HAW HAMBURG

BACHELORTHESIS
Kristi Kola

Design and Implementation of a Reliable Multicast Using the OUTPOST Software Bus for Embedded Spaceflight Software

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

HAMBURG UNIVERSITY
OF APPLIED SCIENCES

Hochschule für Angewandte
Wissenschaften Hamburg

Kristi Kola

Design and Implementation of a Reliable Multicast Using the OUTPOST Software Bus for Embedded Spaceflight Software

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Paweł Buczek
Second examiner: M.Sc. Jan-Gerd Meß

Day of delivery: 31 March 2022

Kristi Kola

Title of Thesis

Design and Implementation of a Reliable Multicast Using the OUTPOST Software Bus for Embedded Spaceflight Software

Keywords

OUTPOST Software Bus, Library, Moduls, Embedded Systems, Multicast, Reliability, Synchronous Link, Synchronous Communication, Sender, Receiver, Bus Channel

Abstract

This thesis is part of the *OUTPOST (Open modUlar sofTware PlatfOrm for SpacecrafT)* library of the German Aerospace Center (DLR), which provided the author an opportunity to design and implement a feature for one of their libraries- *OUTPOST*. The *OUTPOST* library is developed at the German Aerospace Center (DLR) and provides an execution platform targeted at embedded systems running mission-critical software. The library is set up to be modular, allowing the user to choose which modules to use and which modules to leave out. The goal of this thesis is to introduce a synchronous link feature called *SyncLink* for the *Software Bus* module of the *OUTPOST* library that allows for a reliable and synchronous multicast of messages from one or more senders to one or more receivers. Potentially, this work can result in the use of the *OUTPOST* library for multicasting and synchronous communication in space applications from satellites to communication between components inside a spacecraft.

Kristi Kola

Thema der Arbeit

Design und Implementierung eines zuverlässigen Multicasts mit dem OUTPOST - Software Bus für Embedded Spaceflight Software

Stichworte

OUTPOST Software Bus, Bibliothek, Modul, Eingebettete Systeme, Multicast, Zuverlässigkeit, Synchrone Verknüpfung, Synchrone Kommunikation, Sender, Empfänger, Bus Channel

Kurzzusammenfassung

Diese Arbeit ist Teil der Bibliothek *OUTPOST (Open modUlar sofTware PlatfOrm for SpacecrafT)* des Deutschen Zentrums für Luft- und Raumfahrt (DLR), die dem Autor die Gelegenheit bot, ein Feature für eine seiner Bibliotheken zu entwerfen und zu implementieren - *VORPOSTEN*. Die *OUTPOST*-Bibliothek wird am Deutschen Zentrum für Luft- und Raumfahrt (DLR) entwickelt und bietet eine Ausführungsplattform, die auf eingebettete Systeme ausgerichtet ist, auf denen unternehmenskritische Software ausgeführt wird. Die Bibliothek ist modular aufgebaut, sodass der Benutzer auswählen kann, welche Module verwendet und welche weggelassen werden sollen. Das Ziel dieser Diplomarbeit ist die Einführung eines synchronen Link-Features namens *SyncLink* für das *Software Bus*-Modul der *OUTPOST*-Bibliothek, das ein zuverlässiges und synchrones Multicasting von Nachrichten von einem oder mehreren Sendern ermöglicht an einen oder mehrere Empfänger. Potenziell kann diese Arbeit zur Verwendung der *OUTPOST*-Bibliothek für Multicasting und synchrone Kommunikation in Weltraumanwendungen von Satelliten bis zur Kommunikation zwischen Komponenten innerhalb eines Raumfahrzeugs führen.

Acknowledgements

I would like to thank my supervisors and mentors Prof. Dr. Buczek Pawel from the Faculty of Engineering and Computer Science, Hamburg University of Applied Sciences and M.Sc. Jan-Gerd Meß from the German Aerospace Center (DLR). Their guidance, expertise and support were crucial to the completion of this thesis.

Contents

List of Figures

List of Tables

1 Introduction

This section highlights the motivation behind the selection of the bachelor thesis topic and the relevance it presents to the "Deutsches Zentrum für Luft- und Raumfahrt e.V.", in this paper interchangeably referred to as DLR.

1.1 Motivation

The paper's topic was decided in a mutual agreement between the author and the main supervisor after the propositions of the second supervisor on the relevant research topics that can be concluded inside the scope of a bachelor thesis.

From lunch, to reaching the outer space, spaceflight systems need to operate efficiently in a challenging environment with some of the most extreme conditions known to humankind. In order to successfully reach the goals of their missions, many of these complex hardware and software components transmit their data through multiple communication paradigms that accommodate for the general resource - constraint embedded nature of these systems.

A typical communication paradigm takes into account aspects such as memory, speed, accuracy, availability, power consumption etc., and not all aspects of communication can be efficiently achieved by one single communication paradigm, therefore a trade-off needs to be made especially when it comes to speed and reliability.

The German Aerospace Center (DLR) developed the *Software Bus* (SWB) module of the *OUTPOST* library, which allows for asynchronous exchange of messages between senders and receivers. Senders can send a message from their own memory pool and receivers can register channels that filter for certain types of messages using their id

and/or data as filtering criteria. This module is designed for a loosely-coupled asynchronous communication that offers a flexible and versatile means of communication.

In its current state, the Software Bus cannot fulfill the needs for a synchronous type of communication, meaning that there is no way for a sender to know if the receivers registered on a *Bus Channel* have received the data sent through the *Software Bus*, or if they have failed to do so. Additionally, the *Software Bus* does not allow for a *Reliable Multicast* of messages.

1.2 Objectives

The objective of this thesis paper is to design and implement a reliable multicast communication to DLR's OUTPOST Software Bus Module. This feature should be able to provide senders with a synchronous and reliable means of communication to the receivers, and also allow for the senders to choose the receivers for specific messages. This feature is to be referred as *SyncLink* (short for synchronous link).

1.3 Thesis Structure

The remainder of this paper is structured as follows:

- The next chapter 2 provides a short background on the institution, the *OUTPOST* library architecture, the *Software Bus (SWB)* module, some of the missions where the *OUTPOST* library was and is going to be used, and the communication paradigms of a synchronous communication and reliable multicast of messages.
- Chapter 3 focuses on an overview of the stakeholders, requirements analysis, use cases and introduces the *Synchronous Link* (or short SyncLink) feature.
- Chapter 4 lays down the conceptual design and the architecture of the *SyncLink*.
- Chapter 5 describes the implementation of *SyncLink*.

- Chapter 6 describes the process of testing, the results and evaluation of the usage of *SyncLink* on a modified application example, as well as the space for future extensions.
- The work is concluded on chapter 7.

2 Background

This chapter provides a small background on the *DLR* and gives insight into the *OUTPOST* library, the *OUTPOST - Software Bus* Module as well as some missions where the *OUTPOST* library was and will be used. Finally, a theoretical background on the communications paradigms and the concept of reliability is provided.

2.1 Deutsches Zentrum für Luft- und Raumfahrt e.V.

The Deutsches Zentrum für Luft- und Raumfahrt e.V., commonly known as the DLR is the Federal Republic of Germany's research center for aeronautics and space. The DLR conducts research and development activities in the fields of aeronautics, space, energy, transport, security, and digitalization. The German Space Agency plans and implements the national space program on behalf of the federal government.[4]. DLR's space research contributes towards addressing societal challenges such as climate change, secure communications, health, and demographic change. Its research and development work covers all areas of technology and applications in spaceflight.

In the spaceflight domain, DLR develops spaceflight infrastructures and technologies, and its activities range from the development of new engines to the development and use of satellites and spacecraft. It is also working on new communications and navigation technologies. The aim of its research is to generate knowledge and technologies in the fields of climate research, environmental monitoring, and disaster management.

The Institute of Space Systems located in Bremen designs and analyzes future spacecraft and space missions (launch systems, orbital and exploration systems, satellites) and evaluates them with regard to their technical performance and costs. The Avionics department does research and development in the avionics field, especially on spaceflight applications. [5]

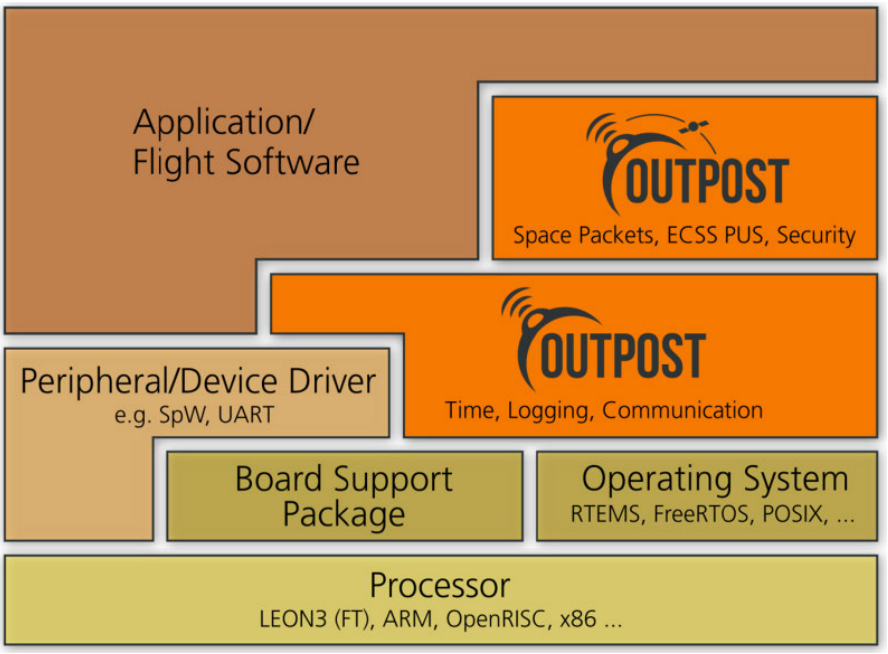

Figure 2.1: DLR Flight Software Stack [7]

2.2 OUTPOST

The *OUTPOST* library (Open modUlar sofTware PlatfOrm for SpacecrafT) is a library developed by the DLR to promote an open-access, modular-based platform for research and development activities.

2.2.1 OUTPOST Architecture

The target environment of OUTPOST is the on-board computer of a spacecraft, which consists of a microcontroller. Supported processors include the LEON3FT SPARC V8 processor or the ARM Cortex M1-4 processors. As shown in Figure 2.1, the library is located between the underlying OS and the board support package (BSP) on the one hand and the applications on the other hand. It is comprised of the OUTPOST - core (time, logging, communication) and OUTPOST - satellite (space packets, ECSS PUS,

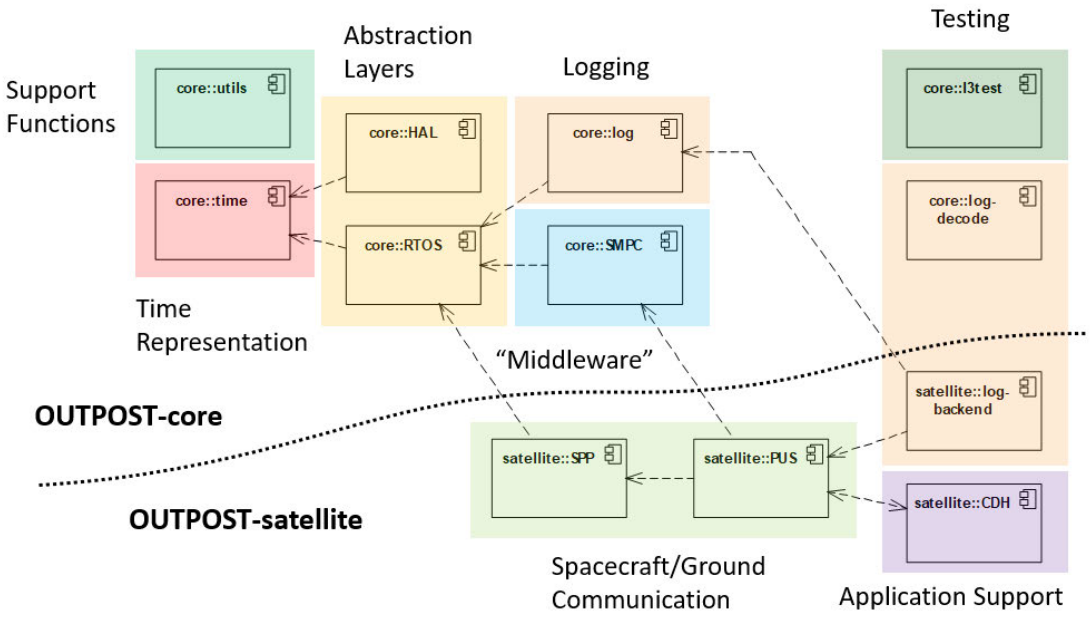

Figure 2.2: OUTPOST-core and OUTPOST-satellite modules. [7]

security) libraries as shown in figure 2.2. To keep the software applications independent from the embedded RTOS used and facilitate later re-use, the software does not access the OS directly, but through a thin abstraction layer. This RTOS layer provides C++-style access to e.g. the synchronization mechanisms like mutexes and semaphores.[8] Currently, the RTOS layer is ported upon the following Operating Systems:

1. RTEMS
2. FreeRTOS
3. POSIX compatible (e.g. Linux)

The implementation is selected at link/compile time. The setup, initialization, and resource management of the corresponding OS are beyond the scope of this library and have to be done by the user itself during the system initialization[8].

