Modeling and simulation of the TTEthernet communication protocol

Alexander Zafirov

Kongens Lyngby 2013
IMM-M.Sc.-2013

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013

I dedicate this master thesis project to my grandfather Alexander Velikov(1927-2011).

Abstract

Embedded systems have countless application areas - household electronics, computer networks, medical equipment, aircraft controllers, etc. Their responsibility varies depending on the system they are integrated in. Hard-real time systems are the ones where computing the correct result by a task and it meeting its deadline is of great importance. The correctness of these results depends also on the time when these are produced. By extending the list of constrains to the application with high reliability and fault tolerance, the outline of the safety-critical systems characteristics is given. Nowadays complex embedded real-time systems are implemented using distributed architectures, composed of heterogeneous processing elements interconnected using communication networks.

The master thesis objective is to model and simulate the TTEthernet - targeted to distributed safety-critical real-time systems. A TTEthernet network is composed of a set of clusters. Each cluster consists of a set of End Systems (ESes) interconnected by links and Network Switches (NSes). The links are full duplex, allowing thus communication in both directions. The protocol has three traffic classes of different timing criticality: Time-Triggered (TT), Rate-Constrained (RC) and Best Effort (BE). Time-Triggered(TT) messages are the ones with highest priority and take precedence over other messaging types in the network. TT communication is done through offline scheduling of static scheduling tables i.e. messages are sent at predefined periods of time. RC transmission has less priority than a TT one and is executed whenever no time-triggered communication is present. RC provides bounded end-to-end latency and delay limitation. BE messages are the ones with lowest critical level - thus least priority. They do not provide any timing constraints or guarantees that a message will be received. This makes them less reliable for tasks with high temporal requirements.

TTEthernet is compliant with ARINC 664p7 [afd09] which defines the concept of virtual links. TTEthernet implements them as logical point-to-point connections in the network. They create "tree" structures with an End System as the root node and a set of End Systems each of which defined as a leaf node. These structures are used to route frames from the root to the leafs. Each virtual link carries a single message.

The result of the work done is the creation of a simulator. This tool takes as input the network topology, set of messages in the system and multiple simulation characteristics via command line. The output files are a comma separated value (CSV) file containing the worst-case and average delays, a GraphViz file presenting the network topology from each virtual link's perspective and a Joint Photographic Experts Group (JPEG) file showing a Gantt chart with the scenario that led to the worst-case end-to-end delay for a predefined frame.

In order to evaluate the working of the simulator, different tests were done both with artificial and real world examples. The artificial ones were presented as ten test cases. The real world examples were based on NASA's Orion Crew Exploration Vehicle, represented as a topology in two versions - a simple and an enhanced one. The output produced by the simulator was used for different purposes. In the case of Orion it determined the appropriateness of a given topology for the needs that the system presents. In the other test cases - the steady-state was sought with respect to the amount of simulation performed. Finally, the output was compared to previously performed analysis in the form of a discussion which gave a broader understanding of the relation analysis-simulation.

Acknowledgements

First and foremost I want to thank Paul Pop for being a great supervisor. The communication that we had throughout the master thesis project was always timely, concise and valuable.

I would also like to thank Domitian Tamas-Selicean for his extensive support. His feedback - both technical and theoretical - helped me numerous times get pass difficult problems.

Lastly, I would like to thank my family and friends for the being there for me when I needed them most. I love you!

Contents

List of Figures

Chapter 1

Introduction

The term "embedded systems" describes systems that repeatedly extract and analyze information from sensors, generating output sent to actuators. They are also known as closed-loop control systems. There are various ways one can classify them - according to their cost, performance, safety, dependability etc.

Hard-real time systems have the highest demand on the programs they utilize due to the nature of their domain. Applications executing on such systems should work in a timely manner - any deadlines that are not met lead to catastrophic consequences loss of data, financial resources or potential threat to human life. Systems with such characteristics are further refined as safety-critical. An example one can point out are applications such as fly- and drive-by-wire where there are no direct connections between a pilot operating the control system of an aircraft and its control surfaces. Such an environment poses multiple requirements - ultra-high reliability (e.g. minimum delay of a command send from the aircraft control to its surfaces), fault tolerance, extensive redundancy.

To address these and other demands, separation of the communication in multiple types is needed. Subsequently, two of the approaches developed to deal with this in real-time systems ([Kop11]) are **event-triggered** and **time-triggered** ([Sue12]). The former describes the action of triggering a signal upon the occurrence of a specific event. This implies a dynamic strategy of dealing with events. The later is managed by the progression of time. Each communication that hap-

pens is a predefined static periodic event. A time-triggered system interacts with the world according to an internal schedule, whereas an event-triggered system responds to stimuli that are outside its control.

The term mixed-criticality ([BBB+09]) applications denotes the integration of both approaches onto the same system. It refers to those applications that comprise non-critical and critical traffic with different Safety-Integrity Levels (SIL). To achieve this they are divided through temporal and spatial separation. The functionality of mixed-criticality applications is implemented on top of an integrated structure of interconnected heterogeneous processing elements.

A setting as complex as an aircraft encompasses another class of embedded systems - distributed systems. They were initially devised as being **federated**. This definition implies that each applications in the system (e.g. autopilot) comprises of a fault-tolerant embedded control system that connects to others of its kind through minor interconnections. This is also known as partitioning ([Rus99]). The newer applications adapt the approach of **integrated** solutions, where resources are shared throughout multiple applications. The trade-off between both approaches is as follows: in the case of federated architectures there are higher expenses for replicating the systems a great amount of times as well as protection against fault propagation. On the other hand, integrated solutions lower the costs for integrating the applications but introduce risk of cascading failures. Both architectures represent the safety-critical core of the applications built on top of them. Deciding how to implement them and what services to provide them with, are major key points in the construction and certification of safety-critical embedded systems.

The two systems designs approaches - time-triggered and event-triggered - find application in different areas. The time-triggered approach is generally preferred for integrated safety-critical systems. An integrated system brings different applications together - whereas a safety-critical system keeps them apart. This is a reference to the previously denoted term partitioning. It allows single applications to be "deconstructed" into smaller components that can be developed to different safety levels. Also although the purpose of partitioning is to exclude fault propagation, it has the added benefit that it promotes composability.[1] Partitioning and composability concern the predictability of the resources and services perceived by the clients (applications and their subfunctions) of an architecture. One of predictability's two dimensions is value - logically correct behavior. The other is time - predictable rate of delivery, latency and jitter of services. Especially in context of fault-tolerant systems temporal predictability is difficult to achieve in event-triggered architectures. This makes

[1] A composable design is one in which individual applications are unaffected by the choice of the other applications with which they are integrated.

time-triggering the only option for safety-critical systems.

1.1 Databus

One of the primary architectural components is the bus (databus). It can be a physical or logical entity (communication protocol) and is used for control and transmission of communication across the network. Buses such as Ethernet resolve contention probabilistically and therefore can provide only probabilistic guarantees of timely access. Thus they give no assurance at all in the presence of faults. Buses for embedded systems such as CAN, LonWorks, or Profibus (Process Field Bus) use various priority, preassigned slot or token schemes to resolve contention deterministically. Time-triggered buses provide static preallocation of communication bandwidth in the form of a global schedule - each node knows the schedule. It therefore knows when it is allowed to send messages and when it should expect to receive them. This means gives the benefit of resolving contention at design time (i.e. as the schedule is constructed), rather than runtime. This allows for thorough assessment of the impact of the schedule on the system. Because all communication is time-triggered by the global schedule there is no need to attach source or destination addresses to messages sent over the bus - each node knows the sender and intended recipients of each message by virtue of the time at which it was sent. Time-triggered operation provides efficiency, determinism, and partitioning.

1.2 Communication protocols

Here are the architectures of two avionics and two automobile communication protocols in the interest of deducing principles common to all of them, the main differences in their design choices, and the trade-offs made. On one hand we have the avionics buses the Honeywell SAFEbus ([HD93]) and the SPIDER protocol ([MGPM04]). On the other - the automobile buses Time-Triggered Architecture (TTA) ([KB03]) and FlexRay ([SJ08]). All four of the considered examples are primarily time-triggered ([PSG+11]). This is a fundamental design choice that influences many aspects of their architectures and mechanisms and sets them apart from event-triggered buses such as Controller Area Network (CAN), Byteflight and LonWorks.

Figure 1.1 depicts a bus interconnect topology similar the one utilized by SAFEbus. The Bus Interface Units (BIUs) are duplicated and the interconnect bus is quad-

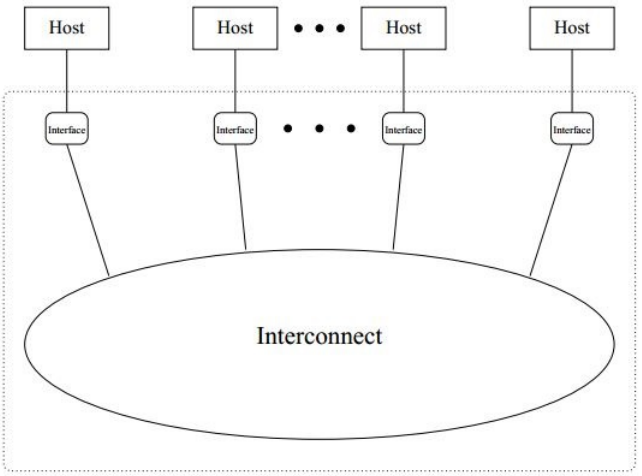


Figure 1.1: J. Rushby. "Generic Bus" Figure. [Rus01], 5p.

redundant. Features like clock synchronization, message scheduling and transmission functions are implemented on SAFEbus main unit - the BIU. The access control to the interconnect is done by the bus guardian of BIU's partner. Every Bus Interface Unit of a pair drives a different pair of interconnect buses. It is, however, able to read all four of them. The interconnect buses, on the other hand, are composed each of two data lines and one clock line.

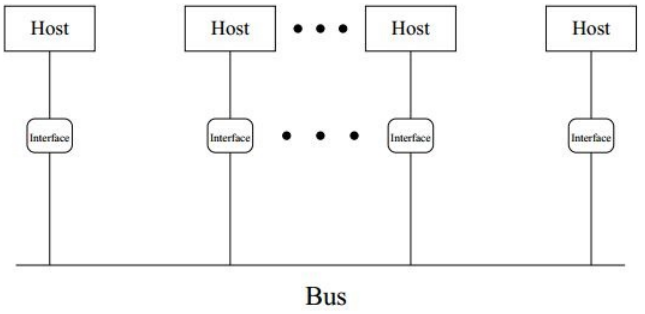


Figure 1.2: J. Rushby. "Interconnect Bus" Figure. [Rus01], 6p.

There are two types of implementation of a Time-Triggered Architecture - the currently used TTA-bus (a bus interconnect topology similar to that shown in Figure 1.2) and the next generation TTA-star (a star interconnect topology such as the one in Figure 1.3). Both designs have the same interfaces(also

called controllers), which implement the TTP/C protocol - the heart of TTA. It is responsible for clock synchronization, message sequencing and transmission functions. In a TTA-bus each controller drives the buses through a bus guardian, whereas in a TTA-star implentation the guardian functionality is carried out in the central hub. TTA-star provides a setup for distributed configurations where subsystems are connected by hub-to-hub links.

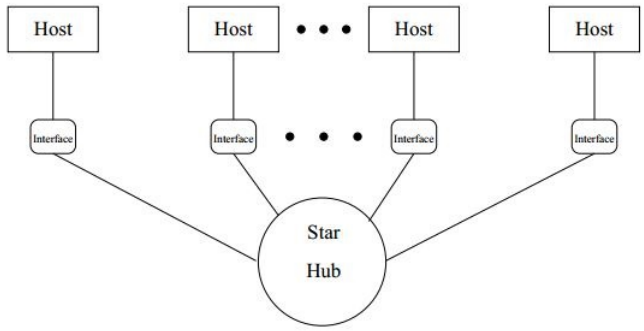


Figure 1.3: J. Rushby. "Star Interconnect" Figure. [Rus01], 7p.

The SPIDER interconnect is composed of active elements called Redundancy Management Units (RMUs). Its topology can be organized either as shown in Figure 1.4, where the RMUs and interfaces(the BIUs) form part of a centralized hub, or as in Figure 1.3, where the RMUs form the hub, or similar to Figure 1.1, where the RMUs provide a distributed interconnect. The lines connecting hosts to their interfaces are optical fiber, and the 12 whole system beyond the hosts (i.e., optical fibers and the RMUs and BIUs) is called the Reliable Optical Bus (ROBUS). Clock synchronization and other services of SPIDER are achieved by distributed algorithms executed among the BIUs and RMUs.

FlexRay can use either an "active" star topology similar to that shown in Figure 1.3, or a "passive" bus topology similar to that shown in Figure 1.2. In both cases, duplication of the interconnect is optional. Each interface f (communication controller) drives the lines to its interconnects through separate bus guardians located with the interface. As with TTA-star, FlexRay can also be deployed in distributed configurations in which subsystems are connected by hub-to-hub links.

In the context of communication protocols implementing the time-triggered architecture, the master thesis takes a closer look at TTEthernet (see chapter 2). This network protocol supports safety-critical applications of mixed-criticality character. It enables interconnection of heterogeneous processing elements in hard-real time safety-critical distributed embedded systems. This is

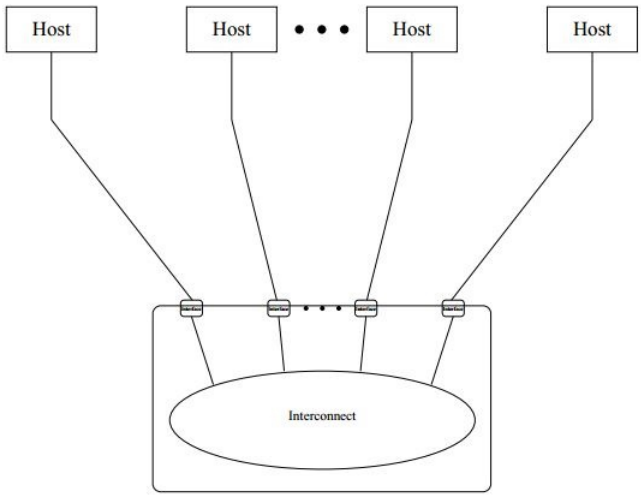


Figure 1.4: J. Rushby. "Spider Interconnect" Figure. [Rus01], 8p.

done through the management of three traffic classes - Time-Triggered (TT), Rate-Constrained (RC) and Best Effort (BE). Time-Triggered(TT) messages are the ones with highest priority. Because they have the highest level of criticality, they take precedence over other messaging types in the network. TT communication is done through offline scheduling of static scheduling tables i.e. messages are sent at predefined periods of time. This type of message exchange is most suitable for the construction of deterministic distributed systems where the operation of each element can be specified with high precision.

Event-Triggered communication presents two types of traffic in TTEthernet - Rate-Constrained(RC) and Best Effort(BE). RC messaging is next in the criticality scale of after TT. Thus an RC transmission has less priority than a TT one and is executed whenever no time-triggered communication is present. RC provides bounded end-to-end latency and delay limitation.

BE messages are the ones with lowest critical level - thus least priority. They cannot provide any timing constraints or guarantees that the message will be received at all because they are executed whenever no other communication is present. This, of course, makes them less reliable and useful for tasks with high temporal requirements.

To utilize a "mixed-criticality" application on a single system and enable the transmission of all three classes two key components are needed - spatial and temporal separation. Spatial separation is done through the concept of virtual

links. As introduced in ARINC 664p7, virtual links represent logical point-to-point connections that create "tree-like" structures in the network - a single end system as the root node and one or more end systems as leaf nodes. Temporal separation, on the other hand, is achieved in two ways depending on the messages being transmitted. One is the above mentioned static preallocation of bandwidth (offline scheduling). It is used in case of TT communication. If the messages are of RC type, separation is provided through bandwidth allocation. With its fault-tolerance design, TTEthernet comprises various capabilities like redundancy, scalability and multiple fault containment.

1.3 Thesis objectives

The goal of the master thesis project is to develop a simulator based TTEthernet protocol. The simulator has to be fast and accurate. The requirements for the simulator are:

- model the two simulation paradigms - action- and event-oriented
- model all the three integration policies (conflict resolution mechanisms) - timely block, shuffling and preemption
- determine the average end-to-end delays for all BE and RC messages and the worst-case end-to-end communication delays for the RC messages
- compare and evaluate results from simulation to an existing analysis presented in a paper
- simulator should be designed and implemented so that it can be used inside an optimization loop

1.4 Thesis structure

The structure master thesis report is as follows:

- **Chapter 1** makes an introduction to the topic of protocols for embedded systems by briefly comparing the characteristics of multiple protocols. It describes the goals and the motivation behind the project.

- **Chapter 2** presents the theory used throughout the project with regards to Time-Triggered Ethernet.
- **Chapter 3** describes the process of simulation and modeling, the various types of simulations that can be performed and the appropriateness of each one with regards to the master thesis project. It must be noted that the purpose of chapters 2 and 3 is not to educate the reader or repeat the numerous textbooks written in the field. It is intended to support and clarify the decisions made during the implementation process.
- **Chapter 4** focuses on the development of two simulators following the action and event-driven paradigms. It gives a detailed view of their common features as well as the differences they have with respect to the implementation, performance and development issues.
- **Chapter 5** reflects on the verification of the simulators correctness and the evaluation of the output data.
- The thesis report completes with **chapter 6** which provides conclusions in the form of a general overview of the theory and the achieved results that were presented. It also gives a description of the possible future extensions.

Chapter 2

TTEthernet

The following chapter presents TTEthernet in detail. It is a protocol that interconnects heterogeneous processing elements in hard-real time safety-critical distributed systems. The chapter is divided into sections that describe the background of TTEthernet, the architecture model that it supports, its actual operation, scheduling policies and fault-tolerance techniques that it implements. It also provides an example of the protocol's workflow. Finally, a short summary presents TTEthernet's key features.

2.1 Background and Definition

TTEthernet has multiple characteristics, which will be described step-wise in this section. Firstly, TTEthernet is based on IEEE 802.3 Ethernet standard. This means that is supports communication between applications over heterogeneous media. The problem with Ethernet is that it is implemented with half-duplex switching, thus if a message collides with another transmission it is resent after a certain time period depending on the back-off strategy used. No matter how small, the possibility that messages will collide each time still exists. Subsequently these collisions result in unbounded transmission times.

To address these issues, TTEthernet is also made compliant with the ARINC

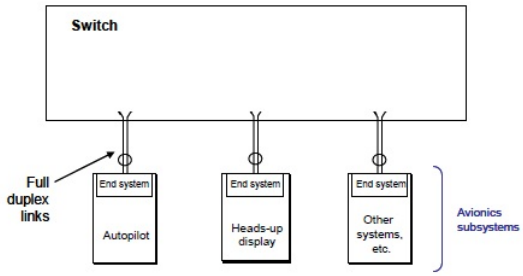


Figure 2.1: "Full-Duplex, Switched Ethernet Example" Figure. [Eng05], 12p.

664p7 (see [afd09]) protocol - a full-duplex switched Ethernet with predictable event-triggered communication (see Figure 2.1). The full-duplex mechanism does away with frame collisions but introduces a new threat to the protocol - congestion. In order to handle it and make the protocol congestion free, there should be control over the message transmissions. This is directly related to the determinism in TTEthernet. The protocol has predictable operation achieved by a mixed asynchronous/synchronous approach (see Figure 2.2). The asynchronous approach is described in ARINC 664, where bandwidth partitioning is done through rate-constrained communication. Also maximum latency in the system is controlled. The synchronous approach on the other hand manages jitter and has a strict time base - implements time-triggered messaging. It is presented in the SAE AS6802 standard which is a fault-tolerant self-stabilizing synchronization strategy.

