Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Silje Susort

Distributed Real-Time Systems using Lingua Franca and Zephyr RTOS

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
Co-supervisor: Erling Rennemo Jellum
June 2023

NTNU
Norwegian University of
Science and Technology

Silje Susort

Distributed Real-Time Systems using Lingua Franca and Zephyr RTOS

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
Co-supervisor: Erling Rennemo Jellum
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Table of Contents

1 Introduction **1**

1.1 Problem Context 1

1.2 Problem Description 2

1.3 Related Work 3

1.4 Report Outline 3

2 Theory **5**

2.1 Real-Time Systems 5

2.2 Zephyr RTOS 7

2.3 Networking 15

2.4 Distributed Systems 17

2.5 The Reactor Model 21

2.6 Lingua Franca 30

2.7 Other Software 39

2.8 Hardware Platform 40

3 Specifications **41**

3.1 Project Objectives and Scope 41

3.2 Functional Description 41

3.3 Other Goals 45

4 Design **46**

4.1 Hardware Overview 46

4.2 Choice of Zephyr RTOS 46

4.3 Federation Topologies 47

4.4 Zephyr Platform Support for Federates . . . 47
4.5 Zephyr Platform Support for RTI . . . 50
4.6 Adapted Main Function for RTI and Termination . . . 50

5 Implementation **52**
5.1 Project Structure . . . 52
5.2 Custom commands for West . . . 53
5.3 Project Setup . . . 54
5.4 Build and Run . . . 57
5.5 Zephyr Configurations . . . 58
5.6 Zephyr Platform Support for Federate . . . 61
5.7 Zephyr Platform Support for the RTI . . . 63
5.8 Adapted Main Function for RTI . . . 63
5.9 Zephyr Fatal Errors and Causes . . . 64
5.10 Github Contributions . . . 64

6 Evaluation and Results **66**
6.1 Clock Synchronization Test . . . 66
6.2 Message Sending Test . . . 68
6.3 Example: Smart Grid . . . 72

7 Discussion **78**
7.1 Thesis Outcomes and Future Work . . . 78
7.2 Areas of Improvement . . . 80

8 Conclusion **81**

Bibliography **82**

A Joint RTI main function **85**

B List of Acronyms **86**

C Hyperlink Reference **88**

D Presentation to LF team **90**

Abstract

Real-time systems require careful design and specialized hardware, including the use of small and scalable real-time operating systems (RTOS), such as Zephyr RTOS. Real-time systems are often physically distributed by nature or could benefit from a distributed design. Distributed real-time systems offer benefits like redundancy and load balancing, but introduce additional design complexities. Coordination languages, such as Lingua Franca (LF), offer a promising solution for managing interactions between system nodes. LF provides a framework for distributed coordination, but it is currently only available for Linux and MacOS. This thesis addresses this limitation by adapting the framework to Zephyr RTOS, thus allowing use in a range of new applications. Message delays as low as 4 milliseconds are achieved with the decentralized coordination, and initial clock synchronization is found to be in the sub-millisecond order. A case based on self-healing in smart grids is explored, where using logical delays and safe-to-process offsets allow for explicit manipulation of the system's reaction time. The outcomes in this report have been presented to the LF team at the University of California, Berkeley, and contributions have been made to the open-source LF project.

Sammendrag

Sanntidssystemer krever nøye design, spesialisert maskinvare, og ofte bruk av små og skalerbare operativsystemer, som for eksempel Zephyr RTOS. Sanntidssystemer er ofte allerede distribuerte, eller kan ha mye nytte av et distribuert design. Distribuerte sanntidssystemer har fordeler relatert til redundans og balansering av beregningsoppgaver, men introduserer samtidig høyere kompleksitet til designet. Koordinasjonsspråk, som Lingua Franca (LF), kan være en lovende løsning på å håndtere interaksjoner mellom distribuerte noder. LF har et rammeverk for distribuert koordinasjon, men det er for øyeblikket kun tilgjengelig for Linux og MacOS. Dette prosjektet adresserer denne begrensningen ved å tilpasse rammeverket til Zephyr RTOS, noe som gjør det mulig å bruke det til mange nye applikasjoner. Meldingsforsinkelser så korte som 4 millisekunder oppnås med desentralisert koordinering, og initiell klokkesynkronisering er demonstrert til å være under ett millisekund. En demonstrasjon basert på rekonfigurering i smartnett utforskes, hvor bruk av logiske forsinkelser og "safe-to-process" (norsk: trygt-å-prosessere) perioder tillater eksplisitt manipulering av systemets reaksjonstid. Resultatene i denne rapporten er presentert for LF-teamet ved University of California, Berkeley, og bidrag er gjort til LF's åpne kildekode.

Preface

This Master's thesis is the concluding work in the author's 5-year Master's degree programme in Cybernetics and Robotics. The work has been carried out in the spring semester of 2023, under the guidance of Sverre Hendseth and Erling Rennemo Jellum. The thesis is structured as a software report.

There is a small overlap between the work carried out in this thesis with the author's specialization project. Firstly, the results achieved in the specialization project provide part of the motivation for this thesis. Secondly, the following theory overlaps, but is rewritten for this thesis: the reactor model and Lingua Franca (excluding federated execution), real-time systems, real-time operating systems, and GDB debugging.

There is an overlap with the subject TTK8, where a report was written on a comparison of FreeRTOS and Zephyr. Parts of the presentation of Zephyr have been used in this report.

The work on Lingua Franca builds upon the contributions of previous contributors, as it is an open-source repository. Completing this thesis also involved presenting the work to the Lingua Franca team, which resulted in valuable perspectives.

I want to thank my supervisor, Sverre, for his belief in me. I want to thank my co-supervisor, Erling, for his discussions, ideas, insights and proofreading assistance concerning this thesis.

I would also like to thank my boyfriend and life companion, Yngve, for providing emotional support and exceptional "rubber duck debugging" during this journey.

Chapter 1

Introduction

1.1 Problem Context

A real-time system is a system where the timing of the computational output is as significant as the output itself. Real-time systems must adhere to timing constraints, or deadlines, that are dictated by the environment. To be able to consistently reach its timing requirements, a real-time system must be carefully designed. The severity of the real-time constraint will significantly affect the design choices. No matter the severity, real-time systems require specialized hardware such as embedded systems. To provide convenient synchronization mechanisms, resource management, and access to hardware and I/O functions, a specialized type of operating system, a real-time operating system (RTOS), is used. RTOSes are designed to be small, highly configurable, and cater to the specific needs of real-time applications. Zephyr is an open-source RTOS that especially emphasizes connectivity.

As the robustness and reliability of real-time systems are paramount, it is natural to extend the system to encompass several, physically separate nodes. It is then called a distributed real-time system (DRTS). Distributed real-time systems have the possibility of adding redundancy to the design, meaning that even in one node fails, then the system can keep operating. They can dynamically balance the computational load to reliably keep execution times within deadlines. Another aspect is that real-time systems are often distributed by nature, for example by having physically separate measuring units and computational units.

However, distributed systems introduce new design complexities. Nodes must communicate reliably through networks, they must keep sufficiently synchronized local clocks, must sufficiently agree on the system state, and meet temporal real-time requirements. These challenges pose a significant burden on developers and system designers. At the same time, real-time processing across multiple nodes is essential in various industries and applications, especially for safety-critical applications. Everything from monitoring a patient's vital signs to flight control systems, or creating a self-healing electricity grid. DRTS requires cutting-edge solutions from multiple technical disciplines to be employed for new and innovative applications in our society.

Coordination languages offer a promising solution to effectively handle the intricacies of distributed real-time systems. These languages focus on managing the interactions between system nodes rather than the actual computation, enabling developers to design

applications with formal guarantees and intuitive interfaces. Lingua Franca (LF) [1] is an open-source coordination language that provides an implementation of the reactor model [2]. The reactor model combines principles from synchronous-reactive models, discrete-event models, logical execution time models, and the actor model [2].

Xronos [3] is an open-source framework for distributed applications, and provides two types of predictable coordination based on PTIDES [4], [5] and High-Level Architecture (HLA) [6]. Xronos is developed on top of the LF coordination language. While parts of LF are ported to FreeRTOS [7] and Zephyr [8], the Xronos framework has not been ported to either. In fact, Xronos is currently only available for Linux and MacOS and has not been run on resource-constrained systems before. This thesis aims to confront this limitation by adapting the framework to Zephyr and will provide some initial results from testing the timing properties of the implementation.

1.2 Problem Description

The framework provided by Xronos [3] will be extended to the RTOS domain to allow coordination of distributed real-time systems. As Lingua Franca handles time explicitly, this could be a major advantage for handling real-time constraints [2]. A number of interesting implementations in the framework let the developer directly influence various trade-offs in distributed system design.

As the Xronos framework, also called federated LF [2], has not been run on resource-constrained embedded systems or real-time operating systems before, this thesis aims to experimentally implement this for the Zephyr platform. To do that, the LF runtime must be modified. A goal of the thesis is to contribute to the open-source Lingua Franca project. Additionally, it is of interest to find out how to set up a multi-platform federation, and to sufficiently document it to ease further work.

The central questions to answer are:

- How viable is federated LF on Zephyr RTOS for the design and implementation of distributed real-time systems?
- What are the strengths and weaknesses of this approach?

In order to investigate the potential of distributed real-time system development using federated Lingua Franca, some central questions need to be answered:

- What is the achievable clock synchronization error over a LAN network using LF's own clock synchronization? Since there is no underlying synchronization from the network, the implementation is dependent on LF's own synchronization scheme
- What is the overhead of sending a message using different coordination schemes? The coordination schemes use a different amount of system resources and could have interesting effects on inherently resource-constrained platforms.
- What is the maximum possible message throughput that the distributed system can handle correctly in the different coordination schemes? This measure would depend on the structure of the distributed system, of course, but a baseline measure would be ideal for future considerations.

- How fast and how reliable can a federation react to environment input? How do different timing primitives affect these measures? To investigate this, a demonstrative example based on self-healing in Smart Grid systems will be explored.

1.3 Related Work

Robot Operating System (ROS 1) [9] is a popular, open-source middleware for robotics systems, usually distributed, and is based on the publish-subscribe coordination mechanism. The ROS master sets up peer-to-peer communication between nodes, and nodes can publish or subscribe to topics [10]. The release of ROS 2 brings many new features, among them micro-ROS [11] which aim to bridge the gap between resource-constrained embedded devices and larger processors using ROS. Micro-ROS features several optimizations for embedded systems, including a new client type, additional drivers and uses a POSIX-supporting Real-Time Operating System (RTOS), such as Zephyr, FreeRTOS or NuttX [11].

MQ Telemetry Transport (MQTT) [12] is a popular messaging protocol for Internet of Things (IoT) applications, and is also based on publish-subscribe coordination. The sender and receiver are decoupled from one another by communicating through topics. The MQTT broker holds session data, topology information, authorization of MQTT clients (which are devices running the MQTT library), receives all messages and relays them to the correct recipients [13]. Both Zephyr and FreeRTOS provide MQTT client library implementations [14], [15].

The pub-sub coordination common to ROS and MQTT produce non-deterministic inconsistencies in message ordering when published and observed on multiple topics [3]. These inherent problems lead to additional, error-prone application logic, and it is arguably the underlying model that should ensure deterministic coordination [3].

Timed C [16] is an extension to the C programming language providing simple timing primitives aimed at real-time systems. [16] argues that building real-time systems on RTOS APIs makes the code error-prone, complex, and less portable, and that it is better to have the programming language itself support simple timing primitives. Timed C is suitable for non-distributed, resource-constrained embedded systems, and can generate code for FreeRTOS and POSIX-compliant systems. Timed C is not suitable for distributed systems, as no network transport is included [17].

Federated Timed C [17] extends Timed C with deterministic network channels in a reference implementation, such that Timed C can be used to create robust distributed real-time systems. Time Sensitive Networking (TSN) [18], which is a networking technology providing time-triggered networking over Ethernet, is used to provide strict timing guarantees, such as bounding the end-to-end latency. For hard real-time systems, these guarantees are critical. A weakness of ROS, MQTT and Xronos is that they are based on TCP and UDP [10],[13],[3], which cannot guarantee latencies.

1.4 Report Outline

The report is outlined as follows. Section 2 presents the relevant theory related to the thesis. This section introduces real-time systems and presents a real-time operating system

called Zephyr. Basic networking theory is introduced. Important problems in distributed systems are outlined. A deterministic and concurrent model of computation called the reactor model is presented, followed by an implementation of it as a coordination language called Lingua Franca. Other software used in the project and the hardware platform are presented at the end. Section 3 covers the specifications of the project, presenting both functional descriptions and some further goals. Section 4 covers important design considerations in the project solution. Section 5 covers how the project is structured, how it should be set up, and how it was implemented. Section 6 presents four tests measuring some important aspects of the implementation, as well as a study of a larger example using the implementation. Section 7 discusses the thesis results in a wider sense, and project improvements and future work is presented.

Chapter 2

Theory

This chapter covers the necessary theory and background for the thesis. The chapter will start by covering two classes of systems central to this thesis, namely real-time systems and distributed systems. The real-time operating system called Zephyr will be presented. Next, a deterministic model of computation called the reactor model, and its implementation in the coordination language called Lingua Franca will be given a section each. The section about the reactor model focuses on the model properties, while the chapter about the language focuses on syntax and how it is implemented using a runtime. Next, networking concepts are presented. Finally, some information about the software and hardware used in the thesis will be presented.

2.1 Real-Time Systems

Real-time systems are computer systems characterized by finite time constraints from the environment [19]. The time constraints are called deadlines. The correctness of a real-time system is not only based on the calculations, but also on *when* the calculations arrive.

A consequence of having to meet deadlines is that the worst-case execution times (WCET) are more important than the best-case and average-case execution times in these systems. This is contrary to general-purpose computing and CPU design [20], where maximizing the throughput of the system is the most important consideration. Therefore, real-time systems often require dedicated hardware systems and special real-time operating systems [19]. These elements give higher predictability and reliability to the overall system, making guarantees possible.

The importance of reaching a deadline depends on the severity of the real-time constraint. There are commonly referred to as three types of constraints, classified by their severity level: soft, firm and hard real-time constraint [19]. A real-time system may have one or several real-time constraints. If there are constraints of several types, then it is called a mixed-criticality real-time system. The least severe constraint is visualized as the leftmost graph in Figure 2.1. The value of the result will decrease the further it is from the deadline. An example of such a system could be a measurement system where the older the measurements are, the less value they have. A more severe constraint is a firm real-time constraint, shown in the middle graph in Figure 2.1. If the deadline is violated, then the result is not useful anymore. An example of such a system is a classification system on an assembly line. If the classification comes too late, then the item may have already

passed, and the result is useless. The rightmost graph in Figure 2.1 shows a hard real-time constraint, which is characterized by system failure if the deadline is violated. In the most severe case, damage to people and property could be caused. An example of a system with a hard real-time constraint is an automatic brake system on a car. If the brake system fails, then the car could potentially crash.

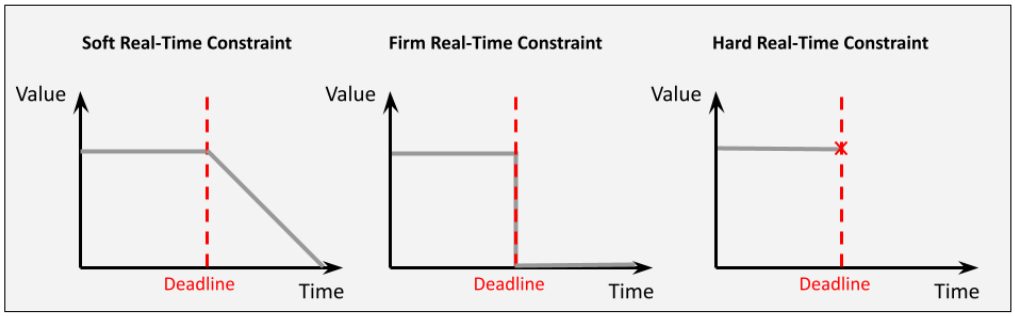


Figure 2.1: The figure shows the common classifications of real-time constraints.

2.1.1 Real-Time Operating Systems

Real-Time Operating Systems (RTOS) are small and scalable operating systems characterized by a focus on time predictability. RTOSes are typically used with embedded systems [19]. The main functions of an RTOS are to provide convenient access to hardware, efficient resource management and input/output functions [19]. The RTOS kernel is configurable such that specific application needs can be met [19]. Real-time operating systems are not only for real-time systems, but also for embedded systems of a certain complexity [21]. RTOSes provide concurrency interfaces, timing APIs, and possibly network stacks.

Tasks are the computational entities executed by the processor, and are also called processes or threads [21]. A task consists of instructions, data and task administration information needed by the operating system [22]. Operating systems have task models which determine which states a task can be in [22]. The most basic task model is simply if it is currently running or not running. Usually, the task model will also have states for a ready state, blocked or waiting state, states related to memory swapping and more.

Scheduling is a crucial process in any operating system, as it is the act of deciding which tasks should run on the processor over time. Scheduling can refer to three types of scheduling; short-term, medium-term and long-term scheduling [22]. Short-term scheduling is deciding the best task to run at any time. Different scheduling strategies optimize different aspects of the behavior. A (short-term) scheduler can be preemptive or non-preemptive [19]. A non-preemptive scheduler will let an already running task complete, or the task will voluntarily yield, before switching to another task. A preemptive scheduler can switch tasks even before they have finished. The two scheduling strategies are shown in Figure 2.2. Notice when the tasks are ready to run and when they finish in the two strategies.

Memory management is a complex task in an operating system. Memory management concerns how to allocate and share memory for tasks. Memory placement algorithms have consequences for the time overhead of allocating blocks of memory, but also for memory allocation determinism and the risk of memory fragmentation [22]. Memory fragmentation is when several blocks of memory are too segmented to use contiguously. Thus large quantities of memory may become unusable. There are strategies to mitigate

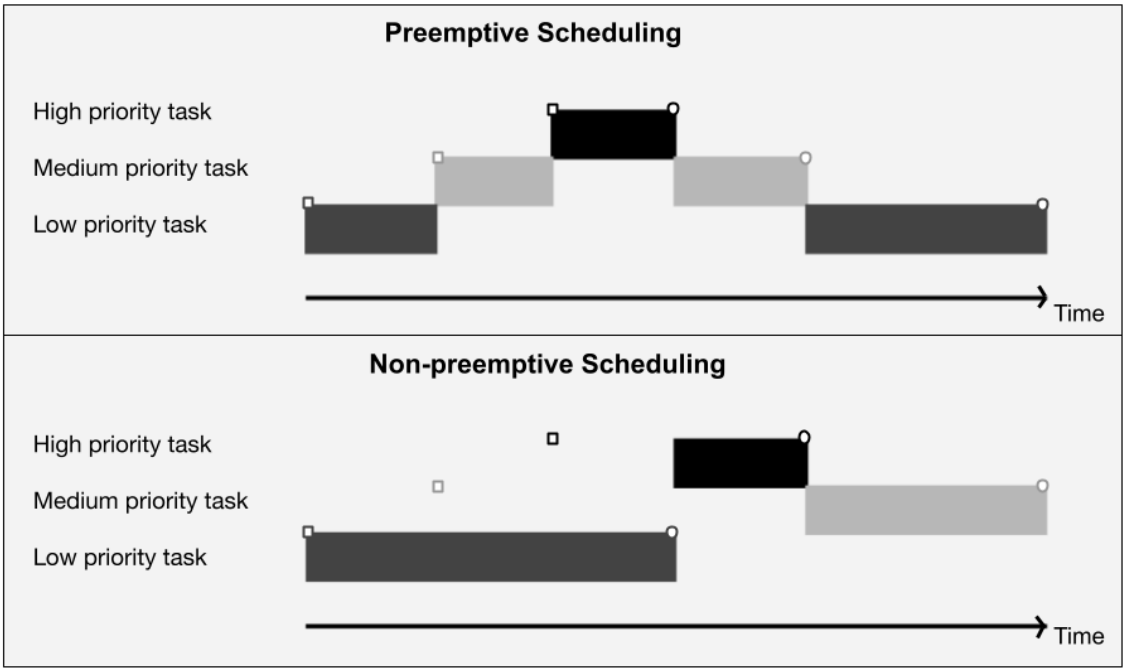


Figure 2.2: The figure shows the two types of short-term scheduling. The squares signify the time the task is ready to run, and the circles when they are finished processing.

memory fragmentation, but in a resource-constrained system, it may be time-consuming [22].

2.2 Zephyr RTOS

The Zephyr Project was launched in 2016 with the goal as described on their webpage:

> "The Zephyr Project strives to deliver the best-in-class RTOS for connected resource-constrained devices, built to be secure and safe."

In other words, Zephyr is an embedded RTOS that specifically focuses on connectivity, security and safety. These points match well with Internet of Things requirements and technology, and make Zephyr particularly attractive to these applications.

The Linux Foundation is behind the open source project, but it also has several important semiconductor and other companies as members [23]. The members engage and contribute to the project. It has an Apache 2.0 license, which is a permissive free software license, giving few conditions for use, modification and reuse.

Zephyr supports a wide range of boards, see [24].

Most of the information in the following sections is collected from the Zephyr Kernel Documentation pages [25] and the Github page [26].

Figure 2.3: The figure shows the Zephyr logo. Found via Wikimedia Commons, Zephyr Project, Apache License 2.0.

2.2.1 Structure

Zephyr RTOS on Github [26] is a large project containing the kernel, ports, drivers, libraries, tests, examples, documentation, protocol stacks, file systems and more. The kernel folder itself contains 37 source files in C, where 10 of those are included in any build. The 10 main kernel files are: main_weak.c, banner.c, device.c, errno.c, fatal.c, init.c, kheap.c, mem_slab.c, thread.c and version.c. 11 more files will be included if multithreading is enabled. The rest of the kernel files correspond to certain configurations. Ports to specific boards and peripheral drivers are not part of the kernel, but part of the operating system. A part of the structure of Zephyr is summarized in Figure 2.4.

2.2.2 Setup and Configuration

Installation of Zephyr is done by installing all dependencies, downloading and setting up the Zephyr SDK, installing West and initializing the Zephyr directory. More on this process in the Zephyr Install Documentation page [27]. Note the minimum required versions of the dependencies. Creating a Python virtual environment for the installation is recommended to avoid dependency incompatibilities.

The Zephyr SDK contains a range of samples that can be run with a single command if the given board is supported. All necessary configurations are contained in the project configuration file, as well as in the configuration files associated with the board. Build a basic sample using the following command:

```
$ west build -b <BOARD> samples/HelloWorld
```

Zephyr provides an advanced configuration system, both for software properties (Kconfig) and hardware (Devicetree), together with a custom build tool (west). These systems make the integration of submodules simple and provide hardware independence.

Kconfig
Kconfig is a configuration system consisting of Kconfig files distributed throughout the directory tree. The relevant files for a project are then merged. Each Kconfig file contains some configuration options. The total number of configuration options is very large, so it is a good idea to look up sample configurations, and search the individual options and their dependencies at Kconfig Search.

The Kconfig system is used by the Linux kernel [28]. Both application-specific configuration and hardware-specific configuration are handled by Kconfig files. Notice how Kconfig

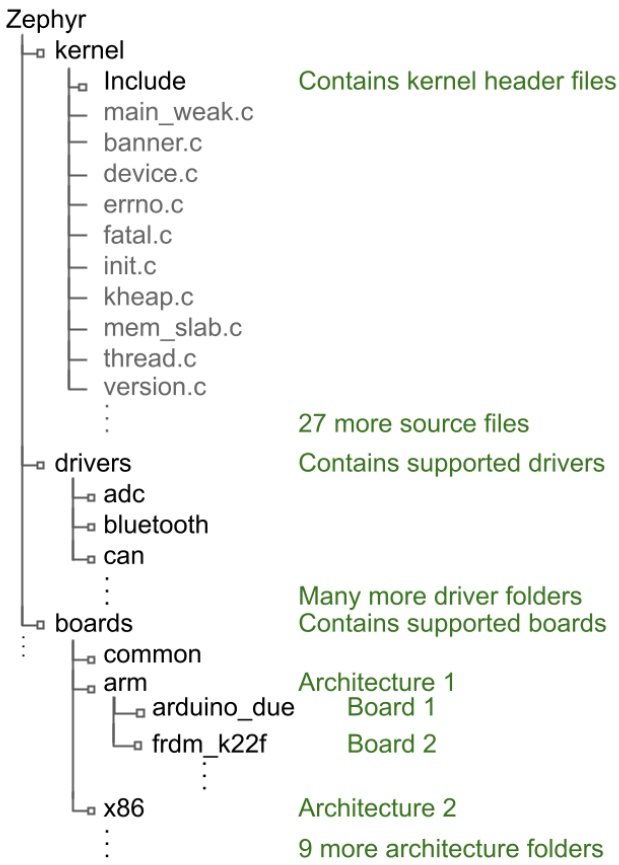


Figure 2.4: The figure shows part of the structure of the Zephyr RTOS. Note that the figure is not at all complete and contains much more than is shown.

uses DeviceTree output in Figure 2.5. Application-specific configurations are made in a file typically called prj.conf in the project folder.

DeviceTree
DeviceTree is a hierarchical hardware configuration system used with Zephyr. Figure 2.6 shows an example of the hierarchical structuring of the hardware for a LED. The DeviceTree sources, shown in Figure 2.5, consist of a main DeviceTree file, which specifies the available features on the hardware and its initial configuration. It also possibly consists of overlay files that can override the main DeviceTree source file. DeviceTree bindings specify rules for the contents of the DeviceTree, such as structure, possible values, etc. All this information is then used to produce a final configuration file, *zephyr.dts*, and generated C headers.

West
West is a command-line tool developed for the Zephyr project. It is a convenience tool developed for the most common workflows when developing; repository management, building and flashing. It is also possible to create custom commands. One of the main motivations behind developing a new tool (as opposed to using some existing repository management tool) was to be able to separate externally maintained repositories from the main Zephyr repository.

