**VL Direction**	**Application map**
VL1	ECN_4 → ECN_1,ECN_2,ECN_3	HVAC control
VL2	ECN_4 → ECN_1,ECN_2,ECN_3	Brake control
VL3	ECN_4 → ECN_1,ECN_2,ECN_3	Door control
VL4	ECN_1 → ECN_4	CCTV
VL5	ECN_2 → ECN_4	CCTV
VL6	ECN_3 → ECN_4	CCTV
VL7	ECN_3 → ECN_4	Diagnostics

Table 7.5: Virtual Link profile for TTEthernet use-case for Railway

7.4.4 Results and discussions

A C++ software is used to analyse the results obtained; to obtain the latency of individual VLANs/VLs, jitter and number of dropped packets. This work first investigated a *golden run* scenario, before running an investigation under the given failure scenario. It is used to obtain a baseline to compare and determine the effect of faults when injected.

Behavior of TCN-ETB/ECN

Table 7.6 shows the results obtained from a *golden run* experiment and an experiment under the network flooding scenario. The measurements are taken for the ECNs that were not conditioned to fail; ECN_2, ECN_3 and ECN_4. It can be observed from Table 7.6 that the average latency is reflected in the number of hops and distance between the ECNs as expected. In the *golden run* scenario, the average latency value of VLAN1(ECN4 - ECN2), VLAN2(ECN4 - ECN2) and VLAN3(ECN4 - ECN2) is 179.8 μs, while all VLANs from ECN4 to ECN3 is approximately 89.8μs. The values reflect the number of hops since ECN 2 is farther than ECN3 from ECN4.

VLAN[map]	**GR-Avg Latency** (µs)	**Failure-AvgLatency** (µs)	**GR-Jitter** (µs)	**Failure-Jitter** (µs)
VLAN1(ECN4 - ECN2)[1]	179.8568	179.6034	9.2200	8.7700
VLAN1(ECN4 - ECN3)[2]	89.8794	89.6724	9.2900	17.4000
VLAN2(ECN4 - ECN2)[3]	179.8549	179.6263	11.0100	8.1700
VLAN2(ECN4 - ECN3)[4]	89.8779	89.5754	10.8200	8.6900
VLAN3(ECN4 - ECN2)[5]	179.8529	179.5749	7.6500	8.7100
VLAN3(ECN4 - ECN3)[6]	89.8832	89.6052	9.1800	8.6600
VLAN5(ECN2 - ECN4)[7]	179.8053	210.2321	8.5900	87.5600
VLAN6(ECN3 - ECN4)[8]	89.8675	148.5840	0.3400	167.4300
VLAN7(ECN3 - ECN4)[9]	89.8684	125.5891	0.3400	162.5200

Table 7.6: Experiment results for TCN-ETB/ECN

Figure 7.12 illustrates the impact of BIF on the average latency. The VLANs in the plot are assigned a VLAN map number to indicate the direction and participants (i.e. the ECNs involved). For example, the plot represented with the number ***1*** on the x-axis, is used to represent VLAN1(ECN4 - ECN2) in Figure 7.12 which corresponds to the number shown in the first column of Table 7.6. It can be observed that VLAN5, VLAN6 and VLAN7, corresponds with VLAN mapping 7-9 respectively in the figure, and it is affected by the failure of ECN_1 (configured on *VLAN4*). The affected VLANs of the Ethernet full-duplex communication are the traffic sent in the same direction with the network flood. The average latency is increased due to the shared egress access of the

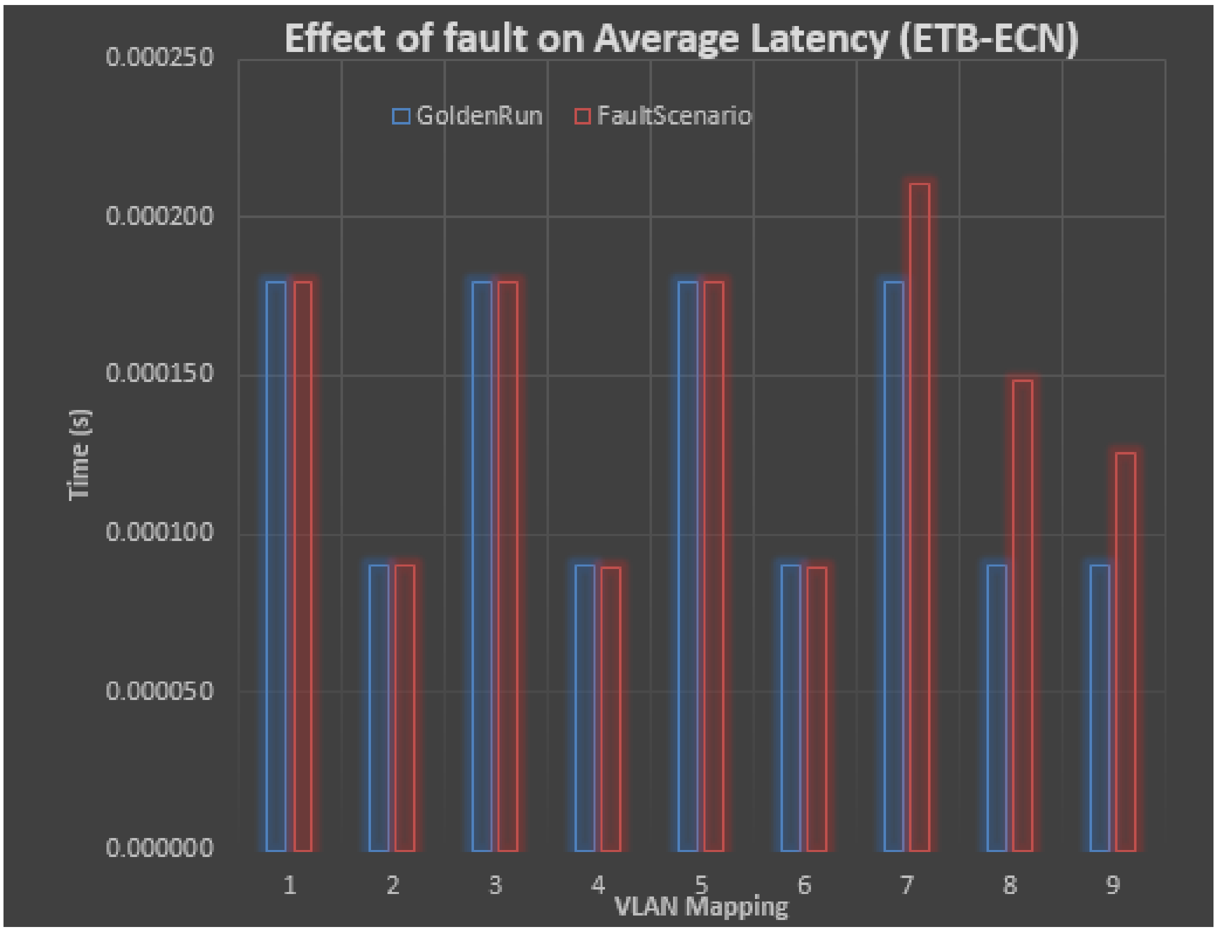


Figure 7.12: Effect of fault on Average Latency for TCN-ETB/ECN

switches.

The sporadic generated messages from ECN_1, emulated by the FI component, has an impact on all ETBNs. Thereby, increasing the average latency of VLAN5(ECN2 - ECN4), VLAN6(ECN3 - ECN4) and VLAN7(ECN3 - ECN4) by 30.4μs, 58.7μs and 35.72μs respectively. In this work, the transmission selection algorithm used is the strict priority. VLAN4 shares the same egress resource with VLAN5 on SW1 and SW2. It also shares the same egress resource with VLAN6 and VLAN7 on SW3 and SW4. The effect of networking flooding, therefore, increases the average latency on the affected VLANs and results in a significant jitter increase.

The effect of the failure on jitter is shown in Figure 7.13. Similar to the average latency, it can be seen that the VLAN5(ECN2 - ECN4), VLAN6(ECN3 - ECN4), and VLAN7(ECN3 - ECN4) are profoundly affected, with a jitter increase of 78μs, 167μs, and 162.2μs respectively.

