PHYSICALLY-ASYNCHRONOUS LOGICALLY-SYNCHRONOUS (PALS) SYSTEM
DESIGN AND DEVELOPMENT

BY
ABDULLAH AL-NAYEEM

DISSERTATION
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:
Professor Lui Sha, Chair and Director of Research
Associate Professor Marco Caccamo
Assistant Professor Sayan Mitra
Dr. Darren D. Cofer, Principal Systems Engineer, Rockwell Collins Inc.
ABSTRACT

Cyber-physical systems, such as avionics and automobiles, are real-time distributed systems, where many of the information processing functions require consistent views and actions across distributed computing nodes. Guaranteeing consistency in these distributed computations is challenging. In particular, distributed systems are physically asynchronous because system clocks at each node cannot be perfectly synchronized. Such physical asynchrony, if not properly dealt with, can lead to distributed race conditions and subsequently result in inconsistent actions and anomalous system behaviors.

In this thesis, we address this problem and introduce a novel design methodology that guarantees consistency in real-time distributed computations. At the core of this approach is a complexity-reducing architectural pattern, called the Physically-Asynchronous Logically-Synchronous (PALS) system. The PALS system is a formal architectural pattern that engineers can use to develop distributed applications as if they would operate on a globally synchronous architecture with a single global clock. The pattern maps the globally synchronous design as a logically synchronous design executing on the physically asynchronous architecture. It provides significant benefit in terms of the verification of safety and correctness. The formal verification cost is greatly reduced since engineers only verify the simple globally synchronous model.

The thesis makes several contributions to the design and development of the PALS system:

C1 - Architectural model definitions: We propose architectural model definitions of the globally synchronous design and its equivalent logically synchronous design using SAE Architecture Analysis and Design Language (AADL), an industry-standard modeling language.

C2 - Formal pattern specification and analysis: One of the biggest challenges in model-based engineering is to preserve the verification properties as engineers refine and extend the models during the development process. We therefore give a formal specification of this pattern and perform static analysis to detect any error during the system design.
C3 - Multi-rate PALS system: We extend the PALS system to support multi-rate distributed computations. We provide an architectural analysis to support composition of multiple instances of this pattern in a given system model.

C4 - Middleware design for PALS system: We have developed a middleware to implement the PALS applications in C++. The middleware addresses several implementation challenges, e.g. node failure, integration with underlying infrastructure components.
To my family members for their love and support.
ACKNOWLEDGMENTS

This work is part of a research collaboration of a large team from both industry and academia. I acknowledge the contributions of the team members. I am particularly grateful to my advisor Dr. Lui Sha not only for providing me the right guidance throughout this Ph.D. program, but also for his advice for my future career and life. His research philosophy on “complexity reduction in a large-scale system design” will surely be something that I want to follow in my future endeavor. I am also grateful to Dr. Steven Miller and Dr. Darren Cofer. Their advice and suggestions have been instrumental toward formulating the solutions. I would like to thank Dr. Cheolgi Kim for his work on the PALSware middleware. For any graduate student, the first paper is always challenging. I am grateful to Mu Sun for his help during my first paper published in RTSS 2009. I thank Dr. José Meseguer, Dr. Peter Ólveczky, and Kyungmin Bae for the early discussions on the PALS system and the joint work on the Synchronous AADL specification. I also thank my colleagues, Dr. Min Young Nam, Po-Liang Wu, and Dr. Woochul Kang for their helps and suggestions throughout the Ph.D. program.

I would also like to thank Dr. Marco Caccamo, Dr. Sayan Mitra, and Dr. Darren Cofer for being in my thesis committee and for their helpful suggestions and advices. I thank the AADL community for their ongoing efforts on the development of AADL. I thank our department officials for their help during the entire Ph.D. program at UIUC. I thank the funding agencies for supporting my research.

I am truly grateful to the local Bangladeshi community for providing a friendly, social environment in a foreign land. I will always cherish the memories of my stay in Champaign-Urbana. I am thankful to my childhood friends, BUET friends, and other friends for being with me during past 2 decades. I dedicate this thesis to my wife Nusrah and my family members. It is their love, support and inspiration that keep me cheerful and motivated. Finally, all praises belong to Allah, the most gracious, the most merciful. Without His blessings, nothing would have been possible.
TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION .................................................. 1
1.1 Problem Statement .................................................. 2
1.2 Contributions of This Thesis ........................................ 4
1.3 References ............................................................ 6

CHAPTER 2 PALS SYSTEM OVERVIEW ..................................... 8
2.1 PALS System Model .................................................. 8
2.2 PALS System Rules .................................................. 10
2.3 Overview of the PALS System Design ......................... 14

CHAPTER 3 SYNCHRONOUS AADL SPECIFICATION ..................... 16
3.1 Synchronous Model .................................................. 17
3.2 AADL Model Definition ............................................. 18
3.3 Active-Standby System ............................................. 21
3.4 Static Analysis ....................................................... 25

CHAPTER 4 PALS PATTERN SPECIFICATION AND ANALYSIS ............. 28
4.1 AADL Model Definition ............................................. 29
4.2 Active-Standby System ............................................. 31
4.3 PALS Transformation and Analysis ............................... 35
4.4 Auto-Generation of Synchronous Model from PALS Model .... 40
4.5 PALS Middleware Specification .................................... 41

CHAPTER 5 MULTI-RATE PALS SYSTEM ................................ 47
5.1 Main Concepts ........................................................ 47
5.2 Pattern Assumptions and Guarantees ............................. 52
5.3 Pattern Specification in AADL ...................................... 57
5.4 Compositional Analysis ............................................. 62

CHAPTER 6 MIDDLEWARE DESIGN FOR PALS SYSTEM ............... 66
6.1 Middleware Architecture ............................................ 67
6.2 PALS Tasks and Communications in PALSware ............... 68
6.3 PALS Fault-Tolerant Communication Protocol .................. 73
6.4 Atomicity of Logically Synchronous Computations .......... 81
6.5 Fault Managers in PALSware ...................................... 85
CHAPTER 1
INTRODUCTION

Next generation cyber-physical systems, such as avionics and automobiles, face formidable challenges in managing the software cost and complexity. With the advancement of hardware technologies, many of these systems are now capable of executing complex, software-based functions such as unmanned and autonomous operations, integrating a variety of sensors and monitoring components, adaptive operations in unknown environments. The size of the on-board software of these systems has grown exponentially in recent years [1, 2]. The cost of software development has also increased significantly in these systems. For example, recent studies show that the aircraft software has doubled every four years since the mid-1990 and is estimated to have 27M SLOC by 2014 at a possible cost of over $10B [3]. This overwhelming cost has become a serious concern for the growth of these systems.

In these systems, the cost is primarily involved in guaranteeing the system safety and satisfying the rigorous requirements of the certification authorities. For example, the avionics certification standard DO-178B [4] requires that testing of Level A software provide complete coverage of all logical conditions and decisions in the code. Thus, the whole certification process requires more effort and money to keep up with the growing complexity of these systems. In addition, the recent avionics certification standard DO-178C (with its formal methods supplement DO-333) [5] provides guidance using formal methods to satisfy certification activities. Unfortunately, many of the software components suffer from the state-explosion problem. Their formal verification is extremely difficult, if not impossible. Furthermore, in the past, the research and engineering communities have developed numerous techniques for improving the system safety with fault-avoidance, runtime monitoring, replication, and design diversity [6, 7, 8, 9]. These techniques are often reused in new application domains. However, the lack of systematic reuse of the design leads to repeated verification when applied in a new configuration or application environment.

We need a new breed of software engineering methodologies and theories of complexity control
to afford the advanced capabilities and maintain progress in these technologies. In a recent collaboration with Rockwell Collins Inc. [10, 11], we addressed this problem and proposed a model-based design methodology where complexity-reducing formal architectural patterns play a significant role. This methodology extends the concept of classical software patterns [12]. It reduces the cost of both development and verification of cyber-physical systems by systematically reusing formally verified architectural patterns and components. In this approach, the system architecture is iteratively composed of pre-verified patterns and formally specified components. As long as the pattern assumptions are satisfied, formal analysis of the system properties can reuse the pattern guarantees without re-verifying them. This thesis contributes to this broader design objective of composing cyber-physical systems with architectural patterns. In this thesis, we develop a new formal architectural pattern that can significantly reduce the design and verification complexities of real-time distributed applications.

1.1 Problem Statement

Cyber-physical systems are commonly implemented as networked real-time systems consisting of a network of distributed devices and their controllers. In these systems, many of these computations are periodic in nature and are triggered by periodic timers based on the local system clock. These clocks are not perfectly synchronized. Their relative skew can be bounded but cannot be entirely eliminated.

Distributed computations in these systems are similar to the *Globally-Asynchronous Locally-Synchronous (GALS)* system, originally proposed in the hardware community [13, 14]. Embedded system designers model these computations using the GALS design philosophy. In this design, the computations in a node execute synchronously by the local clock, but execute asynchronously with respect to the computations of other nodes.

1.1.1 Design and Verification Complexities of the GALS System

In this thesis, we propose an alternative model of computation to address the design and verification complexities of the real-time distributed systems. *Designing distributed protocols to guarantee coordination and consistency in a GALS system is extremely difficult.* The main source of difficulties is the asynchronous interactions between different nodes. Because of the clock skews, even a subtle
timing difference in the execution and communication delay can lead to distributed race conditions. These race conditions may lead to many serious bugs that are often very difficult to reproduce. In these cases, a system may operate correctly for a long time, even for years, and suddenly fail after a change in some logically unrelated hardware, software, or workloads.

To illustrate this problem, consider an example of a triplicated redundant control system that receives a new reference position or setpoint command from a supervisory controller. Because of the non-zero clock skews, one controller (say, Controller A) could be in period \( j + 1 \), while the other two (Controller B and C) still be in period \( j \) at the time of receiving the setpoint. Since they receive the setpoint at different periods, Controller A’s control command might diverge from other two control commands and thus would be voted out. As a result, it leads to an invalid failure detection of Controller A (even though it is not actually faulty) and results in an undesirable system configuration where the system is no longer capable of handling a valid fault such as a controller producing incorrect data.

There are other related technical challenges in the GALS system. First of all, tracking down the root causes of these race conditions is nontrivial, even with the help of formal analysis tools, such as model checking. To verify distributed protocols, a model checker has to explore all application states under all possible event interleaving of messages and node executions. This easily leads to the state-space explosion problem in a GALS system. For example, during the early stage of this research, Miller et al. demonstrated this problem in case of an active-standby design of a dual-redundant avionics application [15]. Even in this simple example, model checking of the asynchronous model took over 35 hours in NuSMV [16] to discover a counter example.

Secondly, the problem becomes more challenging when an operation requires distributed computations that interact at different rates. For example, in a fly-by-wire aircraft, the control surfaces are each locally controlled by higher-level supervisory controllers operating at different rates. These controllers are deployed redundantly for fault-tolerance. Consistent views and actions are mandatory when a component interacts not only with its redundant components for replica management, but also with other components of the hierarchical system to exchange discrete commands, such as setpoint, mode change command. The race conditions within these multi-rate distributed computations significantly complicate the problem.

The complexity of the GALS system can also be contrasted with the simplicity of the globally synchronous system. A globally synchronous system has a single global clock that drives the
executions of instructions in each node in lockstep. As a result, a globally synchronous system is conceptually easy to grasp. Designers are less likely to make errors in this system because of the single global clock. Hence, system engineers often start the design phase with a globally synchronous model. However, a perfect clock synchronization is not realistic for networked systems. The design must be modified to make it work in the GALS system. Such modifications are usually non-trivial and may introduce more errors when performed in an ad-hoc manner [17]. These changes also limit the reusability of the design and verification without any formal specification and analysis. For example, if the network topology or the processor mapping changes, one must be able to prove that the behavioral characteristics can still be preserved. Furthermore, these solutions also add complexity by requiring additional handshaking protocols to guarantee consistency [18, 19, 20].

1.2 Contributions of This Thesis

This thesis provides the design and development techniques to address these problems. These techniques are based on a complexity-reducing architectural pattern for logical synchronization, called the Physically-Asynchronous Logically-Synchronous (PALS) system [21, 22, 15]. This pattern is applicable to hard real-time systems that guarantee message delivery in bounded time and bounded clock skews. In this pattern, the architecture is physically asynchronous since computations at different nodes are still driven by asynchronous local clocks; but logically synchronous since the computations have the same logical behavior as those in a globally synchronous architecture.

This pattern facilitates a new system engineering approach for the design and verification of real-time distributed systems. In this approach, engineers design and verify distributed applications as though the computations would execute on a globally synchronous architecture. Later, this synchronous design can be systematically distributed over a physically asynchronous architecture without any modification of application logic and properties. Thus, it gives a feasible verification for large scale systems since the state space of the globally synchronous design is orders of magnitude smaller than that of the GALS system.

In this thesis, we provide a comprehensive view of the recent works on the PALS system design and development. In our earlier work [23], we have presented a model-based implementation of the globally synchronous design as a logically synchronous design in the physically asynchronous architecture. We have also developed an early prototype of a middleware for this pattern. The basic features of this prototype are described in [15]. This thesis extends these concepts of model-
based engineering and middleware development. In the following, we summarize these extensions and overall contributions:

**C1 - Architectural model definitions:** Architectural models are more commonly used to manage the complexity of cyber-physical systems. In this thesis, we provide architectural model definitions to incorporate the PALS pattern in model-based engineering. We use SAE Architecture Analysis and Design Language (AADL) [24] for the pattern modeling. AADL is an architecture description language to model embedded applications and their execution platforms. We have chosen AADL because of its support for architectural analysis and growing popularity in the cyber-physical system community. We propose an architectural model definition of the globally synchronous design in AADL (Section 3.2). Our colleagues, Kyungmin Bae, Peter Ölveczky, and José Meseguer provide formal semantics of this model to verify the globally synchronous design in Real-Time Maude [25]. We also provide architectural model definitions to model the logically synchronous design in the physically asynchronous architecture (Section 4.1, 4.5).