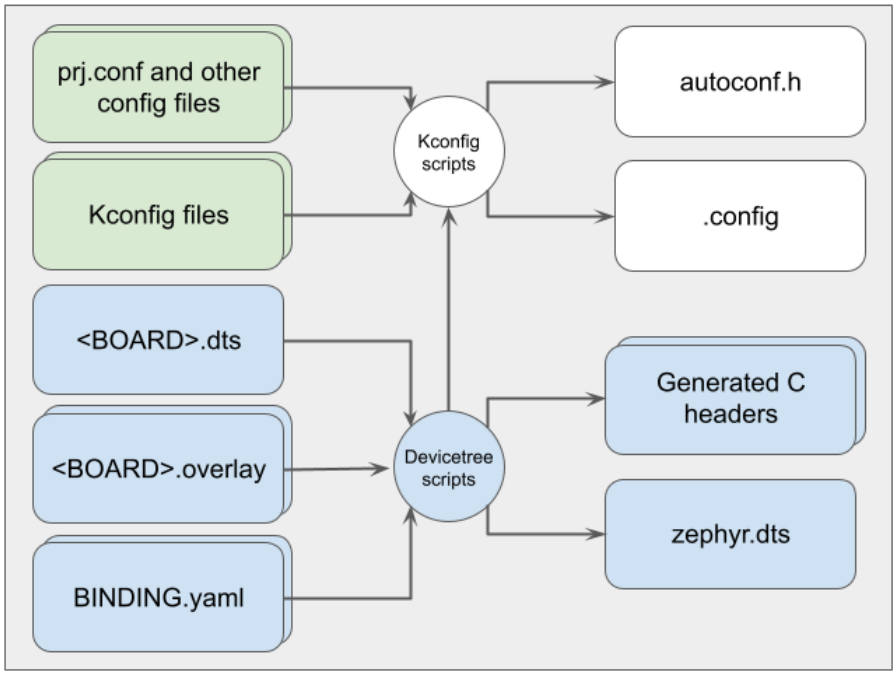


Figure 2.5: The figure shows the input configuration files for DeviceTree and Kconfig, and how they generate output files.

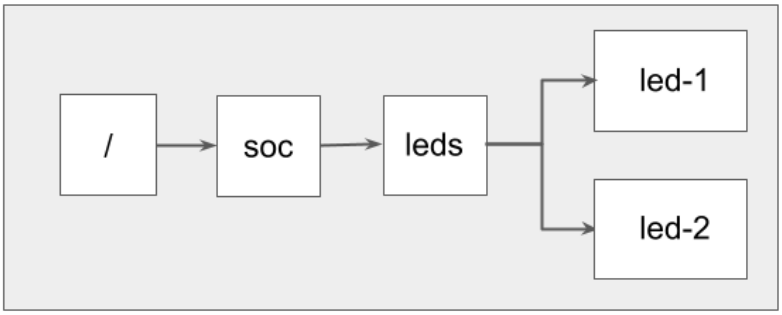


Figure 2.6: The figure shows the DeviceTree hardware hierarchy with a LED example.

2.2.3 Task Management

Tasks
In Zephyr, tasks are called threads, but they are the same concept.

A thread will typically run infinitely, but Zephyr also allows for the thread to terminate on its own. It is then required that it releases any shared resources before doing so. To avoid race conditions, program code should use synchronization mechanisms, like k_thread_join() or k_thread_abort().

A thread can be either ready or unready to execute. If the thread is ready, it may be selected to execute by the scheduler. If the thread is unready, it cannot be selected. A thread can be unready for a variety of reasons. It may not have started, waiting for synchronization or time-related events, suspended, terminated, or aborted. An overview of the thread state structure is shown in Figure 2.7.

In Zephyr, a thread has a given priority. The lower the numerical value, the higher the

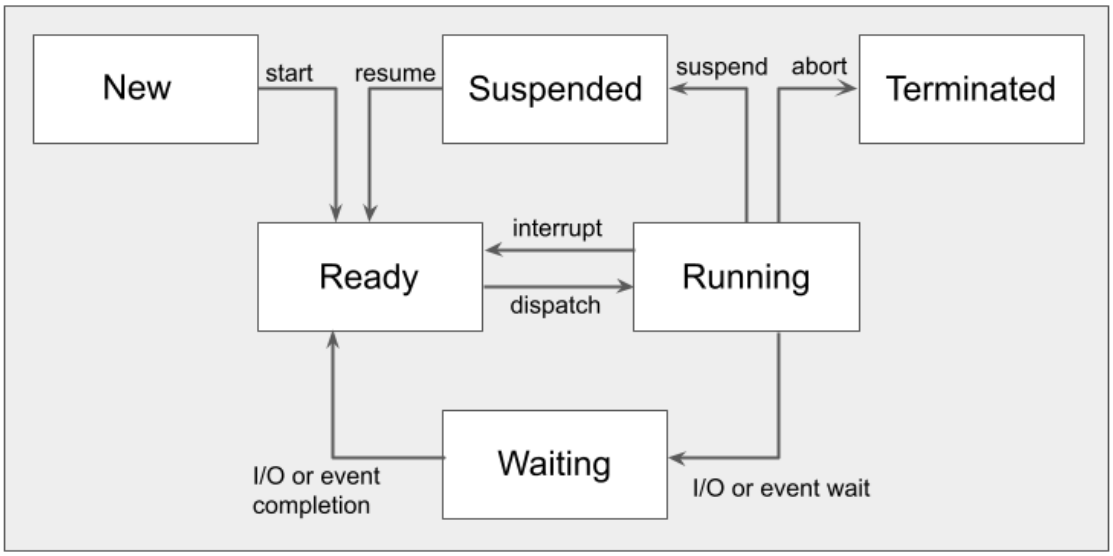


Figure 2.7: The figure shows the thread states in Zephyr. Based on [25]

.

priority. The priority is set at thread creation, but may also be changed during runtime. If the priority value is negative, then the scheduler cannot preempt the thread; it may only be cooperative. If the priority value is positive; it may be preempted. Thus, two classes of threads are constructed; cooperative and preemptible threads. A cooperative thread always has precedence over any pre-emptive threads. The number of priority levels in both categories may be configured.

Preemptive threads are typically used for prioritizing more or less time-sensitive tasks, while cooperative threads are typically used for work that is critical for performance, or for automatic mutual exclusion.

In Zephyr, there are automatically created threads called system threads. The Main thread is a system thread that is responsible for system initialization and for executing the main function. The idle thread is a thread that executes when no other thread can, ie. it has the lowest possible priority and will yield as much as possible. It either activates power-saving mechanisms or simply busy waits. If the idle thread aborts, the system will raise a fatal system error.

Scheduling
The scheduler in Zephyr decides which of the tasks in the ready state will execute at any given time. As indicated above, the scheduler uses a task's numerical priority value and whether it is preemptive or cooperative to decide which thread should be the current thread. The current thread may only be switched at certain rescheduling points; when a new thread becomes ready, the current thread yields execution, the current thread is suspended or waiting, or after processing an interrupt.

The scheduler can be configured to use time slicing, ie. partitioning time into a series of slices with a certain length [25]. Cooperative time slicing is also possible, where a cooperative thread voluntarily calls k_yield() or k_sleep(). If the thread calls k_yield(), then the scheduler will check if any threads of equal or higher priority can run. A preemptive thread may temporarily become a cooperative thread by calling k_sched_lock(), if it needs to avoid being preempted. When it can be preempted again, it must call k_sched_unlock().

The scheduler can be configured to use different implementations for the ready queue, which will affect overhead and context switch speed.

Interrupt Service Routines (ISR) will always interrupt tasks, no matter the task priority or class. That is, unless interrupts have been temporarily disabled.

2.2.4 Memory management

Zephyr application memory may be completely statically allocated at compile time, or may use dynamic allocation. Zephyr allows the use of several different types of heaps in the same configuration. They have different properties.

k_heap is a heap allocator that uses kernel synchronization tools. It may allocate and deallocate memory in a way that is very similar to the C standard malloc() and free().

The different heap types can be summarized in the following Table 2.1.

Name	Deterministic?	Heap size	Allocation Algorithm	Fragmentation
No Heap	Yes	-	-	Not possible
k_heap	No	Configurable	best fit (buckets)	Possible
system heap	No	Configurable	best fit (buckets)	Possible
Memory slabs	No	Configurable	first-fit (assumption)	Not possible

Table 2.1: The table shows (a subset of) the different dynamic allocation options in Zephyr.

2.2.5 Inter-process communication methods

Table 2.2, collected from Zephyr's kernel documentation pages [25], summarizes the different inter-task and ISR communication methods.

Object	Bi-directional?	Data Structure	Data item size	Data Alignment	ISRs can receive/ send?	Overrun handling
FIFO	No	Queue	Arbitrary	4 B	Yes	N/A
LIFO	No	Queue	Arbitrary	4 B	Yes	N/A
Stack	No	Array	Word	Word	Yes	Undefined behaviour
Message queue	No	Ring Buffer	Power of two	Power of two	Yes	Pend thread or return -errno
Mailbox	Yes	Queue	Arbitrary	Arbitrary	No	N/A
Pipe	No	Ring Buffer	Arbitrary	Arbitrary	Yes	Pend thread or return -errno

Table 2.2: The table shows different data passing structures in Zephyr, and their characteristics.

A **queue** is a data structure that implements ordered data storage, which can be accessed by both tasks and ISR. It is implemented as a linked list. A **FIFO** queue is a queue data structure that has the limiting functionality of being a First In First Out order. The data elements can be of any size. Multiple items may be added at once if they are chained together. A **LIFO** queue is a queue data structure that has the limiting functionality of being a Last In First Out order. The data elements can be of any size. Multiple items may be added at once if they are chained together. A **stack** is a LIFO queue with limitations on the data elements. A **message queue** is a data structure that allows for asynchronous data passing. The data element that is sent by a thread is copied into the message queue. If it is full, then the thread has the option to wait. A **mailbox** is a message queue with extended functionality. It can use both asynchronous and synchronous communication. A **pipe** is a synchronous process-to-process data passing method in which streams of bytes are sent. It can be configured to use a buffer that holds the data.

Synchronization structures like semaphores and mutexes are implemented. A **condition variable** is a synchronization structure that enables threads to wait until some condition is true. Several threads may wait on the same condition variable. It can be used to signal changing states in threads.

2.2.6 Time Utilities

Zephyr provides different ways to access time, differing in unit and precision. The standard way is to provide real-time values, in milliseconds or microseconds, which are easy to handle by the application, but not as precise. To get more precise values, the application designer can poll the kernel cycle count at k_cycle_get_32(). The kernel also uses ticks for interrupt handling and internal bookkeeping. All these different values may be easily converted to the others.

Timers
A Zephyr timer uses the system clock to measure time and generate interrupts as configured by the user. When the timer generates an interrupt, a custom expiry function will execute (or can be set simply to NULL). The timer first expires after a set duration, and then every period duration. The period may be set to the macros K_NO_WAIT or K_FOREVER to get a one-shot timer instead of a periodic one. A timer has a stop function in addition to the expiry function, that will execute when the timer has been stopped too early. There is no set limit on the number of available timers.

2.2.7 Networking

Zephyr comes with a highly configurable, native network stack [29]. The stack is divided into layers as follows:

- **Network Application** provides application-level protocol APIs and configuration options for the related network parameters. A BSD socket implementation is available to directly interface with connections.
- **Network Protocols** provide implementations of network protocols. Application-level protocols like CoAP and MQTT are implemented. Communication protocols like TCP, UDP, IPv4 and IPv6 are implemented.

- **Network Interface Abstraction** provides the basic network infrastructure, needed by all protocols.
- **L2 Network Technologies** provides the underlying data handling from network devices. This includes technologies like Ethernet, Bluetooth and IEEE 802.15.4.
- **Network Device Drivers** provides the low-level drivers.

2.2.8 POSIX API

Zephyr offers a partially implemented POSIX API. This API is available under the configuration option *CONFIG_USE_POSIX_API*. It was developed with the intention of easy integration with other libraries, and to simplify porting POSIX applications to Zephyr. The POSIX API is under development, and is not complete. Therefore some functions may not be implemented.

2.3 Networking

This section provides an overview of key networking concepts needed for the thesis.

2.3.1 TCP/IP Protocol Suite

The TCP/IP protocol suite is a model organizing computer networks into five layers. As this is assumed well-known by the reader, it is only briefly covered.

Layers
The *physical* and *data link* layers do not define specific protocols, and support all standard and proprietary protocols [30]. The *network* layer supports the Internet Protocol (IP) versions 4 and 6 (IPv4 and IPv6). The *transport* layer supports User Datagram Protocol (UDP), Transmission Control Protocol (TCP) and Stream Control Transmission Protocol (SCTP). The *application* layer supports several protocols related to different application aspects, such as file transfer, hypertext transfer and so on.

Addresses
The TCP/IP model defines four types of addresses, belonging to four of the different layers [30]. The data link layer has physical addresses, the network layer has logical addresses, the transport layer has port addresses and the application layer has application addresses. The physical address is a 48-bit number that must uniquely identify a node on the network and is commonly referred to as the Media Access Control (MAC) address for ethernet [30]. The logical address for IPv4 is a 32-bit number uniquely identifying a node on the network, and commonly referred to as the IP address [30]. The port address identifies the process to communicate with and is a 16-bit number [30].

TCP and UDP
TCP and UDP both provide process-to-process communication. UDP is a very lightweight transport protocol that offers fast, unreliable and connectionless communication [30]. It is suitable for applications where speed is prioritized over error-free delivery. TCP is a much more advanced protocol than UDP, and ensures data delivery in the correct order, without loss, duplication, or corruption [30]. TCP is a connection-based protocol, meaning TCP establishes a virtual connection between the processes, and exchange data in both directions.

Ethernet
Ethernet is a protocol for the hardware and data link layer. It is widely used in Local Area Networks (LANs), and can provide 10 Mbps, 100 Mbps, 1 Gbps and 10 Gbps transfer rates [30].

2.3.2 Network Switch

A network switch enables connections between devices in a network, and can either be a two-layer or three-layer switch, depending on which layers in the TCP/IP stack it operates. A two-layer switch uses the physical and data link layer of the TCP/IP stack only, thus enabling faster forwarding [30].

2.3.3 BSD Sockets

Berkeley Software Distribution (BSD) Sockets are an API for process communication. A socket is a communication endpoint and must be bound to a local IP address and port address. The use of a port address enables a device to have several communicating sockets, for example for different local processes sending and receiving at the same time, or for loopback communication. Typically, several underlying network transport protocols are implemented, like UDP and TCP, and are both available for the application.

BSD Sockets are based on the client-server paradigm. The server will be listening for incoming connections and may be connected to several clients. A client socket will try to connect to a single server. When connected, the communication may be initiated either way. In applications, sockets are represented as file descriptors that may be read from or written to.

A state machine representation of the BSD socket model is shown in Figure 2.8. The transitions correspond to API calls, explained in table 2.3. The sockets need to be created and bound to a local address, meaning an IP address and port combination. The server address needs to be explicitly specified, while this is optional for the client. If not specified, the client will be automatically assigned one by the kernel. When the client is ready, it can actively try to connect. If the server is listening for connections, it can also accept connections from clients. A three-way handshake is made to establish a connection. After the sockets are connected, data may be sent over the connection.

The socket connection may be temporarily disabled in one or both ways (ie. receiving or sending direction). It can also be closed permanently. It is a good idea to disable the connection before destroying the socket to ensure a clean shutdown. This has to do with messages already sent being handled correctly.

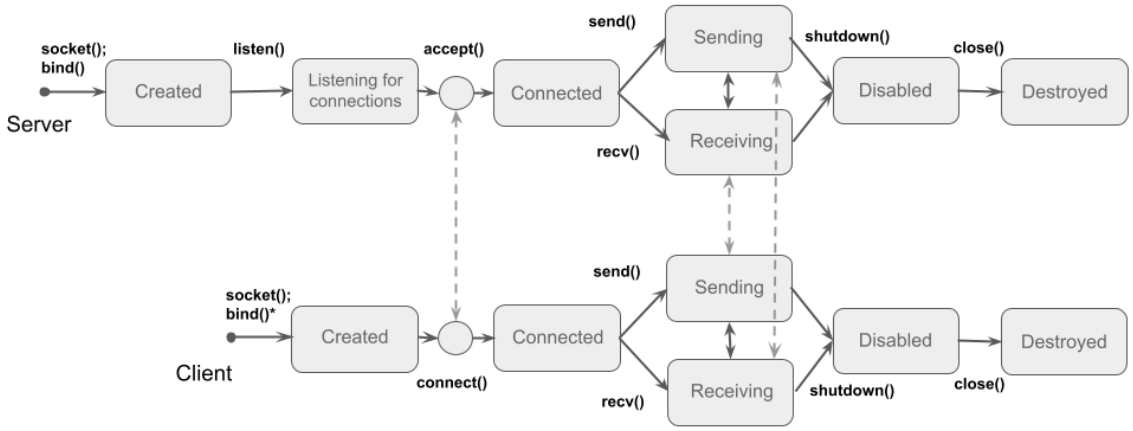


Figure 2.8: The figure shows a state diagram of socket creation, communication and destruction. Some possible transitions are left out for simplicity. The client can optionally be bound to a specific port, but will otherwise be automatically assigned one by the kernel. The transitions correspond to API function calls.

API Function	Used by	Description
socket()	Server/Client	Create a new socket
bind()	Server/Client	Bind a socket to a specified local address (IP:port)
listen()	Server	Listen for incoming connection requests
connect()	Client	Attempt to establish a connection
accept()	Server	Accept a connection
send()	Server/Client	Send data over socket connection
recv()	Server/Client	Receive data over socket connection
shutdown()	Server/Client	Disable connection in some direction or both
close()	Server/Client	Destroy a socket

Table 2.3: The table shows the meaning of each mentioned API call, and if it is used on the client- or server-side.

2.4 Distributed Systems

A system is said to be *distributed* if message transmission delay is non-negligible compared to the time between events in the system [31]. This definition includes, and most often refers to, systems with physically separate nodes. A number of important considerations and nuances must be handled when dealing with distributed systems. As Schwarz and Mattern (1994) [32] puts it:

> *"It seems that implementing distributed programs is still an art rather than an engineering issue; understanding the behavior of a distributed program remains a challenge."*

This section will present some of these aspects.

2.4.1 Global Ordering of Events

Distributed nodes should agree on the system state. This is called *consistency* [33]. Consistency in a distributed system is difficult since causal and temporal relationships between events are not always clear, and this results in nondeterministic [32] and hard-to-understand execution sequences.

In a non-distributed system, the concepts of before and after are straightforward. Since any two time-tagged events share a clock reference, it is easy to establish which one was first (given enough time precision). There will always be a globally correct order of events.

The ordering of events in a distributed system is conceptually different. If physical time is to be used to time-tag the events, then each node must use its available physical clock. The system clocks can be synchronized closely, but will always have some synchronization error. Physical time must be measured, and is therefore not an available "true" reference. In this situation, it can be fundamentally impossible to tell which of two events truly happened before another [34]. This is because the order of such events depends on the frame of reference, as in the theory of special relativity. Distributed systems can therefore have genuinely ambiguous ordering of events.

Consequently, a globally correct order of events based on physical time may not exist for distributed systems. [31] proposed a method to define a globally correct order using

logical clocks. Firstly, the system clocks must be sufficiently synchronized. A logical time granularity can be based on the maximum allowable synchronization error. This logical time may then be used to assign event time tags. A semantic global order can then be defined, even for truly ambiguous events [31]. Simultaneity is then semantically possible as well, and how to handle this will depend on the application.

Another consequence of this method is that better application time granularity is possible with better clock synchronization [31].

2.4.2 Clock Synchronization

Some level of clock synchronization is required for distributed systems. Often it is required that the clocks be synchronized within some bound. Different synchronization algorithms and protocols have different achievable performances.

The **Network Time Protocol (NTP)** is a popular synchronization protocol where a trusted time server synchronizes with clients. It is a hierarchical structure where the first levels are reference clocks, such as national standard clocks or atomic clocks, and each level synchronizes with the level below [35]. A survey of NTP in use on the internet showed that most NTP-synchronized clocks were within 20 - 30 milliseconds of each other [35]. Better precision is achievable within a Local Area Network (LAN) [35].

The NTP synchronization is implemented such that each client sends periodic timestamp requests to the most suitable server. Which server to synchronize with is decided by algorithms evaluating the level and synchronization distance. The server will then respond with its timestamp.

The **Precision Time Protocol (PTP)**, standardized as IEEE 1588 [35], is a high-precision synchronization protocol initially developed in the automation industry. As with NTP, PTP uses a hierarchical structure. At the top of the hierarchy is a GPS receiver, a high-precision local oscillator, or an atomic clock. This clock then sends multicast messages to the next level, while the lower level responds back with unicast messages.

PTP can be implemented purely in software, but also hardware-assisted software [35]. PTP has been shown to deliver sub-millisecond synchronization on LANs (software implementation), and sub-microsecond synchronization with hardware implementations [35].

The basic idea is that the clock error e can be estimated by finding the round trip delay [35]. The error can be estimated with

$$\tilde{e} = ST3 - MT3 - d \tag{2.1}$$

where d is the one-way delay defined as

$$d = \frac{(MT2 - MT1) - (ST2 - ST1)}{2} \tag{2.2}$$

Where the timestamps from the slave's local clock are $ST1$, $ST2$ and $ST3$, and from the master's clock are $MT1$, $MT2$ and $MT3$. A synchronization round is shown in Figure 2.9. $MT1$ is the first timestamp sent from the master clock. $ST1$ is the timestamp of the arrival of $MT1$ at the slave's local clock. The reply is sent at timestamp $ST2$ and received

at the timestamp $MT2$ from the master clock. A last message is sent at $MT3$ to the slave, and the error can be estimated as in Equation 2.1. If the communication latency is exactly symmetric, the error will be exactly determined. The real implementation of PTP is more involved than described here, see [35] for more details.

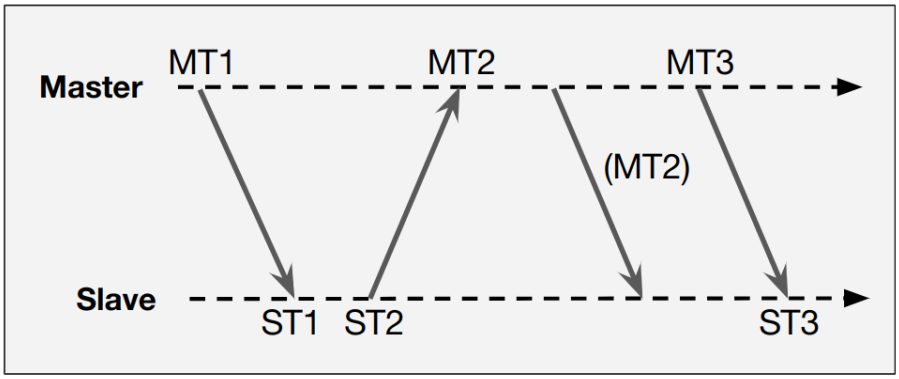


Figure 2.9: The figure shows the conceptual exchange of messages in PTP synchronization.

2.4.3 Consistency/Availability Tradeoff

Hosts that do not share memory must keep individual copies of system state information. Any change to that state must be propagated through the network until all nodes have updated their copy. During this time, the state information is inconsistent throughout the system. If the system must have consistent state information at all times, then the system must be unavailable during this time. Alternatively, if the system must be available at all times, then the system must sacrifice consistency. Thus, there exists a fundamental trade-off between consistency in the system and availability. The CAP theorem, introduced in [33], formulated the trade-off as that one can only have two of the three following traits in a shared-data system:

- Consistency: Nodes agree on the value of a shared variable
- Availability: The experienced latency/response time for a user
- Network Partitioning: Tolerance to communication failure between nodes

However, it has been pointed out that network partitioning is not a binary property. Network partitioning can be seen as an extreme case of network latency. It has been observed that the consistency/availability trade-off is present to some degree for network latency in general. Lee et al. [34] has formalized the CAL theorem (L for Latency), building further on the CAP theorem and the aforementioned observation. Lee et al. [34] observes that both clock synchronization and execution times have the same effect on the consistency/availability tradeoff as network latency, and are thus all under the term "latency". Lee et al. [34] expands the theorem to include latency instead of network partitioning, and thus consistency and availability become non-binary properties as well.

- Latency: The total latency from clock synchronization error, execution times and network latencies

The CAP theorem is an edge-case of this theorem when latencies become unbounded. The reader is referred to Chapter 3 of [34] for the formal theorem and definitions.

2.4.4 Coordination Strategies

Coordination strategies emphasize either consistency or availability. A distributed system for simulation would prioritize consistency over availability since the most important thing is for the calculations to be correct in each node [34]. A distributed control system would prioritize availability since it is more important for nodes to keep operating compared to agreeing on the system state.

High Level Architecture (HLA) is a centralized coordinator scheme, originally designed for distributed simulations by the US Department of Defense [6]. HLA is now a fully developed standard specification [6]. In a distributed simulation, it is more important with consistency in results (ie. getting the same result if running on a distributed system and a non-distributed system) than with availability.

A key property of the HLA architecture is the separation between computation (simulations) and coordination (interoperability among simulations) [6]. All computation is handled in modular components called *federates*. Little restrictions are put on what is represented in the federates or how, only that objects must be able to interact with objects in other federates. The coordination between federates is handled by the RunTime Infrastructure (RTI). A runtime interface provides specifications for the interactions between RTI and federates.

PTIDES (Programming Temporally Integrated Distributed Embedded Systems) [4] is a decentralized coordination scheme that is based on exploiting explicit assumptions on the communication latency to specify a *safe to process event* time offset. **Google Spanner** [36] developed a very similar strategy independently. The safe-to-process offset allows for optimistic coordination that opens up for increased fault tolerance and concurrent execution, without the use of rollback or null messages. There is only a need for null messages in certain circumstances [4].

The key property of the coordination scheme is the safe-to-process offset. It is the physical time it takes an entity to decide to send a message until it is received by the recipient. This would include a bound on communication latency, including network overhead, a bound on the clock synchronization error and the execution lag [2].

2.5 The Reactor Model

This section covers a deterministic concurrent model of computation called the reactor model, with a special focus on how it handles the coordination of distributed systems.

2.5.1 Overview

The reactor model, introduced in [2], is a model of computation providing deterministic coordination for cyber-physical systems. The model treats time explicitly as part of the semantics. The model borrows concepts from the actor model, logical execution time models, synchronous reactive languages and discrete-event models [3]. The model has been realized by [2], [1], as a coordination language called Lingua Franca. This language will be described in detail in chapter 2.6.

Reactors are the primary components in the reactor model. Each reactor consists of an internal state, blocks of computations called *reactions* and ports that handle message-passing communication. Reactors may only communicate with messages containing time-tagged values, which are discrete events [2]. The reactor model keeps two separate notions of time; logical and physical time. Logical time progresses only at suitable moments during execution [2], while physical time is a measure of time from a local clock.