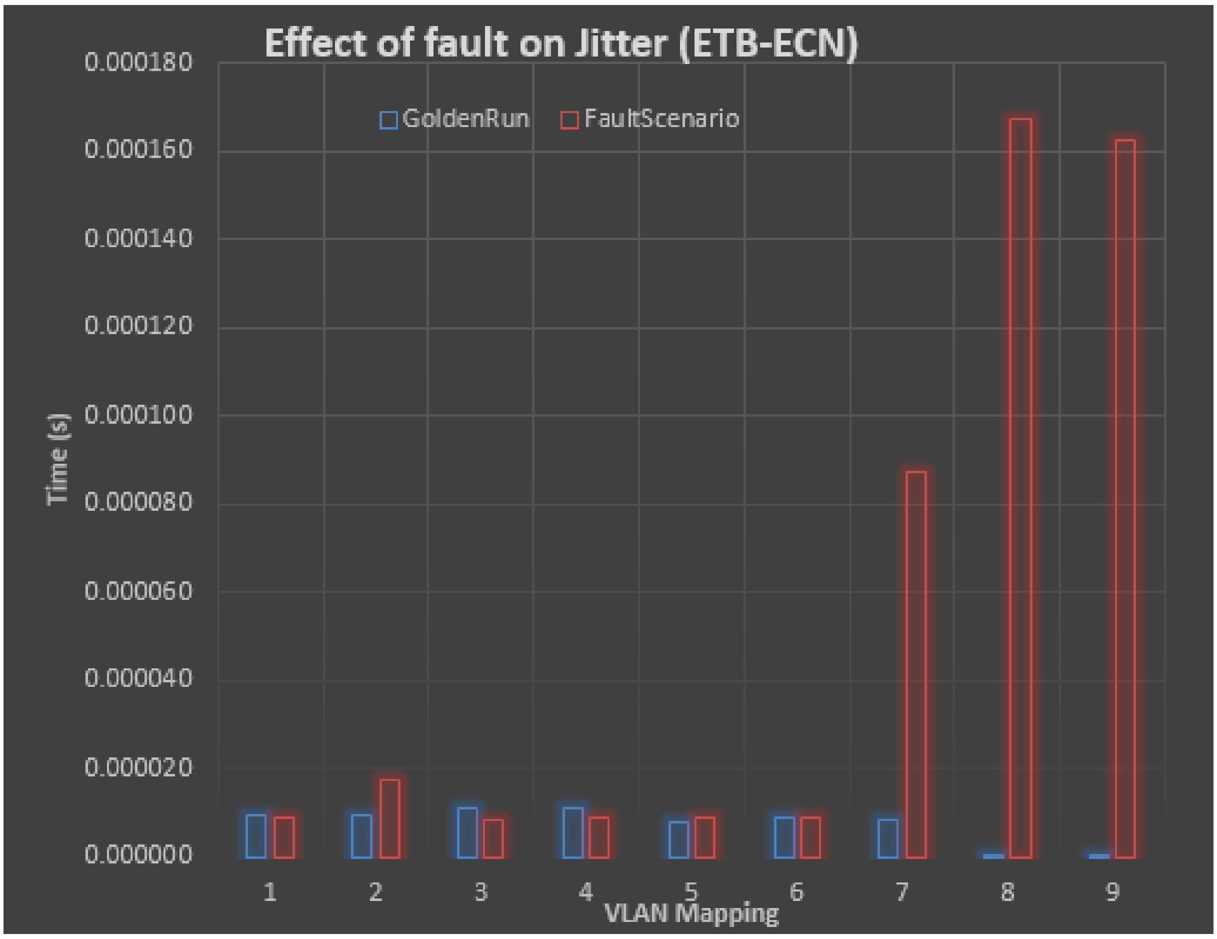


Figure 7.13: Effect of fault on Jitter for TCN-ETB/ECN

Behaviour of TCN-TTEthernet

The train use case in this work observed RC behaviour on TTEthernet. TT behaviour is minimally affected by the network flooding, as every VL is assigned a time slot that provides temporal isolation. TT traffic can only be delayed for a maximum of one message length, when *shuffling* [Obermaisser, 2011] is used for conflict resolution. The TT delay inflicted by shuffling is due to the concept of not having fragmented packets, so lower priority traffic is allowed to finish transmission even if the time slot allocated to TT traffic is reached. However, it remains a design decision whether to halt transmission of low priority traffic on the arrival of TT traffic.

Table 7.7 shows the results obtained with an underlying TTEthernet platform. The first column represents the virtual link members and the direction of traffic. The rest of the column headers have similar explanations to Table 7.6. In the case of TTEthernet, only one virtual link is significantly affected; VL5(ECN2 - ECN4)[7]. VL5 is in the same direction as VL4 and share common ETBN switches (SW1 and SW2). Since VL4 and VL5 are both RC messages, the sporadic generation of VL4 messages affects VL5, but not beyond the redundant ETBN switch-pair SW1 and SW2. Although VL6 and VL7 are in the same direction and share the same egress port on SW3 and SW4, they remain unaffected by the failure due to the frame filtering and policing performed by AFDX compliant switches (ARINC 664 switches).

VL[map]	**GR-Avg Latency** (µs)	**Failure-AvgLatency** (µs)	**GR-Jitter** (µs)	**Failure-Jitter** (µs)
VL1(ECN4 - ECN2)[1]	183.8353	183.5187	81.3000	14.6800
VL1(ECN4 - ECN3)[2]	91.9437	91.7744	81.2900	14.6500
VL2(ECN4 - ECN2)[3]	602.7124	602.7246	0.5400	7.6000
VL2(ECN4 - ECN3)[4]	203.0437	201.6934	0.5800	7.5700
VL3(ECN4 - ECN2)[5]	602.6584	602.7232	157.4700	7.1300
VL3(ECN4 - ECN3)[6]	203.0677	201.6906	160.3500	7.0500
VL5(ECN2 - ECN4)[7]	183.5132	27253.3649	6.7900	4997.7000
VL6(ECN3 - ECN4)[8]	91.7610	92.3172	4.8400	78.4000
VL7(ECN3 - ECN4)[9]	91.7655	92.0790	6.8900	84.0300

Table 7.7: Experiment results for TCN-TTEthernet

Figure 7.14 further illustrates the impact of the failure on the average latency of VL5 (VL mapping representation on chart no. 7). The average latency is increased by 27 ms. Figure 7.15 shows the effect of failure on Jitter. Similar to its average latency, only VL5 is significantly affected. The jitter value is increased considerably by 5 ms as RC messages Queue at the egress of SW1 and SW2. The network failure injected via VL4 creates