**C2 - Formal pattern specification and analysis:** We give a formal specification of this pattern. This pattern gives a generic architecture for a correctness-preserving mapping of the globally synchronous AADL design to the logically synchronous AADL design (Section 4.3). One of the biggest challenges in model-based engineering is to preserve the verification properties as engineers refine and extend the models during the development process. We therefore support an analysis framework to validate the correctness of the implementation (Section 4.3). Based on the specification, we also support a reverse transformation from the PALS model to the globally synchronous model (Section 4.4).

**C3 - Multi-rate PALS system:** The original PALS pattern works for distributed tasks executing at the same rate. While this is useful for some common fault-tolerant applications (executing at the same rate), the pattern needs to be extended to support multi-rate computations. This thesis gives an extension, called *multi-rate PALS system*, to support multi-rate computations (Section 5.2-5.3). For example, we can apply this pattern in a hierarchical control system to coordinate the hierarchical computations executing at different rates. We

---

1In Appendix A, we give a brief introduction of AADL. Readers, who are not familiar with AADL, are encouraged to read the chapter in the appendix first before reading Chapter 3-5.
also provide an architectural analysis to support the composition of multiple instances of this pattern (Section 5.4).

**C4 - Middleware design for PALS system:** With our colleague Dr. Cheolgi Kim, we have developed a new middleware, called *PALSware*, to support a robust implementation of the PALS system. We present a layered design for this middleware that is both reusable in different system architectures and extendable with architecture-specific solutions for fault management (Section 6.1). The middleware guarantees consistency in distributed applications by eliminating the asynchronous interactions resulting from the node failure (Section 6.3). It also guarantees atomicity in logically synchronous interactions (Section 6.4). In this thesis, we demonstrate the middleware in an academic control testbed and show the consistency in a fault injection framework designed for this middleware (Chapter 7).

### 1.3 References

This dissertation is partially based on the materials previously published in peer-reviewed conference papers and other technical reports [21, 23, 25, 26, 27, 15]. Note that materials previously appeared in [23, 26, 15] are copyright of the Institute of Electrical and Electronics Engineers (IEEE) and the material previously appeared in [25] is copyright of the Springer-Verlag Berlin, Heidelberg. They are reprinted with permission.


2011 Springer Berlin / Heidelberg.

CHAPTER 2
PALS SYSTEM OVERVIEW

In this chapter, we give an overview of the PALS system. The objective is to provide an intuition of how the PALS system can have logically equivalent behavior as the globally synchronous model. In Section 2.1, we discuss the system characteristics that must be guaranteed prior to applying this pattern. In Section 2.2, we discuss the main rules of this pattern. These rules are first described by Sha et al. [21]. Meseguer and Ölveczky give the formal proof of these rules in [22]. At the end of this chapter, in Section 2.3, we give an overview of the model-based design of the PALS system.

2.1 PALS System Model

The PALS system is applicable to hard real-time systems that guarantee real-time, reliable message transmission and bounded clock skews. It envisions a logically synchronous distributed real-time architecture, in which a group of periodic distributed tasks \( (M_1, M_2, \ldots, M_N) \) collaborate for fault-tolerant monitoring and control of physical systems and processes. In this pattern, these tasks execute periodically with the same period \( T \).\(^1\) The pattern is applied to guarantee consistent views and actions during the coordination of these computations.

The pattern assumes following system parameters and their bounds:

1. Each node of the distributed architecture has a monotonically non-decreasing local clock, \( c : Time \rightarrow Time \ (Time = \mathbb{R}_{\geq 0}) \). Here, \( c(t) = x \) is the local clock time \( x \) at the “ideal” global time \( t \). Furthermore, we define \( t_c : Time \rightarrow Time \) such that \( t_c(x) \) denotes the earliest global time at which a local clock shows \( x \), i.e. \( t_c(x) = \inf \{ t \mid c(t) \geq x \} \).

2. Since the clocks are not perfect, they drift apart over the period of time. We assume that the clock drift rate is bounded and the local clocks are synchronized to extent that corresponding local clock times happen within a \( 2\epsilon \) interval in global time. If the maximum clock drift rate

\(^1\)In Chapter 5, we consider the logical synchronization between multi-rate computations.
is $\rho$, then

\[ 1 - \rho \leq \frac{c(t_1) - c(t_2)}{t_1 - t_2} \leq 1 + \rho \]

Let $c(t_1) = x$, then $t_1$ happens in the global time interval $[x - \epsilon, x + \epsilon]$. Therefore, for any two clocks, $c_1$ and $c_2$, if $c_1(t_1) = c_2(t_2) = x$, then $|t_1 - t_2| \leq 2\epsilon$. $\epsilon$ is defined as the worst-case clock skew with respect to the global time. In this thesis, we assume that when a clock is ahead/behind, it should be corrected by decreasing/increasing its rate of progress. A clock value that goes backwards or has large jumps generates serious errors in the computation of velocity and acceleration in control systems.

3. Response time of a computation task, $\alpha$ is bounded, i.e. $0 < \alpha^{\min} \leq \alpha \leq \alpha^{\max}$. The response time of a task is based on the scheduling policy and other tasks running in the same node.

4. Message transmission delay, $\mu$ is bounded, i.e., $0 < \mu^{\min} \leq \mu \leq \mu^{\max}$. $\mu$ includes the network queuing and scheduling delays during the message transmission.

5. Nodes are fail-stop and may recover later. The output of a crashed node is assumed to be ‘null’. A failed node must not be able to send extra messages during a period. This could result in tasks receiving messages inconsistently.\(^2\)

Note 1: These assumptions are realizable in existing real-time systems, such as avionics. These systems have bounds for clock skew, response time, and message transmission delay. For example, the nodes in an avionic system communicate through a fault-tolerant, real-time network. The communication architecture guarantees real-time, reliable delivery of messages. Furthermore, the nodes synchronize the clocks to minimize the jitter of sampling and control operations.

Note 2: The PALS pattern is applicable in the coordination of distributed computations that require consistent views and actions at different nodes. For example, the supervisory control logic of a flight control system performs various discrete mode control, e.g. selecting the primary controller in a dual-redundant flight control system. The supervisory control logic uses the PALS pattern to guarantee consistency during the mode changes. On the other hand, there are local computations in a node that only depend on the local state. For example, health monitoring of the internal subsystems at a node is a local computation. These local computations can be implemented without this pattern since they do not have to be synchronized with other nodes.

\(^2\)We discuss the fault model of the PALS system in Chapter 6.
2.2 PALS System Rules

(a) Globally Synchronous System

(b) PALS System

Figure 2.1: Logically equivalent globally synchronous system and PALS system.

The main concept of logical synchronization in the PALS system is simple and is illustrated in Figure 2.1. In the globally synchronous system, the distributed tasks are triggered periodically by the global clock with a period $T$. Thus, these tasks dispatch in lockstep at the same global time. At each dispatch, a task reads messages from its input ports, processes the messages and sends its output messages to other nodes. In this lockstep execution, messages generated during the period $j$ are always consumed by their destination tasks in the period $j + 1$.

The PALS system guarantees an equivalent lockstep (synchronous) execution in the physically asynchronous architecture. In this architecture, the distributed tasks are now triggered periodically by the local clocks with the same period $T$. Since the local clocks are asynchronous, these tasks do not dispatch at the same global time. Despite these asynchronous dispatches, the PALS system will guarantee that these tasks dispatch within well-defined periodic intervals and messages generated during the period $j$ are always consumed by their destination tasks in the period $j + 1$. As a result,
the input views and operations of the tasks become identical to those in the synchronous model running with the same period.

The PALS system must satisfy some timing and external input constraints to achieve the logical synchronization. In the following subsections, we discuss these rules. (Later in Chapter 4, we will present a checker that validate these rules on the AADL models of the physically asynchronous system.)

2.2.1 R1 - PALS Clock Events

The PALS system defines a logical clock or timer, called PALS clock, to dispatch each distributed task. This logical clock defines a sequence of periodic events, called PALS clock events. In the PALS system, these PALS clock events happen at a regular interval when the local clock \( c \) equals to \( jT, \forall j \in N \). Here, \( T \) is the period of the distributed task. \( T \) is also referred to as the PALS clock period interval. In this case, we assume that the PALS clock events start from a pre-defined origin, called epoch\(^3\).

However, the PALS clock events do not happen at the same global time in different nodes. In a physically asynchronous architecture with a bounded clock skew of \( \epsilon \), the \( j^{th} \) PALS clock event happens between the global time \( jT - \epsilon \) and \( jT + \epsilon \). We refer the interval between two consecutive PALS clock events at \( t_c(jT) \) and \( t_c((j+1)T) \) as the PALS clock period \( j \).

2.2.2 R2 - PALS Causality or Output Hold Constraint

Since the PALS clock events are not perfectly synchronized, delivering a task’s outputs too early may result in the violation of the lockstep synchronization. It is possible that a message sent at the PALS clock period \( j \) could be consumed in the same PALS clock period \( j \) at a destination task. This situation is illustrated in Figure 2.2. \( M_1 \) is a sender and \( M_2 \) is a receiver with local clocks \( c_1 \) and \( c_2 \), respectively. In this figure, the PALS clock period \( j \) starts at the global time \( t_1 = t_{c_1}(jT) \) in \( M_1 \) and at the global time \( t_2 = t_{c_2}(jT) \) in \( M_2 \). Suppose that the end-to-end delay from \( M_1 \) to \( M_2 \) is smaller than \( 2\epsilon \). That is, \( \alpha + \mu < 2\epsilon \). In this condition, if the \( M_2 \)'s clock lags behind the \( M_1 \)'s clock, \( M_2 \) might receive the message before \( t_2 = t_{c_2}(jT) \). In a globally synchronous system, this could not happen without violating the causality. For the clock skew condition shown in Figure 2.2,

\(^3\)For example, the epoch for the UNIX system is January, 1, 1970 00:00.
this potentially leads to inconsistent input views at different destination tasks, when other tasks consume the same message during the PALS clock period $j + 1$.

![Figure 2.2: Violation of causality.](image)

In order to prevent this erroneous condition, a task executing during the PALS clock period $j$ is not allowed to send a message earlier than $t_{c_1}(jT + H)$, where $H = \max(2\epsilon - \mu_{\text{min}}, 0)$.

### 2.2.3 R3 - PALS Clock Period Constraint

The distributed computation of a node requires at least the delay of end-to-end computation and message transmission to know the state of the computations in other nodes. Hence, the distributed computations in a PALS system must run at a period longer than this delay. The PALS system defines an optimal lower bound for the PALS clock period, which is given below:

$$T > 2\epsilon + \max(\alpha^{\text{max}}, 2\epsilon - \mu_{\text{min}}) + \mu^{\text{max}}.$$  \hspace{1cm} (2.1)

![Figure 2.3: PALS clock period constraint.](image)
To explain this constraint, consider a worst-case clock synchronization illustrated in Figure 2.3. Here, the clock \( c_2 \) of \( M_2 \) (the receiver) leads the clock \( c_1 \) of \( M_1 \) (the sender) by \( 2\epsilon \). In this figure, the PALS clock period \( j \) starts at the global time \( t_2 = t_{c_1}(jT) = jT + \epsilon \) in \( M_1 \). The PALS clock period \( j + 1 \) starts at the global time \( t_3 = t_{c_2}((j + 1)T) = (j + 1)T - \epsilon \) in \( M_2 \). We need to ensure that a message sent by \( M_1 \) from the PALS clock period \( j \) will reach \( M_2 \) before the next PALS clock period \( j + 1 \) at \( t_3 = t_{c_2}((j + 1)T) \). As illustrated in Figure 2.3, the latest time at which \( M_1 \) transmits its messages on the network is \( t_{c_1}(jT + \max(H, \alpha_{max})) \), which can be as late as \( (jT + \max(H, \alpha_{max}) + \epsilon) \). The maximum message transmission delay is \( \mu_{max} \). It is easy to show that \( M_2 \) receives a message before the next PALS clock period \( j + 1 \), when PALS clock period, \( T > 2\epsilon + \max(\alpha_{max}, H) + \mu_{max} \).

Based on this constraint, the PALS system achieves an optimal logical synchronization. In [21, 22], we prove the optimality of the PALS system. The argument for the optimality is simple. The lower bound of the PALS clock period is equal to the worst-case end-to-end delay considering the worst-case clock skew. In a distributed system, we need at least the worst-case end-to-end delay time to achieve consistency at the distributed nodes.

Note: If a task transmits its message always after an interval \( H = \max(2\epsilon - \mu_{min}, 0) \), the PALS causality constraint is trivially satisfied. Thus, \( \alpha_{max} \) is always greater than \( \max(2\epsilon - \mu_{min}, 0) \). Hence, the PALS clock period constraint is \( T > 2\epsilon + \alpha_{max} + \mu_{max} \). This is also true when \( 2\epsilon \leq \mu_{min} \).