A top-level reactor, called the main reactor, instantiates other reactors and defines their connections [2]. Reactors can always be composed hierarchically.

2.5.2 Reactions

Reactions contain the blocks of computation that need to be coordinated by the model. The contents of reactions are abstracted away from the reactor model itself, and are thus not part of the model analysis [2].

Reactions are triggered by events at the logical time of the triggering event's tag. If several reactions in a reactor are triggered simultaneously (i.e. at the same logical time), then they will execute in the order they are defined within the reactor [2].

The possible output events of a reaction must be declared in its definition, even if it may not produce such outputs for every execution [2]. The model cannot know if an event will be produced or not, as this depends on the specific contents of the reaction. These design choices result in a conservative, but transparent approach. It is conservative since the model must always allow for the possibility of the event being produced. It is transparent because all dependency information is available upfront, greatly simplifying the dependency analysis.

2.5.3 Time

Time and timing are central concerns in the reactor model. Two concepts interact with a notion of time; logical time and physical time. Logical time is the time controlled by the model. The model progresses logical time only at suitable moments during execution - depending on if a logical instance has been completed or not [2]. Physical time is collected as input to the model from some physical clock [2]. The model has primitives to collect

logical and physical time values and compare them. Logical time always lags behind physical time[2].

A logical time tag follows the *superdense model of time* [2]. It has two values, a logical time instance and a microstep value. The microstep is a unitless number that indicates precedence in case of simultaneous time instances. A useful comparison can be made with Newton's cradle impact. As the first ball hits the rest, a simultaneous impulse sends the energy to the last ball, which swings out. In this almost simultaneous impact, there is a clear order, or precedence, to the impacts. Equivalently, if an event triggers a reaction that itself triggers another reaction at the same logical instant, then there is a clear order. The newly triggered event will be assigned a tag with one microstep more than the previous. Because of the microstep, two events are only simultaneous if *both* the logical time value and the microstep are the same.

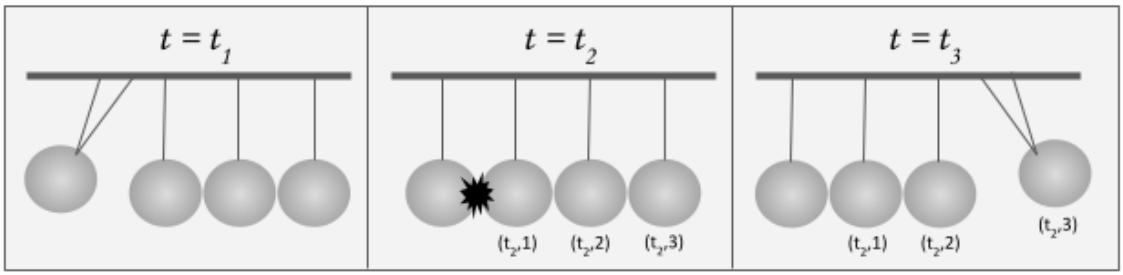


Figure 2.10: The figure shows Newton's cradle with illustrating tags. The impact at t_2 creates three events, with increasing microstep values.

2.5.4 Ports, Actions and Timers

Ports, timers and actions are related concepts. When triggered, ports and actions can execute a reaction and possibly generate events. While ports can be connected to and handle events from other reactors, actions are not visible to other reactors. Actions can schedule events at future logical times.

There are two types of actions; logical actions and physical actions. These differ in what tag is assigned to the generated event [2]. Events from logical actions are assigned some logical time tag, while events from physical actions are assigned a physical time value.

Timers generate events either periodically or once (i.e. one-shot timers).

2.5.5 Determinism

Determinism can be defined as, given an initial state and a set of inputs, it produces exactly one behavior [2]. In order to assert whether a *model* is deterministic, one has to define what state, inputs and behavior mean. Timing is not a property of a model, but rather a property of the physical implementation of it [2]. It is therefore not considered part of the behavior in the model. Inputs can be defined as a sequence of time-stamped events, while the behavior can be defined as the sequence of output events.

The reactor model is deterministic in the sense that for a given state and sequence of time-stamped events, it produces exactly one sequence of time-stamped output events.

This is true even for distributed execution (as long as the assumptions hold, see Section 2.5.6).

This is in contrast to the actor model, or pub-sub coordination, as sequences of events can produce more than one output sequence. These models do not provide any guarantees on the ordering of messages that have been sent and received (or published and observed) on multiple channels or topics [2], [3]. In a single channel or topic, ordering will be preserved due to the underlying network protocol, and not due to the model.

There are three main aspects of the model that need to be proven to be deterministic in order for the whole execution to be deterministic. Firstly, the model needs to be deterministic in each tag. Secondly, the model needs to be deterministic over all tags (i.e. a whole program execution). Thirdly, the contents of reactions need to be deterministic.

This last point is worth noting especially. The contents of reactions are abstracted away in the reactor model, and are not part of the analysis of it [2]. In practice, this means that it is up to the application designer to make sure the reactions are deterministic, or to purposely make some parts nondeterministic. Nondeterminism may be useful when used consciously, for example performing sensor measurements.

The following sections are meant to give the reader an idea of how discrete-event formalisms can be utilized to prove deterministic semantics in the reactor model [2]. Some details are left out.

Determinism in each tag instance To prove the determinism in each tag, a reactor may be modeled as a function with events as inputs and outputs. An event is simply a value-tag pair. In addition, events should also have values signaling known absent (symbol: ϵ) and unknown absent (symbol: $\perp$).

$$\tilde{T} = T \cup \{\perp, \epsilon\} \tag{2.3}$$

where T represents the set of all possible values of an event. $\tilde{T}$ is the relevant set to represent a single input or output of a reactor. Input dimension N or output dimension M may then simply be modeled as

$$F : \tilde{T}^N \longrightarrow \tilde{T}^M \tag{2.4}$$

No input or no output can be modeled with $\tilde{T}^0$, since only one possible value is essentially that the value does not matter.

Any reactor network can be transformed into an equivalent feedback structure with one reactor. A tag instance can then be analyzed to see if the structure reaches a fixed point, expressed mathematically as

$$F(s) = s, s \in \tilde{T}^N \tag{2.5}$$

where N is the number of inputs or outputs (which will be equal in the feedback structure). If such a fixed point can be proven to exist for a given reactor function, then the model at tag t will converge to a single behavior (i.e. deterministic).

To prove a fixed point we use Kleenes fix point theorem. First, we need to define what a monotonic function on a partially ordered set means. A partially ordered set is a set with

ordering between elements, but not between all the elements. The order in this case is

$$\begin{aligned} \bot < \epsilon \\ \bot < t, t \in T \end{aligned} \tag{2.6}$$

This ordering means that ϵ and any value t are not comparable. For tuples we can say that $(\bot, \bot) < (\bot, \epsilon) < (\epsilon, \epsilon)$. A monotonic function on this set means that for every $t, p \in \tilde{T}^N$ where $t < p$, then $f_n(t) < f_n(p)$ is also true.

Theorem 2.5.1: Kleenes fix point theorem

For every monotonic function $f : A \to A$ on a partially ordered set $(A, \leq)$ with finite depth, let

$$C = \{f^n(\bot) : n \in \mathbb{N}\} \tag{2.7}$$

where C is called a chain. Then the least upper bound of C is also the fixed point of f.

Determinism over sequences of tags

To analyze the model over sequences of events, a reactor can be modeled as a function with signals as inputs and outputs. A signal is a sequence of events, modeled as a partial function mapping tags to values. It is a partial function since only tags corresponding to ticks are defined.

$$s : (T \rightharpoonup V) \tag{2.8}$$

Thus, a reactor function can be expressed as

$$F : s^N \to s^M \tag{2.9}$$

$$F : (T \rightharpoonup V)^N \to (T \rightharpoonup V)^M \tag{2.10}$$

where N is the number of inputs and M is the number of outputs [2].

To find a fixed point in a function that handles signals, one must have an appropriate distance measurement. The Cantor metric, defined in 2.11, is possible to use.

$$d(s_1, s_2) = 1/2^\tau \tag{2.11}$$

where τ is the index of the first tag where the two signals differ. In the Cantor metric two signals are more similar if they are alike for longer. The distance metric fulfills the requirements of an ultrametric space.

A function is *contracting* in a metric space if the distance between two input points is larger than the corresponding output points. More precisely, a function is contracting if

$$d(F(s_1), F(s_2)) \leq d(s_1, s_2) \tag{2.12}$$

where d is the distance metric. A function is strictly contracting if

$$d(F(s_1), F(s_2)) < d(s_1, s_2) \tag{2.13}$$

and delta contracting if

$$d(F(s_1), F(s_2)) \leq \delta * d(s_1, s_2) \tag{2.14}$$

where $\delta < 1, \delta \in \mathbb{R}$.

Now, the Banach fix point theorem gives how to arrive at a fixed point [2].

Theorem 2.5.2: Banach fix point theorem

Let (X,d) be a complete metric space and $F : X \to X$ be delta contracting. Then F has a unique fix point, and the following sequence converges to that point.

$$\begin{aligned} s_0 &= s \\ s_{k+1} &= F(s_k) \end{aligned} \tag{2.15}$$

2.5.6 Distributed Execution

The reactor model has been extended by [3] to be possible to use on a distributed system of spatially separate hosts, where reactors need to communicate with each other through a network. This is called a *federation*. Each instantiated reactor inside the top-level reactor will become a *federate*, and can be assigned to run on different hosts. A RunTime Infrastructure (RTI) [3], [6] acts as a central coordinator at startup and shutdown. This is necessary for distributed systems, since there is no common physical clock. To have a consensus on a logical start time, each federate must synchronize its physical clock with the RTI, and then send a measured time value. The RTI chooses the federation start time as

$$max(\phi_g) + d$$

where ϕ_g is the collected list of time values received from federates, and d is an automatically calculated delay based on federation size and communication delay [3].

To handle the shutdown of a federation in a deterministic manner, the RTI is also necessary [3]. When a federate determines to shut down, it notifies the RTI. The RTI collects suggestions for a final tag from the rest of the federates, and decides on the maximum of these. The decided final tag is sent to all federates, which will process events normally up to, and including, this tag.

A federated LF program must follow the LF semantics. This means that a federate must not process an event with tag g before it is guaranteed that no event with tag $t \geq g$ will arrive. To ensure this in the most efficient way possible in a distributed system, two coordination schemes are proposed by [3]. The first is centralized coordination, which uses the RTI to handle all federate-to-federate communication and federation and time management. The centralized coordinator emphasizes consistency over availability, ensuring correct behavior even if network latency increases and data is unavailable or inconsistent. This approach is based on the HLA standard. The second approach is to use decentralized coordination, where the federates themselves handle communication and time management. This approach is based on PTIDES and Spanner. The decentralized coordinator

emphasizes availability over consistency, ensuring responsiveness even if network latency increases and data is unavailable or inconsistent.

The choice of coordination scheme will depend on if the application should prioritize consistency or availability. Two parameters added by [3], *stp_offset* and *after*, allow further control over these properties, letting consistency and availability be prioritized differently *within the same program*. For example, this would be used if federates have different importance, as in a system with one federate reading and reporting sensor measurements and the other representing an emergency system.

The *after* parameter represents a logical delay on a connection. This means that if a message has a tag (x, 0) and the connection has an *after* delay of a, then the reactor receiving the message will see its tag as (x + a, 0). This can be useful if tags before x + a should be allowed to be processed while waiting for the message to be received. If this is allowed, then the consistency is sacrificed, since events with tag $t \geq x$ at the receiver could be executed before the event with tag x, since x has a new tag $x + a$. The semantics are followed in regard to the new tag. On the other hand, the availability of the receiver reactor is increased, since other events (which might be more important) can be processed. If the logical delay accounts for the whole connection latency, then events may be safely processed at the tag $x + a$, without any noticeable delay.

Centralized Coordination
Centralized coordination is achieved with an HLA-like approach, using a control structure called an RTI. All coordination is handled by the RTI, and all federate-to-federate messages are passed through it. Tag and event processing information is passed from each federate to the RTI using control messages. From this information, the RTI can update its state variables, and let federates advance following the reactor semantics.

The RTI is a single point of failure in the program. If the RTI becomes unavailable, no federate may progress. Similarly, if the RTI needs some information from a federate that has become unavailable, then the rest of the federates may not proceed. Centralized coordination is a conservative coordination approach with slower performance. This makes sense as consistency is emphasized over availability.

The different values recorded on each federate by the RTI are listed in Table 2.4. The Next Message Request (NMR) message contains the Logical Tag Complete (LTC) and Next Event Tag (NET) values, and is sent from each federate to the RTI at the start of execution and at the completion of each tag.

The LTC value represents the latest tag in which all events with that tag have been completed. Consequently, no messages with this tag or smaller should ever be sent to or received from this federate. The LTC value is recorded separately for each federate, since the federates may progress asynchronously. The initial LTC value sent is $-\infty$.

The NET value represents the tag of the earliest event in the event queue of a federate. Since it represents the tag of the next event, it is ∞ if the event queue is empty. If the RTI has not received the NET value previously, then the stored value is $-\infty$.

The TAG value is sent from the RTI to each federate. It represents the next tag that the federate may safely advance to. The evaluation of this value for each federate is the core of the coordination scheme, and requires all the other values in Table 2.4. The TAG value is initially $-\infty$.

The EIMT value represents the earliest tag a message from an upstream federate may have.

RTI Value Record	Description	to/from	Is ∞ if	Is -∞ if
Logical Tag Complete (LTC)	All reactions or events with tags smaller than or equal to LTC have been completed	RTI/federate (NMR)	-	Initial
Next Event Tag (NET)	The earliest event in event queue of federate f has tag NET	RTI/federate (NMR)	Event queue empty	No NET message recieved
Tag Advance Grant (TAG)	The most recent tag sent to federate f to permit advancement to that tag	federate/RTI	-	Initial
Earliest Incoming Message Tag (EIMT)	The earliest tag an incoming message can have from upstream federates (recursive)	-	No upstream federates for federate f	-

Table 2.4: The table shows what values the RTI stores, their significance, origin and special values.

Since any federate may have many upstream federates, this value is found by recursively checking tags of the next events of those upstream federates. If a federate does not have any upstream federates, then the value of EIMT is ∞. This value is important to keep track of since it is not safe for the federate to progress beyond this value, as upstream federate messages may generate events with tags at this value or later. The EIMT value is kept by the RTI, and not sent in any message. Mathematically, the EIMT value can be calculated with

$$EIMT_f = \min_{u \in U(f)} (\min(EIMT_u, NET_u) + a_{uf}) \tag{2.16}$$

where $U(f)$ is the set of upstream federates from federate f, $EIMT_u$ is the tag of the earliest event in upstream federates (recursive), NET_u is the next event tag of the upstream federate u and a_{uf} is the logical delay between the upstream federate u and federate f.

When the RTI receives an NMR message from a federate f (NMR_f), it first stores the new values for LTC_f and NET_f for that federate, then sends a TAG message to downstream federates. This allows downstream federates to advance their tags. The value of the TAG message sent to downstream federates is the smallest LTC value of their upstream federates. This ensures that upstream federates always progress before downstream federates that possibly depend on them. Mathematically, this can be expressed as

$$g_d = \min_{u \in U(d)} (LTC_u + a_{ud}) \tag{2.17}$$

where U(d) means the set of all upstream federates of downstream federate d, LTC_u is the latest completed tag of the upstream federate u and a_{ud} is the logical delay on the connection between the upstream federate u and the downstream federate d. The result, g_d, must be compared to the latest TAG_d. If g_d is larger, then a new TAG_d value is sent.

If there is a next event at a tag later than what federate f is currently at (i.e. $TAG_f < NET_f$) and there are no possible messages from upstream federates until after federate

f's next event (i.e. $EIMT_f >= NET_f$), then the RTI allows federate f to proceed to its next event (i.e. setting $TAG_f = NET_f$).

A message is called *tardy* if its tag is smaller than the LTC value of the federate that received it. This means that the federate has already progressed beyond the relevant tag, thus not honoring the reactor semantics. The coordination scheme should, by design, avoid all occurrences of tardy messages.

In cases where there is a feedback loop in the system, and the RTI can not send a TAG, it can send a Provisional TAG (PTAG). What the PTAG means is that events with no dependency on network inputs (which can still arrive with the same tag) may execute. Another way to solve this would be to have logical delays which would allow the RTI to send a TAG.

If a federate f has a physical action, downstream federates cannot be allowed to proceed until it is known that the physical action was not triggered at tag t. Therefore, physical time must exceed NET_f, which does not retain its original meaning. In this case, the tag of the next event cannot be known, and NET_f is therefore simply the next tag. The federate f must then send updates (NMR messages) to downstream federates every time physical time exceeds NET_f. Only then can downstream federates proceed. Using the *after* keyword may relax this constraint, as a degree of inconsistency is then tolerated.

Decentralized Coordination
Decentralized coordination is achieved through a PTIDES-like approach [3], where the main idea is to optimistically process events based on the calculation of a safe-to-process (STP) offset. An event with tag t is safe to process if physical time satisfies

$$T_i \geq t + S_i \tag{2.18}$$

where T_i is the local measurement of physical time (i.e. from the federate's local clock), t is the event tag and S_i is the STP offset.

The offset expresses the necessary time to wait to be sure that no events with tags earlier than t arrive. The event with tag t may then be safely advanced to and processed, as all new events will have a tag with a value higher than t. Thus, events will be processed in tag order, honoring the semantics of the reactor model.

The STP offset is calculated based on assumptions on the total latency between a reaction deciding to send a message and the time the message was delivered on the receiving end. This includes the execution time of the reaction that sends the message, any communication overhead, physical communication latency and the clock synchronization error. Additionally, any logical delay (i.e. *after* keyword) on the connection must be included as well. A logical delay will increase the time value of the tag at the receiver end by some amount $\alpha \geq 0$. Thus, logical delays will account for some of the STP offset, meaning the STP offset can be decreased by the equivalent amount. If the logical delay is large enough, then the STP offset can be negative. If a federate contains physical actions, then the STP offset must be positive or zero, since a physical event can occur at any current physical time.

If the STP offset is violated, then the processing of tags can not be guaranteed to happen in tag order. Since decentralized coordination relies on explicit assumptions on physical time durations to calculate the STP offset, it gives the possibility to detect when faults occur. This gives the possibility of recovering after a fault.

To calculate the STP offset, it is necessary to identify the bounds of the different contributors to message delay. These bounds can then be added together to set the final STP offset.

Assume reaction r must be invoked within a deadline D. The reaction must then be invoked when physical time satisfies equation 2.20.

$$T_i \leq t + D_i \tag{2.19}$$

where T_i is the local measurement of physical time, t is the event tag triggering reaction r and D_i is the reaction deadline. In addition, no reactions at tag t can be invoked before physical time satisfies (2.18). The launch lag L can then be defined as

$$L_i = D_i - S_i \tag{2.20}$$

The launch lag bounds the delay between the start of the step at tag t, and the invocation of reaction r. It contains any scheduling overhead and execution time of any reactions set to execute before r.

The communication latency bound N_{ij} is the assumed maximum physical time between invocation of reaction r at federate i and delivery of the message at the receiving end at federate j. This includes the execution time of r, any network overhead and the message propagation time.

Lastly, the clock synchronization error bound E_{ij} represents the assumed maximum difference between two synchronized clocks at federates i and j.

The STP offset is calculated as the maximum relative offset between any two federates. The relative offset M_{ij} is calculated as shown in equation 2.18.

$$\begin{aligned} S_j &= \max_{i \in F}(M_{ij}) \\ M_{ij} &= \max(0, D_{ij} + N_{ij} + E_{ij} - \alpha_{ij}) \end{aligned} \tag{2.21}$$

Where α_{ij} is the minimum set logical delay over all connections between two federates i and j. Equivalently, The STP offset can be expressed as

$$S_j = \max(0, \max_{i \in F}(S_i + X_{ij})) \tag{2.22}$$

This can be expressed compactly in Max-Plus algebra, the reader is referred to pages 7-8 in [3].

2.6 Lingua Franca

This section covers the programming language Lingua Franca, which realizes program coordination based on the reactor model. The section will cover both the syntax of the LF language with examples, but also give an idea of how it is implemented and compiled.

2.6.1 Project Overview

Lingua Franca (LF), introduced in [1], is a realization of the reactor model as a coordination language. It is currently under development as an open-source project, with major contributors from universities in Berkeley, Dallas, Dresden, Kiel, and Seoul [37]. The project was started in 2019, and their logo is shown in Figure 2.11.

A coordination language differs from a general-purpose programming language in that only coordination between blocks of computation is handled, not the computation itself. In Lingua Franca, this is implemented such that a range of other programming languages may be used in the computational code. Currently, languages like C, C++, Python, TypeScript and Rust are supported target languages.

Figure 2.11: The figure shows the LF logo, taken from [37].

2.6.2 Installation

Lingua Franca is possible to install in several ways. The easiest way to install LF is to use the Visual Studio Code extension. A custom, Eclipse-based IDE is also available, called Epoch. Epoch can be downloaded from LF's documentation website at https://www.lf-lang.org/. A third option is to download the command-line tools. For developers and contributors to the project, it can be accessed in its entirety on Github.

2.6.3 Syntax and Diagrams

The syntax of Lingua Franca is based on other common programming language syntaxes, and should be relatively easy to understand. An LF program consists of at least one top level reactor; the main reactor. The top-level reactor may form a simple program on its own, but often it instantiates more reactors and sets connections between their inputs and outputs. All reactors can have a *startup* and *shutdown* reaction. See the simplest main reactor printing "Hello World!" in Figure 2.12. Another main reactor, which instantiates other reactors and connects their inputs and outputs is shown in Figure 2.15.

Inputs, outputs, timers, states and actions are declared inside a reactor, see two examples in Figure 2.13 and 2.14. If a type is relevant, then it must also be declared. The state variable in Figure 2.13 has also been initialized to a value of zero. The reactors would be instantiated in the main reactor in Figure 2.15, which also connects the output and input.

```
target C;

main reactor {
    reaction(startup) {=
        printf("Hello World!");
    =}
}

```

Figure 2.12: The figure shows a minimal LF program with a main reactor.

```
reactor ExampleReactorA {
    output SomeOutput:int;
    state SomeState:int(0);
    reaction(startup) -> SomeOutput {=
        // set output
        lf_set(SomeOutput, self->SomeState);
    =}
}

```

Figure 2.13: The figure shows a reactor that has one output and a state variable. The reactor sends its local state variable to its output.

```
reactor ExampleReactorB {
    input SomeInput:int;
    reaction(SomeInput) {=
        // Do something with input
    =}
}
```

Figure 2.14: The figure shows a reactor that has one input.

```
main reactor {
    a = new ExampleReactorA();
    b = new ExampleReactorB();

    a.SomeOutput -> b.SomeInput;
}

```

Figure 2.15: The figure shows a main reactor that instantiates two reactors and connects the output with the input.

The content of a reaction is defined with the delimiter symbols = ... =. Inside these delimiters, there will be code in a target language. The target language is set at the target declaration, see a basic example at the top of Figure 2.15 and a more advanced version in Figure 2.16. A target declaration can set a range of application and compilation-related settings, as in Figure 2.16. In this example an extra compilation flag is given, fast mode is set, logging is enabled and an application timeout of 1 second is given.

Timers can be defined by specifying a time offset and timer period. Time units are available as types in LF. The available units are *weeks*, *days*, *hours*, *minutes*, *secs*, *msecs*, *usecs* and *nsecs*. An example is shown in Figure 2.17

```
target C {
    flags: "-O3",
    fast: true,
    logging: log,
    timeout: 1 secs,
};
```

Figure 2.16: The figure shows an example of a target declaration.

```
reactor ExampleReactorA {
    timer t(1 secs, 100 msecs);
    reaction(t) {=
    =}
}
```

Figure 2.17: The figure shows a reactor that has a timer that generates events with a period of 100 milliseconds, after an offset of 1 second.

Preambles are pieces of code that can be specified outside of reactions, and that are available for the whole file, if specified outside of reactors, or the whole reactor, if specified inside a reactor. In the C runtime, there is no distinction between these two options. An example of a preamble is shown in the topmost part of Figure 2.18, and inside ExampleReactorA in the same figure.

```
preamble{=
    #define LED_PIN 2
=}
reactor ExampleReactorA {
    preamble{=
        void func() {
            // some code that can generate events
            // through physical action led_blink
        }
    =}
    physical action led_blink;
    reaction(startup) {=
        lf_thread_create(&thread_id, &func, NULL);
    =}
    reaction(led_blink){=
       // toggle LED_PIN
    =}
}
```

Figure 2.18: The figure shows an example of uses for a preamble in an LF program. This program starts a thread that can asynchronously interact with the LF program through a physical action. In this case, it blinks an LED.

Figure 2.18 also shows how to create threads in the application. The function lf_thread_create() uses the runtime API for thread creation. The function to execute can be specified in a preamble, as shown. This could be useful to execute asynchronous functions, like sensor measurements. Further, physical actions can be created to interface with the thread.

Deadlines can be specified for a reaction. If a deadline is not met (ie. measured physical time exceeds the deadline), then a deadline handler can define an alternative code block to execute. The deadline functionality is implemented in a so-called lazy manner, where it is executed instead of the reaction. It is triggered by the late event, not at the time the reaction should have been executed. An example is shown in Figure 2.19.

```
reactor ExampleReactorA {
    input i:int;
    reaction(i) {=
        // main code
    =} deadline(500 msecs) {=
        // alternative code
    =}
}
```

Figure 2.19: The figure shows a reactor that has one input with a deadline.

Distributed Execution

Distributed execution of an LF program is implemented as explained in Section 2.5.6. To run a program as a federation, the *main reactor* must be changed to *federated reactor*, see an example in Figure 2.20. Doing this signals that each top-level reactor should be an independent binary that could be deployed on different systems. A shell script is also provided for making the deployment process easier. The shell script is generated together with the program binaries under the name of the file, see an example in Figure 2.21. Here, the shell script is *bin/Federated*, while the two binaries correspond to the top-level reactors in Figure 2.20.

```
federated reactor at user@host1 {
    ExampleReactorA = new ExampleReactorA() at user@host2;
    ExampleReactorB = new ExampleReactorB() at user@host2;

    ExampleReactorA.SomeOutput -> ExampleReactorB.SomeInput;
}
```

Figure 2.20: The figure shows a basic federation.

```
bin/Federated
bin/Federated_ExampleReactorA
bin/Federated_ExampleReactorB
```

Figure 2.21: The figure shows the file output of building Figure 2.20.