The utils module contains support functions such as shared buffers which are containers used to store data in data structures such as arrays, fixed-size arrays, maps, lists, or views/slices. The time module contains secure time management functions and provides classes for converting between different time representations such as Spacecraft Elapsed

Time (SCET), GPS Time, TAI Time, and Unix Time (leap-second correction). The hardware abstraction layer for communication interfaces resides in the HAL module, which focuses on communication interfaces such as SpaceWire, Serial (UART), Ethernet, and platform-specific hardware drivers located in platform repositories like LEON and Zynq (ARM)[7].

The RTOS wrapper layer allows OUTPOST to build on different operating systems. The standard C++ RTOS interface is used for functions such as the Clock, Thread, Mutex, Semaphore, Periodic Task, Timer, Queue, Failure Handler, and Checkpoints. The Simple Message Passing Channel (SMPC) module offers a simple and fast public/subscriber-based communication middleware for objects located in the same address space. The l3test module contains a Lua-based test framework that enables the execution of Lua scripts as part of unit tests[7]. Figure 2.2 is not up-to-date as further modules were added to the OUTPOST libraries, therefore the SIP and SWB modules are missing from the figure. A further explanation for this module is given in section 2.2.2. At the time this paper was written the public outpost-core repository is also not up-to-date with these changes.

Some of the architectural goals of the OUTPOST library in whole and the Software Bus module in specific are type safety, testability, portability and resource efficiency. Type safety is achieved by not using any void pointers or raw values, testability is achieved with unit testing coverage being above 90% and separation of functional logic and timing, portability by running on RTEMS, FreeRTOS and POSIX, and finally resource efficiency by avoiding unnecessary copying.

2.2.2 OUTPOST Software Bus

The Software Bus (SWB) is a module of the OUTPOST library which allows for asynchronous exchange of messages between senders and receivers. Senders can send messages from their own memory pool or use the bus's pool. Receivers can register channels that filter for certain types of messages using their id and/ or data. Housekeeping is also provided through a number of counters. The Software Bus Module is built following the Publisher-Subscribe pattern [1]. As such, it supports loose coupling between components and provides scalability. The implementation is efficient as it allows for a direct func-

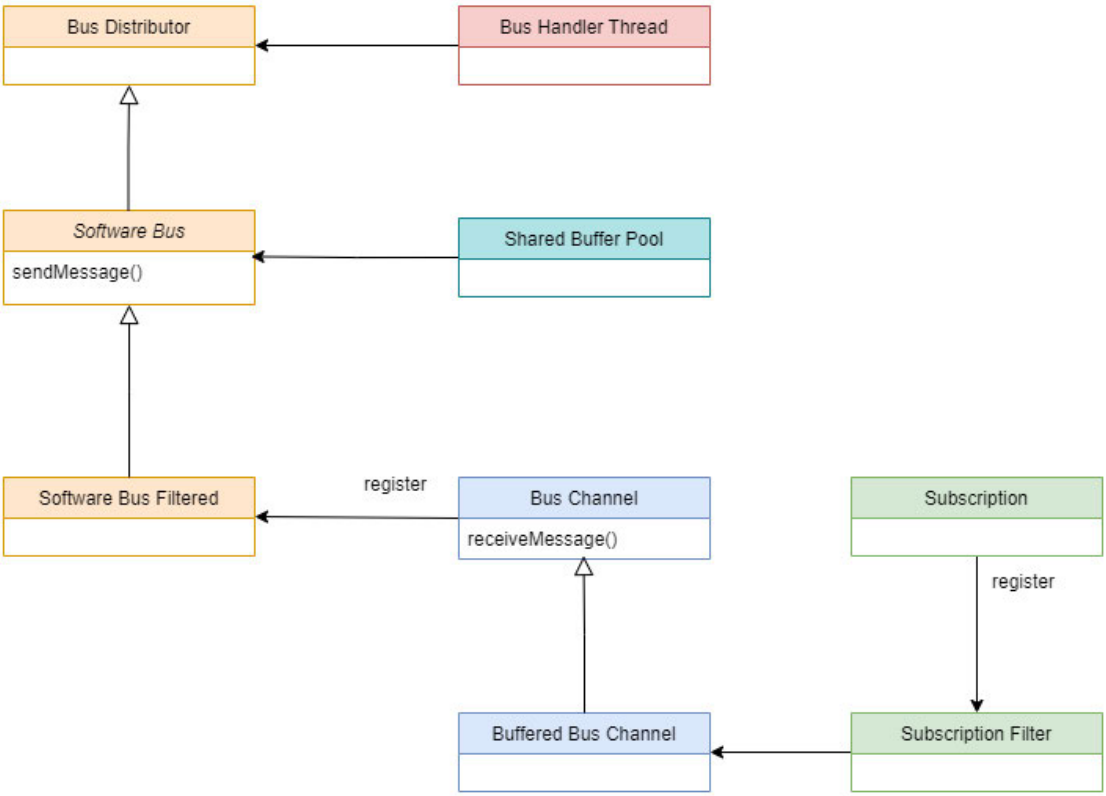

Figure 2.3: Class Diagram - OUTPOST swb module.

tion call. Type safety is guaranteed by safely matching the topic type to the function parameter type. The Software Bus has the following parameters:

1. pool: Shared Buffer Pool from which the Software Bus shall allocate its memory.
2. queue: Reference Queue Base to which the Message will be sent. For internal use only.
3. priority: Priority of the internal Bus Handler Thread.
4. heartbeatSource: Heartbeat Source for the software watchdog.

Per definition, a Message is the fundamental data structure that is passed around in the Software Bus, and it consists of an individual ID, which can be used for filtering the messages, and of a Buffer which consists of the data that is transferred.

In Figure 2.3 the main architecture of the Software Bus is shown through a UML - Class Diagram. The classes functionalities are described as follows:

1. The Bus Distributor holds the underlying thread that handles passing around of messages as well as a list if registered channels. It is not meant to be used directly but through the SoftwareBus by inheritance.
2. The Bus Handler Thread handles incoming Messages and distributes them to registered BusChannels.
3. Shared Buffer Pools are pre-allocated memory blocks (cf. SharedBufferPool). They are accessed by SharedBufferPointers, which as long as they are referenced can be passed around the system without the need to ever copy their contents. This is especially useful for applications that push around big chunks of data.
4. Software Bus Filtered is a variant of the SoftwareBus that uses in incoming filter to limit the type of Message that is accepted for distribution. It consists of parameters ID and filter.
5. The Bus Channel interface is used to receive messages from the SoftwareBus. A SoftwareBus will distribute messages to registered BusChannels according to their filtering.

6. Buffered Bus Channel is a variant of the Bus Channel with an additional Filter parameter that decides whether the channel wants a message and a buffer in order to store messages.

7. Bus Subscription can be used together with a SubscriptionFilter on a BusChannel to filter by (masked) Message id.

8. Subscription Filter uses Subscriptions for Filtering.

In figure 2.4 an asynchronous communication between a Sender and the Receivers using the OUTPOST - Software Bus is shown. The Sender sends a message to the Software Bus by calling a public method named sendMessage(). The message should consist of a message ID and the Data that needs to be transmitted. The ID of the message could be used by the Software Bus or the Bus Channels to filter messages depending on the needs of the application it is being used. In case the Software Bus uses a "Copy Once" method, the message will be stored in the SharedBufferPool, otherwise, in case of a "Zero Copy", the Sender will use its own memory pool to send messages.

The message is then transmitted to the Distributor, which holds the underlying BusHandlerThread. Using this thread, the Distributor will send the message to the registered Bus Channels. According to their filtering settings, there could be one, many, or all registered Bus Channels that can receive the message.

On the other side, one or many Receivers will from time to time request to receive a message from the Bus Channels. If there is a message, the return of the function will deliver the message to them, otherwise, if no message is available they will have to try again at a later time.

In the current asynchronous type of communication that the Software Bus is based upon, there is no way for the sender to know if any of the receivers have received the message or has failed to do so, making it unreliable for applications where there is a need for a confirmation of delivery.

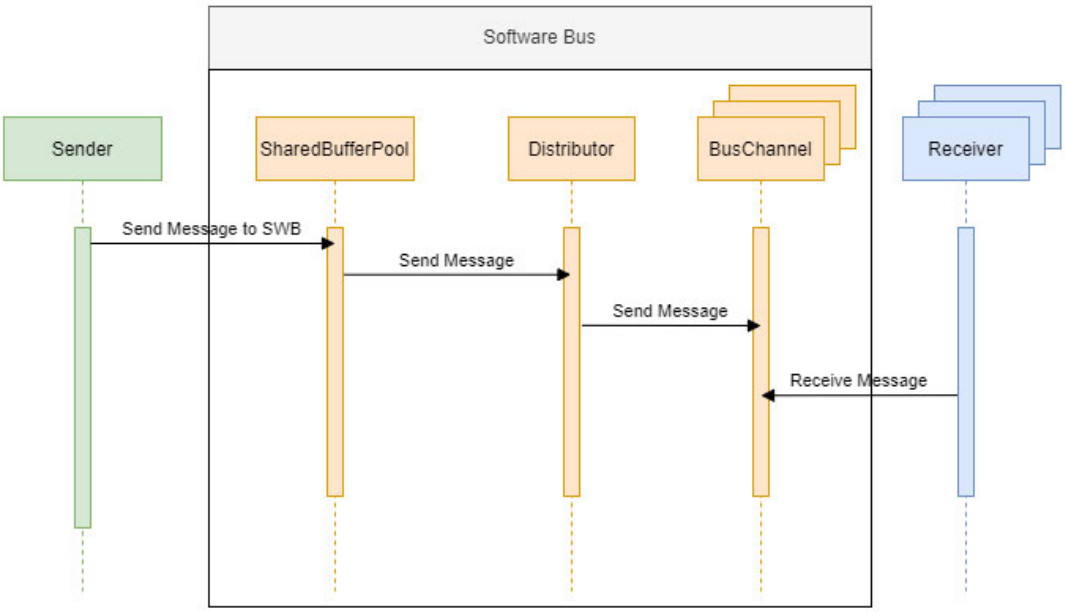

Figure 2.4: Asynchronous communication using the OUTPOST - Software Bus

2.2.3 Current Usage

The OUTPOST library was and will be used by several projects by the DLR. A few of them are:

1. The Eu:CROPIS satellite mission (figure 2.5) hosted a biological experiment from the National Aeronautics and Space Administration (NASA), testing photosynthetic cyanobacteria for the production of food for non-photosynthetic microbes. The application of this experiment is to be seen in the provision of a biological source of energy for future Space colonies.

2. The Martian Moons Exploration is a mission to travel to Mars and survey its two moons named Phobos and Deimos. The spacecraft will explore both moons, it will collect a sample from Phobos surface and will bring it back to Earth. The launch of MMX is planned for 2024, and the spacecraft will also carry a rover which is shown in figure 2.6. The project is a collaboration between the Japanese (JAXA), French (CNES) and German (DLR) Space Agencies. The DLR is responsible for the development of the rover housing, the robotic locomotion system as well as

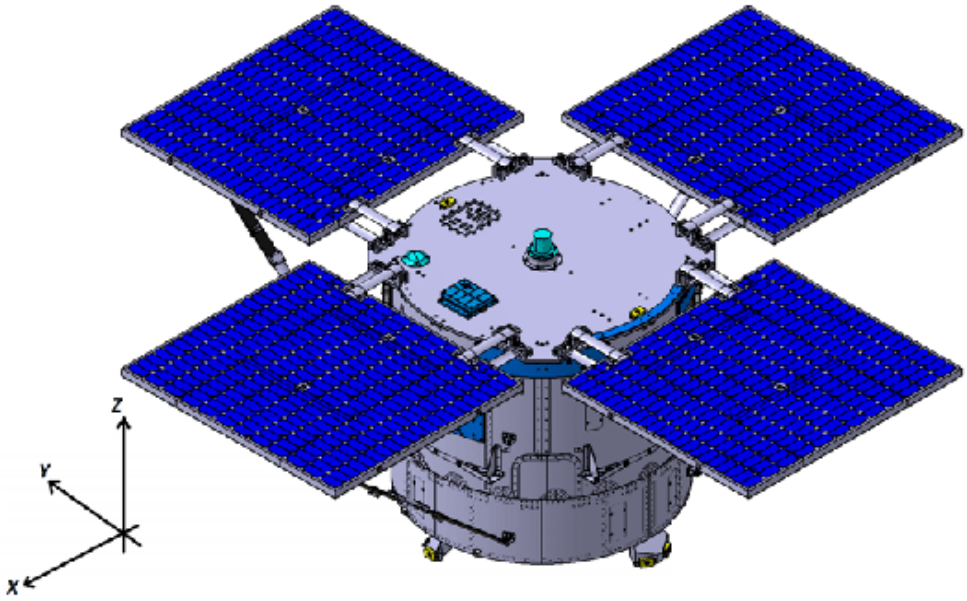

Figure 2.5: DLR EU-Cropis Compact Satellite [3]