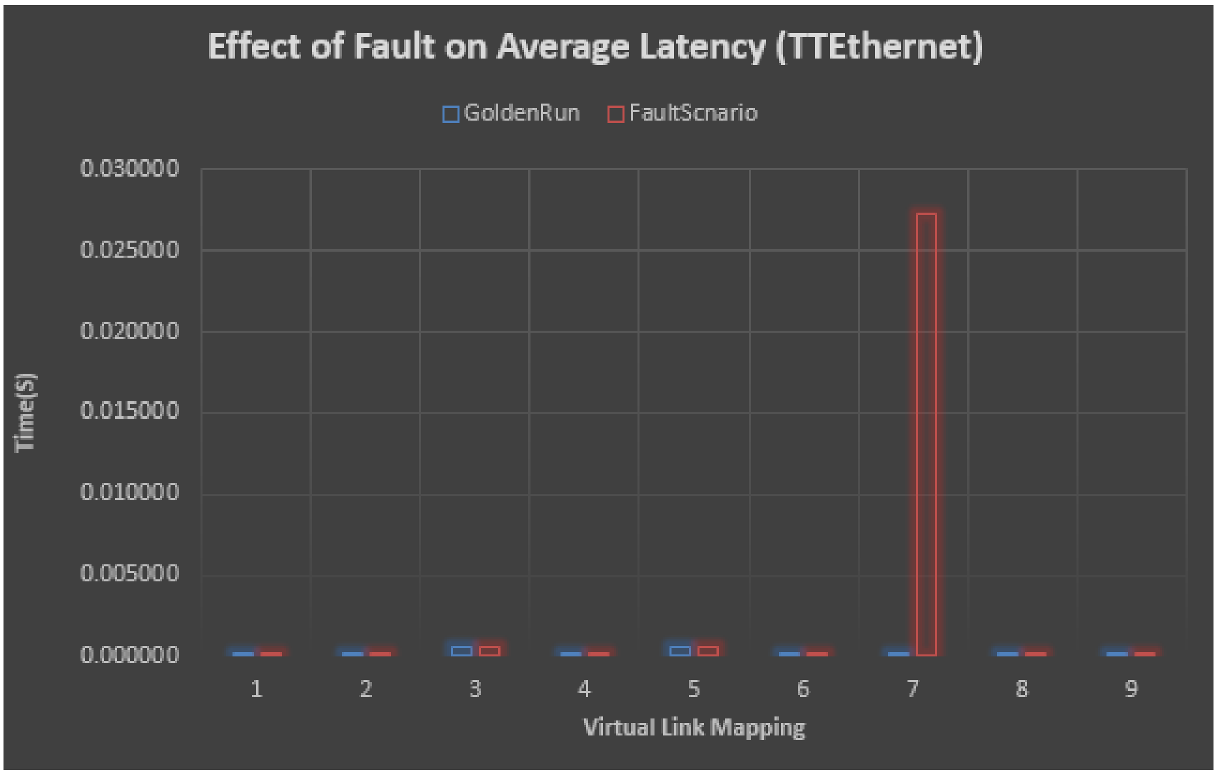


Figure 7.14: Effect of fault on Average Latency for TCN-TTEthernet

contention on the shared resource (egress port of SW1 and SW2). ARINC 664-Part 7 defines two priorities for virtual links: *high* or *low*. In this work, the priority of VL4 and VL5 is set to *high*. Unlike TT traffic, the RC traffic does not share any notion of global time and are not synchronised to the other network participants (network switches and end systems). Therefore, frames from VL4 and VL5 that ingresses SW1 and SW2, and then egresses via the same output port compete. The data contention issue between VL4 and VL5 (RC messages) are addressed through buffering, which is a technique specified by ARINC 664-Part 7. It is then in the part of designers to ensure that delays imposed by the frames (even in such a network flooding failure mode) is lower than the accepted delay for a given virtual link and that there is no frame loss as a result of buffer overflow. A detailed study on delay encountered by frame traversing an ARINC 664-Part 7 compliant switch and the techniques used to ascertain the worst-case delays can be found in [Coelho, 2017]. However, this work further pushes the need for critical consideration of failure cases, and its impact on the worst-case delay.

7.4.5 Conclusion

The impact of failure on TCN for mixed-criticality applications is analysed in this section. TTEtherent is further explored as the underlying platform for TCN, and its behaviour

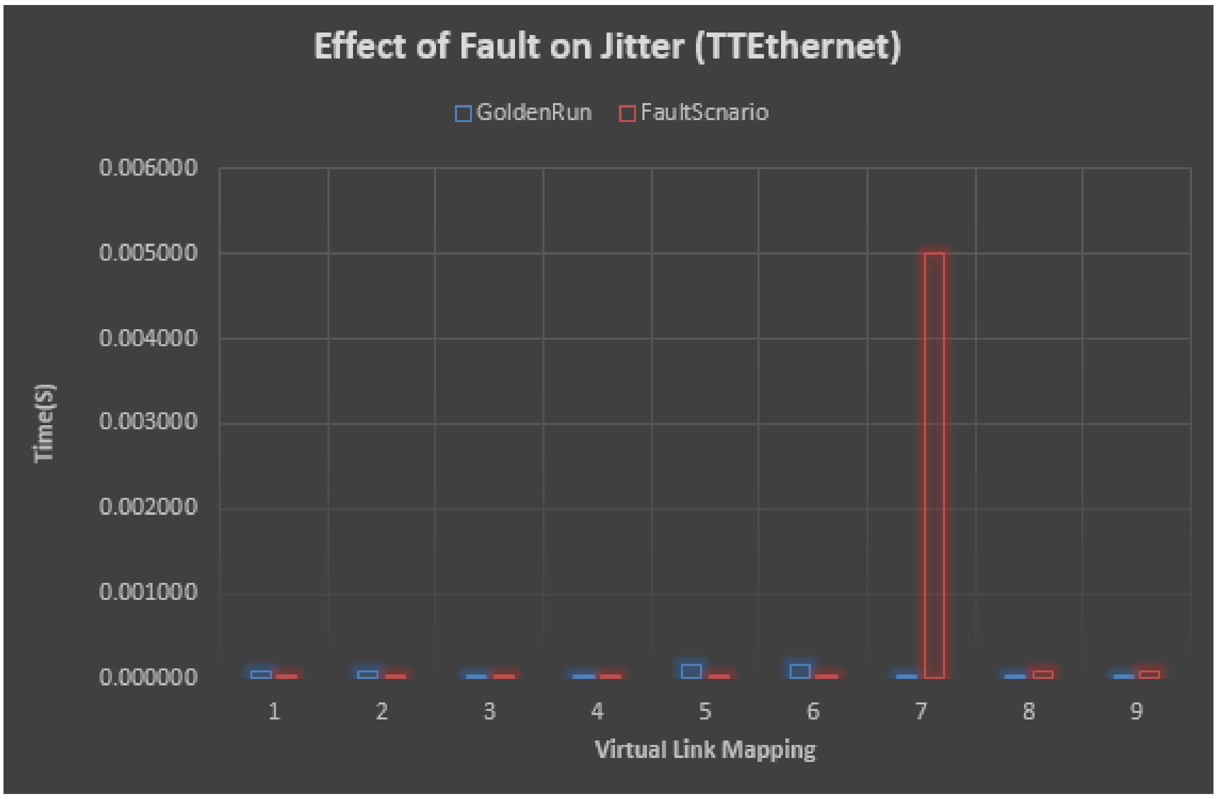


Figure 7.15: Effect of fault on Jitter for TCN-TTEthernet

evaluated under the presence of BIF. The effect of BIF on the performance of TCN based on ETB/ECN and TCN based on TTEthernet is evaluated. In the case of TTEthernet, only the latency and jitter of RC virtual link which shares the same egress resource with the injected failure messages are affected. Although the effect was profound, it was as a result of the buffering mechanism used in conflict resolution. In the case of standard Ethernet used in ETB/ECN specification, although logical isolation using VLANs were used in the experiment, the failure effects on latency and jitter were not contained in the first set of switches in which the frames traverses. Fault containment is appealing to safety criticality applications. Therefore, TTEthenet is a suitable candidate for the underlying platform for TCN.

The experiment in this section 7.4 demonstrates the use of TRAITOR for a realistic use case (railway use case). The number of network participant in this section 7.4 is larger than that of section 7.3, further highlighting the scalability of TRAITOR, and its applicability to different network topologies. The scalability property of TRAITOR is also shown from the increased network size. As a consequence, the observation probes of TRAITOR are increased to capture data from more network component of the TTEthernet setup compared to section 7.2. *Requirement 1*, *requirement 2*, and *requirement 3* are demonstrated as TRAITOR targets the integrated behaviour of a physical setup of

TTEthernet network. *Requirement 5* is demonstrated from the abstraction of TRAITOR from the NUT. The experiment also shows the portability of TRAITOR (*requirement 6*), where TRAITOR is also applied both for TCN based on ETB/ECN and TCN based on TTEthernet.

7.5 Failure detection in TSN startup using deep learning

The determinism offered by the time-triggered protocols is desirable for safety criticality systems that must meet hard deadlines. The primary purpose of periodic bandwidth allotment for message transmission is to ensure that there is no transmission conflict for critical messages. The timing of transmissions and receptions of critical messages is predictable, hence deterministic. At the core of the time-triggered network is the *synchronisation* mechanism. The synchronisation mechanism ensures that all connected network nodes have a common notion of time.

Time-triggered networks will fail to execute correctly if the synchronisation mechanism fails. In particular, the correct execution of the synchronisation mechanism is essential during the startup process of time-triggered systems. This is why time-triggered protocols such as the TTEthernet implemented a set of fault-tolerant protocols for clock synchronisation during startup [Obermaisser, 2011]. Some time-triggered protocols use dedicated startup algorithms subdivided into "coldstart" and "integration" as discussed in Chapter 3. Coldstart describes the synchronisation phase, where the network participants negotiate/select and agree to an initial synchronisation point. Integration describes the procedure used by the network participants whose startup is late to join an existing synchronisation.