To specify where to deploy, LF provides the *at* keyword. Both the RTI and the federates may be deployed to another host. How to do this is shown in Figure 2.20. If this is excluded, then the federation will run on *localhost*.

The RTI is a separate program entirely and must be installed on the system that should run it prior to running any federated LF program. Figure 2.22 shows the necessary steps to install the RTI. As of now, it is only possible to install from source files, meaning

federated LF is still in a relatively early phase of development. The auto-generated build script also runs the RTI.

```
$ git clone https://github.com/lf-lang/reactor-c.git
$ cd reactor-c/core/federated/RTI/
$ mkdir build && cd build
$ cmake ../
$ make
$ sudo make install
```

Figure 2.22: The figure shows how to install the RTI on your system.

Centralized coordination is the default coordination scheme. To use Decentralized coordination, the target declaration parameter *coordination* must be set to *decentralized*, as shown in Figure 2.23.

Decentralized coordination provides a unique handler for STP violations, shown in Figure 2.24. This handler works equivalently to a deadline miss handler, where if the STP offset is violated, the execution is routed to the code within the handler. This gives the opportunity to handle the fault and recover. As seen in chapter 2.5.6, the STP offset value is based on explicit assumptions on message delay bounds, which may be violated in any physical system.

As explained in chapter 2.5.6, an alternative to setting an STP offset is to use the *after* construct. The syntax is shown in Figure 2.25. A combination of both may also be used.

```
1
2     target C {
3         coordination: decentralized
4     }
5
```

Figure 2.23: The figure shows the target declaration if the program should use decentralized coordination instead of the default.

```
1
2     reaction(in) {=
3         ...
4     =} STP(0) {=
5         // Handle input that violates STP offset
6     =}
7
```

Figure 2.24: The figure shows the STP handler structure.

The target declaration parameter *clock-sync-options* allow the user to set options for clock synchronization.

```
1
2     federated reactor {
3         ExampleReactorA = new ExampleReactorA();
4         ExampleReactorB = new ExampleReactorB();
5
6         ExampleReactorA -> ExampleReactorB after 200 msec;
7     }
8
```

Figure 2.25: The figure shows how to set an *after* delay.

Diagrams
Automatically created program diagrams are synthesized from LF code. The diagrams help with understanding a program's structure, especially as they are interactive. An example diagram is shown in Figure 2.26. The graph represents a distributed program called "DistributedMessageTest", where the RTI is deployed with the visible IP address and port number. The top-level reactor contains two reactors. The Source reactor sends messages periodically, using a timer (the clock in the figure). The Drain reactor receives the messages and executes some reaction at their receipt. Additionally, both reactors have startup reactions, designated with a circle in the diagram, and shutdown reactions, with a diamond shape.

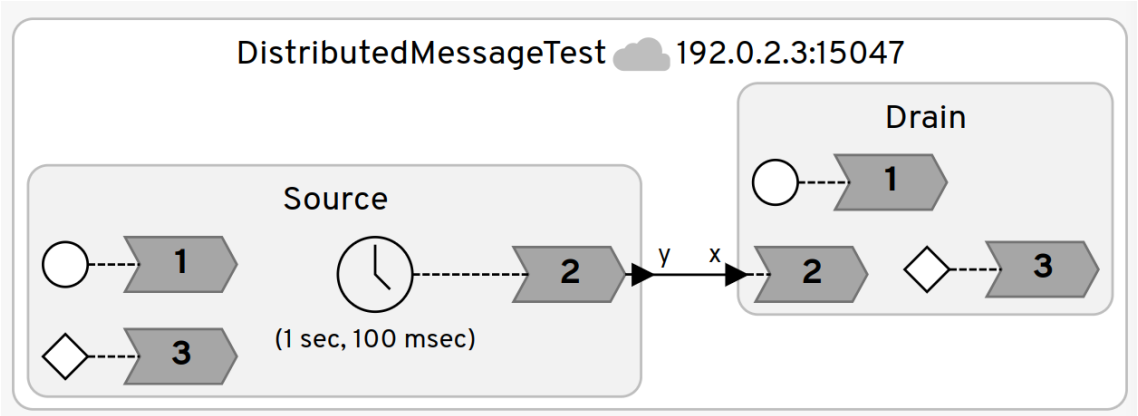


Figure 2.26: The figure shows an LF diagram for a federated program.

2.6.4 LF Compiler

The LF compiler interprets and validates LF code, and generates code in the target language. This generated code is then combined with the runtime code (see the next section), as can be seen in the flow chart in Figure 2.27. It is then compiled by the target compiler, and if valid, generates an executable. The compiler is written in Java. To add new features to the LF language, generate new code, and include new files, the compiler must be modified.

2.6.5 Runtime Environment

A runtime, runtime system, or runtime environment is, in general, the layer of abstraction that implements the features of a language. A user of the language will not need to study this implementation, only use the provided API.

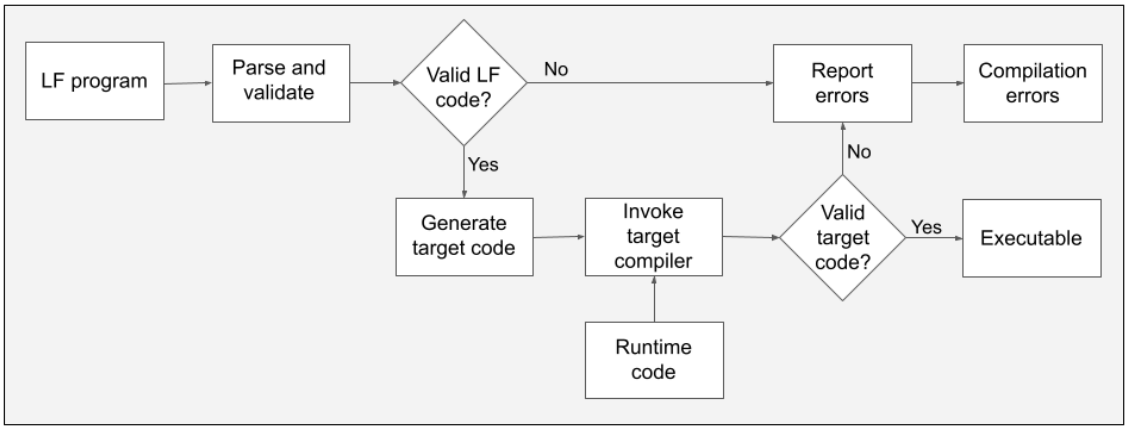


Figure 2.27: The figure shows the general structure of the LF compilation process.

Lingua Franca uses a runtime environment that implements the reactor model and the language features. Each supported target language has a runtime written in that language and must be ported to any new platform. The runtime code is divided into general parts and platform-specific parts. The parts that depend on the platform are typically the functionality that directly accesses the platform, such as reading a system clock. All time values in Lingua Franca are given in nanoseconds, however, the final time granularity is dependent on the available platform clock.

The C runtime has the least overhead and is the dominant language for use in embedded systems. It has been ported to various OSes and a few bare-metal platforms [1]. Notably, a recent addition to the runtime has provided support for Zephyr RTOS, although it is still experimental for the threaded part of the runtime.

The C runtime keeps track of two queues; the event queue Q_E and the reaction queue Q_R. The event queue contains all future scheduled events, sorted after tag order. Physical time is then compared to the tag of the next event, and if it is greater, then the events at that tag will be processed. All triggered reactions are put into the reaction queue. Thus, the reaction queue contains the triggered reactions of the current logical time. This loop repeats until there is no next event in Q_E (unless the keepalive property is specified), the timeout has been reached, or a call to shutdown is made in the application. This process is shown in Figure 2.28.

In each step, the reactions are evaluated whether they are ready to execute or not. The ready-to-execute reactions can not possibly depend on any other reactions in Q_R or that are currently executing. These reactions are then run when there is an available thread. This is then repeated until all reactions have finished execution.

Roughly, the structure of the code can be summarized in Figure 2.29. The parts common to all variations of the runtime can be found in the file *reactor_common.c*. The unthreaded parts are in *reactor.c*. The threaded runtime, together with the implemented schedulers are in the *threaded* folder. The platform-dependent code is in the *platform* folder and thus contains all the platform ports. The *federated* folder contains the federated parts of the runtime, together with the self-contained RTI.

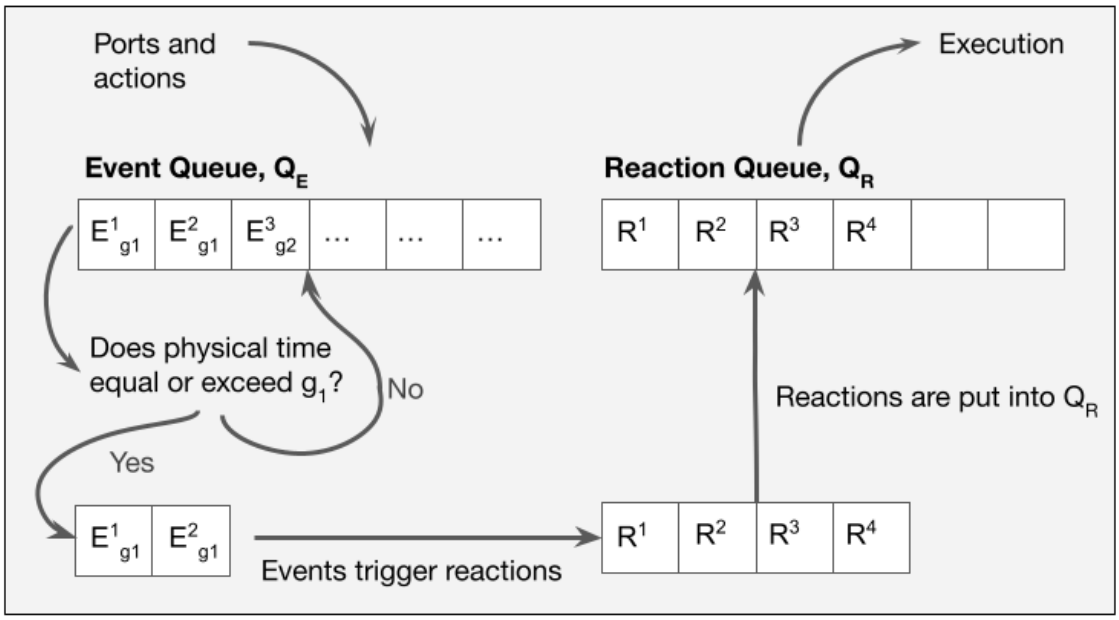


Figure 2.28: The figure shows a simplified description of the runtime.

```
reactor-c/core
├── federated
│   ├── RTI          // Self-contained RTI
│   │   ├── message_record        //
│   │   ├── rti.c          // Contains main function
│   │   ├── rti_lib.c          // RTI function definitions
│   │   └── rti_lib.h          // RTI function declarations
│   ├── clock-sync.c          // Clock synchronization functions
│   ├── federate.c          // Federated functions
│   └── net_util.c          // Communication related functions
├── modal_models          //comment
├── platform          // Platform-dependent functions
│   ├── lf_zephyr_support.c          // Zephyr port
│   └── ...
├── threaded          // Threaded part of runtime
│   ├── reactor_threaded.c          // Main threaded runtime logic
│   └── ...
├── utils
├── reactor.c          // Unthreaded part of runtime
├── reactor_common.c          // Common runtime logic
└── ...
```

Figure 2.29: The figure shows the C runtime code structure.

2.6.6 The Federated Runtime

The federated runtime is an expansion of the C runtime that implements the federated functionality. It uses BSD sockets for communication. Sockets are explained in detail in Section 2.3. The federated runtime implements decentralized and centralized coordination as described in Section 2.5.6. Federated execution is only supported on platforms Linux, Windows and MacOS.

A federate is implemented as a structure containing the federate's state information. Threads receive messages on sockets, either connected to the RTI (centralized coordination) or both the RTI and federates that connect to it (decentralized). There are more threads in the decentralized coordination scheme. The federated LF runtime supports IPv4 addresses.

2.6.7 The RTI Program

The RTI, relevant for federated execution as discussed in Section 2.5.6, is implemented separately from the rest of the runtime environment. It is implemented as an independently compilable program that can be launched on Linux or MacOS platforms. The RTI accepts command-line arguments specifying the expected number of federates that should connect to the federation, the federation ID, the port that federates should connect to, clock synchronization options and authentication options. Only the number of federates to connect is mandatory to provide. By default, clock synchronization is set to *init*, meaning it is only performed once at the execution start. Authentication is, by default, off.

The RTI utilizes one communication thread per federate, with both centralized and decentralized coordination. Although, in decentralized coordination, it is not used during execution.

2.6.8 LF Clock Synchronization

Clock synchronization can be set to *off*, *init* and *on*. With *init*, one clock synchronization round is done once at startup and over the TCP socket. With clock synchronization set to *on*, clock synchronization is performed at some frequency over a UDP socket. The target parameter *clock-sync-options* specifies more options, for example, whether federates on the same physical platform should perform it, or the period of the clock synchronization. Setting the option to *off* will turn off LF clock synchronization, and rely on clock synchronization from the network.

The LF clock synchronization is based on clock synchronization as described in Section 2.4.2.

2.7 Other Software

2.7.1 Wireshark

Wireshark [38] is an open-source network protocol analyzer. It allows you to capture and analyze network traffic in real-time, as packets of data traverse the network interface. It supports various protocols, including Ethernet, Wi-Fi, TCP/IP, and many others. Wireshark provides a detailed and customizable display of the captured data, allowing you to examine the contents of individual packets, inspect protocol headers, and analyze network behavior. It is therefore useful for debugging of network-related functionality, as in this thesis.

2.7.2 PuTTY

PuTTY [39] is an open-source serial console, terminal emulator and network file transfer application. In this thesis, it is used solely as a serial console. To use it as a serial console, you need to configure the appropriate serial settings such as the serial line name, baud rate, data bits, parity, stop bits, and flow control. Once the settings are configured, you can establish a connection to the serial device and interact with it through the PuTTY terminal interface.

2.7.3 GNU Debugger (GDB)

GNU Debugger (GDB) is an advanced debugger in the GNU ecosystem [40]. In this thesis, it will refer to arm-gdb, which is the GNU toolchain version for ARM targets. The components for debugging embedded applications with GDB are shown in Figure 2.30. The host computer runs a GDB client, which is the GDB user interface. The GDB client then communicates with a GDB server, which can run on the host computer or on a remote machine [40]. The GDB server interfaces with a debug probe through a USB connection [41]. The debug probe can be an external probe or it can be fully integrated on a board. The debug probe interacts with the debug interface (JTAG or SWD protocols) on the microcontroller.

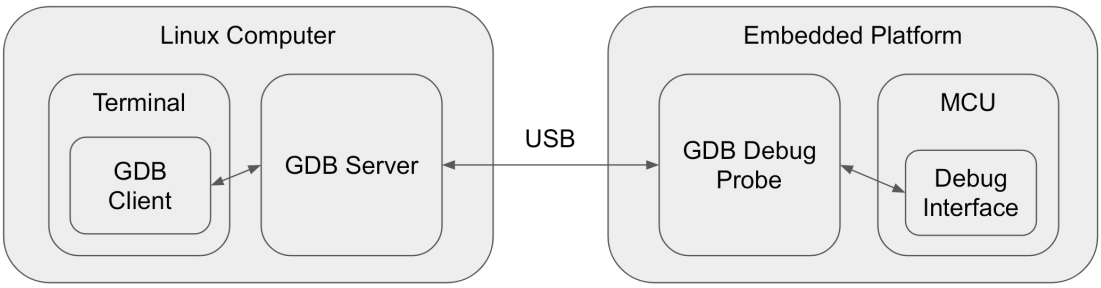


Figure 2.30: The Figure shows the components necessary for GDB debugging.

2.7.4 pyOCD

pyOCD [42] is an open-source debugging tool for Arm Cortex-M microcontrollers based on Python. It supports several debug probes, and provides a GDB server and flashing tools.

2.8 Hardware Platform

This section presents relevant hardware information.

2.8.1 NXP MIMXRT1170-EVK

The development platform for this thesis is NXP's MIMXRT1170-EVK. This development kit provides high performance and a wide range of features, only some of which will be used for this thesis. The board uses the i.MX-RT1170 MCU which has two cores; 1GHz Arm Cortex-M7 and 400 MHz Cortex-M4. The board has an integrated, programmable debugging interface and two ethernet connectors (one with 10/100M and one 10/100/1000M). The board is supported by Zephyr RTOS and by Lingua Franca's Zephyr port, although gigabit ethernet is not yet supported.

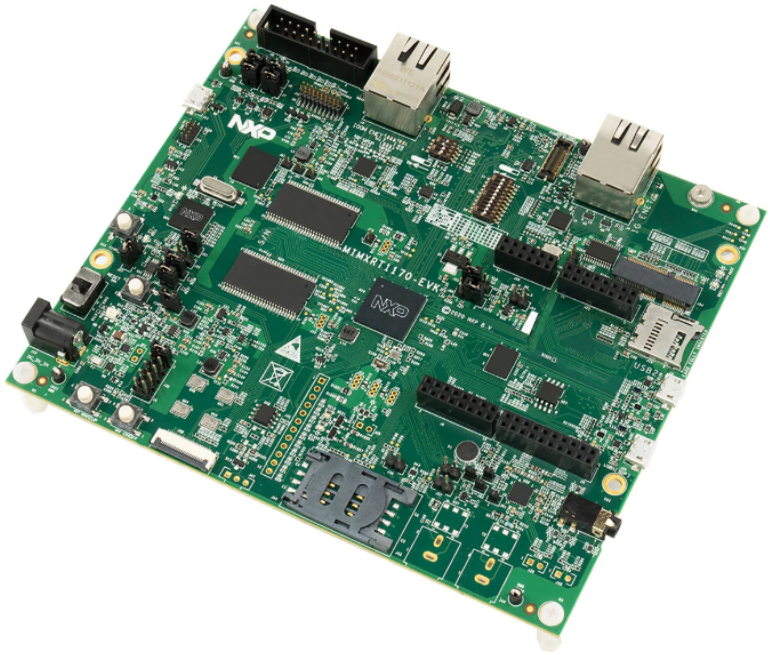


Figure 2.31: The figure shows the NXP MIMXRT1170-EVK. Image taken from NXP's webpage.

2.8.2 Other hardware

- Two-layer network switch: tp-link TL-SG105
- Oscilloscope: Rhode&Schwarz RTB2004 2,5 GSa/s

Chapter 3

Specifications

This section presents the specifications of the project. It must be taken into account when reading this section that very select parts of a large code (LF runtime) base and surrounding structures (LF compiler, Zephyr application configurations) will be altered.

3.1 Project Objectives and Scope

The main objective of the project is to expand the platform support of LF, specifically federated LF, to Zephyr RTOS. The federated functionality as described in Section 2.5.6 is implemented in the C runtime for platforms Linux, MacOS and Windows. A number of adaptions must be made to be able to expand the platform to Zephyr.

The scope of this project will include adapting federated execution with centralized and decentralized coordination, initial clock synchronization and continuous clock synchronization to the Zephyr platform. The authentication functionality in the federated runtime will not be included to limit the scope.

3.2 Functional Description

In order to expand the platforms for federates and the RTI, a number of changes to the runtime, Zephyr application configurations and changes to the LF compiler must be made. All functional adaptions are listed in specification Tables 3.1, 3.2 and 3.3.

The goals are sorted in a hierarchical structure, where the main points are at the top. The points needed to adapt in order to reach the main goal are at the second layer, the points needed for the second layer are at the third layer, etc. Some select points have additional comments in the "Comment" column. The last column, "Where change", gives information about where the functional change must be made, either in the LF runtime (which includes the federated runtime), the RTI (which is not technically part of the runtime since it is currently its own program), Zephyr config (Zephyr application code, configuration files, etc.) or Linux config (command-line arguments to configure certain things).

Project Functional Specifications, part 1				
Goal			Comment	Where change
1	Multi-platform federation support		Federation that may have Zephyr federates	
1.1	Zephyr platform support for federate			
1.1.1	Platform-independence for federate			runtime
1.1.1.1	Library support			runtime
1.1.2	Function porting			runtime
1.1.2.1	RTI connection			runtime
1.1.3	Runtime thread creation		Federated runtime creates a number of threads depending on coordination scheme, connections and clock-sync	runtime
1.2	RTI connection			
1.2.1	TCP socket support		TCP sockets used for federate communication and inital clock synchronization	runtime
1.2.1.1	TCP socket creation			runtime
1.2.1.2	TCP socket connection			runtime
1.2.1.3	Send and receive data			runtime
1.2.1.4	TCP socket close and shutdown			runtime
1.2.2	UDP socket support		UDP sockets used for clock synchronization	runtime
1.2.2.1	UDP socket creation			runtime
1.2.2.3	Send and receive data			runtime
1.2.2.4	UDP socket close and shutdown			runtime

Table 3.1: The table shows

Project Functional Specifications, part 2					
			Goal	**Comment**	**Where change**
		1.2.3	IP address configuration		Linux/ Zephyr config
			1.2.3.1 Set interface IP addresses		Zephyr config
			1.2.3.2 IP address routing		Linux/ Zephyr config
	1.3	Federate connection			
		1.3.1	TCP socket support	Same as 1.2.1	
		1.3.2	IP address configuration	Same as 1.2.3	
	1.4	Multiple Zephyr federates			
		1.4.1	MAC address configuration		Zephyr config
2	Embedded-only federation support			Zephyr federation	
	2.1	Zephyr platform support for federate		Same as 1.1	
	2.2	Zephyr platform support for RTI			
		2.2.1	Platform-independence for RTI		RTI
			2.2.1.1 Sleep functionality abstraction		RTI
			2.2.1.2 Thread synchronization abstraction		RTI
			2.2.1.3 Thread abstraction		RTI
			2.2.1.4 Library support		RTI
		2.2.2	Runtime thread creation	RTI creates a number of threads depending on connections and clock-sync	runtime
	2.3	Federate connection			
		2.3.1	TCP socket support	Same as 1.2.1	
		2.3.2	UDP socket support	Same as 1.2.2	
		2.3.3	IP address configuration	Same as 1.2.3	

Table 3.2: The table shows

Project Functional Specifications, part 3			
Goal		Comment	Where change
2.5	Adapted main function	Main function must be changed for the federate and RTI topology	compiler
2.5.1	Hardware initialization		compiler
2.5.2	RTI arguments		compiler
2.5.3	Create RTI entry thread		compiler
2.5.4	Create Federate entry thread		compiler
2.6	Thread termination order	Threads must terminate in the correct order to avoid race conditions	runtime

Table 3.3: The table shows

3.2.1 Target Declaration Functionality

The target declaration options that will be tested are shown in Table 3.4, together with their priority and a small description. The high-priority options must work for the project to work. The medium priority would be beneficial if worked. All other target declaration options are implicitly low priority and will not be tested.

Target Declaration Parameters		
Priority	**Parameter Name**	**Description**
High	*coordination*	Coordination scheme; both decentralized and centralized coordination should work
High	*threading*	Multi-threaded runtime; necessary for federated LF
High	*workers*	Number of worker threads to execute program code
High	*platform*	Runtime platform to choose; Only Zephyr will be tested
High/ medium	*clock-sync*	Clock synchronization setting; either ”init”, ”on” or ”off”. Only ”init” must work. ”on”/”off” is medium priority
Medium	*no-compile*	LF compiler should not call target language compiler
Medium	*clock-sync-options*	Clock synchronization options; frequency of UDP messages etc. Only relevant for ”on” option above.

Table 3.4: The table shows the target declaration options that are important to the project, ranked after priority.

3.3 Other Goals

A goal of the thesis is to contribute to the LF Github project. The possible repositories to contribute to are *lingua-franca*, *reactor-c* and *lf-west-template*. As not all adaptions listed in the functional specification tables 3.1-3.3 are changes to the runtime, these must be documented thoroughly and can be added to a README-file in the listed repositories. Applications, application files and documentation should be merged into *lf-west-template* repository. Working, simple examples should be provided.

It would be ideal to implement proper integration with the LF language and syntax. However, as the syntax for federations are not entirely reliable and the scope of the project is already large, it is not of high priority.

Chapter 4

Design

This section presents the design considerations, choices and justifications for those choices. It also presents a logical separation of the work based on the deployment topology.

4.1 Hardware Overview

The hardware for the project consists of three main units; a Linux computer and two identical MIMXRT1170-EVK boards. The reason to use MIMXRT1170-EVK boards is that they have ethernet connectors. The ethernet connector used is 100M ethernet, since Zephyr has not implemented 1G ethernet. All nodes are connected with ethernet cables through a 2-layer network switch. Further, USB cables connect the Linux computer to the two boards through their debug connector. This is both for easily flashing and debugging the binaries, for powering the boards and to allow serial communication with the boards.

Furthermore, a 4-channel digital oscilloscope with probes will be used to measure timing with high precision. See Section 2.8 for more information on the hardware.

4.2 Choice of Zephyr RTOS

Zephyr was chosen as the RTOS for this project. The main motivation for choosing Zephyr instead of, for example, FreeRTOS is for hardware independency. From porting the single-threaded runtime to FreeRTOS in [7], it was a definite downside that the FreeRTOS port would need to be ported, to some extent, to each new hardware platform. Because of Zephyr's use of Devicetree, explained in Section 2.2.2, almost all hardware-specific configurations are already taken care of. This is given that the board is supported, and that the necessary board modules are supported. The MIMXRT1170-EVK and necessary modules (ethernet, GPIO, etc.) are supported already by Zephyr. The consequence of using Devicetree is that the actual code will be the same (ie. no need for porting to another board), only configuration files will differ. Additionally, most configurations (see Figure 2.5 in Section 2.2.2) are loaded by specifying the board name to the *west* tool.

Another aspect is that Zephyr has a partial POSIX implementation. This could greatly simplify the changes that need to be done to the federated runtime. The POSIX implementation is continuously expanded, and so are many other features, such as PTP

synchronization, networking technologies, debugging and shell functionality.

Both the unthreaded and threaded runtime has been ported to Zephyr already, although the threaded port was still in an experimental state at the start of this project. For FreeRTOS, only the unthreaded runtime has been ported (and not added to the official repository).

Another conclusion from [7] was that the possible time precision with FreeRTOS was limited due to hardware clocks not being available, and that the kernel clock could not run faster than at 1 kHz. For more details about FreeRTOS and timing, the reader is referred to [7]. Zephyr, on the other hand, can access hardware timers. For MIMXRT1170-EVK, a high-precision timer running at 26 MHz can be used. The kernel clock cycles can be read using high-precision APIs, and give highly accurate and stable time measurements.