2.2.4 R4 - Environment Input and Output Synchronizer

In the PALS system, it is mandatory that the participating computations have consistent inputs in each PALS clock period. However, when an input source does not follow the above-mentioned timing constraints, the input may be potentially received in different PALS clock periods at the destinations tasks.

The PALS system therefore supports consistent delivery of external input messages. In a PALS system, we define a special component called, environment input synchronizer. An environment input synchronizer guarantees that the receiving tasks receive the external messages consistently in the same PALS clock period. It receives external messages asynchronously but delivers them synchronously to the intended receiving tasks. In the simplest form, an environment input synchronizer is a periodic task satisfying both PALS clock period constraint and PALS causality constraint. In this design, if an external input is received during the PALS clock period \( j \), the environment in-
put synchronizer re-transmits this input to the distributed computations at the PALS clock period \( j + 1 \). Thus, the PALS tasks receive an external input consistently in the same period. In [15], we give an implementation of a fault-tolerant, environment input synchronizer.

Similarly, one may use an environment output synchronizer to have a consistent view of the PALS computations at the external observers. The environment output synchronizer also executes in a similar way as other PALS computations and satisfies the above-mentioned timing constraints.

### 2.3 Overview of the PALS System Design

Based on the rules and system parameters of Section 2.1 and 2.2, one can design a PALS system that is logically equivalent to a globally synchronous system [21, 22]. The pattern thus enables a new design approach to apply the globally synchronous design systematically over a physically asynchronous architecture and preserve its correctness. In this thesis, we demonstrate this process for the architectural models in AADL. Figure 2.4 illustrates the steps of the design and verification of the PALS system models in AADL.

The first step is the design and verification of the synchronous design. With our colleagues at UIUC and University of Oslo, we have proposed a specification of the synchronous design in AADL [25]. This synchronous AADL specification models the lockstep execution of the globally

---

**Figure 2.4:** PALS pattern based design flow.

---

The first step is the design and verification of the synchronous design. With our colleagues at UIUC and University of Oslo, we have proposed a specification of the synchronous design in AADL [25]. This synchronous AADL specification models the lockstep execution of the globally
synchronous computations (Chapter 3). The synchronous AADL models are verified in Real-Time Maude [28], a formal verification tool for model checking and simulation.

We then map the synchronous AADL model onto the logically synchronous AADL model in a physically asynchronous platform according to our proposed pattern specification. In this thesis, we obtain the final PALS models in two steps. In the first step, we obtain a logically synchronous model by extending the globally synchronous model. We validate the timing requirements to guarantee the logical synchronization in the physically asynchronous model. This model however does not provide any modeling of the middleware or implementation components. It assumes that the implementation of this model satisfies the model’s timing and scheduling parameters. In the second step of the implementation, we extend the model to add middleware components to execute the verified globally synchronous model. In Chapter 4, we provide a detailed discussion of the PALS pattern specification and implementation. Eventually, the AADL specification must be translated into the software. An automatic or semi-automatic code generation is out of scope of this thesis.
CHAPTER 3
SYNCHRONOUS AADL SPECIFICATION

The first step of the PALS system implementation is the design and verification of the globally synchronous design. With our colleagues, we have proposed a specification of the globally synchronous design in AADL [25]. We later use the Synchronous AADL model as the input to the proposed PALS pattern in Chapter 4. The contents of this chapter are based on our paper published at ICFEM 2011 [25]. In the original work, we solve four problems:

- We specify a fragment of AADL, called *Synchronous AADL*, in which synchronous models can be defined.
- We provide the formal semantics to map models in Synchronous AADL to rewrite theories in Real-Time Maude, where such models can be simulated and verified by model checking.
- We develop an automatic verification tool, called *SynchAADL2Maude*, that automatically translates the Synchronous AADL model to Real-Time Maude specification.
- We illustrate this formal verification process for a dual-redundant active-standby model. We can formally verify this Synchronous AADL model in less than 10 second, whereas it is impossible to verify the corresponding asynchronous model because of the state-space explosion.

My contribution to this paper is limited to the specification and validation of a Synchronous AADL design. Since this thesis focuses on the architectural description of the PALS system, we only discuss the Synchronous AADL model definition and illustrate it with an example in this chapter. We refer to the original paper for the formal verification of the AADL model. We also present an architectural analysis that validates the Synchronous AADL model before verifying it in Real-Time Maude.
3.1 Synchronous Model

In the following, we provide a brief intuitive explanation of the synchronous model. This model is formalized in [22]. We assume that there are \( N \) distributed tasks that participate in a distributed interaction on a globally synchronous architecture. In the Synchronous AADL model, the behavior of each distributed task is modeled as a deterministic typed machine \( M \). Each typed machine is defined as an automaton \( M = (D_i, S, D_o, \delta_M) \), consisting of a set of states \( S \), sets \( D_i \) and \( D_o \) of outputs, and a next-state-and-output-function, \( \delta_M : (D_i \times S) \rightarrow (S \times D_o) \). In this model, the input and output sets can be given by the Cartesian products of the data sets of basic data types of the task’s input and output ports.

For the external components that are not synchronized with the distributed tasks, this model considers an environment component that generates nondeterministic input to the tasks. In this model, the environment generates input messages based on user-defined constraints. In this synchronous model, constraints on the values generated by the environment can be defined as a satisfiable predicate \( c_e : D_o^e \rightarrow \text{Bool} \) so that \( c_e(d_1, \ldots, d_{m_e}) \) is true if and only if the environment can generate output \( (d_1, \ldots, d_{m_e}) \).

The Synchronous AADL forms a synchronous composition of a collection of deterministic typed machines, a nondeterministic environment, and a wiring diagram that connects the machines. Fig. 3.1 gives an example of this “wiring diagram”. An ensemble has a synchronous semantics that all machines perform a transition simultaneously, and whenever a machine has a feedback wire to itself and/or to any other machine, then the corresponding output becomes an input for any such machine at the next step. Meseguer and Ölveczky show a further improvement by composing

![Figure 3.1: A machine ensemble.](image-url)
these distributed state machines into a single state machine so that model checking becomes more efficient [22].

3.2 AADL Model Definition

We define Synchronous AADL as an annotated sub-language of AADL. In this specification, each typed machine of the synchronous model is designed as an AADL thread that executes periodically. In each period, the thread reads messages from the input ports, performs the state transitions and produce output messages. The execution of each thread in each period is independent of the other threads, and where output generated by a thread in a period is available as input at the receiving thread exactly at the beginning of the next period.

Since Synchronous AADL is intended to model synchronous designs as opposed to asynchronous implementations, it ignores the hardware and scheduling features of AADL. Synchronous AADL therefore focuses on the behavioral and structural subset of AADL, namely, hierarchical system, process, and thread components, ports, connections, and thread behaviors.

We next discuss the definition of Synchronous AADL.

Property set. Synchronous AADL adds a property set SynchAADL to declare Synchronous AADL-specific properties as explained below.

Dispatch. The dispatch protocol is used to trigger an execution of an AADL thread. A periodic thread is dispatched at the beginning of each new period of the thread. In aperiodic, sporadic, timed, and hybrid dispatch, a thread is dispatched when it receives an event. Such event-triggered dispatch is not suitable to define a system in which all threads (with a possible exception for the environment thread) should execute in lockstep, since the sending thread triggers the execution of the receiving thread, which would read in its $j^{th}$ period the output generated by the sender in the same period. Therefore, each thread must have periodic dispatch. Furthermore, since each thread must execute in each period, the period of all the threads must be the same.

Communication. There are three kinds of ports in AADL: data, event, and event data ports. Event and event data ports can be used to dispatch event-triggered threads. To have only AADL

---

1 The thread behavior is modeled in the Behavior Annex standard [29]. An annex allows a user to extend the AADL language with specialized notations and modeling capabilities. The Behavior Annex defines the states, the variables, and the state transitions of a thread or a subprogram.
constructs that define “synchronous behaviors”, the communication primitives must ensure that all output generated in a period is available to the receiver at the beginning of the next period, and not earlier.

The AADL standard defines three timing semantics for the data connections: sampled, immediate, and delayed connection timing. According to the AADL standard [24, Section 9.2.5], a periodic destination thread uses the sampled data port connection to sample a data stream. Since the sampling at a destination thread is independent of the source thread, the sampling can result in a non-deterministic message exchange depending on the availability of the output message. In Synchronous AADL, we intend to support deterministic message communication. We therefore do not use sampled data connections.

AADL supports deterministic message communication for immediate and delayed connections between periodic threads. We however find the delayed data connections to be suitable for the lockstep synchronous communication.

The immediate data connections impose a strict scheduling order between the source and the destination threads. When both source and destination threads dispatch at the same time, the scheduler delays the dispatch of the destination thread until the source thread has completed its execution. In our setting, since the threads execute with the same period and the dispatch time of the source and destination threads coincide, an immediate connection will cause the destination thread to receive the output messages in the same period. As a result, the immediate connection timing violates the intended “lock-step” semantics of the Synchronous AADL model.

On the other hand, for a delayed connection, the value from the sender is transmitted at its deadline and is available to the receiver at its next dispatch. In our setting, where all threads have periodic dispatch with the same period, the output generated in a period is, therefore, available at the start of the next period. Since only data ports have delayed connections, and since event-triggered dispatches are excluded, only data ports are used in Synchronous AADL, and all connections between non-environment threads must be delayed.

**Execution times.** In a real-time distributed system, the execution times of the threads are bounded. Since the threads execute in lock-step and the destination threads receive the messages only at the next period, for simplicity, we can assume that thread executions are instantaneous.
Deterministic threads. In the systems targeted by PALS and Synchronous AADL, the nodes that communicate with the environment are invariably deterministic. We therefore assume that the the behavior of a non-environment thread is deterministic. Each such thread has the property \texttt{SynchAADL::Deterministic => true}.

Environment thread. In Synchronous AADL, the environment thread generates output non-deterministically in each period. The possible outputs can often be defined by an environment constraint \( c_e \) so that \( c_e(\vec{o}) \) is \textit{true} if and only if the environment can nondeterministically generate output \( \vec{o} \) in any iteration. The property \texttt{SynchAADL::IsEnvironment => true} denotes that the thread is an environment thread, and \texttt{SynchAADL::InputConstraints => ("Boolean formula")} defines an input constraint on a set of Boolean-valued outputs. We assume that a Synchronous AADL system has \textit{at most one environment thread}.

It seems natural to regard the system as responding to the \textit{current} environment output. We, therefore, support only \textit{immediate} connections from the environment. According to the AADL semantics, this forces the environment to execute before the other nodes in each period.

\textit{Note}: In PALS, external inputs are propagated through the environment input synchronizer. In Synchronous AADL, the environment input synchronizer must be modeled as a deterministic typed machine. The environment input synchronizer acts as a relay between the environment thread and other deterministic typed machine. The environment input synchronizer receives the external inputs from the environment thread in each period by using immediate connections. On the other hand, the environment input synchronizer uses delayed connections to propagate the data to other deterministic typed machines.

Declaring synchronous systems. The top-level \texttt{system} component declares the entire system to execute synchronously by declaring \texttt{SynchAADL::Synchronous => true}. The period of the system can be declared by \texttt{SynchAADL::SynchPeriod => p}. A Synchronous AADL model defines a synchronous machine ensemble in the obvious way, as mentioned above.

In summary, Table 3.1 lists the AADL properties of \texttt{SynchAADL} that we use to annotate the components and the connections in Synchronous AADL. In Appendix B.1, we provide the AADL definitions of these properties.
<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>Boolean</td>
<td>If the top-level encapsulating system contains synchronously executing threads. The threads inherits this property, too.</td>
</tr>
<tr>
<td>SynchPeriod</td>
<td>Time</td>
<td>Period of the synchronously executing threads.</td>
</tr>
<tr>
<td>IsEnvironment</td>
<td>Boolean</td>
<td>If a thread is environment or not.</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Boolean</td>
<td>If a thread is deterministic or not. A synchronously executing, non-environment thread is assumed to be deterministic.</td>
</tr>
</tbody>
</table>

Table 3.1: Synchronous AADL property set, SynchAADL

3.3 Active-Standby System

We exemplify Synchronous AADL with fragments of a model of a dual-redundant system, called the active-standby system. This example is originally presented in our previous paper [15].

3.3.1 Description

The active-standby system consists of two physically separated components: Side1 and Side2. They are connected by a fault-tolerant real-time network. Only one side should operate in the active mode, while the other stays in the standby mode.

In this illustration of Synchronous AADL, we focus on the logic for deciding which side is active. Each side can fail independently, but both of them cannot fail at the same time. A failed side can recover after failure. When one side fails, the non-failed side should be the active side. In addition, the user can send commands to these components to toggle their active-standby status and switch the active side. The full functionality of each side depends on the two sides’ perception of the availability or accuracy of other subsystems. Each side can sense the status of the subsystems and determine if the subsystems are fully available or not. There are five requirements that the coordination logic must satisfy [15]:

1. **Both sides should agree on which side is active** (provided neither side has failed, the availability of a side has not changed, and the user has not issued a command for switching the active side).

2. **A side that is not fully available should not be the active side if the other side is fully available** (assuming that at least one side is fully functional and the partial failure is detectable).

3. **The user can always change the active side when both sides are fully functional.**

4. **If a side is failed the other side should become active.**
5. The active side should not change unless the availability of a side changes, the failed status of a side changes, or the user commands.

Figure 3.2: The architecture of the active-standby system.

