UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Evaluation of On-line Reconfiguration Techniques for a Distributed Avionic Middleware

Glen te Hofsté

M.Sc. Thesis

05-07-2023

Supervisors:
Dr. ir. M. Ottavi
Dr. ir. A. Lund
Committee:
Prof. dr. ir. A.L. Varbanescu
Dr. ir. A. Chiumento
Dr. ir. A. Menicucci

Computer Architecture for Embedded Systems (CAES)
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Preface

This master thesis project is executed by Embedded Systems student Glen te Hofsté, studying at the University of Twente in the Netherlands. The master thesis project is executed at the Deutsche Zentrum für Luft- und Raumfahrt e.V. (DLR) Oberpfaffenhofen in Germany.

The DLR is Germany's research centre for aeronautics and space. At the DLR, Germany's national space program is planned and implemented on behalf of the federal government. This master thesis project falls under the DLR's Institute for Software Technology and is part of the ScOSA project [1]. Current activities of the Institute for Software Technology focus on software for distributed systems and intelligent systems, artificial intelligence, software technologies for embedded systems, visualization, data science, and high-performance computing [2].

Acknowledgements

Many thanks to Dr. Ir. M. Ottavi from the University of Twente for being willing to supervise this thesis remotely. I am also grateful to the graduation committee and Dr. Ir. S.H. Gerez for putting in the time and effort to support me and providing me with feedback in the final phase of the thesis. The graduation committee consists of the following members:

- Dr. ir. M. Ottavi (University of Twente and the University of Rome Tor Vergata)
- Prof. dr. ir. A.L. Varbanescu (University of Twente)
- Dr. ir. A. Chiumento (University of Twente)
- Dr. ir. A. Menicucci (Delft University of Technology)

It has been a pleasure working at the Institute of Software Technology of the DLR. They helped me through their feedback and let me enjoy my period abroad in Germany. Finally, this journey would not have been possible without the help of Dr Ir. A. Lund from the DLR. His effort and support have been a vital contribution towards the completion of this project.

Abstract

On-board Computers are at the centre of space-faring systems. They provide computational performance to the system, with high availability and dependability. However, these systems commonly consist of expensive, slow, fault-tolerant hardware to overcome errors or failures during a mission. Commercial Of The Shelf (COTS) components provide higher performance but do not provide these fault-tolerance mechanisms. To tackle this, the DLR utilises a distributed system of COTS components, called nodes, that run a middleware called ScOSA. ScOSA manages the nodes in the distributed system and, upon a node failure, mitigates the effects by reconfiguring the system to a configuration that excludes the failed node. During a reconfiguration, the tasks on the failed node get scheduled to another node in the system, depending on a pre-determined configuration. These configurations are calculated *offline* and have an exponentially growing memory usage depending on the number of nodes in the system, limiting the system's scalability. An *online* algorithm is seen as a solution to this scalability problem, as it does not need pre-determined configurations and can use the real-time state of the system to make scheduling decisions instead.

Therefore, in this project, an online algorithm was implemented in the middleware of ScOSA, designed as a combination of the online algorithms acquired during literature research. After defining specific requirements for the tailor-made online algorithm, it was tested on the target hardware and by using virtual nodes for its feasibility and scalability. It was compared with the offline situation on *time* and *network usage*. The design of an online algorithm for ScOSA proved to be feasible. It was not only capable of creating configuration dynamically but also proved to be a solution to the scalability problem. Time and network usage increases were observed for the online algorithm, although none of these violated any constraints while running the test program. Furthermore, with its fault-tolerance being an integral part of the online algorithm, it is well-suited for its target application in space.

Contents

1 Introduction **9**
1.1 Background 10
1.2 Problem definition 11
1.2.1 Fault model 11
1.2.2 Fault detection 11
1.2.3 Fault handling 12
1.2.4 Questions and objectives 13
1.3 Project outline 13

2 System Overview **14**
2.1 Software stack 15
2.1.1 SpaceWire-IPC 15
2.1.2 Distributed Tasking Framework 16
2.1.3 System Management Services 16
2.2 ScOSA system model 16
2.2.1 Configurations 18
2.2.2 Existing algorithm requirements 19

3 Design methodology **21**
3.1 Additional online algorithm requirements 21
3.2 Literature research 21
3.2.1 Search string 22
3.2.2 Filtering 22

4 Design **24**
4.1 Defining additional requirements 24
4.2 Literature research findings 25
4.3 Literature Review 27
4.3.1 Literature research conclusion 28
4.4 Design approach 29
4.4.1 Simulation approach 29
4.4.2 Functional approach 29
4.4.3 Preferred approach 30
4.5 Algorithm design 30
4.5.1 Algorithm requirements 31
4.5.2 Combining algorithms 31
4.5.3 Implementation 38

5 Evaluation **39**
5.1 Evaluation methodology . . . 39
5.2 Evaluation approach . . . 39

6 Results **45**
6.1 Timing analysis . . . 45
6.1.1 Reconfiguration time . . . 45
6.1.2 Decision-making time . . . 47
6.2 Network traffic analysis . . . 50
6.2.1 Three nodes . . . 50
6.2.2 Scaling . . . 51

7 Discussion **53**
7.1 Interpretations . . . 53
7.1.1 Literature research . . . 53
7.1.2 Design . . . 53
7.1.3 Time . . . 54
7.1.4 Network traffic . . . 56
7.2 Implications . . . 58
7.3 Limitations . . . 58
7.3.1 Design . . . 58
7.3.2 Test program . . . 60
7.3.3 Test setup . . . 60
7.3.4 Network delays . . . 60
7.3.5 Exploratory statistics . . . 60
7.4 Recommendations . . . 61
7.4.1 Design improvements . . . 61
7.4.2 Test additional parameters . . . 61

8 Conclusion **63**

References **64**

Appendices

A Software diagrams **68**
A.1 Scheduling phase flow chart . . . 69
A.2 Node Failure handling sequence diagram . . . 70
A.3 Scheduling procedure sequence diagram . . . 71

B Scripts **72**
B.1 HPN testing script . . . 72

C Traces **73**
C.1 Coordinator failure trace . . . 73
C.2 Node failure cache load trace . . . 73

List of acronyms

API	Application Programming Interface
COTS	Commercial Of The Shelf
DAG	Directed Acyclic Graphs
DLR	Deutsche Zentrum für Luft- und Raumfahrt e.V.
FDIR	Fault Detection, Isolation and Recovery
FIFO	First In – First Out
HEFT	Heterogeneous Earliest Finish Time
HPN	High Performance Node
I/F	Interface
IC	Integrated Circuit
IPC	Inter Process Communication
KDE	Kernel Density Estimation
LIFO	Last In – First Out
MBU	Multiple Bit Upset
OBC	On-Board Computer
OBC-NG	On-Board Computer - Next Generation
RCN	Reliable Computing Node
ScOSA	Scalable On-board computing for Space Avionics
SEU	Single Event Upset
SMT	Satisfiability Modulo Theory
SoC	System on Chip
TM/TC	Telemetry and Telecommand
TMR	Triple Modular Redundancy
WCET	Worst Case Execution Time

List of Figures

1.1 ScOSA offline configurations 12

2.1 ScOSA System Architecture [3] 14
2.2 ScOSA Software Stack [3] 15

4.1 Reconfiguration Manager inheritance class diagram 38

5.1 Test program configuration 40
5.2 Test program 1 decision graph 41
5.3 Target hardware network containing three HPNs and an RCN 42
5.4 Example of a simulated node failure using online scheduling 44

6.1 Reconfiguration time online vs offline density plot 46
6.2 Reconfiguration time online histogram and density plot 46
6.3 Decision making time cached vs non-cached density plot 47
6.4 Decision making time (ms) cache load vs no cache load density . . . 48
6.5 Decision making time (ms) cache load vs no cache load box plot . . . 49
6.6 Three nodes offline vs online network traffic density plot 50
6.7 Scaling online algorithm with caching enabled vs disabled plot 51

List of Tables

6.1 Reconfiguration time (ms) 45
6.2 Decision making time (ms) Cache enabled vs disabled 47
6.3 Decision making time (ms) cache load vs no cache load 48
6.4 Decision making time (ms) per node 49
6.5 Three nodes offline vs online network traffic (bytes) 50

Chapter 1

Introduction

The paradigm of space systems is very diverse, ranging from international space stations to a wide variety of rockets to an even wider variety of satellites. Ever since the first artificial satellite to orbit the Earth, called Sputnik 1 satellite (1957), the field of space-faring systems has expanded drastically. These space-faring systems are often incredibly complex and designed to survive the challenging environment in space. The radiation and electromagnetism experienced when being in orbit while needing to be lightweight, robust and cost-effective to get there in the first place are just some of the examples of challenges that make this field so demanding. At the centre of these space-faring systems, the On-Board Computer (OBC) can be found.

The OBC provides the space-faring system with its computational performance, often along with features such as trajectory control, data acquisition and processing, and communication. As space systems became more advanced, the requirements for the OBC did so as well. Missions themselves became more complex in an attempt to get the most value out of expensive launches into orbit. To facilitate this, OBCs became more complex, with higher computational performance due to improved Integrated Circuit (IC) design and lower power consumption [4].

With the downsizing of the hardware in OBCs, however, new problems arose. The higher density of processors meant that they become more prone to environmental effects, such as radiation, which can cause *errors* or even *failures* due to, for example, Single Event Upset (SEU)s and Multiple Bit Upset (MBU)s. These errors reduce the reliability and thus dependability of the OBC.

To counteract these problems, reliable space-qualified hardware is often used. This can be through dedicated hardware, such as the proven RAD750 by BAE Systems [5] or synthesizable VHDL models for a System on Chip (SoC) such as the LEON processor by Cobham Gaisler [6]. These solutions have built-in low-level fault-tolerance mechanisms but have the downside that their performance lacks compared to modern COTS hardware. This led the DLR to investigate alternative techniques to find a cost-effective solution that is reliable and high-performance for their next generation of space systems.

1.1 Background

The DLR has a large number of active projects where OBCs are used. These OBCs tended to be very expensive due to the specialized radiation-hardened hardware and their application-specific design. With COTS hardware being available with much higher computational performance at a significantly lower cost, the question arose whether the radiation-hardened hardware was still the best solution for the next generation of OBCs. To investigate this, the DLR started the On-Board Computer - Next Generation (OBC-NG) project [7] to search for other means of making a dependable system with a higher computational performance at a lower price. It is designed as a scalable, *distributed* parallel computing system to prevent single points of failure by distributing its computational resources. The distributed system consists of a network of nodes on which tasks can be executed. This also enabled alternative error mitigation techniques on the software level instead of hardware, therefore omitting the need for specialized hardware and allowing for a more cost-effective solution.

The distributed system proved promising and continued as the Scalable On-board computing for Space Avionics (ScOSA) project [1]. With the ScOSA project, the work from the OBC-NG project continued and expanded, with the addition of heterogeneity and several additional features to aid application developers [3]. A variety of different nodes are allowed, as ScOSA is abstracted to a monolithic execution platform by implementing it as a *middleware*. The custom-built middleware, written in C++, allows applications to be executed using the computing resources of all nodes simultaneously while masking the underlying complexity of the distributed system. As ScOSA can be run on a variety of platforms, different types of hardware can be mixed, introducing heterogeneity into the system. This allows the usage of high-performance COTS hardware to be used alongside space-grade hardware, getting the best of both worlds. Using COTS components in space is not new and has been shown to have its value [8]. High-performance hardware such as the Xilinx Zynq-7000 [9] provide computational power that is very desirable for OBCs, and using ScOSA, a distributed system of these High Performance Node (HPN)s, running *Linux*, can be assembled. As the HPNs are not space-grade, they are expected to fail at some point. To mitigate the failure of a node, ScOSA employs services which implement Fault Detection, Isolation and Recovery (FDIR), such as Triple Modular Redundancy (TMR), check-pointing and restoration services, to name a few.

Additionally, to increase the availability of the system even more, the distributed system also supports Reliable Computing Node (RCN)s, running *RTEMS*. These RCNs are mostly used to control the distributed system and as a backup, to ensure that even if all HPNs would fail simultaneously, there is still a reliable fallback mechanism to restore the system. In case of a failing node, the tasks which were running on that node need to be moved to another healthy node according to a predefined *configuration*. This configuration has been designed and verified ”offline”, using model-based reconfiguration planning [10]. A configuration defines exactly how tasks are mapped to nodes at any point in time in case of an event.

The mapping problem, however, is NP-Complete, making it only feasible for small systems. As soon as the system starts to scale, the exact solving becomes unfeasible, and the memory footprint of the configuration becomes too large for the hardware it intends to run on. Furthermore, the configuration contains situations that might never occur, occupying memory resources that could have been used more effectively. This led to the desire to reconfigure ”online” instead. An online reconfiguration is, compared to an offline one, not predefined. Instead, an online reconfiguration uses an algorithm to determine at run-time how the nodes have to be (re)mapped when a node fails. The node to which a task is to be moved, or *scheduled*, is thus decided during run-time, not requiring a large set of memory-intensive configurations. This project specifically focuses on the development of this online (reconfiguration) algorithm for ScOSA to solve the shortcomings of the current offline approach.

1.2 Problem definition

Besides the advantages of an online algorithm, there will always be problems that make implementation challenging. However, there is a common fault model that both the online and offline reconfiguration approaches have to deal with.

1.2.1 Fault model

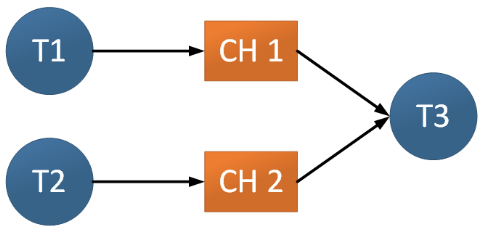
As explained, the hardware on which ScOSA is running is susceptible to errors and failures when operated in space, which is mainly caused by radiation. When an ionizing particle strikes operational microelectronics, it can cause the sporadic ”flipping” of bits in memory or elsewhere on a chip, depending on the technology. This can result in data corruption, causing systems to fail and greatly decreasing the availability of the system. ScOSA has means of detecting and correcting errors due to sporadic bit flips but cannot do much when the entire node fails to operate. This is called a *node failure*, characterised by a node becoming unresponsive, requiring it to be rebooted. With ScOSA’s focus on COTS components, a way had to be found to cope with these failures. The idea of ScOSA is not to attempt to get rid of errors and failures completely but to allow them to occur and deal with the consequences. In the case of ScOSA, this means that the COTS nodes (HPNs) are expected and allowed to fail at any point in time. As the network consists of several nodes, the tasks from the failing node can be moved to another node instead, allowing the overall network to remain operational.

1.2.2 Fault detection

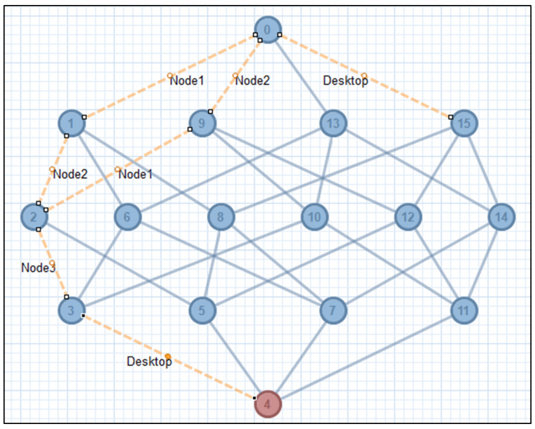
To move a failing node’s tasks to another node, one first needs to be able to detect that a failure occurred. To do this, ScOSA has several detection and mitigation techniques as part of its system management services and through a software layer on top of SpaceWire, enabling Inter Process Communication (IPC) and the communication mechanism between the nodes. This library, called SpaceWireIPC, is the backbone of ScOSA's reconfiguration mechanism [11]. SpaceWireIPC is the first point to detect a failed node through missed message acknowledgements. If a node continuously fails to respond to messages, it is considered to have failed, after which a reconfiguration is triggered. The system management services from ScOSA provide additional detection mechanisms, such as a Monitoring Service, which monitors nodes for their periodic heartbeat, and a Voter Service, which implements TMR, to detect and correct errors.

1.2.3 Fault handling

Once a fault is detected, a reconfiguration is triggered. Based on the configuration, a new path will be created for data to flow through another healthy node. These data flows are going through a *channel*. These channels make the connections between tasks more flexible, as they can be changed during runtime. An example of a network graph where tasks are interconnected using channels can be seen in figure 1.1a, where two tasks (T1 & T2) communicate through two channels (CH1 & CH2) with a receiving task (T3). When one of the nodes on which task T1, T2 or T3 is running fails, the tasks will be moved to another node, determined by the next configuration it reconfigures to. Which configuration is next is determined by a decision graph, of which an example can be seen in figure 1.1b. In this graph, each vertex resembles a configuration, and each edge a transition to the next configuration due to a failed node. If, for example, the system is in *configuration 0* and *Node 2* fails, the system reconfigures itself to *configuration 9*.



(a) Task graph example, containing three nodes and two channels



(b) Reconfiguration decision graph example, containing four nodes and sixteen configurations

Figure 1.1: ScOSA offline configurations

The decision graph of figure 1.1b is what an offline reconfiguration looks like. The decision graph is created by a tool that can generate and test the configurations for ScOSA using an offline algorithm [10]. Problems arise when the system scales by adding more nodes, and the number of configurations starts to grow exponentially. The exponential growth of the number of configurations has the same effect on the program's memory usage. In fact, memory usage becomes so large that it cannot be facilitated by the system's hardware, hindering the ability of the system to scale.

It is hypothesised that an online reconfiguration algorithm is a solution to this problem as it provides the opportunity to eliminate the need for pre-determined configurations. Instead of pre-determined configurations, an *online algorithm* can make task scheduling decisions based on the real-time state of the system, which has the additional benefit that it can take run-time effects into account. It is unclear whether or not implementing an online algorithm is even feasible in ScOSA, not to mention how its performance would be. An online algorithm is considered feasible if it is able to replace the functionality of the offline reconfiguration algorithm by dynamically creating configurations through online scheduling decisions without the need for any pre-determined configuration.

1.2.4 Questions and objectives

The objective of this thesis is to *evaluate* whether or not an online algorithm is *feasible* in ScOSA. Besides feasibility, it should also be determined how well it performs by implementing it into ScOSA, while determining whether or not it solves the offline algorithm's issue of scalability.

This lead to the following main research question:

- *Is an online algorithm feasible in ScOSA?*

If an online algorithm is indeed feasible, then add the following sub-questions:

- *Does an online algorithm solve the scalability issue of the offline algorithm?*
- *How well does it perform compared to the offline algorithm?*

1.3 Project outline

In the *System Overview* chapter 2, the existing work of ScOSA is discussed through an in-depth view of its architecture, software stack and, most important, its system model. This is followed by the *Design methodology* chapter 3, where the methodology for designing the online algorithm is discussed. Chapter 4 presents the *Design* of the online algorithm as a combination of what was found during the literature research and review, followed by the approach to evaluate the design in the *Evaluation* chapter 5. The results of the test setup are then present in the *Results* chapter 6, as acquired by running tests virtually and on the target hardware. The results are then discussed in the *Discussion* chapter 7. Here the results are interpreted while also discussing their limitations and openings for future work. Finally, chapter 8 will provide a *Conclusion*, reflecting on how the project was executed while providing recommendations for the future.