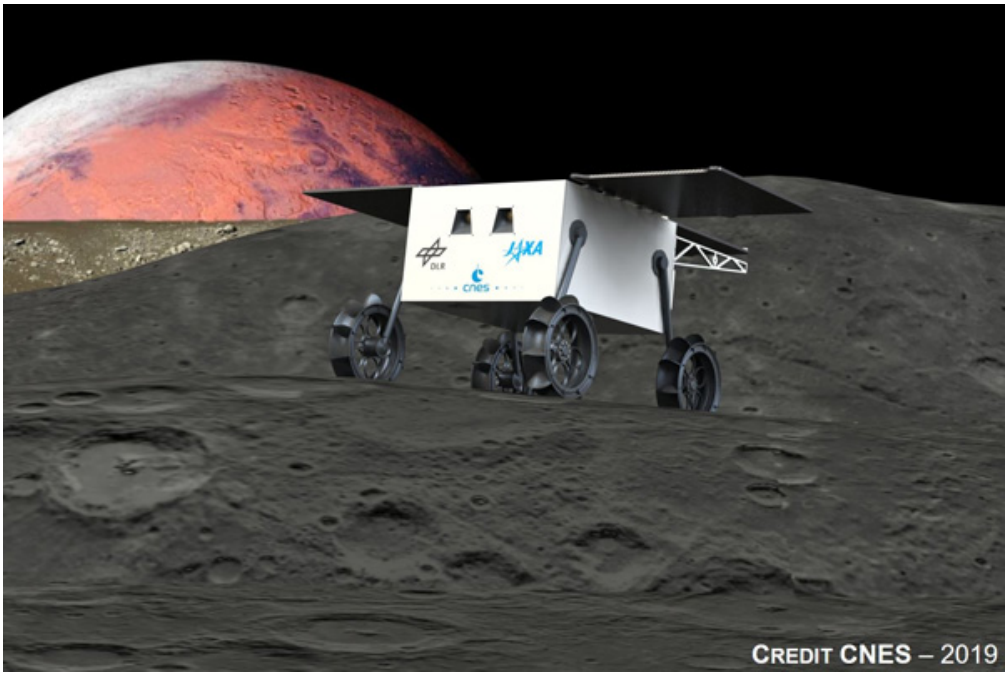

Figure 2.6: Martian Moons Exploration (MMX) Rover[6]

a spectrometer and a radiometer, each of which will measure surface composition and texture. [6]

3. SCORE: Payload 4 is a technology demonstrator for next-generation on-board computing in hardware and software. SCORE is complemented by a set of three digital cameras that are commanded via SCORE. [2]

2.2.4 Housekeeping

The housekeeping mechanism is an application that uses the OUTPOST - Software Bus module for communication, developed by the DLR's Avionics Group. It is made from a single controller that is used for collecting status and environmental data throughout the spacecraft. Housekeeping is a typical mechanism on board of a spacecraft, therefore it was found suitable to be used as an application example for the OUTPOST - Software Bus module.

2.3 Communication paradigms

2.3.1 Asynchronous and Synchronous Communication

An asynchronous type of communication is a simple form of communication that allows for the delivery of messages from point A to point B, as showcased in figure 2.7 where the Sender actor initiates the communication by sending a message to a Receiver actor. This form of communication is fast in execution due to the one direction of messages, as in most scenarios it does not use clocks or blocking/busy-wait methods. Nevertheless, in the asynchronous communication paradigm, the sender will not know if the message has been successfully transmitted or if it has been lost from any hardware or software failure on the receiver's side.

In figure 2.8 a sequence diagram illustrating the basic synchronous communication process used in one embodiment for receiving and processing acknowledgement messages is displayed. The process begins with the Sender actor sending a message and proceeds to the receiving of the message from the Receiver actor, who in turn processes the message it received and generates an acknowledgement message, which is then delivered to the original sender. Many synchronous communication applications use clocks or

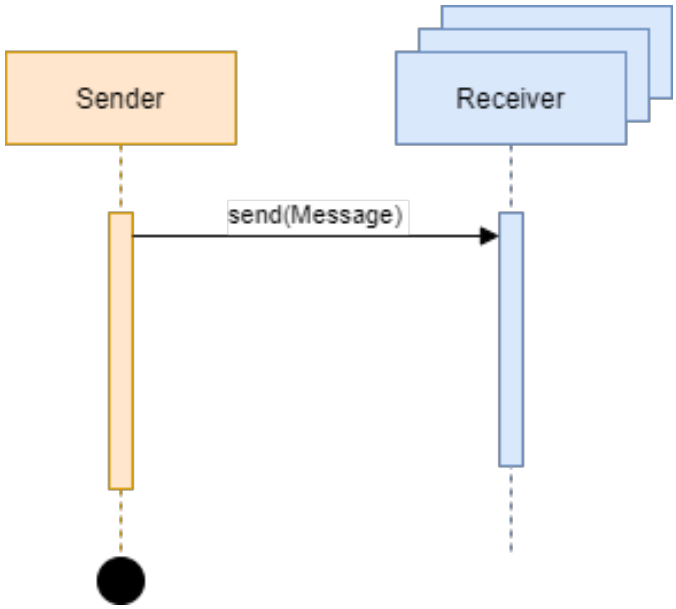

Figure 2.7: Communication using the asynchronous communication paradigm

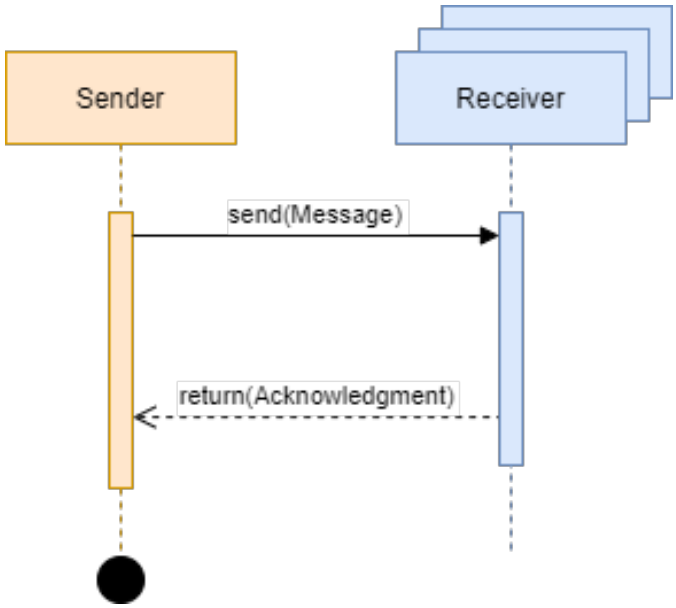

Figure 2.8: Communication using the synchronous communication paradigm

blocking/busy-wait methods, which compared to the asynchronous design, trades speed and simplicity for additional information and complexity. [12]

In conclusion, a synchronous communication although it can provide additional information from the receiver, might not be suitable for simple and straightforward communications that rely on speed, and the choice between those paradigms is made on a need basis.

2.3.2 Broadcast and Multicast Communication

Broadcast networks are one of the most common forms of communication and are broadly found in local-area communication systems. The data and messages transmitted by such a network are available to all the recipients who are able to receive messages. However, it is a very common problem for this kind of network to lose a message from multiple types of transmission errors, including buffer overflow errors within the receivers. Compared to point-to-point communications where protocols are implemented to recover the lost messages, in a broadcast this is not the case, therefore broadcasting is considered an unreliable method of communication. [11]

Multicast communications are used to communicate data from a sender to multiple receivers. In a multicast network, the sender can choose the receivers who can get the message. This can be done by using filtering, most commonly done by filtering the identification for receivers. In a simple multicast usage, the receivers are the actors who are responsible for compensating for lost messages, and often one of the options used is to ignore lost messages. Multicast on the OUTPOST - Software Bus is implemented by filtering the IDs of messages that go through the Bus Channel(s).

Both broadcast and multicast communications are at their core asynchronous-based, which makes them unsuitable in many applications where a certain degree of reliability is required for the delivery of the data based on knowing which of the receivers has failed to receive the message. Therefore a reliable communication must be established between the senders and receivers. Per definition, a reliable multicast communication is a computer network communication protocol that offers a reliable delivery of data or packets to multiple recipients or nodes. [12]

2.4 Reliability

Neither synchronous nor asynchronous communications are considered to be reliable on their own. Reliability is a concept based on error handling that can be crucial to critical applications that require a degree of guarantee for delivery. A fundamental restriction on reliability is the fact that there is no perfectly reliable system, therefore some trade-offs are unavoidable in relation to the needs for reliability and core functionality of the system. Depending on how critical the delivery of data is, error detection and handling can include multiple concepts such as parity checks, checksums, timeouts, re-delivery of messages, etc, but the more reliable the communication is, the more complex and heavy it becomes.

Another common problem on the topic of reliability is that it is difficult to know for sure that the receiver has received a message, with the same exact data sent by the sender. Without a deep understanding of the receiver, a generic system can only offer so much in terms of guarantee of delivery, as there are many reasons that can lead for a receiver to fail to receive a message and create an error. A hardware component failure or software failures like buffer overflows and infinity loops, or failures caused by the receiver might have not completely failed, but due to an internal busy processing might delay the delivery of an acknowledgement, which in turn can arrive after a timeout has occurred on the system, and that would still lead to an error.

Reliability in itself is delivered through three general steps firstly error detection, second isolation of the unit where the error happened in order not to lead to a critical system failure and finally handling and recovery from the error. The error handling and recovery steps to reach reliability heavily depend on the implementation of the receiver, and a deep understanding of the receiver's functionality and design is required.

Nevertheless, a synchronous communication with an acknowledgement management system can provide a good basis towards reaching the desired guarantee level, especially if such a system introduces error mitigation through timeouts during busy waits and keeping track of messages and receivers who might have failed to receive the message on the first try, as such information can allow for further recovery processes such as the re-delivery of messages. The system which will be introduced in the later chapters of this thesis is a flexible generic system, that follows this philosophy and provides some of the means to achieve reliability in a more specific system.

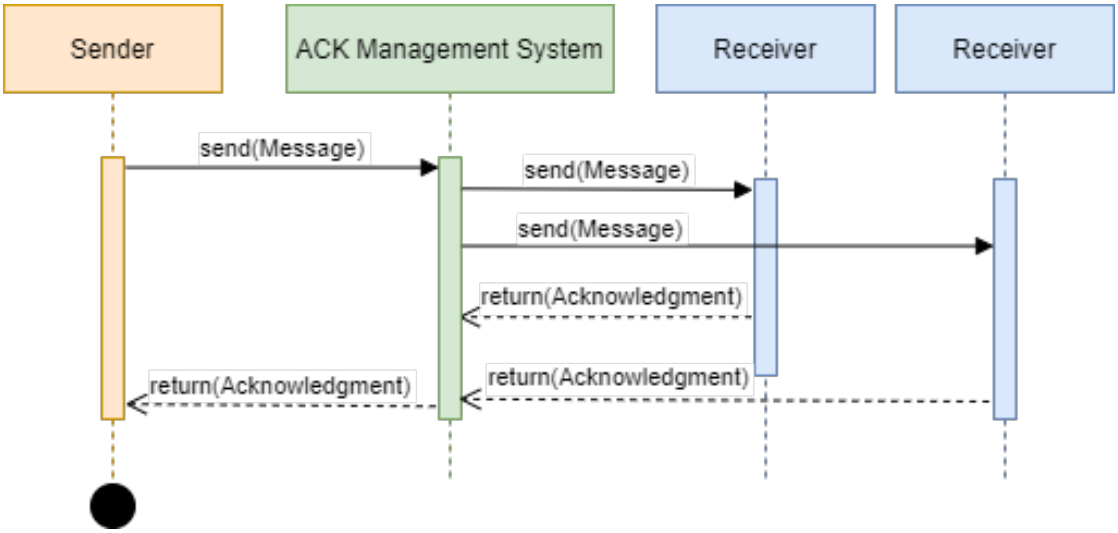

Figure 2.9: Synchronous communication using an ACK management system

2.5 Reliable Multicast Communication

One of the ways that can introduce reliability to multicast communication is to implement a synchronous communication with a timeout. Synchronicity in a multicast design can be achieved by creating a management system that handles the acknowledgement of messages. There are two ways that a message can be acknowledged, either by immediately acknowledging messages or delaying the acknowledgement of messages. In the first method, a receiver will send an acknowledgement directly to the sender after it receives the message, which can be beneficial for a relatively small number of receivers. In this method, the control is given back to the sender by the receiver, as shown in figure 2.8.