The architecture of the system is shown in Figure 3.2. We model an environment component as **Environment**. Each time **Environment** dispatches, it nondeterministically sends 5 Boolean values, one through each port. The environment data, **side1Failed** and **side2Failed**, are used to inject failure to these sides. **manualSelection** is used to model a switch command from the user. If **manualSelection** = true, then the user commands to switch the active side. **side1FullyAvailable** and **side2FullyAvailable** are two boolean inputs to model the full availability of Side1 and Side2, respectively. These two variables in our model capture the overall statuses of subsystems dedicated for both Side1 and Side2. In this model, we assume that the statuses of different subsystems are available to both sides.

In this model, both sides exchange their status through the outputs: **side1ActiveSide** and **side2ActiveSide**. They capture the active, standby or failure status of these components. For example,

- If Side1 is active, then Side1 outputs **side1ActiveSide** = 1.

- If Side2 is active, Side1 outputs **side1ActiveSide** = 2.

\[\text{In the physically asynchronous model, an environment input synchronizer distributes these data.}\]
Otherwise, if Side1 is failed, the default output of \textit{side1ActiveSide} is 0. Side2 then becomes active observing this value.

Thus, both sides agree on which one is active when \textit{side1ActiveSide} = \textit{side2ActiveSide} (Requirement 1).

### 3.3.2 The Synchronous AADL Model

The following top-level system implementation declares the architecture of the system, with the three subcomponents \texttt{sideOne}, \texttt{sideTwo}, and \texttt{env}, and with immediate data connections (denoted with property \texttt{Timing=>Immediate}) from the environment to the two sides, and with delayed data connections (\texttt{Timing=>Delayed}) between the two sides. The top-level system component also shows the thread dispatch and the \texttt{SyncAADL} properties for the synchronous threads in the system. Parts of the model are replaced by ‘...’. We provide the complete Synchronous AADL model of the active-standby system in Appendix C.

```aadl
system implementation ActiveStandbySystem.impl
  subcomponents
    sideOne: system Side1.impl;
    sideTwo: system Side2.impl;
    env: system Environment.impl;
  connections
    C1: port sideOne.side1ActiveSide -> sideTwo.side1ActiveSide;
    C2: port sideTwo.side2ActiveSide -> sideOne.side2ActiveSide;
    C3: port env.side1FullyAvail -> sideOne.side1FullyAvail;
    C4: port env.side1FullyAvail -> sideOne.side1FullyAvail;
    ...
  properties
    SynchAADL::Synchronous => true;
    SynchAADL::SynchPeriod => 2 ms;
    Timing => Delayed applies to C1;
    Timing => Delayed applies to C2;
    Timing => Immediate applies to C3;
    Timing => Immediate applies to C4;
    SynchAADL::Deterministic => true applies to sideOne.sideProcess.sideThread;
    SynchAADL::IsEnvironment => false applies to sideOne.sideProcess.sideThread;
    DispatchProtocol => Periodic applies to sideOne.sideProcess.sideThread;
    Period => 2 ms applies to sideOne.sideProcess.sideThread;
    ...
end ActiveStandbySystem.impl;
```

Inside the \texttt{sideOne} and \texttt{sideTwo} systems, there is a thread that interacts in this configuration. For example, \texttt{Side1Thread} is defined in Side1. We model the state transitions of these threads
using the AADL Behavior Annex notations for states and transitions. Here, we show the thread implementation inside Side1. **Side1Thread** defines the thread *type* and Side1Thread.impl is the thread *implementation*.

thread Side1Thread
features
  side1FullyAvail: in data port Base_Types::Boolean;
  side2FullyAvail: in data port Base_Types::Boolean;
  side2ActiveSide: in data port Base_Types::Integer;
  manualSelection: in data port Base_Types::Boolean;
  side1Failed: in data port Base_Types::Boolean;
  side1ActiveSide: out data port Base_Types::Integer;
end Side1Thread;

thread implementation Side1Thread.impl
annex behavior_specification {**
  states
  preInit: initial complete state;
  side1ActiveState, side2ActiveState, ...: complete state;
  side2ActiveState_tmp , ... : state;
  state variables
  prevSide2ActiveStatus: Base_Types::Integer; prevManSwitch: Base_Types::Boolean;
  transitions
  ... 
  side2ActiveState -[ on dispatch ]-> side2ActiveState_tmp;
  side2ActiveState_tmp -[side1Failed = false and side2ActiveSide != 0 and
  side1FullyAvail = true and ((prevManSwitch = false and
  manualSelection = true) or
  side2FullyAvail = false)]-> side1ActiveState
  {side1ActiveSide := 1; prevSide2ActiveStatus := side2ActiveSide;
   prevManSwitch := manualSelection};
  ...
**};
end Side1Thread.impl;

We show only one state transition in this thread, in which Side1 becomes active upon receiving the manualSelection input from the user. The transition takes the thread from state side2ActiveState to state side1ActiveState if the input received in the side1Failed port is false, the value received in the port side2ActiveSide is different from 0, etc. As a result of applying the transition, the value 1 is sent through the output port side1ActiveSide, and the local variables prevSide2ActiveStatus and prevManSwitch are assigned the values received in the ports side2ActiveSide and manualSelection, respectively.3

---
3 A state transition in a periodic thread typically happens in two steps in Behavior Annex. The first step is about the dispatch of the thread. In the next step, we can use the input and state variables to define the actual transition.
The following code shows the thread component implementing the environment. We do not show the definition of the system env containing this thread component:

```plaintext
thread EnvironmentThread
features
  side1FullyAvail: out data port Base_Types::Boolean;
  side2FullyAvail: out data port Base_Types::Boolean;
  manualSelection: out data port Base_Types::Boolean;
  side1Failed: out data port Base_Types::Boolean;
  side2Failed: out data port Base_Types::Boolean;
end EnvironmentThread;

thread implementation EnvironmentThread.impl
properties
  SynchAADL::InputConstraints => "(not (s1F and s2F))";  SynchAADL::IsEnvironment => true;
  Dispatch_Protocol => Periodic;
  Period => 2 ms;
annex behavior_specification {**
states
  s0 : initial complete final state;
state variables
  s1FA: Base_Types::Boolean;  s2FA: Base_Types::Boolean;  mS: Base_Types::Boolean;
  s1F: Base_Types::Boolean;  s2F: Base_Types::Boolean;
transitions
  s0 -> [on dispatch] s0 { side1FullyAvail := s1FA;  side2FullyAvail := s2FA;
    manualSelection := mS;  side1Failed := s1F;  side2Failed := s2F};
**};
end EnvironmentThread.impl;
```

The environment has a single transition, that sends the values of the state variables s1FA, s2FA, mS, s1F, and s2F to the corresponding output ports. In each periodic dispatch of this thread, we execute this transition and assign any values that can satisfy the constraint not (s1F and s2F). Hence, both sides do not fail at the same time.

### 3.4 Static Analysis

We have developed a static analysis tool to check the rules of Synchronous AADL. A model using Synchronous AADL needs to satisfy these rules before it can be considered for the formal verification. We have developed a plug-in for the Open Source AADL Tool Environment (OSATE), the AADL development environment [30]. OSATE is the development environment for AADL-based model development and analysis. It is built as a set of plug-ins in the open source Eclipse platform.
We use the OSATE API to traverse the Synchronous AADL model and check if the required AADL properties of threads and connections follow the Synchronous AADL specification.

**Explanation of the symbols.** For brevity, we use first-order logic to describe the rules. We consider that Synchronous AADL configuration of a top-level system component $S$ is given by $G_{\text{sync}} = (\text{Threads}_{\text{sync}}, \text{Conns}_{\text{sync}})$. It consists of the set of threads, $\text{Threads}_{\text{sync}} = \{ M_i \}$ and the set of thread connections, $\text{Conns}_{\text{sync}} = \bigcup_{i,j} \text{Conns}_{\text{sync}}(M_i, M_j)$, where $\text{Conns}_{\text{sync}}(M_i, M_j)$ is the set of all connections from a thread $M_i$ to another thread $M_j$. The set of all connections to a thread $M_j$ is also given by $\text{Conn}(\ast, M_j)$. For each connection $C_n$, we assume that $C_n.\text{SrcPort}$ and $C_n.\text{DstPort}$ provide information on the source and the destination port of the connection, respectively. We also assume that $C_n.\text{Src}$ and $C_n.\text{Dst}$ give the corresponding source and destination thread, respectively.

We use the notation $T.\text{PropertyName}$ to give the values of an AADL property of $T$, where $T$ is an AADL construct, e.g. component, connection or port. For example, $M_i.\text{DispatchProtocol}$ gives the DispatchProtocol property value of a thread $M_i$. The return values may be Periodic, Aperiodic, etc. $M_i.\text{Period}$ gives the Period property value of a thread $M_i$. Similarly, we get the values of the SynchPeriod, Deterministic, IsEnvironment, Synchronous, SynchPeriod properties defined in Synchronous AADL. We assume that if a property is not defined, the property returns a default undefined value.
Sync_R1. The top-level system component $S$ is synchronous.

$$S.\text{Synchronous} = \text{true}.$$  

Sync_R2. The threads of the Synchronous AADL model are periodic.

$$\forall M_i \in \text{Threads}_{\text{sync}} \ M_i.\text{Dispatch Protocol} = \text{Periodic}$$

Sync_R3. Both SynchPeriod and Period must be defined for each thread. They must be equal.

$$\forall M_i \in \text{Threads}_{\text{sync}} \ M_i.\text{SynchPeriod} \neq \text{undefined} \land M_i.\text{SynchPeriod} = M_i.\text{Period}$$

Sync_R4. All non-environment threads are deterministic.

$$\forall M_i \in \text{Threads}_{\text{sync}} \ M_i.\text{IsEnvironment} \neq \text{false} \rightarrow M_i.\text{Deterministic} = \text{true}$$

Sync_R5. All data connections in Synchronous AADL are data port connections.

$$\forall Cn \in \text{Conns}_{\text{sync}} \ (Cn.\text{SrcPort}).\text{type} = (Cn.\text{DstPort}).\text{type} = \text{‘data port’}$$

Sync_R6. The connections to the non-environment threads use delayed timing.

$$\forall Cn \in \text{Conns}_{\text{sync}} \ Cn.\text{Dst.IsEnvironment} = \text{false} \rightarrow Cn.\text{Timing} = \text{delayed}$$

Sync_R7. The connections from the environment threads use immediate timing.

$$\forall Cn \in \text{Conns}_{\text{sync}} \ Cn.\text{Src.IsEnvironment} = \text{true} \rightarrow Cn.\text{Timing} = \text{immediate}$$

Table 3.2: Synchronous AADL rules.
The PALS system executes the globally synchronous model onto a physically asynchronous architecture. The proof of the logical equivalence between the PALS system and the globally synchronous system is provided in [21, 22]. However, such logical equivalence does not immediately produce correct asynchronous designs. Validation of the asynchronous implementation is still a developer’s responsibility. In the absence of any generic analysis framework, each implementation must go through a costly validation process to prove the logical synchronization.

We therefore propose a formal pattern specification and analysis framework for the PALS system in this thesis. In our proposed approach, system engineers model the Synchronous AADL model of an application and transform this model to work on a physically asynchronous architecture. In a model-based design, this architectural transformation involves component inheritance, modification of architectural features or properties, etc. In this thesis, we do not provide a tool to have an automatic transformation of the Synchronous AADL model. Since the goal for the PALS design philosophy is to provide a correct-by-construction architecture for logical synchronization, we instead provide an analysis framework that works for different instantiations of this pattern. This framework validates the assumptions and rules of Chapter 2 to show that the transformation is correctly implemented and the resulting model is logically synchronous.

In following section, we present the AADL model definition of the logically synchronous design. We illustrate this specification in an active-standby system in Section 4.2. We then present the mapping of the Synchronous AADL model to the PALS model and the analysis rules in Section 4.3. In Section 4.4, we discuss a reverse transformation process to generate the synchronous model automatically from a PALS model. Finally in Section 4.5, we discuss the basic middleware components of the PALS system in AADL.
4.1 AADL Model Definition

In this section, we describe an architectural specification of the PALS system. We first proposed this specification with Steven P. Miller and Darren D. Cofer in [11].

**Property set.** In this specification, we define a small set of AADL properties to map the globally synchronous components and connections in the physically asynchronous model. We use these properties to describe the scope of the logically synchronous interactions. The properties are defined in a AADL property set, called \texttt{PALS\_Properties}.

**Scope of logical synchronization.** In a distributed application, each node participates in more than one computation. Only a subset of the computations may depend on the global system states and thus, need to be logically synchronized. The PALS system rules are applicable only to these logically synchronous computations. In the asynchronous model, we annotate the components and connections of these computations with two AADL properties, \texttt{PALS\_Properties::PALS\_Id} and \texttt{PALS\_Properties::PALS\_Connection\_Id}. These properties accept a \textit{string} literal as the value. With these two properties, we define a \textit{logical synchronization group} that consists of a set of components and connections with the same value for these properties.

**Thread execution.** In the PALS system, the logically synchronous components execute periodically with the same period. In this model, these components are modeled as AADL \textit{threads}. We annotate these threads with a property, \texttt{PALS\_Properties::PALS\_Period=>T}, to define the PALS clock period of $T$. Based on the PALS clock event rule ($R1$) of Section 2.2.1, these threads dispatch at the local clock time $jT$ during a period $j$, for all $j \in \mathbb{N}$.

In this model, we also use other pre-defined AADL properties to declare the scheduling and timing parameters. For example, we declare the \texttt{Deadline} of these threads to define the latest time by which a thread must finish its computation. The deadline can also be viewed as a parameter for the maximum response time, given as $\alpha_{max}$ in the PALS system model.