Chapter 2

System Overview

The ScOSA project was started with the goal of designing a highly reliable onboard computer for space avionics [12]. As part of ScOSA, middleware has been developed to provide an abstract layer to the heterogeneous distributed system it runs on. An example of a system architecture with two RCNs, an *N* amount of HPNs and how they are interconnected using SpaceWire or Ethernet can be seen in Figure 2.1. The RCNs are connected to the HPNs and Telemetry and Telecommand (TM/TC) units via SpaceWire routers. The Interface (I/F) nodes connect the system to the sensors and actuators [3].

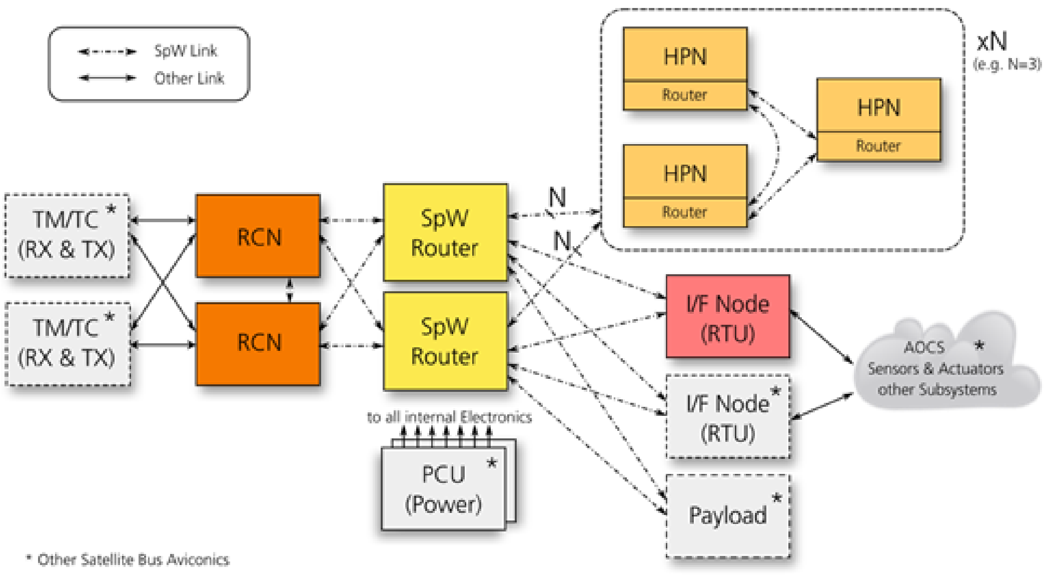


Figure 2.1: ScOSA System Architecture [3]

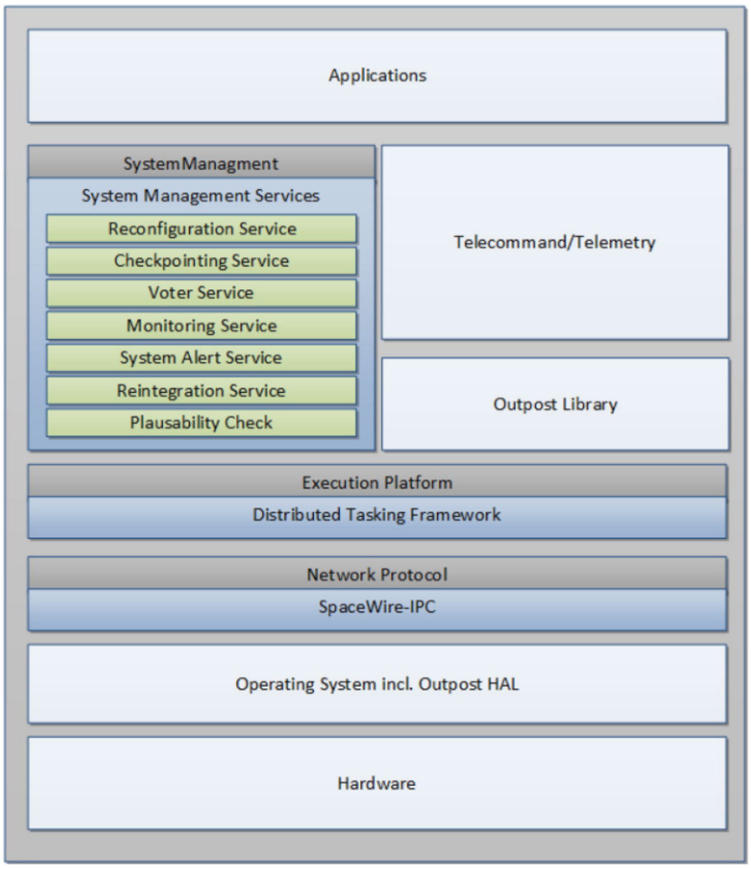


Figure 2.2: ScOSA Software Stack [3]

2.1 Software stack

The ScOSA software stack [3] is built as a layered architecture upon three main components:

- SpaceWire-IPC
- Distributed Tasking Framework
- System Management Services

The OUTPOST library [13] and the operating system[1] provide the abstraction to the hardware. Figure 2.2 shows a layered overview of the software stack.

2.1.1 SpaceWire-IPC

SpaceWire-IPC can be described as the transport layer (Layer 4 in the OSI model) of ScOSA [3]. It extends the standard SpaceWire protocol with features such as a reliable communication link, error detection and handling and several node management features. Besides SpaceWire, the SpaceWire-IPC has been ported to support Ethernet as well.

[1]ScOSA is mainly designed to run on Yocto Linux and RTEMS

2.1.2 Distributed Tasking Framework

The Tasking Framework is an open-source project by the DLR that implements applications as a graph of *tasks* and *channels* [14]. It is designed as an event-driven multi-threading execution platform for space applications. It is written in C++ and compatible with several (real-time) operating systems, such as Linux, RTEMS or FreeRTOS. It consists of an *execution platform*, which schedules ready task instances to one of the available executors according to a scheduling policy through an Application Programming Interface (API). The currently supported scheduling policies are First In – First Out (FIFO), Last In – First Out (LIFO) and priority-based. Currently, only the FIFO policy is used by ScOSA.

Each task of the Tasking Framework can have multiple inputs and outputs to which a task can be connected via a channel. How tasks and channels are interconnected can be defined manually or via a model-driven tool environment which automatically generates the required function calls to the Tasking Framework API. ScOSA implements the Tasking Framework and extends it to work on its distributed system architecture.

2.1.3 System Management Services

The System Management Services are used to implement FDIR in ScOSA. The FDIR services are performed in the background and masked from the application developer. Node failures are detected using the *Monitoring Service*, which monitors the state of nodes through a heartbeat mechanism. Nodes have different roles in this mechanism: either a *coordinator, observer* or *worker*. At any time, there is only one coordinator and a number of observer nodes. The observer nodes keep an eye on the coordinator. How many observers there are in the system can be determined by the system designer. Worker nodes do not monitor other nodes but only execute tasks as their duty. The *Voting Service* is used to implement TMR, which can be used to detect and correct soft errors. By processing an operation on three different nodes and afterwards comparing the results, a final result can be voted for. The *Reintegration Service* is used when a node starts up to initialise the system or reintegrate into an existing one. The *Reconfiguration Service* (re)configures the mapping between tasks and channels on a node based on a pre-determined configuration. The *Reconfiguration Manager* in the coordinator node initiates the reconfiguration if a node failure is detected or a node reintegrates into the system.

2.2 ScOSA system model

ScOSA is modelled as a distributed system, structured as a collection of communicating *periodic* tasks running on a network with an arbitrary topology. [10, 15]. The DLR implemented the system design for a mission as a two-stage process, where first, a task-node mapping is generated on which model checking is performed. The mapping is then used for system verification in the second stage.

For the first stage, the offline task-node mapping problem is approached as a *combinatorial optimization problem*, where the optimal solution is found from the *bounded* task graph. The task-node mapping, referred to as a **task graph**, is modeled as a directed graph TG:

$$TG = (T, C) \tag{2.1}$$

where:

T = is a set of **tasks**
$C \subseteq T \times T$ = is a set of **channels** or **events**

Channels and events C are associated with a pair (t, t') where the channel is said to start at task t and end at task t', as represented by the adjacency matrix $T \times T$. An event can be connected to a task similar to a self-loop which starts at t and ends at t. In practice, however, the middleware supports an event to be attached to multiple tasks.

Tasks and channels are mapped onto a distributed system's physically (partially) interconnected nodes. This **hardware network** is modeled as an undirected graph HN:

$$HN = (N, E) \tag{2.2}$$

where:

N = is a set of **nodes**
E = is a set of undirected communication **links**

Knowing the components of the graph, their properties can be defined. Every task t has a **period** and an **output message size**. Also, every task has a **Worst Case Execution Time (WCET)**, depending on the node n it runs on. For these properties, we have:

$$\forall t \in T : period_t > 0, msg_t \geq 0, wcet_{tn} > 0 \tag{2.3}$$

The utilisation of a task t on a node n is then defined as:

$$u_{tn} = wcet_{tn} / period_t \tag{2.4}$$

The message size of a task is said to be zero if it has no outgoing edges:

$$\forall t \in T = \begin{cases} msg_t > 0 & (t, t') \in C \\ msg_t = 0 & \text{otherwise} \end{cases} \tag{2.5}$$

The **traffic** between two tasks defines how much data task t on node n is sending to task t' per time unit. It is defined as:

$$traffic(t, t') = \begin{cases} msg_t / period_t & (t, t') \in C \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

Every node n has a **load limit** ll_n:

$$\forall n \in N : ll_n \in [0, 1] \tag{2.7}$$

And every link e has a certain **bandwidth** b_e:

$$\forall e \in E : b_e > 0 \tag{2.8}$$

The **path** from a node n to a destination node n' is represented as a set of links through which a message in a channel needs to pass:

$$path(n, n') \subseteq E \tag{2.9}$$

The goal of the optimisation problem is to find a mapping function m which maps the set of tasks T onto the set of nodes N, to define a configuration:

$$m : T \to N \tag{2.10}$$

2.2.1 Configurations

The modelling, verification, and generation of configurations for ScOSA have been implemented into the application engineer's workflow by incorporating it into a program called Virtual Satellite. In Virtual Satellite, based on network and task information provided by the application engineer, the problem of mapping tasks onto nodes is translated to an Satisfiability Modulo Theory (SMT) problem [15], which is then solved by an SMT solver Z3 [16], taking into account a set of constraints. Additionally, Z3 can create mappings that minimise (or maximise) an objective, such as minimising network traffic or the computational load on nodes. Constraints, in particular, are required for the reliable functioning of the distributed system, while the objectives are mainly used to maximise performance.

When the solver finds a solution that meets all constraints, and with all objectives minimised or maximised, this solution is considered optimal and is outputted as a configuration. This method is then repeated for different mission phases, resulting in different *configuration sets*. Each configuration set has a starting configuration (configuration 0), where all nodes are healthy, and the system is fully operational. Then, a separate configuration is generated for every situation where one or more nodes have failed. This means that depending on the number of nodes *n* in the system, per configuration set, there will be around $2^n - 1$ configurations without compression [10]. A changing topology is therefore handled by reconfiguring the system to the topology's corresponding configuration. This makes the behaviour of the system for any topology fully defined. The resulting configurations in a configuration set form a *reconfiguration graph*. The reconfiguration graph is modelled as a decision graph, as in Figure 1.1b.

A configuration holds the following information:

- Node configurations (for all nodes)
- Task mappings (for all tasks)
- Network paths (for all interconnections)

A node configuration determines the role of the node in the configuration, which can be either a coordinator, observer or worker node. Task mappings define to which node tasks are mapped, while the network paths define how the tasks are interconnected. The network path is modelled as a sequence of nodes from the source to the destination.

2.2.2 Existing algorithm requirements

The existing requirements in [3, 10, 12] for an offline algorithm are implemented as a set of *constraints* and *objectives*, using a task graph model (2.1) and a hardware network model (2.2). The offline configuration generation aims to find an optimal mapping function (2.10) for mapping tasks onto the hardware network by adhering to a set of constraints.

Constraints

The first constraint defines that the **total utilization** on all nodes n should not exceed their node limit 2.15:

$$\forall n \in N : load_n \leq ll_n \tag{2.11}$$

where:

$$load_n = \sum_{\substack{t \in T \\ m(t)=n}} u_{tn} \tag{2.12}$$

The second constraint defines that the **total traffic** on all links e should not exceed their bandwidth 2.8:

$$\forall e \in E : tt_e \leq b_e \tag{2.13}$$

where:

$$tt_e = \sum_{\substack{s,d \in N \\ e \in path_{sd}}} \sum_{\substack{t,t' \in T \\ m(t)=s \\ m(t')=d}} tt(t,t') \tag{2.14}$$

An optional constraint determines that a task t can only be executed on a set of compatible nodes S_t, where:

$$S_t \subseteq N \tag{2.15}$$

This is due to the system's heterogeneity, which means that some specialized tasks cannot run on the hardware of all nodes. Task t is **compatible** with the hardware on node n if:

$$compatible_{tn} = \begin{cases} 1 & n \in S_t \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

Objectives

An objective can be defined to determine to which respect a mapping is considered optimal. The following two objectives have been defined:

- Minimise the total amount of traffic in the network = $\sum_{e \in E} traffic_e$
- Minimise the maximum load among all nodes = $max_{n \in N} load_n$

Minimising the maximum load among the nodes can effectively be seen as **load balancing** while minimising the total amount of traffic as **traffic minimisation**.

Note that classical scheduling optimization parameters such as *task schedulability* or *peak loads* have not been considered for the generation of offline configurations.

Assumptions

With the constraints and objectives for the optimal path problem being defined, it is important to look at some of the assumptions that can be made on the model, as this will define the *bounds* of the system. The system is assumed to consist of a maximum of:

- 128 nodes
- 200 task
- 200 channels and events

The nodes, tasks, channels and events are all identified by their *id*. Note that the channels and events share the same id range, resulting in a maximum combination of 200 channels and events.

Furthermore, as part of the abstraction of the system model, only the tasks as defined in T are assumed to consume runtime in the system. Processes of the system management services and other processes running on the HPNs and RCNs are not considered. Similarly, it is assumed that only tasks in T generate traffic over the communication links in E. Because of these assumptions, it is important to pick appropriate values for the *load limit* and *bandwidth* when moving to a physical system. Also, the network of the system is always assumed to work reliably, meaning that link failures are not considered.

Finally, node failure detection is performed by the System Management Services. For the reconfiguration to work correctly, the initial detection of failures should do so as well. The System Management Services is assumed to always detect failing nodes correctly upon a heartbeat loss, and correctly reintegrate them as they recover.

Chapter 3

Design methodology

The online algorithm operates fundamentally differently from the offline algorithm. Although it solves the same problem, unique metrics need to be considered when making reconfiguration decisions during runtime. The first step is, therefore, to determine the additional requirements for an online algorithm, followed by literature research to find works for a similar situation. Finally, using the literature, a design and evaluation approach can be defined.

3.1 Additional online algorithm requirements

Determining the requirements for the online algorithm is performed in cooperation with the DLR. When moving towards the online reconfiguration, additional parameters need to be considered in addition to the existing ones for the offline reconfiguration. As the decision-making for an online reconfiguration is performed during runtime, the computation time of the decision-making needs to be considered. Furthermore, an online reconfiguration must be *as good as* the offline reconfiguration, in the sense that all the constraints, objectives and assumptions still hold for the online reconfiguration algorithm. The definition of decision-making time is to be defined and bounded during the design phase.

3.2 Literature research

Online algorithms in distributed systems are not new. To determine what work in this field has already been done and whether or not a similar problem has already been solved can be determined by performing literature research. Specifically for the literature research, two literature research questions are proposed:

- *What online reconfiguration algorithms exist for task mapping?*
- *Which of these algorithms would be most applicable to adhere to the requirements of an online algorithm for ScOSA?*

The first question focuses on the reconfiguration algorithms themselves and how task mapping can be performed on a running distributed system. With the second question, the algorithms of the first question are evaluated against the requirements of an online algorithm in ScOSA to determine their relevance through a literature review. Systematic literature research will be performed to gather information. A systematic approach was chosen to acquire literature in a structured, reproducible way to increase the robustness of the literature review.

3.2.1 Search string

A preliminary search was performed to get familiar with the topic and the amount of literature that is available while also determining the keywords to determine the search string. It was found that the field of *online* or *dynamic* algorithms is quite broad, as it is used in several types of distributed systems. The search was thus limited to only include algorithms that focus on task scheduling or allocation for heterogeneous systems. The Scopus academic database was used for the search. The search has been limited to include works from 2015 or newer, which, combined with the keywords, result in the following search string:

> *TITLE-ABS-KEY(distributed AND system AND heterogeneous AND (task* AND (schedul* OR allocat*)) AND ((online OR dynamic) AND algorithm)) AND PUBYEAR >2014*

This search resulted in **207** papers in Scopus on the 8 of March 2023 when searching in the title, abstract and keywords. Most of the results, however, were found to be irrelevant, requiring an iterative filtering process.

3.2.2 Filtering

First iteration

In the first filtering iteration, the papers were filtered on their eligibility by scanning the paper titles and abstracts. Additionally, the papers were roughly sorted based on the *type* of distributed system they were aimed at, resulting in the following:

- Cloud computing, **17 articles**
- Grid computing, **6 articles**
- Edge computing, **4 articles**
- Cluster computing, **3 articles**
- Distributed computing, **36 articles**

After the first filtering iteration, a total of **66** papers remained. The distributed computing type was used as a general term for papers which were not written for a specific target, thus resulting in the largest group. It's apparent that cloud computing is a popular type of distributed system where online algorithms for task scheduling are used.

Second iteration

For the second iteration, the paper abstracts and texts are scanned to filter for the applicability of the papers. Some of the distributed systems, for example, are too fundamentally different from ScOSA, making their approach largely irrelevant. *Edge computing*, for example, with its loosely coupled, dynamic system topology, is not compatible enough with ScOSA, while for *Cloud computing*, often monetary parameters are considered, which do not apply for ScOSA. After the second filtering iteration, a total of **29** papers remained.

Third iteration

For the third and final iteration of the papers from Scopus, the full paper texts were used to determine their relevance to the topic. With the remaining papers from the second iteration stage already being quite relevant, this filtering iteration is more strict, focusing on the algorithms that were presented. Only online algorithms should be considered while also filtering based on the capabilities of the algorithms. After the third filtering iteration, a total of **14** papers remained. Three common references of the remaining papers with a high impact have been explored and added as well to ensure that the development of online algorithms is properly captured. This resulted in a total of **17** papers that strongly relate to the topic of this project in terms of the algorithms being discussed.

Chapter 4

Design

This chapter presents the design of the online algorithm, starting with the definition of additional requirements, followed by the literature research finding and a literature review. After the literature review, the design approach and finally the algorithm design are presented.