The ARINC 664p7 protocol also introduces the concept of virtual links ([ASBCH13]). They represent logical point-to-point connections in the network. Virtual links provide one of the key features needed to implement mixed-criticality applications - spatial separation. A thorough discussion on virtual links can be found in the follow up subsection.

As a further extension to event-triggered communication of ARINC 664p7, TTEthernet provides a static pre-scheduled time-triggered messaging. This augmentation makes the protocol perfect for networks implemented in highly critical hard real-time systems, where deadlines must always be respected. To summarize, TTEthernet is defined as a synchronous, deterministic and congestion free.

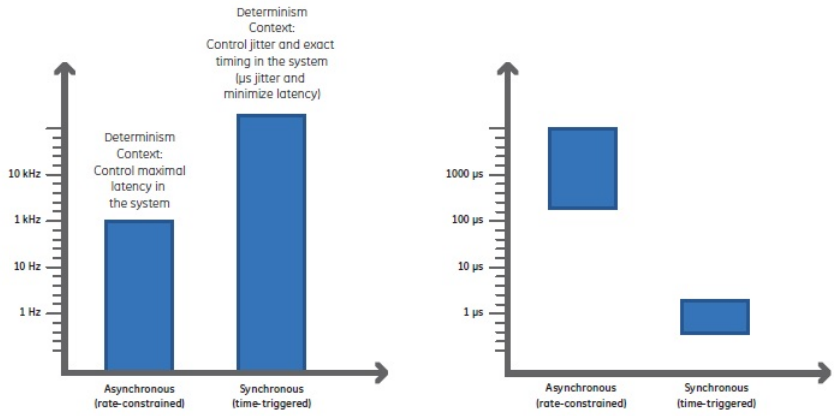


Figure 2.2: "Determinism context in Ethernet networks depends of the application (max. sampling rate) and the approach to system design asynchronous (coordination and synchronization among functions is conducted at higher layers) or synchronous (control of timing and synchronization at network level)." Figure. [Pla09a], 4p.

2.2 Virtual links

Virtual links are used to route frames from a sender to one or multiple receivers. In TTEthernet the Virtual Link ID (VLID) is of size 16 bits and type unsigned integer. Traffic is navigated through the network based on the resulting 48 bit destination address (see Figure 2.3).

Because switches in the TTEthernet network are configured so that they redirect messages to one or multiple links and because end systems can only have a single VLID, virtual link connections create structures that resemble "trees" with an end system as the root node and a set of end systems each of which defined as a leaf node. Figure 2.3 depicts the routing of a frame from end system 1 with VLID 100 through the network to the designated communication ports in end systems 2 and 3.

2.2.1 Virtual link isolation

As previously described virtual links are logical connections between two end systems which provide spatial separation among traffic of different character i.e. critical and non-critical. Figure 2.5 shows that any given physical link can

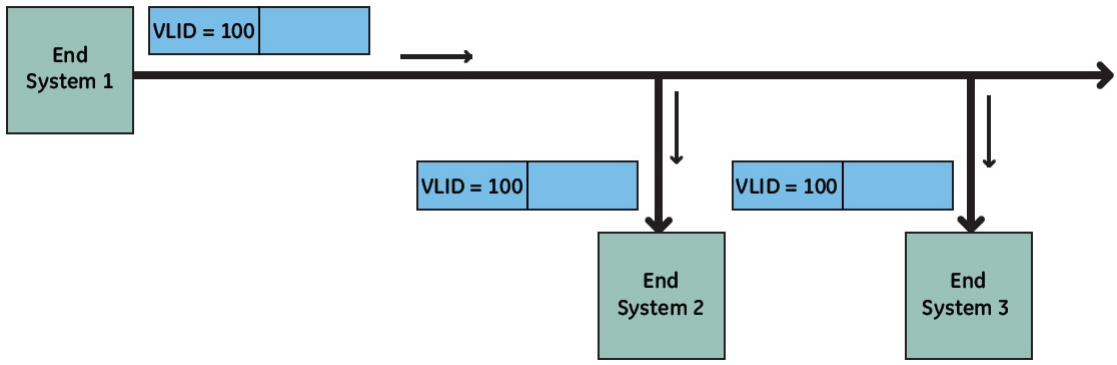


Figure 2.3: "Format of Ethernet Destination Address in AFDX Network." Figure. [Pla05], 12p.

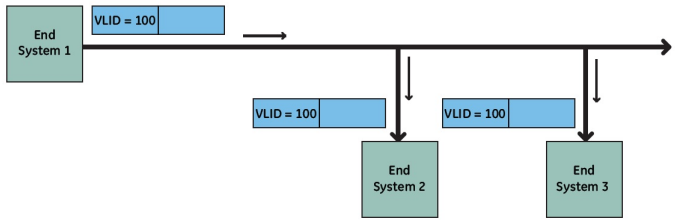


Figure 2.4: "Packet Routing Example." Figure. [Pla05], 11p.

contain one or more virtual links, which in turn transport frames to a single or multiple destination ports.

In order to isolate the traffic being transmitted and to make sure that no interference between communication of different messages is possible, TTEthernet limits the frame rate and size that passes through the virtual links. Thus each frame is supplied with a Bandwidth Allocation Gap (BAG) and a Lmax value. BAG is the minimal time interval (in ms) between the transmission of two frames in the network. Lmax is the maximum size (in bytes) one frame can have in the given virtual link. Appropriate values for BAG and their matching transmission frequencies are given in Figure 2.6. These parameters are assigned on per-virtual link basis for in each end system.

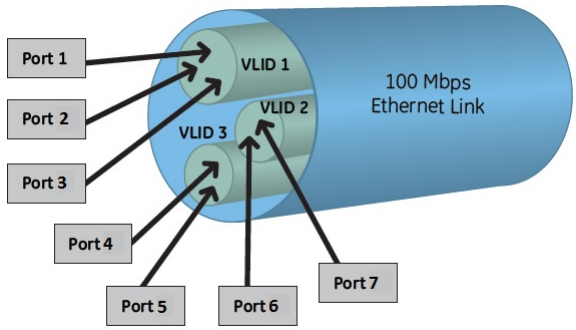


Figure 2.5: "Three Virtual Links Carried by a Physical Link." Figure. [Pla05], 14p.

BAG milliseconds	Hz
1	1000
2	500
4	250
8	125
16	62.5
32	31.25
64	15.625
128	7.8125

Figure 2.6: "Allowable BAG Values." Table. [Pla05], 14p.

2.2.2 Virtual link scheduling

As shown in the previous subsection, communication ports are linked to virtual links. Messages coming from the ports that are to be sent by the end system must follow the scenario depicted in Figure 2.7. First they are wrapped into a TTEthernet frame and placed in a transmission queue. Virtual Link Scheduler (VLS) supervises whether the BAG and Lmax limitations for the given virtual link are respected. It regulates the amount of jitter of the communication by transmitting the traffic passing through it. Some sources of jitter are congestion in the virtual link queues, multiplexing the scheduled frames into the Redundancy Management Unit (RMU) and their actual transmission through the physical links.

The formulas given below determine a bound on the output jitter that every end system must comply with. The first describes the upper bound to the amount of

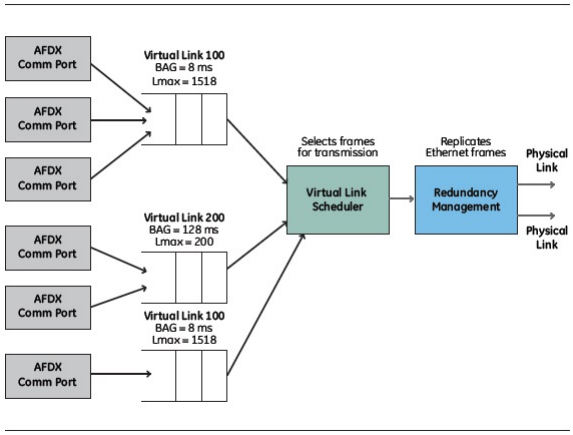


Figure 2.7: "Virtual Link Scheduling." Figure. [Pla05], 15p.

jitter on the delay that a frame can experience when other frames are scheduled on other virtual links. The second one is limit on the overall jitter of the end system. If these limitations are met by all end systems, the cluster will prove to be "deterministic".

$$\max\_jitter \le 40\mu s + \frac{\sum_{j\in\{set\ of\ VLs\}} (20 + L\max_j) \times 8)}{Nbw}$$

$$max\_jitter \le 500\mu s$$

Figure 2.8: Figure. [Pla05], 15p.

When the VLS passed a frame on, it receives a sequence number and gets replicated by the RMU, if that is needed. The complete frame is then transmitted over the network through the physical link.

2.3 Architecture

The following section presents the architecture of TTEthernet through a simple example that is situated on the "cluster level" in the TTEthernet synchronization topology i.e.there is one synchronization domain and one priority (see Figure 2.9).

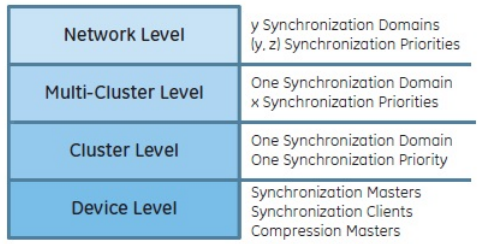


Figure 2.9: "The TTEthernet synchronization topology has four levels." Figure. [Pla09a], 15p.

Figure 2.10 shows a cluster containing four end systems (ES_1 , ES_2, ES_3, ES_4) which communicate among themselves through the means of physical links and network switches (NS_1, NS_2). The path from one ES to another is called a communication channel and encompasses all the communication media in between. Every ES is supplied with a CPU, RAM, ROM (or some other type of non-volatile memory) and network interface card (NIC) used to identify the ES on the TTEthernet network. Communication in the cluster is implemented as full-duplex and is denoted with solid black bi-directional arrows. Some other characteristics of the cluster are that it is multi-hop i.e. messages can travel through multiple NSes and that TT messages are synchronized per cluster.
The Figure 2.9 depicts two applications A_1 and A_2 that have different criticality level tasks. A_1 maps its highly critical tasks τ_1, τ_2 and τ_3 onto ES_1, ES_3 and ES_4 respectively. The non-critical application A_2 places its tasks τ_1 and τ_4 on ES_1 and ES_4. The spatial separation that TTEthernet provides addresses the system's mixed-criticality character. In the example below it can happen so that messages sent from task τ_1 and τ_4 intersect in a physical connection or a network switch. In this case the virtual links vl_1 and vl_1 are used to isolate critical from non-critical messages. Virtual links consists of multiple dataflow paths which in turn contain uni-directional flow constructs called dataflow links.

Besides the example presented in Figure 2.10 there also exist other topologies such as single cluster with redundant communication channels or cascaded multi-clusters implementing a master-slave strategy.

2.4 Traffic classes

As previously noted, TTEthernet implements both event-triggered and time-triggered communication in order to satisfy applications of mixed-criticality

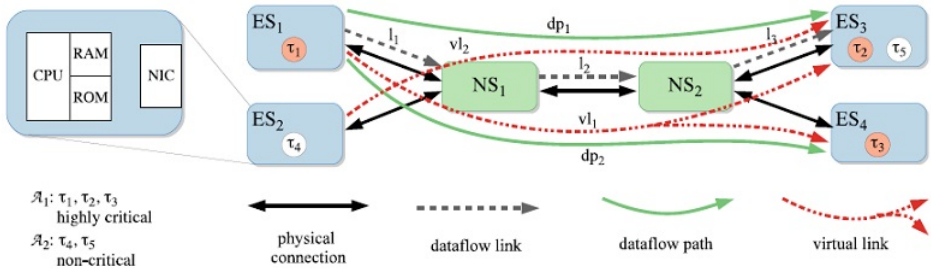


Figure 2.10: "TTEthernet cluster example." Figure. [TSPS12], 3p.

character. This is reflected in the traffic that the protocol generates by creating three categories of messages. Time-Triggered (TT) messages are the ones with highest priority. Because they have the highest level of criticality, they take precedence over other messaging types in the network. TT communication is done through offline scheduling of static scheduling tables i.e. messages are sent at predefined periods of time. This type of message exchange is most suitable for the construction of deterministic distributed systems where the operation of each element can be specified with high precision. Event-Triggered communication presents two types of traffic in TTEthernet - Rate-Constrained (RC) and Best Effort (BE). RC messaging is next in the criticality scale of after TT. Thus an RC transmission has less priority than a TT one and is executed whenever no time-triggered communication is present. RC provides bounded end-to-end latency and delay limitation.

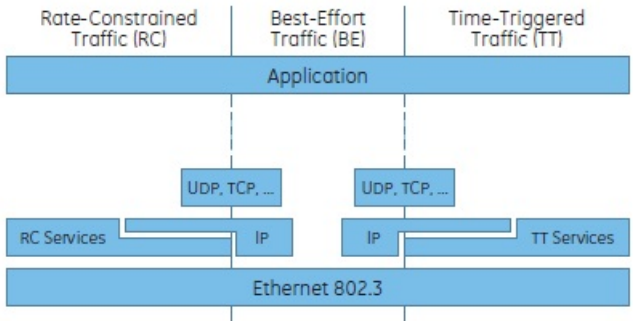


Figure 2.11: "Relation of TTEthernet to existing communication standards." Figure. [TSPS12], 10p.

BE messages are the ones with lowest critical level - thus least priority. They cannot provide any timing constraints or guarantees that the message will be

received at all because they are executed whenever no other communication is present. This, of course, makes them less reliable and useful for tasks with high temporal requirements.

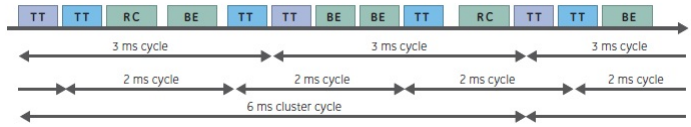


Figure 2.12: "TTEthernet includes TT, RC and BE messages." Figure. [Pla09a], 11p.

As already stated both temporal and spatial separation is needed to utilize a "mixed-criticality" application on a single system. The spatial separation is done through the concept of virtual links. Temporal separation is achieved in two ways depending on the messages being transmitted. Offline scheduling is used in case of TT communication. If the transmissions are of RC type, separation is provided through bandwidth allocation.

TTEthernet is a transparent synchronization protocol which enables it to exchange "foreign" types of traffic on the same network. To provide fault-tolerance some part of the devices in that network can be implemented so that they generate synchronization messages.

2.5 Protocol operation

Now that the elements of the TTEthernet topology have been presented, this subsection will delve into how the actual transmission of messages takes place. The focus of the example will be critical communication in the protocol i.e. TT and RC messages.

Figure 2.13 presents a cluster consisting of two end systems (ES_1 and ES_2) and three network switches (NS_1, NS_2, NS_3). The communication channel that will be the focus of this example starts at ES_1, continues through NS_1 and ends in ES_2. Application A_1 aims to transfer a RC message m_1 from task τ_1 to τ_3, whereas A_2 seeks to transmit the time-triggered message m_1 from τ_2 to τ_4. In the ESes, CPUs partition the two tasks so that the aforementioned spatial separation is achieved. In order to visualize the transmission flow better, the messages' paths are labeled on each step. Each of them has a distinct color - green for RC and blue for TT.

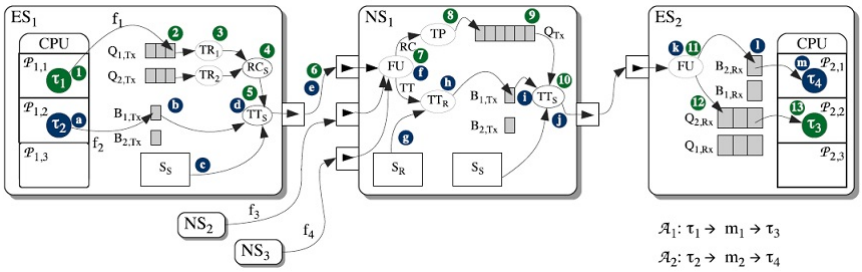


Figure 2.13: "TT and RC message transmission example." Figure. [TSPS12], 4p.

2.5.1 Time-Triggered Communication

The process starts with task τ_2 putting message m_2 in frame f_2 (a). Next, the frame is placed in buffer B_{1,T_x} designated especially for that frame (b). Each frame that arrives overrides the value stored in the buffer before because of the single cell that it has. A look up in the send schedule S_s (c) must be performed in order to follow the predefined offline scheduling done before the start of the transmission. Scheduling tables are stored in every end system and network switch in the cluster (S_s and S_r). When the predefined moment comes, scheduler TT_s sends f to NS_1 (d) through the data link between the two (e). The Filtering Unit (FU) is the first in the switch to receive the incoming frame. FU checks its the validity and integrity (f) and separates frames based on the type of communication they implement. The receiving schedule S_r gives information on whether the frame arrived in a certain window interval (g). If the it did and messaging is of type TT, the frame is forwarded to the receive scheduler TT_r (h). If the frame exceeds the predefined window interval or if it had already arrived, the data is dropped. This fault-tolerance mechanism is known as fault containment.

From then on communication resembles the pattern described above. TT_r places the frame in a specific buffer - in this case B_{1,T_x}. With correspondence to schedule S_s in NS_1, TT_s sends the frame to its final destination - ES_2. When it reaches the right partition - $P_{2,1}$ - it is unwrapped and message m_2 is read when the currently active task is t_4.

2.5.2 Rate Constrained Communication

Like with Time-Triggered communication, the transmission starts by packaging message m_1 in a frame (1). The first contrast comes with the queue, the frame is put in (2). It is one per virtual link and can contain multiple cells. The general difference, however, is that RC transmission is event-triggered. This implies that no scheduling tables are needed to oversee the time of sending and receiving in the end systems and network switches. Another consequence is that temporal separation is done through "bandwidth allocation". As mentioned before, TT transfer is predefined so TT messages don't have to worry about a particular mechanism for temporal separation once the offline scheduling is done. This is not the case with RC traffic seeing as it is dynamic. In order to apply "bandwidth allocation" for each channel, the protocol designer must define the minimum time interval between each consecutive pair of RC messages. That is the definition of the previously mentioned Bandwidth Allocation Gap (BAG). A key characteristic of BAG is that it must be less or equal to the reciprocal value of the rate at which a given frame is transmitted. BAG is defined by the Traffic Regulator (TR) (3). Let there be two TR tasks with different sized BAG that have two messages. Figure 2.14 depicts how these messages are multiplexed by the RC scheduler (4). So when both TRs try to send their message simultaneously, the phenomenon called jitter occurs. In this case the jitter is equal to the sum of the transmission duration of all the other messages that were sent before the particular message.

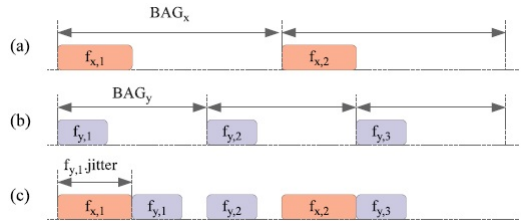


Figure 2.14: "Multiplexing two RC frames." Figure. [TSPS12], 5p.