**Output timing** In the PALS system, we model the output times of a thread to satisfy the PALS output hold constraint. There is a pre-defined property \texttt{Output\_Time} that we use to declare output times of the ports of a thread. The output time is specified with respect to a reference time, e.g.
dispatch, completion, deadline, etc. of a thread. For example,

\[
\text{Output\_Time} \Rightarrow ([\text{Time} \Rightarrow \text{Dispatch}; \text{Offset} \Rightarrow 1\text{ms..2ms;}], \\
[\text{Time} \Rightarrow \text{Completion}; \text{Offset} \Rightarrow 0\text{ns..0ns;}]);
\]

Here, \text{Output\_Time} defines two output time ranges with respect to the dispatch and completion of a thread.

In the PALS system, we only care about the earliest and the latest time at which a thread transmits its output since the dispatch. In order to provide a simple output time representation for the PALS pattern, we define a property, called \text{PALS\_Properties::PALS\_Output\_Time} for the logically synchronous threads. We derive its value from the output time of the ports of a thread. This property accepts a single time range. The minimum and maximum value of this time range is equal the earliest and the latest output time of the ports of a logically synchronous thread.

\textbf{Latency.} In AADL, the connections between the two threads define the message transmission delay by using an AADL property, called \text{Latency}. This property accepts a time range, specifying the minimum and the maximum delay of a connection. In the PALS system, the message transmission delay parameters ($\mu^{\text{min}}, \mu^{\text{max}}$) are computed from the \text{Latency} values of the connections between the logically synchronous threads.

\textbf{Clock skew.} The clock skew of the system is defined as a pre-defined AADL property, called \text{Clock\_Jitter}. The system and the processor components declare this property. Maximum clock skew of a system is equal to the maximum of these \text{Clock\_Jitter} values.

\textbf{Environment input and output synchronizer.} In the PALS system, the logically synchronous threads receive external inputs from an environment input synchronizer. An environment input synchronizer is also implemented as an AADL thread. It executes periodically at the same rate as the logically synchronous threads. In this model, an environment input synchronizer declares a property, \text{PALS\_Properties::PALS\_Synchronizer\_Type} => \text{Environment\_Input\_Synchronizer}. It also declares the above-mentioned properties: \text{PALS\_Properties::PALS\_Id, Period, Deadline, PALS\_Properties::PALS\_Output\_Time, Output\_Time, Latency, and Clock\_Jitter}. The connections between an environment input synchronizer and the logically synchronous threads also define \text{PALS\_Properties::PALS\_Connection\_Id} with the same value as \text{PALS\_Properties::PALS\_Id}.
Similarly, an environment output synchronizer declares the property, `PALS_Synchronizer_Type` with the value of `Environment_Output_Synchronizer`. Since the external components do not directly contribute to the logically synchronous interactions, we do not require any analysis on the connections between the environment output synchronizer and the external components.

Optional environment inputs. During the system design and verification, developers often consider optional inputs to model certain behaviors, e.g. faults, environmental influences. For example, in the verification of the fault-tolerant applications, abstract fault injection variables are used to inject faults in a component. In the analysis tools, e.g. model checkers, the environment component of the Synchronous AADL model generates these external inputs to control the execution of the applications. However, these input ports may not be connected to an input source in the actual implementation. In order to support these optional environment input ports of the globally synchronous model, we use an AADL property, `Required_Connection=>false`. We use a default value for the input ports when the connection is not defined. The default value is specified with a pre-defined AADL property, `Data_Model::Initial_Value`. These two properties allow us to preserve the component interface of the globally synchronous threads in the physically asynchronous model.

4.2 Active-Standby System

In this section, we illustrate the specification of Section 4.1 with an example of a dual-redundant control system. In this example, two redundant controllers execute an active-standby logical synchronization. Only the active controller controls the physical system, while the other controller acts as a standby. In Section 3.3, we describe the globally synchronous coordination of the active-standby design. In that model, each side implements a globally synchronous thread to agree on “who is active”. In this section, we map this globally synchronous model on a physically asynchronous model of this control system.

Figure 4.1a gives a partial view of the top-level system. In addition to the two control sub-systems, this model has other subsystems, e.g. sensor, actuator, user interface. In the physically asynchronous model, we declare both software and hardware components. Each subsystem executes on a separate processor and communicates through bus components (not shown in the figure). In this model, user input of `manualSelection` originates from a device (InputDevice) at
(a) Top-level system component.

(i) In the synchronous model

(ii) In the asynchronous model

(b) Transformation from synchronous model to asynchronous model.

Figure 4.1: Asynchronous active-standby system.
the **UserInterface** subsystem. An environment input synchronizer (**InputSynchronizerThread**) reads this **maualSelection** input from **InputDevice**. This environment input synchronizer then propagates **maualSelection** to the control subsystems, **Side1** and **Side2**. At each control side, we implement two threads: one for active-standby coordination (e.g. **Side1Thread**) and another for the feedback control logic (e.g. **ControlThread**). These two threads execute in the same address-space of an AADL process and on the same processor. Similarly, the sensor and the actuator subsystems execute on their own processor and define periodic computations for sensing and actuation.

In this physically asynchronous model, the control thread in each side uses the output of the active-standby coordination logic to act as an active controller and control the actuator device. Figure 4.1b shows the mapping of a globally synchronous thread at the subsystem **Side1** to the corresponding thread in the physically asynchronous model. In the globally synchronous design, we model a periodic thread, **Side1Thread** in **Side1** that performs the active-standby coordination with **Side2** and the environment threads. The same thread is used in the asynchronous model along with the control thread, **ControlThread**. The control thread uses the output (**side1ActiveSide**) of **Side1Thread** to know the active/standby status of **Side1**.

The following AADL code shows the instantiation of **Side1Thread** and its surrounding process of the Synchronous AADL model in the physically asynchronous model. **Side1Thread**(shown in Figure 4.1b) is directly used in the process, **AsyncSide1Process.impl** of the asynchronous active-standby system. In this model, the transformation happens by extending the process element (**Side1Process**) of the the Synchronous AADL model and adding the local control thread.

```aadl
-- Side1Process in the Synchronous AADL Model.
process Side1Process
features
    side1ActiveSide: out data port Base_Types::Integer;
    side2ActiveSide: in data port Base_Types::Integer;
    side1Failed: in data port Base_Types::Boolean;
...
end Side1Process;

process implementation Side1Process.impl
subcomponents
    sideThread: thread Side1Thread.impl;
connections
    port sideThread.side1ActiveSide -> side1ActiveSide;
    port side2ActiveSide -> sideThread.side2ActiveSide;
    port side1Failed -> sideThread.side1Failed;
...
end Side1Process.impl;
```

33
-- Asynchronous process implementation with Side1Thread and the ControlThread.
process AsyncSide1Process extends Side1Process
features
    sensor: in data port Base_Types::Float;
    command: out data port Base_Types::Float;
end AsyncSide1Process;

process implementation AsyncSide1Process.impl extends Side1Process.impl
subcomponents
    controlThread: thread PalsController::ControlThread.impl;
connections
    port sensor -> controlThread.sensor;
    port controlThread.command -> command;
    port sideThread.side1ActiveSide -> controlThread.sideActiveSide;
properties
    Required_Connection => false applies to sideThread.side1FullyAvail;
    Data_Model::Initial_Value => ("true") applies to sideThread.side1FullyAvail;
    Required_Connection => false applies to sideThread.side2FullyAvail;
    Data_Model::Initial_Value => ("true") applies to sideThread.side2FullyAvail;
    Required_Connection => false applies to sideThread.side1Failed;
    Data_Model::Initial_Value => ("false") applies to sideThread.side1Failed;
end AsyncSide1Process.impl;

In the transformation, not all input connections of the Synchronous AADL model are used in the asynchronous model. For example, side1Failed is used for fault injection into Side1 and is only meaningful in the verification of the Synchronous AADL model. In the asynchronous model, this port declares two properties, Required_Connection and Data_Model::Initial_Value.

Finally, we define the AADL properties of the threads and the connections according to the specification of Section 4.1. In this example, we declare these properties at the top-level system component.

system implementation ActiveStandbySystem.impl
subcomponents
    sideOne: system PalsSide1::Side1.impl;
    sideTwo: system PalsSide2::Side2.impl;
    console: system PalsConsole::Console.impl;
    sensor: system PalsSensor::Sensor.impl;
    actuator: system PalsActuator::Actuator.impl;
connections
    Side1toSide2AS: port sideOne.side1ActiveSide -> sideTwo.side1ActiveSide;
    SensorToSide1: port sensor.output -> sideOne.sensor;
    ...
properties
    Dispatch_Protocol => Periodic applies to sideOne.sideProcess.sideThread;
There are alternative modeling solutions for mapping the globally synchronous threads in the physically asynchronous model. For example, it is not necessary to extend the process component of the Synchronous AADL model in the asynchronous model. We could directly use a globally synchronous thread in a process component of the asynchronous model. In either approaches, we must ensure that the asynchronous architecture guarantees the logical synchronization. In the following section, we therefore generalize this transformation process.

4.3 PALS Transformation and Analysis

In this section, we discuss the formal mapping between the Synchronous AADL model and the logically synchronous model in an asynchronous platform. The pattern can be viewed as a model transformation process, $PALS(G_{sync}, AP)$ to obtain the asynchronous model in a real-time distributed application, $AP$. Here, $G_{sync} = (Threads_{sync}, Conns_{sync})$ is a Synchronous AADL model discussed in Chapter 3. The Synchronous AADL model consists of the set of threads $Threads_{sync}$ and the set of thread connections $Conns_{sync}$. We use the notations defined in Section 3.4.

In the pattern, we map the following set of components and connections of the Synchronous AADL model on the physically asynchronous model:
• Non-environment threads:

\[ \text{Threads}_{\text{sync}}^{\text{ne}} = \{ M \in \text{Threads}_{\text{sync}} \mid M.\text{IsEnvironment} = \text{false} \} \]

• Delayed connections between non-environment threads:

\[ \text{Conns}_{\text{sync}}^{\text{delayed}} = \{ Cn \in \text{Conns}_{\text{sync}} \mid Cn.\text{Src}.\text{IsEnvironment} = \text{false} \land Cn.\text{Dst}.\text{IsEnvironment} = \text{false} \} \]

• External (environment) input connections from an environment thread:

\[ \text{Conns}_{\text{sync}}^{\text{ext}} = \{ Cn \in \text{Conns}_{\text{sync}} \mid Cn.\text{Src}.\text{IsEnvironment} = \text{true} \land Cn.\text{Dst}.\text{IsEnvironment} = \text{false} \} \]

The physically asynchronous model consists of both hardware and software components. The software components can have both logical synchronous computations and local asynchronous computations. In this pattern transformation, we care about the threads, the connections, and the processors of a physically asynchronous model given by \( \text{Threads}_{\text{async}}, \text{Conns}_{\text{async}}, \) and \( \text{Processors}_{\text{async}} \), respectively. The pattern considers following functions for the target application \( \text{AP} \):

• A function mapping a synchronous non-environment thread to an asynchronous thread,

\[ T_{\text{thread}} : \text{Threads}_{\text{sync}}^{\text{ne}} \rightarrow \text{Threads}_{\text{async}}. \]

• A function mapping a delayed connection of the synchronous model to a connection of the asynchronous model,

\[ C_{\text{delayed}} : \text{Conns}_{\text{sync}}^{\text{delayed}} \rightarrow \text{Conns}_{\text{async}}. \]

• A function mapping an external input connection of the synchronous model to a connection of the asynchronous model,

\[ C_{\text{ext}} : \text{Conns}_{\text{sync}}^{\text{ext}} \rightarrow \text{Conns}_{\text{async}}. \]

• A function mapping a synchronous non-environment thread to the processor on which the
corresponding asynchronous thread executes,

\[ P_{\text{proc}} : \text{Threads}_{\text{sync}}^{ne} \rightarrow \text{Processors}_{\text{async}}. \]

In addition, not all connections of the synchronous model may be mapped onto the asynchronous model when the **Required_Connection** property is set to *false*. In these cases, we assume that a mapping function returns \( \phi (\text{null}) \). Furthermore, in the PALS pattern, external inputs are received through environment input synchronizer. We therefore define a function, \( \mathcal{E}_{\text{sync}} : \text{Conns}_{\text{sync}}^{ext} \rightarrow \text{Threads}_{\text{async}} \), to obtain the environment input synchronizer for an external connection.

Once these functions are defined, we validate the PALS pattern rules to guarantee the logical synchronization in this transformation.

### 4.3.1 Pattern Analysis

We have developed a static analysis framework for the correctness of the PALS transformation in the asynchronous system. This framework validates the timing rules of Chapter 2 as well as the interactions between the globally synchronous components and other environment components. Furthermore, system design is an evolving process. Designers may extend the design after the pattern transformation with additional features. This framework is also useful to preserve the logical synchronization during the design evolutions.