4.1 Defining additional requirements

Additional requirements have been added for the online algorithm to define and bound its behaviour during execution. Unique for the online algorithm is that for each task t it should always make a **decision** to which node n it should be scheduled within a bounded time (converge), where time is defined as clock c:

$$decision_{tc} \in \mathbb{R}_{\geq 0} \tag{4.1}$$

Where:

$$c \in \mathbb{R}_{\geq 0} \tag{4.2}$$

A decision can either be that a task can be successfully scheduled or that the algorithm did not find a task mapping which meets all constraints within the bounded period, resulting in a switch to safe mode. Similar to an offline reconfiguration, tasks can not be scheduled anymore at some point, as the required runtime is not available anymore in the system when too many nodes have failed. At this instance, the system switches to safe mode, where it awaits instructions from the ground. A *decision* to schedule a task t between start time c_0 and time c of the algorithm is defined as:

$$decision_t = \begin{cases} T \rightarrow N & schedulable_t \wedge (c - c_0) \leq bound_{algo} \\ safemode & \text{otherwise} \end{cases} \tag{4.3}$$

Where a task t is considered schedulable if there exists a task mapping where the utilization constraint on node n holds, the traffic constraint on link e holds, and the hardware is compatible:

$$schedulable_t = \begin{cases} 1 & \exists (T \rightarrow N) : load_n \leq ll_n \wedge tt_e \leq b_e \wedge compatible_{tn} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

After a scheduling decision has been made, it should be applied by reconfiguring the system. The **reconfiguration time** of the system's switch to the next configuration, in terms of clock c 4.2 is defined by:

$$reconfiguration_c \in \mathbb{R}_{\geq 0} \tag{4.5}$$

The **total reconfiguration time** is then determined by the sum of the decision-making time $decision_{tc}$ and the reconfiguration time $reconfiguration_c$. The total reconfiguration time trt for a set of tasks t_{set} is defined by:

$$trt = \sum_{\substack{t_{set} \in T \\ t \in t_{set}}} decision_{tc} + reconfiguration_c \tag{4.6}$$

The total reconfiguration time can then be used to specify the maximum time the system is allowed to be in a "state of reconfiguration", in which the online algorithm should make the scheduling decisions for all tasks in t_{set} and reconfigure at the end. The **maximum reconfiguration time** is defined by the trt_b bound:

$$trt \leq trt_b \tag{4.7}$$

Finally, to be able to analyse and improve the decision-making of the online algorithms **traceability** needs to be added. This will improve the understanding of the states of the system, as decisions are made by the online algorithm.

Additional system objectives

Using the offline algorithm, the system's load can be optimally balanced as part of the objectives. The online algorithm, however, will not always be able to do this as it is bounded by time. Therefore, as the online variant will likely be less balanced than the offline one, it is expected that the situation where tasks cannot be scheduled anymore will occur earlier than for the offline algorithm, influencing the *availability* of the system. Availability is an important metric on which space avionics are judged by the DLR. Therefore, keeping the availability high is added as the *availability maximization* objective, which means that the trt should be kept as low as possible.

4.2 Literature research findings

The findings from the literature research method as defined in (3.2) are discussed with regard to their respective fields. Also, the algorithms are compared in terms of their unique capabilities.

Cloud scheduling

Starting with cloud computing, which is one of the largest distributed systems fields nowadays. In the cloud environment, clients can request services or run applications, commonly referred to as tasks, while the underlying infrastructure is being masked. When a client (dynamically) sends a request, a task is scheduled to one or several nodes (servers) of the distributed system. For this, online scheduling algorithms are used, for which a flexible approach has been proposed by Zohrati *et al.* [17], which focuses on reducing waiting times and makespan while also supporting load balancing. It is based on a combination of the greedy and min-max approaches for scheduling. Karmakar *et al.* [18] proposed another approach for the cloud, focusing on makespan to reduce monetary cost and heterogeneity by introducing a dynamic version of the Heterogeneous Earliest Finish Time (HEFT) algorithm. The tasks are modelled as a Directed Acyclic Graphs (DAG), similar to the task model of ScOSA.

Priority based scheduling

Zheng *et al.* [19] also used DAGs to evaluate an online scheduling algorithm but did not focus on the cloud specifically. A priority-based scheduling scheme is presented, aimed at maximizing parallelism. Liu *et al.* [20] also focuses on parallelism by implementing an algorithm for precedence-constrained tasks on heterogeneous systems in general, modelled as a DAG. Similar to priority-based scheduling, a rank is assigned based on priority criteria. This algorithm uniquely also considers the readiness of successor tasks to improve the makespan. Sahoo and Padhy [21] implemented an alternative approach to a priority-based scheduling algorithm by using a neural network for high-performance computing. The algorithm consists of two phases: phase one assigns a priority to a task, and the second phase maps the tasks to the processors. The algorithm optimizes for both makespan and maximizing processor utilization. Hu *et al.* [22] also uses a priority-based algorithm for DAG models but focuses on mixed-criticality systems. A separation is made between high and low-criticality tasks to determine how important a deadline miss is. The HEFT algorithm is extended by allowing mixed criticality and the introduction of virtual deadlines. Canon *et al.* [23] focuses on (heterogeneous) high-performance computing in general, looking at the competitiveness of online algorithms compared to offline algorithms.

Multi-objective scheduling

Krishnan and Thiyagarajan [24] focus on a multi-objective algorithm for fog computing. They propose an algorithm that also consists of two phases, where the first phase focuses on task ordering using the HEFT algorithms and the second phase on task assignment. They use a multi-objective ordering algorithm which ranks tasks based on a ranking equation. Another multi-objective algorithm is proposed by Chatterjee and Setua [25], which is also based on the HEFT algorithm and a ranking function. The algorithm also looks at the deadline satisfaction rate while optimising for the total network execution time and mean load.

Alternative scheduling techniques

Xu *et al.* [26] focuses more on hard deadline constraints aimed at volunteer computer platforms. In volunteer computer platforms, the nodes in the system themselves volunteer to execute tasks to a master who can decide to allocate tasks to them. Here, nodes can thus indicate how many resources they have left. The online algorithm uses a prediction model to predict the completion risk of each task based on historical data. Based on this risk, tasks are assigned priorities which are used to assign them to a node. Looking at the time aspect of an online algorithm, finding an optimal solution according to objectives is challenging within a bounded time. Therefore, Eskandari *et al.* [27] proposed an iterative algorithm combined with DAG partitioning. It aims at making the search for a solution more efficient by not searching through the entire state space but through smaller partitions to improve the scheduler's efficiency. Ahmad *et al.* [28] proposes a DAG workflow scheduling algorithm for heterogeneous computing based on a genetic algorithm to pursue a schedule with an optimal makespan. Genetic algorithms can search larger state spaces while ultimately converging in a relatively small time span. The genetic algorithm is combined with HEFT to reach an optimal schedule in fewer generations than classical genetic algorithms.

Fault-tolerant scheduling

Mei *et al.* [29] uses a DAG to research the scheduling problem of tasks for fault-tolerant heterogeneous systems specifically. Their system architecture resembles ScOSA in the sense that a similar node hierarchy is used, with coordinators (masters) and workers. The performance of different online scheduling algorithms is compared with respect to their makespan and guarantee of successful execution. Dan *et al.* [30] focuses explicitly on the fault-tolerant scheduling problem for heterogeneous onboard systems. They adapted it to suit the resource constraint characteristics of onboard systems while focusing on a balance between makespan, reliability, and power consumption. Also, the concept of backup scheduling is proposed, including convergence monitoring in case of faults during the reconfiguration.

4.3 Literature Review

As a result of the literature research to answer the first literature research question, several algorithms have been discovered. It's clear that a wide variety of online algorithms are available, all designed for particular heterogeneous distributed systems. The online scheduling algorithms have been split up into *cloud scheduling*, *priority-base scheduling*, *multi-objective scheduling*, *alternative scheduling approaches*, and finally *fault-tolerant scheduling*. All of these approaches have unique properties that can be applied in ScOSA. The algorithms for the cloud feature some interesting path-searching algorithms, while the priority-based (heuristic) algorithms show how ranking functions can be applied. Multi-objective scheduling can be used to optimise for a set of objectives instead of one, increasing the balance in the system in terms of load and network usage. The alternative scheduling techniques show that several other scheduling approaches can also be feasible while being just as good, if not better, in some aspects. The fault-tolerant scheduling approaches, however, resemble the scheduling problem of ScOSA the most and provide some interesting insights into the importance of successful convergence to keep reliability high. The wide range of online algorithms indicates that no single algorithm will be best for all scheduling situations. Since ScOSA will operate in different mission phases with varying expectations of the system, the performance of an algorithm might be different for each phase. This must be considered when selecting one or more online algorithms for ScOSA.

The literature indicates the diversity of heterogeneous distributed systems and their solutions. The field stretches from loosely coupled cloud systems to more tightly coupled fault-tolerant systems. It is clear that the fault-tolerant systems share the most similarities with ScOSA, although this does not mean that the other systems are irrelevant. Looking at the fault-tolerant scheduling algorithms as proposed by Mei *et al.* [29] and Dan *et al.* [30], it is clear that the focus was more on the reliability and convergence of the algorithms, where optimisation has been mostly left out. Also, they do not include parallelism in their algorithms to improve the system's efficiency. The other algorithms, however, do include these missing aspects, such as multi-objective scheduling [24] [25] or the focus on parallelism [19]. Some solutions focus on the nodes in the system working more closely together by each node volunteering to execute tasks [26] or by operating them in smaller partitions [27].

4.3.1 Literature research conclusion

To answer the first literature research question, it was found that the existing literature provides a lot of concepts that could be useful in an online scheduling algorithm for ScOSA. A clear gap is that no algorithm exists that provides fault-tolerance, parallelism and the ability to optimise for multiple objectives in one solution. Therefore, none of the algorithms found during the literature research will take full advantage of the capabilities of ScOSA. To answer the second literature research question, fault-tolerance algorithms come out on top as the most desirable due to their focus on reliability. Especially the work from Mei *et al.* [29], which shares the most similarities regarding system architecture with ScOSA.

As was found in the literature, all of the discovered algorithms can find a schedule which complies with the constraints of a system, but the way optimisation is performed differs significantly. Most algorithms implement a ranking function to optimise for one or multiple goals but do not consider parallelism or availability. The discovered algorithms are thus only partially suitable to adhere to the requirements of the online algorithm for ScOSA.

4.4 Design approach

With none of the discovered algorithms being fully compliant with the requirements for ScOSA, the design approach is proposed. With many of the important features required by the online algorithm being partially present in the literature, a combination of the best of these algorithms is proposed. The combination would form an entirely new, tailor-made algorithm specifically designed for ScOSA. To implement this design, two design approaches were identified:

- Simulation Approach
- Functional Approach

4.4.1 Simulation approach

The *simulation approach* is often described in the literature [17, 19, 21, 22, 24, 25, 28–30]. It is a model-based approach based on an abstraction of the target system to evaluate certain aspects of the system quantitatively. In an abstraction of the system, one or more algorithms can be evaluated without the effects of other parts of the system in a completely controlled environment. This makes evaluating different aspects of individual algorithms easier while allowing multiple algorithms to be compared in the same controlled environment. Using the simulation approach, algorithms can be numerically compared one-to-one to provide a solid argument on which one is more desirable.

The advantage of this approach is that it can be used to *test and compare multiple algorithms* in various controlled situations. As a result, a decision can be made on which algorithm performs best. The disadvantage of this algorithm is that due to the model's abstraction, many real-life effects in the distributed system are ignored. An example is network delays' effect on the system, as this is expected to impact the efficiency of decision-making in the algorithm severely. This means that even though an algorithm is found to be feasible during a simulation, it provides no guarantee that it will be so as well on the target system.

4.4.2 Functional approach

The *functional approach* does not rely on an abstract model of the system but on an implementation on the target system itself. Here, one or more algorithms can be evaluated on the actual ScOSA system, allowing the algorithms to be evaluated with all real-life effects taken into account. The real-life, non-deterministic effects of the system are expected to degrade the performance of the algorithm. It is difficult to incorporate these non-deterministic effects accurately into a model, meaning that the evaluation of an algorithm in the presence of non-deterministic system behaviour can only be accurately evaluated on the target system itself.

The advantage of the functional approach is that the algorithm can be directly *tested for feasibility*, by implementing and testing it on the target system. The algorithms can even be tweaked based on the non-deterministic behaviour of the system, which is not as straightforward to do in a simulation. Additionally, the implementation on the target system will require less time than first making an abstract model of the system and implementing the algorithm in ScOSA afterwards. The disadvantage of this approach is that if the algorithm does not work well on the target system, it is difficult to determine whether this is due to the real-life effects of the system or due to the insufficient performance of the algorithm. It is, therefore, a less desirable approach when an extensive comparison between algorithms needs to be made.

4.4.3 Preferred approach

The *functional approach* was selected as the most desirable of the two approaches. Although the simulation approach is better for comparing online algorithms, one of the goals of this project is to implement an online algorithm on the ScOSA middleware so it can be used on the real-life system itself. The simulation approach would have been a good starting point to move to the functional approach afterwards, but within the limits of this project, only the functional approach is feasible. The functional approach has the ability to find a solution for *all* the objectives while being directly usable within ScOSA. It is likely that the solution will not be optimal, but it is capable of finding and solving the difficulties of an online algorithm in a real-life system. Therefore, the functional approach is said to provide the most value to the project and the DLR.

4.5 Algorithm design

The online algorithm, designed using the functional approach, is implemented in the existing ScOSA code base as a part of the *System Management Services*. Four services are involved in the detection, mitigation and recovery of the system in case of an error:

- Monitoring Service (detection)
- Reintegration Service (recovery)
- Reconfiguration Service (mitigation)
- Reconfiguration Manager (detection, mitigation, recovery)

The **monitoring service** monitors the state of other nodes by periodic heartbeat messages. If the situation occurs where a node does not answer anymore, the service will notify the reconfiguration manager about the node failure.

The **reintegration services** is used upon startup or recovery of a node. It broadcasts a system information request to all (known) nodes in the system to determine the set of healthy nodes. If it receives a response containing the node id of the coordinator, it will request the reconfiguration manager of the coordinator to be reintegrated into the system. The node will initialise the system and become the coordinator itself if no response is received within a timeout period.

The **reconfiguration manager** (service) is only active on the coordinator node. It listens to node failures or node reintegration events and will select the next configuration to reconfigure to accordingly. Once a configuration is selected, it sends a reconfiguration request to the reconfiguration service of all nodes, including itself, to keep the system running at its full potential.

The **reconfiguration service** is used to mitigate node failures by reconfiguring to a configuration that does not include the failed node. The reconfiguration service reconfigures the tasks and channels on the node and its role. The configuration id to reconfigure to is received from the reconfiguration manager.

4.5.1 Algorithm requirements

The four services above need to be adapted to implement the online algorithm. All other parts of the ScOSA middleware do not need to be changed. The online algorithm should comply with constraints to ensure dependability, as explained in (3.1). Unique for the online algorithm is that it should schedule a set of tasks within the (bounded) maximum reconfiguration time trt_b *4.7). Also, the existing requirements for the offline algorithm still stand, such as for the total utilisation $load_n$ 2.12 and for the total traffic tt_c 2.14. With the decision to go for the *functional approach*, the algorithm's constraints are partially determined by the target hardware it is designed to run on, and by the DLR.

As the online algorithm will run on the target hardware, the bounds are given values that are determined by the limitations of this hardware. For the *maximum reconfiguration time* trt_b (4.7), the value of **4 seconds** is assigned by DLR, determined by previous design goals and internal testing. The *load limit* ll_n (2.15 bound is set as a combination of the percentages of the **CPU load and memory load**, as it is currently not possible to perform a utilisation calculation with the task WCETs being unknown. The *bandwidth* b_e (2.13) is set by the network interface with the lowest link speed, as determined by SpaceWire with a link speed of *2 Mbit/s* [31]. To ensure that there will always be enough bandwidth for other services using the network and to avoid network traffic congestion, 80% of the link speed is set as the bandwidth constraint for ScOSA, resulting in a b_e of **1.6 Mbit/s**. Finally, the online algorithm should implement traceability to give insight into its decision-making process and on what information a decision has been based.

4.5.2 Combining algorithms

Several online algorithms were found for heterogeneous distributed systems. However, none of these algorithms are fully in line with the requirements for ScOSA. Therefore, a combination is proposed consisting of parts of the algorithms that were found in the literature. Utilising the best parts of these algorithms results in a novel, tailor-made online algorithm that is new within the field of distributed space avionics.

Fault tolerance

ScOSA is specifically designed to be used in space environments where dependability and availability are key. Using the offline algorithm, little communication is required to request the system to switch to another configuration, minimizing the effects of network failures. For the online algorithm, however, communication with the rest of the system is far more frequent, increasing the chances of messages getting lost. The online algorithm, therefore, requires fault-tolerance, built into its scheduling procedure to *ensure* that messages have successfully reached their destination and that the destination performed the correct action.

Therefore, a fault-tolerant *scheduling* algorithm is needed, similar to what Dan *et al.* [30] and Mei *et al.* [29] propose. The work from Feng *et al.* focuses specifically on the guaranteeing of successful scheduling, which is an important guarantee to give for the dependability of the system. The guarantee of successful task execution is changed to a guarantee of a successful task allocation, as it is currently not possible to check for the successful execution using the *Tasking Framework*. Convergence monitoring and the backup schedule that Feng *et al.* propose can improve the system's response time in case a scheduling failure occurs. Although this would improve the efficiency of the algorithm, it is not essential for the online algorithm's functioning and was, therefore, not incorporated.

The fault-tolerance of the scheduling algorithm is seen as the ”shell” that wraps around the scheduling procedure. It is used to check at several instances that its execution and communication are working as intended. Upon a detected failure, the scheduling process should be re-attempted or cancelled based on the type of error and state of the system.

Events

The online algorithm is designed to act upon the arrival of *events*. Upon the arrival of an event, the online algorithm is designed to *adapt* the system to the changing situation. The algorithm should respond to the following events:

- New task event
- Scheduling failure event
- Node recovery event
- Node failure event

The **new task event** is generated when a new task is submitted to be scheduled which has not been scheduled before. This task can, for example, be dynamically loaded during runtime. This could be useful for runtime updates without interrupting the system.

The **scheduling failure event** is generated when a task is unsuccessfully scheduled to a node. This can occur due to insufficient resources being available on the node or the communication being severed. The task can be rescheduled to another node, or in case this is also not possible, *graceful degradation* can take place or a switch to safe mode. If the scheduling failure event occurred due to a node failure, the node failure event will be called instead.

The **node recovery event** is called when a node request reintegration into the system. When a node has been in the system before, the system can recover to a state where the node was included. When a new node that has not been seen before enters the system, the system can optimise to redistribute the load over all nodes equally.

The **node failure event** is called when a node failure is detected. If a node fails, the tasks that were running on it should be scheduled to other nodes. The whole system should also be made aware of the failure so other nodes no longer attempt to engage with it.

Algorithm input

As mentioned above, all events will trigger a scheduling procedure using the online algorithm. The decision-making of the online algorithm will be based on its input parameters. The input of the online algorithm will be a data structure containing the following:

- A set of tasks to schedule
- A set of healthy nodes

The set of tasks to schedule contains all the tasks that need to be scheduled. For example, after a node failure, all the tasks that were running on the failed node will be put in this set. Next, the set of healthy nodes will contain all the operational nodes in the system, thus excluding the node that just failed. The online algorithm runs in its own separate service, sequentially handling sporadically arriving events.