In TSN, a revised version of IEEE 802.1AS is used for the optimisation of the distributed time synchronisation. IEEE 802.1AS implements an algorithm known as the Best Master Clock Algorithm (BMCA) to select a grandmaster clock from which other network participants will draw their reference time. A peer-delay mechanism is also used to measure path delays. However, diagnostic services to detect corruption or omission failures during startup are not defined within the scope of the standard. The manifestation of a corruption failure does not necessarily mean that messages will not satisfy the latency constraints of the peer-delay mechanism. The transmitted messages can be corrupted before the computation of the frame check sequence. It is a widespread practice to use the computation of checksums to identify corrupted frames. However, due to the regular re-computation of checksums by intermediate nodes when adding the residence time, it is even possible to compute new checksums using corrupted data. Mainly when the failure occurs internally between the time of frame reception by a network bridge and before checksum re-computation. In the case of omission failures, a priory knowledge can be used to detect when a message is not received at all or when a message is not received on schedule. Nevertheless, the global time that makes it possible to identify omission failures is only established after the startup process of the synchronisation mechanism. Therefore, it is not applicable to rely on the knowledge of the scheduler to identify omission failures during startup.

Herein, we introduce the use of deep learning to detect omission and corruption failures during startup. This is to avoid cases of perceiving a correct network synchronisation startup which has uncertainties and elements of non-determinism caused by the omission and corruption failures. The approach enriches the startup mechanism of TSN by applying deep learning techniques to detect network failures during the startup process. Deep learning is proposed for TSN nodes to capture failure. The safety requirements of safety criticality systems may be as high as SIL4 (Safety integrity level 4) [IEC61508, 2010], where field experience by itself is not sustainable for evaluation. Therefore, fault injection is used to accelerate the occurrence of faults to verify the functionality of fault-tolerant mechanisms or validate the system itself. Waiting to observe a single fault by field experience for a high criticality application could mean waiting for an infeasible number of years. Besides, far more than a single observation is required to establish reasonable confidence. Therefore, this work uses TRAITOR to generate the data required to train the neural network. This provides the ability to capture the behaviour of different communication fault profiles during startup.

7.5.1 Usecase description and experiment setup

A total of 6000 experimental runs were carried out. Two thousand experiments each for a golden run, corruption failure scenario, and omission failure scenario respectively. Corruption and omission failure is induced with a fault error rate (FER) of three; this means faults are injected in the system with a frequency of one fault in every three messages. The golden run scenario describes a case in which no fault is injected as the behaviour of the system is captured under the desired operating mode. Figure 7.16 illustrates the experimental setup used to acquire the data used to train the neural network. The setup represents a redundant architecture of a robot controller communicating with a set of sensors connected locally in ES1 and ES2. Actuators, not shown in the diagram, are connected through SW1 and SW2. A fault injector represented as *FI* is placed between

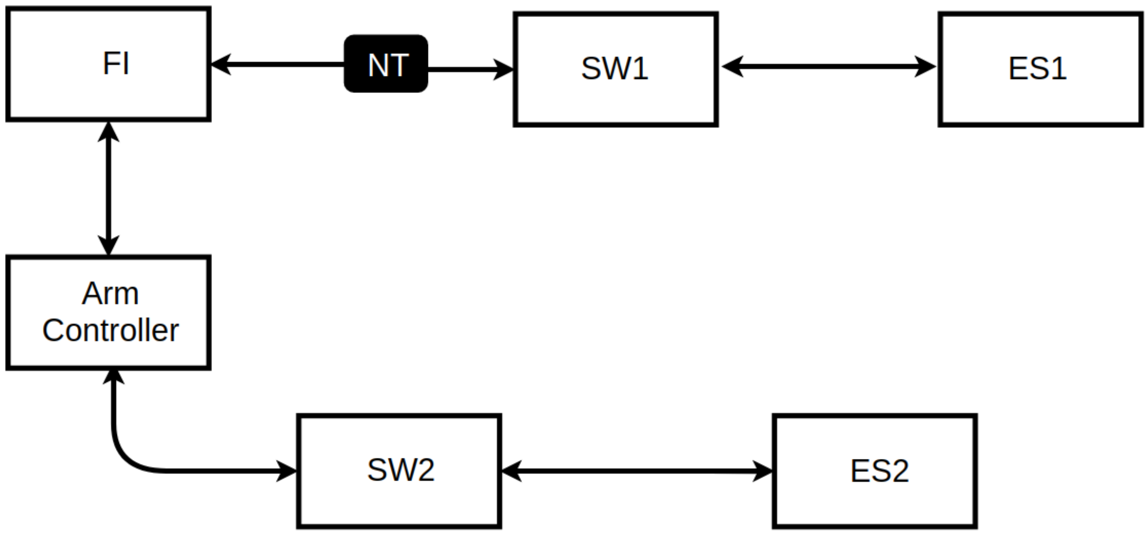


Figure 7.16: Experimental setup for data acquisition

the robotic controller and SW1 according to a cut-through paradigm to manipulate the data that traverses the network. The *FI* component only introduces a delay of 700 nanoseconds, making it possible for the ports of SW1 and the arm controller to be TSN capable ports. Figure 7.16 also shows a network tap device represented as *NT*. The network tap is used to sniff traffic in the network; this was designed using a Xilinx FPGA (Zedboard), which is the same board used to implement the *FI*.

The traffic sniffed by *NT* is captured by a NAC designed by Napatech. The packets are then saved using the Tshark application as comma-separated files (CSV) files for offline processing in Rstudio. The *time* vector (column) of the captured data represents the timestamps of message transmission and omission. This vector is used to train the neural network. The *time* vector is normalised to fall between the range of 0 to 1. After which 200 experiments are taking each from the set of 2000 experiments of the golden run, corruption and omission run. The 200 experiments are used as the testing data. The remaining 1800 experiments for the three cases are used to train the neural network. Finally, the testing set is introduced into the neural network for classification.

7.5.2 Results and discussions

The neural network is trained to classify three labels. These include the fault-free behaviour label (referred herein as golden run), corruption, and omission. An average accuracy above 99% was realised in the classification of test data to match the appropriate label in all three cases. The result of the experiments are shown in Figure 7.17.

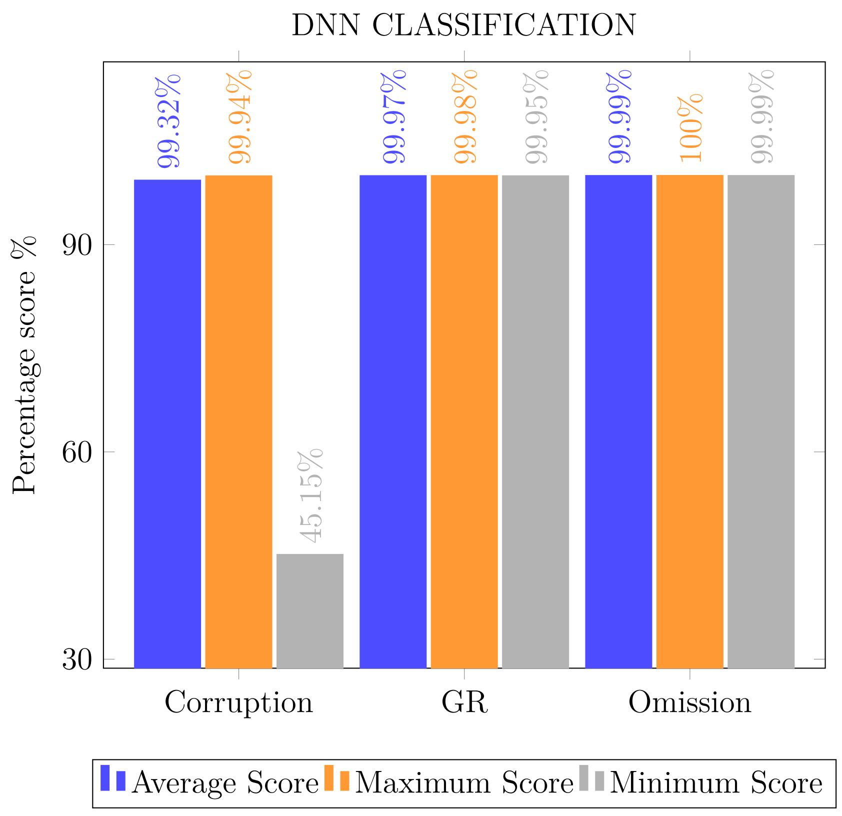


Figure 7.17: Results of neural network classification of failure.

The average accuracy obtained in classifying the set of 200 golden run test data is 99.97% with a minimum value of 99.95%. In the case of omission fault, the test data was classified with a score of 100%. However, in the case of corruption, 99.32% was obtained for the average accuracy. The minimum score is seen at 45.15%. Nevertheless, the plot in Figure 7.18 shows the individual score for each corruption fault experiment. In the experimental results obtained for corruption, it can be observed that only two cases out of the 200 experiments are below 95%: one is seen at 80% and the minimum at 45.15%. Nevertheless, the average accuracy still poses the potential to place strong confidence in the ability to classify the different failure types.