The idea of delaying the delivery of acknowledgement messages is related to the acknowledgement implosion problem occurring in systems that have a very large number of receivers. If there is a large number of receivers who will send an acknowledgement message directly to their original sender, the sender might receive in the worst-case scenario a total of number of acknowledgements equal to the number of receivers it has sent the message to. As shown in figure 2.9, in order to tackle the implosion problem, an intermediate acknowledgement management system can be created, which will delay the delivery of acknowledgements to the sender until all of the receivers have sent an acknowledgement, or until a timeout occurs, thus still forming a synchronous communi-

cation. That can then allow for further error handling and recovery which depends on the implementation of the applications where it will be used.

3 Requirements Analysis

This chapter presents the relevant stakeholders, actors, and use cases for the system that is being developed. After that functional and non-functional requirements are discussed.

3.1 Stakeholders

There are multiple stakeholders of the product. They can be categorized into internal and external groups. The internal group refers to DLR groups, while the external one is composed of 3rd parties interested in the product.

- DLR Avionics Group
- Users of the OUTPOST library
- Thesis supervisor

The users of the OUTPOST library can be both internal to the DLR and external such as other space-related companies or agencies that are currently collaborating or will in the future collaborate with the DLR's Avionics Group, and use the OUTPOST - Software Bus module for their applications.

Although the first version of the Synchronous Link is a prototype written as a proof of concept, there is an expectation that the end result of this Thesis will be further developed and used in practice by the DLR and the OUTPOST library users eventually.

3.2 Use Cases

This section describes the main functionalities from the perspective of the actors participating in the system. The feature being developed is essentially an interface that uses the OUTPOST Software Bus module to let senders and receivers interact in a synchronous method by using the system referred to as SyncLink or Synchronous Link interchangeably in this paper.

The use case diagram of the SyncLink System is illustrated in figure 3.1. The Analysis for each of the use cases will follow on chapter 3.2.2.

3.2.1 Actors

The actor is a role specified in a use case diagram for someone (or something) who interacts with the system, has a responsibility toward the system (inputs), or has expectations from the system (outputs).[15] An actor can be a person, an organization, or another system as well as another external device that interacts with the system under design to achieve its goals. The actors are external objects, which is why they are placed outside of the system as shown in figure 3.1.

Actors can be categorized as primary and secondary, where a primary actor initiates the use of the system (or a part of it) and is placed on the left, while a secondary actor is thought as a reactionary, thus placed on the right of the system. Based on the analysis described above, the actors of the system are the following:

- Primary Actors
    1. Sender
    2. Receiver
- Secondary Actors
    1. Software Bus

The sender is defined as any system or device that can interact with the system described in this paper, and its end goal is to multicast a message. A receiver on the other hand is a system or device similar to the sender, but whose end goal is to receive a message. They

are categorized as primary users as they are both independent entities that initiate the system or a part of it, and are reacting as a result of their initial request to the system and not because of the actions happening in the system.

The OUTPOST - Software Bus module is described as an actor for the reason that it is an already existing system, that can be used to deliver and receive messages. The Software Bus does not provide any sort of capability of sending acknowledgements, however the Software Bus and its Bus Channels are used by the SyncLink to achieve that goal. This actor is categorized as secondary, as it is reactant to the actions of the System described in this thesis.

3.2.2 Use Case Analysis

For the purpose of keeping an organizational level between the use case relations, the use cases are classified into the categories of parent and child-use cases. This helps create a structure in the existing relationships between the use cases. Depicted in figure 3.1, are the use cases, the actors, and their respective relationships. The «include» relationship between use cases means that i.e. use case "Check ID" is called every time the parent use case "Send Message" is also initiated.

Table 3.1 describes shortly all the use cases. An individual Use Case analysis is provided in the following tables: UC1 table 3.2, UC1.1 table 3.3, UC1.2 table 3.4, UC2 table 3.5, UC3 table 3.6, UC4 table 3.7, UC4.1 table 3.8 and finally UC4.2 table 3.9.

3.3 Functional Requirements

The functional requirements were derived after several discussions with the stakeholders. These requirements are listed in table 3.10. They are identified by their ID in the format of "F[Num]" and will be referred to throughout this paper in that manner.

The requirements F1 and F8 state that the sender should be able to send messages and receive acknowledgements. The rationale for these requirements is related to the sender's capability of communicating with the system synchronously and creating a basis for reliable communication.

Requirement F2 is introduced as a need to manage acknowledgement implosions from multiple receivers, in order to not block the sender's workflow with a large number of acknowledgements that might be incoming. F3 is related to the capability of the system to send the acknowledgements to the sender, respecting the requirement from F2 which requests to handle a possible implosion of acknowledgements scenario.

Requirement F4 states that the system should use the Software Bus and the Bus Channels from the OUTPOST - Software Bus Module in order to achieve its goals of sending and receiving messages. Requirement F5 imposes additional restrictions on the sender's communication with the system, however, it increases interoperability in the system, and it does not affect efficiency.

The rationale for requirement F6 is that the receivers should register to the system in order for the system to know which of them might have failed to receive a message. Requirement F7 is imposed on the receivers who should from time to time request to receive a message from the system.

3.4 Non-Functional Requirements

The Non-Functional requirements listed in table 3.11 describe the quality attributes of the system. Similar to the functional requirements they are identified by their ID in the format of "NF[Num]".

NF1 mentions the necessity of having the system integrated in the OUTPOST library. This is for the reason that the system needs to use the OUTPOST - Software Bus Module, and be able to run on all the platforms that the OUTPOST - Software Bus Module can.

NF2 brings up the modularity, maintainability, and testability points, which are already emphasised in the architecture of the OUTPOST library. As an integral part, this system should also fulfill these requirements.

NF3 describes the need for the code to be easily understood and well documented for future use. This increases its maintainability.

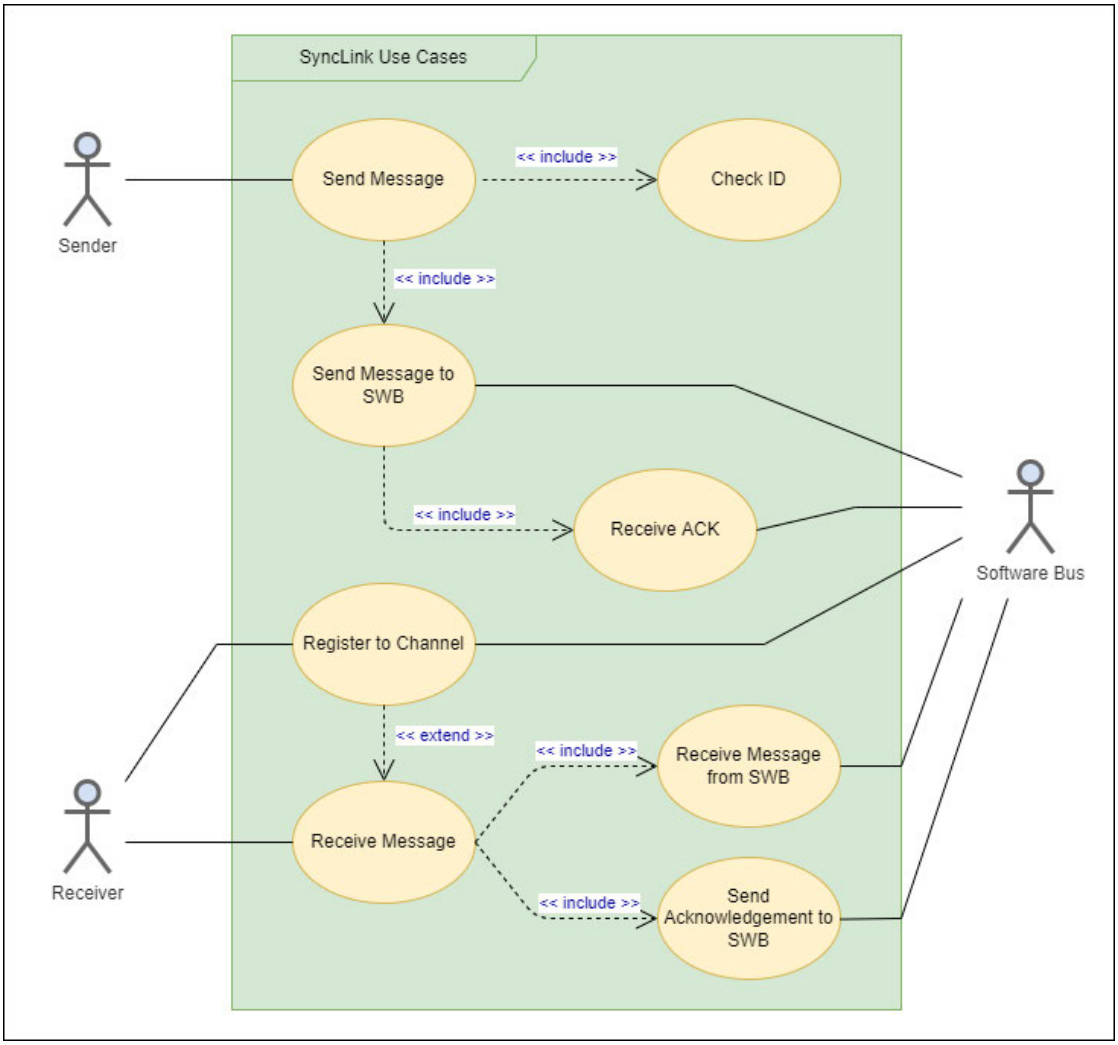

Figure 3.1: Use case diagram

NF4 is a vital requirement of the technologies and programming features allowed. The system is going to be deployed in embedded environments, therefore it should be resource-efficient, and not use i.e. dynamic memory allocations.

Use case	Title	Description
UC1	Send Message	Parent Use Case: Initiates the system. Send Message to the Receiver.
UC1.1	Check ID	Child of UC1: Checks and stores Message and Acknowledgement IDs.
UC1.2	Send Message to SWB	Child of UC1: Delivers the message to the Message Channels of SWB.
UC2	Receive ACK	Parent Use Case: Wait to receive an ACK from the Acknowledgement Channel of SWB.
UC3	Register To Channel	Parent Use Case: Registers the Receiver to an existing Message Channel
UC4	Receive Message	Parent Use Case: Manages the Receiver's requests to receive message from the Message Channel of SWB.
UC4.1	Receive Message from SWB	Child of UC4: Receives message from Message Channel of SWB if available.
UC4.2	Send Acknowledgement to SWB	Child of UC4: Sends an ACK message to the Acknowledgement Channel of SWB.

Table 3.1: List of Use Cases

Use Case 1	Send Message
Description	Initiates the system. Send Message to the receiver.
Actors	Sender
Trigger	Sender requests to send a message
Precondition	The System is ready to receive a message
Success end condition	Message is sent to the Software Bus Message Channel
Failure end condition	Message failed to be delivered to the Software Bus
Typical flow	1. Sender sends a message by calling the SendMessage() function. 2. SyncLink checks if the message is an ACK or an ID by calling UC1.1 - "Check ID" 3. If the message is an ID, UC1.2 - "Send Message to SWB" is called and the Acknowledgement is sent to the sender. 4. Else a failure message is sent

Table 3.2: "Send Message" - Use Case 1

Use Case 1.1	**Check ID**
Description	Child of UC1: Checks IDs and stores Message and Acknowledgement IDs.
Actors	Sender
Trigger	Parent Use Case 1: Function flow calls checkMessage()
Precondition	Sender has initiated the system by sending a message
Success end condition	The message has a compatible message ID
Failure end condition	The message has an incompatible message ID
Typical flow	1. checkMessage() is called 2. Check the message ID 3. If compatible, store the ID and its respective ACK and then return a success message 4. Else return fail message

Table 3.3: "Check ID" Use Case 1.2

Use Case 1.2	**Send Message to SWB**
Description	Child of UC1: Delivers the message to the SWB Message Channel of SWB.
Actors	Sender, Software Bus
Trigger	Parent Use Case 1: Function flow calls sendMessage() on SWB
Precondition	The message ID is compatible
Success end condition	Message is sent to Message Channel
Failure end condition	Software Bus failure
Typical flow	1. Function flow calls Software Bus's sendMessage() 2. The Software Bus delivers the message to the Message Channel

Table 3.4: "Send Message to SWB" Use Case 1.2

Use Case 2	**Receive ACK**
Description	Parent Use Case: Requests to receive an ACK from the Acknowledgement Channel of SWB.
Actors	Sender, Software Bus
Trigger	Use Case 1: Function flow waits for ACK to give feedback back to sender
Precondition	The message has been sent to the Message Channel
Success end condition	ACK is received in the ACK Channel
Failure end condition	Timeout has occurred
Typical flow	1. Function receiveACK() is called 2. A timeout is initiated 3. Receive from ACK channel 4. If the ACK is the same as the stored ACK return success 5. Else try again until timeout

Table 3.5: "Receive ACK" Use Case 2

Use Case 3	**Register to Channel**
Description	Parent Use Case: Registers the Receiver to a Message Channel of the SWB.
Actors	Receiver
Trigger	Receiver requests to register to SyncLink
Precondition	None
Success end condition	The Receiver is successfully registered to a Message Channel of the SWB
Failure end condition	No message available
Typical flow	1. Receiver call function registerChannel() referencing SyncLink 2. SyncLink checks if there are available message channels 3. If true, SyncLink checks if the Receiver is already registered 4. If false, the receiver is registered 5. Else an "Invalid State" message is returned to the Receiver