Algorithm phases

The online algorithm has been split into six phases, which it will step through sequentially. These phases take care of two scheduling responsibilities: assigning node roles and assigning tasks to nodes. This sequential scheduling approach makes the algorithm easily adaptable and extendable. The flow chart of the scheduling phase can be seen in appendix A.1 and is explained, per phase, in further detail.

Phase 1: Node roles

The algorithm starts at *phase 1* when an event arrives. In this phase, the algorithm takes care of the assignment of node roles. It attempts to assign the *coordinator*, *observer 1*, *observer 2* and worker roles to the healthy nodes in the system. If no coordinator is present in the system, one is to be selected by the online algorithm. Ideally, the coordinator is in the ”centre” of the system, with it being as physically close as possible to all nodes in the network. By placing the coordinator as central as possible, communication delays are expected to be reduced. To implement this, a path-searching algorithm is needed. Currently, however, the facilities required to implement this are not implemented in the *SpaceWireIPC*, requiring an alternative coordinator selection method based on the node id. In the alternative method, the node with the lowest id is automatically selected as the new coordinator from the set of healthy nodes initially in the system. The node with the second lowest node id becomes observer 1, and the node after that observer 2. All other nodes then get the ”worker” role assigned.

The system needs to be aware of the location of the coordinator, observer 1 and observer 2. If any of these nodes roles change, this update is sent to all nodes in the system through a *partial reconfiguration*. If only a change takes place considering a worker node, other nodes do not need to be made aware of this.

A partial reconfiguration consists of a message that contains the ids of the coordinator, observer 1, observer 2 and a task id, *if* a task should be scheduled to the receiver node. The task id field is ignored when only the node role is updated through a partial reconfiguration message.

Phase 2: Check cache

After the node roles have been assigned, and if the node is the coordinator, the algorithm continues with phase 2. In this phase, caching is implemented to provide quick responses to situations that have already occurred before. For example, if a system setup with a set of healthy nodes has occurred before, the system can load the task scheduling scheme it used before for this set. There is, however, a limit on how many cache entries can be stored due to the limited availability of memory. The cache is limited by its fixed size and should only contain entries of situations that occur most frequently. However, providing a mechanism to determine which entries are most relevant for caching falls outside of this project’s scope, and a simplified version is implemented instead. The simplified caching mechanism simply appends all the scheduling decisions it can store until it fills up. In doing so, the performance of a cache load can still be evaluated with respect to the time it takes to handle an event. If a cache entry is loaded, no further scheduling needs to take place, and the algorithm proceeds by moving directly to *phase 6* to finish the reconfiguration. If no cache entry is present, the algorithm continues to *phase 3*.

Phase 3: Prioritise tasks

Without a cache entry to load, the full program flow through all six phases is taken. In *phase 3*, the tasks are prioritised to determine which tasks from the input set should be scheduled first. First, the set of tasks that *still need to be scheduled* is copied to a list. If the list is empty, meaning that there is no more task to schedule, the algorithm continues with *phase 6* to finish the reconfiguration. Otherwise, it continues with prioritising the tasks in the list.

Tasks are implemented by ScOSA using the *Tasking Framework*, as explained by the model of 2.2. Important to note is that in its current form, tasks in ScOSA do not implement *deadlines* and *mixed-criticality*, ruling out task prioritisation based on these parameters. There are several other ways tasks can be prioritised, for example, with the objective of:

- Reducing response time [17, 32]
- Reducing makespan [17, 20, 21, 28–30, 32]
- Reducing execution time [18, 32]
- Reducing resource consumption [18, 26, 27, 29, 30]
- Maximising parallelism [19, 20]
- Balancing multiple objectives [24, 25]

However, the current code base does not support any of these tasking prioritisation objectives. Several time-related parameters are currently not available through the Tasking Framework API, such as the arrival time, execution time, and finish time of (periodic) tasks. Therefore, prioritisation based on *response time* and *execution time* cannot be implemented. Ideally, a multi-objective task prioritisation mechanism is implemented, focusing on utilising parallelism, reducing resource consumption and minimising makespan. A multi-objective prioritisation mechanism can be achieved by utilising a *ranking-function* to achieve a balance between multiple parameters that influence the system's performance.

However, for the evaluation of the online algorithm, the task prioritisation step is arguably less important than the next phase, where nodes get prioritised. With the algorithm needing to be evaluated more on *feasibility* than on *performance*, it was decided to opt for a simplified task ranking step. In the current task prioritisation phase, tasks are prioritised based on the amount of *task outputs*. A task output is used as the input of a successor task, meaning that if a task with a lot of outputs is not running, successor tasks will not be activated due to a missing result from its predecessor. It was therefore chosen to schedule these tasks first, with the aim of keeping as many tasks in the system available as possible. With the tasks prioritised, the algorithm will continue to *phase 4*.

Phase 4: Prioritise nodes

In the node prioritisation phase, the "best" node is selected to execute the highest priority task. Here, the coordinator cannot select the "best" node from the limited amount of information it has on other nodes but instead has to gather this information by requesting it from the them. The coordinator requests each node to calculate its priority *itself* rather than doing this itself. This is effectively a parallelisation step by spreading the priority calculation over the distributed system. Crucially, each node should implement the exact same, *normalised*, priority calculation method.

The normalised priority is defined by the sum of three values:

- Eagerness
- Suppressor
- Accelerator

The **eagerness** is a positive value that describes how eager a node is to calculate a task, similar to volunteer computing [26]. The eagerness will be based on a normalised representation of the available resources in the system, such as CPU or memory, looking only at the node itself.

The **suppressor** is a negative value representing the network's load. If a task causes an increased load on the network that approaches the load limit, the suppressor value's magnitude increases.

The **accelerator** is a positive value to give a boost to nodes that are deemed as more desirable. For task scheduling, a task with a predecessor or successor task on that same node will not cause more traffic on the network and will reduce delays between tasks. In this case, the accelerator value is rewarded for task *locality*, to avoid tasks from being scheduled physically far apart from each other in the system.

The *suppressor* and *accelerator* calculation, however, can currently not be implemented due to the limited access to the network layer by *SpaceWireIPC*. This makes it not possible to access the network load or to determine who the neighbouring nodes are. Nevertheless, the *eagerness* calculation already provides some metrics to prioritise the nodes. The OUTPOST library provides only limited access to lower-level system information, so only the percentages of CPU and memory usage are currently used for the eagerness calculation.

After calculating the eagerness value, each node sends back its result to the coordinator. The coordinator waits for all the responses, limited by a timeout, and puts them in a list for sorting based on eagerness. The eagerness values are then used to prioritise the nodes by means of a sorted list, containing the highest priority nodes in ascending order to be used in *phase 5*.

Phase 5: Schedule task

The results from *phase 3* and *phase 4* will be used in *phase 5* to schedule the highest priority task to the highest priority node. This involves a single partial reconfiguration request directed at the highest priority node, containing the request to execute the highest priority task. Upon reception by the *Reconfiguration Service* of the highest priority node, a *dynamic configuration* is used to store the task change before actually applying it in *phase 6*. Upon an acknowledgement of the partial reconfiguration, the online algorithm removes the task from the set of tasks that need to be scheduled and goes back to *phase 3* to schedule any remaining tasks. If there are no more tasks to schedule, the algorithm moves to *phase 6* to make the changes definite.

If a partial reconfiguration request cannot be applied and a failure is communicated back to the coordinator, the task should not be removed from the set. Instead, it should now be attempted to schedule the task to the second highest priority node. This however, is currently not fully implemented, and instead, the task is simply not removed from the set of tasks to schedule to let it go through all the scheduling phases again.

Phase 6: Finish reconfiguration
In this final phase, the changes that have been made by the online algorithm through the partial reconfiguration request are made definite. Note that all nodes continued executing tasks in *phase 1 to 5*, even while receiving partial reconfiguration requests. In *phase 6*, however, the affected nodes will temporarily stop executing to apply the changes that were made by the partial reconfigurations through a reconfiguration to the *dynamic configuration*. To do this, the coordinator sends a reconfiguration request to only the nodes that were affected by the scheduling process. At this moment, the affected nodes stop executing tasks and reconfigure to the dynamical configuration. Upon completion, a reconfiguration successful message is sent back to the coordinator. The coordinator waits for all reconfiguration successful messages from the node, and finally applies the new configuration by sending a reconfiguration finish message. At this moment, the nodes start executing tasks again, making the whole system available again.

Graceful degradation

At some point, when there are a lot of tasks already scheduled and a low number of healthy nodes are available, the situation will occur that not enough resources are available to schedule a task. When this situation occurs, the system can either move to *safe mode*, as was standard for the offline algorithm or opt for a *graceful degradation* of the system. With graceful degradation, decisions can be made to stop ”less important” tasks to schedule other more important ones. This, however, would require the availability of a mixed-criticality system, so that the decision of what tasks are less important than others can be determined by the application developers. As this is currently not available, a *safe mode* is used, similar to the offline algorithm.

Optimisation

With the algorithm focusing on the scheduling of tasks, a different approach is needed when a node reintegrates into the system. With a reintegrating node, more resources become available, meaning that nodes with a high utilisation can have some of their tasks off-loaded to the reintegrating node to create a better overall balance while also having more opportunities to exploit parallelism. Therefore, when a node reintegrates into the system, the coordinator should *balance* the load over the system. In an operational system, optimisation is important to continue to get the most out of the system. However, this was considered outside of the scope for evaluating the online algorithm, as it would not utilise a scheduling procedure such as for the online algorithm.

4.5.3 Implementation

Software

The online algorithm and its six phases are implemented in the *Online Reconfiguration Manager* in ScOSA. It is a derived class from what was originally the *Reconfiguration Manager*. The original offline algorithm is also moved to a derived class, called the *Offline Reconfiguration Manager*. The *Reconfiguration Manager* is now turned into the base class and contains common functionality between the offline and online managers. Figure 4.1 shows a visualisation of this class structure.

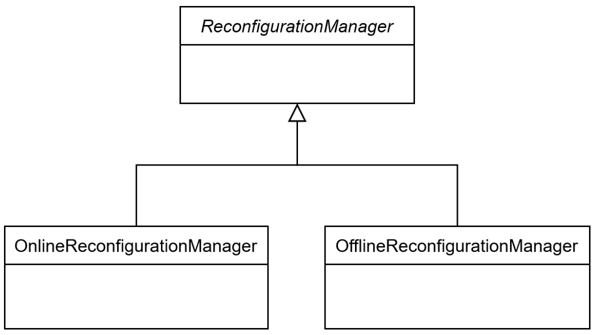


Figure 4.1: Reconfiguration Manager inheritance class diagram

The actions taken by the online algorithm, from an event to the reconfiguration finish message, can be seen in appendix A.2 through an example where the failure of node B is shown. Upon a heartbeat loss, SpaceWireIPC, the NetworkDispatcher, and eventually the *Online Reconfiguration Manager* are called. With Node A being the coordinator, the node failure is handled by processing the scheduling action. In the scheduling loop, a partial reconfiguration is used to schedule a task to node C. The scheduling loop is visualised in more detail in appendix A.3, visualising the phases of a coordinator scheduling a task to a node.

Traceability

Traceability is achieved by utilising the existing logging capabilities of ScOSA. ScOSA can output logging information to the terminal or to an output file, which is ideal for post-processing. Important is that it needs to be clear by operators on the ground on what information the decisions by the online algorithm are based on. Therefore, when a scheduling action takes place, the calculated *eagerness* of all nodes in *phase 4* are logged, followed up by the decision to schedule a task to a node. Finally, the *time* to schedule the full set of tasks, from *phase 1* up till *phase 6* is logged as the *decision-making time*, together with the *reconfiguration time* of *phase 6*.

Chapter 5

Evaluation

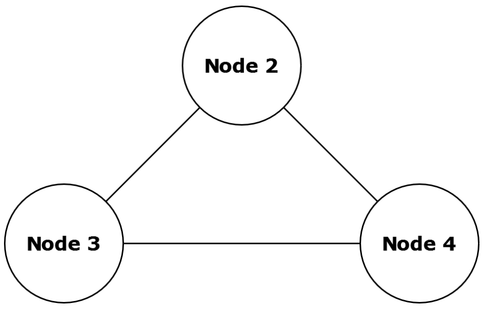
In this chapter, the method of evaluating the online algorithm is presented, focusing on the test setups and how they should be used for gathering results.

5.1 Evaluation methodology

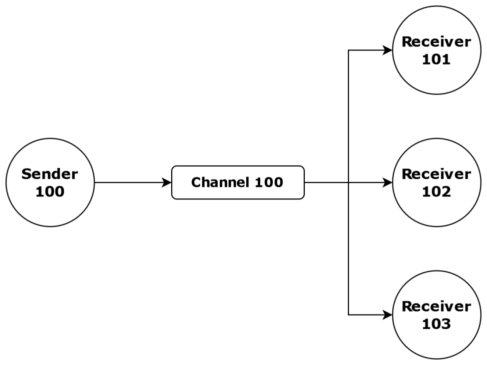
The functional approach has the advantage that it can be evaluated on ScOSA's target hardware. Therefore, to test the online algorithm, a test setup is needed. As the online algorithm is to be compared to the offline algorithm, the setup for the offline algorithm can be used. There are already some test programs to demonstrate and test the existing code base, of which one can be selected. An example program (*threeNodesExample*) designed for three nodes, running four tasks, is selected for its simplicity and well-defined offline configurations. Using this program, the algorithm's compliance with the requirements can be evaluated, as well as the functioning of the overall design of the algorithm.

5.2 Evaluation approach

Two separate test setups have been created to evaluate the online algorithm, a *time setup*, focusing on the **time** and a *network setup*, focusing on the network **traffic** required for an online scheduling procedure. Using these test setups, the feasibility of the online algorithm can be verified by testing for functionality and compliance with the constraints. Both setups can be used to compare the online algorithm with the offline algorithm by providing a constant testing environment. For both test setups, the selected example program is used.



(a) Test program of the hardware network



(b) Test program task graph

Figure 5.1: Test program configuration

Test programs

Different test programs are compiled for the test setups. They are compiled for the target hardware and to run on a server. The exact same code is used in the test programs, consisting of four tasks to test the scheduling functionality:

- Task-Id 100: Sender
- Task-Id 101: Receiver
- Task-Id 102: Receiver
- Task-Id 103: Receiver

The four tasks can communicate with each other using a channel with Id *100*, which the receiver tasks use to listen to the output of the sender task, as visualised by task graph (2.1) of figure 5.1b. All tasks are compatible with all nodes, meaning that $compatible_tn$ in (2.16) will always be *true*. The hardware network (2.2) consists of three nodes, numbered 2,3 and 4, interconnected via Ethernet, as shown in figure 5.1a.

For each test setup, three test programs are used to compare the offline and online algorithms:

- Test program 1: Offline algorithm
- Test program 2: Online algorithm with caching enabled
- Test program 3: Online algorithm with caching disabled

The two online algorithm test programs can be compared to determine the impact of caching on time and traffic. Note that the program with the offline algorithm is the original version DLR uses. It is used to compare the online algorithm programs with the offline situation to determine whether the performance of the online algorithm is satisfactory. The offline version *test program 1* reconfigures the system according to the decision graph of figure 5.2. The vertices represent the configuration id, and the edges represent a failed node. For example, when the system is configured to configuration *0* and node 1 fails, the system reconfigures to configuration *1*. If node 2 then fails as well, the system reconfigures to configuration *4*.

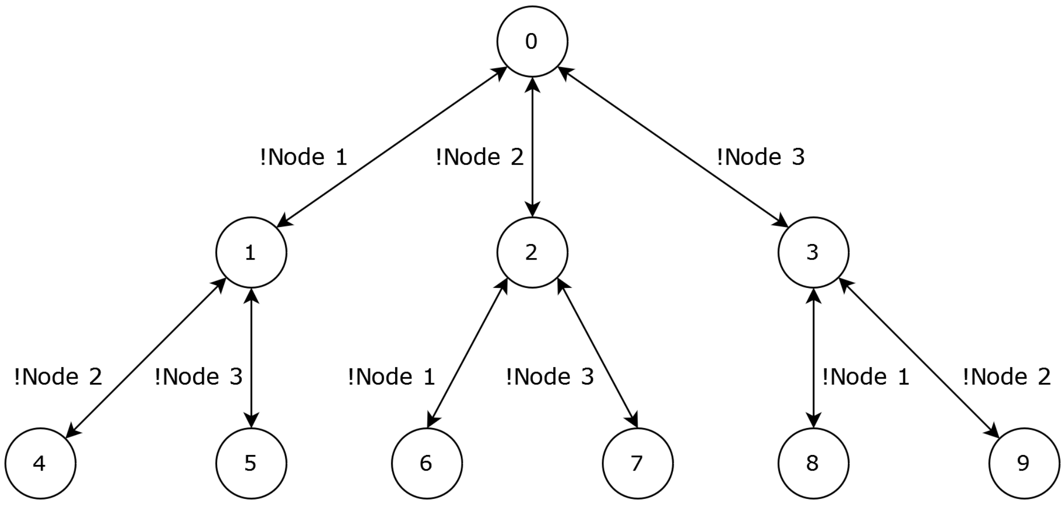


Figure 5.2: Test program 1 decision graph

Test setup 1: Time analysis

The first test setup is used to analyse time. Time is measured by looking at two delta times, the ”Reconfiguration Delta Time”, which has been defined as $reconfiguration_c$ (4.6) and the ”Decision-Making Delta Time”, which has been defined as $decision_{tc}$ (4.1).

The $reconfiguration_c$ parameter tracks how long it takes for the entire system to reconfigure to a new configuration, initiated by the coordinator. The online and offline algorithms use this parameter to track how long a ”switch” to the next configuration takes. The new configuration has been dynamically set up for the online algorithm but essentially contains the same information as a configuration for the offline algorithm. However, node role changes are faster on the online algorithm when no tasking changes are made, meaning that a varying $reconfiguration_c$ is expected for the online algorithm.

Besides $reconfiguration_c$, for the online algorithm, additional time is required for the decision-making of the online algorithm in the coordinator node. The $decision_{tc}$ parameter keeps track of how long the coordinator takes to decide and schedule a set of tasks to a node. Only the online algorithm has a decision-making time, as for the offline algorithm, the decisions are predetermined in a decision graph 5.2.

In accordance with the proposed algorithm design in (4.5), the coordinator makes the decision based on the current system state. The coordinator receives the system's state during the decision-making time over the network. Due to this dependency on the network, the network delay is also part of the overall decision-making time. The parameter is only outputted by the coordinator, as this is the only node which, at any point in time, is allowed to perform a scheduling procedure. Finally, $reconfiguration_c$ and $decision_{tc}$ can be used to calculate the total reconfiguration time trt of (4.6) to compare it with the bound trt_b.