Overall, Zephyr seems like a solid choice that has the potential to be used for all kinds of embedded applications with Lingua Franca. It has both good functionality, in terms of advanced APIs, (partial) POSIX support and an integrated network stack, as well as good convenience, in terms of hardware independence, configurations and a build tool.

4.3 Federation Topologies

Four main federation topologies can be created in a system with two embedded nodes and one Linux machine node. The different topologies are shown in Figure 4.1. The figure shows the physical platforms together with the program components. Federation topology 1 is the first to implement, as it is the simplest topology. It will involve the steps listed under point 1 of the specifications in Tables 3.1-3.2 from Section 3. The Linux computer handles the RTI and all federates, except one federate that is launched on a board. Federation topology 2 uses both boards with a federate launched on each. The RTI on the Linux computer may run an arbitrary amount of federates. In federation topology 3 the RTI is launched alongside a federate on a board. This will involve the steps listed under point 2 of the specifications in Tables 3.2-3.3 from Section 3. In topology 3, the whole federation is run on embedded boards. Topology 4 is more of a proof-of-concept that the platforms running federates or RTI is arbitrary. It will not really be tested during this project. Although notice that the RTI can be run alone on a board as well, however, in any real system the RTI would likely run on a central, general-purpose computer if it is available or jointly with a federate in the decentralized coordination mode.

No connections are shown in Figure 4.1. Physically, all nodes are networked with each other. The RTI must be able to communicate with all federates, regardless of the coordination scheme. If the coordination is centralized, the federates do not need to be connected to each other. If the coordination is decentralized, then a federate must be connected to another federate if they are to communicate.

4.4 Zephyr Platform Support for Federates

In this section, special considerations regarding point *1.1 Zephyr platform support for federate* from specification Table 3.1 is presented.

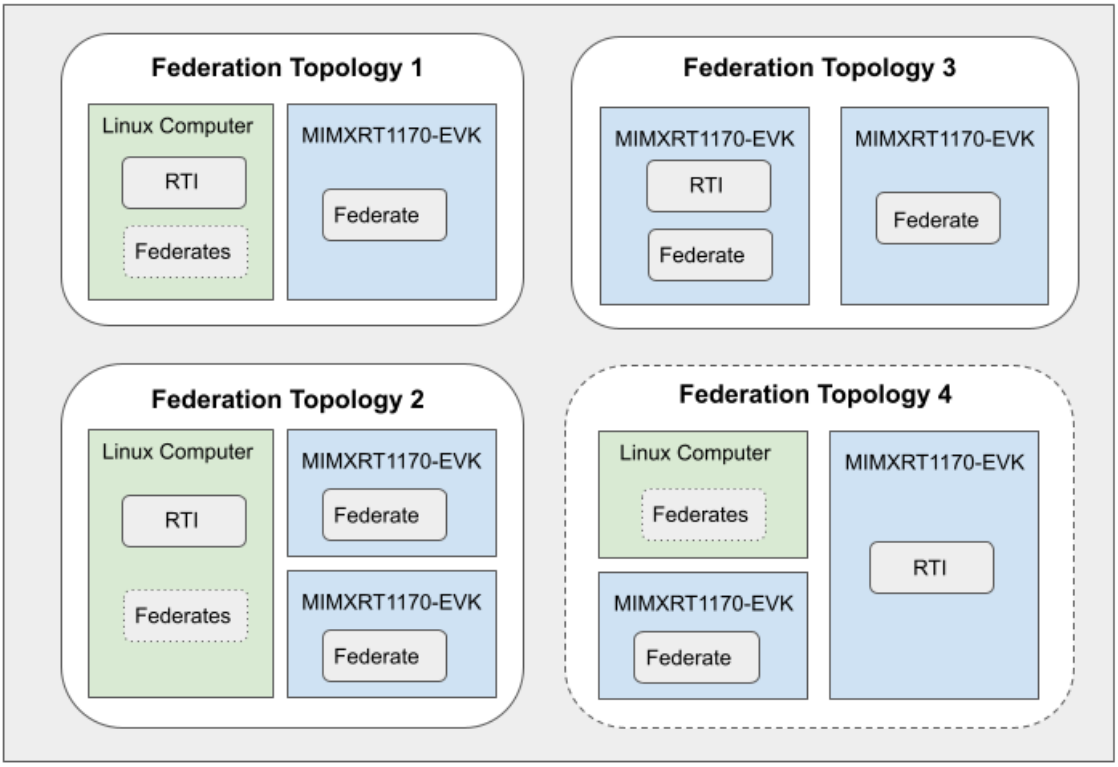


Figure 4.1: The Figure shows the relevant federation topologies for the project.

4.4.1 Choice of Zephyr Socket API

The choice of socket library is part of the point *1.1.1 Platform-independence for federate* and *1.1.1.1 Library Support*. Zephyr offers two strategies for implementing POSIX sockets. One strategy is to use Zephyr's native socket API together with a configuration option called *CONFIG_NET_SOCKETS_ POSIX_NAMES*. This option makes Zephyr's socket implementation available with the standard POSIX names, such as *socket()*, *send()* and *close()*. Without it, the functions are available as *zsock_socket()*, *zsock_send()* and *zsock_close()*.

The other strategy is to use the configuration option *CONFIG_USE_POSIX_API*. This option also makes the Zephyr sockets available under standard POSIX naming. The main difference is that this option enables all POSIX library implementations, not just for sockets. This is the ideal option for more advanced applications, like the federated LF runtime, since it uses several POSIX libraries that need to integrate well together. If these libraries have been implemented in Zephyr's POSIX API, then using *CONFIG_USE_POSIX_API* is an obvious choice. Using the *CONFIG_NET_SOCKETS_POSIX_NAMES* is ideal only for very simple socket applications for this reason.

However, as mentioned in Section 2.2.8, the POSIX API is not completely implemented, and this should be taken into account by reading the documentation thoroughly. Some functions might not be available or might not work exactly as expected.

4.4.2 RTI Connection function

This section relates to point *1.1.2 Function porting*, as the function *gethostbyname()* is not supported by Zephyr's POSIX libraries and is considered obsolete, according to the Linux man pages. The suggested alternative is *getaddrinfo()*. This will implicate changes in the whole RTI connection function.

4.4.3 Static Thread Stack Definition

This section relates to point *1.1.3 Runtime thread creation*, as in the Zephyr platform-dependent runtime files (See Figure 2.29 from Section 2.6.5), thread stack memory regions are defined statically depending on the number of needed threads. In threaded LF, this is easy to do, since the only threads are worker threads and user-created threads. The number of worker threads are available in the runtime, and can thus be used directly. The number of user threads has been added as a platform target property that the user must specify. This number could then be used for the static stack definition.

For federated LF, however, there are significantly more threads, as presented in Section 2.6.6. To do this statically, one can utilize macros to define a precise number of needed threads in each federate or RTI program. This is preferred to defining a maximum amount of threads, since that would define a hard limit for the number of federates in the application, or occupy unused memory area of a potentially significant size.

The main stack size, and other operating system thread stacks, are set in Zephyr's Kconfig file.

4.4.4 IP Address Configuration

Regarding points *1.2.3/1.3.2 IP address configuration* (also *2.3.3 IP address configuration*). The IP addresses of the ethernet interface of the boards can be set using Zephyr Kconfig options. The IP address of the Linux machine ethernet interface must be set using Linux command-line options *ip*. The routing is also possible to set with these tools.

Since the IP addresses are specified manually in Kconfig files, one file per federate is needed. However, having separate configuration files is natural since each federate may need totally different configurations.

4.4.5 MAC Address for Multiple Boards

Regarding points *1.4 Multiple Zephyr federates* and *1.4.1 MAC address configuration*. The MAC addresses of each board must be unique. The boards will be assigned the MAC address in their network interface card. This will actually be the same address for both boards. If left unchanged, then this will surely create problems. In addition, configuring the MAC address cannot be done using KConfig options. Therefore, a solution with a DeviceTree overlay file must be used. The local MAC address can be manually specified with an option called *local-mac-address*. Thus an overlay file per federate should be created.

It is also possible to set the option *zephyr,random-mac-address* instead of specifying the MAC address manually. Then there would be no need for an overlay file per federate,

however, the randomness of this option is limited to only the last byte. This means that the boards will occasionally be assigned the same MAC address. Therefore, the solution with multiple overlay files is therefore preferred to setting *zephyr,random-mac-address*.

Additionally, having separate overlay files is natural if there are several boards, or if the boards do different things in the application.

4.5 Zephyr Platform Support for RTI

In this section, special considerations regarding point *2.2 Zephyr platform support for RTI* from specification Table 3.2 are presented.

The RTI application was written with the intention of only launching it on a Linux machine. Therefore it is not made in a platform-independent way. Making the RTI compatible with Zephyr should also be a step toward a platform-independent RTI. This can be done by using the already provided extra layer of abstraction, such as using *lf_thread*, *lf_sleep* and *lf_mutex_t*. These structures are implemented by the platform-dependent part of the runtime and should be relatively easy to extend to the RTI and federate runtime in general.

The RTI thread creation (*2.2.2 Runtime thread creation*) will then be adapted easily by simply adding more thread stacks statically, equivalently to Section 4.4.3.

Some libraries not supported in Zephyr must be excluded (*2.2.1.4 Library support*).

4.6 Adapted Main Function for RTI and Termination

In this section, special considerations regarding points *2.5 Adapted main function* and *2.6 Thread termination order* from specification Table 3.3 are presented.

To augment the main function generated by the LF compiler, the compiler itself needs to be modified. To be able to choose between a standard LF main function, and the adapted main function, a target property should be added to the language. The target property could look like in Figure 4.2.

```
1    target C {
2        platform: Zephyr,
3        joint-rti: true,        // new target property
4    }
5
```

Figure 4.2: The figure shows how a target property could be added to Lingua Franca to allow the joint RTI functionality.

4.6.1 Joint RTI Thread Structure

Regarding points *2.5.3 Create RTI entry thread* and *2.5.4 Create Federate entry thread*. When the program has both RTI and federate threads, created from the main function, it will be referred to as "Joint RTI" for simplicity. Since the RTI is normally run using its

own main function, the RTI main function must be adapted to be available to be launched in a created thread (ie. creating a function with similar content to the RTI main function). The federate threads will then be created as normal.

Regarding point *2.5.1 Hardware initialization*. Hardware initialization (ie. starting the LF clock) must be done only once, and at the correct time. To avoid potential race conditions, it should be done once before starting any of the threads.

Another potential race condition could arise during program termination (*2.6 Thread termination order*) if the main thread running a federate exits before the thread running the RTI, and thus hinders the RTI from notifying other federates to terminate. This means that the main thread must also wait for the RTI to exit, before exiting.

Since the RTI uses some of the same functions in the federated utility file as federates, and this file has global variables, then the implementation must be careful to handle this accordingly.

4.6.2 RTI Command-line arguments

The RTI program takes five command line arguments (*2.5.2 RTI arguments*), where one is mandatory. The RTI must be started with an option specifying the total number of federates in the program. Federation ID, port number, clock synchronization and authentication can be specified additionally. How to give these arguments to a joint RTI structure is an interesting design choice. The most reasonable way to do this is to set as many options automatically, using information that is already available through compiler definitions or other variables in the LF compiler. If the option cannot be set automatically, then it should probably be set through a joint RTI options structure in the target declaration, like in Figure 4.3.

```
1  target C {
2      platform: Zephyr,
3      joint-rti: true,
4      joint-rti-options: {
5          federation-id: ExampleFederation,    // new target property
6          port: 8080,                          // new target property
7      }
8  }
9
```

Figure 4.3: The figure shows how a target property structure should be added to Lingua Franca to allow setting RTI options.

Chapter 5

Implementation

This section will give an overview of the project structure, setup and implementation. The full code is available at the following repositories on Github: lf-west-template, lingua-franca and reactor-c.

5.1 Project Structure

The project stretches out across three repositories: a forked Lingua Franca repository, a forked reactor-c repository and a forked Zephyr template repository. All these repositories are part of the Lingua Franca project. The lingua franca repository contains the LF compiler, command-line tools, automated testing, IDE files, and all target runtimes. The reactor-c repository is the C runtime. The Zephyr template repository is a basic setup for building LF applications with West and Zephyr. The template repository has been expanded on for this project.

The Zephyr template repository has been restructured as shown in Figure 5.1. The top-most structure mostly contains Zephyr-related files, such as manifest files, custom west commands, and the Zephyr repository itself. User applications are put in the application folder. Inside each user application folder, there will be a *src* folder containing the LF source code and quite a few configuration files for Zephyr. As explained in Section 2.2.2, Zephyr uses two configuration systems.

DeviceTree is the hardware configuration system, and as the board in this project is already supported by Zephyr, most basic hardware configurations are done already. The only necessary configurations are those that are specific to an application, such as using an additional LED, GPIO or specifying properties in the ethernet interface. These types of configurations can be made in overlay files which overwrite certain hardware properties. As can be seen in Figure 5.1, there are two overlay files, one for each federate. The reason for this is explained in Section 4.4.5.

Kconfig is the software configuration system. The software configuration files specify which libraries and functionalities to enable, as well as a range of properties. Each federate has its own configuration file, as argumented for in Section 4.4.4. A *debug.conf* file provides additional configurations that help with debugging, such as configurations related to logging and statistics.

To make building and flashing federated programs easier, two custom west commands

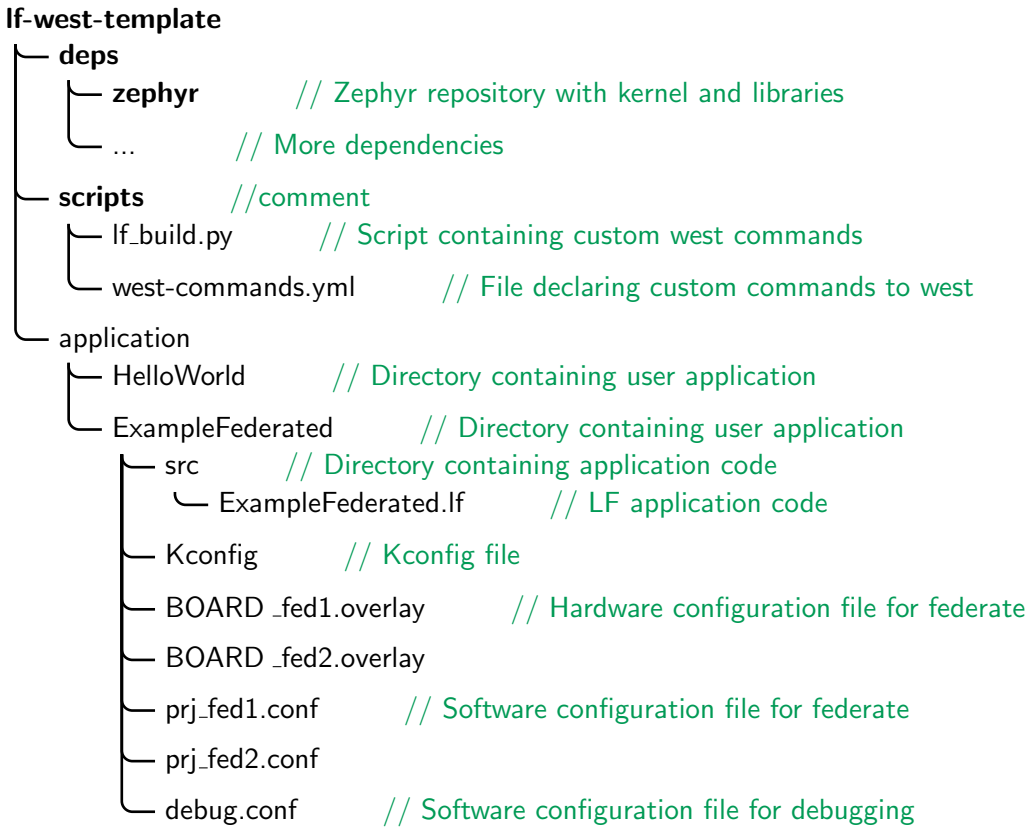


Figure 5.1: The figure shows the application template structure.

have been added to the repository. They can be inspected in the *lf_build.py* file under the *scripts* folder.

5.2 Custom commands for West

While the custom command *lf-build* was a pre-existing command created for the *lf-west-template* repository, two additional commands were added in this project. The first one *lf-fed-build*, builds directly from *lf-build*, but has been altered for federated programs. Essentially, the command finds the names of the federates, copies all necessary configuration files and other files to each federate folder, and then builds each federate sequentially. An option to specify which federates to build was added for convenience when dealing with multi-platform systems. The command and option may be used to build two federates, for example, as follows.

```
$ west lf-fed-build src/ExampleFederated.lf -w "-b mimxrt1170
  _evk_cm7 -p always" -f fed1 fed2
```

Additionally, debug configurations will be added to the build if adding the option "-c debug.conf". An option to not recompile LF code was added, "-no-lfc/-n", mostly for development convenience. It makes it easier to test auto-generated code without having to change the actual LF compiler, or to modify the main function if deploying federation topologies 3 or 4.

The command *lf-fed-flash* was added to make flashing to multiple boards easy. This

command recognizes how many boards are connected to the computer, and collects their unique IDs. It then flashes a specified number of federates to the connected boards. This command can be expanded to launch federations in different topologies, and start the RTI. The command can be used to flash three federates, for example, as follows.

```
$ west lf-fed-flash -n 3
```

The code creating both commands can be inspected in the file *lf_build.py* under the *scripts* folder.

5.3 Project Setup

5.3.1 lf-west-template Setup

Lingua Franca should be set up as described in the Lingua Franca Handbook at [37], which is simply cloning the lingua-franca repository, and initializing the submodules. For simplicity, the commands are repeated below.

```
$ git clone git@github.com:lf-lang/lingua-franca.git
$ cd lingua-franca
$ git submodule update --init --recursive
```

The path to the LF compiler should be added to your path environment variable, so that is it available simply as *lfc*. Do this by adding the following line (modified with the correct path to the LF repository) to your *.bashrc* file.

```
$ export PATH="<YOUR_PATH>/lingua-franca/bin:$PATH"
```

The LF compiler must be built before it can be used, and when changes have been made to the compiler itself. This can be done by running the following command.

```
$ sudo <YOUR_PATH>/lingua-franca/bin/build-lf-cli
```

To set up Zephyr and west for this project, follow the guide in lf-west-template's README. The guide mostly follows the steps in Zephyr: Getting Started Guide in Zephyr's documentation pages, except for where the Zephyr repository is cloned to. It is recommended to install Zephyr in a virtual environment to avoid Python dependency incompatibilities. It should be assumed that this environment is activated for the whole of this project.

5.3.2 Debug and Flashing Setup

Setting up debug and flashing functionality is essential to any embedded project. As explained in Section 2.8, the board is pre-packed with a full debugging system. The out-of-the-box firmware is DAPlink (preceded by CMSIS-DAP), which is an open-source debug interface firmware. There are several debug host tools that are possible to use in this project. This project uses the pyOCD host tools, which are already compatible with DAPlink firmware. However, if Jlink host tools are preferred, then the firmware must be changed. How to change the firmware and set up JLink host tools are covered in Section 5.4.1 in [7]. The rest of this subsection covers how to set up debugging with pyOCD.

Using PyOCD without any setup could be possible, and should be tested first. Run the following commands to flash a hello world sample program, with the runner specified to be pyOCD.

```
$ cd deps/zephyr/samples/hello_world
$ west build src/main.c
$ west flash -r pyocd
```

The following error could occur: "[Errno 13] Access denied (insufficient permissions) while trying to interrogate a USB device. This can probably be remedied with a udev rule. See https://github.com/pyocd/pyOCD/tree/master/udev for help.". This means that pyOCD can not access the debug probe, as permissions to access USB devices must be set up explicitly in Linux [42]. This is done via udev rules. To add a udev rule, create a new file as follows

```
$ cd /etc/udev/rules.d
$ sudo touch 50-cmsis-dap.rules
```

Then, copy the contents from the file 50-cmsis-dap.rules from pyOCD's Github repository into the newly created file. Reload the new rules by running

```
$ sudo udevadm control --reload
$ sudo udevadm trigger
```

It should then be possible to run the flash command. Similarly, debugging may be started with

```
$ west debug -r pyocd
```

If several boards are connected, the board to debug must be specified through its unique ID which can be found by running:

```
$ pyocd info
```

5.3.3 Networking Setup for Topology 1

Linux-side Required Setup

To connect the sockets of an external federate (running on the MCU) and the RTI (running on the Linux machine) both Zephyr configuration and network interface configuration with Linux is necessary. Firstly, the ethernet interface connecting the physical units must be initialized with IP addresses on both ends. Check all network interface addresses on the Linux machine by running

```
$ ip address
```

The relevant ethernet interface should start with an 'e'. If several possible interfaces appear, then it is possible to check further hardware information (like connection info) on the suspected interface with the command

```
$ sudo dmesg | grep -i <ethernet interface name>
```

This should make it possible to differentiate different ethernet interfaces. The Linux machine side can then be assigned an address as follows

```
$ sudo ip address add dev <ethernet interface name> 192.0.2.2
```

To verify the interface address, check the interface addresses again. Then, add an entry to the routing table with gateway and netmask information, with the following command. This is needed in order to correctly route the data traffic.

```
$ sudo ip route add 192.0.2.0/24 dev <ethernet interface name>
```

The routing table entries can be checked with

```
$ ip route
```

Take special note that these changes are not necessarily persistent, and will be lost with a restart, unless the commands are added to the user's script file or the Linux distribution configuration files.

Board-side Required Setup
On the federate application side, the IP address, gateway address and netmask must be set with Zephyr configuration options, see table 5.4 discussed in section 5.5. The IP addresses in the network must be in the same subnet. The subnet is defined by the gateway address and netmask.

Depending on the number of federates, the number of file descriptors and maximum network connections might need to be adjusted.

```
# Max possible network connections
CONFIG_NET_MAX_CONN=16
CONFIG_NET_MAX_CONTEXTS=16
# Max open file descriptors
CONFIG_POSIX_MAX_FDS=16
```

5.3.4 Networking Setup for Topology 2

Setting up two external federates (one for each board) involves all the same steps as in section 5.3.3. Each board must be assigned its own IP address. This can be done by creating a configuration file for each federate that will run on a board. Then, simply specify a unique IP address inside the same subnet as the other addresses.

The MAC address can be set in a DeviceTree overlay file per federate, see section 4.4.5 for why this particular solution was made. An excerpt from an overlay file is shown in Figure 5.2.

```
1 &enet {
2     local-mac-address = [00 0a 35 00 00 01];
3     status = "okay";
4 };
5
```

Figure 5.2: The figure shows how to specify the MAC address of the ethernet interface using a DeviceTree overlay file.

5.3.5 Networking Setup for Topologies 3 and 4

Setting up the joint RTI structure requires the same steps as in the two previous topologies. Some additional configurations are needed. Particularly, the RTI sets timeout values for each socket, and this needs to be enabled in KConfig options.

```
# Socket timeout configuration
CONFIG_NET_CONTEXT_RCVTIMEO=y
CONFIG_NET_CONTEXT_SNDTIMEO=y
```

The number of file descriptors and network connections might need to be adjusted when using the joint RTI topologies. This is done by adjusting the options *CONFIG_NET_MAX_CONN*, *CONFIG_NET_MAX_CONTEXTS* and *CONFIG_POSIX_MAX_FDS* from tables 5.1 and 5.2.

5.4 Build and Run

Run Topology 1
As of now, there is no integrated way to specify a platform for each federate, so if launching federates on different platforms, then the program's platform needs to be modified and rebuilt. For an example with two federates, one running on Linux and one on a board, first build your LF program with the following target declaration.

```
target C {
    platform: "Zephyr",
    no-compile: true,
}

```

Use the custom west commands discussed in section 5.2, specifying the federate that should run on a board.

```
$ west lf-fed-build src/ExampleFederated.lf -w "-b <BOARD>
-p always" -f fed1
```

To flash it to the board, run

```
$ west lf-fed-flash -n 1
```

Then, modify the target declaration within your LF application file.

```
target C {
    platform: "linux",
}

```

Build using the normal *lfc* command:

```
$ lfc src/ExampleFederated.lf
```

Now, launch the RTI by

```
$ RTI -n 2 -i "Unidentified Federation"
```

Launch the board federate by pressing the board reset button, and Linux federate with

```
$ fed-gen/ExampleFederated/bin/federate__fed2
```

Both federates should then connect to the RTI at the specified IP address and port (default is 15045). Execution should then start as normal.

```
Federate 1: Successfully connected to RTI.
Federate 1: Connected to RTI at 192.0.2.2:15045.
```

Run Topology 2
Only running board federates is easier than topology 1. The target declaration should be at least this:

```
target C {
    platform: "Zephyr",
    no-compile: true,
}

```

Use the custom build command without needing to specify federate

```
$ west lf-fed-build src/ExampleFederated.lf -w "-b <BOARD>
-p always"
```

To flash it to the board, run

```
$ west lf-fed-flash -n 2
```

Now, launch the RTI by

```
$ RTI -n 2 -i "Unidentified Federation"
```

Launch the board federates by pressing the board reset buttons. Both federates should then connect to the RTI at the specified IP address and port (default is 15045). Execution should then start as normal. Output from the boards is from Putty (see Section 2.7).

```
Federate 1: Successfully connected to RTI.
Federate 1: Connected to RTI at 192.0.2.2:15045.
```

Run topologies 3 and 4
Running topologies 3 and 4 involves the same ideas as in the previous sections, depending on the platform of the federates. Because of time limitations, the joint RTI target properties envisioned in Section 4.6 and 4.6.2 have *not* been implemented. The joint RTI functionality can still be achieved by specifying the correct IP address and port in the LF code, then generating target code using *lfc*. Then, the main function in one of the federates must be overwritten by a new main function, shown in Section 5.7 and provided fully in Appendix A.

5.5 Zephyr Configurations

Configuration options in Zephyr are very important, either if it's enabling crucial features or allocating enough memory - it could make or break your application. It is therefore necessary to go through these options in more detail. All configuration options necessary for

federated applications with Zephyr (except for the basic options that are not application-specific) are listed in tables 5.1, 5.2, 5.3 and 5.4. Some options have the same value as the default, but they have been included for clarity.