Table 3.6: "Register to Channel" Use Case 3

Use Case 4	**Receive Message**
Description	Parent Use Case: Manages the Receiver's requests to receive message from the Message Channel of SWB.
Actors	Receiver
Trigger	Receiver requests to receive a message from SyncLink System
Precondition	The Receiver is registered
Success end condition	ACK is sent to the ACK Channel and the Receiver receives the message
Failure end condition	No message available
Typical flow	1. Receiver call function receiveMessage() referencing SyncLink 2. SyncLink checks if the Receiver was registered 3. Check if there are messages on the Message Channel 4. If message available proceed to UC4.1 and UC4.2 5. Else send "No message available" to receiver

Table 3.7: "Receive Message" Use Case 4

Use Case 4.1	**Receive Message from SWB**
Description	Child of UC3: Receives message from Message Channel of SWB if available.
Actors	Receiver, Software Bus
Trigger	Parent Use Case 4: Function flow calls receiveMessage() from SWB
Precondition	A message is sent to the SWB's Message Channel
Success end condition	Retrieves a message from the Message Channel
Failure end condition	Software Bus or Bus Channel failure
Typical flow	1. Function flow calls receiveMessage() from Message Channel 2. The message is retrieved from the Message Channel

Table 3.8: "Receive Message from SWB" Use Case 4.1

Use Case 4.2	**Send Acknowledgement to SWB**
Description	Child of UC4: Sends an ACK message to the Acknowledgement Channel of SWB.
Actors	Software Bus
Trigger	Parent Use Case 4: Function flow calls sencACK() to SWB
Precondition	Receiver has requested a message from SyncLink and a message is available at the Message Channel
Success end condition	Sends an ACK to the Acknowledgement Channel
Failure end condition	Software Bus or Bus Channel failure
Typical flow	1. Function flow calls sendACK() to the Acknowledgement Channel 2. If the ACK to be sent is the same with the stored ACK send it to the Acknowledgement Channel.

Table 3.9: "Send Acknowledgement to SWB" Use Case 4.2

ID	Description	Derived from	Priority	Acceptance criteria
F1	The System shall be able to receive messages from senders.	UC1	High	The sender shall be able to send messages.
F2	The System shall manage acknowledgements from multiple receivers.	UC4	High	The system should provide a method to manage the acknowledgement implosion.
F3	The System shall send acknowledgements to the senders.	UC1, UC2	High	The system should send an acknowledgement back to the sender for the correct message.
F4	The System shall use the OUTPOST - Software Bus and the corresponding Bus Channels to send and receive messages and acknowledgements.	UC1.2, UC2, UC4.1, UC4.2	High	The system shall be integrated inside the OUTPOST - Software Bus Module, thus having access to the whole of the OUTPOST library.
F5	The System shall differentiate messages from acknowledgements.	UC1.1	High	The system should distinguish messages by imposing a restriction to the IDs of messages and acknowledgements.
F6	The Receivers shall be able to Register to Bus Channels.	UC3	High	The Receivers shall be able to successfully register to the Bus Channels of the SWB.
F7	The Receivers shall request to receive messages from the System.	UC4	High	The receiver should have access to a function in the system that allows it to request a message if one is available.
F8	The Senders should be able to receive ACK messages/reports.	UC1, UC2	Middle	The sender should be available to receive the ACK messages from the system.

Table 3.10: Functional requirements

ID	Description
NF1	The System should be integrated in the OUTPOST library.
NF2	The code should be modular, maintainable and testable.
NF3	The code should be intuitive.
NF4	The code should be resource efficient as it is used in Embedded Systems.

Table 3.11: Non-Functional requirements

4 Design

This chapter describes the required parts to building a reliable multicast for the OUTPOST Software Bus module. It also explains the main implementation decisions for the Synchronous Link System.

4.1 Reliable Multicast Solution

It should be mentioned that the development of a system that is specific to the OUTPOST library was the only valid option as OUTPOST has a generally unique and complex architecture and requirements that have no existing equivalent. The OUTPOST library is an in-house library built mainly from scratch by the German Aerospace Center (DLR), so publicly marketed solutions of such a software do not exist as of the writing of this thesis.

4.1.1 Developing Own Solution

A main decision and motivation for the Thesis was to write a custom solution for the reliable multicast using the OUTPOST Software Bus module, as that would provide full adaptability, scalability and maintainability of the implementation.

4.1.2 Devising a Problem Schedule

The OUTPOST library is an in-house project of the DLR with a modular architecture which allows it to be used as a development advantage. The current architecture of OUTPOST was explained on chapter 2, however, this paper focuses only on the OUTPOST - Software Bus module.

The main goal of this thesis is to develop a management interface that can allow for a reliable multicast communication. This interface is to be built on top of the existing Software Bus module, thus it can be safely concluded that there is no motivation to change the underlying code used in the OUTPOST library and of the Software Bus module.

The idea behind a synchronous communication lies on a sender, or a sender's thread waiting to receive an acknowledgment message in order to continue its operation, bringing forward the need to exchange the current asynchronous and loosely-coupled broadcast type of communication to a synchronous one. This connection of the SWB module to the management system that could synchronously link the senders to the receivers inspired the name of the system under design to be Synchronous Link or short SyncLink.

In the previous status of the Software Bus module, the IDs of the messages were implemented alongside the filters on the bus and bus channels for the purpose of a possibility to multicast messages by using the filters to exclude receivers from receiving a message. Nevertheless, this type of multicast is not reliable, as it does not provide the sender for a confirmed delivery of the data, and a re-delivery of a message is not implemented from the sender's as they wouldn't know which of the receiver's have failed to receive a message.

4.2 Architecture

To come up with the architecture, the requirements discussed on the previous chapter and the background provided on chapter 2 were used as a starting point. Depicted in figure 4.1 is a sequence diagram of a communication use the system under design called SyncLink.

4.2.1 Actors

To increase the understanding of the roles, a coloring system is provided for grouping the actors, systems, and subsystems into their specific categories based on their functionality and interaction with the system as follows:

- Blue
    1. Sender
    2. Receiver(s)
- Red
    1. Software Bus
    2. Acknowledgement Channel
    3. Message Channel(s)
- Green
    1. SyncLink

The sender and the receiver are depicted in blue and belong to the same category of external actors who interact with the system, but are not part of it. A sender can reach multiple receivers using the system, however the actions leading to the synchronous delivery of a message to the receiver are similar. In red are depicted the OUTPOST - Software Bus, and the Bus Channels who serve different roles, namely an Acknowledgement Channel for the Acknowledgement messages, and one or many Message Channels depending on the registered number of Receivers. In green, the system under design is depicted. SyncLink acts as an intermediary between the external actors and the Software Bus system, and fills the requirements described on chapter 3.

4.2.2 Sequence Flow

The synchronous delivery of the message is achieved using the following flow.

As a prerequisite, at least one receivers should be registered to a channel in order to allow for the intended usage of the system. This is not done directly to the Software Bus as it would have normally be done in the OUTPOST - Software Bus module, but instead through SyncLink, which after processing the registration request and storing the receiver's ID internally, will deliver the request to the Software Bus, which will then

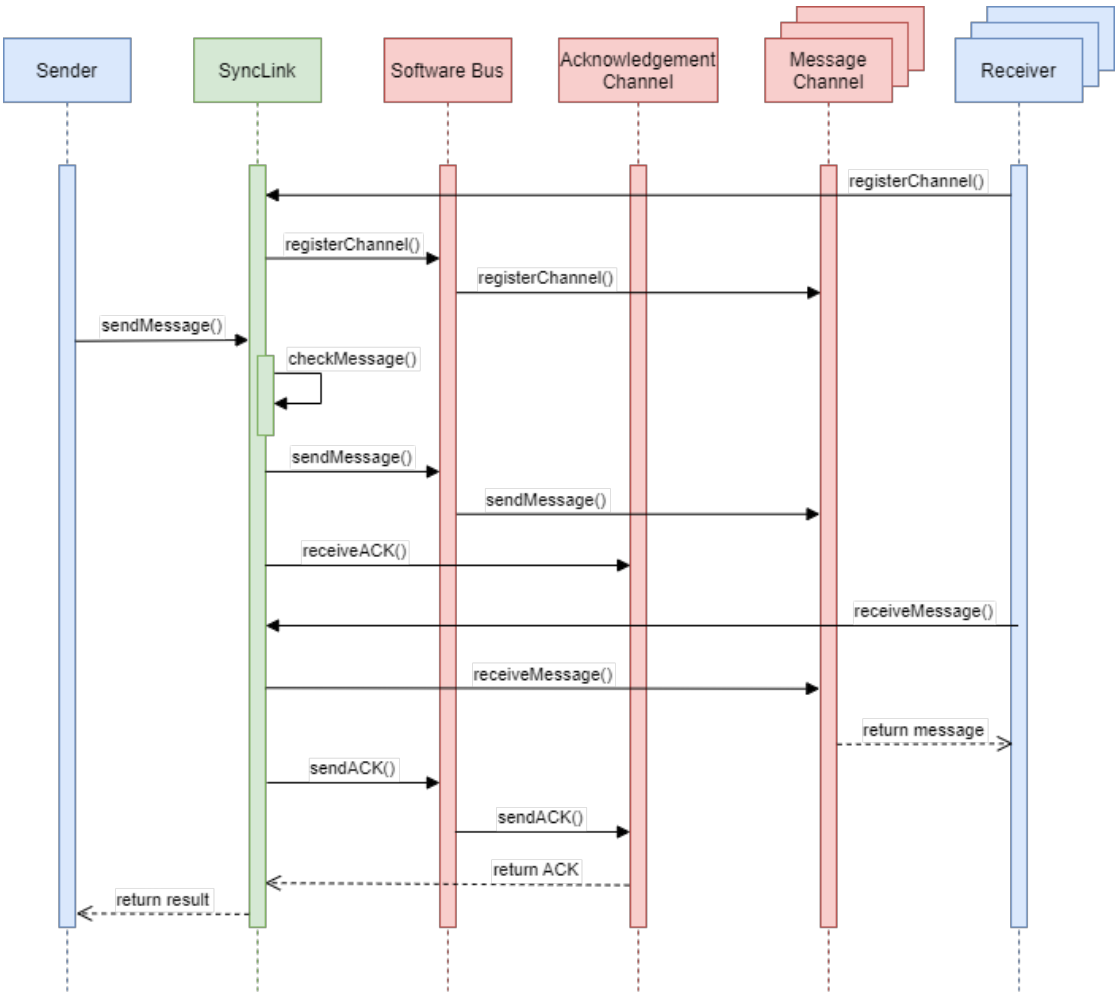

Figure 4.1: Communication using the asynchronous communication paradigm

assign a message channel to the receiver. The receiver ID is stored to allow for identification in case of a failed delivery of messages to the intended receiver. This process should be performed only once for each receiver, and after this step, the receiver will be registered.

The sender will try to deliver a message and the expected number of receivers to be reached by referencing SyncLink. A message should consist of an ID and the data. The ID should have it's first bit set to 1, otherwise it will not be accepted. The reason for setting the limitation on the first bit of the ID of messages to 1 is a result of filtering in order to distinguish between the message and acknowledgement channels, which for the later, the first bit of the ID must be a 0. This check is performed by the SyncLink on the CheckMessage() sequence. Once the message is checked and fills the criteria, the SyncLink will proceed to forward the message to the Software Bus, in which through a series of internal processes as described in subsection 2.2.2 will be delivered to the Message Channel(s). Immediately after that, SyncLink will wait for an acknowledgement for the delivered message on the Acknowledgement Channel.

The receiver in its own time intervals is from time to time checking if there are any messages available by requesting to receive a message from SyncLink. If a message is available on the receiver's designated Message Channel, the receiver will receive the message. Immediately after that, an acknowledgement message for the corresponding ID that was delivered alongside the ID of the receiver who received the message will be sent to the Acknowledgement Channel.

The previous wait to receive acknowledgements on the receiveACK() sequence, will now verify that the acknowledgement ID is the one that was initially expected, and will internally confirm that the receiver has received the message, however SyncLink will not send any results to the sender if the expected number of receivers is not reached. Assuming that in this sequence, the number of expected receivers would be 1, the operation result would be a success.

4.2.3 Error mitigation and recovery

The design of SyncLink uses timeout based error mitigation. Unless the message has reached the number of receivers that was initially expected by the sender, SyncLink will

wait until a timeout occurs. In that case, the sender will receive a timeout result, and can further access information to the IDs of the receivers that failed to receive the message. This way, SyncLink through its synchronous communication and timeout generates an error detection.

Regarding recovery, once it receives a timeout operation result, the sender can decide to re-deliver the message, however, without having a deep knowledge of the receiver's implementation there can not be any correct information if the receiver has failed completely, if it is in a frozen state and might need to be restarted, or if any other error like buffer overflows have occurred. For this reason, SyncLink's generic architecture does not provide a sophisticated recovery method.