Yet another key difference between TT and RC messages is presented when an RC one reaches TT (5) - the priority. Having a lower priority because of the lesser critical level, RC messages can be transmitted only when there are no TT messages around. Thus TTEthernet must provide a way to integrate the two so that the bandwidth is fully utilized and yet all traffic starting from the highest level of criticality - must reach its destination in time. The possible cases are two: a TT message is transmitted over the dataflow link and a RC message is sent by the scheduler for transmission, and the reverse. The former case is

trivial - TT messages have precedence of RC, so lower critical messages have to wait until the bandwidth is free. The latter case, however, is non-trivial - thus of greater interest. It will be discussed in an upcoming section in depth.
As previously described, once a frame is received by the Filtering Unit, it is then checked for validity and integrity. If the frame passes the test, it is sent to the Traffic Policy (TP) (7) which implements fault-containment just like TT (8). This is done by an algorithm called leaky bucket which looks at the time at which any given frame is received and that of the one before it. It then compares the result to the BAG defined for the virtual link. If the BAG has a lower value - the frame continues its transmission. If the BAG is greater then the frame is dropped. After that the procedure of reaching the designated task - in our case task 3 - resembles TT messaging.

2.5.3 Integration Policies

There are three ways in which a network switch in TTEthernet deals with the conflict that arises when high priority message(TT) becomes ready for transmission while a low priority message(RC) is being relayed through the network: shuffling, timely block and preemption. All of them will be discussed in greater detail in this section with respect to their resource utilization, quality of transmission and other key aspects to the message transfer.

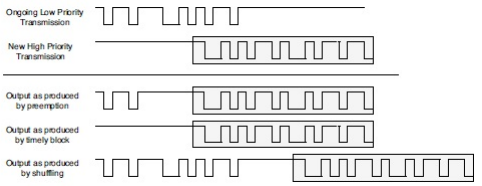


Figure 2.15: "Integration Methods for High-Priority (H) and Low-Priority (L) Traffic." Figure. [TSPS12], 192p

In case of preemption, the transfer of the RC frame is discontinued. The minimal "silence" period is then executed on the active channel by the network switch, immediately followed by a TT message. *Real-Time Quality* is high because preemption makes sure that TT messages are transmitted with the lowest latency - constant and known in advance in the best case. *Resource utilization* is inefficient. In every case where a RC frame is withheld from completing its transfer, there needs to be a re-transmission. This, subsequently, results in a loss of bandwidth. The drawback can be fixed by using additional functionality for message reconstruction. In general the truncated messages must be

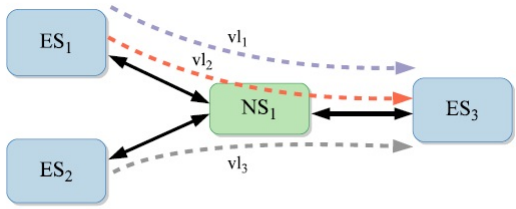
perceived as faulty messages, so mechanisms for dealing with that are to be applied. One such mechanism can be the generation of a signal that breaks the rules for correctly generated RC messages. Thus the end systems that receive the traffic will be able to notice the difference.

The second option is timely block. It is available only if a TT frame is scheduled before the RC frame is fully sent through the dataflow link. If this happens, the RC frame is blocked on the link for a certain time period and TT communication is delayed. *Real-Time Quality* is high because delay of high critical traffic is constant, which implies deterministic behavior. *Resource utilization* is inefficient. Because of RC messages with unknown length, solutions implementing timely block must postpone communication for the maximum size of RC message that is defined. To address this low-criticality frames can contain their length as the value of field. Thus only RC traffic that can be fully transmitted is guaranteed to be relayed.

Lastly there is shuffling, which describes the exchange of priorities between TT and RC messages. In the case of shuffling a TT frame waits until the RC frame is being fully transmitted. The worst-case scenario is examined when the RC communication has the maximal length defined in the network. *Real-Time Quality* is low. Having to wait for multiple RC frames will degrade the performance substantially. As a relief comes the fact that TT messages are dispatched according to their static pre-scheduled time tables. Having that in mind synchronization based on the times known a priori reduces the jitter that is created because of the priority exchange *Resource utilization* is efficient because, unlike preemption and timely block, frames are not interrupted - thus no truncated traffic is present. So, in a sense, utilization is optimal for shuffling. A drawback of this approach is that it has increased complexity in scalability for TT communication. Also, although controlled and bounded, integration of BE traffic within shuffling may pose a problem especially when its source is unknown.

2.6 Example

After describing the architecture model and communication in detail, the report supplies an example of how scheduling is performed in TTEthernet. The cluster presented here consists of three end systems(ES_1, ES_2, ES_3) and a single network switch NS_1. There are also three virtual links that are used to transmit three frames. Figure 2.16(a) is used to provide visualization purposes. 2.16(b) gives the period, deadline, transmission time and virtual link for each frame. Timely block is used to manage ambiguities in the traffic.



(a) Example architecture model

	period (μs)	deadline (μs)	C_i (μs)	$\mathcal{M}$
$f_1 \in \mathcal{F}^{RC}$	300	300	75	vl_1
$f_2 \in \mathcal{F}^{TT}$	200	200	50	vl_2
$f_3 \in \mathcal{F}^{TT}$	300	300	50	vl_3

(b) Example application model

Figure 2.16: "Example system model" Figure. [TSPS12], 5p.

The problem here is to find the best way to schedule both TT and RC messages so that they meet their deadlines. Because TT communication is pre-scheduled, a look up in the scheduling table S present in all ESes and in NS will be enough to determine if any deadlines are missed. The worst-case end-to-end delay needs be calculated when RC frames are to be scheduled. To portray the actual schedule order of communication, Gantt charts are given as Figure 2.17 for the initial scheduling and Figure 2.18 for the optimized variation. Both schedules present valid strategies for TT scheduling. The charts give the communication over the three dataflow links - [ES_1, NS_1], [ES_2, NS_1] and [NS_1,ES_1] - over 600 μs time.

Frame f_1 is event-triggered, which means that it can be scheduled in many ways considering the static scheduling of f_2 and f_3. Figure 2.19 shows the situation where all the TT frames are dispatched as soon as possible i.e. their delay is minimal. Let frame f_1 be sent from ES_2 to NS_1 at 105 μs. A timely block is issued once the frame is compared to the higher priority frame f_3. Because the next instance of f_2 interrupts its full transmission, f_1 cannot finish the operation. Once all f_2 and f_3 frames are transferred, f_1 is passed. This results in the worst-case end-to-end delay for f_1 - 470 μs, which makes it not schedulable, seeing as its deadline is 300 μs.

Figure 2.17 shows a greatly reduced worst-case delay - 270 μs - making it schedulable. This is the result of 50 μs delay of frame f_3 's second instance, which gives frame f_1 the needed time to get scheduled.

With these visual representations of the scheduling process, the reader is able to identify the enormous effect of RC over TT traffic.

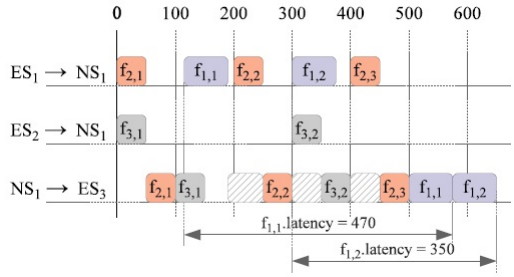


Figure 2.17: "Initial TT schedule" Figure. [TSPS12], 6p.

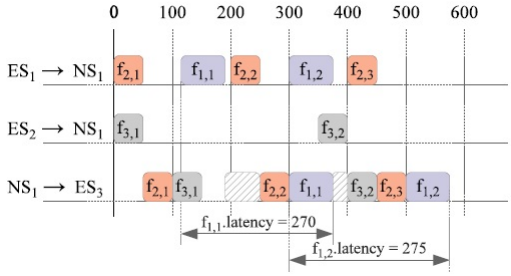


Figure 2.18: "Optimized TT schedule" Figure. [TSPS12], 6p.

2.7 Fault-tolerance

TTEthernet is designed with the intention of being fault-tolerant. This is achieved through various capabilities that the protocol comprises. Firstly it is characterized by its ability to implement redundancy. This can be done in multiple elements of the network - end systems, network switches and other segments. The degree to which redundancy is incorporated within the system is dependent to the amount of fault-tolerance that is required in the system.

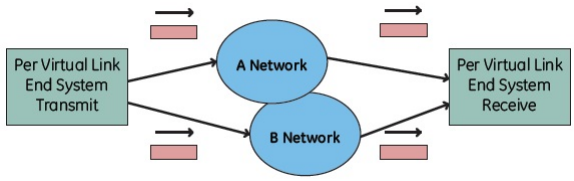


Figure 2.19: "A and B Networks." Figure. [Pla05], 13p.

Redundancy management is a property of the whole network in general because TTEthernet comprises of two, independent of each other, networks A and B(see Figure 2.19). This means that there is double amount of generated traffic that is routed through the network. Also for each created frame, the receiving end system will obtain two identical frames.

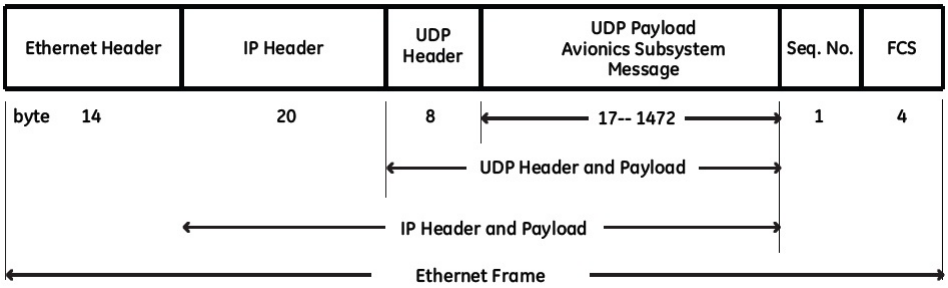


Figure 2.20: "AFDX Frame and Sequence Number." Figure. [Pla05], 13p.

A problem that arises is how to distinguish that a replica packet has been received. To address this problem TTEthernet frames are equipped with a field of length 1 byte called sequence number. This byte is situated between the IP header and payload and the FCS fields. Because the frame is of fixed size the byte used as a sequence number is taken from the IP/UDP payload(see Figure 2.20).
The functionality that checks a frame's sequence number is named "Integrity Checking". It is applied by the end system upon receiving of the frame for each link and network port. If the communication has already been presented to the end system the packet is dropped, otherwise it passes through.
A visual summary of the process of redundancy management is provided within Figure 2.21.

After that comes **scalability**. It increases latent failure detection probability on a multi-platform system and decreases implementation costs for systems that expand rapidly. Different configurations with equality between the end systems(multi-master synchronization) or multiple end systems being controlled from a single one(master-slave synchronization) are possible. In systems that have similar criticality levels TTEthernet provides "service history".

Next the protocol integrates **tolerance to multiple inconsistent faults**. This means that when multiple concurrent failures occur in a network with equally

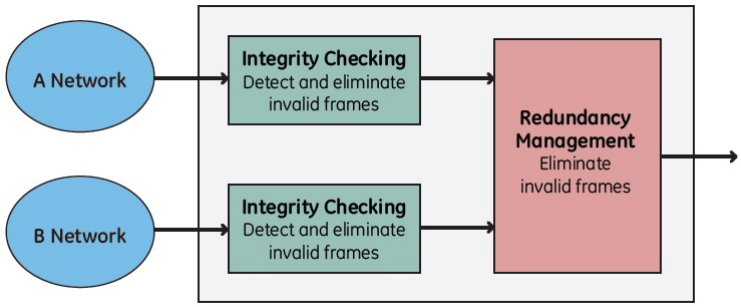


Figure 2.21: "Receive Processing of Ethernet Frames." Figure. [Pla05], 13p.

ranked end systems, they will be dealt with in a cost-efficient way. Faults can be present either in the communication or in the end system(inconsistent-omission faulty communication path or end system) or simultaneously in both.

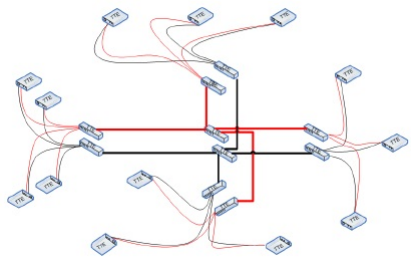


Figure 2.22: "TTEthernet provides implicit fault tolerance mechanisms." Figure. [Pla09b], 7p.

Network switches and end systems are implemented so that they can operate with **guardian functions**(see Figure 2.22). These functions aim to determine whether the traffic throughout the network is working according to the initial intentions of the designers. In case an element has become faulty, the guardian simply disconnects it from the rest of the network segments. To give a higher level of fault-tolerance, a system may implement multiple guardians at various places. In a system where a single network switch interconnects multiple end systems, the guardian function may be implemented as a central bus guardian. This allows for masking a set of end systems that have become faulty.(e.g. experiencing the "babbling idiot" state where they sent repetitive messages in a short time interval). If the architecture is distributed, the guardian function

takes care that no faulty nodes corrupt the communication bus. Thus TTEthernet provides tolerance to arbitrary end systems failures.

Lastly the protocol is designed with **self-stabilization capabilities**. This means that after a case of multiple faults across a distributed system, synchronization will be re-established.

2.8 Summary

TTEthernet is a protocol made to accommodate the needs of hard real-time systems. Moreover its design allows for integration on distributed systems where transmission types vary in safety integrity levels. Each type is supplied with specific fault-tolerant mechanisms in order to insure that safety-critical constraints are met. TTEthernet can extend an existing network to different topologies located on heterogeneous media without introducing major changes to its current state. This is achieved through the protocol's main network components - end systems and network switches. TTEthernet's scalability and fault-tolerance facilitate its suitability for a wide range of applications where problems like cost, efficiency, safety and predictability are of key importance.

Chapter 3

Modeling and Simulation

This chapter presents the notion of modeling and simulation in the context of system development. It discusses key characteristics of a simulation such as selecting input probability distributions, random number generators and how to perform output data analysis. The two simulation paradigms - continuous- and discrete-event - are presented and compared by their appropriateness with respect to the master thesis project. A closer look at the existing world views of discrete-event simulation provides theoretical background needed to support the simulator implementation.

3.1 System

The main concept that lays in the center of this chapter is that of a **system**. It is a set of interacting or interdependent components (e.g. people, machines) forming an integrated whole. As an example Figure 3.1 depicts a Local Area Network (LAN) which is essentially a system consisting of computers that communicate with servers. To describe it we need to observe the parameters that define the system at a given point in time. This is the definition of **state**. In the displayed computer network these state variables could be the number of servers that are currently working, the computers that are being serviced, etc.

Figure 3.1: "Computer Networking - LAN Networking". Digital image. Accessed 16 August 2013

When talking about types of systems we can distinguish between two types - discrete and continuous. A discrete system is one for which the state variables change instantaneously at separated points in time. The LAN above is a discrete system - a change to its state of occurs for example in case of a server malfunction or when a computer receives an acknowledgment from a server. A server can either be functional or nonfunctional - it doesn't have an intermediate state. The term continuous system is used to represent state variables changing continuously with respect to time. A system like that is a ship traveling in the sea. The ship's state changes with the continuous change of its speed and position through time.

3.2 Model

There are many ways one can study the workings of a system. This is visualized in Figure 3.2. The two main ways are - performing actual experiments with the system and construction of a model used to represent the system. The choice of a model over actual experiments can easily be explained with the following example. If the LAN network from Figure 3.1 has one million computers, an actual experiment would be a poor choice seeing as the funds needed for its fruition are formidable. This is the case with a great number of systems in real life and that is way building a model is often a favorable choice.

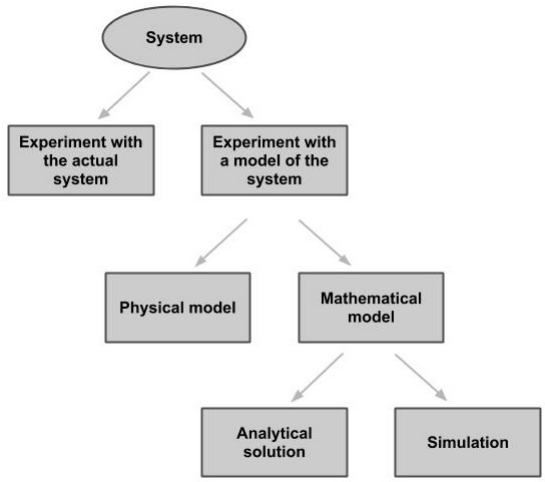


Figure 3.2: "Ways to study a system". Figure. [LK99], 4p.

A **model** is a simplified representation of a system which is created with the purpose of studying it. It consists of a set of assumptions, concerning the aspects of a given system, that affect the problem under investigation. A model is expressed through mathematical or logic relationships of entities that give understanding about the behavior of the system in detail.
There are various types of models each of which will briefly be discussed here. They are:

- static vs dynamic
- deterministic vs stochastic
- discrete vs continuous

Static simulation models depict a "snapshot" of a system - a single moment in its evolution. It can also display a system without any notion of time. A **dynamic** model, on the other hand, is a representation of system that evolves with the progress of time.
Determinism is predictability. Thus if a model does not comprise probabilistic (random) elements it is a **deterministic**. For a given set of input values such models repeatedly produce the same output. In the case where a model contains element introducing randomness to the system, it is a **stochastic** model.

The definitions for **discrete** and **continuous** models resemble the ones given previously for the existing types of systems.

Taking into account the definitions given above, the model of TTEthernet protocol used for the simulation can be categorized as discrete, dynamic, stochastic.

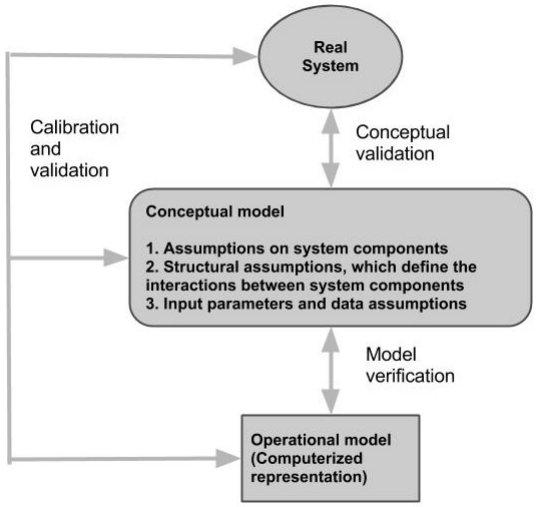


Figure 3.3: "Construction of a model". Figure. [BCNN00]

3.2.1 Verification and Validation

As discussed the first process of building a model is deciding what kind of model best replicates the existing system. When a conceptual model is constructed the question whether it is a valid representation of the real system arises. The answer to this question has three parts - verification, validation and credibility. Because credibility address the project lifecycle with respect to a company or another institution (i.e. have the model approved by a manager) this section focuses on the former two parts.

The creation of a model is a repetitive process that is described in Figure 3.3. It can be done through a comparison either between the model and operational model or between the model and the real system. In the former case the process is called **verification**. It checks whether the conceptual model is accurately portrayed by the operational model (computerized representation). As this is

a computer program, this process will often result in debugging the simulation. The complexity of this task grows with that of the modeled system. This is the case because the amount logical paths in big systems tend to become extremely large.

Verification of a model can be done through the following methods:

- creation of flow diagrams
- study of the reasonableness of output from the operational model
- verification that input parameters have not been changed
- documentation of working process
- visualization of working process (GUI, animation)
- debugging the program (trace)

After a comparison between the conceptual and operational model has been made a step called **calibration** is introduced. It is simply the process of refining (readjusting) the model by removing flows discovered during the verification and fine-tuning it. This process continues until both models have an acceptable degree of difference in the output that they produce. The calibration can be done through either a subjective or objective test.