7.5.3 Conclusion

The work carried out in this section utilised TRAITOR to generate the training data required for a neural network that can be used to diagnose failures in the time-sensitive networking startup protocol. Three scenarios were considered: golden run, omission fault scenario, and corruption fault scenario. The deep learning procedure was able to classify the presented scenarios using only the behaviour pattern of the startup traffic captured in the *time* vector(i.e. a vector which contains the timestamp of transmitted and received

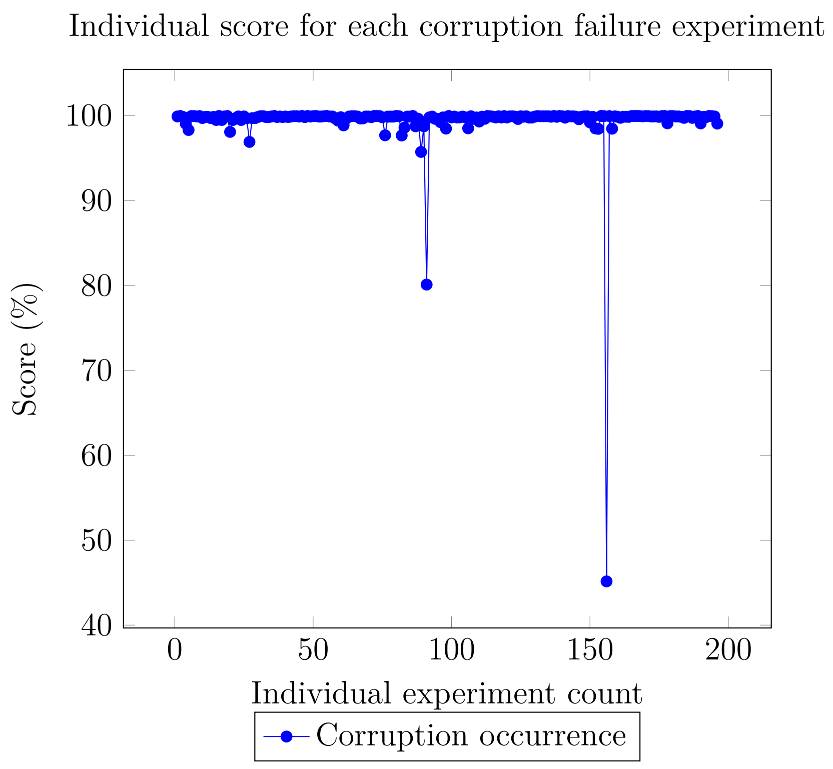


Figure 7.18: Results showing neural network score for corruption.

packets). The results obtained provides and demonstrates a useful approach to diagnose the health of time-sensitive networking devices during startup. This approach presents a solution to diagnosing failures that can occur before the run-time of the TDMA rounds. For a given safety criticality setup, gathering sufficient data for training is a challenge, TRAITOR solves this issue by accelerating the process of data generation.

The experiment carried out in this section shows the applicability of TRAITOR to a different use case (robotic arm). The TSN startup mechanism requires that the transmission delay between two connected ports must be $1\mu s$ for the ports to be capable of transmitting the synchronisation frames. TRAITOR introduced a delay of 700 nanoseconds, making it possible to be used for TSN startup. This shows the low intrusive capability of TRAITOR (*requirement 4*). The fault injection component of TRAITOR is abstracted from the network under test, thereby demonstrating *requirement 5*. TRAITOR is applied herein to TSN, which shows its portability to TSN networks, demonstrating *requirement 6*. Finally, TRAITOR also demonstrates how to generate sufficient data for training a neural network for failure detection, thereby fulfilling *requirement 7*.

7.6 Experiment summary

This chapter conducted series of experiments to fulfil the experiment goals described in section 7.1. Table 7.8 illustrates the various requirements and goals fulfilled in the different experiments organised by sections. The demonstration of scalability is fulfilled in section 7.3 and section 7.4. The network topology given in section 7.3, section 7.4, and section 7.5 are used do demonstrate the network topology independence of TRAITOR. Realistic examples citing a railway use case and a robotic arm used for medical surgery are accomplished in section 7.4 and section 7.5. Section 7.2, section 7.3, and section 7.4 are used to demonstrate *requirements 1-3*. Requirement 4 is demonstrated in section 7.5. Requirement 5 is demonstrated in all experiment. Requirement 6 is demonstrated in section 7.4 and section 7.5. Finally, requirement 7 is demonstrated in section 7.5.

	Section 7.2	**Section** 7.3	**Section** 7.4	**Section** 7.5
Scalability demo		✓	✓	
Demonstration of topology independence		✓	✓	✓
Realistic examples			✓	✓
Requirement 1	✓	✓	✓	
Requirement 2	✓	✓	✓	
Requirement 3	✓	✓	✓	
Requirement 4				✓
Requirement 5	✓	✓	✓	✓
Requirement 6			✓	✓
Requirement 7				✓

Table 7.8: Overview on the fulfilment of experimental goals

Chapter 8

Conclusion and Perspective

In this thesis, a cut-through fault injection framework named "TRAITOR" is developed for the dependability evaluation of time-triggered Ethernet network. Dependability evaluation is essential for safety-criticality applications widely adopted in various domains such as the health sector, military, and industrial applications. Several methods that can be used to verify and validate time-triggered Ethernet network protocols have been studied in this work. The need for the use of physical fault injection technique and why it presents a higher confidence level than other techniques (i.e. formal methods and simulation) is argued in this work. The argument presented herein was based on the gaps that exist between the dependability analysis using formal methods or simulation and by physical testing.

Such gaps include the inability to design a model that comprises all system functions due to the state-space explosion. Also in the case of simulation, it remains an abstraction of the real system, even though environmental factors are modelled, simulators still require parameters from prototype testing to tune the simulators or to confirm the fidelity of the simulator. Prototype testing such as physical fault injection is time-consuming and an expensive strategy. Nevertheless, for safety-criticality systems where the cost of failure or maintenance is higher than the developmental value, an evaluation framework for prototype testing is justifiable; hence the research and development of TRAITOR.

8.1 Features and advantages of TRAITOR

TRAITOR provides a cocktail of attributes that are directed towards the exhibition of low intrusiveness for the network under test. Low intrusiveness is attained by delivering the implementation of a TRAITOR at the Media Access Control (MAC) layer of the Ethernet protocol stack as opposed to implementations in the application layer. Therefore, the delay effect of computations by a processing system or the time taken to travel through the protocol stack is avoided. TRAITOR hence does not introduce high delay values that affect the operation of the target protocol. Also, the concept of abstracting the fault injector from the network components under test ensures that TRAITOR does not interfere with the operations of the network under test.

TRAITOR achieves portability by separating the fault injector components from the components of the network under test. This abstraction feature of TRAITOR is made possible by designing the fault injection component according to a cut-through paradigm to the network under test, as opposed to modifying the hardware elements of the network under test to achieve fault injection. The approach herein ensures that the normal operation of the network components remains unaffected without any form of intrusion. The cut-through has been achieved using a Xilinx FPGA where the fault injection is also activated.

The portability feature provided by TRAITOR makes it an application and vendor-independent framework. The operation of TRAITOR is not tied to any specific implementation of TTEthernet and TSN by a vendor. This work has shown TRAITOR to exhibit the capability to be used across multiple implementations of TTEthernet and TSN to verify and validate the protocol implementation and to validate applications running on top of these protocols. As TRAITOR is applicable and targeted towards prototypes/real systems, the results obtained from the framework satisfy the aim of having a framework that provides high confidence. In addition to the above-mentioned features, TRAITOR has been designed to show high observability using network test access points that can be placed in a distributed manner.

This work further pushes the bounds of TRAITOR to generate data that can be used to train a neural network to enable a node to posses fault detection capabilities. A series of experiments, discussions and results using TRAITOR were delivered herein, in which various use cases are shown. It can be seen therein that TRAITOR proved to satisfy the intended aim of this work.