Table 5.1 contains general configurations. As can be seen in the table, the main stack size is set to be 4096 bytes. This is where all initialization prior to the application will run, and eventually main() will run. It must be larger than the default since the LF application (with runtime) is quite large. If the main thread is too small, a "Data access violation" CPU exception will arise, or the application will not get through Zephyr's initialization phase. The test random generator option is necessary when enabling networking. By enabling this option, the kernel's random number generator is allowed to use non-random generation sources, even if a true entropy generator is available. Where possible, the true random number generator will be preferred. If, in the future, federated LF on Zephyr will be expanded to include authentification, then this option should be looked more into. Further, Zephyr's POSIX API is enabled, as discussed in section 4.4.1. The last option relates to the available number of file descriptors, which in this case relates to the number of sockets.

Zephyr Configuration Option Name	Description	Default value	Chosen value
CONFIG_MAIN_STACK_SIZE	Size of stack for initialization and main thread	1024	4096
CONFIG_TEST_RANDOM_GENERATOR	Non-random number generator is OK	n	y
CONFIG_POSIX_API	Use available POSIX APIs	n	y
CONFIG_POSIX_MAX_FDS	Maximum number of open file descriptors	16	32

Table 5.1: The table shows Zephyr configuration options related to general options.

Table 5.2 contains general networking configurations. Federated LF uses both TCP and UDP communication (see points *1.2.1/1.3.1/2.3.1 TCP socket support* and *1.2.1/2.3.2 TCP socket support* from specification Tables 3.1 and 3.2 in Section 3), and therefore needs both enabled. The application uses the socket API. Ethernet must be enabled. IPv6 is not available yet, and thus we must use IPv4 (See Section 4.4.4). The max connections and contexts must be increased since there will be multiple connections simultaneously. These numbers should be the same.

Table 5.3 contains networking configurations related to buffers. These options all deal with the number of possible packets to cache, either in send or receive. These options have implications for how data will be sent, and can generate errors in the application if the resources are insufficient.

Table 5.4 contains IP address configurations. To set the IP address, gateway and netmask, the first setting must be enabled.

Table 5.5 contains debugging options. The type of compiler optimization can be configured. Zephyr offers many types of logs, here only the networking log is included. Enabling logs makes more information be printed to the serial output. A shell functionality

Zephyr Configuration Option Name	Description	Default value	Chosen value
CONFIG_NETWORKING	Enable link layer and networking	n	y
CONFIG_NET_UDP	Enable UDP	y	y
CONFIG_NET_TCP	Enable TCP	y	y
CONFIG_NET_IPV4	Enable IPv4	n	y
CONFIG_NET_SOCKETS	Enable BSD socket lAPI	n	y
CONFIG_NET_L2_ ETHERNET	Enable ethernet support	n	y
CONFIG_NET_MAX_ CONN	Number of supported network connections, including TCP and UDP	8	16
CONFIG_NET_MAX_ CONTEXTS	Number of allocated network contexts	6	16

Table 5.2: The table shows Zephyr configuration options related to networking.

Zephyr Configuration Option Name	Description	Default value	Chosen value
CONFIG_NET_PKT_RX _COUNT	Max number of pending packet receives	14	32
CONFIG_NET_PKT_TX _COUNT	Max number of pending packet sends	14	32
CONFIG_NET_BUF_RX _COUNT	Number of network buffers allocated for receiving data	36	128
CONFIG_NET_BUF_TX _COUNT	Number of network buffers allocated for sending data	36	128
CONFIG_NET_BUF_ FIXED_DATA_SIZE	Each network buffer has fixed size	n	y
CONFIG_NET_BUF_ DATA_SIZE	Size of network buffer	128	1024

Table 5.3: The table shows Zephyr configuration options related to network buffers.

can be enabled. This makes it possible to use the serial communication client as input for some select commands, for example, to print network configuration info from the board. The command to list network interface information is

```
$ net iface
```

Lastly, the stack initialization option is useful if analyzing register values and addresses.

Zephyr Configuration Option Name	Description	Default value	Chosen value
CONFIG_NET_CONFIG_SETTINGS	Allow setting network application settings in config file	n	y
CONFIG_NET_CONFIG_NEED_IPV4	Application needs IPv4	n	y
CONFIG_NET_CONFIG_MY_IPV4_ADDR	My IP addr	-	<FED_IP>
CONFIG_NET_CONFIG_MY_IPV4_GW	My IP gateway	-	192.0.2.0
CONFIG_NET_CONFIG_MY_IPV4_NETMASK	My IP netmask	-	255.255.255.0

Table 5.4: The table shows Zephyr configuration options related to IP address configuration.

Zephyr Debug Configuration Option Name	Description
CONFIG_DEBUG_OPTIMIZATIONS	Enable compiler debug optimizations
CONFIG_SHELL	Enable Zephyr's shell functionality
CONFIG_NET_SHELL	Enable network shell commands
CONFIG_LOG	Enable logging
CONFIG_NET_LOG	Enable logging from networking
CONFIG_INIT_STACKS	Initialize the stack to a known value (0xaa)

Table 5.5: The table shows Zephyr configuration options useful for debugging.

5.6 Zephyr Platform Support for Federate

Most of the runtime code was directly portable across the platforms because of the POSIX library configuration option explained in Section 4.4.1. Only macros to handle libraries for the Zephyr platform, and for other platforms, were added. The RTI connection function discussed in Section 4.4.2 was implemented.

5.6.1 RTI Connection Function

As discussed in Section 4.4.2, the function handling the RTI socket connection had to be adapted to use *getaddrinfo()*. A pseudocode of the change is included in Figure 5.3. The new function uses a *hints* structure which constrains which address structures are returned from the function. These hints are specified to be IPv4 with TCP stream sockets. Refer to Section 2.3 for what the other functions do.

```
...
struct addrinfo hints;
struct addrinfo *res;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;          /* Allow IPv4 */
hints.ai_socktype = SOCK_STREAM;    /* Stream socket */
hints.ai_protocol = IPPROTO_TCP;    /* TCP protocol */
hints.ai_addr = NULL;
hints.ai_next = NULL;
hints.ai_flags = AI_NUMERICSERV;    /* Allow only numeric port numbers */

while (result < 0) {

    // Get address structure matching hostname and hints criteria, and
    // set port to the port number provided. There should only
    // ever be one matching address structure, and we connect to that.
    int server = getaddrinfo(hostname, &port, &hints, &res);

    // Create a socket matching hints criteria
    _fed.socket_TCP_RTI = socket(res->ai_family, res->ai_socktype, res->
    ai_protocol);

    result = connect(_fed.socket_TCP_RTI, res->ai_addr, res->ai_addrlen);
    if (result == 0) {
        // Successfully connected to RTI
    }
...     // function continues

```

Figure 5.3: The figure shows the pseudocode of an excerpt of the RTI connection function. Some details are left out for brevity.

5.6.2 Static Thread Stack Definiton

As discussed in Section 4.4.3, statically allocating memory for thread stacks was implemented through a number of macros that define the final thread count. The final macro is shown in Figure 5.4. The macros not shown in the figure were decided from a compiler definition such as how many federates there were, and also from options being enabled or disabled, such as clock synchronization.

```

#define NUMBER_OF_THREADS (NUMBER_OF_WORKERS \
                           + WORKERS_NEEDED_FOR_FEDERATE \
                           + RTI_SOCKET_LISTENER_THREAD \
                           + FEDERATE_SOCKET_LISTENER_THREADS \
                           + P2P_HANDLER_THREAD \
                           + CLOCK_SYNC_THREAD \
                           + USER_THREADS)


```

Figure 5.4: The figure shows how the NUMBER_OF_THREADS macro is altered to allow static thread stack allocation for federated LF on Zephyr.

5.7 Zephyr Platform Support for the RTI

Most of the RTI code was directly portable across the platforms because of the POSIX library configuration option explained in Section 4.4.1. Only macros to handle libraries for the Zephyr platform, and for other platforms, were added, as explained in Section 4.5. Some naming conflicts with *federate.c* resulted in function name changes.

The abstractions for sleep functionality, thread synchronization and threads, explained in Section 4.5 have all been implemented, so that the RTI has become platform independent.

5.8 Adapted Main Function for RTI

Figure 5.5 shows the adapted main function launching the federated program and the RTI program. Notice how macros are used to decide the clock synchronization scheme, clock synchronization parameters and number of federates automatically. The port number and federation ID are hard-coded. The complete code is provided in Appendix A.

```
1
2  void main(void) {
3
4      // Set arguments to the RTI
5      #ifdef _LF_CLOCK_SYNC_INITIAL
6      rti_set_args(NUMBER_OF_FEDERATES, "Unidentified Federation", 15045,
       clock_sync_init, CLOCK_SYNC_PARAMS);
7
8      #elif _LF_CLOCK_SYNC_ON
9      rti_set_args(NUMBER_OF_FEDERATES, "Unidentified Federation", 15045,
       clock_sync_on, CLOCK_SYNC_PARAMS);
10
11     #else _LF_CLOCK_SYNC_OFF
12     rti_set_args(NUMBER_OF_FEDERATES, "Unidentified Federation", 15045,
       clock_sync_off, CLOCK_SYNC_PARAMS);
13     #endif
14
15     // hardware initialization
16     lf_initialize_clock();
17
18     // Create thread for RTI
19     lf_thread_create(lf_rti_main);
20
21     // Call normal runtime main
22     res = lf_reactor_c_main();
23     exit(res);
24 }
25
26
```

Figure 5.5: The figure shows the pseudocode of the new main function in the joint RTI structure.

As described in Section 4.6, a new RTI main function had to be added. This function is available as *lf_rti_main* in the *rti_lib.c*, and is visible in the code in Figure 5.5. It is almost identical to the original main function, with the exception that it does not handle command-line arguments. Instead, a function to set these arguments automatically was added (explained in Section 4.6.2, shown in Figure 5.5). This function takes arguments

specified from the new main function. Hardware initialization is done prior to anything else, as explained in Section 4.6.1.

5.8.1 Integration with LF Language

The joint RTI functionality remains highly experimental, and has not been integrated with the LF language, as envisioned in Section 4.6 and 4.6.2. The reason for this is that it was not prioritized, and is therefore left for future work. The consequence of this is that it is not well-tested and will be more difficult to use for future contributors.

How to set it up experimentally is explained in Section 5.4. Because of its experimental nature, it is not included in Chapter 6.

5.9 Zephyr Fatal Errors and Causes

If the federated program fails, it is often because of a fatal error from Zephyr. Since these errors are relatively common, they will be listed here for the reader's convenience.

During development:

- Unaligned memory access: Something is not initialized before use.
- Illegal use of the ESPR: A socket handle is incorrectly accessed. This could be because it has not been initialized.

Running an application:

- Buffer allocation failed, followed by connection timeout. Buffer options must be modified, see section 5.5.
- Data access violation: The thread indicated by fault has a too-small stack. Increase the stacks in configuration options or, for LF threads, inside the lf_zephyr_support.c in the runtime.
- Precise data bus error: Federate ID fails to send. Board federates must connect first to RTI.
- Illegal use of the ESPR: A socket handle is incorrectly accessed. This could be because it has already closed.

5.10 Github Contributions

The following has been merged into the main LF repository:

- Platform abstraction replacing POSIX functions in RTI, further simplifying RTI logic
- Implementation of a target property allowing specification of user threads in LF programs on Zephyr

The following has been added in a pull request to the main LF repository:

- Obsolete function mentioned in Section 5.6 replaced with new alternative, simplifying the logic
- Federated LF port to Zephyr platform
- Federation examples, new west commands and change of structure in lf-west-template

Chapter 6

Evaluation and Results

This section presents tests to evaluate the implementation of Zephyr platform support for federated LF and its results. First is the clock synchronization functionality, then two different aspects of message sending and finally a larger example concerning smart grids. Figure 6.1 shows how the hardware is set up during the evaluation tests.

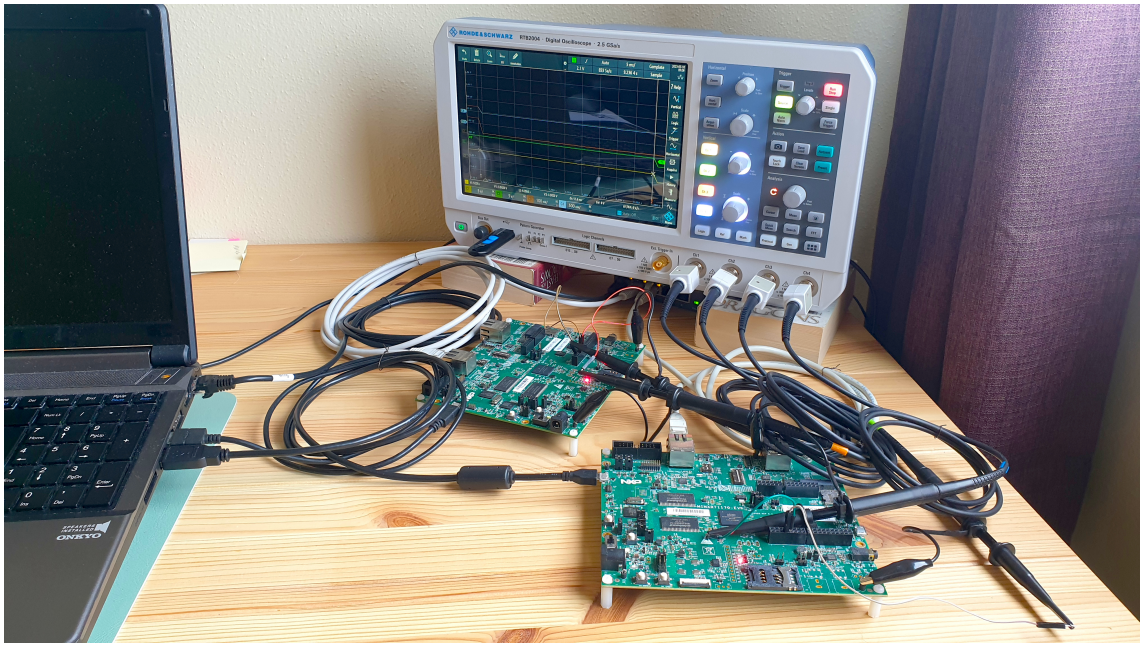


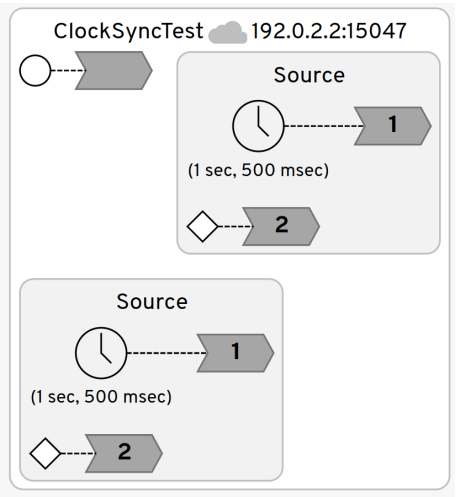
Figure 6.1: The figure shows the hardware setup.

6.1 Clock Synchronization Test

The clock synchronization functionality is important for distributed programs. To test clock synchronization, the simple test in Figure 6.2b was made. The test simply consists of two instances of a reactor that toggles a GPIO pin on each tick of the timer. The timer is set to a period of 500 milliseconds. The GPIO pin toggles can then be measured and compared with an oscilloscope. This is a measure of clock synchronization and of drift, since the program should ideally toggle these GPIOs exactly at the same time. The difference will be the error in how the local clocks are synchronized.

```
target C {
    platform: Zephyr,
    clock-sync: off/init/on,
    clock-sync-options:
    {local-federates-on: true,
    period: 500 msec},
}
```

(a)



(b)

Figure 6.2: The figure shows the target declaration in (a) and the LF diagram in (b) for the clock synchronization test.

This test is included to both test the LF clock synchronization functionality, as all the tests rely on some level of clock synchronization, and to give preliminary results as to the achievable precision. The target declaration of the test is shown in Figure 6.2a.

6.1.1 Clock Synchronization Test Results

clock-sync: off
As expected, turning off LF clock synchronization yields no clock synchronization, as there is no underlying clock synchronization present.

clock-sync: on
The test revealed that this feature is not currently working. This is unsurprising as this feature has not been prioritized during this project. Looking at the network interaction over Wireshark showed that UDP messages were sent, but that probably the UDP messages are interpreted incorrectly by the runtime.

clock-sync: init
The result of the test over 60 seconds is shown in Figure 6.3. Only every second toggle is measured by the oscilloscope, meaning there is a total of 60 data points. The figure presents the data points, but also a trend line found using linear regression.

The initial synchronization is in the sub-millisecond order. This is consistent with the reported clock synchronization schemes over LANs, presented in Section 2.4.2. This result is quite interesting, and the potential for consistently precise clock synchronization is present given that this step is repeated periodically, such as in *clock-sync: on*.

The drift after initial synchronization is about 26 microseconds per second. After one minute, the clock synchronization error between the boards is still under 2 milliseconds. It seems like using initial clock synchronization will yield acceptable results in an experimental phase and with short-lived tests.

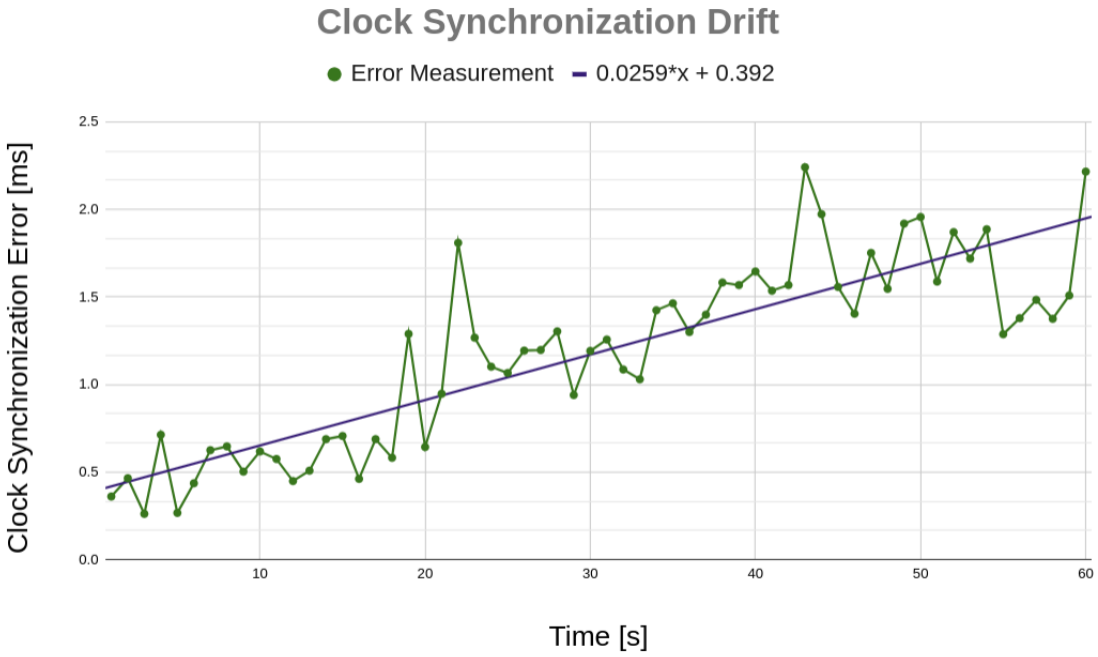


Figure 6.3: The figure shows measurements over 60 seconds of clock synchronization error after an initial synchronization round.

6.2 Message Sending Test

A test with a minimal, sender-receiver structure is interesting to test. See the LF diagram of the test in Figure 6.4 and a pseudocode description in Figure 6.5. The two main aspects to test are

- Message delay, ie. the time from the end of the sending reaction until the execution of the receiving reaction, with centralized and decentralized coordination enabled
- Message throughput, ie. a measure of how many messages can be sent (and processed correctly) per time unit, with centralized and decentralized coordination enabled

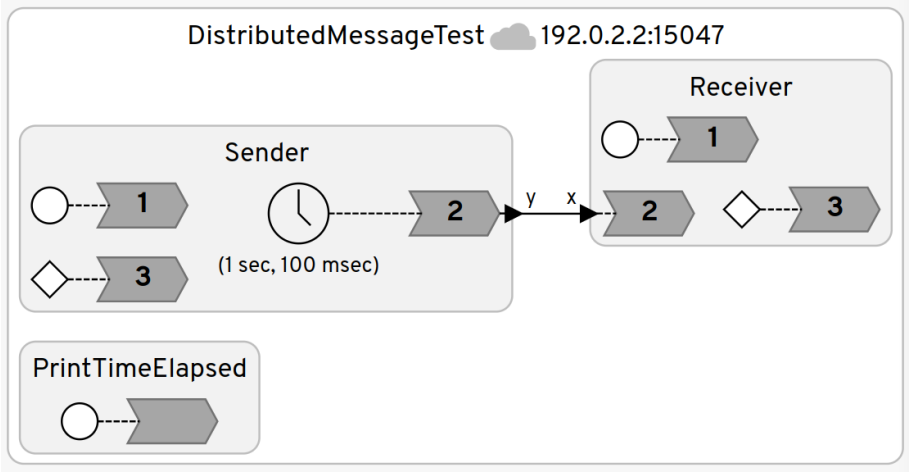


Figure 6.4: The figure shows an LF diagram of the message-sending test.

```
reactor Sender(period: time(100 msec), iterations:int(3000)) {
    output y: int
    timer t(1 sec, period)
    state sendCount: int(0)

    reaction(t) -> y {=
        sendCount++;
        gpio_pin_toggle();
        lf_set(y, sendCount);
        if (sendCount == iterations) {
            lf_request_stop();
        }
    =}
}

reactor Receiver {
    input x: int
    state receiveCount: int(0)

    reaction(x) {=
        receiveCount++;
        if (receiveCount != x->value) {
            // Received the wrong message
            lf_request_stop();
        } else {
            gpio_pin_toggle();
        }
    =}
}

```

Figure 6.5: The code snippet shows a pseudocode description of the test.

6.2.1 Message Delay test

For measuring message delay, the timer periods in Figure 6.5 is set to be 100 milliseconds. The reason for this is that it should be apparent which timer toggles correspond to which when measuring the output. The oscilloscope is configured to measure the delay between the rising and falling edges of the square signals.

The expected message delay is in the millisecond range, because TCP communication over ethernet takes some time, as well as the runtime itself. It is also expected that centralized coordination will have a longer delay than decentralized coordination. This is because more messages need to be sent (RTI control messages) and all messages need to go through an extra device.

Test Results
The test results from examining message delay are shown in the histogram in Figure 6.6. As expected, decentralized coordination has a lower delay overall, by about 2 milliseconds. It has a relatively low variance as well. This makes sense because there are not many other tasks running on the system. In comparison, centralized coordination has a high variance, and possibly a multimodal distribution. Centralized coordination must have the RTI as a middle step in communication, and the RTI runs on a general-purpose operating system running thousands of other tasks. Therefore, the delay introduced will vary highly. Since it must go through another device, a multimodal distribution could make sense. Another system introduces a second independent stochastic variable.

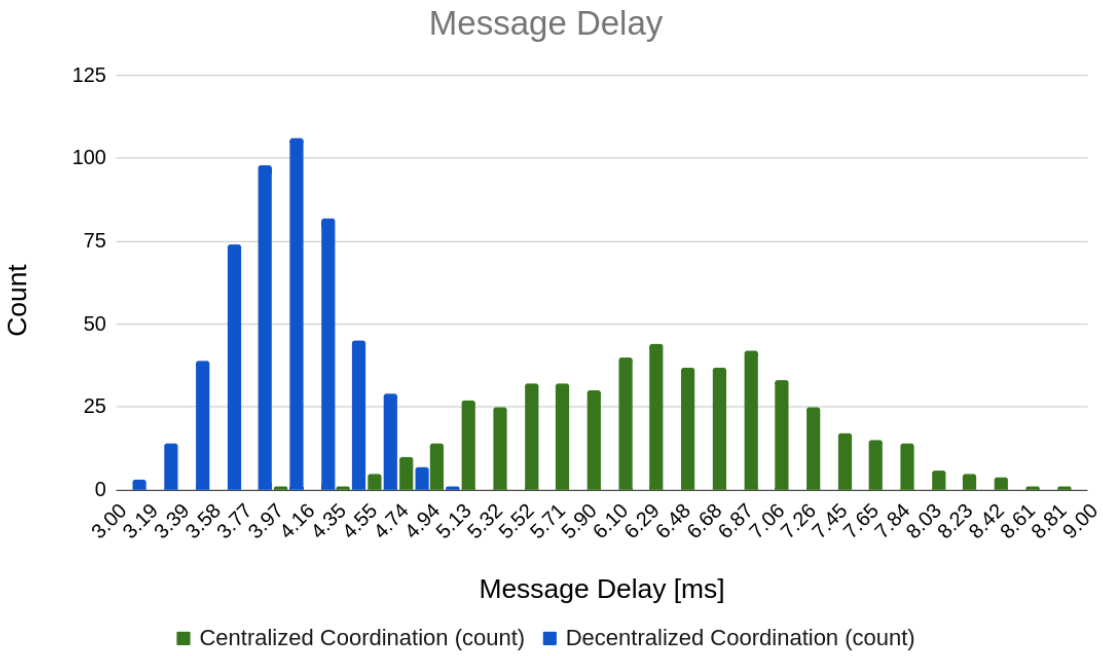


Figure 6.6: The figure shows measurements of message delay in a simple source-drain structure.

6.2.2 Message Throughput test

For measuring message throughput, the timer periods in Figure 6.5 are set to be 100 microseconds, which is much faster than the expected throughput. This is done to "push" the system to see what the maximum throughput can be. The test sends a count value that is compared to a local count to ensure that no messages are skipped or in the wrong order. The test is set to finish 3000 iterations, ie. send and receive 3000 messages. The time to finish this is then measured by adding a third reactor, running on the Linux machine, to the federation. The elapsed physical time is then used to calculate the number of messages per second.

Message throughput will be limited by the runtime overhead of sending and receiving messages. The network stack provided by Zephyr will also have a certain overhead affecting this measure. Decentralized coordination is expected to perform better than centralized coordination in this measure, since control messages have a limiting factor.

Test Results
The result from 10 iterations of the message throughput test is shown in Table 6.1. The average throughput is slightly higher for centralized coordination, than for decentralized coordination, see the averages at the bottom of Table 6.1.