5 Implementation

This chapter describes the project environment and the implementation details for the architecture described in chapter 4.

5.1 Project environment

In this section, the environment of the project, including tools and setup.

5.1.1 Tools

The programming language of the implementation is C++ with an embedded coding style specific to the OUTPOST library. The project is developed using the Visual Studio Code from Microsoft Corporation[13] as an editor and compiled with Debian GNU/Linux[14], a Linux-based operating system. The library is set up on an online repository on GitLab - a DevOps software provided by GitLab Inc.[9] and is version controlled using Git - a free version control software by Software Freedom Conservancy[10].

Apart from Visual Studio Code, which was a tool choice of the author used only for writing and editing code, the OUTPOST - Software Bus module was already using a combined C++ and Debian environment, and the repositories were already set up on GitLab and version controlled by Git, therefore it was only logical to continue with the same tools.

5.1.2 File setup

The implementation is located on the outpost-core/modules/swb/src on a separate branch on GitLab of the swb-communication named "/feature/SyncLink". The implementation

consists of two files named "sync_link.h" and "sync_link_impl.h" where the first file holds the SyncLink class, its constructor, its class methods and member variables with their respective public, protected and private restrictions, while the second implement file holds the implementation of the methods and is where the core functionality of SyncLink lies.

5.2 Class methods and implementation

The constructor of the SyncLink class references a Software Bus and is instantiated with an Acknowledgement Channel. A destructor is also provided. A limitation in this class is the maximum number of receivers provided by a member variable of the class, as dynamic allocation is avoided due to the embedded nature of the system. A timeout for the receiveACK() method is arbitrarily chosen to be 3 seconds, but that can also be modified regarding the needs of the application. Another arbitrarily chosen decision is the counter for the number of while-loops happening in receiveACK(), which is chosen to be 5.

The main methods that implement the core functionality and that will further be discussed in this section are the following:

- registerChannel(receiverID)
- sendMessage(id, data, nrOfExpectedReceivers)
- receiveACK(timeout)
- receiveMessage(message. timeout, receiverID)

5.2.1 Registering a Channel

The registering of a channel is implemented by firstly looping with a for-loop through a receiver array where the receiver IDs are stored. If the receiver ID exists, the operation is ended with an error operation result because the receiver cannot register again as a message channel was already assigned to that receiver ID. Another check is also performed to see if there is an available message channel, and only if there is an available channel, the registration of the receiver can take place, otherwise, an error operation result will be returned. If both conditions are fulfilled respectively, the registration will proceed

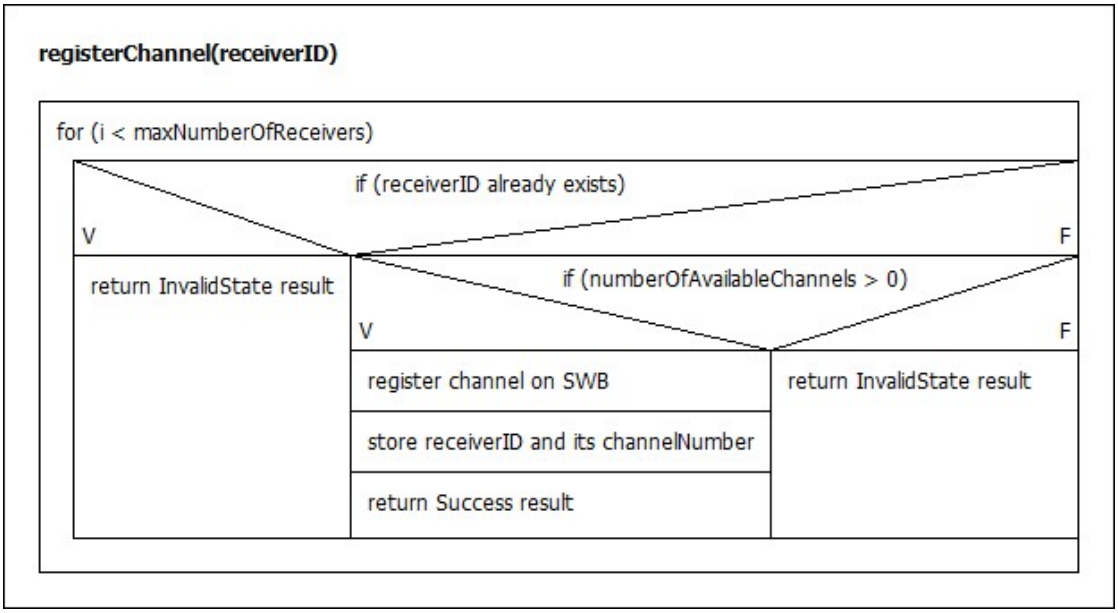

Figure 5.1: Nassi–Shneiderman diagram of registerChannel method implementation

to happen by registering the ID to the channel, and the receiver ID will then be stored on the receiver array alongside the channel number which is assigned to it. Then, a success operation result will be returned to the receiver. This process is described using a Nassi-Shneiderman diagram in figure 5.1.

5.2.2 Sending a message

The send of a message is implemented by firstly checking the ID of the message. As per restrictions that were discussed in the previous chapter, this part checks if the first bit of the ID is a 1, and if that is true, it stores the corresponding message ID and the expected acknowledgement ID for that message internally. If the ID is valid, the message is sent to the SWB, which further transmits this message to the message channels. A while-loop is then initiated, with the condition for the number of expected receivers to be bigger than 0. This number is given as an input parameter by the sender. Inside the while-loop, the receive acknowledgement method is called and its operation result is analysed by a conditional if. If the result is a success, the number of expected receivers is decreased, otherwise, a timeout is returned. This will happen until the initial condition of the while-loop is not met, and there are no more expected receivers. This will result

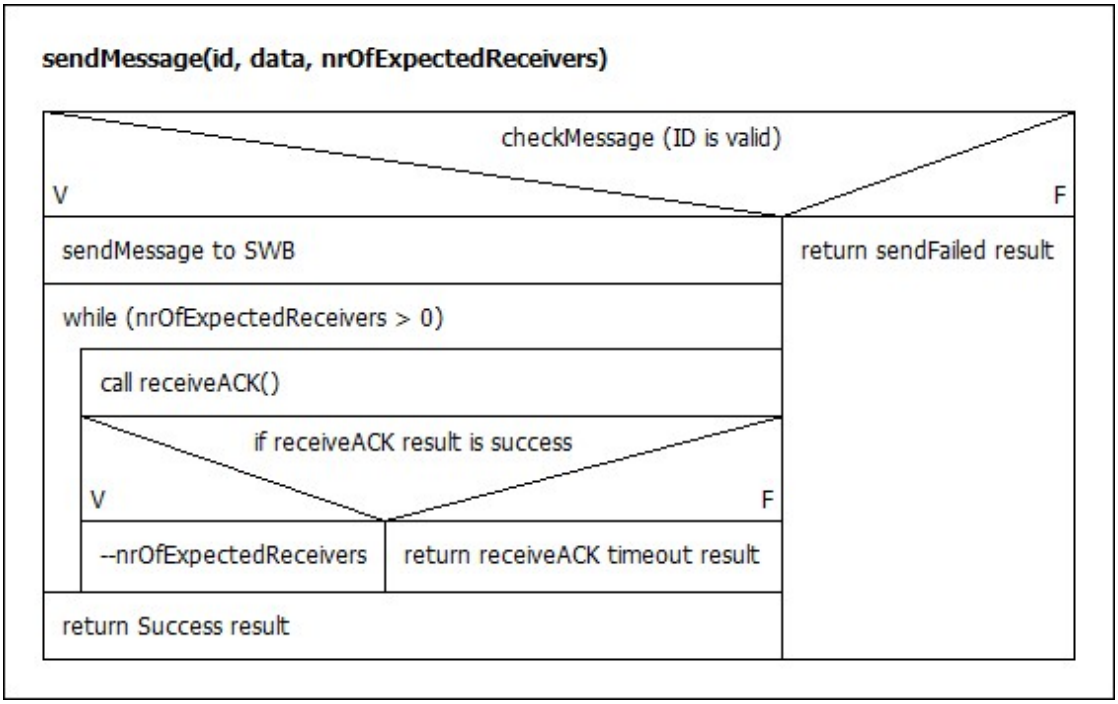

Figure 5.2: Nassi–Shneiderman diagram of sendMessage method implementation

in a break from the while-loop and will return a success operation result to the sender. This process is described using a Nassi-Shneiderman diagram in figure 5.2.

5.2.3 Receiving an acknowledgement

The receive of an acknowledgement method happens immediately after the message has been sent for distribution to the SWB. This function is instantiated with a counter loop, arbitrarily chosen to be 5. A while-loop checks if the counter has already reached 0. If that is the case the method will return a timeout operation result. If the counter is still bigger than 0, a request will be made to receive a message from the Acknowledgement Channel. If there is a message, the logic will then compare the received acknowledgement ID from the channel to the expected acknowledgement ID. If that is the case, SyncLink will mark the receiver ID internally as successful, and will return a success operation result, otherwise it will decrease the counter and restart the loop operation. A break point is if the receive message request will receive an operation result other than success, which will immediately break the while loop and return a timeout operation result. This process is described using a Nassi-Shneiderman diagram in figure 5.3.

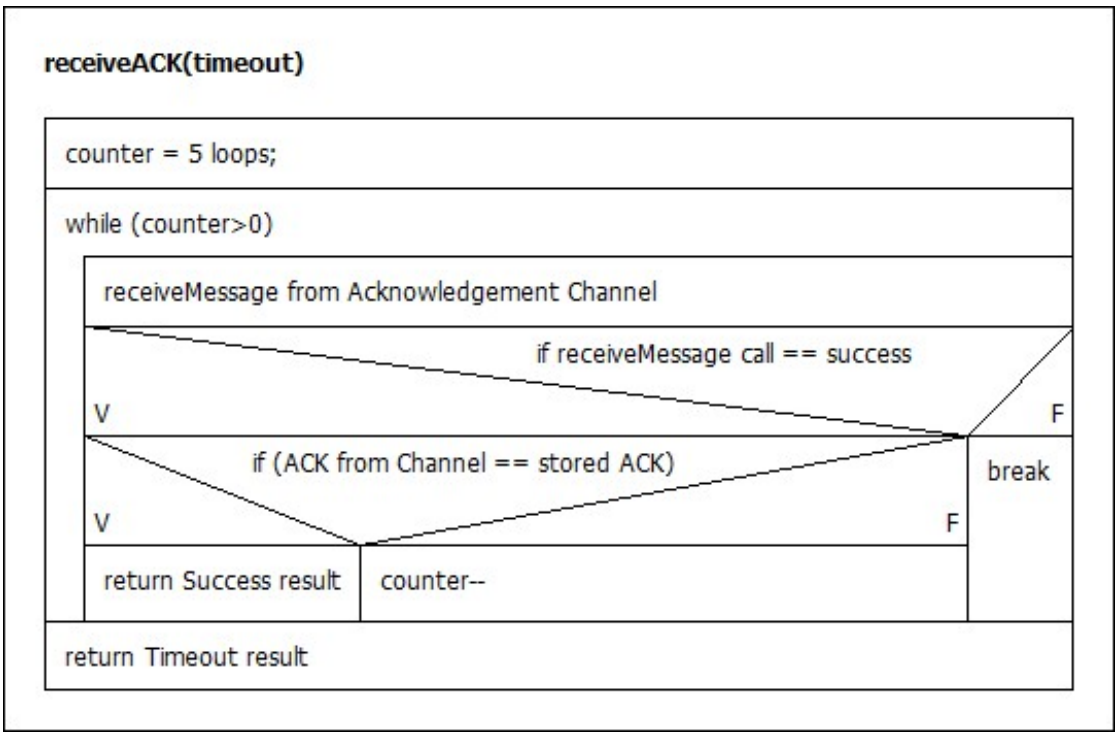

Figure 5.3: Nassi–Shneiderman diagram of receiveACK method implementation

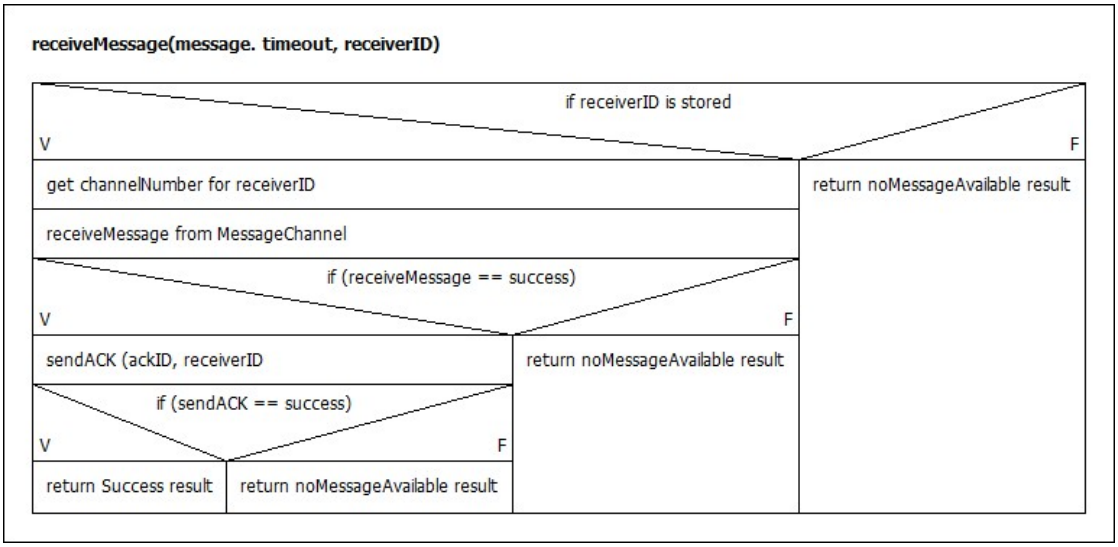

Figure 5.4: Nassi–Shneiderman diagram of receiveMessage method implementation

5.2.4 Receiving a message

The receive of a message is implemented by firstly checking if the ID of the receiver is already stored. If that is true, the channel is then found internally and a request to receive a message from the assigned message channel is then initiated. The operation result of this receive message request is then checked if it was successful. If that is true, an acknowledgement ID is generated by flipping the first bit of the message id to 0, and together with the receiver ID, the sent to the SWB and eventually from the SWB to the acknowledgement channel. This will then return a success operation result to the receiver. This process is described using a Nassi-Shneiderman diagram in figure 5.4.