8.2 Significance of TRAITOR

The implication of this work is that TRAITOR can be used by several vendors that build applications over time-triggered Ethernet networks for verification and validation of either the protocol or application. TRAITOR provides a means for vendors to observe the impact of faults on their design. TRAITOR can also be used to validate fault tolerance implementations, and to test the impact of faults on different configuration settings, especially to find optimal settings under different failure scenarios. The low intrusive feature of TRAITOR is particularly useful for testing over multiple applications without any design modification of the network components under test.

8.3 limitation

TRAITOR covers the failure modes in the IEC 61508 except for re-sequencing failure. Re-sequencing occurs when the frame order in a buffer-queue is misread. For example, let us consider a First-In-First-Out (FIFO) buffer implemented in a switch. Let's say a frame k enters the buffer at time t_{e1}, and is scheduled for dispatch at time t_{d1}, and another frame p enters at t_{e2} and is scheduled for dispatch at time t_{d2}. Re-sequencing failure is said to occur if at time t_{d1} frame p is sent to the egress port instead of frame k, and at time t_{d2} frame k is subsequently sent. Re-sequencing is a failure mode that occurs in the switch.

The reason for not implementing re-sequencing failure is that TRAITOR was aimed at abstracting the fault injector from the components of the network under test. Therefore, implementing re-sequencing will require modifying the switch component. An attempt to implement re-sequencing failure for TRAITOR will cause the unintentional introduction of an additional failure mode. If re-sequencing failure were to be implemented, it would require holding a frame in the FPGA for an amount of time. This time is equal to the maximum time it takes to transmit the bytes of the total frame plus the inter-frame gap between the frame and the next frame. It would then allow the next frame to pass through the cut-through FPGA, before releasing the frame that was held. This will introduce delay failure, which is not intended before performing re-sequencing failure. For this reason, re-sequencing failure is not integrated into TRAITOR.

8.4 Future work

The use of TRAITOR to generate data that describes the startup behaviour of time-triggered networks subjected to failure has been demonstrated herein. However, only two failure modes were covered: corruption and omission failure. It is intended to investigate other failure modes such as babbling idiot, delay, and crash failure. Currently, TRAITOR is implemented in the MAC layer of the protocol stack; it is also intended to exploit the applicability of TRAITOR in the physical layer of the protocol stack, thereby further reducing the delay introduced.

Bibliography

[Abuteir and Obermaisser, 2013] Abuteir, M. and Obermaisser, R. (2013). Simulation environment for time-triggered ethernet. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 642–648. IEEE.

[Aidemark et al., 2001] Aidemark, J., Vinter, J., Folkesson, P., and Karlsson, J. (2001). Goofi: Generic object-oriented fault injection tool. In *2001 International Conference on Dependable Systems and Networks*, pages 83–88. IEEE.

[Ammar and Mohamed, 2011] Ammar, M. and Mohamed, O. A. (2011). *Formal verification of Time-Triggered Ethernet protocol using PRISM model checker*. IEEE.

[Arlat et al., 1990] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., Martins, E., and Powell, D. (1990). Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on software engineering*, 16(2):166–182.

[Arlat et al., 2003] Arlat, J., Boué, J., Crouzet, Y., Jenn, E., Aidemark, J., Folkesson, P., Karlsson, J., Ohlsson, J., and Rimén, M. (2003). Mefisto: a series of prototype tools for fault injection into vhdl models. In *Fault injection techniques and tools for embedded systems reliability evaluation*, pages 177–193. Springer.

[Arlat et al., 1993] Arlat, J., Costes, A., Crouzet, Y., Laprie, J. C., and Powell, D. (1993). Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923.

[AS6802, 2011] AS6802, S. (2011). Time-triggered ethernet. *SAE International*.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

[Babuska and Oden, 2004] Babuska, I. and Oden, J. T. (2004). Verification and validation in computational engineering and science: basic concepts. *Computer methods in applied mechanics and engineering*, 193(36):4057–4066.

[Baeten, 2005] Baeten, J. C. (2005). A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146.

[Barnes, 2017] Barnes, C. (2017). *Verification and validation of wireless sensor network protocol properties through the system's simulation and emulation.* PhD thesis, Université Côte d'Azur.

[Barry, 1989] Barry, J. (1989). Design and analysis of fault-tolerant digital systems.

[Bartols et al., 2011] Bartols, F., Steinbach, T., Korf, F., and Schmidt, T. C. (2011). Performance analysis of time-triggered ether-networks using off-the-shelf-components. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 49–56. IEEE.

[Benso and DiCarlo, 2011] Benso, A. and DiCarlo, S. (2011). The art of fault injection. *Journal of Control Engineering and Applied Informatics*, 13(4):9–18.

[Benso and Prinetto, 2003] Benso, A. and Prinetto, P. (2003). *Fault injection techniques and tools for embedded systems reliability evaluation*, volume 23. Springer Science & Business Media.

[Benso et al., 1998] Benso, A., Prinetto, P., Rebaudengo, M., and Reorda, M. S. (1998). Exfi: a low-cost fault injection system for embedded microprocessor-based boards. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(4):626–634.

[Blanton and Zimmermann, 2015] Blanton, E. and Zimmermann, A. (2015). A roadmap for transmission control protocol (tcp) specification documents. *RFC7414*.

[Boehm, 1984] Boehm, B. W. (1984). Verifying and validating software requirements and design specifications. *IEEE software*, 1(1):75.

[Bosch, 2010] Bosch, G. (2010). Industrial ethernet: The key advantages of sercos iii. Accessed = 2020-01-18.

[Brooks, 2001] Brooks, P. (2001). Ethernet/ip-industrial protocol. In *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 01TH8597)*, volume 2, pages 505–514. IEEE.

[Buja et al., 2005] Buja, G., Zuccollo, A., and Pimentel, J. (2005). Overcoming babbling-idiot failures in the flexcan architecture: a simple bus-guardian. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 2, pages 8–pp. IEEE.

[Carreira et al., 1998] Carreira, J., Madeira, H., and Silva, J. G. (1998). Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136.

[Choi and Iyer, 1992] Choi, G. S. and Iyer, R. K. (1992). Focus: an experimental environment for fault sensitivity analysis. *IEEE Transactions on Computers*, 41(12):1515–1526.

[Christen and Bakalar, 1999] Christen, E. and Bakalar, K. (1999). Vhdl-ams-a hardware description language for analog and mixed-signal applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272.

[Coelho, 2017] Coelho, R. F. (2017). *Buffer Analysis and Message Scheduling for Real-time Networks.* PhD thesis, Technical University Kaiserslautern.

[Combs, 2006] Combs, G. (2006). Tshark-the wireshark network analyser. *URL http://www. wireshark. org.*

[Committee et al., 1999] Committee, J. S. S. S. et al. (1999). Software system safety handbook: A technical and managerial team approach, joint services computer resource management group, us navy. *US Army, US Air Force.*

[d. Souto et al., 2016] d. Souto, P. F., Portugal, P., and Vasques, F. (2016). Reliability evaluation of broadcast protocols for flexray. *IEEE Transactions on Vehicular Technology*, 65(2):525–541.

[de Moura et al., 2004] de Moura, L., Owre, S., Harald, R., John, R., N, S., Maria, S., and Ashish, T. (2004). *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114. Springer.

[Deng et al., 2014] Deng, L., Yu, D., et al. (2014). Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387.

[Diarra et al., 2015] Diarra, A., Hogenmueller, T., Zimmermann, A., Grzemba, A., and Khan, U. A. (2015). Improved clock synchronization start-up time for ethernet avb-based in-vehicle networks. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–8. IEEE.

[Dutertre et al., 2012] Dutertre, B., Easwaran, A., Hall, B., and Steiner, W. (2012). Model-based analysis of timed-triggered ethernet. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 9D2–1. IEEE.

[Dutertre and Sorea, 2004] Dutertre, B. and Sorea, M. (2004). Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 199–214. Springer.

[Eidson, 2006] Eidson, J. C. (2006). *Measurement, control, and communication using IEEE 1588*. Springer Science & Business Media.

[Entrena et al., 2001] Entrena, L., López, C., and Olías, E. (2001). Automatic generation of fault tolerant vhdl designs in rtl. In *FDL (Forum on Design Languages)*.

[Farkas et al., 2017] Farkas, J., Bello, L. L., and Gunther, C. (2017). Time-sensitive networking task group. Available at `http://www.ieee802.org/1/pages/tsn.html` (2018/08/15).

[Farkas et al., 2018] Farkas, J., Bello, L. L., and Gunther, C. (2018). Time-sensitive networking standards. *IEEE Communications Standards Magazine*, 2(2):20–21.