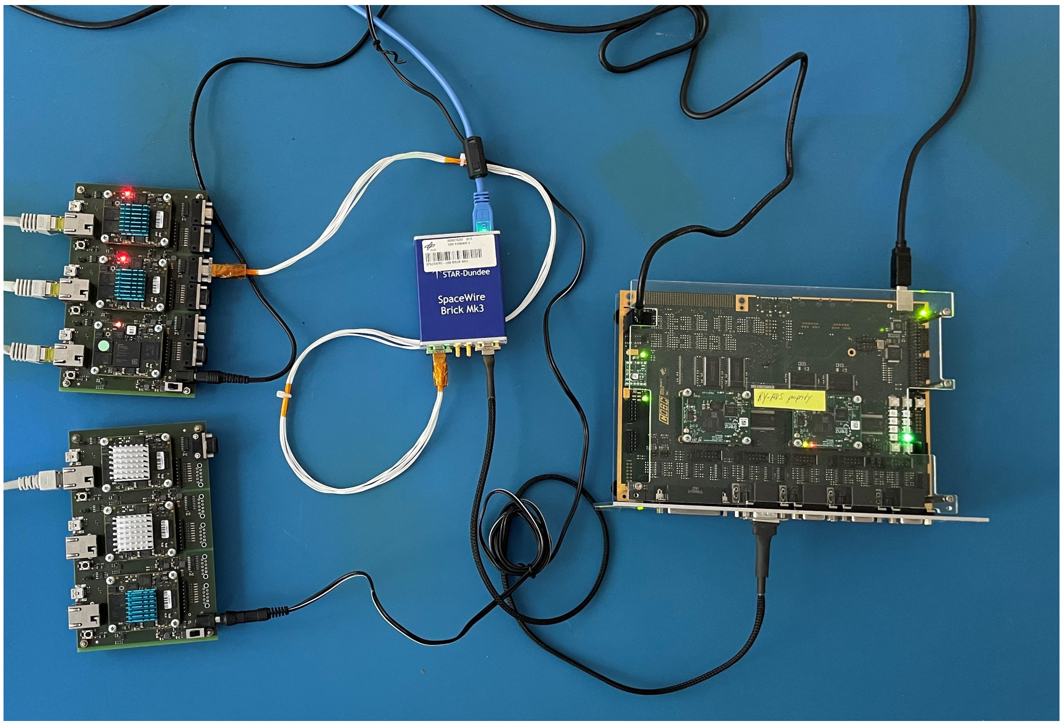


Figure 5.3: Target hardware network containing three HPNs and an RCN

In *test setup 1*, test programs are run on the actual target hardware, consisting of a network of three HPN nodes interconnected via Ethernet. By running the test programs on the target hardware, the real-life temporal behaviour of the algorithms can be determined. RCNs are not part of the network. The target hardware, with two carrier boards containing three HPNs and one RCN, can be seen in figure 5.3.

To automatically run each test program on the HPNs individually, a test script B.1 is used. On each HPN, the script is executed, which runs and kills the test program using random time intervals to simulate the behaviour of nodes failing and reintegrating. The test program outputs logging information (traceability) containing the temporal data to a log file for post-processing. The test script can then be executed for a configurable amount of time. An execution period of *5 hours* for each test program was selected to collect enough data points for statistical analysis.

Test setup 2: Network analysis

Next to changing temporal behaviour, an increase in network traffic is expected for the online algorithm compared to the offline algorithm. With the online decision being based on the real-time information of the system, naturally, more traffic will be generated to communicate this information to the coordinator, depending on the number of nodes in the system.

Again, the three test programs are used in *test setup 2*. They will, however, not be run on the HPN network but on the server with the x86_64 desktop processor using internal loop-back routing for the virtual network traffic. Using the server, network traffic can more easily be gathered, and it has the additional benefit that more *virtual* nodes can be run than just three.

The online and offline algorithms are tested for their behaviour when a single node fails, which happens to be the coordinator. A failing coordinator is chosen, as this is the ”worst case” node failure that can occur. With a failing coordinator, a new coordinator has to be selected. The new coordinator then also needs to schedule the tasks that were running on the node that failed. The coordinator selection, combined with the task scheduling, results in the largest amount of traffic.

A network test script is used to start the network capture tool *tshark* and a configurable number of nodes. The node with Id *2* is then killed, which triggers a reconfiguration procedure for both the online and offline cases. During this time, the network traffic is captured and filtered. The sum of the total traffic tt_e, as defined in equation 2.14, is then stored in an output file for post-processing. When the node *2* fails, the online algorithm will assign the new coordinator role to the node with the lowest node Id. An example of what the scheduling procedure may look like when node two fails can be seen in figure 5.4, where after a failure, node *3* becomes the new coordinator and task 100 and 101 are rescheduled to node *4*.

After the total traffic (tt_e) of all the runs has been captured, a comparison can be made with the bandwidth b_e (2.13) after converting from bytes to bits. The bandwidth is defined as bits per second, and we say that the total traffic in bits may never exceed the number of bits available per second to avoid network traffic congestion and ensure the quality of other services using the network.

With the ability to scale the system up, a unique opportunity appears to test the network traffic of the online algorithm for a system with a higher number of nodes. As the dependency on memory for the offline algorithm, as described in 1.1, shifted to a dependency on the network, it is important to determine how the network traffic scales. Additionally, it can be shown that the number of nodes in the system can scale up without an increase in memory intensity, as is the case for the offline algorithm.

The network test script is used on the server to create a virtual network with a configurable amount of nodes and triggers a reconfiguration *25* times. It iteratively creates a network consisting of *3, 4, 5, 6, 7, 8, 9, 16, 32, 48, 64, 80 and 96* nodes. A network of *128* nodes was found to suffer from buffer overflows outside of this project’s scope and was therefore not considered. The network traffic outputs from tshark are saved, filtered to contain only reconfiguration bytes, and accumulated. The accumulated result for each of the 25 runs is then stored in an output file. This data structure is then used to analyse the different test cases statistically.

The test programs *1* and *2* can be used to directly compare the online and offline algorithms when running a network of three nodes. The ”worst-case” situation is used for this comparison, with a failing coordinator and all four tasks needing to be scheduled while having no cache entry available for loading.

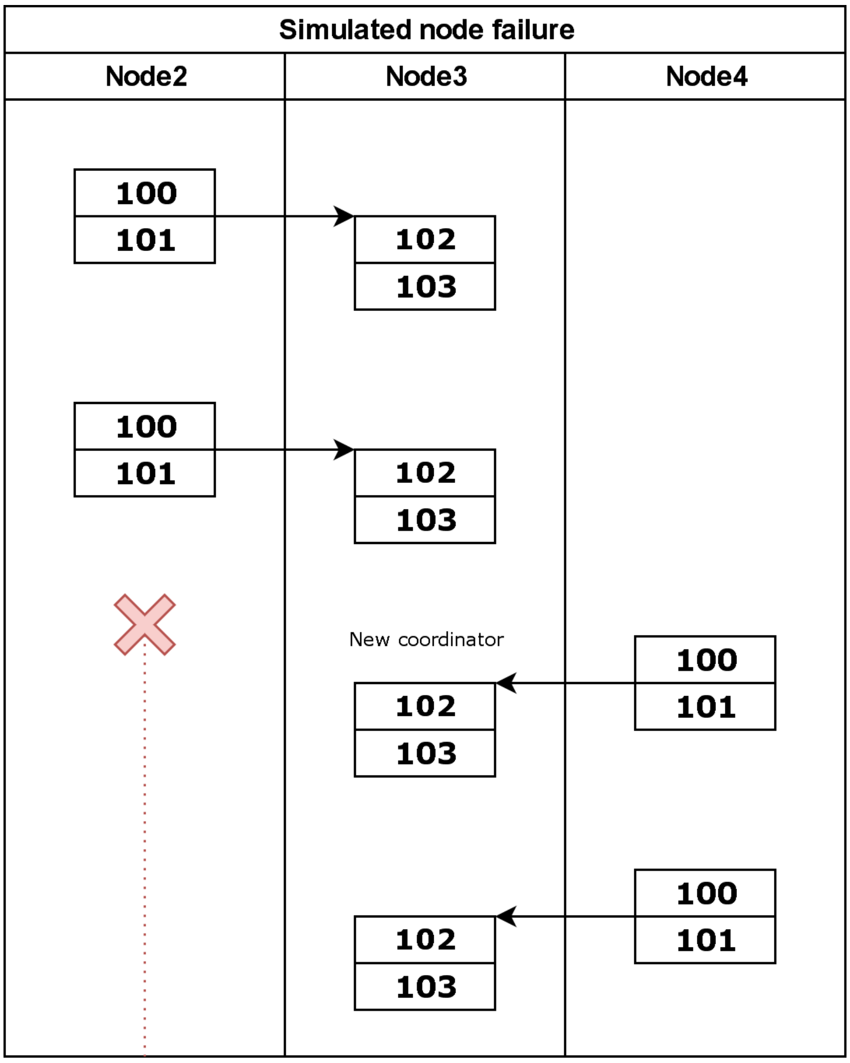


Figure 5.4: Example of a simulated node failure using online scheduling

Chapter 6

Results

The performance of the design is assessed based on the results that have been gathered by the tests as described in section (5.2), looking at temporal and network performance. The data is prepared for statistical analysis for each test setup.

6.1 Timing analysis

For the timing analysis, the test outputs from *test setup 1* are used.

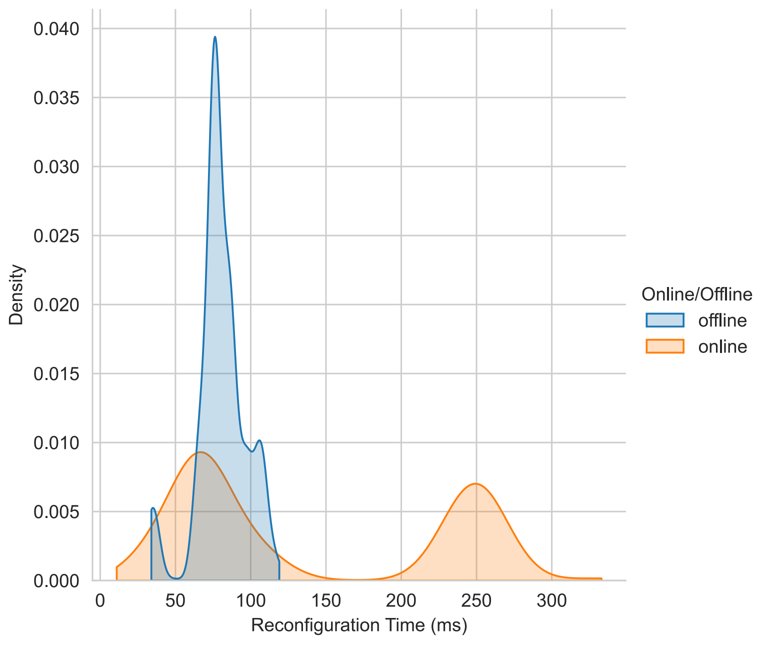
6.1.1 Reconfiguration time

The reconfiguration times of the three test programs are compared to find the differences between the offline and online algorithms, of which the results can be seen in table 6.1. Note that for the reconfiguration time, the output of the test programs *2 and 3* are combined, as caching showed to have no influence on the reconfiguration time. The difference in standard deviation between the offline and online cases can be identified and visualised with a distribution in figure 6.1, where a bi-modal distribution can be identified for the reconfiguration time of the online algorithm. This indicates two reconfiguration time ”groups” for the online algorithm, as is visualised in figure 6.2 in more detail.

Statistic	Count	Mean	Standard Deviation	Min	Max
Offline	779	80.38	16.61	34	119
Online	2327	139.04	90.33	11	333

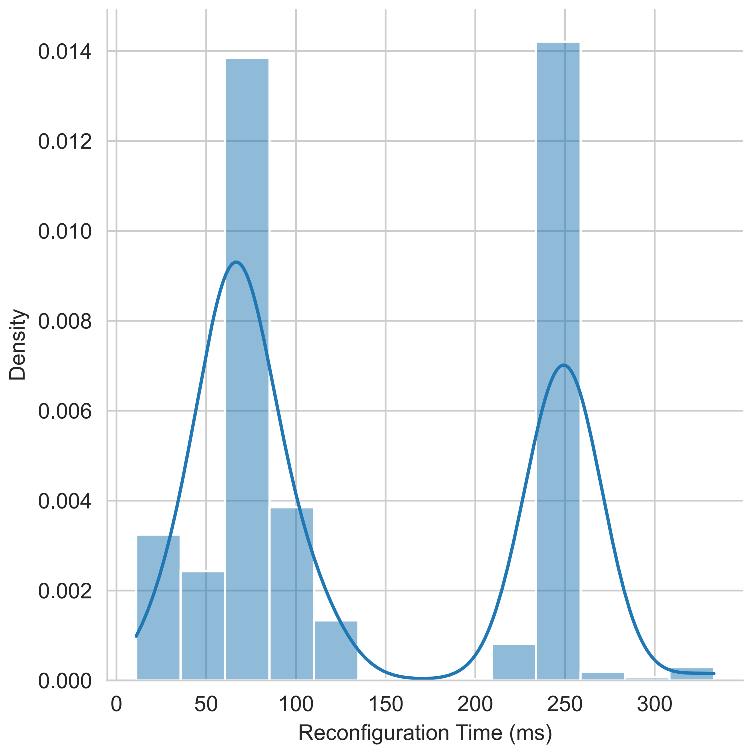
Table 6.1: Reconfiguration time (ms)
Count represents the number of reconfiguration occurrences during tests

Figure 6.1: Reconfiguration time online vs offline density plot



Kernel Density Estimation (KDE) with Gaussian kernel
Smoothing bandwidth = 1, with independent function normalisation

Figure 6.2: Reconfiguration time online histogram and density plot



KDE with Gaussian kernel
Smoothing bandwidth = 1

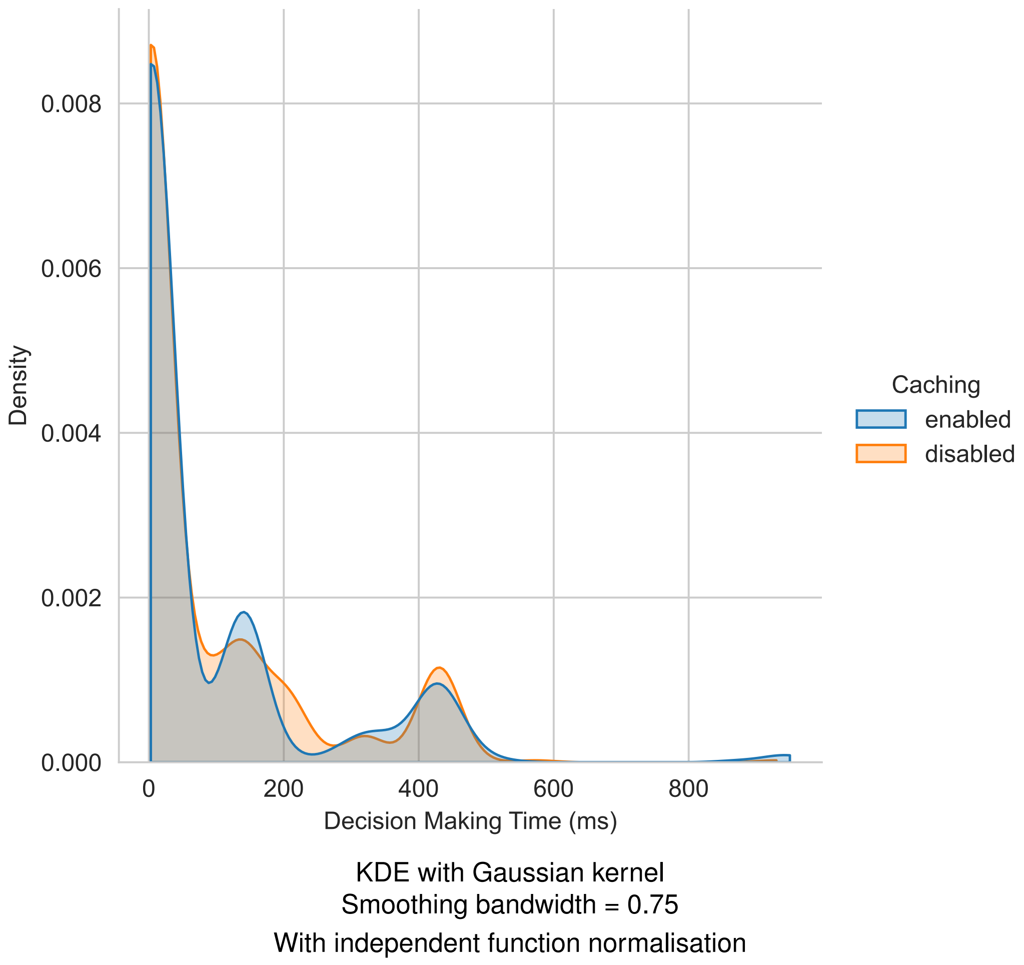
6.1.2 Decision-making time

For the decision-making time of the online algorithm, one can look at the output of the *test program 2* and *test program 3*, for the time with cache *enabled* and cache *disabled*. The results can be found in table 6.2. The two cases appear very similar, as is further backed by the plot in figure 6.3

Statistic	Count	Mean	Standard Deviation	Min	Max
Non-cached	563	85.34	140.23	3	930
Cached	570	84.92	155.91	3	950

Table 6.2: Decision making time (ms) Cache enabled vs disabled

Figure 6.3: Decision making time cached vs non-cached density plot

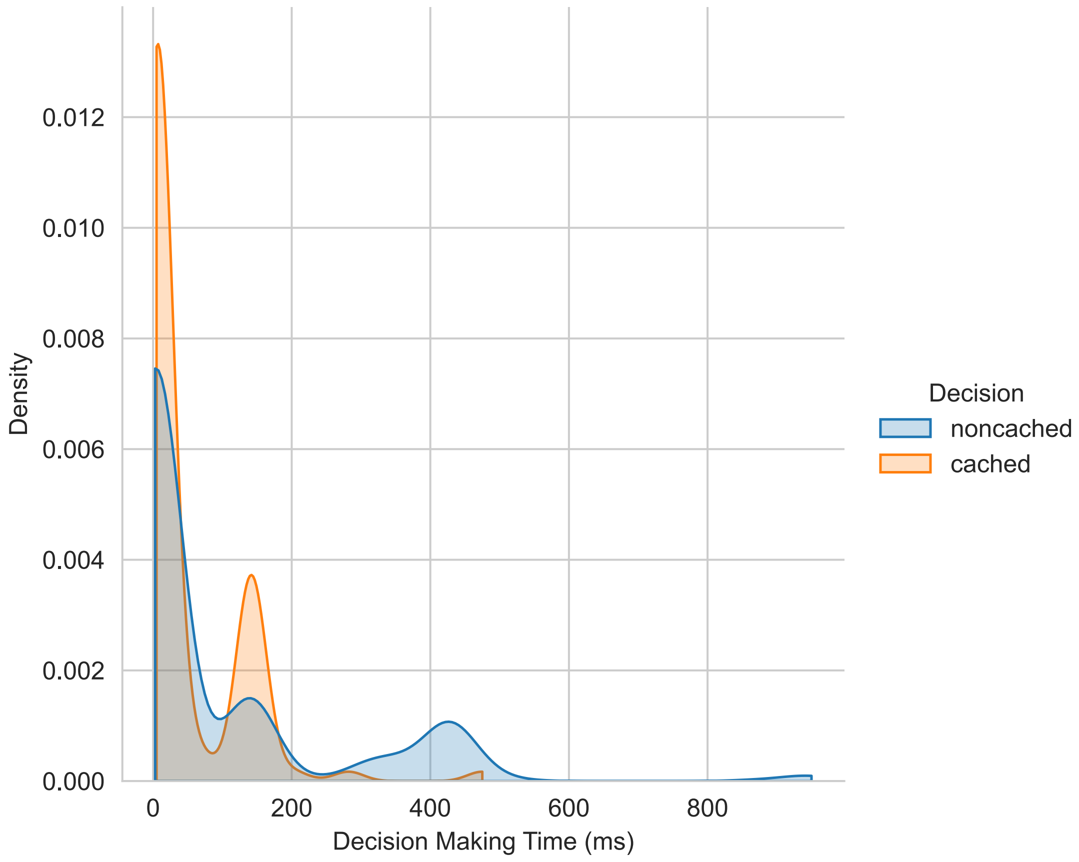


The two cases appear to be very similar, which would suggest that caching does not have a noticeable effect on the decision-making time of the algorithm. However, looking at the results from the test program with caching enabled only, and separate the *cached* decisions from the *noncached* decisions, caching does appear to have different decision-making time, as can be seen in figure 6.4. The *mean* of the test result with cache *enabled* seemed to be caused by the fact that only *108* of the *570* decisions were actually loaded from cache, as can be seen in table 6.3. With only about a fifth of the decisions being loaded from cache, the resulting *means* in (6.2) ended up very similar. Of the *108* cached decisions loaded from the cache, it can be seen that the mean is about half that of a noncached decision. Also, the deviation is smaller while having fewer outliers and a smaller max value, as visible in the box plot of figure 6.5. A difference can thus be observed when an actual load from the cache occurs, with the probability appearing higher that the decision-making time is shorter for a cached decision then for the noncached decision.