The comparison between a conceptual model and the real system and their corresponding behavior is called **validation**. It should be noted that absolute validity of a model cannot be achieved. This is because a simulation model is an approximation of the real system. Thus a model can be made more valid the more it is calibrated. It is also true that a model is developed with specific requirements. It can be the case that two models of a same system have different purposes and so differ in their operation.

Validation of simulation models in a three-step approach. The first step is the creation of a high face validity model. This is a subjective measure of the extent to which this selection appears reasonable. It is tested by having an external view - from people knowledgeable with the real system - test the model output for admissibility and discover flaws in the process. The following step is validation model assumptions. It can be done though:

- structural assumptions - describe how the system operates and usually involve simplifications and abstractions of reality

- data assumptions - based on the collection of reliable data and correct statistical analysis of the data

The last step is done by comparing the model to real system input-output transformations. In order to perform this step, however, there should be data recorded while observing the system. They will be used to support to calculate the necessary system characteristics. The validation test consists of comparing the output from the real system to that of the model for the same set of input conditions.

3.3 Simulation

As seen previously in Figure 3.2 the generation of a mathematical model leads to two options of studying a system - perform analytic solution or create a simulation. The goal of the master thesis is to model and simulate the TTEthernet protocol, therefore this section defines the term simulation. It also gives theoretical support and explanation to key design decisions when constructing a computer simulation.

Simulation is an imitation of the operation of a real-world process or system over time. It simulates potential changes in the system by evaluating the mathematical model numerically. The data accumulated by the simulation allows the study of systems in design stage - before their actual construction. It also gives insight of the desired characteristics of the model. A simulation enables the generation of artificial history of the system and helps draw conclusions for the real system.

3.3.1 Selecting input probability distributions

A stochastic simulation uses random inputs. These could be arrival times, arrival order, etc. For such simulations there needs to be a source of randomness - a defined input probability distribution. There are three approaches that can be used in order to specify a distribution - trace-driven, define empirical distribution, fitted "standard" distribution.
The term **trace-driven** simulation denotes usage of the generated random inputs themselves directly in the simulation as data values directly in the simulation. The pros with this choice are that the simulation is done with historically ordered stream. This is very useful when trying to compare a simulator to an existing system. In favor of this approach is also model validation as it allows

for easy comparison between pre-generated output with that of the simulator. The problems with trace-driven simulation are that it just reproduces historical results which results in insufficient data for a multiple simulation runs.
The second way to handle random variables is to use them to define an **empirical distribution** function. Its strengths lay in generating values between the min and max data points. This is valued in the cases where it is known in advance that a random variable can never exceed a certain value. We also use empirical distribution when theoretical distributions may not be an adequate "fit" for observed data. The shortcomings of this approach lay in the fact that values will never be larger than max (the upper bound). This means that extreme events cannot be simulated. Output data and cumulative probabilities are generated for given input data - cumbersome if input is large.
Fitted "standard" distribution describes inserting a theoretical distribution form to the random variables through standard techniques of statistical inference. This approach helps generate values outside the observed data range and provides a compact way of data representation. An obstacle to using this approach can be the lack of an adequate "fit" for observed data.

The trace-driven approach was the one chosen in the master thesis project. As previously stated, this approach allows for comparison with existing outputs. This is the case here since in chapter 5, there is a comparison between the results of an analysis and the simulators.

3.3.2 Random Number Generation

A stochastic simulation with a probability distribution needs to use random numbers (random variates). A "good" arithmetic random number generator (RNG) has the following characteristics:

- provides a independent and identical uniform distribution of random values in the interval of [0,1]. This is also known as IID U (0,1).
- no correlation between values
- good time and space complexity
- ability to reproduce given stream (subsegment of numbers produced by the generator) of values
- ability to reproduce separate streams (independent generators) of values. The last two bullets can be used to study the correct workings of a simulator

The TTEthernet simulators use a Prime Modulus Multiplicative Linear Congruential Generator (PMMLGC) proposed by Marse and Roberts (1983). It is utilized when the arrival times for RC and BE messages are generated.

3.3.3 Output Data Analysis

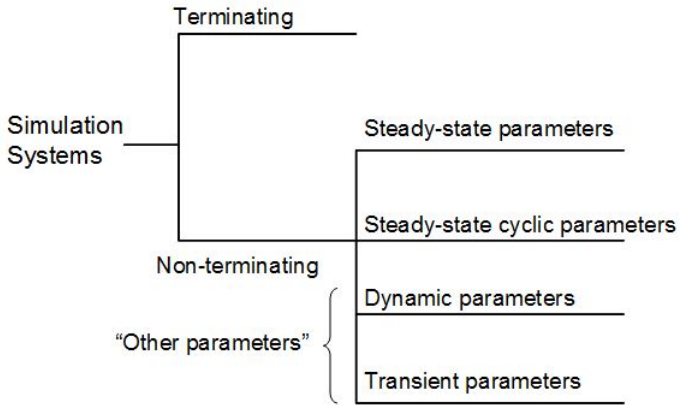


Figure 3.4: "Types of simulations with regard to Output Analysis". Figure.

Studying the output of a simulator plays a role as important as that of making it - especially when using empirical or "fitted" standard distributions. In order to analyze the data accumulated by the simulator, it needs to be classified with respect to its terminating conditions. Figure 3.4 shows the various possibilities. The two basic types here are terminating and non-terminating. **Terminating** simulations are those for which there is a "natural" event E that specifies the length of each run. Because of the limited operation that such simulators have, designing them aims to study the operation control. It should be noted that in terminating simulators only conditions that are specific to the system should be set as initial since they affect the measures of performance. **Non-terminating** simulators are the ones for which there is no event specifying the run. They have no concrete duration of execution, thus their focus is on long-term problems.

For non-terminating steady-state systems the following parameters are of interest:

- steady-state parameters
- steady-state cycle parameters - certain parameters change with given cycles during simulation

- dynamic parameters - certain parameters change over time, but not cyclic
- transient parameters - there are certain significant "events" that interrupt the operation of an otherwise steady-state system

The output of the simulator can be also viewed as a stochastic process. Therefore the analysis of the data can behave as a part of either a transient or a steady-state distribution. A **transient**state is an interval of time in which our system is either "warming up" or taking its time to respond to progress. **Steady-state** is the opposite of transient. Steady-state is a condition where our system continues with an easily predictable behavior and few values of it are changing (if any are changing at all). Figure 3.5 gives a comparison between both distributions.

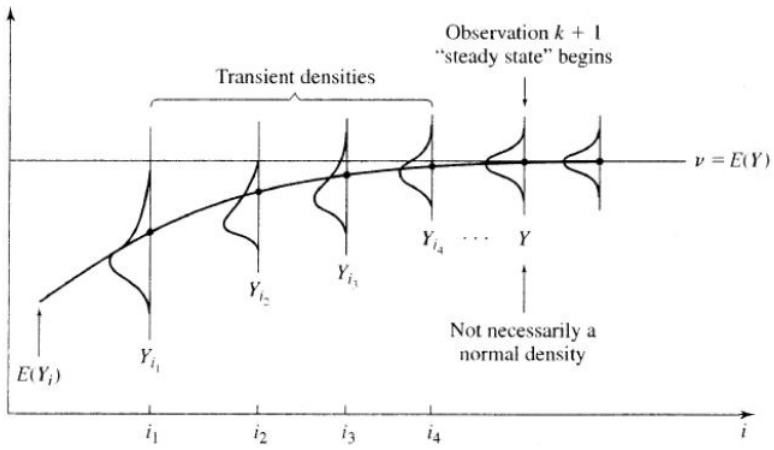


Figure 3.5: "Transient and steady-state density functions". Figure. [LK99]

3.4 Simulation paradigms

Previously the report presented separated systems in two types - continuous and discrete. This differentiation can also be done with respect to the simulation paradigms. Explanation to both types is provided in the form of two examples. Firstly, a simulation of a given system which evolves through time is of interest. A system like that is the weather forecast. The humidity in a city changes daily - continuously with respect to time. The easiest way to view this would be to make a diagram of the humidity values - most likely a continuous curve. Thus events that are simulated are continuous hence the name **continuous-event**

simulation (CES).
As a counter example is to examine the queue in a cinema. Customers arrive at the counter, they get serviced and leave. Variables of interest could be the time spent waiting in line, service time, number of waiting customers, etc. Plotting this would result in multiple continuous lines with breaks in between them. This definition describes a step function. The events - the change in the number of customers - are discrete variables. The name of this type of simulation is **discrete-event** (DES).

The nature of the TTEthernet protocol classifies its simulation as a discrete-event one. Therefore the rest of this chapter looks closer at it and the different approaches (world views) to developing such a simulation.

3.4.1 DES

To reiterate, a discrete-event simulation involves modeling a system as its state variables change instantaneously at separate points in time. Being discrete, however, limits the changes in the simulation to a countable number of points in time.
As simulation needs to keep track of multiple parameters, it also needs a mechanism that advances time. The variable keeping track of time (in simulation specific time units) is called **simulation clock**. Depending which of the two approaches discrete-event simulations use, they are divided into two - using the **next-event time advance** or the **fixed-increment time advance** approach.

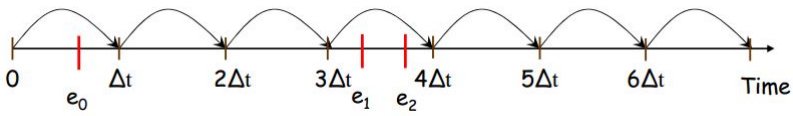


Figure 3.6: "Fixed-increment time advance". Figure. [LK99], 9p.

The fixed-increment time advance approach splits time into smaller increments. It starts off from time 0 and continues incrementing the time units until a predefined max. While it is a subcase of the next-event time advance approach, the focus is shifted to the advancing time. Every time unit is simulated to facilitate a possible action. That is why it is also known as the **activity-oriented paradigm** (see [Mat08]). Clearly, such a program is going to execute a lot of "empty" simulation runs thus a greater extent of the time will be wasted. These will produce no change in the state to the system - wasting CPU time. This is a big issue in simulations nowadays since large scale simulations require a

tremendous amount of time run. Depicted in Figure 3.6 is the operation of the approach. Δt is the simulation specific increment. It is easy to see that only three events are actually simulated during the seven cycles.

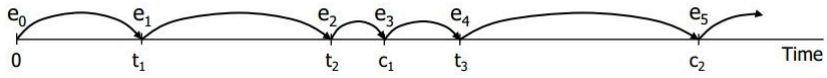


Figure 3.7: "Next-event time-advance approach". Figure. [LK99], 93p.

With the next-event time advance (**event-oriented paradigm**) approach the times of future events are explicitly coded into the model (stored into an event set) so that they arrive at a scheduled point in the future. The simulation starts by initializing the simulation clock to zero. As with the previous approach, simulation clock advances to the occurrence of the most imminent increment. The difference here is that the event that changes the state of the system, not the progress of time, is of interest. Thus when the next event is executed, the system is updated to point to next most imminent one. This process continues until a predefined event occurs. This way there will always be at least one event pending. Figure 3.7 shows the process.
On one hand this approach saves precious CPU time by skipping over periods of inactivity. On the other hand it introduces a new obstacle - find the earliest most imminent (minimum) operation within the event set. The fear of having too many wasted cycles is replaced by that of having too complex computations for ordering the events that are to be executed. These computations can be simplified if an appropriate data structure is used to maintain the event set.

Lastly comes the process approach (**process-oriented paradigm**) whose basis is the process[1]. The approach represents simulation in terms of the process' actions as they are created in the system. In the case of a cinema queue a process approach would involve having three processes - one simulating arrival of clients, one simulating the person of the counter and one thread managing the event set.

The master thesis project comprises of two simulators of the TTEthernet protocol. One is implemented with the action-oriented and the other with the event-oriented paradigm.

[1] In the process approach, the process is an idea similar to the notion of a Unix process. Modern systems represent it by the "lightweight" version of processes - threads.

Chapter 4

Simulator design and implementation

Chapter 4 gives insight into the actual implementation of the TTEthernet simulation. It discusses the requirements formulated for the master thesis project with respect to features, user interaction and performance. A section that describes the design decisions that were made, precedes the one dedicated to the implementation details. The later explains the common features between the activity-oriented and event-oriented version of the simulator. It also discusses their distinct characteristics separately.

4.1 Requirements

The goal the of master thesis project is to model and simulate the TTEthernet protocol. The primary requirements that were formulated include:

- use offline generated TT schedules as input files to the simulator. This is done in order to integrate it in an optimization loop as well as allow for comparison between the TTEthernet analysis and simulation

- model all three integration policies (conflict resolution mechanisms) - timely block, shuffling and preemption
- model the two simulation paradigms - action- and event-oriented
- model a terminating and a non-terminating (steady-state) simulator
- model a step-wise simulator that produces results up to the given point of the simulation
- determine the average end-to-end delays for all BE and RC messages
- determine the worst-case end-to-end communication delays for the RC messages.
- provide output in the form of GraphViz[1] files to visualize the topology given by the input files
- output a file containing the above mentioned delays
- depict the actual scenario that lead to the worst-case end-to-end delay for a given RC message
- compare and evaluate results from simulation to an existing analysis presented in a paper

4.2 Simulator design

In order to model a simulator for the TTEthernet protocol, one needs to make design decisions that abstract the model to a level that fits the requirements stated previously. These decisions, on the other hand, demand the formulation of a set of assumptions. According to them, the simulation characteristics are:

- a single TTEthernet cluster; simulation of a single clock synchronization domain
- a homogeneous network - no AFDX network components are present
- one message is carried by one frame; the terms "message" and "frame" are used interchangeably throughout the section
- there are no erros, packet losses and link failures

[1] Graphviz is open source graph visualization software.

- TT schedule input files contain all necessary information for generation of TT traffic
- each virtual link has a frame assigned to it
- a single "moment" is equal to 1ms
- it is possible that the dataflow links have no TT traffic scheduled on them

Based on the OMNeT++ INET framework, the simulator described in [SKKS11] covers a much broader set of characteristics of the TTEthernet protocol.

Taking into consideration the assumptions listed above, Figure 4.1 provides a simplified abstract view of the computerized simulator that portrays it as a "black box" which takes certain four files and user defined features as inputs, and produces a set of output files.

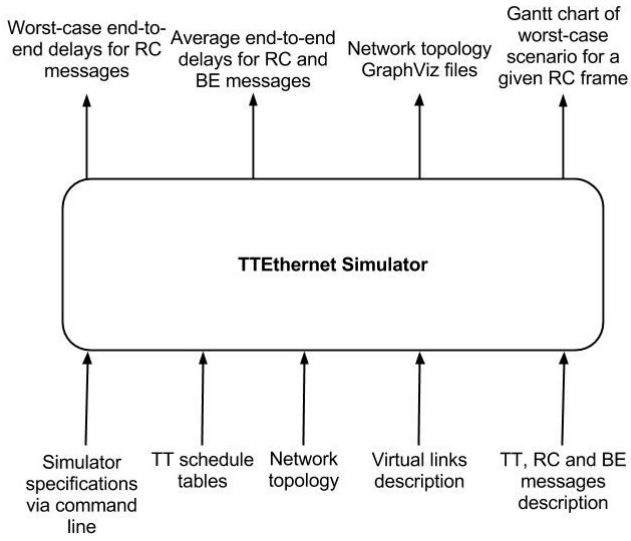


Figure 4.1: Abstract representation of the TTEthernet simulator

The "simulator specifications" address different characteristics of a simulation. Because the program is designed to have user interaction via command line, all of them are defined as a set of command line parameters. Their full list is:

- <network.graphml> - a GraphML[2] file that specifies the layout of the

[2] an XML-based format for representing graphs

network i.e. the ESs, Switches and one way edges (dataflow links) which connect them

- <schedule> - a file containing the TT schedules i.e. offline generated tables with the sending and receiving times on the dataflow links
- <vl file> - a file describing each virtual link as a composition of dataflow links
- <messages> - a file that specifies the parameters (id, size, deadline, VL to transmit on, period, rate) of TT, RC and BE messages
- <steady state> - specifies whether the simulation will run until a steady state (<true>/<false>)
- <delta> - only in case of a steady state simulation - the value that determines whether the simulation is in a steady state or not (<value>/leave blank)
- <steady state cycles> - only in case of a steady state simulation - number of cycles to observe for steady state behavior
- <cycles> - specifies the duration of the simulation (the number <cycles> until completion)
- <stepwise> - specifies whether this is a stepwise simulation (<true>/<false>)
- <integration policy> - shuffling/preemption/timely block
- <transmission speed> - speed in Mbps (Megabits per second)
- <record history> - records history of a given frame (<frame ID>/leave blank)

The input files allow for any combination of end systems, network switches, virtual links and messages to be simulated making the simulator a very generic tool.
The output from the simulator consists of three files:

- a comma separated value (CSV) file containing the worst-case and average delays
- a GraphViz file presenting the network topology from each virtual link's perspective
- a Joint Photographic Experts Group (JPEG) file showing a Gantt chart with the scenario that led to the worst-case end-to-end delay for a frame specified in the inputs

Based on the produced csv file we can determine the schedulability of the RC and BE messages given as input. Because of the nature of a simulation, certainty for scheduling a given set of messages cannot be given. In case of a missed deadline, however, messages can safely be defined as non-schedulable.

As shown in [TSPS12], the calculation of end-to-end delays comprises:

- queueing delays from higher priority messages
- network delays - the duration of the message's transmission on the dataflow links

These calculations are based to great extent on the object of interest - RC or BE messages - and the integration policy used for the simulation. The computation of RC worst-case end-to-end delay for timely block and preemption can be expressed in a more formal way by the following equations:

$$R_{f_i} = \sum_{v_j,v_k \in V,[v_j,v_k] \in vl_i} (Q_{f_i}^{[v_j,v_k]} + C_{f_i}^{[v_j,v_k]}) \tag{4.1}$$

$$Q_{f_i}^{[v_j,v_k]} = Q_{f_i,[v_j,v_k]}^{TT} + Q_{f_i,[v_j,v_k]}^{RC} + Q_{f_i,[v_j,v_k]}^{TL} \tag{4.2}$$

In Equation 4.1 R_{f_i} denotes the RC worst-case end-to-end delay for frame f_i transmitted on virtual link vl_i. $Q_{f_i}^{[v_j,v_k]}$ is the queuing delay and $C_{f_i}^{[v_j,v_k]}$ - the network delay. Equation 4.2 examines the queuing delay's components. The delay generated by the TT traffic starting from the time frame instance f_i arrives at network node v_j until the instance is sent to the next node is v_k denoted as $Q_{f_i,[v_j,v_k]}^{TT}$. The delay that f_i experiences from instances arrived before it in the FIFO RC queue is $Q_{f_i,[v_j,v_k]}^{RC}$. The technical latency is denoted as $Q_{f_i,[v_j,v_k]}^{TL}$.

In the case of shuffling the second equation looks like this:

$$Q_{f_i}^{[v_j,v_k]} = Q_{f_i,[v_j,v_k]}^{TT} + Q_{f_i,[v_j,v_k]}^{RC} + Q_{f_i,[v_j,v_k]}^{BE} + Q_{f_i,[v_j,v_k]}^{TL} \tag{4.3}$$

Equation 4.3 adds BE traffic as a source of delay because of the precedence low priority has over higher priority traffic.