6 Results and Evaluation

This chapter looks at the results of the implementation after being tested and simulated, discusses possible shortcomings of the system, gives an evaluation of the desired goals, and concludes with an overview of future work.

6.1 System requirements

As per the requirements discussed in chapter 3, the program should be able to run on any application that the OUTPOST library and more specifically the OUTPOST - Software Bus module is able to run. The program is written in C++ and is tested by a modified simulation example for a housekeeping mechanism originally provided by the DLR.

6.2 Testing

This section outlines the testing procedure used for the SyncLink system.

Testing is performed on a modified version of the existing housekeeping example provided by the DLR, which instead of using the software bus, uses SyncLink to communicate synchronously. The program is built on Debian, having three separate Debian terminals for the simulation, the receiver, and the sender. The sender has to toggle the housekeeping, by using the command "toggle hk" on the terminal.

6.2.1 Failure scenario

In figure 6.1, SyncLink is run on a situation where the receiver has completely failed and is non-responsive. In terms of simulation, this would mean that the receiver is not built, and only two terminals are used, the simulation and the sender terminals. The

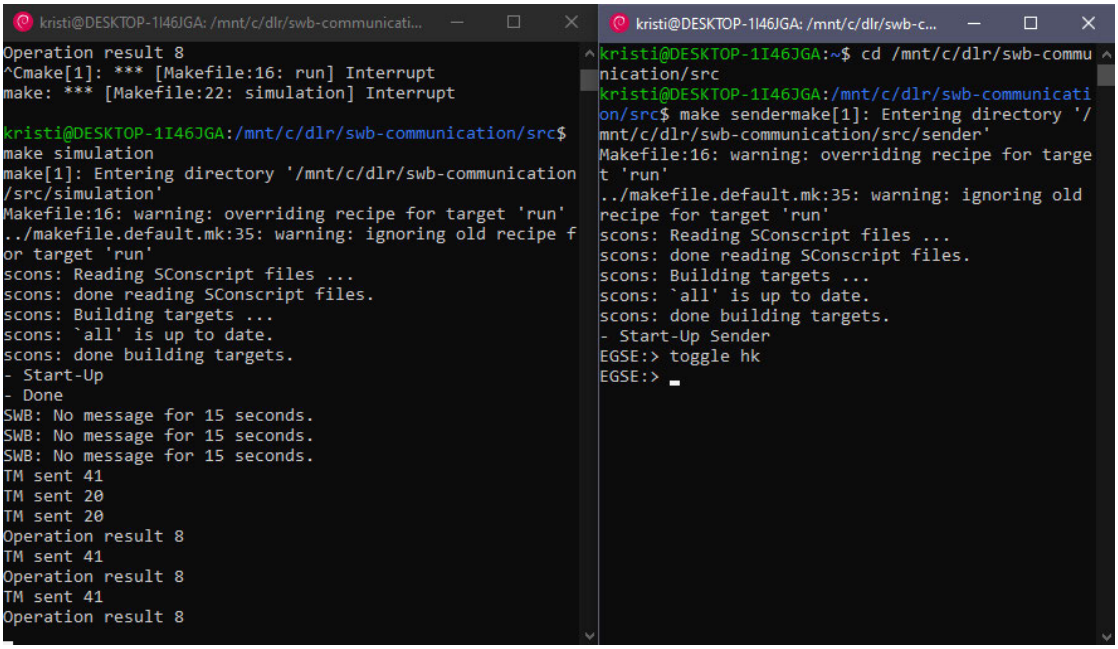

Figure 6.1: Simulation without a receiver - a timeout error occurring

expected result would be a timeout coming originally from the receive acknowledgement method, as the communication fails to receive an acknowledgement message because the message is not picked up by a receiver. The timeout operation result has the code 8, as specified on the internal outpost class "outpost::swb::OperationResult", located on "swb/types.h".

6.2.2 Success scenario

In figure 6.2, SyncLink is run on a situation where the receiver has been available and has received the message. In terms of simulation, this would mean that the receiver along the simulation and the sender terminals are built. The expected result would be a success operation coming originally from the receive acknowledgement function, as the communication successfully receives an acknowledgement message as the message is picked up by the receiver on its message channel, and matches the expected receiver number from the sender. Using prints, on the simulation terminal can be seen the IDs of the messages. The sender sent a message with an ID 0x8123, with the first bit being a 1. The acknowledgement for the message received on the acknowledgement channel, as expected had a reversed first bit to 0 as the ID had the code 0x0123, and matched

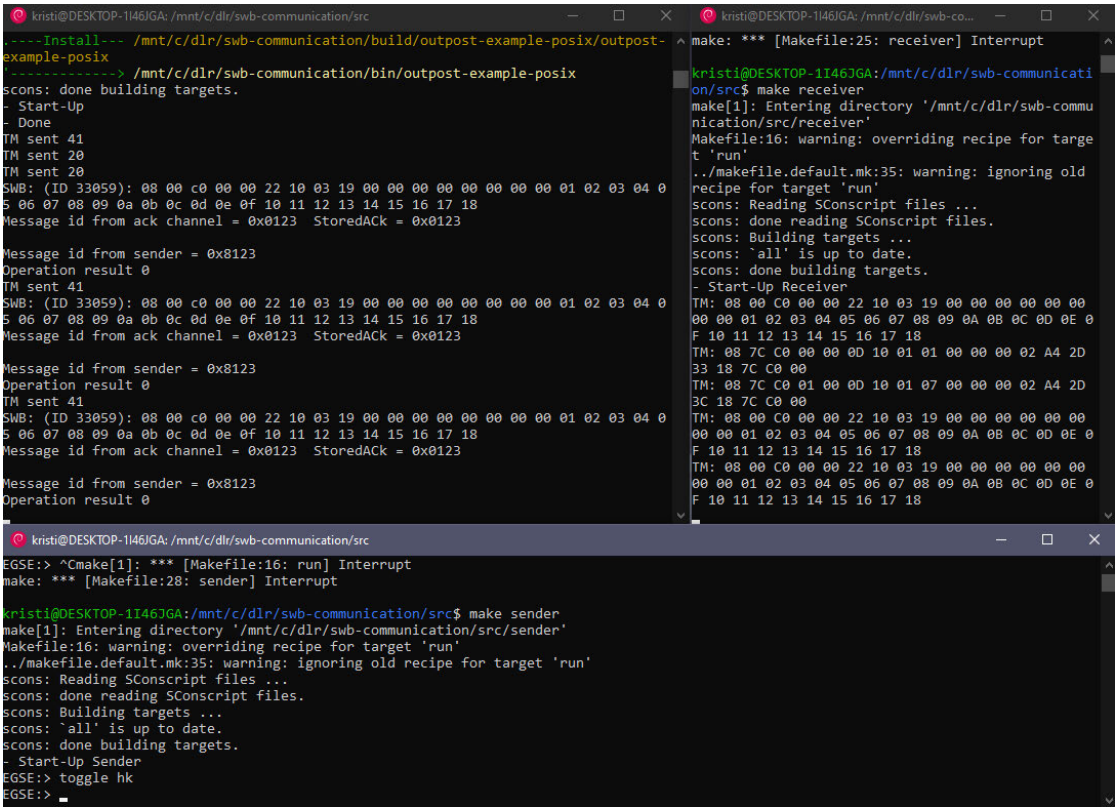

Figure 6.2: Simulation with a success operation result

the expected acknowledgement ID which is calculated and stored internally after the message is sent to the SWB. The success operation result has the code 0, as specified on the internal outpost class "outpost::swb::OperationResult", located on "swb/types.h".

6.2.3 Unit testing

In order to assert the correct functionality of the SyncLink class and its methods, the development process of SyncLink included a continuous manual logging/printing of different values on the Debian terminal for every method, however, this process was not formalized and integrated into a full-scale automatic build.

6.3 Evaluation

This section provides an evaluation of the results of SyncLink.

6.3.1 Synchronous communication

As demonstrated in the testing section, the receiver was able to successfully receive messages using SyncLink, and the sender was able to receive acknowledgements for their delivery of messages, provided by SyncLink 's implementation.

6.3.2 Reliability and error mitigation

By internally using a timeout-based system, SyncLink was able to successfully deliver error operation results to the sender when a timeout had occurred and the message was not delivered to all receivers. SyncLink also stored the IDs of all the receivers, the number of the message channel to which they were connected, and the result of delivery, thus further allowing the sender to know which receiver had failed to receive a message if needed.

6.4 Future work and extensions

Due to time constraints, some functionalities were not implemented in the system. The system can be extended to have an internal mechanism for re-sending the messages to the failed receivers, instead of having the sender send the message again, as well as having an unsubscribe method for the receivers. The current version of SyncLink is only good for delivering one message at a time and cannot deal well with subsequent messages as it cannot distinguish between two messages with the same ID but different data. A suggestion to solve this problem is to add a checksum for the data. Additionally, it is currently supporting a 1:n relationship, and the implementation can be extended to better manage an m:n relationship which at this moment was neither tested nor implemented. Another interesting problem to solve for future developers is dealing with late acknowledgements that might arrive after the timeout has occurred.

6.4.1 Gateway for distributed systems

SyncLink can be extended to work on distributed systems that run on different processors, connected through a gateway that serves as a link between two or more such systems that run their own SyncLink and SWB.

6.4.2 Reliable asynchronous communication

SyncLink can have an asynchronous counterpart that can manage the acknowledgements, and after some time it can deliver an acknowledgement report to the sender, or the sender running its own thread can check from time to time for the acknowledgement of messages, but with the highlight that this exchange of information happens asynchronously, instead of using a blocking - poll or a busy-wait as the synchronous communication of SyncLink currently does.

7 Conclusion

This paper is part of a project at the German Aerospace Center (DLR), who developed an asynchronous software bus module as part of the OUTPOST library, and this thesis's objective is to build a synchronous link system between senders, receivers and the software bus, that at the same time offers the means for reliability by using a timeout, delivering error operation results and storing information for failed delivery of messages to receivers.

This paper looked at the required features for this system; researched the available libraries and simulation examples; chose an appropriate architecture based on multiple communication paradigms, and finally, it delivered a basic prototype version.

The current prototype can be further extended. A gateway for distributed systems can be built. Another reliable asynchronous version can be implemented.