For the analysis of the asynchronous model, we derive the performance bounds of the system. For example,

- **Maximum message transmission delay** is equal to the maximum latency of all mapped connections of the delayed connections of the Synchronous AADL model in the physically asynchronous model.

\[
\mu_{\text{max}} = \max\{\{Cn.Latency \mid Cn \in \mathcal{C}_{\text{delayed}}(\text{Conns}_{\text{sync}}^{\text{delayed}})\}\}
\]

- **Minimum message transmission delay** is equal to the minimum latency of all mapped connections of the delayed connections of the Synchronous AADL model in the physically asynchronous model.

\[
\mu_{\text{min}} = \min\{\{Cn.Latency \mid Cn \in \mathcal{C}_{\text{delayed}}(\text{Conns}_{\text{sync}}^{\text{delayed}})\}\}
\]
• **Maximum clock skew:** Maximum clock skew is based on the maximum clock jitter of the processors in the physically asynchronous model.

\[ \epsilon = \max(\{Pr.Clock\_Jitter \mid Pr \in \text{Processors}_{\text{asyn}}\}) \]

Here, \( \max(.) \) and \( \min(.) \) functions return the maximum and the minimum value of a set.

### Table 4.1: PALS specification rules.

<table>
<thead>
<tr>
<th>PALS_R1</th>
<th>The periods and PALS clock periods of the logically synchronous threads, including the environment input synchronizers must be equal in the physically asynchronous model (PALS system rule R1 of Section 2.2.1).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let ( M_S = (T_{\text{thread}}(\text{Threads}<em>{\text{sync}}) \cup E</em>{\text{sync}}(\text{Conns}_{\text{ext}})) \setminus {\phi} ).</td>
<td></td>
</tr>
<tr>
<td>( \forall M_i, M_j \in M_S \ M_i.\text{Period} = M_j.\text{Period} = M_i.\text{PALS_Period} = M_j.\text{PALS_Period} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PALS_R2</th>
<th>The earliest output time of a logically synchronous thread must satisfy the PALS causality or output hold rule (PALS system rule R2 of Section 2.2.2).</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall M_i \in M_S \ \min(M_i.\text{PALS_Output_Time}) &gt; \max(2\epsilon - \mu^{\text{min}}, 0) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PALS_R3</th>
<th>The periods of the logically synchronous threads, including the environment input synchronizers must satisfy the PALS clock period requirement (PALS system rule R3 of Section 2.2.3).</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall M_i \in M_S \ M_i.\text{Period} &gt; 2\epsilon + \max(M_i.\text{Deadline}, 2\epsilon - \mu^{\text{min}}) + \mu^{\text{max}} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PALS_R4</th>
<th>All external connections must be received first at an environment input synchronizer, unless the Required_Connection is set to false (PALS system rule R4 of Section 2.2.4). If the required connection is false, ( C_{\text{ext}}(Cn) ) returns ( \phi ) (‘null’).</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall Cn \in \text{Conns}<em>{\text{ext}} \ C</em>{\text{ext}}(Cn) \neq \phi \rightarrow C_{\text{ext}}(Cn).\text{Dst}.\text{PALS_Synchronizer_Type} = \text{Environment_Input_Synchronizer} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PALS_R5</th>
<th>The deadline must be greater than the latest output time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall M_i \in M_S \ M_i.\text{Deadline} &gt; \max(M_i.\text{PALS_Output_Time}) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PALS_R6</th>
<th>The source and destination of a delayed connection must correspond in the asynchronous model. That is, the source and destination of a delayed connection in the Synchronous AADL model are same as the source and destination of the mapped connection in the physically asynchronous model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall Cn \in \text{Conns}<em>{\text{sync}} \ T</em>{\text{thread}}(Cn.\text{Src}) = C_{\text{delayed}}(Cn).\text{Src} \land T_{\text{thread}}(Cn.\text{Dst}) = C_{\text{delayed}}(Cn).\text{Dst} )</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 lists the rules of this analysis. The first 4 rules (PALS_R1-PALS_R2) correspond to the PALS system rules (R1-R4) of Section 2.2. In this model, we also validate the derived value of PALS\_Output\_Time property to ensure that it is correctly computed and is obviously less than the
Deadline (PALS_R5). Similarly, other transformation specific properties are validated for sanity check. For example, a delayed connection of the Synchronous AADL model is correctly mapped between the same source and destination in the asynchronous model (PALS_R6).

We have developed this analysis framework in Open Source AADL Tool Environment (OSATE) [30]. The tool traverses the AADL component hierarchy and selects the relevant PALS components and connections for each logical synchronization group based the declared values of the properties, PALS\_Properties::PALS\_Id, PALS\_Properties::PALS\_Connection\_Id. We then apply the rules of Table 4.1 to validate the pattern requirements. Figure 4.2 gives a snapshot of the GUI of this tool.

![Figure 4.2: A snapshot of the PALS design tool. The GUI has three segments: (Segment 1) the thread and connection configuration of the PALS system, (Segment 2) user buttons for analysis and code generation of Section 4.4, and (Segment 3) results of analysis and code generation.](image)

**Discussion.** A complete PALS system analysis may be divided into two procedures: 1) analysis of the individual thread and connection properties, and 2) analysis of the PALS timing properties. The proposed architectural modeling and analysis of this section is only relevant for the second procedure. We assume pre-computed numerical values for the thread and connection properties, such as period, latency, and output time. For the first procedure, we can compute these properties by applying worst-case timing analysis on a system model with relevant hardware and software components. For example, the connection property, Latency declares the message transmission
delay between two AADL threads or devices. In practice, a single property as **Latency** abstracts much of the implementation details of an architecture since the precise computation of the latency requires us to model network topology, message flows, queuing delay, network devices, etc. One can use an extended architectural model to define all these relevant components and apply the theories of end-to-end delay analysis, e.g. network calculus [31] to compute the latency. There are research tools that perform these timing analysis. For example, ASIIST [32] is a tool for performing schedulability and bus analysis of AADL models. During our research, we have used ASIIST to measure end-to-end delays, worst-case response and output time. Once the thread and connection properties are measured, they can be used in our proposed analysis framework.

### 4.4 Auto-Generation of Synchronous Model from PALS Model

In this work, we also support a reverse transformation process that generates a Synchronous AADL model from the PALS model. The objective is to support a reverse verification flow, especially in legacy systems. In legacy systems, designers may apply the PALS pattern to simplify some critical distributed interactions and verify the requirements. However, it may not be always possible to start a design phase with the globally synchronous model due to the size and complexity of the legacy system architecture. In particular, the hierarchical model structure of AADL requires significant overhead to incorporate the globally synchronous model into the legacy system.

The alternative approach is to refactor the legacy components of the physically asynchronous model in-place by directly using a logically synchronous design. The basic principle is based on the same bisimulation property of the PALS pattern. As long as the PALS pattern constraints are satisfied, we can refactor and verify a subset of the physically asynchronous model as the globally synchronous model.

We have developed a tool in OSATE to generate the Synchronous AADL model from the PALS model. We generate the Synchronous AADL model for each logical synchronization group of threads and connections that define a common value for the `PALS_Properties::PALS_Id` and `PALS_Properties::PALS_Connection_Id` properties. Let these selected threads and connections are `Threads'async` and `Conns'async`, respectively. The model generation works in the following way:

- We define a single process with these threads and connections and make this process a sub-component in a top-level system component `S`.\(^1\)

\(^1\)Formal verification of the Synchronous AADL threads is not dependent on the process allocation of the threads.
• For each thread in $M \in Threads'_{async}$, we define the AADL properties relevant to the Synchronous AADL specification. For example, we define two properties for these threads: $\text{SynchAADL::IsDeterministic} \Rightarrow \text{true}$ and $\text{SynchAADL::IsEnvironment} \Rightarrow \text{false}$. We also define the period of these threads, $\text{Period} \Rightarrow 1\text{ms}$. The exact value of the period is also not relevant, but they have to same.

• At the top-level system $S$ surrounding the configuration, we define two properties, e.g. $\text{SynchAADL::Synchronous} \Rightarrow \text{true}$ and $\text{SynchAADL::SynchPeriod} \Rightarrow 1\text{ms}$.

• For the data connections between two threads in $Threads'_{async}$, we define a property, $\text{Timing} \Rightarrow \text{delayed}$.

• For other incoming connections to a thread, $M \in Threads'_{async}$, we define an environment thread called $\text{Environment}$. We define the property, $\text{Timing} \Rightarrow \text{immediate}$ for each connection from $\text{Environment}$ to $M$.

There are however some manual modifications that must be done prior to the formal verification. Designers have to specify the constraints on the environment inputs in $\text{Environment}$. They also have to define the verification properties based on the auto-generated model.

4.5 PALS Middleware Specification

In Section 4.1, we use simple AADL constructs to map the globally synchronous model to the logically synchronous model. In that specification, we abstract many of the implementation details. Instead, we use high-level AADL properties, e.g. $\text{PALS\_Id}$, $\text{PALS\_Period}$, $\text{PALS\_Output\_Time}$, $\text{PALS\_Synchronizer\_Type}$, etc. to declare the scope of the logically synchronous interactions and have a generic analysis for different implementations. In some sense, one can view this architectural model as a contract for subsequent model refinements and extensions. This specification must be extended with implementation details and conform to the design contract to guarantee the logical synchronization.

In this section, we discuss an extension of the architectural specification of Section 4.1 and implement the PALS pattern as a middleware service. This implementation has two parts: 1) globally synchronous computations, and 2) PALS clock events and message communications. Since the
PALS clock events and message communication services are generic and do not depend on the application logic, we implement them in the middleware. In addition to validating the timing constraints, we also validate some structural properties of the middleware components. The combined analysis ensures the logical synchronization in this implementation.

The contributions of this implementation model first appeared in [23]. The original work was done in AADLv1 [33]. We now support this model in AADLv2 [24]. We note that this implementation model does not handle any failure. We discuss our recent work on the PALS middleware in Chapter 6, which handles task failure in the middleware.

![Figure 4.3: Transformation for the PALS middleware specification.](image)

### 4.5.1 AADL Model Definition

In this section, we describe the main components of this implementation model in the context of the active-standby system. Since the middleware components are common for all logically synchronous computations, we only show the implementation model of Side1 in Figure 4.3. In this model, a logically synchronous thread (e.g. Side1Thread in Side1) is transformed into an AADL thread group (e.g. PALSSide1ThreadGroup in Side1) encapsulating the middleware components for Side1Thread. This newly formed thread group has the same interface as the original computation thread.

**PALS event generation.** According to the PALS requirements, the logically synchronous computations are performed at the PALS clock event. The outputs are also delivered after a certain interval to avoid causality violation. In this middleware specification, the PALS timing constraints
requirements are enforced by two events: \textit{PALS clock event} (\texttt{palsClockEvent}) and \textit{PALS output event} (\texttt{palsOutputEvent}).

We use an AADL thread, called \texttt{PalsEventGenerator} to output these events. This thread is dispatched periodically at each PALS clock period. In the PALS system, it dispatches at local clock times $jT$ for all $j \in N$ and the PALS clock period $T$. According to the pattern rules, \texttt{PalsEventGenerator} outputs \texttt{palsClockEvent} immediately after its dispatch at the local clock time $jT$. On the other hand, it outputs \texttt{palsOutputEvent} at $t_c(jT + H)$ where $H = \max(2\epsilon - \mu^{\min}, 0)$.

In order to guarantee that the PALS clock events are generated within a $2\epsilon$ interval across all nodes, we assume that each node implements a clock synchronizer process to synchronize the local clock. The clock synchronizer is required component for the PALS architecture. However, it is not a part of the middleware specification. It can be instantiated from a fault-tolerant clock synchronization algorithm [34, 35]. (See Section 6.5 for more details.)

\textbf{Globally synchronous computation.} In the logically synchronous model of Section 4.1, we model each computation as a periodic thread such as \texttt{Side1Thread} for Side1 in Figure 4.3a. In the PALS implementation model, we use the same computation. However, instead of having an explicit property for the periodic dispatch, we model the thread as an aperiodic thread and trigger its computations by using the PALS clock event \texttt{palsClockEvent} (Figure 4.3b). AADLv1 defines a predeclared input event port, called \texttt{Dispatch}, for each thread. One can use this event port to control the execution of an aperiodic thread. AADLv2 now has removed this option. It now requires an explicit declaration of the input event port(s) that can dispatch an aperiodic thread. In order to support the implementation model of [23] in AADLv2, we declare the \texttt{Dispatch} port explicitly in a computation thread. This input port is not used in the verification of the Synchronous AADL model. We define the property \texttt{Required\_Connection} $\Rightarrow$ \texttt{false} for this new port in the Synchronous AADL model.

\textbf{PALS message communication.} In order to enforce the \textit{PALS causality constraint}, outputs of the computation threads are delivered to an output delay thread. For example, \texttt{SideThread} forwards its outputs to the output delay thread, \texttt{Side1OutputDelayThread}. This thread holds the outputs until it is safe to send the output to the network. We model this thread as an AADL aperiodic thread.
The declaration for message communication through Side1OutputDelayThread is simple. In AADL, each thread completion is signaled by an event on a predeclared event port, Complete. We use the Complete event of the computation thread to notify the completion to the output delay thread. The Complete event from the computation thread, such as Side1Thread, is received at the palsComplete port of the output delay thread. The output delay thread also declares another input event port, called palsOutputEvent. This port is connected with the palsOutputEvent port of PalsEventGenerator.