Analogically equations can be derived for BE traffic as well.

$$B_{f_i} = \sum_{v_j,v_k \in V,[v_j,v_k] \in vl_i} (Q_{f_i}^{[v_j,v_k]} + C_{f_i}^{[v_j,v_k]}) \tag{4.4}$$

For timely block and preemption 4.3 is fully valid. The case with shuffling is different for the above mentioned reasons. The resulting equation is 4.5.

$$Q_{f_i}^{[v_j,v_k]} = Q_{f_i,[v_j,v_k]}^{BE} + Q_{f_i,[v_j,v_k]}^{TL} \tag{4.5}$$

The simulation is modeled such that a TCycle is equal to the Least Common Multiple (LCM) of all TT frame periods. If an RC and BE frame is scheduled to run after the TCycle, it will not be able to execute. It is also the case that if a lower priority frame doesn't manage to finish its transmission in time (i.e. before the end of the TCycle), it will not be able to execute and its worst-case end-to-end delay will stay unknown. After a series of tests, the simulation was modeled so that for each cycle given as input by the user, the simulator executes 4 TCycles, each having less and less RC and BE frames. The frames that are delayed simply execute on the subsequent TCycles with the accumulated LCM added to their delay. The ones that do not manage to finish even with the extra TCycles, experience the "snowball effect". This is the continuous inability to run (e.g. because of too big arrival times).

4.3 Implementation

The simulator was implemented using the Java programming language and the IntelliJ IDEA 12.0.4. It is designed as a standalone application with command line interaction.

The simulation starts by reading the command line parameters described in 4.2. The class *Program* is the access point to the simulation which provides feedback to the user in case of wrong number of command line arguments. The arguments themselves are relayed to the **SimulatorWrapper** class. It takes the input files, the simulation features defined by the user and processes them. This class also navigates the simulation and counts the time between the simulation runs (i.e. every 4 TCycles). Finally, SimulationWrapper replenishes the BE and RC messages that need to be transmitted.
The **Simulator** class initializes the simulation. As seen in Figure 4.2 this class is the heart of the whole program. It generates the network components (ESs, Switches, dataflow links), virtual links, traffic for the simulation, schedules for the dataflow links. It must be noted that the correspondence between the data

structures used and network components was kept straightforward. The class also keeps track of the simulation time, advances the simulation and does the actual transmission of the messages.

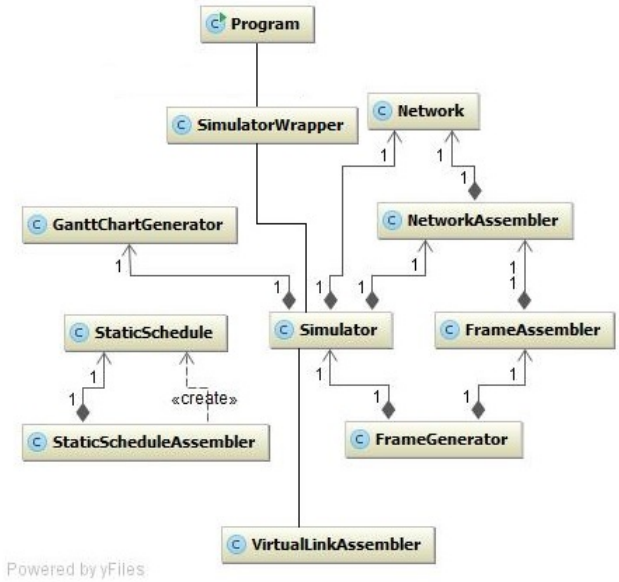


Figure 4.2: Simulator. UML diagram.

The first data structures created by the **Simulator** class are the ones holding the network components. Each of the components extends the **NetworkElement** class. A **Network** class aggregates **NetworkNode** (either **EndSystem** or **Switch**) and **DataflowLink** objects. Figure 4.3 gives an overview on the hierarchy on the network elements with their respective abstract classes and interfaces.

To keep the graph representation of network components from the GraphML input file, the simulator uses the Java Universal Network/Graph (JUNG)[3] and the Network class extends the DirectedSparseMultigraph coming from JUNG. By extending NetworkNode, the EndSystem and Switch classes get their main methods - **addTraffic()** and **simulateMoment()**. They are defined in the **INetworkNode** interface. addTraffic() adds arriving frame instance to an ES or Switch queues and marks their time of arrival. simulateTime() is called

[3] a Java open-source library of graph modeling and visualization

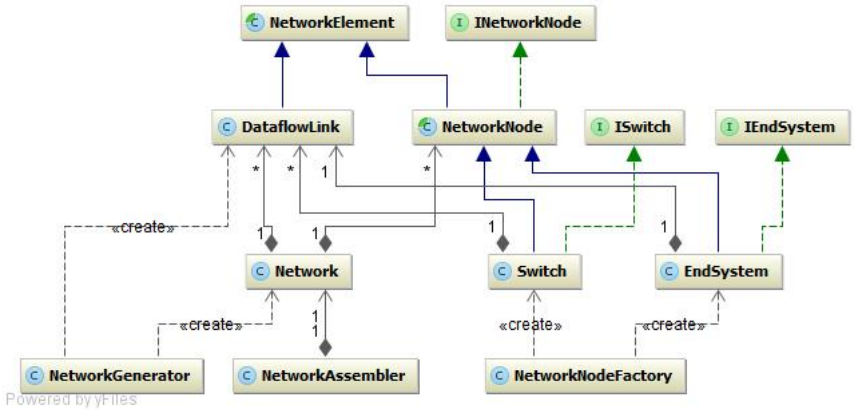


Figure 4.3: Network. UML diagram.

every time a moment has to be simulated by a given network node. It takes into account the integration policy used and compares the priority of the incoming traffic and to the one on given dataflow link.

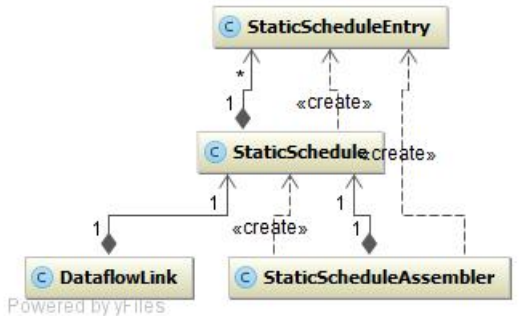


Figure 4.4: Dataflow links use StaticSchedule. UML diagram.

The DataflowLink class assigns the schedules for the TT messages, read from the input file, to the dataflow links. It helps to implement the integration policies by comparing the simulated moment to the upcoming moment for TT transmission. The schedules themselves are abstracted by the **StaticSchedule** class.

The generation of network components is followed by reading the virtual links input files by the **VirtualLinkAssembler**. It generates all virtual links and stores them in hash maps. The **VirtualLink** class is the actual model class

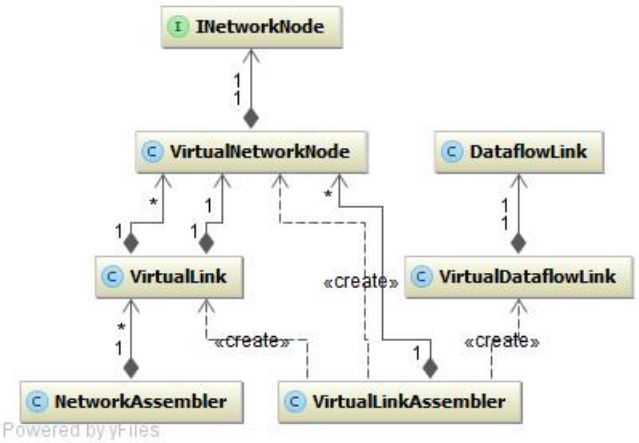


Figure 4.5: Virtual links. UML diagram.

designed as a tree structure with a **VirtualNetworkNode** at one end and a Set of nodes at the other. It should be noted that the class extends the JUNG DirectedSparseMultigraph as well. Because its edges and vertices are IDs of the network elements on the its dataflow paths, a virtual link can be represented as a subgraph of the network. The supplementary classes **VirtualDataflowLink** and VirtualNetworkNode are model classes used to abstract the DataFlowLink and NetworkNode classes respectively.

The extraction of message info from the designated file is done from the **FrameAssembler** class. This class orchestrates the process of frame generation by calling the **FrameGenerator** (the class that does the actual work), storing the created frames and generating corresponding **FrameInstance** objects for each frame. As stated in the assumptions, one message is represented by a single frame. Thus when a line from the message input file is read, one of the three implementations of the **Frame** abstract class are used - **TTFrame**, **RCFrame** and **BEFrame**. The factory design pattern was found best suited for the task. The Frame class is a container for all common parameters of the traffic classes. These are *frameID*, *size*, *deadline*, *type* and *virtual link ID*. The abstract class **LessCriticalFrame** further refines RC and BE messages and gives them a couple of extra features. These are transmission *rate*, *frameInstances* belonging to the frame and *maxDelay* - the worst-case end-to-end delay of the messages. Hierarchy of the messages is shown in Figure 4.6.

FrameInstance objects are those who are actually transmitted in the network during the simulation. They keep a linked list of dataflow links which shows

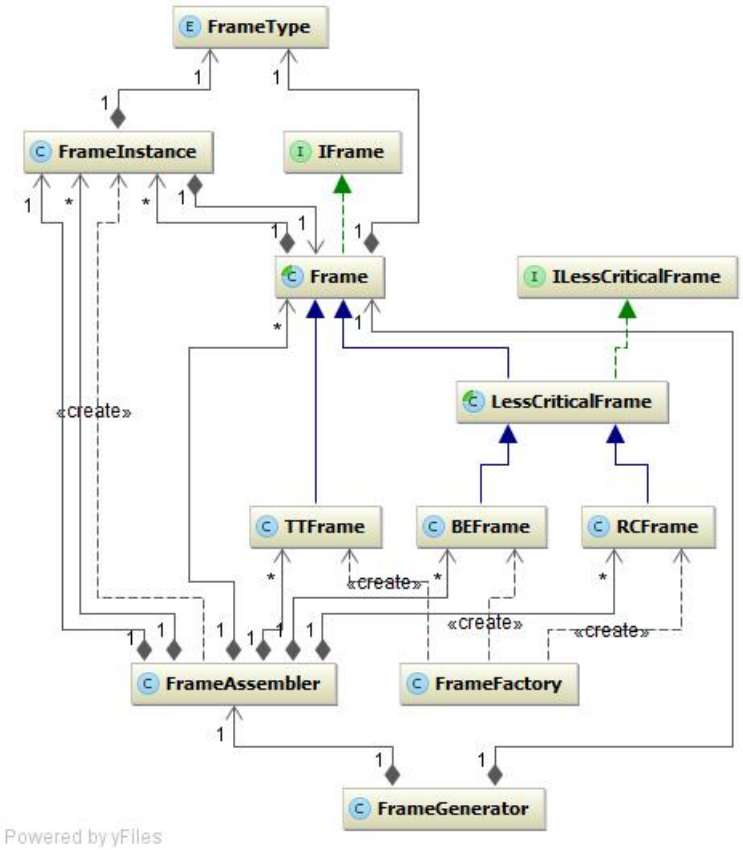


Figure 4.6: Messages. UML diagram.

them their way on the virtual link. They also calculate the time the have left until being transmitted to the next link. All the fields of the FrameInstance class is given in Figure 4.7.

After the initialization phase of the simulation comes the first branching moment. Depending on the user's input the simulator runs either a regular or a stepwise simulation (Figure 4.9). When this step is done, the **SimulationWrapper** starts the first run from the cycles as a command line parameter. As previously described, the simulation runs 4 TCycles for every such cycle. Step 1 in Figure 4.10 refers to the stepwise or regular simulation shown later in this section. In Step 2 all the queues containing information about the previous TCycles are cleared and all RC and BE messages are regenerated. Step

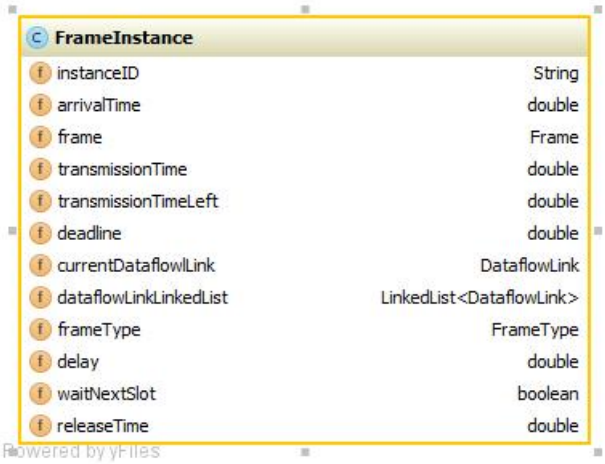


Figure 4.7: Frame instance

3 also clears the queues of all network nodes in the simulation. The difference with step 2 is that it regenerates only those messages that were not able to get transmitted during the previous TCycle. The output files are produced, once the all simulation cycles are over (step 4).

The stepwise simulation can be called an "interactive" mode of simulation. It enables the user to issue commands to the simulator via command line in order to get the latest information about the schedulability of the RC and BE frames. The available commands are:

- begin (press **b**) - start the stepwise simulation
- pause (press **p**) - pause the stepwise simulation
- continue (press **c**) - resume the stepwise simulation
- stats (press **s**) - produce statistics(worst case end-to-end delay for a frame instance and delay for all RC and BE frames)
- exit (press **e**) - permanently stop the simulation

Figure 4.8 depicts the activity diagram for the stepwise simulation. After time is initialized in step 1, the program enters a while loop that listens for the user's input. If he/she presses **p**, the simulation is paused (step 2). Step 3 shows the situation where the simulator produces data after the key **s** is pressed. Step 4

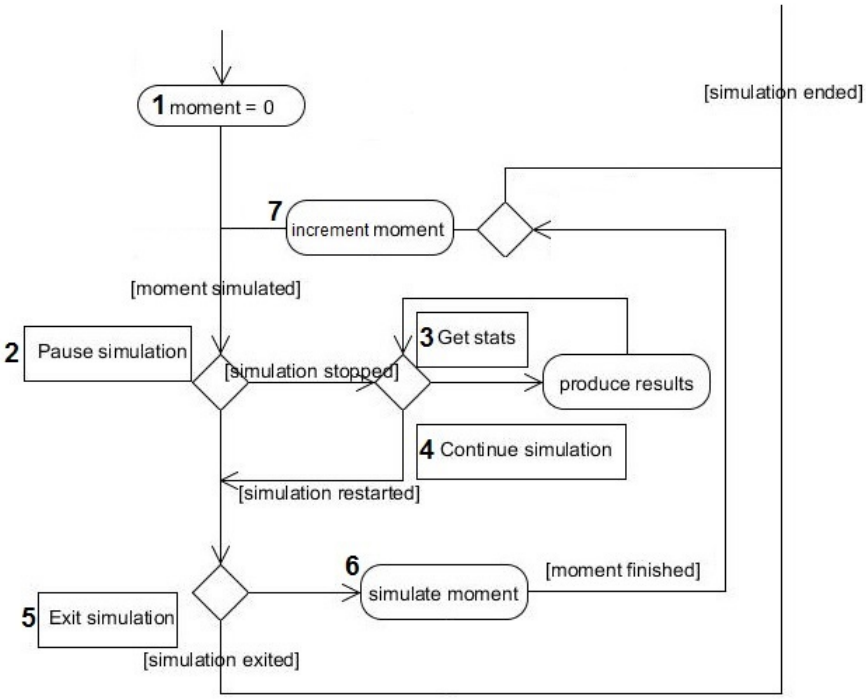


Figure 4.8: Stepwise simulation. Activity diagram.

follows the pressing of key **c**, which resumes the simulation. The press of **e** ends the simulation prematurely (step 5). If there is no input from the user a moment is simulated (step 6). This step represents the activity and event-oriented implementation of the regular simulator. If this is the last moment before the TCycle is reached, the simulation finishes. If not - the moment is incremented to the next ms and the loop is repeated until one of the terminating conditions is met. They are discussed in the paragraph below.
From Figure 4.8 it is visible that a press of a key cannot interrupt the actual simulation of a cycle. Depending on whether the program is currently in the process of simulating or not, the simulator acknowledges the input of a key right away or does that once the simulation cycle is over.
The Java Threads package was used for the implementation of the stepwise simulatin. Figure 4.11 shows the classes implementing the stepwise simulator logic. Managing the whole process of simulation is the **StepwiseSimulator** class. It reads the input from the keyboard and navigates the simulation. The actual passing of time is registered in the **ConsoleThread** class. It increments the number of ticks (i.e. moments) and for each one calls the *regularSimulator()*

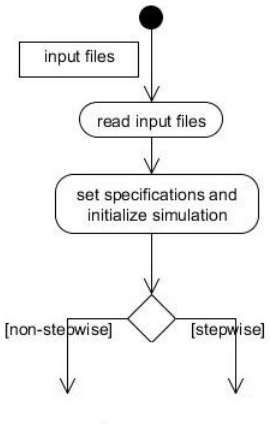


Figure 4.9: Simulation initialization. Activity diagram.

function of the Simulator class. The thread listens for activity from the user and if receives any, it stops its working process and returns control to the StepwiseSimulator for an appropriate action. If not - the ConsoleThread looks at the terminating conditions and determines when to stop the simulation. This can happen in the following cases:

- the TCycle is reached
- steady-state is reached by all RC frames
- the exit key has been pressed

The steady-state is a property of a system described in Section 3.3.3. The following expression 4.6 describes the properties that the RC frames have to satisfy in order for the simulation to reach a steady-state:

$$(f_{m_j}^{i-1} - f_{m_j}^{i}) \leq \triangle where \forall j \in RC \tag{4.6}$$

The expression states that when examining a currently transmitted RC frame instance, the difference between the delay of the last transmitted frame instance and that of the current one, must be less or equal to the DELTA given by the user. If all the frame instances preserve the expression for a user defined amount of cycles, it can be said that the simulation has reached a steady-state.
Implementing this task required the use of a hash map that keeps track of each RC frame ID and its latest end-to-end delay. When all frames respect the upper

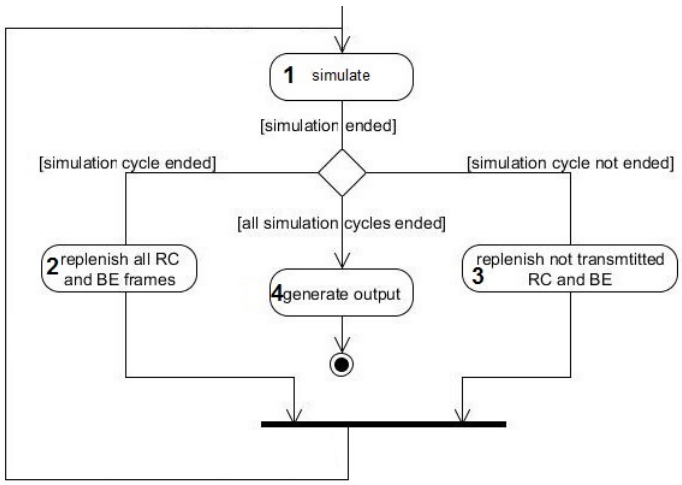


Figure 4.10: Main simulation loop. Activity diagram.

bound set by the DELTA value for the defined number of cycles, the simulation is stopped and a message stating that a steady-state has been reached is generated.

No matter if the user decides to run a regular or a stepwise simulation, the simulator always uses the *regularSimulator()* function of the Simulator class. The differences in the implementation of this function for the two simulator paradigms will be described in the corresponding subsections. The common thing about both versions, is that a moment is always being simulated in the end systems or switches.