Statistic	Count	Mean	Standard Deviation	Min	Max
Cache load	108	44.94	74.19	5	475
No cached load	462	93.89	168.13	3	950

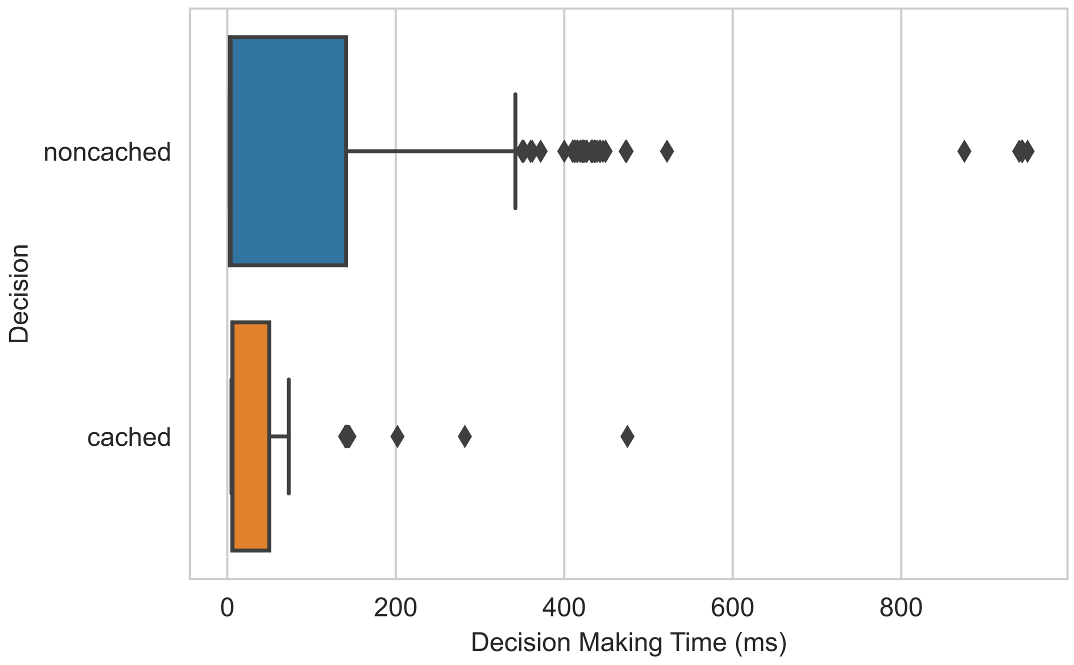
Table 6.3: Decision making time (ms) cache load vs no cache load

Figure 6.4: Decision making time (ms) cache load vs no cache load density



KDE with Gaussian kernel
Smoothing bandwidth = 0.75
With independent function normalisation

Figure 6.5: Decision making time (ms) cache load vs no cache load box plot



Finally, the outputs of *test program 2* and *test program 3* are combined to test for the decision-making time per node. The table in 6.4 shows the differences between the nodes. The count here is interesting, as it can be seen that there were more entries for node *2* and node *3* than for node *4*. This can be explained by the way the coordinator is selected in the test setup, where it is selected based on the lowest node Id. This will thus make it a lot more likely that when a coordinator fails, the lowest node Id node will take over as the coordinator, making it less likely that node *4* takes over as the coordinator.

Statistic	Count	Mean	Standard Deviation	Min	Max
Node 2	518	97.45	160.01	3	950
Node 3	432	88.19	149.55	3	944
Node 4	183	42.09	93.30	3	940

Table 6.4: Decision making time (ms) per node

6.2 Network traffic analysis

For the network analysis, the test outputs from *test setup 2* are used.

6.2.1 Three nodes

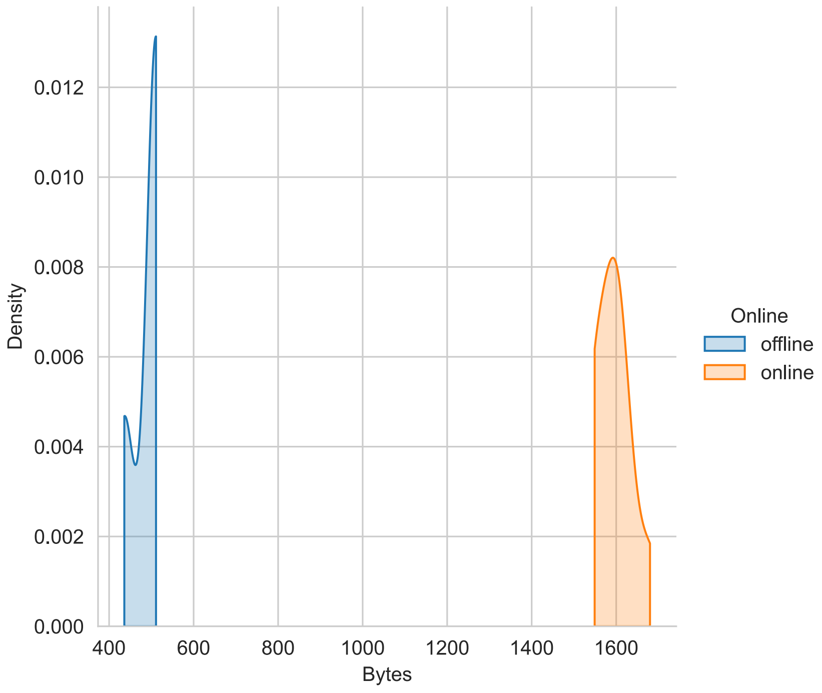
Test program 1 and *test program 2* are used to compare the online and offline algorithms one-to-one for a setup with three nodes. The results can be seen in 6.5, where an increase in the mean network traffic of over three times can be seen. In figure 6.6, the network traffic distribution can be seen for the three nodes network of *test program 1* and *test program 2*.

Statistic	Count	Mean	Standard Deviation	Min	Max
Offline	23	491.43	33.67	436	511
Online	17	1592.94	41.80	1549	1680

Table 6.5: Three nodes offline vs online network traffic (bytes)

Note also that the *count* does not always correspond with the actual 25 runs, as tshark was found to have occasional memory errors when starting a capture with already a lot of traffic going on, resulting in malformed packets. These runs were discarded and not considered in the statistics.

Figure 6.6: Three nodes offline vs online network traffic density plot

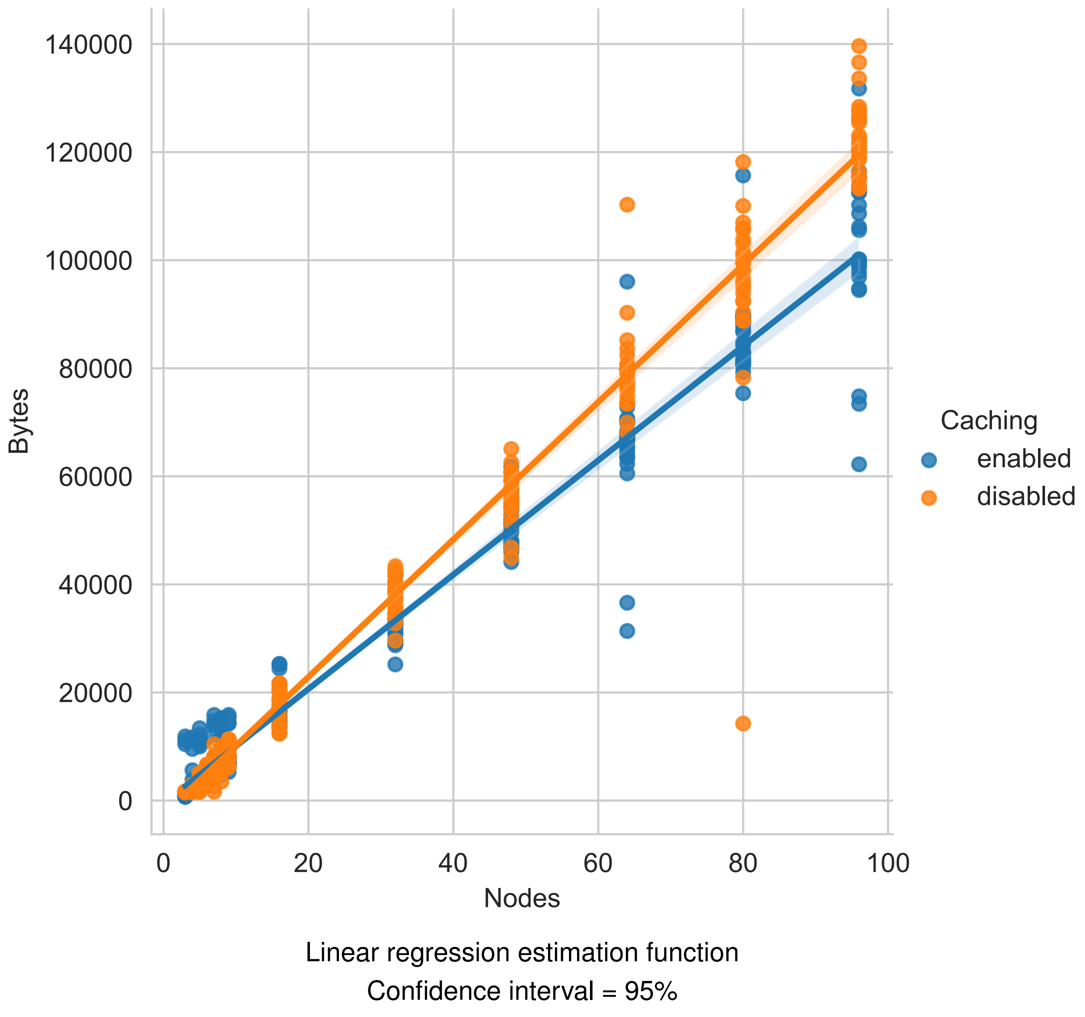


KDE smoothing with Gaussian kernel
Smoothing bandwidth = 1.25
With independent function normalisation

6.2.2 Scaling

Another important parameter is to see how the network traffic for the online algorithm *scales* when the number of nodes in the network increases. Caching is also expected to affect the amount of traffic, as a cache load does not need to request the eagerness of the nodes. In figure 6.7, the network traffic can be seen for increasing nodes, both with caching enabled and disabled. It appears that for both *test program 2* and *test program 3*, the network traffic increases linearly with the number of nodes, with the caching-enabled version requiring slightly less traffic. It thus seems like the changing network state and the subsequent informing of *all* the nodes in the system about this generates the most traffic. This is caused directly by the design decision to make the coordinator stateless and decentralise more responsibilities to the nodes, like figuring out the channel configuration.

Figure 6.7: Scaling online algorithm with caching enabled vs disabled plot



Besides some outliers, it can be read from figure 6.7 that the amount of network traffic increases linearly, with a confidence interval of 95%. Looking at the worst-case situation of *test program 3* where caching is disabled, an equation can be derived 6.2, describing the total traffic in bytes tt as a function of the number of nodes n and the number of tasks.

$$n \in \mathbb{R}_{\geq 3} : t \in \mathbb{R}_{\geq 1} \tag{6.1}$$

Where:

$$tt = (c_1 t + c_2)n - 1564 \tag{6.2}$$

The coefficients, as denoted by c_1 and c_2 in equation 6.2, determine the effect of the number of tasks on the scaling of the network traffic. From figure 6.7 equation 6.3 can be estimated for the test program with *4* active tasks.

$$tt = 1168n - 1564 \tag{6.3}$$

It was observed that a varying number of tasks also generated a linear increase in network traffic, but no time was available to implement the required changes into the ScOSA code-base and test setup to find the c_1 and c_2.

Chapter 7

Discussion

The discussion chapter describes and interprets and reflects on the the Design methodology chapter (3), Design chapter (4) , Evaluation chapter (5) and Results chapter (6) while highlighting what these results mean within the context of the project and for DLR. What the results cannot say is also discussed as part of the limitations of this research and how these are recommended to be addressed in future work.

7.1 Interpretations

7.1.1 Literature research

The literature research in the early phase of this project was important for identifying the wide range of literature regarding online algorithms for distributed systems. The literature showed the diversity of distributed systems and how online algorithms can be used as a solution for a wide range of situations. None of the literature, however, proposed a solution for a situation similar to the one of ScOSA. The literature did provide many design concepts for task prioritisation and node prioritisation, as well as for fault-tolerance and testing methods. The literature research was an important initial step towards the algorithm and test setup design. The systematic approach that was used to search for literature proved to be an effective method for filtering out irrelevant literature in a reproducible fashion.

7.1.2 Design

Algorithm design

With no existing literature available with an online algorithm that was usable for ScOSA, the decision to design a new, tailor-made algorithm turned out to be an effective way to test for feasibility. Designing, implementing, and running the design in ScOSA proved that an online algorithm is indeed *feasible*, answering the main research question in (1.2.4). Thus, The functional approach proved to be effective in testing for feasibility and provides more direct insight into the algorithm's behaviour on the target hardware than the simulated approach could have done.

Splitting up the algorithm into phases made the algorithm straightforward and easier to understand. Also, the phases were made with maintainability and flexibility in mind, meaning that they can be modified individually without having to change others. The addition of caching, although rudimentary, is seen as a potential solution to mitigate the downsides of online scheduling. Combined with cache pre-loading, an optimum between online and offline scheduling can potentially be achieved.

The online algorithm's behaviour turned out to be satisfactory. It is, however, not fully implemented yet, meaning that it is also not ready yet for deployment. Nonetheless, its implementation allows for continued development to prepare the system for its use case is space. Now that the number of nodes in the system can be scaled up again, an even higher availability can potentially be achieved compared to the offline algorithm.

Test setup design

The test setup was essential for evaluating the feasibility and behaviour of the online algorithm. The nodes of the target hardware, as well as the virtual nodes on the server, were valuable for testing and gathering results. Performing tests focusing on the temporal behaviour of the target hardware meant that a one-to-one comparison could be made between the two algorithms. On the other hand, the virtual nodes on the server provided additional testing flexibility, allowing the isolation of the worst-case reconfiguration scheduling actions. The virtual nodes made test automation easy, allowing several tests to be run for a configurable number of times. The test programs that were used during the tests proved to be simple to use and well analysable. Due to them not being complex, it allowed bugs to be spotted during the implementation phase while simplifying the post-processing of results.

7.1.3 Time

Worst-case

The time the online algorithm requires is important for the performance of the system, as it affects the availability. Using the results we can use the worst-case values for $reconfiguration_c$ of *333 ms* and $decision_{tc}$ of *950 ms* to get the **worst case** total reconfiguration time trt (4.6): $333ms + 950ms = 1283ms$. Comparing this with the bound of 4 seconds as defined in 4.7, we can see that $1283ms \leq 4000ms$ indeed holds. It is, however, higher than the worst-case total reconfiguration time of the offline algorithm, which is determined by the reconfiguration time. An over 10-fold increase in time can be observed when comparing with the $reconfiguration_c$ of *119ms* for the offline algorithm.

Reconfiguration time

An interesting pattern is visible in figure 6.1 of the reconfiguration time, as the form of a bi-modal distribution can be observed.

The two modes were found to be caused by network delays and the processing time of nodes, with the first mode around *75 ms* and the second mode around *250 ms*. When a reconfiguration occurs, the coordinator will (only) send out a reconfiguration request to the nodes that need to be changed. It will then wait for these nodes to respond with a confirmation that their reconfiguration was successful. Finally, once all confirmations have been received, the coordinator sends out a *finish* message to the involved nodes to *start* the new configuration.

The two modes are caused by waiting for the confirmations of a reconfiguration after a *node failure* and a *node recovery* event. When a node recovers and gets reintegrated into the system, its *Reconfiguration Service* will initialise the system from scratch. This first reconfiguration after a boot-up was found to be time-consuming, causing all other nodes to wait until that reconfiguration completes, resulting in the second mode. The first mode is caused by a reconfiguration after a *node failure*. Upon a failing node, none of the remaining nodes need to be initialised from scratch, causing the reconfiguration to be faster. It was also observed that a larger number of tasks in a reconfiguration causes the reconfiguration time to increase. This, however, could not be investigated further, as the algorithm currently has no support to extract the network delay times with a fine enough granularity.

Decision making time

When looking at the decision-making time, the worst-case situation to calculate the trt is not the best representation of the online algorithm's performance. It was found that the high decision-making times, visible as some of the outliers in figure 6.5, are caused by a complete restart of the network. This happens when not a single node is alive in the system, and the restarting node must completely set the system up from scratch. Also, it can be seen that these far outliers are not present for the *cached* decision-making. This is because no cached decision can be loaded when the system restarts, as no cache entries exist yet. As explained in the 4.5, the online algorithm is designed to adapt and act to changing situations such as failing nodes. A changing system setup due to nodes exiting or reintegrating into the network can be quickly and dynamically adapted to. However, setting up a large system completely from scratch takes a long time.

When looking at the online algorithm's performance under the circumstances it is designed for, namely for adapting to system changes in a running system, its performance is a lot better. By using the mean values for $reconfiguration_c$ as *139.08 ms* and $decision_{tc}$ from table 6.2 as *84.92 ms* to get trt: $139.08ms + 84.92ms = 224ms$. Comparing this to the mean $reconfiguration_c$ of *80.38ms* for the offline algorithm, a lower increase of about 2.8 times can be seen.