This is a surprising result, as the RTI should be an extra step compared to direct communication, in addition to that control messages need to be sent back and forth. However, in a simple source-drain structure, there are not many control messages necessary at all. The source has no upstream federates, and can therefore execute without receiving TAG messages from the RTI. What is likely occurring is that the source reactor executes all sending operations as fast as it can, limited by the network communication buffer structures (See configuration options in Section 5.5, Table 5.3). The RTI receives the messages,

Message Throughput rates [messages/second]					
	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
Centralized	604.76	619.56	614.00	616.52	599.82
Decentralized	583.43	592.86	585.22	589.41	601.12
	Iteration 6	*Iteration 7*	*Iteration 8*	*Iteration 9*	*Iteration 10*
Centralized	587.96	581.17	674.53	628.42	589.79
Decentralized	583.03	588.81	590.12	588.02	586.82
	Average		**Standard Deviation**		
Centralized	611.65		26.84		
Decentralized	588.88		5.27		

Table 6.1: The Table shows results from the message throughput test. The average value and standard deviation over the 10 iterations are shown at the bottom.

and forwards them to the drain reactor, as fast as possible, but is limited by the same network buffers. In the decentralized coordination mode, the messages are forced to be sent at regular intervals. The waveforms measured by the oscilloscope, in Figures 6.7 and 6.8, is consistent with this hypothesis. The waveform for the centralized coordination run in Figure 6.7 shows that the GPIO toggles have been grouped together, both in the sending end and the receiving end. In the waveform for the decentralized coordination run in Figure 6.8, the GPIO toggles are more evenly distributed.

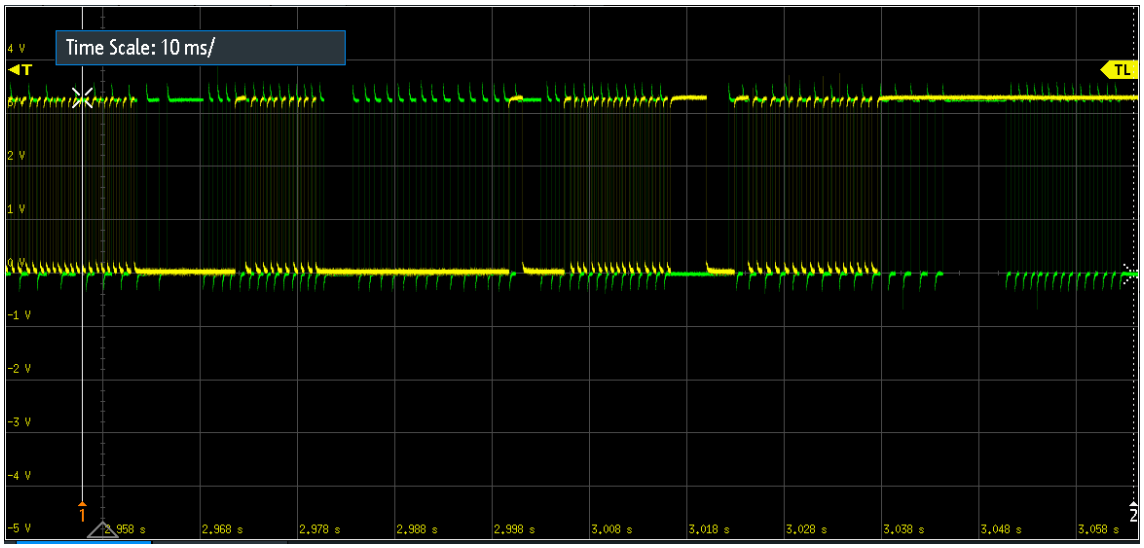


Figure 6.7: The figure shows the generated square waveform from the message throughput test using centralized coordination. The y-axis shows the voltage (though it should be thought of as a logical signal) and the x-axis shows time.

It is worth mentioning that this test gave errors like "Cannot allocate TCP buffer of size x" if the network buffer configurations were a certain way. Probably, setting each network packet buffer to be large, specifically 2048 bytes, caused the traffic to be forced to slow down, thus avoiding the errors. This size is larger than the maximum Ethernet packet payload size, which is 1500 bytes [30]. Thus every message must be sent in two packets. There seems to be no network flow regulation mechanism implemented, so buffers are likely to overflow. It is also worth mentioning that the grouped traffic, as in centralized coordination, is not ideal program behavior, and should be investigated further to find a solution to preventing this. However, setting the period of the timer to be faster than

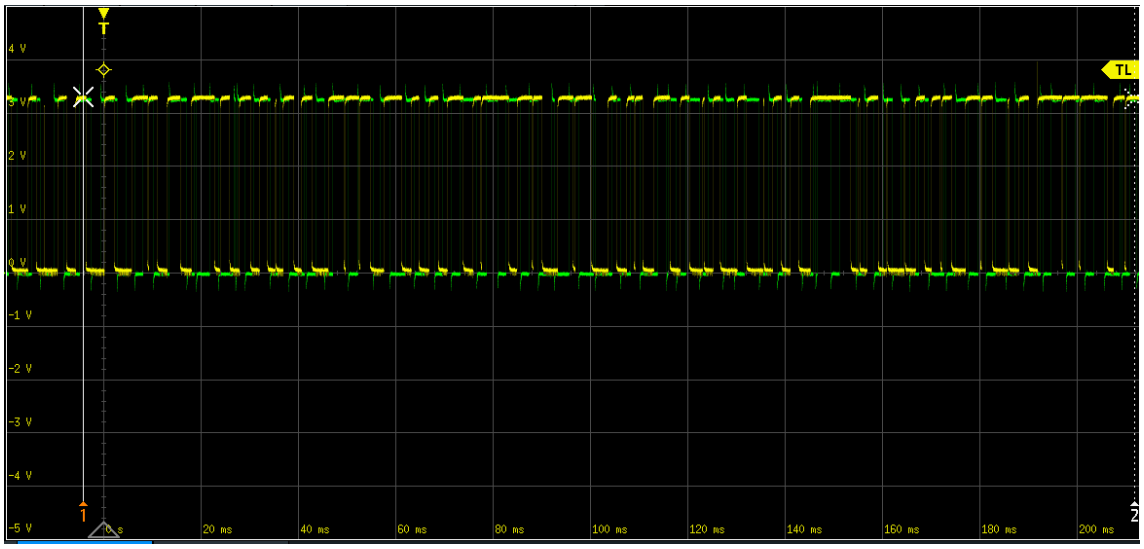


Figure 6.8: The figure shows the generated waveform from the message throughput test using decentralized coordination. The y-axis shows the voltage (though it should be thought of as a logical signal) and the x-axis shows time.

what the system can handle is generally a bad idea.

It was suggested during the presentation to the LF team that this behavior could be due to network switch flow control actions. In that case, it would be an ethernet flow control mechanism kicking in. However, this was investigated using Wireshark with a filter on ethernet pause frames: *macc.opcode == pause*. No such messages were exchanged during a run of the test.

The standard deviations at the bottom of Table 6.1 show that decentralized coordination gave substantially less result variation. This is consistent with what was observed in the message delay test (Figure 6.6); that the RTI introduces significant execution time variance.

6.3 Example: Smart Grid

6.3.1 Case Description and Motivation

The importance of our electrical grids is larger than ever, making them at the same time more vulnerable than ever. Equipment failures, vegetation and weather conditions, human errors and cyber attacks pose significant risks to the reliability and stability of our grids. Smart electrical grids aim to remedy the modern challenges of the electrical grid using instrumentation and automation technology. Particularly, a self-healing system is envisioned where the system itself will be able to detect, localize, isolate and reconfigure the grid to minimize the impact of faults in the grid. These tasks are far from trivial, and require cutting-edge technological solutions.

Detection of a fault comes from a measurement unit. Phasor measurement units (PMUs) measure voltage and current phasors, usually at 50 Hz in Norwegian grids [43]. PMUs are possible to use for real-time monitoring over wide areas because of low latency measurement reporting and millisecond-range time synchronization. PMU measurements are

reported to Phasor Data Concentrators, which can retransmit the data to other units. In this example, since PMU data is hard to find and to reduce complexity, a simple time series of possible voltage measurements will be used.

The localization of a fault is made possible by comparing various aspects of reported measurements from distributed units, for example by a human operator. A promising method to do this would be to use an online ML-based algorithm [43]. For the sake of this example, the localization aspect will be simply comparing the amplitudes of voltage measurements, under the assumption that higher amplitudes will relate to closer electrical proximities. This is somewhat realistic to do for the same geographical area. For widely separated geographical areas, similar reasonings in time can be made, but must be done using far more advanced techniques than can be illustrated in this example.

Isolation of a localized fault, and reconfiguring of the network, would be done by electric devices such as relays and circuit breakers. Reconfiguring the network requires an advanced algorithm which must make several considerations regarding grid stability, constraints and load balancing. All these considerations would be too complex for this toy example, thus the only consideration the algorithm must make is how to connect as many consumers as possible to the grid.

The example system is shown in Figure 6.9. Each unit on the grid is supplied with measurement units, the ability to localize faults, disconnect and reconnect to the network. There are supplies from both sides of the line, thus one unit must be disconnected at all times. When a fault is detected, the units must transmit their synchronized measurements to each other, and decide which two are the closest to the fault. The correct units must then break their connection, isolating the fault. The network should then try to reconfigure the network such that the impact of the fault is minimized, such as Unit D closing in Figure 6.9.

A fault in the power grid can have harmful consequences if left over time. It is therefore desirable to respond to the fault as quickly as possible to reduce the impact of the fault. If the distributed system is unable to reach a decision, then all relays must shut down after some time.

From a computer system perspective, it is apparent that this system constitutes a distributed real-time system, both in the toy example and in reality. Only time-synchronized measurements reported within some time are useful to the system. Also, the system is only useful if it is able to respond to the fault within some time.

Embedded Lingua Franca may be able to provide a distributed real-time system that is able to handle the timing requirements of this example. It is thus an example that could demonstrate some deeper aspects of the project implementation.

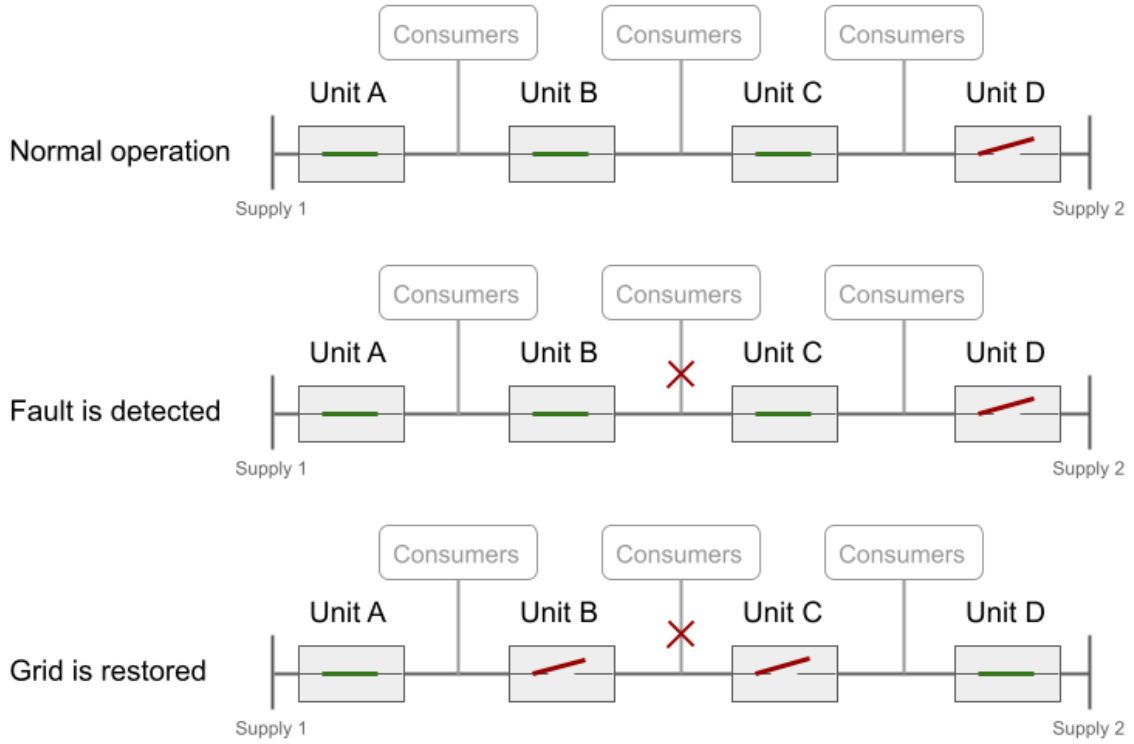


Figure 6.9: The figure shows electrical connections in a simplified self-healing smart grid problem.

.

6.3.2 Case Model in Lingua Franca

In Lingua Franca this example can be modeled with four composite reactors, connected in an all-to-all structure. Within each composite reactor, which corresponds to one unit, there is a voltage measurement reactor (*VoltageSensor*), distributed algorithm reactor for handling inputs (*DistAlg*), and a circuit breaker reactor (*CircuitStatus*). See Figure 6.10 for how a unit's input and output signals, and how it is composed. The voltage measurement reactor has a timer that initiates a voltage measurement. As real voltage data is not available, a voltage time series featuring a voltage drop has been hardcoded in. The time series indicates that the fault is between units B and C. When the voltage drop has been discovered, messages are sent to the other units. A logical action, acting as a security mechanism, is scheduled within the same reactor. In this example, the security mechanism is scheduled after 150 milliseconds. The security mechanism will break the circuit if all messages are not present at the same logical time.

If all replies are received correctly, then the distributed algorithm reactor calculates which two units should close based on the registered voltage measurement of each unit. It then notifies the *CircuitStatus* reactor of the resulting decision for this unit. A logical delay is added to each connection between units by using the *after* keyword. See Figure 6.11 for the connection between the four units.

Units A and D are deployed on a Linux machine, while units B and C are deployed on the two boards. GPIO toggling is used to indicate the time at which the voltage drop is measured, and the time at which it is handled. The precise timing can then be measured by an oscilloscope with four probes. Refer back to Figure 6.1 for the hardware setup.

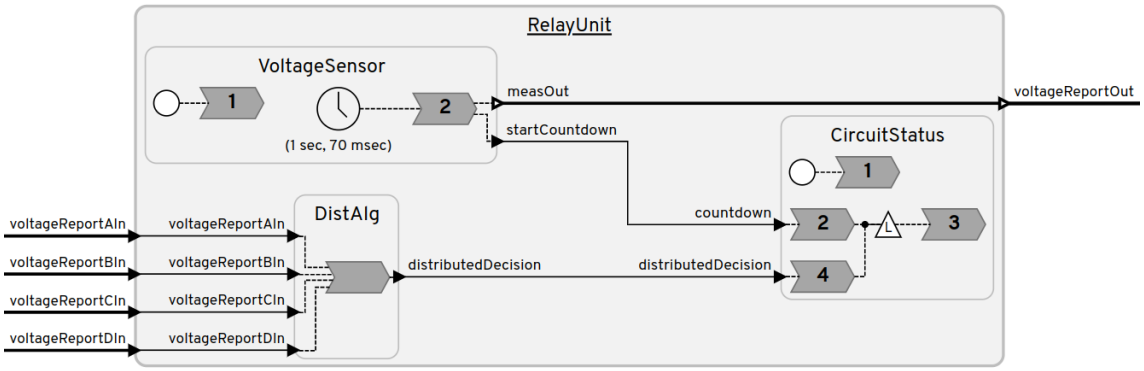


Figure 6.10: The figure shows one of the relay units in the smart grid model in LF.

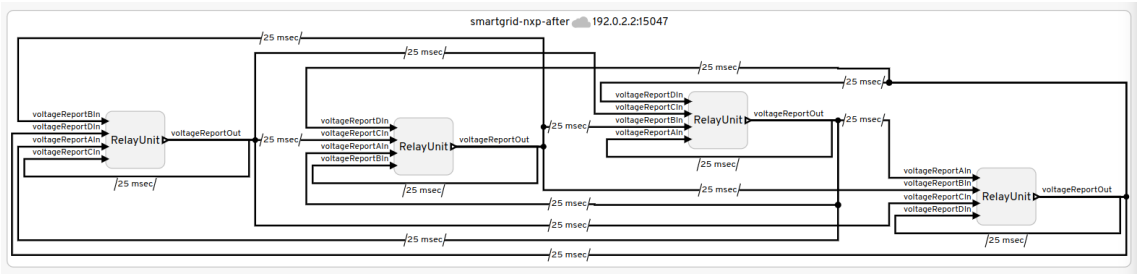


Figure 6.11: The figure shows the units and connections in the smart grid model in LF. The number on each connection is a logical delay on that connection, specified with the keyword *after*.

6.3.3 Test Results

Test 1: Logical Delay covers Total Connection Latency
The result of the test over 10 iterations is shown in Figure 6.2. The maximum reaction time for both units is 26.06 milliseconds, while the average is 25.74 milliseconds, as can be seen at the bottom of the table. The logical delay (*after*) was set to be 25 milliseconds. This time must surpass the total connection latency[1] between the connections, since there are no STP violations registered and the program STP value is set to zero. Since logical time always lags behind physical time, the logical delay results in a physical delay (experienced by the sender) of *at least* 25 milliseconds. Thus, the execution of the distributed algorithm and circuit breaking must take between 0.5 and 1 milliseconds.

Since the logical delay on connections accounts for the total connection latency, the test gives very consistent results. However, the actual arrival times of the messages could be *faster* than 25 milliseconds (see the message delay test results, for example). It could therefore be imagined that the reaction times could have been a lot faster than what was achieved in Table 6.2. Using logical delays therefore adds some unnecessary reaction time delay, but gains low variance in the results. Therefore the line fault is not handled as quickly as strictly possible. In the context of an electricity grid, the fault should be handled as quickly as possible, however, the time of the handling in each unit is important as well. If unit D closes the connection before units B and C have opened them, then the fault could spread to even more consumers. The fault handling should happen before the grid reconfigures. It is therefore better to delay the reaction to ensure that all units react at consistent times, than that they are faster, more unpredictable and happen in an

[1] Refer to Section 2.5.6 about decentralized coordination for the definition of total latency.

arbitrary order.

Smart Grid Reaction Time [ms]					
	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
Unit B	25.67	25.74	25.68	25.90	25.55
Unit C	25.66	25.68	25.79	25.69	25.68
	Iteration 6	*Iteration 7*	*Iteration 8*	*Iteration 9*	*Iteration 10*
Unit B	25.83	25.55	25.84	25.54	25.98
Unit C	25.89	25.70	25.70	26.06	25.56
Maximum Reaction Time, Both units			**Average Reaction Time, Both units**		
26.06			25.74		

Table 6.2: The Table shows the reaction times of units B and C in the Smart Grid example. The connections between units have a logical delay of 25 milliseconds.

Another aspect of using the logical delay functionality is that the unit is free to do other work while the messages are in transit or received, but not to be handled yet. As discussed in Section 2.5.6, using logical delays gives increased availability. In this example, this availability facilitates that the unit can continue to do voltage measurements, break the circuit if the deadline is reached, or (in an extended implementation) react to input from a human grid operator.

Test 2: Logical Delay does not cover Total Connection Latency
The result of the test over 10 iterations is shown in Figure 6.3. The maximum reaction time for both units is 150.25 milliseconds, while the average is 103.78 milliseconds, as can be seen at the bottom of the table. The logical delay (*after*) was set to be 17.5 milliseconds. It is clear that the assumptions are broken in many (but not all) of the cases. This would point to that the total latency on the connections is close to 17.5 milliseconds, but sometimes more. When the assumptions are broken, the inputs are not present in time, and then trigger a security mechanism to break all circuits at 150 milliseconds after the measured fault. This mechanism was explained in Section 6.3.2.

Smart Grid Reaction Time [ms]					
	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
Unit B	18.14	18.72	18.20	148.65	18.18
Unit C	150.25	149.95	150.08	150.01	149.86
	Iteration 6	*Iteration 7*	*Iteration 8*	*Iteration 9*	*Iteration 10*
Unit B	18.21	17.94	17.93	150.10	149.72
Unit C	149.91	149.82	149.89	149.83	150.16
Maximum Reaction Time, Both units			**Average Reaction Time, Both units**		
150.25			103.78		

Table 6.3: The Table shows the reaction times of units B and C in the Smart Grid example. The connections between units have a logical delay of 17.5 milliseconds.

Test 3: Logical Delay and STP offset cover Total Communication Latency

The result of the test over 10 iterations is shown in Figure 6.4. The maximum reaction time for both units is 16.32 milliseconds, while the average is 13.90 milliseconds, as can be seen at the bottom of the table. The logical delay (*after*) was set to be 12.5 milliseconds, while

reactors RelayUnit and DistAlg got STP offsets of 5 milliseconds and 15 milliseconds. From the last test, we know that the total communication latency is around 17.5 milliseconds. Here, we achieve good results with a logical delay of only 12.5 milliseconds, but the STP offset will come in addition. Since we are using the STP offset to compensate for part of the time, we are letting the unnecessary delay introduced by logical delay be minimal.

Smart Grid Reaction Time [ms]					
	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
Unit B	13.25	13.17	12.99	14.75	13.02
Unit C	14.18	16.06	13.92	13.35	14.43
	Iteration 6	*Iteration 7*	*Iteration 8*	*Iteration 9*	*Iteration 10*
Unit B	13.32	14.27	13.06	13.74	13.26
Unit C	13.70	15.34	12.77	13.11	16.32
Maximum Reaction Time, Both units			**Average Reaction Time, Both units**		
16.32			13.90		

Table 6.4: The Table shows the reaction times of units B and C in the Smart Grid example. The connections between units have a logical delay of 12.5 milliseconds and an STP offset of 5 + 15.

Test 4: STP offset covers Total Communication Latency
The last test is setting only STP offsets to account for the total communication latency, for example, setting an STP of 25 milliseconds for the RelayUnit and DistAlg reactors. However, the output indicated in Figure 6.12 shows that several STP violations occur. Since each RelayUnit must have a voltage sensor giving output and computation requiring input, the whole reactor will get an STP offset when setting RelayUnit. Each logical tag will be delayed with the STP offset. This means that sending and receiving are still essentially happening synchronously - but after waiting 25 milliseconds. The problem as formulated above makes it non-viable to only set STP offset. Even if letting the DistAlg STP offset be larger than the RelayUnit, the RelayUnit will still experience STP violations.

```
Federate 1: ERROR: STP violation occurred in a trigger to reaction 1, and there is no handler.
**** Invoking reaction at the wrong tag!
Federate 1: ERROR: STP violation occurred in a trigger to reaction 8, and there is no handler.
**** Invoking reaction at the wrong tag!
Federate 1: ERROR: STP violation occurred in a trigger to reaction 1, and there is no handler.
**** Invoking reaction at the wrong tag!
Federate 1: ERROR: STP violation occurred in a trigger to reaction 4, and there is no handler.
**** Invoking reaction at the wrong tag!
Federate 1: ERROR: STP violation occurred in a trigger to reaction 4, and there is no handler.
**** Invoking reaction at the wrong tag!
```

Figure 6.12: The figure shows the serial output from Putty when STP violations occur when only using STP offset and with no logical delay.

Chapter 7

Discussion

7.1 Thesis Outcomes and Future Work

This thesis has contributed to merge a real-time operating system with a deterministic, time-managing and distributed coordination language. The innovations associated with federated LF are thus available to test on a whole new class of systems; distributed embedded systems and real-time systems. This thesis has provided some initial results and a baseline to improve and investigate further on. A 20-minute presentation was held by the author to the LF team from the University of California, Berkeley, as well as contributors from other universities. The presentation triggered discussions and ideas for future work and testing, as similar tests had not been made previously. The presentation is included in Appendix D.

Zephyr proved to be a promising platform for further work with Lingua Franca. It has some great benefits related to timing precision, hardware independency and library implementations. Many additional LF features are possible to port to Zephyr, making it a good choice for a universal platform for embedded system applications.

Initial clock synchronization was very accurate, with only a few hundred microsecond error. It is possible several optimizations can be made to decrease this error further, for example achieving higher precision and stability in the underlying LF clock for Zephyr. The drift that was measured was also satisfactory for many short-lived applications. While continuous clock synchronization was not a priority to adapt in this project, it should be relatively straightforward as Zephyr offers support for UDP sockets and UDP communication. Continuous clock synchronization could also be implemented using Zephyr's experimental PTP synchronization scheme, if the board has support for it. Which of Zephyr's clock synchronization mechanism and LF's clock synchronization mechanism will be best, is left for future work.

The message delay was measured to be very low in a simple Sender-receiver structure. This delay will increase significantly for more complicated reactor structures, especially for centralized coordination, since many more control messages must be sent. Thus, the simple sender-receiver test was the test where one would expect the two coordination schemes to be the most similar.

Another aspect of the message delay test was the measured variance in the two coordination schemes. For decentralized coordination, the variance was much smaller due to skipping the intermediary step through the RTI. As decreased variance is a very good

property in DRTSes, having the RTI on an embedded platform could prove beneficial when using centralized coordination. Although, the average value would then increase as well, due to an embedded system's decreased computational capacity.

Since this thesis has adapted the RTI to Zephyr RTOS as well, embedded-only federations can now be deployed. A side-effect of this port is that the RTI and federate runtime has become more platform-independent. This is beneficial to any future work involving federated LF on other widely-used platforms, such as Arduino and FreeRTOS.

The message throughput test pushed the implementation to its limits, and revealed some behaviors not previously registered (according to the LF team). No mechanism for limiting a reactor's output on the network has been implemented, thus buffer overflows will crash the program at the receiver end. Buffer overflows were relatively frequent for decentralized coordination as well. The buffer configurations that mitigated these errors were likely effective due to increased resources in sending a message, and having many enough buffers allocated to receive messages for the short amount of time (ie. under ten seconds) the test took place. Thus implementing a message flow control mechanism (ie. a backpressure mechanism) could be a good idea for future work.

The smart grid example demonstrated concrete results from using LF's logical delays and STP offsets to manipulate the program's timing behavior. In the specific example, using logical delays was the preferred method. Using logical delays produced remarkably consistent results when the logical delay exceeded the total communication latency. This makes sense as all messages were received and processed before the logical tag that they should execute. After delays promote availability, since the reactor remains available for other input. For example, if a human operator or other security mechanism were to intervene, then it could receive and process this.

Using only STP offsets on reactors turned out to be non-viable, as each unit had to both produce output and receive inputs. The STP offset reduces availability, since the reactor must physically wait this time before advancing to the next logical time. The consistency is, on the other hand, better, as the tag it received on the sender side will be the same as the tag it has on the receiver side. It is also possible to use both options, thus letting the messages be processed closer to "as fast as possible". Although, this is hardly a concern for DRTS, since it also gives higher variance and unpredictability in the reaction times.