[Fejoz et al., 2018] Fejoz, L., Regnier, B., Miramont, P., and Navet, N. (2018). Simulation-based fault injection as a verification oracle for the engineering of time-triggered ethernet networks. *Proc. Embedded Real-Time Software and Systems (ERTS 2018)*.

[Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Rfc 2616: Hypertext transfer protocol–http/1.1, june 1999. *Status: Standards Track*, 1(11):1829–1841.

[Fisher and Raney, 1969] Fisher, G. A. and Raney, G. N. (1969). On the representation of formal languages using automata on networks. In *10th Annual Symposium on Switching and Automata Theory (swat 1969)*, pages 157–165.

[Geilen, 2002] Geilen, M. C. W. (2002). *Formal techniques for verification of complex real-time systems*. Technische Universiteit Eindhoven.

[Gessner et al., 2014] Gessner, D., Barranco, M., Ballesteros, A., and Proenza, J. (2014). sfican: A star-based physical fault-injection infrastructure for can networks. *IEEE Transactions on Vehicular Technology*, 63(3):1335–1349.

[Gil et al., 2003] Gil, D., Baraza, J. C., Gracia, J., and Gil, P. J. (2003). Vhdl simulation-based fault injection techniques. In *Fault injection techniques and tools for embedded systems reliability evaluation*, pages 159–176. Springer.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Grady, 2010] Grady, J. O. (2010). *System verification: proving the design solution satisfies the requirements*. Elsevier.

[Guideline, 2003] Guideline, P. (2003). Profinet architecture description and specification. *Version V2. 0, Karlsruhe: PNO.*

[Gunneflo et al., 1989] Gunneflo, U., Karlsson, J., and Torin, J. (1989). Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 340–347. IEEE.

[Han et al., 1995] Han, S., Shin, K. G., and Rosenberg, H. A. (1995). Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213. IEEE.

[Hsueh et al., 1997] Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *Computer*, 30(4):75–82.

[IEC61508, 2010] IEC61508 (2010). Iec 61508 functional safety of electrical/electronic/-programmable electronic safety-related systems. *International Electrotechnical Commission, Geneva, Switzerland.*

[IEEE, 2016] IEEE (2016). Ieee standard for local and metropolitan area networks – bridges and bridged networks - amendment 25: Enhancements for scheduled traffic. *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)*, pages 1–57.

[IEEE802.1AS, 2011] IEEE802.1AS (2011). Ieee standard for local and metropolitan area networks - timing and synchronization for time-sensitive applications in bridged local area networks. *IEEE Std 802.1AS-2011*, pages 1–292.

[IEEE802.1Q, 2014] IEEE802.1Q (2014). Ieee standard for local and metropolitan area networks–bridges and bridged networks. *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)*, pages 1–1832.

[Impagliazzo and Fabiomassimo, 2003] Impagliazzo, L. and Fabiomassimo, P. (2003). Development of a hybrid fault injection environment. In *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, pages 81–93. Springer.

[ISO, 2011] ISO, I. (2011). 26262: Road vehicles-functional safety. *International Standard ISO/FDIS*, 26262.

[Jakovljevic et al., 2017] Jakovljevic, M., Geven, A., Ademaj, A., Gaderer, G., Fidi, C., Männel, E., Rox, J., and Elvikis, D. (2017). Projet Safe4RAIL: Deliverable D1.1 -State-Of-The-Art Document on Drive-by-Data.

[Johnson, 2015] Johnson, J. (2015). Fpga network tap: Designing the ethernet pass-through. Accessed = 2016-12-18.

[Kanawati et al., 1995] Kanawati, G. A., Kanawati, N. A., and Abraham, J. A. (1995). Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on computers*, 44(2):248–260.

[Karlsson et al., 1998] Karlsson, J., Folkesson, P., Arlat, J., Crouzet, Y., Leber, G., and Reisinger, J. (1998). Application of three physical fault injection techniques to the experimental assessment of the mars architecture. *Dependable Computing and Fault Tolerant Systems*, 10:267–288.

[Kim et al., 2008] Kim, W. S., Kim, H. A., Ahn, J. H., and Moon, B. (2008). System-level development and verification of the flexray communication controller model based on systemc. In *2008 Second International Conference on Future Generation Communication and Networking*, volume 2, pages 124–127.

[Kirrmann and Zuber, 2001] Kirrmann, H. and Zuber, P. A. (2001). The iec/ieee train communication network. *IEEE Micro*, 21(2):81–92.

[Kirrmann and Zuber, 2001] Kirrmann, H. and Zuber, P. A. (2001). The iec/ieee train communication network. *IEEE Micro*, 21(2):81–92.

[Klensin et al., 2008] Klensin, J. et al. (2008). Simple mail transfer protocol. *RFC7504*.

[Kopetz, 2011] Kopetz, H. (2011). *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media.

[Kopetz et al., 2005] Kopetz, H., Ademaj, A., Grillinger, P., and Steinhammer, K. (2005). The time-triggered ethernet (tte) design. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 22–33. IEEE.

[Kwiatkowska et al., 2002] Kwiatkowska, M., Norman, G., and Parker, D. (2002). Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer.

[Lanigan et al., 2010] Lanigan, P. E., Narasimhan, P., and Fuhrman, T. E. (2010). Experiences with a canoe-based fault injection framework for autosar. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 569–574.

[Laplante, 2004] Laplante, P. A. (2004). Real-time systems design and analysis.

[Laprie, 1992] Laprie, J.-C. (1992). *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems 5*. Springer-Verlag Wien, 1 edition.

[Le Boudec and Thiran, 2001] Le Boudec, J.-Y. and Thiran, P. (2001). *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media.

[leavingbio, 2019] leavingbio (2019). The nervous system - leaving certificate biology. `http://leavingbio.net/nervous-system/`, Last accessed on 2019-10-20.

[Leveson, 2012] Leveson, N. (2012). Engineering a safer world. *MIT Press,*.

[Li et al., 2018] Li, J., Li, Q., and Tang, X. (2018). Modeling ttethernet startup service in systemc for verifying fault-tolerant protocol under fail-omission scenarios. In *TENCON 2018-2018 IEEE Region 10 Conference*, pages 1753–1757. IEEE.

[Liu and Yang, 2011] Liu, Y. and Yang, C. (2011). Omnet++ based modeling and simulation of the ieee 1588 ptp clock. In *2011 International Conference on Electrical and Control Engineering*, pages 4602–4605.

[Madeira et al., 1994] Madeira, H., Rela, M., Moreira, F., and Silva, J. G. (1994). Rifle: A general purpose pin-level fault injector. In *European Dependable Computing Conference*, pages 197–216. Springer.

[Mitchell et al., 1997] Mitchell, T. M. et al. (1997). Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877.

[Mouradian, 2013] Mouradian, A. (2013). *Proposition et vérification formelle de protocoles de communications temps-réel pour les réseaux de capteurs sans fil.* PhD thesis, INSA de Lyon.

[Muller and Valle, 2010] Muller, C. and Valle, M. (2010). System verification of flexray communication networks through behavioral simulations. In *2010 IEEE International Behavioral Modeling and Simulation Workshop*, pages 1–6.

[Murata, 1989] Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

[Napatech, 2017] Napatech ((accessed February 3, 2017)). *nt4e-4-std-product-overview.* `https://www.napatech.com/support/resources/data-sheets/napatech-smartnic-product-overview/nt4e-4-std-product-overview/`.

[Natella et al., 2016] Natella, R., Cotroneo, D., and Madeira, H. S. (2016). Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3):44:1–44:55.

[Navet and Fejoz, 2016] Navet, N. and Fejoz, L. (2016). Cpal: High-level abstractions for safe embedded systems. In *Proceedings of the International Workshop on Domain-Specific Modeling*, pages 35–41. ACM.

[Oberkampf and Roy, 2010] Oberkampf, W. L. and Roy, C. J. (2010). *Verification and validation in scientific computing*. Cambridge University Press.

[Obermaisser, 2011] Obermaisser, R. (2011). *Time-Triggered Communication*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

[Oliver et al., 2018] Oliver, R. S., Craciunas, S. S., and Steiner, W. (2018). Ieee 802.1qbv gate control list synthesis using array theory encoding. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–24.

[Onwuchekwa et al., 2018] Onwuchekwa, D., Foo, J., and Obermaisser, R. (2018). Fault Injection Framework for Time Triggered Ethernet. In *7th Transport Research Arena TRA 2018 (TRA 2018)*. Zenodo.