Bibliography

[1] Chen Chen, Yoav Tock, Sarunas Girdzijauskas: *BeaConvey: Co-Design of Overlay and Routing for Topic-based Publish/Subscribe on Small-World Networks.* [Online]. 2018. – URL https://www.researchgate.net/publication/325884886_BeaConvey_Co-Design_of_Overlay_and_Routing_for_Topic-based_PublishSubscribe_on_Small-World_Networks. – Accessed: 2022-02-12

[2] Deutsche Zentrum für Luft- und Raumfahrt e.V.: *Eu:CROPIS (Euglena and Combined Regenerative Organic-food Production in Space).* [Online]. 2016. – URL https://directory.eoportal.org/web/eoportal/satellite-missions/e/eu-cropis. – Accessed: 2022-02-12

[3] Deutsche Zentrum für Luft- und Raumfahrt e.V.: *Institute of Space Systems Status Report 2007–2016.* [Online]. 2016. – URL https://www.dlr.de/irs/en/Portaldata/46/Resources/2015_dokumente/DLR-RY_Status_Report_2007-2016_Part_I.pdf. – Accessed: 2022-02-12

[4] Deutsche Zentrum für Luft- und Raumfahrt e.V.: *DLR at a glance.* [Online]. 2022. – URL https://www.dlr.de/content/en/downloads/publications/brochures/2019/dlr-at-a-glance-2019.pdf?__blob=publicationFile&v=5. – Accessed: 2022-01-26

[5] Deutsche Zentrum für Luft- und Raumfahrt e.V.: *Institute of Space Systems.* [Online]. 2022. – URL https://www.dlr.de/irs/PortalData/46/Resources/2015_dokumente/Institutsbroschu_re_Bremen_ENG_ONLINE_251115.pdf. – Accessed: 2022-02-16

[6] Deutsche Zentrum für Luft- und Raumfahrt e.V.: *Martian moon mission MMX.* [Online]. 2022. – URL https://www.dlr.de/rb/desktopdefault.aspx/tabid-13789/. – Accessed: 2022-02-16

[7] Fabian Greif, Jan-Gerd Mess: *Introduction to OUTPOST*. [Online]. 2020. – Accessed: 2022-01-26

[8] Frank Dannemann, Fabian Greif: *Software Platform of the DLR Compact Satellite Series*. [Online]. 2014. – URL https://elib.dlr.de/89344/. – Accessed: 2022-01-29

[9] GitLab Inc.: *About GitLab*. [Online]. 2022. – URL https://about.gitlab.com/. – Accessed: 2022-04-16

[10] GitLab Inc.: *Git*. [Online]. 2022. – URL https://git-scm.com/. – Accessed: 2022-04-16

[11] Jo-Mei Chang, Nicholas F. Maxemchuk: *Reliable Broadcast Protocols*. [Online]. 1984. – URL https://dl.acm.org/doi/abs/10.1145/989.357400. – Accessed: 2022-02-12

[12] Mamillapalli Sudhakar, Belair Stephen P.: *Reliable Multicast Communication*. – URL https://www.researchgate.net/publication/302598559_Reliable_multicast_communication

[13] Microsoft Corporation: *VS Code*. [Online]. 2022. – URL https://code.visualstudio.com/. – Accessed: 2022-04-16

[14] Software in the Public Interest, Inc.: *Debian*. [Online]. 2022. – URL https://www.debian.org/. – Accessed: 2022-04-16

[15] Visual Paradigm: *What is Use Case Diagram?* [Online]. 2022. – URL https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/. – Accessed: 2022-03-01

A Glossary

- SWB - Software Bus
- SyncLink - Synchronous Link System
- DLR - Deutsches Zentrum für Luft- und Raumfahrt e.V.
- OUTPOST - Open modUlar sofTware PlatfOrm for SpacecrafT
- ACK - Acknowledgement
- ID - Identification

B Source Code

```

#ifndef OUTPOST_SYNCLINK_H
#define OUTPOST_SYNCLINK_H

/*
 * Authors:
 * - 2021-2022, Kristi Kola (DLR + HAW HAMBURG)
 */
#include "bus_channel.h"
#include "bus_handler_thread.h"
#include "types.h"

#include <outpost/swb/software_bus.h>
#include <outpost/utils/container/container.h>
#include <outpost/utils/container/shared_buffer.h>
#include <stdint.h>
#include <type_traits>

namespace outpost
{
namespace utils
{
class SharedBufferPoolBase;
class ConstSharedBufferPointer;

template <typename T>
class ReferenceQueueBase;
}  // namespace utils

/******************************************************/

namespace swb
{

template <typename IDType>
```

```
36 class SyncLink
37 {
38 public:
39     /**
40      * Constructor initialized with a software bus and channels.
41      */
42     SyncLink(outpost::swb::SoftwareBus<IDType>& bus) :
43         mSoftwareBus(bus)
44     {
45         acknowledgmentChannel.getFilter().setRange(0x0000,0x7FFF);
46         bus.registerChannel(acknowledgmentChannel);
47     };
48 
49     /**
50      * Destructor
51      */
52     virtual ~SyncLink() = default;
53 
54     /**
55      * Receiver registers a Multicast Link channel.
56      */
57     outpost::swb::OperationResult
58     registerChannel(const IDType id);
59 
60     /**
61      * Wrapper to sendMessage() function on the software bus.
62      * @param msg is sent to the software bus.
63      */
64     outpost::swb::OperationResult
65     sendMessage(const IDType id, const outpost::Slice<const uint8_t>& data,
    uint32_t& nrOfReceivers);
66 
67     /**
68      * A request called by the receiver to retrieve a message if there is a
    message on the
69      * message channel.
70     */
71     outpost::swb::OperationResult
72     receiveMessage(outpost::swb::Message<IDType>& message, outpost::time::
    Duration timeout, uint8_t receiverID);
73 
74 protected:
75     /**
76      * Checks an incoming message for validity.
```

```
77      * \return Returns true if the message is valid for sending, false
        otherwise.
78      */
79     virtual bool
80     valid(IDType, const outpost::Slice<const uint8_t>&)
81     {
82         return true;
83     }
84
85     /**
86      * Checks whether an incoming message is a message or an acknowledgement.
87      * If it is a message, then it is stored @param mStoredID, and a message
        is
88      * sent to the swb.
89      * If it is an acknowledgement, then it is stored on @param mStoredAck
90      *
91      */
92     outpost::swb::OperationResult
93     storeID(const IDType id, const outpost::Slice<const uint8_t>&);
94
95     //function that returns the correct index mReceiversArray channel number;
         input: Receiver's ID
96     int
97     findIndex(const IDType id);
98
99     /**
100     * Checks if the ID if it is an ACK
101     * If first bit of ID = 1 => message,
102     * else ID = 0 => ACK
103    */
104    bool
105    checkMessage(const IDType id);
106
107    /**
108     * Flips the first bit of the ID, and stores it at mStoredAck.
109    */
110    void
111    storeAckID(const IDType id);
112
113    /**
114     * Receives an ACK from the Acknowledgment Channel on the SWB
115     * Waits until the receiver has received the message and an ACK has been
        generated.
116    */
```

```
117     outpost::swb::OperationResult
118     receiveACK(outpost::time::Duration timeout = outpost::time::Seconds(3));
119 
120     /**
121      * Sends an acknowledgement to the Swb
122     */
123     outpost::swb::OperationResult
124     sendACK(const IDType id, uint8_t receiverID);
125 
126     outpost::swb::SoftwareBus<IDType>& mSoftwareBus;
127     BufferedBusChannelWithMemory<10 /*size of queue*/, IDType, RangeFilter<
        IDType>> acknowledgmentChannel;
128 
129     static constexpr uint32_t mMaxNumberOfReceivers = 5;
130     BufferedBusChannelWithMemory<10 /*size of queue*/, IDType, RangeFilter<
        IDType>> messageChannelArray[mMaxNumberOfReceivers];
131 
132 private:
133     uint32_t mStoredID;
134     uint32_t mStoredAck;
135     uint32_t mAckCounter;
136     uint32_t mExpectedNrOfReceivers;
137     uint32_t mNumberOfAvailableChannels = 5;
138     uint32_t mReceiversArray[mMaxNumberOfReceivers];
139     bool mHasReceivedMessage[mMaxNumberOfReceivers];
140 }; //Class SyncLink
141 
142 } //namespace SyncLink
143 } //namespace Outpost
144 
145 #include <outpost/swb/sync_link_impl.h>
146 
147 #endif // SYNCLINK_SENDER_IMPL_H
```

Listing B.1: sync_link.h

```
1 
2 #ifndef SYNCLINK_IMPL
3 #define SYNCLINK_IMPL
4 /*
5  * Authors:
6  * - 2021-2022, Kristi Kola (DLR + HAW HAMBURG)
7  */
8 #include <outpost/utils/container/reference_queue.h>
9 #include <outpost/utils/container/shared_object_pool.h>
```

```
10 #include <outpost/utils/container/shared_object_pool.h>
11 #include <iostream>
12
13 namespace outpost
14 {
15 namespace swb
16 {
17     template <typename IDType>
18     outpost::swb::OperationResult
19     SyncLink<IDType>::registerChannel(const IDType id)
20     {
21         for (uint32_t i = 0; i < mMaxNumberOfReceivers; i++)
22         {
23             if (mReceiversArray[i] == id)
24             {
25                 return outpost::swb::OperationResult::invalidState;
26             }
27         }
28
29         if (mNumberOfAvailableChannels > 0)//check for double id of receiver
30         {
31             --mNumberOfAvailableChannels;
32             messageChannelArray[mNumberOfAvailableChannels].getFilter().
    setRange(0x8000, 0xFFFF);
33             mSoftwareBus.registerChannel(messageChannelArray[
    mNumberOfAvailableChannels]);
34             mReceiversArray[mNumberOfAvailableChannels] = id;
35             return outpost::swb::OperationResult::success;
36         }
37         else
38         {
39             return outpost::swb::OperationResult::invalidState;
40         }
41     }
42
43     //function that returns the correct index mReceiversArray channel number;
     input: Receiver's ID
44     template <typename IDType>
45     int
46     SyncLink<IDType>::findIndex(const IDType id)
47     {
48         for (int i = 0; i < mMaxNumberOfReceivers; i++)
49         {
50             if (mReceiversArray[i] == id)
```

```
51              {
52                  return i;
53              }
54          }
55          return -1;
56      }
57
58      template <typename IDType>
59      outpost::swb::OperationResult
60      SyncLink<IDType>::sendMessage(const IDType id, const outpost::Slice<const
         uint8_t>& data, uint32_t& nrOfReceivers)
61      {
62          //Call checkMessage() and assign the output to a bool variable
63          bool messageIsID = SyncLink<IDType>::checkMessage(id);
64
65          if (messageIsID == true)
66          {
67              mSoftwareBus.sendMessage(mStoredID, data);
68
69              while(nrOfReceivers > 0)
70              {
71                  outpost::swb::OperationResult res = SyncLink<IDType>::
    receiveACK();
72                  if (res == outpost::swb::OperationResult::success)
73                  {
74                      --nrOfReceivers;
75                  }
76                  else if(res == outpost::swb::OperationResult::timeout)
77                  {
78                      return res;
79                  }
80              }
81              return outpost::swb::OperationResult::success;
82          }
83          else
84          {
85              return outpost::swb::OperationResult::sendFailed;
86          }
87      }
88
89      template <typename IDType>
90      bool
91      SyncLink<IDType>::checkMessage(const IDType id)
92      {
```

```
93          IDType tmp = id;
94          /*
95          Check if first bit is not a 0
96          Equation:
97          byte * nr of bits in byte - 1 because starts counting from 0
98          */
99          if ((tmp &= (1 << (sizeof(IDType) * 8 - 1))) != 0)
100         {
101             mStoredID = id;
102             storeAckID(id);
103             return true;
104         }
105         else
106         {
107             return false;
108         }
109     }
110
111     template <typename IDType>
112     void
113     SyncLink<IDType>::storeAckID(const IDType id)
114     {
115         //toggle first bit of id and store it as an ack
116         mStoredAck = id ^ (1 << (sizeof(id)*8 - 1));
117     }
118
119     template <typename IDType>
120     outpost::swb::OperationResult
121     SyncLink<IDType>::receiveACK(outpost::time::Duration timeout)
122     {
123         uint8_t counter = 5;
124
125         while (counter > 0)
126         {
127             outpost::swb::Message<IDType> m;
128             if(acknowledgmentChannel.receiveMessage(m, timeout) == outpost::
    swb::OperationResult::success)
129             {
130                 if(m.id == mStoredAck)
131                 {
132                     if (m.buffer[0] < 5)
133                     {
134                         mHasReceivedMessage[m.buffer[0]] = true;
135                         return outpost::swb::OperationResult::success;
```

```
136                     }
137                     else
138                     {
139                         return outpost::swb::OperationResult::invalidMessage;
140                     }
141                 }
142                 else
143                 {
144                     --counter;
145                 }
146             }
147             else
148             {
149                 break;
150             }
151         }
152         return outpost::swb::OperationResult::timeout;
153     }
154
155     template <typename IDType>
156     outpost::swb::OperationResult
157     SyncLink<IDType>::receiveMessage(outpost::swb::Message<IDType>& message,
    outpost::time::Duration timeout, uint8_t receiverID)
158     {
159
160         outpost::swb::OperationResult res;
161         //call the new findIndex function. Return error message to receiver
    in case it is not true.
162         int channelNumber = findIndex(receiverID);
163         if (channelNumber >= 0 && messageChannelArray[channelNumber].
    receiveMessage(message, timeout) == outpost::swb::OperationResult::
    success)
164         {
165             res = sendACK(message.id, channelNumber);
166         }
167         else
168         {
169             res = outpost::swb::OperationResult::noMessageAvailable;
170         }
171         return res;
172     }
173
174     template <typename IDType>
175     outpost::swb::OperationResult
```

```
176     SyncLink<IDType>::sendACK(const IDType id, uint8_t receiverID)
177     {
178         const IDType ackID =  id ^ (1 << (sizeof(id)*8 - 1));
179         uint8_t buffer[1] = {receiverID};
180         outpost::Slice<uint8_t> slice(buffer);
181         return mSoftwareBus.sendMessage(ackID, slice);
182     }
183
184 } //namespace swb
185 } //namespace outpost
186
187 #endif //SYNCLINK_IMPL
```

Listing B.2: sync_link_impl.h

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

City	Date	Signature