Some last remarks regard considerations beyond this thesis. The whole LF runtime frequently uses dynamic memory allocation which will ultimately pose a problem to embedded applications both in terms of performance, but also introduced non-determinism (ie. memory allocations can fail). For DRTS, this should be taken into account as it is not a reliable strategy. An effort to add an alternative memory allocation scheme for embedded systems should be made in order to increase reliability.

Additionally, the runtime uses TCP and UDP communication, which are not reliable. Both TCP and UDP are best-effort protocols that might fail due to network congestion or other reasons. Therefore, work tied to Time Sensitive Networks might be interesting to tie into the federated LF functionality.

Nevertheless, this approach can be used to design and implement soft and firm real-time systems. Although more work is needed to improve various functionality, the approach is definitely viable and shows great promise. The strengths are largely related to explicit control over timing, as demonstrated in the smart grid example, deterministic coordination, shown in Section 2.5.5, and intuitive modular design. The Zephyr platform provides a

good and flexible interface to handle networking and optimizations for embedded real-time systems.

7.2 Areas of Improvement

The thesis has several areas to improve. The implementation is not thoroughly tested, and cannot be added to the generic testing framework that the LF project utilizes. Therefore it remains an experimental addition to LF. All functional specifications except for UDP socket communication were implemented. All high-priority target declaration functionality is working. The project has contributed to the open-source repositories. On the other hand, the integration of the LF language must be improved. However, some of the design choices are part of a larger direction that the Lingua Franca team will decide.

Some of the results show unexpected behaviors that seem to stem from unwanted elements in the implementation of this project, the federated runtime or the Zephyr threaded runtime port. A challenging aspect of this project is that the software base is not a finished piece of software. It is still under active development and bugs are found continuously. The kinds of measurements done in this thesis have not been attempted before, so new bugs might very well be causing some of the unexpected behaviors.

Another consequence of Lingua Franca having various bugs and unworking functionalities was that a lot of time was spent debugging problems due to largely unrelated things. The biggest reason for this is that the project integrated new developments from the *main* branch on Github, instead of only using a *release* (which is less likely to contain bugs - at least obvious functionality-related bugs). However, the argument to do this is that the project would soon become irrelevant as so many big changes were made to the federated runtime. It also makes integration of the finished project easier.

Zephyr has some challenging aspects to work with. It is not so thoroughly documented, especially the configuration options, and not a lot of online resources or examples are available.

Chapter 8

Conclusion

The Xronos framework, or federated LF, has been extended to be compatible with a real-time operating system called Zephyr. Federations may now be deployed on both multi-platform systems and purely embedded systems. The setup process, building and running applications have been documented as part of the contribution to the Lingua Franca project. Though the implementation is still experimental, it marks a significant step towards using LF for distributed real-time applications.

Initial clock synchronization is showing promising results, with only a few hundred microseconds error and low drift.

Message delay is largely as expected, with decentralized coordination having the least expected delay and less variance. Since messages must go through the RTI in the centralized scheme, message delays must be larger. However, it is a possibility to significantly decrease the variance by running the RTI on a single-purpose board.

The message throughput test yielded a throughput of around 600 messages per second, with centralized coordination showing better performance. The test also revealed that a message flow control mechanism for the federated runtime should be implemented.

The smart grid example demonstrated how federated LF allows direct manipulation of the properties of the system's reaction times. For this example, logical delays were the best option to use, providing reliable and consistent timing results.

Based on all these considerations, federated LF on Zephyr RTOS is definitely viable for soft and firm distributed real-time systems. Federated LF allows for intuitive design, deterministic coordination and explicit handling of time. Future work might increase the viability of DRTS even further, as some weaknesses of the approach are related to networking and memory handling.

Bibliography

[1] Soroush Bateni Marten Lohstroh Christian Menard and Edward A. Lee. 'Toward a Lingua Franca for Deterministic Concurrent Systems'. In: *ACM Transactions on Embedded Computing Systems* 20.4 (2021), 36:1–36:27.

[2] Marten Lohstroh. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. Technical Report UCB/EECS-2020-235. http://www2.eecs.berkeley.edu/Pubs/Tech 2020-235.html: Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2020.

[3] Soroush Bateni et al. *Xronos: Predictable Coordination for Safety-Critical Distributed Embedded Systems*. 2022. DOI: 10.48550/ARXIV.2207.09555. URL: https://arxiv.org/abs/2207.09555.

[4] Yang Zhao, Jie Liu and Edward A. Lee. 'A Programming Model for Time-Synchronized Distributed Real-Time Systems'. In: *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. 2007, pp. 259–268. DOI: 10.1109/RTAS.2007.5.

[5] Jia Zou et al. 'Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems'. In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2009, pp. 77–86. DOI: 10.1109/RTAS.2009.39.

[6] J.S. Dahmann. 'High Level Architecture for simulation'. In: *Proceedings First International Workshop on Distributed Interactive Simulation and Real Time Applications*. 1997, pp. 9–14. DOI: 10.1109/IDSRTA.1997.568652.

[7] Silje Susort. *Embedded Lingua Franca and FreeRTOS*. TTK4550 Specialization Project Report. 2022.

[8] Erling Rennemo Jellum et al. 'Beyond the Threaded Programming Model on Real-Time Operating Systems'. In: *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023)*. Ed. by Federico Terraneo and Daniele Cattaneo. Vol. 108. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 3:1–3:13. ISBN: 978-3-95977-268-6. DOI: 10.4230/OASIcs.NG-RES.2023.3. URL: https://drops.dagstuhl.de/opus/volltexte/2023/17734.

[9] Morgan Quigley et al. 'ROS: an open-source Robot Operating System'. In: vol. 3. Jan. 2009.

[10] ROS and Open Robotics contributors. *ROS Documentation*. URL: http://wiki.ros.org/ (visited on 1st June 2023).

[11] *Micro-ROS: micro-ROS puts ROS 2 onto microcontrollers*. URL: https://micro.ros.org/ (visited on 1st June 2023).

[12] A Stanford-Clark and U Hunkeler. 'Mq telemetry transport (mqtt)'. In: *Online]. http://mqtt.org. Accessed June 2023* 22 (1999), p. 2013.

[13] HiveMQ. *MQTT MQTT 5 Essentials*. Landshut, Germany: HiveMQ, 2020. ISBN: 978-3-00-067913-1. Ebook.

[14] Members and individual contributors from The Zephyr Project. *MQTT*. URL: https://docs.zephyrproject.org/latest/connectivity/networking/api/mqtt.html (visited on 1st June 2023).

[15] Inc. Amazon Web Services and its affiliates. *coreMQTT - MQTT C client library for small IoT devices*. URL: https://www.freertos.org/mqtt/index.html (visited on 1st June 2023).

[16] Saranya Natarajan and David Broman. 'Timed C : An Extension to the C Programming Language for Real-Time Systems'. In: *24TH IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM (RTAS 2018) :* IEEE Real-Time and Embedded Technology and Applications Symposium. QC 20180920. 2018, pp. 227–239. ISBN: 978-1-5386-5295-4. DOI: 10.1109/RTAS.2018.00031.

[17] Henrik Austad et al. 'Composable distributed real-time systems with deterministic network channels'. In: *Journal of Systems Architecture* 137 (2023), p. 102853. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2023.102853. URL: https://www.sciencedirect.com/science/article/pii/S1383762123000322.

[18] Lucia Lo Bello and Wilfried Steiner. 'A Perspective on IEEE Time-Sensitive Networking for Industrial Communication and Automation Systems'. In: *Proceedings of the IEEE* 107.6 (2019), pp. 1094–1120. DOI: 10.1109/JPROC.2019.2905334.

[19] K. Erciyes. *Distributed Real-Time Systems: Theory and Practice*. Springer, 2019.

[20] Erling R. Jellum et al. 'InterPRET: A Time-Predictable Multicore Processor'. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. CPS-IoT Week '23. San Antonio, TX, USA: Association for Computing Machinery, 2023, pp. 331–336. ISBN: 9798400700491. DOI: 10.1145/3576914.3587497. URL: https://doi.org/10.1145/3576914.3587497.

[21] Richard Barry. *Mastering the freeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. Real Time Engineers Ltd., 2016.

[22] William Stallings. *Operating Systems: Internals and Design Principles, 9th edition, global edition*. Pearson Education Limited, 2018.

[23] Members and individual contributors from The Zephyr Project. *Zephyr Main Website*. URL: https://www.zephyrproject.org/ (visited on 10th Nov. 2022).

[24] Members and individual contributors from The Zephyr Project. *Zephyr supported boards*. URL: https://docs.zephyrproject.org/3.2.0/boards/index.html (visited on 10th Nov. 2022).

[25] Members and individual contributors from The Zephyr Project. *Zephyr Kernel Documentation Page*. URL: https://docs.zephyrproject.org/latest/kernel/services/index.html#kernel-api (visited on 15th Sept. 2022).

[26] Members and individual contributors from The Zephyr Project. *Zephyr Github Page*. URL: https://github.com/zephyrproject-rtos/zephyr (visited on 15th Sept. 2022).

[27] Members and individual contributors from The Zephyr Project. *Zephyr Install and Getting Started Documentation Page*. URL: https://docs.zephyrproject.org/latest/develop/getting_started/index.html (visited on 15th Sept. 2022).

[28] Members and individual contributors from The Zephyr Project. *Zephyr Configuration Documentation Page*. URL: https://docs.zephyrproject.org/latest/build/kconfig/index.html#kconfig (visited on 10th Nov. 2022).

[29] Members and individual contributors from The Zephyr Project. *Zephyr Network Stack*. URL: https://docs.zephyrproject.org/3.0.0/guides/networking/net-stack-architecture.html (visited on 10th Nov. 2022).

[30] Behrouz A. Forouzan. *TCP/IP Protocol Suite, Fourth Edition*. McGraw-Hill, 2010.

[31] Leslie Lamport. 'Time, Clocks, and the Ordering of Events in a Distributed System'. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[32] Reinhard Schwarz and Friedemann Mattern. 'Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail'. In: *Distrib. Comput.* 7.3 (Mar. 1994), pp. 149–174. ISSN: 0178-2770. DOI: 10.1007/BF02277859. URL: https://doi.org/10.1007/BF02277859.

[33] Eric A. Brewer. *Towards Robust Distributed Systems*. Keynote from Symposium on Principles of Distributed Computing (PODC). 2000.

[34] Edward A. Lee et al. 'Quantifying and Generalizing the CAP Theorem'. In: *CoRR* abs/2109.07771 (2021).

[35] Teodor Neagoe, Valentin Cristea and Logica Banica. 'NTP versus PTP in Com puter Networks Clock Synchronization'. In: *2006 IEEE International Symposium on Industrial Electronics*. Vol. 1. 2006, pp. 317–362. DOI: 10.1109/ISIE.2006.295613.

[36] James C. Corbett et al. 'Spanner: Google's Globally-Distributed Database'. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 261–264. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett.

[37] The Lingua Franca Team. *Lingua Franca Website*. URL: https://www.lf-lang.org/ (visited on 12th Dec. 2022).

[38] Wireshark Foundation. *Wireshark - The world's most popular network protocol analyzer*. URL: https://www.wireshark.org/ (visited on 2nd June 2023).

[39] PuTTY team. *PuTTY User Manual*. URL: https://the.earth.li/~sgtatham/putty/0.78/htmldoc/ (visited on 2nd June 2023).

[40] Richard Stallman, Roland Pesch and et al. Stan Shebs. *Debugging with GDB, tenth edition*. Free Software Foundation, 2018.

[41] Rocco Marco Guglielmi. *Debugging on STM32 with Chibistudio: THe Ultimate Guide*. URL: https://www.playembedded.org/blog/debugging-stm32-chibistudio/ (visited on 2nd Aug. 2019).

[42] PyOCD Authors. *pyOCD*. URL: https://pyocd.io/ (visited on 2nd June 2023).

[43] Christian Andre Andresen et al. 'Fault Detection and Prediction in Smart Grids'. In: *2018 IEEE 9th International Workshop on Applied Measurements for Power Systems (AMPS)*. 2018, pp. 1–6. DOI: 10.1109/AMPS.2018.8494849.

Appendix A

Joint RTI main function

The following code must replace the auto-generated main function to try the joint RTI functionality.

```
#include "../core/federated/RTI/rti_lib.h"
#include "core/platform.h"

void main(void) {
    lf_thread_t thread_id;

    #ifdef LF_CLOCK_SYNC_INITIAL
    rti_set_args(NUMBER_OF_FEDERATES, "Unidentified Federation", 15047, clock_sync_init,
                        LF_CLOCK_SYNC_PERIOD_NS, LF_CLOCK_SYNC_EXCHANGES_PER_INTERVAL);

    #elif LF_CLOCK_SYNC_ON
    rti_set_args(NUMBER_OF_FEDERATES, "Unidentified Federation", 15047, clock_sync_on,
                        LF_CLOCK_SYNC_PERIOD_NS, LF_CLOCK_SYNC_EXCHANGES_PER_INTERVAL);

    #else LF_CLOCK_SYNC_OFF
    rti_set_args(NUMBER_OF_FEDERATES, "Unidentified Federation", 15047, clock_sync_off, 0, 0);

    #endif

    lf_initialize_clock();

    lf_thread_create(&thread_id, lf_rti_main, NULL);

    int res = lf_reactor_c_main();
    exit(res);
}
```

Appendix B

List of Acronyms

List of Acronyms			
Acronym	**Meaning**	**Acronym**	**Meaning**
LF	Lingua Franca	NTP	Network Time Protocol
RTOS	Real-Time Operating System	PTP	Precision Time Protocol
DRTS	Distributed Real-Time System	CAP	Consistency, Availability, network Partitioning
RTI	RunTime Infrastructure	CAL	Consistency, Availability, network Latency
TCP	Transmission Control Protocol	NMR	Next Message Request
UDP	User Datagram Protocol	NET	Next Event Tag
IP	Internet Protocol	LTC	Logical Tag Complete
GDB	GNU Debugging	TAG	Tag Advance Grant
PTIDES	Programming Temporally Integrated Distributed Embedded Systems	EIMT	Earliest Incoming Message Tag
HLA	High Level Architecture	PTAG	Provisional TAG
ROS	Robot Oberating System	STP	Safe-To-Process
POSIX	Portable Operating System Interface	OS	Operating System
MQTT	MQ Telemetry Transport	ID	Identity
TSN	Time-Sensitive Networking	SCTP	Stream Control Transmission Protocol
BSD	Berkeley Software Distribution	MAC	Media Access Control
WCET	Worst-Case Execution Time	USB	Universal Serial Bus
CPU	Central Processing Unit	JTAG	Joint Test Action Group
API	Application Programming Interface	SWD	Serial Wire Debug
SDK	Software Development Kit	MCU	Microcontroller Unit
LED	Light Emitting Diode	EVK	Evaluation Kit

List of Acronyms			
Acronym	**Meaning**	**Acronym**	**Meaning**
ISR	Interrupt Service Routine	IDE	integrated development environment
FIFO	First-In First-Out	GPIO	General Purpose Input-Ouput
LIFO	Last-In First-Out	EPSR	Execution Program Status Register
CoAP	Constrained Application Protocol	LAN	Local Area Network
PMU	Phasor Measurement Unit	ML	Machine Learning

Appendix C

Hyperlink Reference

Hyperlink Reference	
Reference	**URL**
Kconfig Search	https://docs.zephyrproject.org/3.1.0/kconfig.html
Reactor-c Fork	https://github.com/siljesu/reactor-c/tree/zephyr-federated-support
Lingua Franca Fork	https://github.com/siljesu/lingua-franca/tree/zephyr-fed-support-update
lf-west-template Fork	https://github.com/siljesu/lf-west-template/tree/zephyr-federated
CMSIS-DAP firmware rules	https://github.com/pyocd/pyOCD/blob/main/udev/50-cmsis-dap.rules
Zephyr: Getting Started Guide	https://docs.zephyrproject.org/latest/develop/getting_started/index.html
lf-west-template README	https://github.com/siljesu/lf-west-template/tree/main#readme
Lingua Franca Handbook	https://www.lf-lang.org/docs/handbook/developer-setup?target=c

Appendix D

Presentation to LF team

Federated LF on ZephyrRTOS

Distributed Real-Time Systems

Federation Topologies

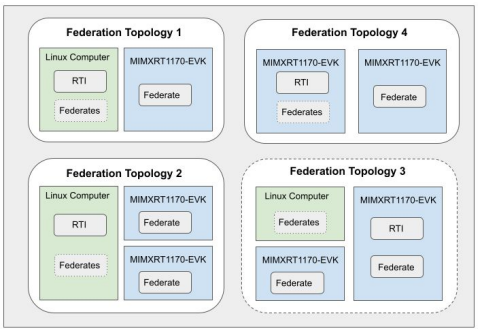
Multi-platform federations

Embedded-only federations

Centralized or decentralized coordination

Clock synchronization

Joint RTI



Benefits of using Zephyr

Partial POSIX implementation - makes porting a lot easier

Higher possible timing precision than, ex FreeRTOS

Hardware independency through configuration systems like DeviceTree and Kconfig (The code itself is unchanged using different boards)

A good choice for most embedded applications for LF

How to set it up

Mostly done through application configuration files

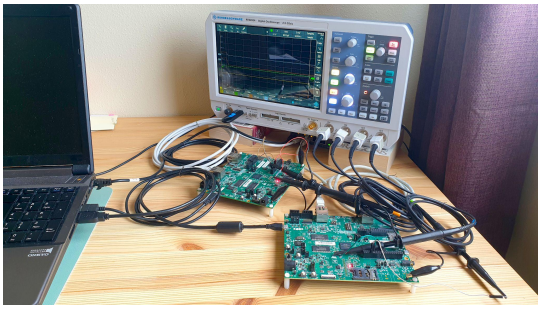
The process is documented in lf-west-app/zephyr-fed-support

Experimental setup

Ethernet cables connect the nodes

An oscilloscope is used to measure timing of GPIO toggles at very high precision

Reactions toggle GPIOs

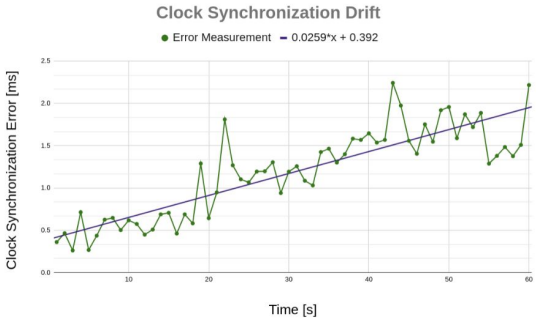


Experimental results - Initial Clock Synchronization

Sub-millisecond error initially

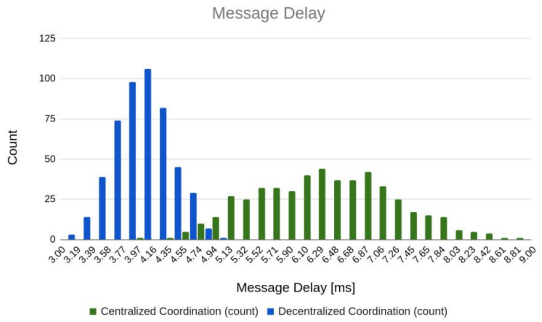
After 1 minute, under 2 ms error

Continuous clock sync doesn't work yet



Experimental Results - Message Delay

Simple Source-Drain structure



Experimental Results - Message Throughput

Send messages as fast as possible

Centralized coordination has a higher throughput rate

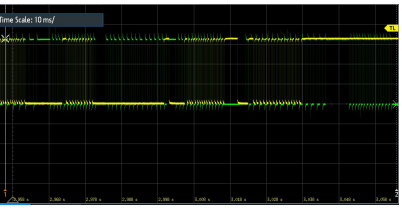
Decentralized has less variance

Message Throughput rates [messages/second]					
	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
Centralized	604.76	619.56	614.00	616.52	599.82
Decentralized	583.43	592.86	585.22	589.41	601.12
	Iteration 6	*Iteration 7*	*Iteration 8*	*Iteration 9*	*Iteration 10*
Centralized	587.96	581.17	674.53	628.42	589.79
Decentralized	583.03	588.81	590.12	588.02	586.82
	Average		**Standard Deviation**		
Centralized	611.65		26.84		
Decentralized	588.88		5.27		

Some strange behaviors

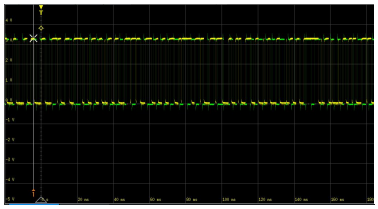
Centralized

Network buffers



Decentralized

More even sending and receiving



A more advanced example

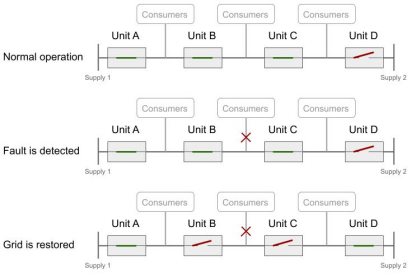
Each unit need to be able to

- measure regularly

- Report measurements to the other units

- Receive measurements from other units to calculate action

Reaction time is measured



Using logical delays on connections of 25 milliseconds

Smart Grid Reaction Time [ms]					
	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
Unit B	25.67	25.74	25.68	25.90	25.55
Unit C	25.66	25.68	25.79	25.69	25.68
	Iteration 6	*Iteration 7*	*Iteration 8*	*Iteration 9*	*Iteration 10*
Unit B	25.83	25.55	25.84	25.54	25.98
Unit C	25.89	25.70	25.70	26.06	25.56
Maximum Reaction Time, Both units			**Average Reaction Time, Both units**		
26.06			25.74		

Using Logical delays and STP offsets

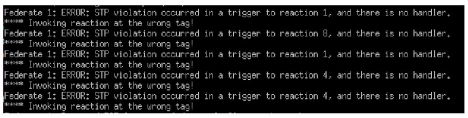
Smart Grid Reaction Time [ms]					
	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
Unit B	13.25	13.17	12.99	14.75	13.02
Unit C	14.18	16.06	13.92	13.35	14.43
	Iteration 6	*Iteration 7*	*Iteration 8*	*Iteration 9*	*Iteration 10*
Unit B	13.32	14.27	13.06	13.74	13.26
Unit C	13.70	15.34	12.77	13.11	16.32
Maximum Reaction Time, Both units			**Average Reaction Time, Both units**		
16.32			13.90		

Using only STP offset

Each unit is a reactor composed of both input and output functionality

Using only STP offsets delay both input and output equally

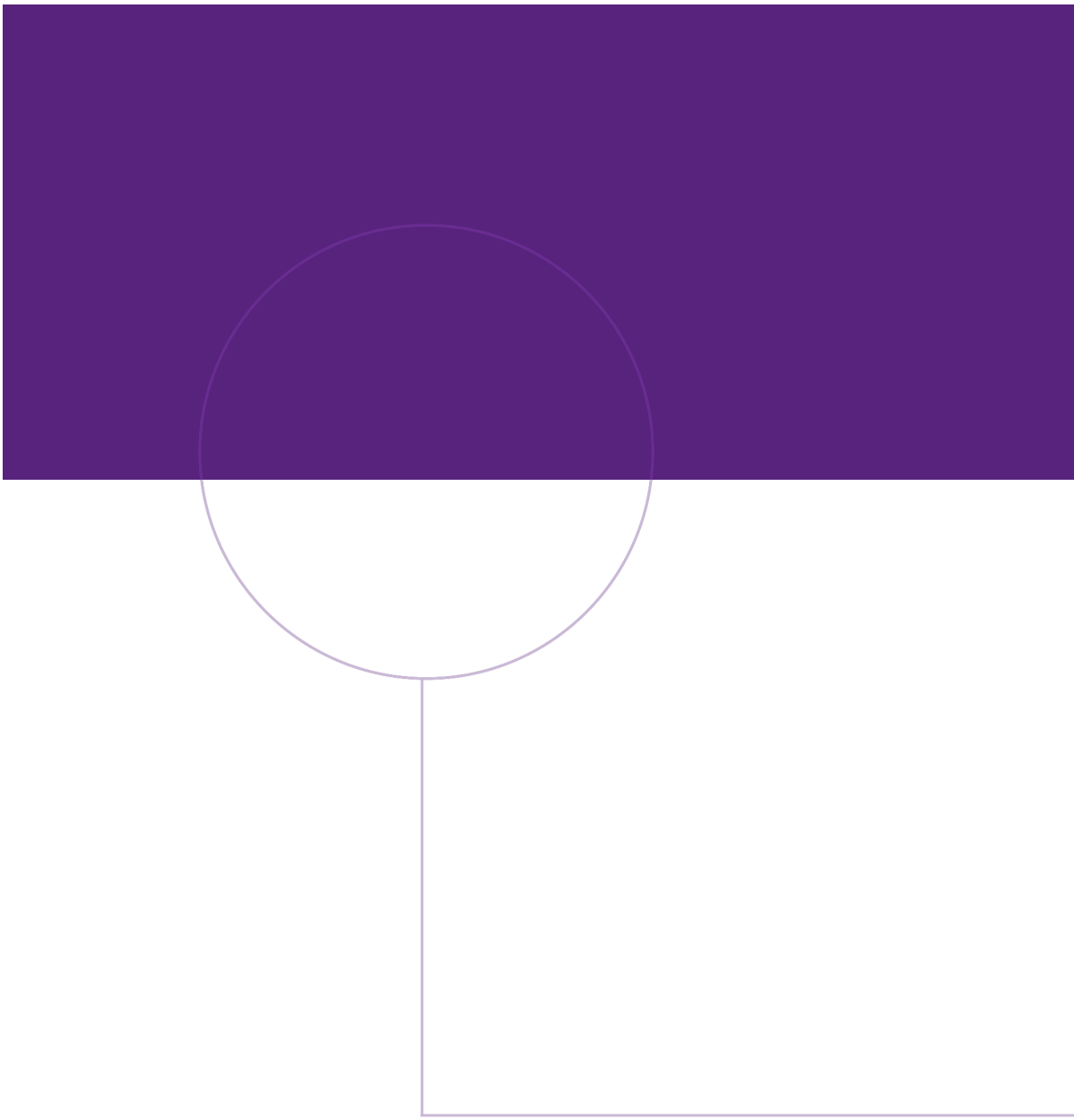
It is not a viable solution in this example



Questions and Contact

Feel free to send me an email at:

siljsus@stud.ntnu.no



NTNU

Norwegian University of
Science and Technology