The method simulateMoment() is common to both end systems and switches. The pseudo code in Algorithm 1 describes its path of execution.
The first two lines show that the silence period after a frame instance was transmitted and a TT instance occupying the dataflow link have highest priority.
Lines 3-6 deals with the transmission of a TT instance. First the entry is taken from the schedule made for the dataflow link. It is used to generate a frame instance "on the fly", which is later on transmitted. The frame checks the integration policy if another frame is occupying the dataflow link and makes a decision based on that.
Lines 7-31 explain how ES handle RC traffic. If there are frame instances waiting to be transmitted, their arrival time is checked. If they have passed the technical latency - they advance to the next step. In it, the algorithm checks whether the frame instances respects the BAG. The last step is to take the first frame instances from all the once that made it so far. It is checked against the

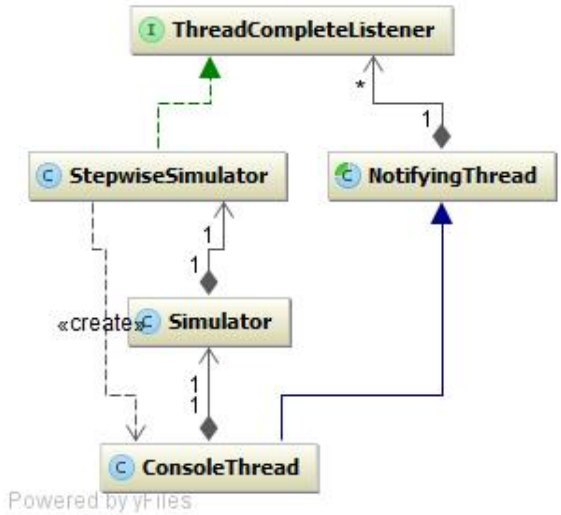


Figure 4.11: Stepwise simulator. UML diagram.

integration policy.
From line 32 and on, the algorithm examines the BE traffic. It behaves exactly the same as in the case with the RCs with the exception of the BAG check.
This same algorithm is used in the Switch implementation as well, with the difference that it has to be run for all the dataflow links (egress links) that the switch is connected to.

The algorithms 2, 3 and 4 explain the implementation of the three integration policies. They are modeled as described in [SBH+09].
The **Timely Block** algorithm is self-explanatory. All it does is makes sure that no RC or BE frame instance gets to transmit unless there is absolute certainty that it will be finish before a TT instance arrives. In the case of an RC instance after being sent for transmission on the dataflow link, the start time of the transmission is recorded. The BAG uses this time for each virtual link to enforce temporal separation. Time is not of interest and is thus not recorded when a BE instances is about to transmit.
With the algorithm for **Preemption**, a TT instance always preempts the other to traffic classes and marks itself as delayed. Preemption is done by removing the transmitting instance from the dataflow link and triggering silence in its place. This procedure is also utilized in the case where an RC instance finds that a BE is transmitting. If this is not the case - the RC transmits and saves its transmission moment. BE instances do not preempt other classes and transmit

Algorithm 1 Simulate time in an End System

```
1: if egressLink is silenced or egressLink is TTBusy then
2:     skip moment
3: else if egressLink should transmit TT then
4:     scheduleEntry = getReleasedEntry(moment)
5:     frameInstance = generateTTInstance(scheduleEntry)
6:     if egressLink occupied then
7:         checkIntegraionPolicy(frameInstance, egressLink, endsystem, moment)
8:     else
9:         transmit(frameInstance, egressLink, moment)
10:    end if
11: else if egressLink should transmit RC then
12:    for all vl_i ∈ VlRCQueue do
13:        frameInstancesList = readyRCInstances(vl)
14:        for all f_i ∈ frameInstancesList do
15:            if moment ≥ f_i.arrival + technicalLatency then
16:                readyInstancesList.add(f_i)
17:            end if
18:        end for
19:    end for
20:    if readyInstancesList is not empty then
21:        for all f_i ∈ readyInstancesList do
22:            lastTransmForVl = transmissionMap(f_i.virtualLink)
23:            if moment ≥ lastTransmForVl + BAG then
24:                readyInstancesList2.add(f_i)
25:            end if
26:        end for
27:    end if
28:    if readyInstancesList2 is not empty then
29:        frameInstance = readyInstancesList2.item(0)
30:        checkIntegraionPolicy(frameInstance, egressLink, endsystem, moment)
31:    end if
32: else
33:    for all vl_i ∈ VlBEQueue do
34:        frameInstancesList = readyRCInstances(vl)
35:        for all f_i ∈ frameInstancesList do
36:            if moment ≥ f_i.arrival + technicalLatency then
37:                readyInstancesList.add(f_i)
38:            end if
39:        end for
40:    end for
41:    if readyInstancesList is not empty then
42:        frameInstance = readyInstancesList2.item(0)
43:        checkIntegraionPolicy(frameInstance, egressLink, endsystem, moment)
44:    end if
45: end if
```

directly.
Lastly, the algorithm for **Shuffling** is discussed. In it a TT frame can be preempted by any other traffic class. Thus in the implementation the TT instance directly marks itself as delayed. RC instances can preempt all classes but BE, so the algorithm checks whether a BE is being currently transmitted. If so - the moment is skipped. If not, in the case a TT is transmitting - it marked as delayed. In the other cases an RC transmission commences. BE frame instances behave exactly as in Preemption.

The strategy design pattern was utilized in the implementation of the Integration policy. Figure 4.12 shows UML representation.

Algorithm 2 Timely Block

```
1: freeTime = egressLink.getUninterruptedTimeUntilNextTT()
2: if frameInstance.type is RC then
3:     if freeTime ≥ frameInstance.transmissionTimeLeft() then
4:         transmit(frameInstance, egressLink, moment)
5:         save transmission moment for virtual link
6:         remove frame instance from waiting queue
7:     else
8:         skip moment
9:     end if
10: else
11:     if freeTime ≥ frameInstance.transmissionTimeLeft() then
12:         transmit(frameInstance, egressLink, moment)
13:         remove frame instance from waiting queue
14:     end if
15: end if
```

Algorithm 3 Preemption

```
1: if frameInstance.type is TT then
2:     remove instance from egressLink
3:     transmit silence on egressLink
4:     mark self as delayed
5: else if frameInstance.type is RC then
6:     if egressLink has BE transmitting then
7:         remove instance from egressLink
8:         transmit silence on egressLink
9:     else
10:        transmit(frameInstance, egressLink, moment)
11:        save transmission moment for virtual link
12:        remove frame instance from waiting queue
13:    end if
14: else
15:    transmit(frameInstance, egressLink, moment)
16:    remove frame instance from waiting queue
17: end if
```

Algorithm 4 Shuffling

```
1: if frameInstance.type is TT then
2:     mark self as delayed
3: else if frameInstance.type is RC then
4:     if egressLink does not have BE transmitting then
5:         if frameInstance.type is TT then
6:             mark self as delayed
7:         end if
8:         transmit(frameInstance, egressLink, moment)
9:         save transmission moment for virtual link
10:        remove frame instance from waiting queue
11:    else
12:        skip moment
13:    end if
14: else
15:    transmit(frameInstance, egressLink, moment)
16:    remove frame instance from waiting queue
17: end if
```

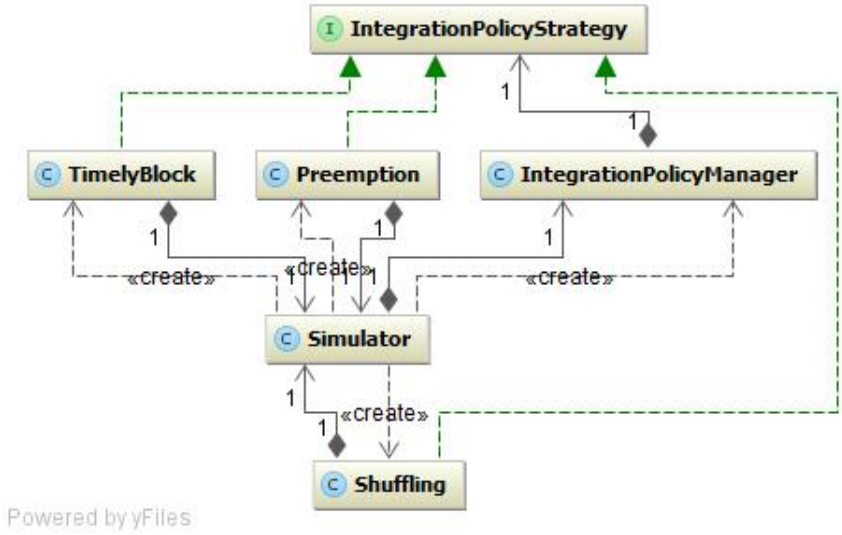


Figure 4.12: Strategy design pattern for Integration policy. UML diagram.

4.3.1 Activity-oriented simulator

The simulator about to be described implements the activity-oriented paradigm (see 3.4.1. This paradigm splits time into equal units (in this case ms) that are continuously incremented until the total amount of units is simulated. Thus the focus is shifted to the advancing time.

Figure 4.13 depicts the be algorithm flow. The paradigm is implemented in the **Simulator** class as a simple for loop with a counter initially equal to 0 (step 1). The simulation process for each moment encompasses a series of events. First, the algorithm checks whether there are any frame instances scheduled to arrive at this point in time. If there are - they are added to the waiting queues of the designated ESs (step 2). If not - the simulation continues with step 3. In it the *simulateMoment()* (see 1) of every ES and Switch is invoked in order to transmit any waiting frame instances. The final step of the loop is to decrement all the counters keeping track of the frame instances transmission by one (step 5). If there are no instances that have finished their transmission on the dataflow links, the simulation of this moment is finished. The for loop then checks if the previously simulated moment was the last one. If not - the moment is incremented by one and steps 2 to 5 are repeated. When after the last cycle the simulation returns control **SimulatorWrapper** class which issues the generation of the output files.

Figure 4.14 takes a closer look at the situation where a frame instance finishes its transmission on a given dataflow link. If the simulation is run with the command line parameter specifying a RC frame whose worst-case scenario is sought, the simulator gathers data if a has been transmitted RC instance. There are two cases, depending on whether the frame instance has reached its destination on the virtual link or not. In the later, the frame instance is simply added to the waiting queue of the next network node in its path (step 8).
In the former the simulator calculates the end-to-end delay for the instances. It checks whether this value is greater than the current maximum delay for the frame and replaces it if that is the case (step 9).
If a steady-state simulation (see expression 4.6) is being run and a RC instance was transmitted, the algorithm compares the delay previously recorded for the frame and the one that the current value experiences. When the DELTA was not respected the counter keeping track of instances reached a steady-state is set to zero and the process is restarted. If the difference between the instance delays is smaller or equal to DELTA, the counter is increment by one. If all frames have reached a steady-state, the simulation exits.
In step 10 silence is sent on the dataflow links. This process does not depend on the type of traffic that finished transmitting.

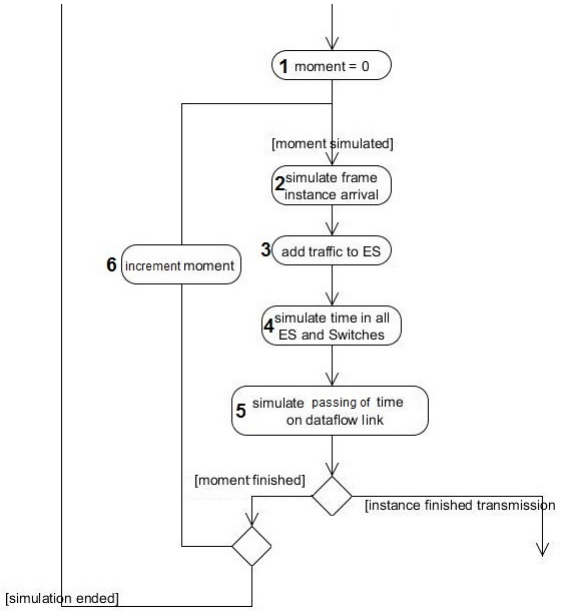


Figure 4.13: Simulate moment. Activity-oriented implementation. Activity diagram.

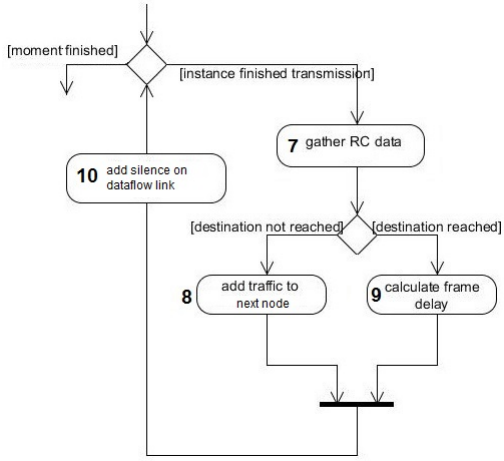


Figure 4.14: A frame instance finished transmitting. Activity-oriented implementation. Activity diagram.

4.3.2 Event-oriented simulator

The second simulator is implemented using the event-oriented paradigm. In it the times of future events are stored into an event queue and arrive at predefined points in the future. When an event is executed, the system is updated to point to next most imminent one. Thus the event, not the progress of time, changes the state of the system.

Figure 4.15 shows classes that are specific to the event-driven simulator. The **Event** class is the core of the simulation as it bears the main characteristics of an event - time and type. There are six types of events:

- `ARRIVAL_RC_BE`
- `RELEASE_RC_BE`
- `RELEASE_TT`
- `FINISH_TT`
- `FINISH_RC_BE`
- `SILENCE`

The simulation is represented by an event queue. All TT frame release and finish times are initially loaded into the event queue which is sorted by time. This distinguishes them from the RC and BE frames which are taken from a separate queue whenever a frame instance is not running. Because of this fact, there is a need for separation between the release and finish events of TT and the other two traffic classes. As with the action-oriented paradigm, arrival of RC and BE frames needs to be modeled - henceforth the `ARRIVAL_RC_BE` type.

The classes that extend the Event class are three - **ReleaseEvent**, **SilenceEvent** and **TransmitEvent**. They are modeled based on the different resources they need - network node, dataflow link and frame instance respectively.

The simulation process is defined as a while loop that continuously picks the first event in a sorted queue. After an event has finished its job the process is repeated until on of the aforementioned terminating conditions is met.

Figure 4.16 shows the case where a TransmitEvent with `ARRIVAL_RC_BE` is received. The process is similar to the one described in the previous subsection. First, the arrival of an instance is simulated (step 1). It is important to note

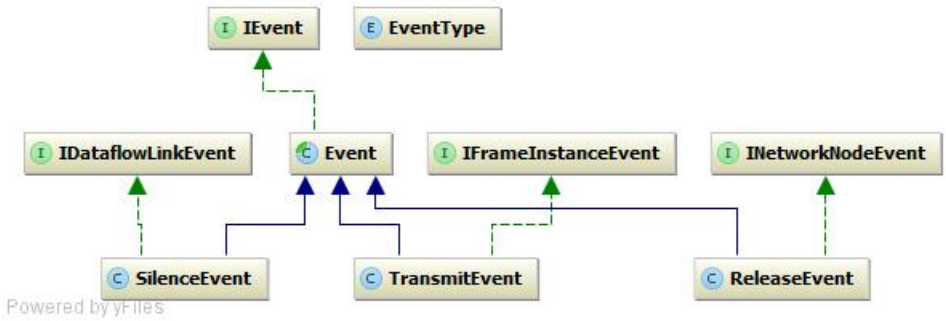


Figure 4.15: Event. Class diagram.

that a TransmitEvent carries a single frame instance. That instance is added to the waiting queues of the designated ES (step 2). The last part of the process involves calculating the proper time to release the instance (step 3). If the dataflow link that it needs to transmit on is currently free or will be occupied until less than the length of the technical latency, the algorithm sets the sum of the current moment and the technical latency as the release time. If it is occupied for a longer period, the release time set is equal to the moment when the dataflow link becomes free.

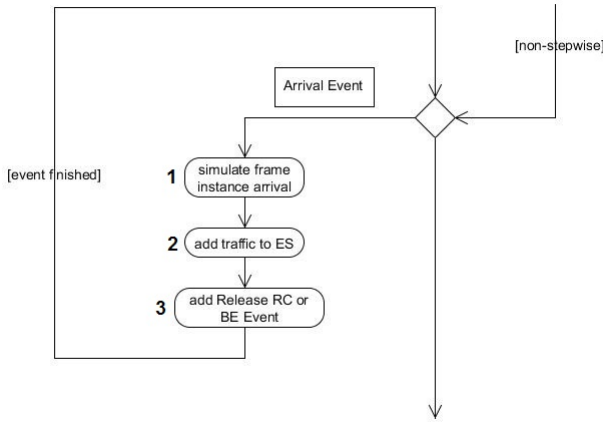


Figure 4.16: Arrival event in event-oriented implementation. Activity diagram.

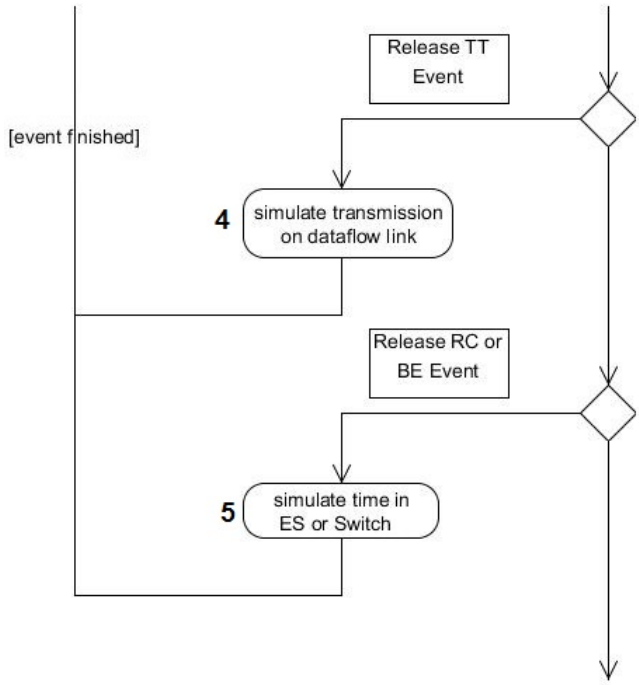


Figure 4.17: Release of TT event and RC or BE event in event-oriented implementation. Activity diagram.

The release event types `RELEASE_RC_BE` and `RELEASE_TT` have similar mechanics. That is way the have been grouped in Figure 4.17. Step 4 examines the event of the release of a TT instance. If the dataflow link of interest is currently free the frame instance transmits directly. If not - it acts according to the integration policy chosen.
When a RC or BE ReleaseEvent is received, the algorithm simply calls the *simulateMoment()* method of the network node that the instance currently resides in (step 5).

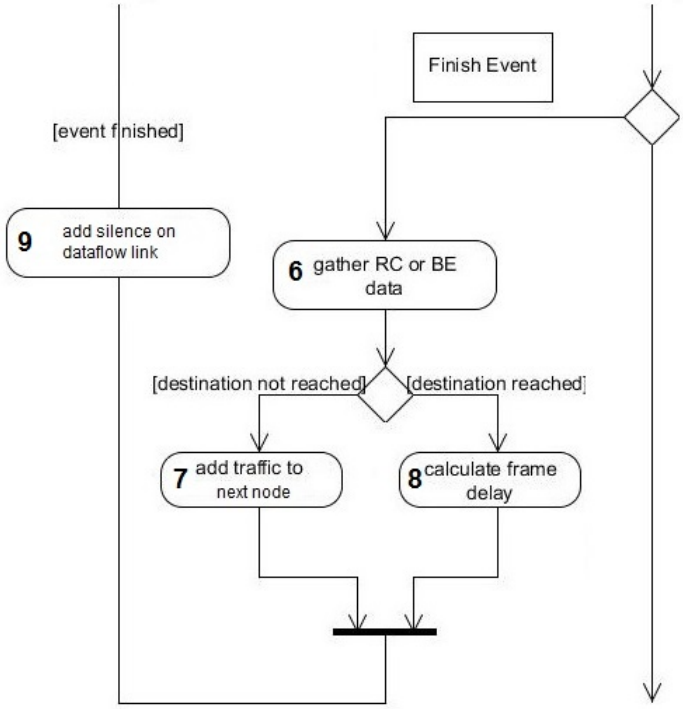


Figure 4.18: Finish event in event-oriented implementation. Activity diagram.