In the output delay thread, the output is delivered only after both palsOutputEvent and palsComplete have arrived. In the AADL Behavior Annex, we use a dispatch trigger condition on these input event ports, given as (on dispatch palsOutputEvent and palsCompleteEvent). The following AADL code snippet shows the AADL declaration for the output delay thread in Side1.

```aadl
thread Side1OutputDelayThread
  features
    palsOutputEvent : in event port;
    palsCompleteEvent : in event port;
    side1ActiveSide_in : in data port Base_Types::Integer;
    side1ActiveSide_out : out data port Base_Types::Integer;
end Side1OutputDelayThread;

thread implementation Side1OutputDelayThread.impl
  properties
    Dispatch_Protocol => Aperiodic;
  annex behavior_specification {**
    states
      s1 : initial complete state;
    transitions
      s1-[on dispatch palsOutputEvent and palsCompleteEvent ]=>s1 {
        side1ActiveSide_out := side1ActiveSide_in
      };
    **};
end Side1OutputDelayThread.impl;

Event propagation. Figure 4.4 shows the event propagation in this implementation model. In this case, once the Side1Thread completes its execution, its Complete event propagates to the Side1OutputDelayThread, which then transmits the output messages to the network. The PALS clock period $T$ is sufficiently large so that the output messages reach Side2 before the next PALS clock period or equivalently, the event palsClockEvent. In this way, we can have a logically equivalent implementation as the globally synchronous model.
4.5.2 Structural Constraints

In this section, we summarize the structural constraints that are necessary for a correct implementation of the PALS middleware. If any of the constraints are violated, counter examples will exist to show that the implementation does not satisfy the synchronous model semantics.

The structural constraints of this middleware specification are as follows:

**PALSMid_R1.** The logically synchronous threads must dispatch only at the PALS clock event, `palsClockEvent`.

**PALSMid_R2.** The middleware enforces the PALS causality constraint by using the output delay thread, which is dispatched by the events `palsOutputEvent` and `palsComplete`.

**PALSMid_R3.** Except for the environment input synchronizer, a logically synchronous thread must interact with other components according to the PALS rules. This can be ensured by checking whether inputs to a logically synchronous thread arrive from an output delay thread.\(^2\)

**PALSMid_R4.** The outputs of a logically synchronous thread must go through the output delay thread.

The first two constraints, although simple, are important since we must make sure that components are asynchronously dispatched on the desired events. On the other hand, the last two constraints are dual constraints to require that all communication between two logically synchronous

\(^2\)We exclude the environment input synchronizer since environment inputs arrive directly from external components.
threads go through an output delay thread. In Figure 4.5, we consider two logically synchronous threads $C_1$ and $C_2$, and $C_1$’s output delay thread $C_{1out}$. If the message communication can occur directly and the clock skew is larger than the end-to-end delay, the PALS causality constraint will be violated. The message line labeled “without $C_{1out}$” in Figure 4.5 shows how this error can occur with direct communication. The output from $C_1$ during period $j$ will arrive early at $C_2$. $C_2$ will receive the output message in the same period $j$. This violates the logically synchronous execution semantics. The message line labeled “with $C_{1out}$” happens in the correct implementation in which the logically synchronous interactions goes through the output delay thread.

Note that there are other constraints that must also be satisfied. For example, $\text{palsClockEvent}$ in different nodes must be synchronized within $\epsilon$ time of the ideal PALS clock event at the global time $jT$. Such correctness can be ensured by verifying the correctness of the clock synchronizer.
The original PALS system has several limitations. It supports only a single rate for the distributed computations. The communication pattern is also restrictive as it allows only one message to be sent between 2 nodes in a PALS clock period. We have extended the PALS pattern to support the logical synchronization of multi-rate distributed computations. The extended PALS system is called *multi-rate PALS system*. In this pattern, application tasks execute at different rates and more than one message transfer is possible per PALS clock period. Applications synchronize the computations based on these messages.

In the multi-rate PALS pattern, a component can be logically synchronized with other components in more than one instantiation of this pattern at different synchronization periods. This is possible by forming separate logical synchronization groups. The composition of these instantiations allows engineers to achieve certain system-level properties, such as distributed consistency and distributed coordination, at the computations of the participating logical synchronization groups. This chapter extends the static analysis framework of the original PALS pattern to validate the integration of these synchronization groups. This chapter gives a detailed description of the pattern and the static analysis in AADL.

The materials of this chapter are based on our paper published at ICCPS 2012 [26]. In this thesis, we extend the multi-rate PALS pattern and unify with our specification of Section 4.1.

5.1 Main Concepts

In this section, we describe the main concepts of the logical synchronization of multi-rate distributed computations using an example system.
5.1.1 Case Study

For illustration purpose, let us consider an example of a hierarchical avionics control system. Multiple devices of this system, such as aileron and rudder, must be coordinated to turn an aircraft in real-time. Ailerons, attached to the left and right wings of an aircraft, coordinate with each other to roll an aircraft about the longitudinal axis by changing the lift on two wings. Since these ailerons move in different directions (upward or downward) to create a differential lift on the wings, they also cause a difference in the drag on the wings. This unwanted side effect, commonly known as adverse-yaw, produces a yawing motion in a direction opposite to the desired roll. One of the commonly applied techniques to counteract this undesired yawing motion is to use the rudder attached to the vertical stabilizer of the aircraft. Proper, synchronized coordination of both ailerons and the rudder at the right speed is important for the safety of the aircraft. Otherwise, improper turn of the rudder or the ailerons may result in undesired and dangerous sideways movement, known as side-slip.

The coordination of these control surfaces is accomplished by a fault-tolerant, hierarchical control system in which replicated supervisory controllers are responsible for coordinating the setpoints of the position, velocity of ailerons and rudder at a desired speed based on the flight mode. The local servo controllers of each control surface, which are also replicated, use the setpoint commands, compute local tracking errors with respect to the setpoints, and generate actuator commands at the acceptable rate for the devices. Here, we assume that the ailerons are controlled at 66.67Hz (15ms) and the rudder is controlled at 50Hz (20ms)\(^1\). For simplicity, we only consider an active-standby replication for the rudder control, where two servo controllers execute at the same rate. While both controllers receive the sensor data and supervisory commands, only the active controller sends the control command to the rudder.

Theoretically, the adjustment of these setpoints need not be synchronous since each of the elements under control is an analog device. However, asynchronous local actions increase coordination errors and are undesirable to prevent inconsistency. The proposed pattern can be applied to guarantee a logically synchronous coordination of these devices and prevent any inconsistency. The system would operate in the same way as it would do in a globally synchronous system (with zero clock skew).

\(^1\)In a commercial aircraft, the ailerons are controlled at 30-100 Hz, and the rudder is controlled at 30-50 Hz.
5.1.2 Pattern Features

There are two main features of this pattern: the logical synchronization period and the synchronization interface. These features differentiate the multi-rate PALS system from the single-rate PALS system.

**Logical synchronization period (i.e. PALS clock period):**

In a globally synchronous system with multi-rate distributed computations, the discrete state updates can happen synchronously only at the hyper-period boundaries, i.e. at an interval equal to the LCM (least common multiple) of the periods. This is unavoidable in a synchronous design since there is no simple scheduling solution to change the setpoints at the same time with a smaller period. A smaller synchronization period may also potentially result in asynchronous actions and potentially affect the system safety. To preserve the same synchronous semantics in a multi-rate PALS system, the PALS clock period is also set to the hyper-period. In this example, the rudder and aileron servo controllers receive the setpoint updates at a period of 60ms. The supervisory controller itself may execute at a faster or slower rate. However, if it needs to receive synchronous updates of the status of the device controllers, it can do so at 60ms.

*Note:* Harmonic rates have been traditionally favored for hierarchical control in industrial systems as they simplify the scheduling. However, the rates offered in such design may not be the best from a control perspective (considering the difference in the physical dynamics of the devices). On the other hand, picking locally optimal control periods may result in a very long LCM and slow the supervisory control. Therefore, the trade-off between local optimized control computations and longer supervisory control period needs to be considered when designing this hierarchical control system. A key benefit of using the proposed pattern is that it provides the simplicity of the synchronous design and does not require the devices to operate in a strictly harmonic rate. The devices can be controlled in a non-harmonic rate in a hierarchical control system since there are no direct communications between them. Thus, engineers can address this trade-off and explore an extended design space with this approach.
Synchronization interface:

In order to ensure the logical synchronization at a slower rate than the actual task rate, the pattern defines a synchronization interface, called multi-rate synchronizer for each input component\(^2\).

The multi-rate synchronizer has two roles in this pattern. Firstly, it ensures that the messages generated from the PALS clock period \((j-1)\) are observed at the receivers only during the PALS clock period \(j\). For example, Figure 5.1 shows the logically synchronous execution of the aileron servo controllers, the rudder servo controllers, and their supervisory controller. In this figure, the synchronizers at the aileron servo controllers and the rudder servo controllers gather synchronous updates of the setpoint commands only at the 60ms period boundary. Since there are many executions of a servo controller in each PALS clock period, the same setpoint is used during these executions. In this example, there are 4 executions of the rudder servo controllers and 1 execution of the supervisory controller in each PALS clock period. Suppose that the setpoint generated by the supervisory controller in its execution period \(p\) for the rudder is given by \(Supervisor.Setpoint^{out}_R(p)\) and the setpoint used by the rudder controller in its execution period \(q\) is given by \(Rudder_i.Setpoint^{in}_R(q), i = 1, 2\). The logically synchronous supervisory control of the rudder controllers can then be shown by

\[
Rudder_1.Setpoint^{in}_R(4j+k) = Rudder_2.Setpoint^{in}_R(4j+k) = Supervisor.Setpoint^{out}_R(j-1), k = 0..3
\]

\(^2\)In this chapter, we liberally use the term ‘synchronizer’ to refer to the ‘multi-rate synchronizer’. We want to note here that the multi-rate synchronizer is different from the environment input and output synchronizers. We will shortly explain the role of environment input and output synchronizer in the context of the multi-rate PALS system. Similar to the PALS system, environment input and output synchronizer are also used to manage external input and output events.
for each PALS clock period $j$. (It can be similarly shown for the aileron controllers.) In the physically asynchronous system, the system clock of each node has a bounded clock skew of $\epsilon$. The pattern abstracts the impact of the clock synchronization error and produces an equivalent execution, which happens within a $2\epsilon$ interval in global time.

Secondly, the synchronizer can be configured to deliver a unique input to the application such as the last received message, a function on the messages received in a PALS clock period. For example, the aileron and the rudder controllers send their status to the supervisory controller. These responses flow upward in the hierarchy as shown in Figure 5.2. It shows that the status at each PALS clock period $j$ is propagated to the supervisory controller in the same PALS clock period. The supervisory controller will consume this status update in the next PALS clock period. Based on these inputs, the supervisory controller may take necessary actions to coordinate these devices properly. While there are many executions of each control application during a PALS clock period, only the update from the last execution matters for the coordination. We show this communication with a solid line in the figure. However, the status from previous executions (as shown by the dashed lines) may be relevant depending on the application requirements such as debugging. If they are transmitted, then the synchronization interface of the supervisory controller may be responsible for delivering the correct input. In the current work, the multi-rate PALS pattern assumes that outputs from other executions are also delivered, but the multi-rate synchronizer filters the outputs to deliver only the last execution’s output.
Composition of different instances:

The multi-rate PALS pattern also allows designers to form logical synchronization groups where a component may participate in different groups. Thus, the components have the flexibility to receive their messages logical synchronously at different rates. For example, in addition to the multi-rate PALS synchronization for supervisory control, the replicated rudder servo controllers participate in another logical synchronization group with the sensors and the actuator. In this instantiation, these servo controllers receive the sensor data and perform discrete mode changes, such as changing the mode to standby upon the failure of the active controller, logically synchronously. This instantiation for the rudder servo control is a special case of this pattern with each task operating at a period of 20ms. Hence, the PALS clock period is also equal to 20ms. In this case, the logically synchronous interaction is simple. For the rudder sensor data ($RSD$), the pattern guarantees that $\text{Rudder}_1.RSD^{\text{in}}(j) = \text{Rudder}_2.RSD^{\text{in}}(j) = \text{Sensor}.RSD^{\text{out}}(j - 1)$ in PALS clock period $j$.