Caching effect

From (6.3), it can be seen that trt can be reduced even more when a cache load occurs. Instead of performing an entire scheduling procedure for a situation that has occurred before, a quick load from the cache can load the scheduling result from the past with a reduced $decision_{tc}$. By using the mean values for $reconfiguration_c$ as *139.08 ms* and $decision_{tc}$ from table 6.3 as *44.94 ms* to get trt: $139.08ms + 44.94ms = 184.02ms$. Comparing this to the mean $reconfiguration_c$ of *80.38ms* for the offline algorithm, an increase of about 2.3 times can be seen.

Caching, therefore, does have an impact on the time required by the algorithm. Still, looking at the distributions in figure 6.3, which compares the test programs with caching enabled and disabled, it can be seen that both graphs are very similar. However, this can be explained by how the test was run. When looking at the *count*(6.3) for many of the decision-making times were actually the result of a cache load, we can see that there are more non-cached (*462*) decisions than cached decisions (*108*). This means that a cache load simply did not occur as often, therefore not having a noticeable impact on the mean $decision_{tc}$. However, it appeared that the decision-making time does get reduced when a cache load occurs. If an increasing number of cache loads during scheduling can be achieved, it is expected to reduce the mean $decision_{tc}$, resulting in a better overall trt.

One way of increasing the number of cache loads is by implementing cache-preloading, as suggested in the design chapter 4.5. Common system states can be pre-calculated, similarly to the offline algorithm, and pre-loaded into the cache. For common situations, the system can then quickly load an optimised configuration. The long decision-making time at startup can be solved by implementing cache-preloading. In the situation where the system is switched on, with all nodes being healthy, a pre-loaded entry can be loaded to get the system quickly up and running with a balanced configuration. After the cache entry is loaded, the online algorithm can continue adapting to dynamic system changes for which it is designed. Even more so, having the ability to pre-load an optimised configuration for the most common system configuration, where the entire system is healthy, will cause cache loads to be far more frequent, resulting in a lower trt, and thus increasing availability.

Therefore, the trt, under situations it is designed for, is several lower than the bound trt_b in (4.7). The trt for non-cached of *224 ms* and cached *184.02 ms* is only a fraction of trt_b. Even the worst-case situation of a trt of *1283 ms* is well below trt_b, meaning that the online algorithm is compliant with this constraint.

7.1.4 Network traffic

Worst case

Another important metric is the amount of network traffic the online algorithm generates. With the dependency on the network for decision-making, the additional traffic that is generated will induce an additional load on the network links. Using the results, we can use the worst-case tt_e (2.14) value of *1680 bytes* to compare with the bandwidth b_e (2.13). By converting the tt_e to bits by multiplying by eight, we get a total of *13440 bits*. Comparing that with the bound of 1.6 Mbit per second, we can see that $13440 bits \leq 1.6 \times 10^6 bits$ holds. Comparing this with the worst-case tt_e of the offline algorithm of *511 bytes*, an increase of about 3.3 times can be seen. This is acceptable for the test program with a network of three nodes, as it is several orders of magnitude smaller than b_e.

The increase in network traffic can be explained by the way system information is gathered by the coordinator and how a new configuration is applied. For every task that is to be scheduled, the system requests the eagerness of each node, followed by a partial reconfiguration request where a task is assigned to a node. This assignment is also communicated to the other nodes in the system. Finally, the nodes where a change has been made apply and acknowledge the new configuration.

Scaling

An interesting situation occurs when scaling the system up by using a larger number of nodes while performing the exact same scheduling procedure. By doing so, a linear increase in network traffic becomes apparent as a function of the number of nodes. This linear increase can be observed by the linear regression estimations in figure 6.7 where caching is enabled and disabled. The largest system with 96 nodes had a worst-case tt_e of *139626 bytes*. Converting this tt_e to bits, we get a total of *1 117 008 bits*. This is a big increase compared to the situation with three nodes, but the bandwidth constraint still holds: $1117008 \leq 1600000$.

Caching

Looking at figure 6.7, it can be observed that enabling caching, comparatively, does not decrease the amount of network traffic by a lot. Although no eagerness calculations need to be communicated with the coordinator, a changing task allocation does need to be communicated to all the other nodes in the system. Therefore, the large number of system updates, in the form of node role changes and task re-allocations, are mainly responsible for the increase in network traffic. Caching in its current form is, therefore, not as effective in reducing network traffic as it is in reducing decision-making time.

Traceability

Traceability was another requirement for the online algorithm that has been implemented as a form of logging. For each algorithm step and decision, a trace is outputted for post-processing. The outputted files by the test programs were crucial for gathering and analysing results and proved to be an effective way of providing insight into the inner workings of the algorithm. Two snippets of a trace have been highlighted in the appendixes. One where a ”worst-case” reconfiguration takes place, where a coordinator fails, in appendix C.1 and one where a fast reconfiguration takes place due to a cache load in appendix C.2. It becomes clear that in the coordinator failure trace, $decision_{tc}$ is *433ms*, while for the cached load, this is just *6ms*. Using these (long) traces, the test results could be successfully parsed and analysed.

7.2 Implications

The results show that the designed and implemented online algorithm can indeed solve the scalability issue of the offline algorithm as formulated by the research sub-question in (1.2.4). The online algorithm requires more time to run, produces more network traffic, and cannot optimise as exhaustively as the pre-calculated offline algorithm. This increase, however, was not desirable and was not found to violate any system constraints. Nevertheless, it can be stated that the online algorithm does not perform better in terms of *time* and *network usage* compared to the offline algorithm. It does, however, solve the scalability problem of the offline algorithm, with the test programs being in line with the online algorithm requirements in (4.5.1). The online algorithm provides a solution for the offline algorithm not being able to support systems with a large number of nodes. The online algorithm, therefore, allows a distributed avionic middleware such as ScOSA to be usable for larger systems as well.

These results provide a unique view of how COTS hardware can be a potential solution for demanding environments such as in space. The related work of ScOSA was already able to show that through software, high dependability can still be achieved by utilising a distributed system. With the online algorithm, however, the system can be scaled up even further than was recently possible with ScOSA, potentially increasing the computational power and reliability of the system even more. The results show a new insight into what implications an online algorithm has on a distributed system through the consumption of resources in other aspects. It contributes to the understanding of fault-tolerant online scheduling algorithms by presenting a novel solution and highlighting its potential and limitations.

The unique approach of ScOSA is one of the many projects aiming to improve computational performance in space without sacrificing dependability. These results, therefore, build on existing evidence that using COTS components in space is a viable option.

7.3 Limitations

7.3.1 Design

Although the algorithm proved to be functional, not all aspects of the design could be implemented into ScOSA. Especially the parts of the algorithm where network information was required were not supported by the current code base and could not be implemented.

Also, *convergence monitoring* in *phase 5* of the algorithm could have been implemented to ensure that the procedure of scheduling a task to a node is successful. If the scheduling fails, the task should be scheduled to the second-best node in the priority list instead of going through phase *3, 4, 5* again. Scheduling to the second-best node, combined with convergence monitoring to keep ensure the scheduling of a task within a specific time, can increase the robustness and efficiency of the online algorithm.

Another aspect that influences the robustness of the algorithm is its ability to deal with failures and corner cases. The online algorithm has been specifically designed for dealing with node failures, not looking at other potential failures such as, for example, failing network *links*. Although these other kinds of failures were outside of the scope of this project, they should be considered to increase the dependability of the system.

During testing, it was observed that messages that require an acknowledgement or a response are often susceptible to sporadically occurring delays, slowing down the decision-making time and the reconfiguration time of the online algorithm. It is, however, currently not possible to extract the exact source of these delays, with it either being in the physical network, SpaceWireIPC or the ScOSA middleware. Additionally, the network *usage* has not been optimised, likely causing more network traffic than necessary. With more traffic needing to be handled, network delays increase even more. These network delays have a negative effect on the performance of the algorithm due to its reliance on the network, resulting in a longer decision-making time.

Cache pre-loading would have been another useful feature to solve the long scheduling time required during the startup of the system. By pre-calculating optimal cache entries for common situations, the system can potentially switch to these entries with speeds similar to the offline algorithm. However, to allow speeds similar to the offline algorithm in these situations, caching should have been extended to nodes with other roles besides the coordinator as well. However, due to the cache pre-loading not having been implemented, the performance of the online algorithm during start-up remains relatively poor compared to the offline algorithm.

In the node prioritisation phase, nodes should calculate their own priority by an accumulation of the *eagerness*, *suppressor* and *accelerator*. Currently, only the eagerness could be implemented due to a missing interface to network information by the SpaceWireIPC. This limitation makes it impossible to consider the network state when calculating the priority, potentially leading to link bandwidths being exceeded. Although such as situation did not occur during tests, in its current state, there is no mechanism to prevent a task from being scheduled to a node with already high network usage.

The functional approach that was used to test for feasibility, although able to prove feasibility, could not be used to find the optimal algorithm *design*. If the simulated approach had been used instead, several algorithms could have been tested to find the most optimal one. Creating just one design on the target hardware only provides limited tuning capabilities, as testing on the target hardware does not provide a large degree of flexibility. Therefore, using the functional approach, it was not possible to explicitly point out what part of the online algorithm could have been designed better, as only limited control over the target hardware's environment was available.

7.3.2 Test program

The test results in (6) cannot be fully generalisable, as not all parameters of a ScOSA program have been tested for. The test program uses a fixed number of *four* tasks, as explained in (5.2). The number of tasks, however, can increase to a maximum of 200. Even though single-task scheduling might be relatively fast, starting up the system with 200 tasks can take an unacceptable amount of time. Similarly, the network traffic generated to schedule 200 tasks will be very high and may violate the link bandwidth. Even though the algorithm is designed to make the system *adapt* to a changing environment, involving only a limited amount of changes, defining its exact limitations was impossible with the test program that was used.

7.3.3 Test setup

Another limitation was the amount of HPN nodes available to test on. With only 5 HPN nodes available, it was known from the beginning that a test on the target hardware with a large number of nodes was impossible. The virtual nodes that could be used in the server environment provided a solution for the network traffic tests, but these values would not be a correct representation of the total reconfiguration time. A test setup with a larger number of nodes could have provided valuable information on the *temporal scalability* of the system.

The absence of heterogeneity in the test setup is another limiting factor in the generalisability of the results. Heterogeneity is an important strength of ScOSA, and through the diversity of hardware, improved dependability can be achieved. The absence of, for example, an RCN in the test setup means that especially a uniform temporal behaviour with slower hardware in the system cannot be guaranteed.

7.3.4 Network delays

The performance of the decision-making by the online algorithm is known to be affected by network delays. Delays have been observed during the scheduling procedure when waiting for an eagerness value and during a reconfiguration when waiting for a reconfiguration confirmation. This networking information was, however, not stored for the HPN test setup, and therefore no conclusion can be drawn.

7.3.5 Exploratory statistics

In the results, exploratory statistics have been used to analyse and visualise time and network traffic results for exploratory purposes. Although these exploratory statistics are helpful in identifying patterns, their results cannot be used for a generalisable conclusion, as not enough parameters have been tested for. Even though the test program proved to be compliant with the system's constraints, this cannot be guaranteed for a program with a maximum number of tasks.

7.4 Recommendations

7.4.1 Design improvements

Although the algorithm could not be fully analysed using the functional approach, some design improvements could still be identified. The following aspects, as previously discussed, should still be implemented into the online algorithm:

- Caching on all nodes to store dynamic configurations.
- Cache pre-loading, to quickly load commonly occurring situations
- Convergence monitoring when scheduling a task to a node
- Implement an interface to get network information from the SpaceWireIPC
- Implement the suppressor and accelerator calculation

Implementing caching on all nodes, together with cache pre-loading, is not difficult to implement and test. It is recommended that these parts of the design should still be implemented to make the online algorithm more efficient and faster. Fault-tolerance mechanisms such as convergence monitoring will further improve the algorithm's efficiency and reliability. Finally, implementing an interface to retrieve network information is an important addition to the node prioritisation phase of the algorithm. With it currently not being possible to detect if links are about to violate their bandwidth, it is possible that this will happen for a large system at some point. Therefore, the large but necessary addition to retrieving network information through the SpaceWireIPC is also recommended.

7.4.2 Test additional parameters

The test setups used to test for *time* and *network usage* proved useful to test the functionality and feasibility of the algorithm but still left some questions unanswered. It is recommended that additional tests are added to evaluate the algorithm, including:

- Heterogeneity, using a network of HPNs and RCNs
- Increase the number of nodes consisting of the target hardware
- Increase the number of tasks by using an improved test program
- Investigate the presence of network delays
- Evaluate the fault-tolerance of the online algorithm in corner cases
- Evaluate the dependability of the system

An important aspect of ScOSA is the support for heterogeneity through HPNs and RCNs. Heterogeneity, however, has not been included in the test setup. Also, the current number of nodes in the test setup should be expanded to be able to perform tests for the temporal behaviour of the algorithm on a larger system as well. It is therefore recommended that the test setup using the target hardware is expanded by introducing RCNs and increasing the number of nodes to a representable number for a mission. It is also recommended to increase the number of tasks in the test setup and, ideally, make the number of tasks in the test program configurable. By being able to configure the number of tasks, the algorithm can be evaluated for its limitations automatically, possibly even through a continuous integration test. Being able to define, mitigate and overcome the limitations of the online algorithm early on will be beneficial for the dependability of the system.

The delays that were observed and expected to be network-related should be investigated. Although this could be considered outside of the scope of the algorithm, its dependency on the network makes it worth investigating. It should be determined where the delay comes from, either the physical network, SpaceWireIPC, the ScOSA middleware, or the online algorithm. Even though it is unlikely that the online algorithm is causing these delays, this cannot be said with certainty.

Fault-tolerance of the online algorithm is also recommended to be evaluated not just during specific run-time situations but also during corner cases. Evaluating how the algorithm behaves during situations of an unreliable network, for example, can provide insight into the robustness of the scheduling procedure, which, again, impacts the dependability of the system. Finally, the dependability of the online and offline algorithms should be evaluated and compared. Currently, the term dependability remains mostly undefined and is not quantifiable. It is recommended that the tests are to be expanded by defining and testing for a quantifiable metric for dependability, potentially consisting of the *availability* and *reliability* of the system.

Chapter 8

Conclusion

This project aimed to evaluate the feasibility and performance of an online reconfiguration algorithm as a solution to the scalability issues experienced by the ScOSA's offline algorithm.

After determining the specific requirements the online algorithm had to adhere to, a literature research was conducted to find works that aimed to solve a similar problem. It was found that even though there exists a diverse range of online algorithms for distributed systems, none could be applied in ScOSA, mainly due to the absence of fault-tolerant scheduling mechanisms. A functional design approach had been chosen to design a new tailor-made algorithm specifically for ScOSA, as implemented in the ScOSA code-base, to test for feasibility. The novel online algorithm has been evaluated by testing on the target hardware, where it proved its feasibility. Even though the algorithm design could not be fully implemented, it was able to show its potential and can dynamically create a new configuration without any prior knowledge based on the real-time status of the system. With maintainability in mind, alterations to the algorithm can easily be made, for example, to further extend and fine-tune the decision-making process.

The test results showed that instead of increasing memory usage, the online algorithm requires other resources, such as *time* and *network traffic*. For the test program, the online algorithm was able to schedule tasks to nodes without violating the time constraint of a reconfiguration while staying within the network traffic constraint as well. Most importantly, when the number of nodes in the system scales up, the amount of network traffic increases linearly, compared to an exponential increase in memory usage in the offline algorithm. Upon starting the project, it was expected that an increase in time and network usage would take place. This increase, however, was found to be acceptable. Furthermore, its more sophisticated fault-tolerance design can potentially achieve higher dependability than the offline algorithm in its current form. The online algorithm can therefore be used to solve the problem of memory growth in the offline algorithm while removing the reliance on pre-determined configurations.

Bibliography

[1] "DLR software technology ScOSA project," accessed: 06-02-2023. [Online]. Available: https://www.dlr.de/sc/en/desktopdefault.aspx/tabid-11139/19481_read-45210/

[2] "DLR Institute for Software Technology," accessed: 06-02-2023. [Online]. Available: https://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1185/1634_read-3062/

[3] A. Lund, Z. A. Haj Hammadeh, P. Kenny, V. Vishav, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke, "ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture," *CEAS Space Journal*, vol. 14, no. 1, pp. 161–171, Jan. 2022, number: 1. [Online]. Available: https://link.springer.com/10.1007/s12567-021-00371-7

[4] A. D. George and C. M. Wilson, "Onboard Processing With Hybrid and Reconfigurable Computing on Small Satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 458–470, Mar. 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8303008/

[5] "Bae systems plc: Rad750 radiation-hardened powerpc microprocessor," accessed: 03-02-2023. [Online]. Available: https://www.baesystems.com/en-media/uploadFile/20210404045936/1434555668211.pdf

[6] "Leon5 processor," accessed: 03-02-2023. [Online]. Available: https://www.gaisler.com/index.php/products/processors/leon5

[7] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Borner, O. Maibaum, T. Peng, B. Weps, G. Fey, and A. Gerndt, "OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft," pp. 1–13, Mar. 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6836179/

[8] D. Keymeulen, S. Shin, J. Riddley, M. Klimesh, A. Kiely, E. Liggett, P. Sullivan, M. Bernas, H. Ghossemi, G. Flesch, M. Cheng, S. Dolinar, D. Dolman, K. Roth, C. Holyoake, K. Crocker, and A. Smith, "High Performance Space Computing with System-on-Chip Instrument Avionics for Space-based Next Generation Imaging Spectrometers (NGIS)," pp. 33–36, Aug. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8541473/

[9] "Xilinx inc.: Zynq-7000 soc data sheet: Overview." accessed: 06-02-2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview

[10] A. Kovalov, T. Franz, H. Watolla, V. Vishav, and ..., "Model-based reconfiguration planning for a distributed on-board computer," *Proceedings of the 12th . . .*, no. Query date: 2023-01-30 11:31:26, 2020, publisher: dl.acm.org. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3419804.3420266

[11] T. Peng, B. Weps, K. Hoflinger, K. Borchers, D. Lüdtke, and A. Gerndt, "A new SpaceWire protocol for reconfigurable distributed on-board computers: SpaceWire networks and protocols, long paper," pp. 1–8, Oct. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7771624/

[12] C. Treudler, H. Benninghof, K. Borchers, B. Brunner, J. Cremer, M. Dumke, T. Gartner, K. Hoflinger, J. Langwald, D. Lüdtke, T. Peng, E.-A. Risse, K. Schwenk, M. Stelzer, M. Ulmer, S. Vellas, K. Westerdorff, and F. Lastname, "ScOSA - Scalable On-Board Computing for Space Avionics," Oct. 2018.