The two event types `FINISH_RC_BE` and `FINISH_TT` are depicted as one in Figure 4.18. The activity diagram shows that the two event types have the same purpose. The difference is that a finishing TT instance is used to trigger the transmission of delayed TT instances (in the case of Preemption or Shuffling). Besides that the procedure that follows has already been described to great extent in the action-oriented section above.
What should be noted is how step 7 is executed. When a frame instance needs to be added to the next node on its path, its release time should be calculated appropriately. In order to do that the algorithm continuously check the spaces in the schedule on the future dataflow link to see whether they are enough for transmission. When such a space is found, its first moment is set as the release time of the frame instance.
In step 9 an event that SilenceEvent for the previously occupied dataflow link is issued.

When a SilenceEvent arrives, the dataflow link that is silenced is specified within the event. The code simply frees it. Figure 4.19

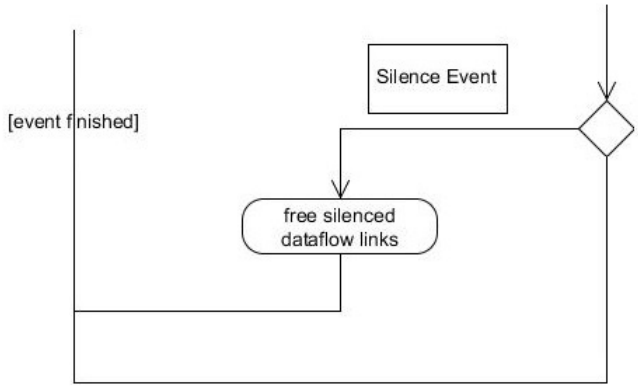


Figure 4.19: Silence event and end of simulation. Activity diagram.

Chapter 5

Testing and Evaluation

This chapter describes the techniques used to verify the actual working of the two simulators. It also provides a comparison made between the results of the analysis given in [TSPS12] and output of the simulators.

5.1 Testing

The process of building the model of the TTEthernet protocol and verifying it required continuous testing via various techniques. The implementation began by small trial-error tests of the basic functionality (e.g. the reading of the input files). The next step was to incorporate unit test for the generation of the network components (e.g. creation of virtual links, ESs, Switches, etc.). Their goal was to proof the correctness of the program and lay a solid foundation for the simulators.

After all data structures behaved accordingly, the stage of continuous "development-test" of the simulator was entered. Various small pen and paper examples were tested after the simulator had generated its output in order to determine this its accuracy. Output of log files to the console as well as standard debugging tech-

niques proved very useful. A milestone, achieved during the process of testing, is the output gained from the files described in the [TSPS12].

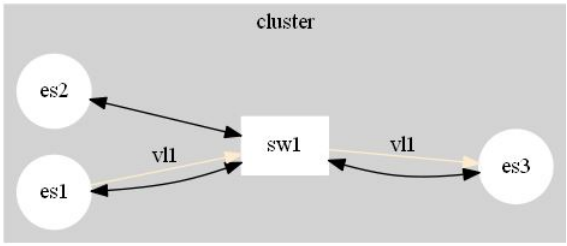


Figure 5.1: Network with vl1

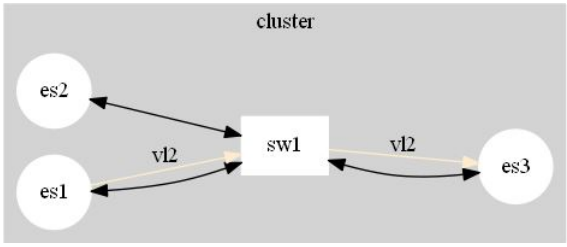


Figure 5.2: Network with vl2

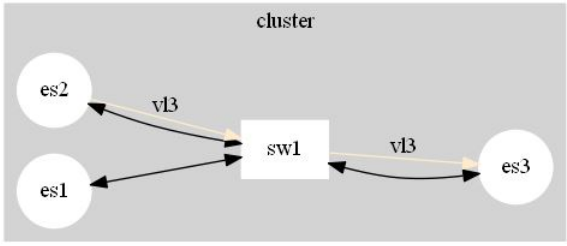


Figure 5.3: Network with vl3

Figures 5.1, 5.2, 5.3 are a visual representation of the network provided in the file containing the virtual links' description. A comparison between the three figures and 2.16 shows the same exact topology, viewed from each virtual link's perspective.

Figure 5.1 presents the route of the frame instance with the worst-case end-to-end delay. The route is depicted with the dataflow links that the instance passes through on Y-Axis (es1sw1, es2sw1, sw1es3) and time progression on the X-Axis. In this scenario the slowest instance is rc1.0_0. A direct comparison between this figure and 2.17 will reveal some differences. The absence of some

Frame	Worst-case Delay	Average Delay
rc1.0	471.0	235.5

Table 5.1: Worst-case results

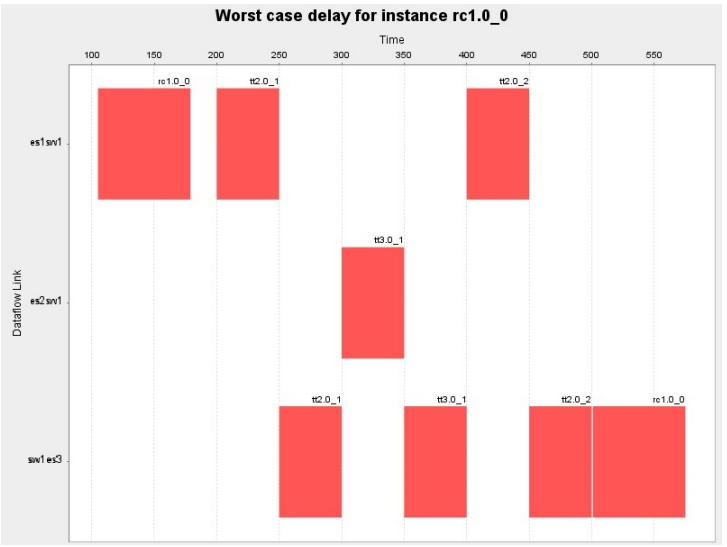


Figure 5.4: Progression of time on the dataflow links

instances and is due to the fact that figure 5.1 displays only the route of frame instances that influenced its delay.
The results from Table 5.1 could also seem strange at first. The worst-case delay time of rc1.0_0 from the table and 2.17 differ by one time unit. This is due to the fact that the granularity used in the simulator is finer (i.e. uses higher floating point precision). The average delay time, on the other hand, is half of worst-case because the simulation finishes before rc1.0_1 gets a chance to finish. Thus rc1.0_1 keeps its initial time when summing up the delay values.

After the results from this comparison, work on the simulators was continued in the previous manner of development for the rest of the implementation period.

5.2 Evaluation

In order to evaluate the results output from the simulator, this section provides three comparisons between:

1. the delay difference for all three traffic classes for all integration policies run for 10 simulation cycles of the action-oriented simulator
2. the two simulators run for 10, 100 and 1000 simulation cycles with the Timely Block integration policy
3. the 10, 100, 500, 800, 900 and 1000 simulation cycles of the action-oriented simulator with the Timely Block integration policy
4. two real world test cases based on the NASA's Orion Crew Exploration Vehicle
5. the results with the biggest worst-case end-to-end delay from point 3 and the TTEthernet analysis

The simulations where run on two environments. The event-oriented simulations where done on a laptop DELL Inspiron N5510 with Intel Core i7-2630 CPU and 6 GB RAM. The smaller action-oriented simulations where executed onto the laptop and the larger ones on the DTU Computing Clusters.

First a comparison between the delay difference of TT, RC and BE frames for all three integration policies is observed. The action-oriented simulator was run for 10 cycles using a single test case. Its characteristics are described in Table 5.6 under number **3**.

TT Frame	Timely Block [s]	Shuffling [s]	Preemption [s]
tt1.0	118	300	150
tt7.0	113	261	160
tt35.0	118	192	140

Table 5.2: Comparison between the TT frame WCD for all three integration policies

Table 5.2 depicts three TT frames that give an overall view of the results for time-triggered communication. Due to the fact that TT frames have highest criticality and highest priority, the difference between the delay times is in the order of seconds. As expected the smallest delay times are experienced in the first column - the one with the values for Timely Block. The implementation of

this policy provides that no RC or BE message can interfere with the transmission of a TT message. With shuffling the values increase because a TT frame can be preempted by any other traffic class. The fact that a delayed frame has precedence over all other frames accounts for the greater delays. With preemption TT frames experience a small but visible increase in delay as well. The reason for that is the silence period added after the preemption by a traffic with lesser priority.

RC Frame	Timely Block [s]	Shuffling [s]	Preemption [s]
rc7.0	1126	547	252481
rc22.0	149	252149	214
rc28.0	200	252137	261

Table 5.3: Comparison between the RC frame WCD for all three integration policies

Figure 5.3 presents a comparison between the worst-case end-to-end delays of three RC frames. With Timely Block, a RC frame instance attempts a transmission over the dataflow link only if the time needed for that is available. Given that the right moment doesn't appear in the first tries, this leads to additional delay for the instance. In case of shuffling RC messages are able to execute when they find a TT instance transmitting. Delayed TT instances and BE instances, however, cause a great increase in the WCD of a RC message. Preemption has almost the consequences for RC frames as Timely Block. The difference is here is that RC frames can attempt transmission in any time. In the case of an incoming TT instances, however, they get a delay of a silence period plus the duration of the TT instance.

BE Frame	Timely Block [s]	Shuffling [s]	Preemption [s]
be11	769	289	300
be13	885	446	467
be20	965	292	252321

Table 5.4: Comparison between the BE frame WCD for all three integration policies

Lastly, the delays for BE frames are given in Table 5.4. It is easy to observe that the Timely Block policy has the greatest effect over delay times of this traffic class. This is due to the fact that it has the least priority and executes only when no TT and RC frames are present. Shuffling is the integration policy that BE frames benefit from the most. They are able to "shuffle" their priority with the two other traffic classes, preempt them and execute on the dataflow link. As previously stated, a delayed TT instance impacts the transmission of all traffic that comes after it. With preemption a BE instance can experience a

great delay since the other traffic classes can stop its transmission on a dataflow link and execute prior to its next attempt.

Section 3.4.1 discussed the advantages of the event-oriented paradigm over the activity-oriented one. The simulation of unnecessary events leads to useless computation load that results in greater running times. The test case from the first comparison was used here as well.
The results can be seen in Table 5.5. Column 1 gives the number of cycles given as command line parameter. Column 2 gives the total running time for the activity-oriented implementation of the simulator in **seconds**. Column 3 gives the corresponding values for the event-oriented simulator.

Simulator runs	**Activity-oriented** [s]	**Event-oriented** [s]
10	232	2
100	1887	18
1000	24189	180

Table 5.5: Comparison between the two simulators

The difference between the running times is two orders of a magnitude. The main reason is that in the event-oriented implementation the TT frame's sending and receiving times are initially inserted sorted into the queue. This saves a great amount of time that could otherwise be wasted on inserting and sorting the queue for each TT instance.
Another factor for the huge difference in execution times is the sorting itself. It is done locally. Even though the queue holding the events is sorted on several occasions (e.g. release RC and BE, transmit events, etc.), the sorting process includes only the events that come before the event causing the sorting and the event itself.

Test case	**ES**	**SW**	**Frames**	**Frame instances**
1	11	4	155	18824
2	13	3	110	14033
3	25	6	106	1540
4	25	6	118	1949
5	25	6	155	2703
6	25	6	249	2893
7	25	6	215	3259
8	35	8	85	2068
9	35	8	246	4160
10	35	8	212	3615

Table 5.6: Test cases used for in the Evaluation section

Next the report presents a comparison between 10, 100, 500, 800, 900 and 1000 simulation cycles executed by the action-oriented simulator. For this comparison ten different test cases (including the aforementioned one) were used. They are described in Table 5.6. In order to group all data that was gathered and represent it in a easy and understandable way, the following approach was adopted. First a percentile difference between the results from the different simulation categories and 1000 cycles was made. This means the percentile difference between 10 and 1000, 100 and 1000, 500 and 1000, etc. This process was repeated for every test case. The figures in Appendix A demonstrate a comparison between the resulting values. It is visible that the values for the WCD obtained from the 1000 cycle simulation are greater than those for all other simulations. Also the tendency that longer simulations generate greater delay times can be observed. All the test cases sooner or later converge to the worst-case end-to-end delays experienced in the 1000-cycle simulations. Table 5.7 gives the average running time in **seconds** for the 1000-cycle simulations.

Test case	**Average running time[s]**
1	221
2	166
3	24
4	24
5	28
6	32
7	32
8	29
9	51
10	47

Table 5.7: Average running times of 1000-cycle simulations

In order to get a deeper insight into whether and when the simulation reaches a "steady-state", further tests were conducted with one particular test case. Test case number **3** was used for this purpose mainly because of its running short running time. Percentile difference between the 1000, 1500, 2000, 2500, 3000, 3500 with respect to 4000 Simulation cycles were chosen to provide more information.

Figure5.5 depicts how the RC delays converge towards the values gained from 4000 simulation cycles, as the number of simulation cycles increases. Given that a "steady-state" of the system is defined as DELTA of 2%, the percentile difference observed after 2000 simulation cycles shows such behavior. Table 5.8 supports this observation. One may argue that a an even deeper understanding may be achieved if the granularity of the simulations cycle intervals is increased in an order of a magnitude. Because of the current running time of the imple-

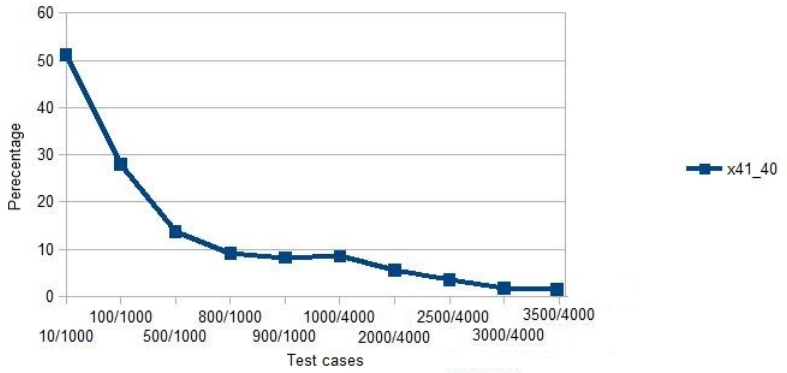


Figure 5.5: Percentile difference between the 1000, 1500, 2000, 2500, 3000, 3500 with respect to 4000 Simulation runs

Percentile difference	**Test case 3**
1000/4000	8.55
1000/4000	8.55
1500/4000	5.61
2000/4000	3.49
2500/4000	1.75
3000/4000	1.44
3500/4000	0.96

Table 5.8: Average percentile difference between 1000, 1500, 2000, 2500, 3000, 3500 and 4000 simulation cycles

mentation, the resources available and the results acquired, it is unfeasible to perform longer observations.

One of the outputs that the simulator generates is the average WCD for the BE frames. Table5.9 shows an average over all average delays acquired from all test cases for 1000 Simulation cycles.

Next the simulator was tested with a real world example - NASA's Orion Crew Exploration Vehicle. Potential Orion mission objectives include delivering a crew to the International Space Station, transporting a crew to a near-Earth objects, and providing emergency return capability from the International Space Station.
The following are some of the Orion's vehicle subsystems:

Test case	Average delay [s]
1	13169.05
2	1657.6
3	236.48
4	218.33
5	238.07
6	644.71
7	1126.51
8	300.16
9	349.6
10	539.11

Table 5.9: Average delays for BE frames

- Propulsion
- Vehicle power
- Life support
- Communications
- Docking adapter
- Structures
- Pyrotechnics

The Orion Avionics subsystem provides an infrastructure to command, control, and monitor all of these subsystems and more.

Orion uses an IMA-based high integrity architecture with the following elements:

- Vehicle Management Computers (VMCs) - provides a central computing platform to host software applications for a variety of vehicle subsystems
- TTEthernet Onboard Data Network - provides priority-based network communications via time triggered, rate constrained, and best effort traffic classes
- Power and Data Units (PDUs) - provides sensor data gathering, actuator control, and power distribution for critical vehicle subsystems

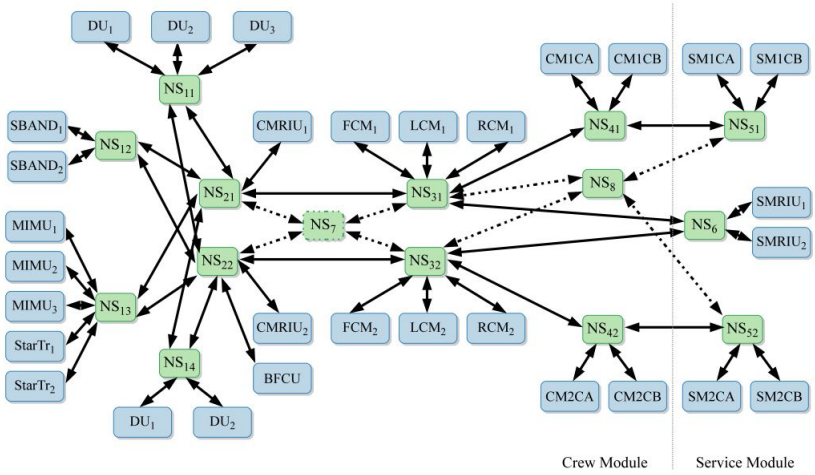


Figure 5.6: Orion topology

Figure 5.6 depicts the topology used for the two test cases in the simulation. The cases themselves are described in Table 5.10. **Orion 1** considers a scenario without the switches NS_7 and NS_8, and their respective dataflow links. **Orioin 2** enhances this topology by adding NS_7 with its supplementary connections.

Test case	ES	SW	Frames	Frame instances	Average run-time[s]	Total run-time[s]
Orion 1	31	13	180	5438	501	250572
Orion 2	31	14	180	5438	602	301008

Table 5.10: Orion test cases

The two test cases were run for 500 simulation cycles. The goal was to test the two network topologies and decide which one is better suited for timing constraints that the Orion poses. Figure 5.7 shows the results in the form of percentile difference between the two test cases for all RC frames. One can observe that the majority of RC frames of **Orion 2** experience a far lesser delay in comparison with those **Orion 1**. This leads to the conclusion that enhanced network of Orion 2 performs quite better in comparison to Orion 2. The concrete values for the RC WCD as well as the percentile difference between the two test cases are shown in Appendix B.