[Onwuchekwa et al., 2020] Onwuchekwa, D., Garcia, J. E., Lua, C., and Obermaisser, R. (2020). Failure detection in tsn startup using deep learning. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 140–141.

[Onwuchekwa and Obermaisser, 2018] Onwuchekwa, D. and Obermaisser, R. (2018). Fault injection framework for assessing fault containment of ttethernet against babbling idiot failures. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–6.

[Onwuchekwa and Obermaisser, 2019] Onwuchekwa, D. and Obermaisser, R. (2019). Performance evaluation of deterministic communication in the railway domain. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 337–343.

[Packard et al., 2000] Packard, H., Marvell, and Broadcom (2000). Reduced gigabit media independent interface (rgmii).

[Pahlevan and Obermaisser, 2018a] Pahlevan, M. and Obermaisser, R. (2018a). Evaluation of time-triggered traffic in time-sensitive networks using the opnet simulation framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 283–287. IEEE.

[Pahlevan and Obermaisser, 2018b] Pahlevan, M. and Obermaisser, R. (2018b). Redundancy management for safety-critical applications with time sensitive networking. In *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–7. IEEE.

[Park, 2002] Park, S. G. (2002). Fieldbus in iec61158 standard. In *Proceedings on the 15th CISL Winter Workshop, Kushu, Japan.*

[Part, 2006] Part, A. D. N. (2006). Avionics full-duplex switched ethernet (afdx) network, arinc specification 664, part 7, aeronautical radio. Technical report, International Aircraft Manufacturer Airbus.

[POWERLINK, 2004] POWERLINK, E. (2004). Epl (ethernet powerlink): Proposal for a publicly available specification for real-time ethernet. *Doc. IEC65C/356a/NP.*

[Qadir and Hasan, 2015] Qadir, J. and Hasan, O. (2015). Applying formal methods to networking: theory, techniques, and applications. *IEEE Communications Surveys & Tutorials*, 17(1):256–291.

[Racek et al., 2012] Racek, S., Herout, P., and Hlavička, J. (2012). Dependability evaluation of time triggered architecture using simulation. *Computing and Informatics*, 23(1):51–76.

[Revsbech et al., 2012] Revsbech, K., Madsen, T. K., and Schiøler, H. (2012). High precision testbed to evaluate ethernet performance for in-car networks. In *2012 12th International Conference on ITS Telecommunications*, pages 548–552.

[Riverbed-Technology, 2019] Riverbed-Technology (2019). Ropnet modeler 17.1 documentation. Available at `https://support.riverbed.com/content/support/software/steelcentral-npm/modeler-index.html`.

[Rodriguez-Navas et al., 2003] Rodriguez-Navas, G., Jimenez, J., and Proenza, J. (2003). An architecture for physical injection of complex fault scenarios in can networks. In *EFTA 2003. 2003 IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.03TH8696)*, volume 2, pages 125–128 vol.2.

[Rousselle et al., 2001] Rousselle, C., Pflanz, M., Behling, A., Mohaupt, T., and Vierhaus, H. T. (2001). A register-transfer-level fault simulator for permanent and transient faults in embedded processors. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, page 811. IEEE.

[Safe4RAIL, 2018] Safe4RAIL ((accessed February 3, 2018)). *Safe4RAIL (Safe architecture for Robust distributed Application Integration in roLling stock).* `https://safe4rail-1.safe4rail.eu/`.

[SAFE4RAIL-1, 2019] SAFE4RAIL-1 ((accessed February 10, 2019)). Safe architecture for robust distributed application integration in roling stock. `https://safe4rail.eul`.

[Saha et al., 2007a] Saha, I., Misra, J., and Roy, S. (2007a). Timeout and calendar based finite state modeling and verification of real-time systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 284–299. Springer.

[Saha et al., 2007b] Saha, I., Roy, S., and Chakraborty, K. (2007b). Modeling and verification of ttcan startup protocol using synchronous calendar. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 69–79. IEEE.

[Saha et al., 2016] Saha, I., Roy, S., and Ramesh, S. (2016). Formal verification of fault-tolerant startup algorithms for time-triggered architectures: A survey. *Proceedings of the IEEE*, 104(5):904–922.

[Sandic et al., 2018] Sandic, M., Pavkovic, B., and Teslic, N. (2018). Impact of anomalies within ttethernet network on synchronization protocol: Analysis using omnet++ simulations. In *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, pages 29–34. IEEE.

[Segall et al., 1995] Segall, Z., Vrsalovic, D., Siewiorek, D., Ysskin, D., Kownacki, J., Barton, J., Dancey, R., Robinson, A., and Lin, T. (1995). Fiat-fault injection based automated testing environment. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995,'Highlights from Twenty-Five Years'.*, page 394. IEEE.

[Sieh et al., 1997] Sieh, V., Tschache, O., and Balbach, F. (1997). Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 32–36. IEEE.

[Spurgeon, 2000] Spurgeon, C. E. (2000). *Ethernet the definitive guide.* O'Reilly Media, Inc.

[Steiner, 2005] Steiner, W. (2005). Model-checking studies of the flexray startup algorithm. *TU Wien, Institut für Technische Informatik, Research Report.*

[Steiner and Dutertre, 2010] Steiner, W. and Dutertre, B. (2010). Smt-based formal verification of a ttethernet synchronization function. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 148–163. Springer.

[Steiner and Dutertre, 2011] Steiner, W. and Dutertre, B. (2011). Automated formal verification of the ttethernet synchronization quality. In *NASA Formal Methods Symposium*, pages 375–390. Springer.

[Steiner and Dutertre, 2013] Steiner, W. and Dutertre, B. (2013). The ttethernet synchronisation protocols and their formal verification. *International Journal of Critical Computer-Based Systems 17*, 4(3):280–300.

[Steiner and Kopetz, 2006] Steiner, W. and Kopetz, H. (2006). The startup problem in fault-tolerant time-triggered communication. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 35–44. IEEE.

[Steiner et al., 2004] Steiner, W., Rushby, J., Sorea, M., and Pfeifer, H. (2004). *Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation*. IEEE.

[Stott et al., 2000] Stott, D. T., Floering, B., Burke, D., Kalbarczpk, Z., and Iyer, R. K. (2000). Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, pages 91–100. IEEE.

[Swales et al., 1999] Swales, A. et al. (1999). Open modbus/tcp specification. *Schneider Electric*, 29.

[Tsai and Iyer, 1995] Tsai, T. and Iyer, R. (1995). Ftape-a fault injection tool to measure fault tolerance. In *10th Computing in Aerospace Conference*, page 1041.

[Varga, 2010] Varga, A. (2010). Omnet++. In *Modeling and tools for network simulation*, pages 35–59. Springer.

[Vector-Informatik, 1996] Vector-Informatik (1996). Testing ecus and networks with canoe. Accessed = 2019-12-18.

[Verma et al., 2011] Verma, A. K., Ajit, S., and Kumar, M. (2011). Dependability of networked computer-based systems. In *Dependability of Networked Computer-based Systems*, pages 169–183. Springer.

[Wang et al., 2009] Wang, K., Xu, A., and Wang, H. (2009). Avoiding the babbling idiot failure in a communication system based on flexible time division multiple access: A bus guardian solution. In *2009 IEEE International Symposium on Industrial Electronics*, pages 1292–1297. IEEE.

[Wani et al., 2020] Wani, M. A., Bhat, F. A., Afzal, S., and Khan, A. I. (2020). *Advances in Deep Learning*, volume 57. Springer.

[Yang et al., 2016] Yang, X., Zhang, S., and Li, H. (2016). Research and implementation of precise time synchronization system of microgrid based on ieee 1588. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 258–261. IEEE.

[Zhang et al., 2016] Zhang, Y., He, F., Lu, G., and Xiong, H. (2016). Clock synchronization compensation of time-triggered ethernet based on least squares algorithm. In *2016 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*, pages 1–5. IEEE.

[Zhao et al., 2018] Zhao, L., Pop, P., Zheng, Z., and Li, Q. (2018). Timing analysis of avb traffic in tsn networks using network calculus. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 25–36.

[Ziade et al., 2004] Ziade, H., Ayoubi, R. A., Velazco, R., et al. (2004). A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186.

[Ziermann et al., 2012] Ziermann, T., Butiu, A., Teich, J., and Ziener, D. (2012). Fpga-based testbed for timing behavior evaluation of the controller area network (can). In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–6. IEEE.