[13] F. Greif, M. Bassam, J. Sommer, N. Toth, J.-G. Mess, A. Ofenloch, R. Rinaldo, M. Goeksu, B. Weps, O. Maibaum, J. Reinking, and M. Ulmer, "Outpost," 2018. [Online]. Available: https://github.com/DLR-RY/outpost-core/

[14] Z. A. H. Hammadeh, T. Franz, O. Maibaum, A. Gerndt, and D. Lüdtke, "Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems," pp. 29–34, 07 2019. [Online]. Available: https://elib.dlr.de/128249/

[15] A. Kovalov, E. Lobe, A. Gerndt, and D. Lüdtke, "Task-Node Mapping in an Arbitrary Computer Network Using SMT Solver," in *Integrated Formal Methods*, N. Polikarpova and S. Schneider, Eds. Cham: Springer International Publishing, 2017, vol. 10510, pp. 177–191, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-66845-1_12

[16] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. R. Ramakrishnan, and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4963, pp. 337–340, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-78800-3_24

[17] L. Zohrati, M. Abadeh, and E. Kazemi, "Flexible approach to schedule tasks in cloud-computing environments," *Iet Software*, 2018, tex.litmapsid: 187455458.

[18] K. Karmakar, R. K. Das, and S. Khatua, "Resource scheduling for tasks of a workflow in cloud environment," *Lecture Notes in Computer Science*, 2020, tex.litmapsid: 22437599.

[19] W. Zheng, Z. Chen, R. Sakellariou, L. Tang, and J. Chen, “Evaluating DAG scheduling algorithms for maximum parallelism,” *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2020, tex.litmapsid: 60716157.

[20] L. Liu, G. Xie, L. Yang, and R. Li, “Schedule dynamic multiple parallel jobs with precedence-constrained tasks on heterogeneous distributed computing systems,” 2015, tex.litmapsid: 145440219.

[21] R. M. Sahoo and S. K. Padhy, “A novel algorithm for priority-based task scheduling on a multiprocessor heterogeneous system,” 2022, tex.litmapsid: 249868526.

[22] B. Hu, Z. Cao, and L. Zhou, “Adaptive real-time scheduling of dynamic multiple-criticality applications on heterogeneous distributed computing systems,” 2019, tex.litmapsid: 205375916.

[23] L.-C. Canon, L. Marchal, B. Simon, and F. Vivien, “Online scheduling of task graphs on hybrid platforms,” *Lecture Notes in Computer Science*, 2018, tex.litmapsid: 208601358.

[24] M. N. Krishnan and R. Thiyagarajan, “Multi-objective task scheduling in fog computing using improved gaining sharing knowledge based algorithm,” *Concurrency and Computation: Practice and Experience*, 2022, tex.litmapsid: 231884464.

[25] M. Chatterjee and S. K. Setua, “A multi-objective deadline-constrained task scheduling algorithm with guaranteed performance in load balancing on heterogeneous networks,” *SN computer science*, 2021, tex.litmapsid: 97081173.

[26] L. Xu, J. Qiao, S. Lin, and W. Zhang, “Dynamic task scheduling algorithm with deadline constraint in heterogeneous volunteer computing platforms,” *Future Internet*, 2019, tex.litmapsid: 122783149.

[27] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, “I-Scheduler: Iterative scheduling for distributed stream processing systems,” *Future Generation Computer Systems*, 2021, tex.litmapsid: 206997489.

[28] S. Ahmad, C. S. Liew, E. U. Munir, T. F. Ang, and S. U. Khan, “A hybrid genetic algorithm for optimization of scheduling workflow applications in heterogeneous computing systems,” *Journal of Parallel and Distributed Computing*, 2016, tex.litmapsid: 26746422.

[29] J. Mei, K. Li, X. Zhou, and K. Li, “Fault-tolerant dynamic rescheduling for heterogeneous computing systems,” *Journal of Grid Computing*, 2015, tex.litmapsid: 81973110.

[30] D. Feng, B. Liu, and J. Gong, “An on-board task scheduling method based on evolutionary optimization algorithm,” 2022, tex.litmapsid: 251743825.

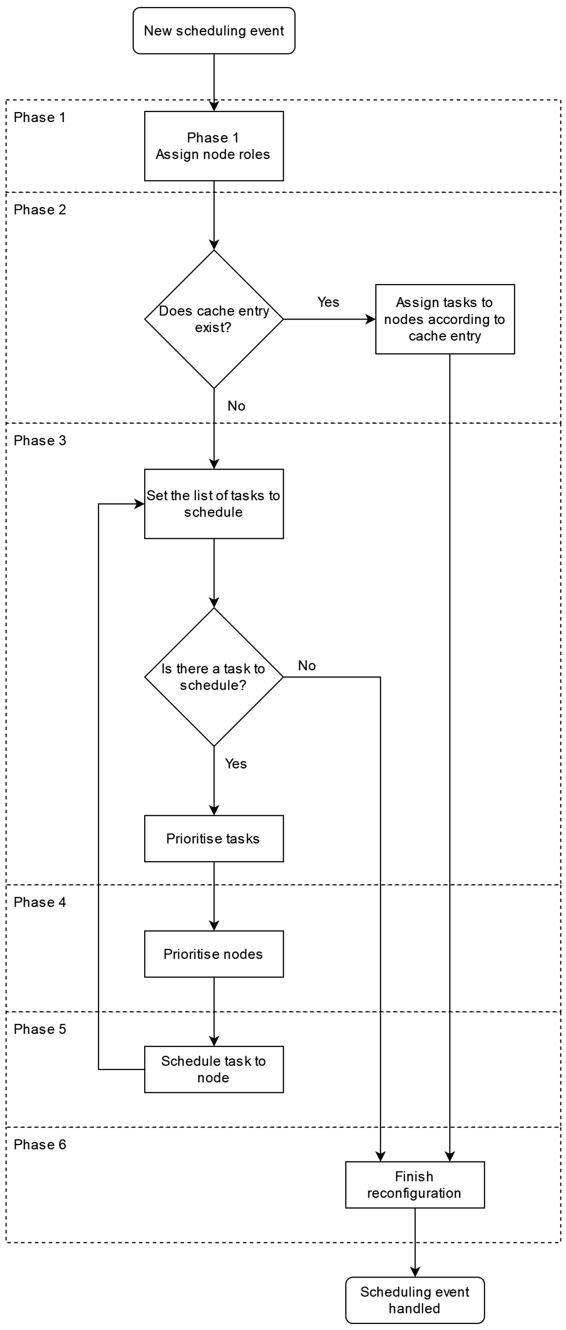
[31] “ECSS-E-ST-50-12C.” Noordwijk, The Netherlands: European Cooperation for Space Standardization: SpaceWire - Links, nodes, routers and networks, 2008.

[32] S. Seth and N. Singh, “Dynamic heterogeneous shortest job first (DHSJF): a task scheduling approach for heterogeneous cloud computing systems,” *International Journal of Information Technology*, 2019, tex.litmapsid: 86563086.

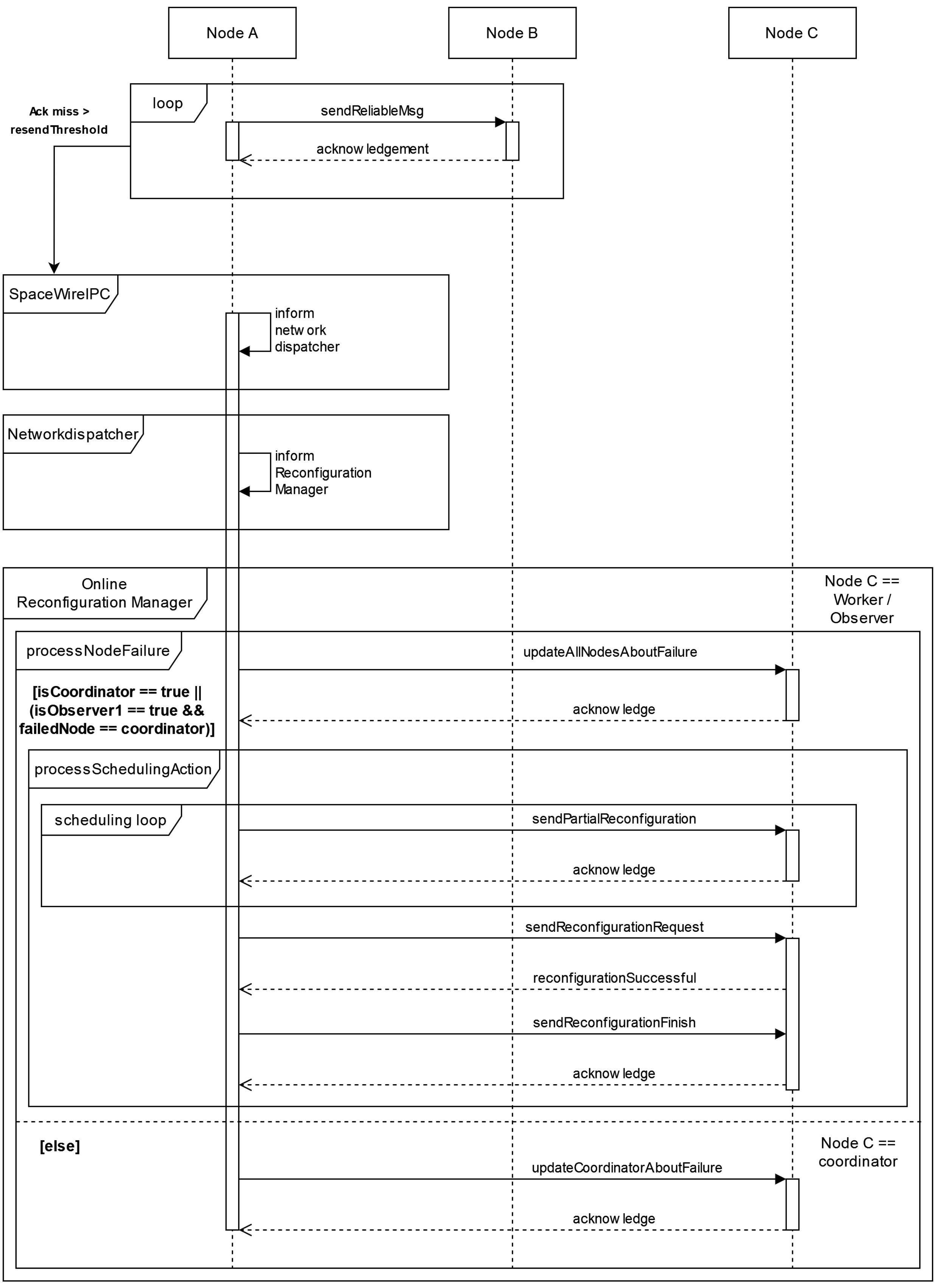
Appendix A

Software diagrams

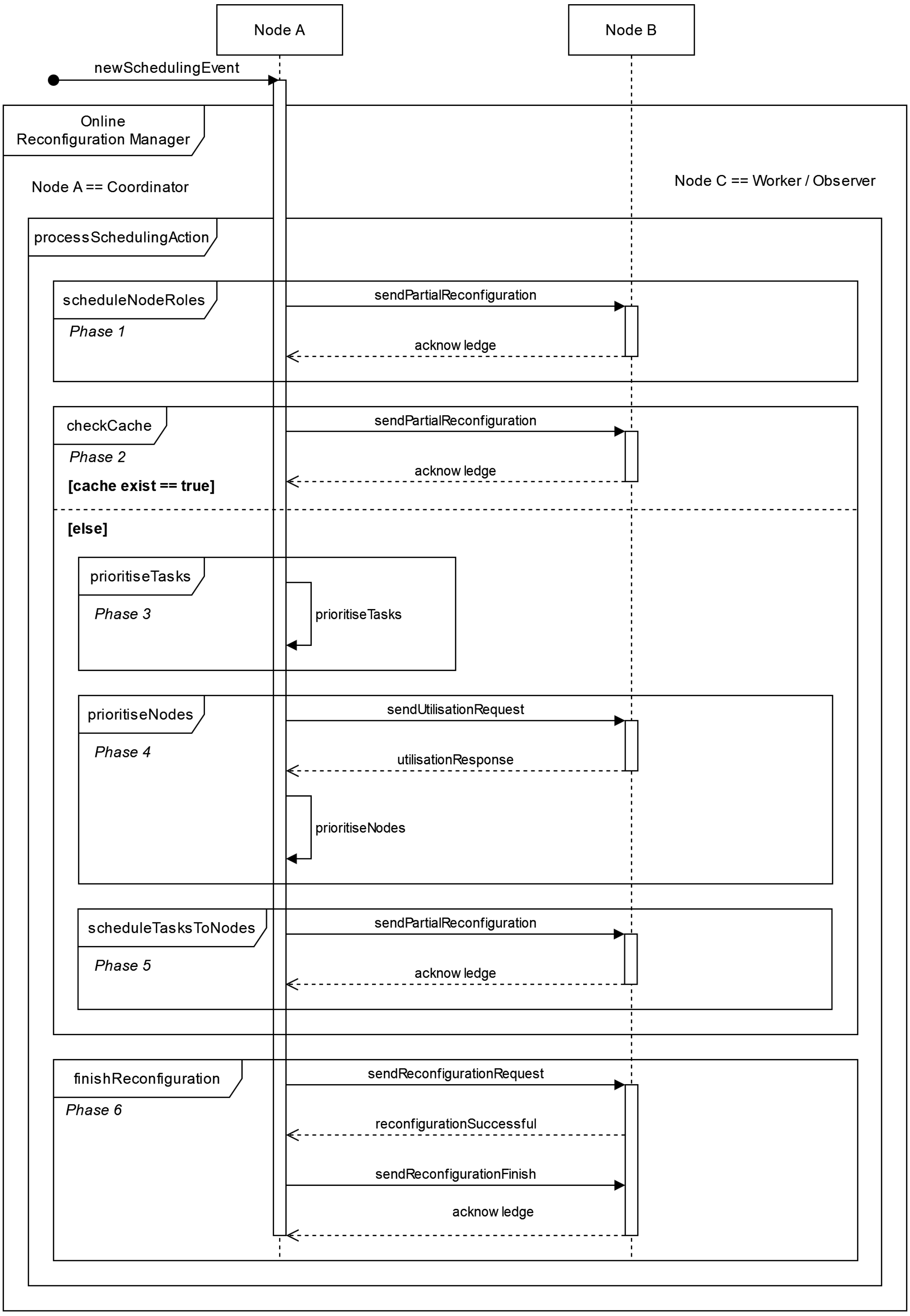
A.1 Scheduling phase flow chart



A.2 Node Failure handling sequence diagram


Node A
Node B
Node C
loop
Ack miss > resendThreshold
sendReliableMsg
acknowledgement
SpaceWireIPC
inform network dispatcher
Networkdispatcher
inform Reconfiguration Manager
Online Reconfiguration Manager
Node C == Worker / Observer
processNodeFailure
updateAllNodesAboutFailure
[isCoordinator == true || (isObserver1 == true && failedNode == coordinator)]
acknowledge
processSchedulingAction
scheduling loop
sendPartialReconfiguration
acknowledge
sendReconfigurationRequest
reconfigurationSuccessful
sendReconfigurationFinish
acknowledge
[else]
Node C == coordinator
updateCoordinatorAboutFailure
acknowledge

A.3 Scheduling procedure sequence diagram



Appendix B

Scripts

B.1 HPN testing script

```bash
#!/bin/bash
echo "Integrating node"

while true; do
    t="$(shuf -i 30-120 -n 1)"
    echo "Node failure in $t seconds"
    timeout "$t"s ./threeNodesExample log.txt info $1 $2 $3 $4
    t="$(shuf -i 20-40 -n 1)"
    echo "Reintegrate after $t seconds"
    sleep "$t"s
done
```

Appendix C

Traces

C.1 Coordinator failure trace

```
[system_management::OnlineReconfigurationManager::processNodeFailure@140] Node 4 became unavailable
[system_management::OnlineReconfigurationManager::processNodeFailure@143] Node 4 was the coordinator!
[system_management::OnlineReconfigurationManager::processNodeFailure@152] Heartbeat from node 4 was lost, handling error.
[system_management::OnlineReconfigurationManager::scheduleNodeRoles@536] No COORDINATOR in the system, appointing new coordinator
[system_management::OnlineReconfigurationManager::scheduleNodeRoles@547] New COORDINATOR appointed with nodeId: 2
[system_management::OnlineReconfigurationManager::scheduleNodeRoles@556] No OBSERVER_1 in the system, appointing new OBSERVER_1
[system_management::OnlineReconfigurationManager::scheduleNodeRoles@581] New OBSERVER_1 appointed with nodeId: 3
[spacewire_ipc::SpaceWireIPC::handleOutEvent@2232] Type 6 exceed resendthreshold, desNode = 4
[spacewire_ipc::SpaceWireIPC::handleNormalExceed@2166] errorReason = 3, failed node = 4
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 2 has an eagerness of: 9.48971
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 3 has an eagerness of: 9.32108
[system_management::OnlineReconfigurationManager::scheduleTasksToNodes@822] Scheduling task 100 to node: 3
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 2 has an eagerness of: 9.48975
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 3 has an eagerness of: 9.32154
[system_management::OnlineReconfigurationManager::scheduleTasksToNodes@822] Scheduling task 103 to node: 3
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 2 has an eagerness of: 9.48984
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 3 has an eagerness of: 9.32218
[system_management::OnlineReconfigurationManager::scheduleTasksToNodes@822] Scheduling task 102 to node: 3
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 2 has an eagerness of: 9.48988
[system_management::OnlineReconfigurationManager::prioritiseNodes@785] Node: 3 has an eagerness of: 9.32382
[system_management::OnlineReconfigurationManager::scheduleTasksToNodes@822] Scheduling task 101 to node: 3
[system_management::OnlineReconfigurationManager::processSchedulingAction@510] Decision making deltaTime is 433ms.
[system_management::ReconfigurationService::FPGAReconfigurationService@505] Error reading bit stream
[system_management::ReconfigurationService::processReconfigurationFinish@348] Reconfiguration FINISH received from the coordinator.
[system_management::SystemConfiguration::switchToNextConfiguration@106] nodeConfigId is set to -1
[system_management::ReconfigurationService::processReconfigurationFinish@373] RECONFIGURATION FINISHED AT 1670509388440
[system_management::ReconfigurationService::processReconfigurationFinish@380] Reconfiguration deltaTime is 103ms.
```

C.2 Node failure cache load trace

```
[system_management::OnlineReconfigurationManager::processNodeFailure@140] Node 3 became unavailable
[system_management::OnlineReconfigurationManager::processNodeFailure@152] Heartbeat from node 3 was lost, handling error.
[system_management::OnlineReconfigurationManager::scheduleNodeRoles@556] No OBSERVER_1 in the system, appointing new OBSERVER_1
[system_management::OnlineReconfigurationManager::scheduleNodeRoles@581] New OBSERVER_1 appointed with nodeId: 4
[system_management::OnlineReconfigurationManager::checkCache@703] CACHE entry found
[system_management::ReconfigurationService::FPGAReconfigurationService@505] Error reading bit stream
[system_management::OnlineReconfigurationManager::processSchedulingAction@505] CACHED Decision making deltaTime is 6ms.
[system_management::ReconfigurationService::processReconfigurationFinish@348] Reconfiguration FINISH received from the coordinator.
[system_management::SystemConfiguration::switchToNextConfiguration@106] nodeConfigId is set to -1
[system_management::ReconfigurationService::processReconfigurationFinish@373] RECONFIGURATION FINISHED AT 1670509209018
[system_management::ReconfigurationService::processReconfigurationFinish@380] Reconfiguration deltaTime is 84ms.
```