Lastly a comparison between the greatest WCD and the TTEthernet analysis is presented. [Ste11] presents different ways how to configure static slots (e.g. "blank intervals") for time-triggered messages and then use the free bandwidth

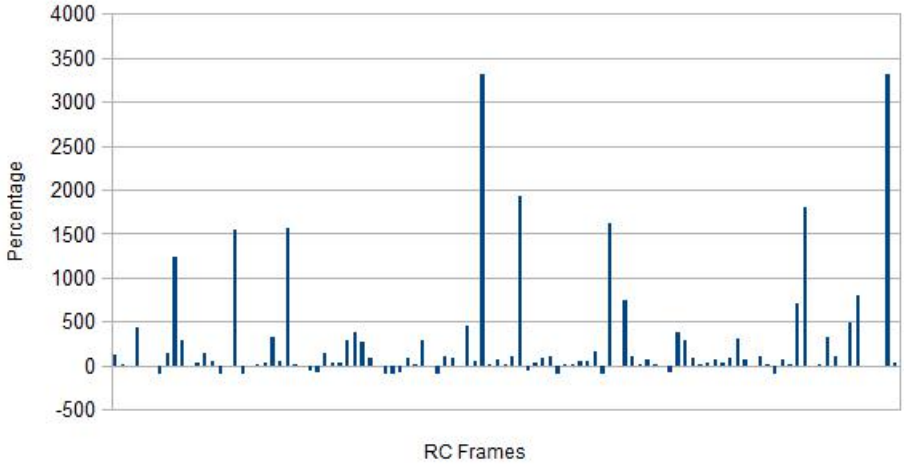


Figure 5.7: Percentage difference between the two Orion test cases

in between two time-triggered messages for RC and BE frames. The analysis presented in that paper and in [TSPS12] compares all the slots available on the dataflow link and chooses the smallest one. This slot then substitutes all others. This leads to a very pessimistic performance (i.e. great worst-case end-to-end delays) caused by the lack of execution time for the RC frames.

As already observed the greatest values are from the 1000 cycle simulation. Table 5.11 gives more info about the test case with regards to the number of their End Systems (column 2), Switches (column 3), total amount of frames (column 4) and frames instances (column 5) that were transmitted during the simulation. The last column uses the approach presented for the comparison between the different simulation cycles. The percentile difference in column 6 shows that the analysis has a far more pessimistic view that the actual results gained from the analysis.

Test case	$\triangle_{e-t-e}$ **delay [%]**
1	21388.23
2	22101.10
3	40357.05
4	66831.41
5	50209.40
6	109484.76
7	156453.61
8	24770.09
9	167413.91
10	116517.49

Table 5.11: Comparison between the WCD of the 1000 Simulation runs and the TTEthernet analysis

Chapter 6

Conclusion

Nowadays embedded systems are found everywhere - consumer electronics, telecommunications systems, medical equipment, etc. A subclass of these systems operates in an area with higher demand on time, efficiency and quality - hard-real time systems. When a system is further categorized as safety-critical, it needs to assure that multiple failure mechanisms execute to prohibit failure propagation.

The master thesis examined, modeled and simulated the TTEthernet - a network protocol made to accommodate the needs of hard real-time systems. Its design allows for integration on distributed systems where traffic classes of messages vary in safety integrity levels. The three classes are Time-Triggered (TT), Rate-Constrained (RC) and Best Effort (BE). To enable their transmission and utilization on a "mixed-criticality" system, two key components are needed - spatial and temporal separation. The former is done, depending on the traffic class, either through offline scheduling of TT messages or through allocation of bandwidth. The later - through virtual links, which are logical point-to-point "tree-like" structures. TTEthernet can extend an existing network to different topologies located on heterogeneous media without introducing major changes to its state. This is done with the addition of End Systems and Network Switches. It comprises various capabilities like redundancy, scalability and multiple fault containment.

Many of the systems that operate today are extremely large and expensive.

TTEthernet itself is a vast and complex system that has multiple features which require thorough examination before being put to use. In order to simulate such environments, one needs to have a fundamental understanding of the main concepts within a simulation. When the characteristics of a system have been defined, the continuous process verification, validation and calibration commences. An precise mathematical model provides for a solid foundation needed to create a simulation. Some of the key points in building a simulator comprise selecting input probability distributions and random number generators as well as performing proper output data analysis.

Meeting the requirements for the master thesis project proved to be a major task. The level of abstraction of data that the project required, called for precise and efficient design decisions that would facilitate all the key concepts of the protocol. When done in this manner the coding process, while long and tiring, proved to be rewarding. The result is the creation of two simulators - one implementing the action-oriented and another the event-oriented paradigm. These tools take as input various network topologies, different message files, allow for to access multiple simulation characteristics and calculate results with supplementary visualizations.

The tests performed on the tool give proof of its correctness on various levels - programming logic and model verification. The output obtained from the simulators was compared in various ways that prove the existing theory and also show a trend for the ratio between simulation runs and worst-case ent-to-end delay. The comparison between the analysis and the simulation gave a more thorough understanding of the relation between the two.

6.1 Future work

The TTEthernet network protocol is constantly being improved. Its application is proven to be of great value, not only to the aircraft industry, but for broader use as well. An area such as optimization of mixed-criticality applications on distributed systems such as backbone infrastructures of vehicles is just a single branch among the many. For example, [OFF13] includes heterogeneous, distributed control for managing unpredictable behavior in distributed and networked systems.

The list of future work suggestions, made on the basis of the work done for the master thesis project as well as the knowledge gained from articles, books and publications, contains:

- packet losses and/or line failures. It would be interesting to model a more dynamic simulator that considers scenarios where the packets experience errors on the dataflow links.
- multiple clusters. A topic that has a lot to offer since communication between different clusters requires clock synchronization mechanisms and message transformation techniques. For example, because the notion of time is different for each cluster, transmitting a TT message from one to another requires that the message undergoes multiple transformations (to RC and then back to TT) before it reaches its destination.
- implementation of heterogeneous networks - a combination between a TTEthernet network and an AFDX network would definitely be interesting to model and observe
- corner-cased simulations - implement features that allow for faster simulation based on previous simulation runs
- model a process-oriented simulator

Appendix A

Appendix A

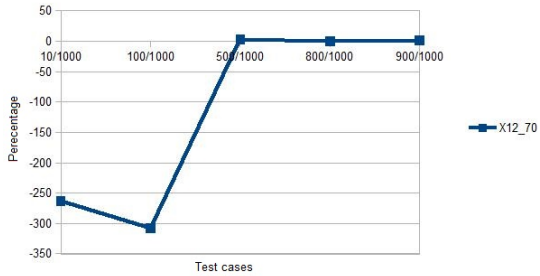


Figure A.1: Results for test case 1

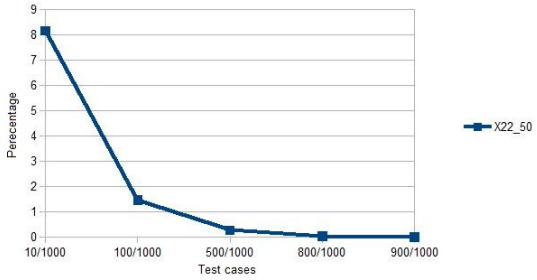


Figure A.2: Results for test case 2

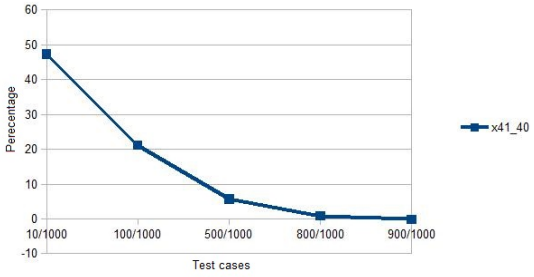


Figure A.3: Results for test case 3

Test case	10/1000 [%]	100/1000[%]	500/1000[%]	800/1000[%]	900/1000[%]
1	-262.94	-307.78	2.77	0.12	1.25
2	8.17	1.46	0.27	0.04	0.01
3	47.28	21.11	5.75	0.84	-0.08
4	57.95	25.26	10.28	2.80	0.18
5	55.58	23.66	6.19	2.00	0.27
6	51.17	20.05	4.45	1.72	0.68
7	49.27	19.29	1.00	0.71	0.36
8	51.35	18.53	3.43	0.33	-0.32
9	53.22	28.54	8.28	2.14	0.56
10	56.56	28.53	8.91	2.67	0.97

Table A.1: Average percentile difference between 10, 100, 500, 800, 900 and 1000 simulation cycles

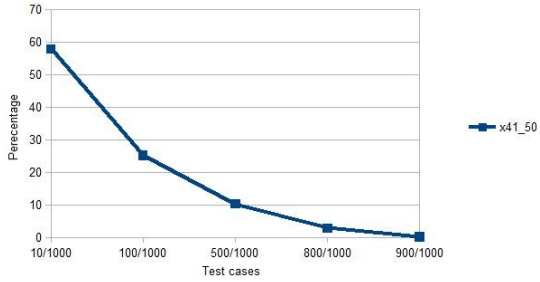


Figure A.4: Results for test case 4

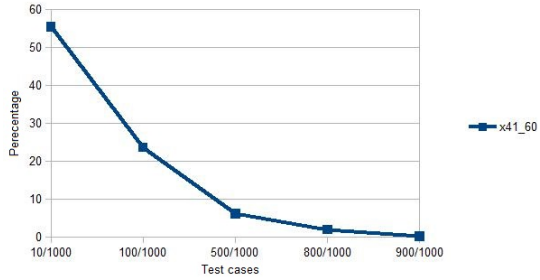


Figure A.5: Results for test case 5

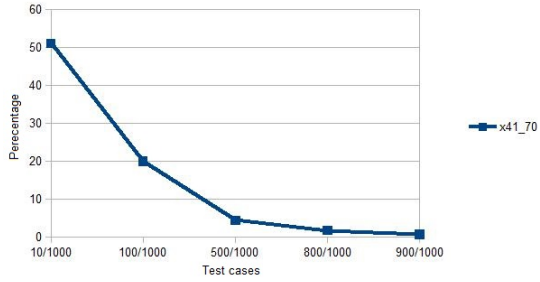


Figure A.6: Results for test case 6

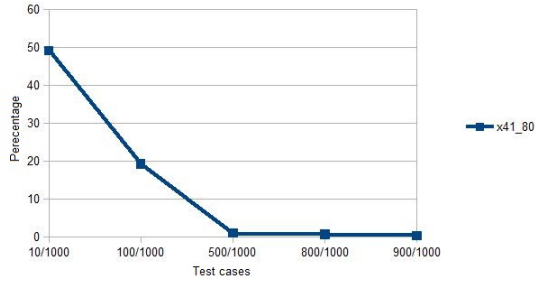


Figure A.7: Results for test case 7

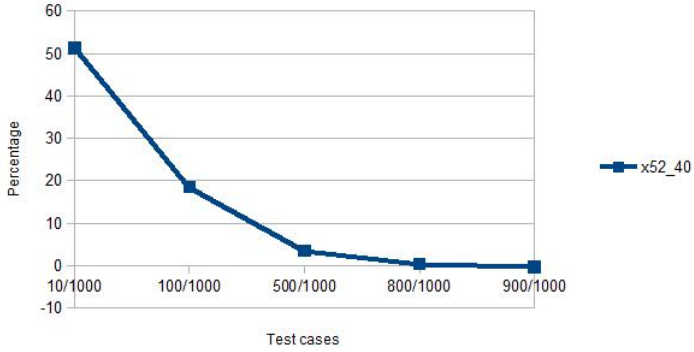


Figure A.8: Results for test case 8

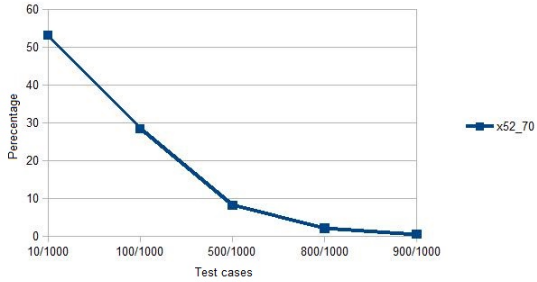


Figure A.9: Results for test case 9

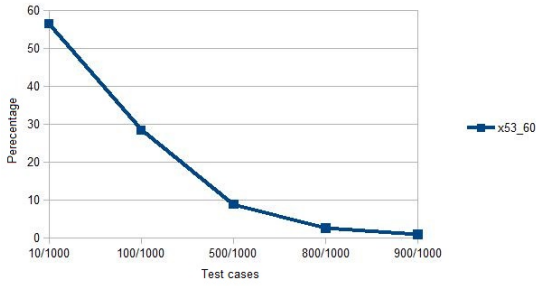


Figure A.10: Results for test case 10

Appendix B

Appendix B

Frame	Orion1	Orion2	Orion1/Orion2[%]
rc1.0	251394	113953	120.61
rc10.0	12310	11208	9.83
rc13.0	121321	23123	424.67
rc14.0	62201	63051	-1.34
rc15.0	31205	31058	0.47
rc16.0	7767	251435	-96.91
rc17.0	18013	7371	144.37
rc182	558924	42012	1230.39
rc19.0	31312	8003	291.25
rc190	511245	527356	-3.05
rc192	352680	252680	39.57
rc195	55123	23131	138.30
rc2.0	15737	10534	49.39
rc20.0	7873	250923	-96.86
rc207	252871	250538	0.93
rc208	254521	15461	1546.21
rc21.0	7432	251196	-97.04
rc210	253635	250565	1.22
rc218	573153	521471	9.91
rc219	12313	9012	36.62
rc22.0	258145	61234	321.57
rc221	253654	163922	54.74
rc225	254716	15241	1571.25
rc227	3490	3016	15.71
rc228	250739	251536	-0.31
rc23.0	123500	250188	-50.63
rc235	61457	252320	-75.64
rc25.0	291349	123165	136.55
rc27.0	319294	251152	27.13
rc28.0	26192	19248	36.07
rc29.0	29592	7469	296.19
rc3.0	251261	51929	383.85
rc30.0	12949	3566	263.12
rc31.0	7517	4124	82.27
rc32.0	254648	252758	0.74
rc33.0	30586	257833	-88.13
rc34.0	3396	257433	-98.68
rc35.0	122389	524268	-76.65
rc36.0	253490	131421	92.88
rc37.0	311512	257966	20.75
rc38.0	131122	33145	295.60
rc39.0	558314	557440	0.15

Table B.1: Percentile difference between Orion 1 and Orion 2 in 500 simulation cycles

Frame	Orion1	Orion2	Orion1/Orion2[%]
rc4.0	7782	251961	-96.91
rc40.0	15336	7380	107.80
rc41.0	561994	300185	87.21
rc42.0	31740	31411	1.04
rc425	123951	22219	457.86
rc43.0	62165	41299	50.52
rc44.0	253184	7410	3316.78
rc45.0	16054	13919	15.33
rc46.0	251149	151299	65.99
rc47.0	290441	250884	15.76
rc48.0	61779	30808	100.52
rc49.0	155519	7693	1921.565
rc5.0	7377	15846	-53.44
rc50.0	41294	30891	33.67
rc51.0	252213	131273	92.12
rc53.0	254005	120167	111.37
rc54.0	7940	251406	-96.84
rc55.0	254826	213149	19.55
rc56.0	313479	256254	22.33
rc57.0	123919	87893	40.98
rc58.0	12059	7783	54.94
rc59.0	39691	15249	160.28
rc6.0	15879	561701	-97.17
rc60.0	525000	30453	1623.96
rc61.0	61867	61957	-0.14
rc62.0	62582	7410	744.56
rc63.0	31274	15660	99.70
rc64.0	273162	252000	8.39
rc65.0	528448	315455	67.51
rc66.0	15375	13521	13.71
rc663	256400	250241	2.46
rc665	62121	253130	-75.45
rc67.0	39139	8106	382.83
rc671	30755	7994	284.72
rc673	252241	141939	77.71
rc675	3451	3199	7.87
rc678	131491	99813	31.73
rc68.0	8051	4913	63.87
rc680	525978	391929	34.20
rc686	39193	21945	78.59

Table B.2: Percentile difference between Orion 1 and Orion 2 in 500 simulation cycles

Frame	Orion1	Orion2	Orion1/Orion2[%]
rc687	251506	61835	306.73
rc69.0	31207	19392	60.92
rc692	525774	527103	-0.25
rc695	132939	62661	112.15
rc697	251683	239194	5.22
rc699	7402	255397	-97.10
rc70.0	31227	19319	61.63
rc701	301991	252004	19.83
rc702	256899	31587	713.30
rc706	252301	13238	1805.88
rc709	253351	267746	-5.37
rc71.0	7500	7013	6.94
rc712	562339	131380	328.02
rc72.0	31573	15678	101.38
rc73.0	253021	250371	1.05
rc75.0	142348	23882	496.04
rc76.0	560837	62485	797.55
rc77.0	253909	253421	0.19
rc78.0	62563	62770	-0.32
rc79.0	3936	4104	-4.09
rc8.0	251531	7356	3319.39
rc9.0	31267	23165	34.97

Table B.3: Percentile difference between Orion 1 and Orion 2 in 500 simulation cycles

Bibliography

[afd09] *ARINC 664P7: Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network.* ARINC (Aeronautical Radio, Inc), 2009.

[ASBCH13] Ahmad Al Sheikh, Olivier Brun, Maxime Chéramy, and Pierre-Emmanuel Hladik. Optimal design of virtual links in AFDX networks. *Real-Time Syst.*, 49(3):308–336, 2013.

[BBB+09] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russel Urzi. A research agenda for mixed-criticality systems. In *Cyber-Physical Systems Week*, 2009.

[BCNN00] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation (3rd Edition).* Prentice Hall, 3 edition, 2000.

[Eng05] Condor Engineering. Afdx protocol tutorial. 2005.

[HD93] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace Electronic Systems Magazine*, 8:34–39, 1993.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. In *PROCEEDINGS OF THE IEEE*, pages 112–126, 2003.

[Kop11] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer, 2011.

[LK99] Averill M. Law and David M. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill Higher Education, 3rd edition, 1999.

[Mat08] Norm Matloff. Introduction to discrete-event simulation and the simpy language. 2008.

[MGPM04] Paul Miner, Alfons Geser, Lee Pike, and Jeffery Maddalon. A unified fault-tolerance protocol. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, volume 3253 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2004. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unified.html`.

[OFF13] ARTEMIS OFFICE. Artemis programme. 2013.

[Pla05] Ge Intelligent Platforms. Afdx/arinc664 tutorial. 2005.

[Pla09a] Ge Intelligent Platforms. Ttethernet - a powerful network solution for advanced integrated systems. 2009.

[Pla09b] Ge Intelligent Platforms. Ttethernet - a powerful network solution for all purposes. 2009.

[PSG+11] Michael Paulitsch, E Schmidt, B Gstöttenbauer, C Scherrer, and H Kantz. Time-triggered communication (industrial applications). In *Time-Triggered Communication*, pages 121–152. CRC Press, 2011.

[Rus99] John Rushby. Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, 1999.

[Rus01] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. Available at `http://www.csl.sri.com/~rushby/abstracts/buscompare`.

[SBH+09] Wilfried Steiner, Günther Bauer, Brendan Hall, Michael Paulitsch, and Srivatsan Varadarajan. Ttethernet dataflow concept. In *Proceedings of The Eighth IEEE International Symposium on Networking Computing and Applications, NCA 2009, July 9-11, 2009, Cambridge, Massachusetts, USA*, pages 319–322. IEEE Computer Society, 2009.

[SJ08] R. Shaw and B. Jackman. An introduction to flexray as an industrial network. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on Industrial Electronics*, 2008.

[SKKS11] Till Steinbach, Hermand Dieumo Kenfack, Franz Korf, and Thomas C. Schmidt. An extension of the omnet++ inet framework for simulating real-time ethernet with high accuracy. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 375–382, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Ste11] Wilfried Steiner. Synthesis of static communication schedules for mixed-criticality systems. *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, 0:11–18, 2011.

[Sue12] E. Suethanuwong. Scheduling time-triggered traffic in TTEthernet systems. In *Emerging Technologies Factory Automation*, pages 1–4, 2012.

[TSPS12] Domitian Tamas-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of communication schedules for ttethernet-based mixed-criticality systems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '12, pages 473–482, New York, NY, USA, 2012. ACM.