The pattern greatly simplifies the system verification in the pattern composition. In this example, we can reuse the pattern guarantee of logical synchronization with respect to the input data such as rudder sensor data ($RSD$) and supervisory setpoint command ($\text{Setpoint}_R$). Based on the logical synchronization property, the servo controllers operate consistently by receiving identical inputs, despite the differences in the rates. The only overhead for validating this system-level property of consistency is that these instantiations indeed follow the pattern requirements. We discuss these requirements in Section 5.4.

5.2 Pattern Assumptions and Guarantees

The pattern is applied to a group of periodic distributed tasks, $\{M_1, ..., M_N\}$. They execute at a period of $T_1, ..., T_N$ respectively. The hyper-period, denoted by $T_{hp}$, is equal to the LCM of these periods. In this section, we summarize the pattern’s assumptions for these tasks. We also prove the logical synchronization guarantee based on these assumptions. Later in Section 5.3, we describe how a developer can use this pattern in AADL.

5.2.1 Assumptions

The assumptions of the pattern are classified into three categories: system context, timing, and external interface constraints.
System context:

The multi-rate PALS pattern is applicable to hard real-time systems that satisfy the requirements of monotonic lock clock, bounded clock skew, bounded response time, and bounded message transmission delay. We however consider two adjustments to the original PALS system parameters for the response time and the message transmission delay. In the PALS system as described in Chapter 2.1, we assume a system-wide maximum and minimum values for all response times and network transmission delays, given as \((\alpha_{\text{min}}, \alpha_{\text{max}})\) and \((\mu_{\text{min}}, \mu_{\text{max}})\), respectively. In the single-rate PALS system, we use these values to derive the bound on the task periods. Since the tasks in the multi-rate PALS system execute at different rates, using system-wide bounds may be inefficient. We therefore use task-specific bounds for the response time and the message transmission delay.

In summary, this pattern makes following assumptions on the system model:

- The maximum skew of each local clock \( c_i \) with respect to the global time is \( \epsilon \).
- A task \( M_i \) completes its execution in bounded time. The minimum and maximum response time are \( \alpha_{i\text{min}} \) and \( \alpha_{i\text{max}} \), respectively.
- Messages from a task \( M_i \) are reliably delivered to their destinations with a latency of \( \mu_i \), where \( \mu_{i\text{min}} \leq \mu_i \leq \mu_{i\text{max}} \).

Timing constraints:

The following constraints relating system parameters of each computation must be also satisfied. We show in next section that these constraints are required to satisfy the requirement that messages generated during the PALS clock period \( j - 1 \) are consumed by their destination tasks in the PALS clock period \( j \).

- **Task period constraint.** The period of a task \( M_i \) gives the upper bound on the worst-case end-to-end delay from \( M_i \). A message must not be sent after \((T_i - \mu_{i\text{max}} - 2\epsilon)\) such that the destination tasks receive it before the next dispatch of the source task.

\[
e_{\text{out}i}^{\text{max}} < \alpha_{i\text{max}} < T_i - \mu_{i\text{max}} - 2\epsilon. \tag{5.1}
\]

\( e_{\text{out}i}^{\text{max}} \) denotes latest time when the task \( M_i \) transmits a message. Equation 5.1 is similar to Equation 2.1 except that we now consider task-specific parameters.
• **Causality constraint.** In order to account for the clock skews, messages should not be delivered before a specific interval given as

\[
e_{\text{out}}_{\text{min}}^i > \max(2\epsilon - \mu_{\text{min}}^i, 0).
\] (5.2)

Here, \(e_{\text{out}}_{\text{min}}^i\) is the earliest time when the task \(M_i\) transmits a message.

**Environment input constraints:**

The last set of assumptions is associated with environment or external inputs from any component outside the logical synchronization group. The pattern assumes that the components consume these environment inputs, such as a user input that changes the global system mode, in the same PALS clock period.

5.2.2 Guarantees

As illustrated in Chapter 5.1, the pattern guarantees logical synchronization between multi-rate asynchronous computations at a period of \(T_{hp}\). In the multi-rate PALS pattern, the PALS clock period is equal to \(T_{hp}\).

Suppose that \(M_a\) is a periodic task of period \(T_a\). \(M_a\) transmits its output messages to other tasks, \(M_b\) and \(M_c\) of period \(T_b\) and \(T_c\), respectively. There are \(n_a = T_{hp}/T_a\), \(n_b = T_{hp}/T_b\) and \(n_c = T_{hp}/T_c\) executions of \(M_a\), \(M_b\) and \(M_c\) during a PALS clock period. The pattern guarantees that \(M_b\) and \(M_c\) receive all \(n_a = T_{hp}/T_a\) messages from \(M_a\) generated during the PALS clock period \(j - 1\). The pattern filters these received messages and delivers the selected messages identically to \(M_b\) and \(M_c\) during the PALS clock period \(j\). If the last received message is selected, the pattern ensures that

\[
M_{b.in}(j.n_a + k_b) = M_{c.in}(j.n_c + k_c) = M_{a.out}(j.n_a - 1),
\]

where \(k_b = 0 \ldots n_b - 1, k_c = 0 \ldots n_c - 1\). Here \(M_{a.out}(i'), M_{b.in}(j'), \) and \(M_{c.in}(k')\) correspond to the input and output port data of the corresponding tasks in their execution period \(i', j', \) and \(k'\) respectively.
5.2.3 Verification of the Multi-Rate PALS Pattern

This section describes the timing model of this pattern. We use this model to prove the logical synchronization guarantee based on the pattern’s assumptions.

Multi-rate PALS timing model:

Each node in the pattern has two components involved in the pattern instantiation: $M_i$ (input task) and $M_{i,syn}$ (multi-rate synchronizer). $M_i$ and $M_{i,syn}$ are periodic computations with period $T_i$ and $T_{hp} = n_i T_i$, respectively.

We assume that the PALS clock period $j$ at each node begins at the local clock time $jT_{hp}$. Let this happen in the global time $t_{j,0}^i$. Since the maximum clock skew is $\epsilon$, $t_{j,0}^i$ is in between the global time interval $[jT_{hp} - \epsilon, jT_{hp} + \epsilon]$.

Both $M_i$ and $M_{i,syn}$ execute on the same processor. Thus, their executions are synchronized based on the local clock time. The $j^{th}$ execution of $M_{i,syn}$ coincides with the $jn_{th}^{th}$ execution of $M_i$. The other $n_i - 1$ executions of $M_i$ happen at local clock time $jT_{hp} + kT_i$ or in global time at $t_{j,k}^i \in [jT_{hp} + kT_i - \epsilon, jT_{hp} + kT_i + \epsilon], k = 1 \ldots n_i - 1$. In this design, $M_{i,syn}$ has a higher priority than $M_i$ when their dispatch events coincide at local clock time $jT_{hp}$.

![Figure 5.3: Multi-rate PALS timeline for $n_i = 2$](image-url)
Figure 5.3 shows a timeline of the computation and communication in a multi-rate PALS pattern instance. Let the task $M_i$ transmit an output message after an interval of $e_{out}^i_{j,k}$ during its $k^{th}$ execution in the PALS clock period $j$. If the message transmission delay is $\mu_i$, this message is expected to arrive after an interval of $e_{out}^i_{j,k} + \mu_i$ at the receiving side. Here, $e_{out}^i_{min} \leq e_{out}^i_{j,k} \leq e_{out}^i_{max}$ and $\mu_i^{min} \leq \mu_i \leq \mu_i^{max}$.

**Lemma 1:** In a physically asynchronous system, when a task $M_i$ sends its messages to the multi-rate synchronizer $M_{r,syn}$ of the receiving component $M_r$, the multi-rate synchronizer receives exactly $n_i = T_{hp}/T_i$ messages in each PALS clock period $j$ if the timing constraints $e_{out}^i_{max} < \alpha_{i}^{max} < T_i - 2\epsilon - \mu_i^{max}$ (Equation 5.1) and $e_{out}^i_{min} > \max(2\epsilon - \mu_i^{min}, 0)$ (Equation 5.2) are satisfied. In other words, messages generated during the PALS clock period $j$ are received by the receiving node in the same PALS clock period $j$ if these timing constraints are satisfied.

**Proof:** There are exactly $n_i$ executions of $M_i$ in each PALS clock period. We prove this lemma by showing that the 1$^{st}$ and $n_i^{th}$ messages generated during the PALS clock period $j$ are indeed received in the same PALS clock period at the receiving node.

The first execution of $M_i$ during the PALS clock period $j$ happens at the local clock time $jT_{hp}$. If the task’s output time is $e_{out}^i_{j,0}$ and the message transmission delay is $\mu_i$, the message arrives at the receiving node in the global time interval $[jT_{hp} + e_{out}^i_{j,k} + \mu_i - \epsilon, jT_{hp} + e_{out}^i_{j,k} + \mu_i + \epsilon]$. Given the minimum latency $\mu_i^{min}$, the earliest message arrival time is $jT_{hp} + e_{out}^i_{j,k} + \mu_i^{min} - \epsilon$. At the receiving node $M_r$, the multi-rate synchronizer $M_{r,syn}$ dispatches during the global time interval $[jT_{hp} - \epsilon, jT_{hp} + \epsilon]$. Since $e_{out}^i_{j,k} \geq e_{out}^i_{min} > \max(2\epsilon - \mu_i^{min}, 0)$,

$$jT_{hp} + e_{out}^i_{j,k} + \mu_i^{min} - \epsilon > jT_{hp} + \epsilon.$$

It implies that this message is indeed received in PALS clock period $j$, which happens after the dispatch of $M_{r,syn}$ in this period.

The $n_i^{th}$ execution in the PALS clock period $j$ occurs at $t_{j,n_i-1}^i$. In global time, this can happen as late as at $jT_{hp} + (n_i - 1)T_i + \epsilon$ in global time. Since the maximum output time is $e_{out}^i_{max}$ and the maximum message transmission delay is $\mu_i^{max}$, the latest output arrival time is given by $t_{arr}$, where $t_{arr} \leq jT_{hp} + (n_i - 1)T_i + \epsilon + e_{out}^i_{max} + \mu_i^{max}$. The PALS clock period $j + 1$ in the receiving component may begin as early as at $(j + 1)T_{hp} - \epsilon = jT_{hp} + n_i T_i - \epsilon$. Given that $e_{out}^i_{max} < \alpha_{i}^{max}$
and $T_i > 2\epsilon + \alpha_i^{\max} + \mu_i^{\max}$, it is easy to show that

$$t_{arr} < jT_{hp} + n_iT_i - \epsilon = (j + 1)T_{hp}.$$ 

This implies that the $n_i^{th}$ message is also received in the PALS clock period $j$.

The proof immediately follows since outputs of the remaining $(n_i - 2)$ executions are also received in FIFO order between the 1st and $n_i^{th}$ executions. ■

**Theorem 1:** The proposed pattern specification satisfies the same message interaction guarantee as the globally synchronous system.

**Proof:** The proof is simple. By generalizing Lemma 1 over all receiving nodes, the multi-rate synchronizers receive the same set of messages at a PALS clock period. The logic of the multi-rate synchronizers are same at the receiving nodes. Since the multi-rate synchronizer $M_{r,syn}$ executes always before the destination task $M_r$ and they select the same message, such as the last received message, the destination tasks $M_r$ apply the same input during its executions in a PALS clock period as it would do in a globally synchronous system. ■

5.3 Pattern Specification in AADL

The multi-rate PALS pattern also transforms an input system model to a new system model with guaranteed properties. In this section, we describe the AADL specification of the multi-rate PALS pattern. This specification extends the AADL specification of Section 4.1 for the model definition of multi-rate synchronizer and the composition of the multi-rate synchronizer and the corresponding distributed computation.

5.3.1 Pattern Parameters

We model the input distributed tasks $\{M_1, ..., M_N\}$ as AADL threads or thread groups\(^3\). Only a subset of the connections between these components are used in the multi-rate logical synchronization in an instance of this pattern. As input parameters of this transformation, these connections define a property, `PALS_Properties::PALS_Connection_Id=>‘GROUP_ID’`.\(^4\) Here, “GROUP_ID”

---

\(^3\) An AADL thread group gives the component abstraction for a logical organization of threads and other thread groups within a process.

\(^4\) PALS_Properties is an AADL property set defined for the PALS system. We describe the property set in Appendix B.
is the identifier of this logical synchronization group. These components also define an AADL
property, \texttt{PALS\_Properties::Computation=>Multi\_Rate\_Base\_Computation} to define the input
components of this transformation.

These input components must define the properties to declare period, deadline, output time, lat-
tency, and clock skew. In AADL, these parameters can be specified by using standard AADL prop-
erties: \texttt{Period}, \texttt{Deadline}, \texttt{PALS\_Properties::PALS\_Output\_Time}, \texttt{Latency}, and \texttt{Clock\_Jitter}.

The multi-rate PALS pattern extends these input components and defines a corresponding
output component $M_i'$ for an input component $M_i$. The output component defines a property
\texttt{PALS\_Properties::PALS\_Id=>'GROUP_ID'} after the instantiations. The \texttt{PALS\_Id} and \texttt{PALS\_Connection\_Id} properties together show the scope of the logical synchronous interactions in a
group of components and connections.

5.3.2 Multi-Rate Synchronizer

In order to guarantee the logical synchronization, the pattern attaches a \textit{multi-rate synchronizer}
$M_{i,syn}$ at each component $M_i$ that serves as a synchronization interface and manages only the input
data used in the multi-rate logical synchronization. It does not affect other inputs that are not
used in this instantiation.

We currently model the multi-rate synchronizer as an AADL thread component and bind it to
the same processor as $M_i$. We use a set of AADL properties to model the expected scheduling and
communication characteristics of the multi-rate synchronizer:

\begin{itemize}
  \item \texttt{PALS\_Properties::PALS\_Synchronizer\_Type}: The multi-rate synchronizer declares this prop-
        erty to distinguish itself as a multi-rate synchronizer thread. The value is \texttt{Multi\_Rate\_Synchronizer}.

  \item It declares the dispatch protocol and the period as \texttt{Dispatch\_Protocol=>Periodic} and \texttt{PALS\_Properties::PALS\_Period=}$T_{hp}$, respectively.

  \item \texttt{Priority}: As discussed in the example section, the thread priority of the multi-rate synchro-
        nizer is set to a value higher than that of $M_i$.

  \item \texttt{PALS\_Properties::PALS\_Output\_Time}: It is the interval during an execution when the
        multi-rate synchronizer transmits its output messages. As discussed in Chapter 4, this PALS
        property is a derived value from \texttt{Output\_Time} of the output ports.
\end{itemize}
• **PALS\_Properties::Multi\_Rate\_Synchronizer\_Operation**: The pattern defines the message selection criteria of the input data ports of the multi-rate synchronizer with this property. Currently, its value is set to *Last\_Message\_Only* to indicate that the multi-rate synchronizer only propagates the last message that it has received in an input port during a PALS clock period. It can be changed to model other alternatives, such as delivering a vector or a function of the received messages.

5.3.3 Composition of $M_i$ and $M_i,\text{syn}$

In order to facilitate the use of this component in subsequent pattern instantiations, the pattern forms a new AADL thread group $M'_i$ with the input component $M_i$ and the multi-rate synchronizer $M_i,\text{syn}$ as its subcomponents. This newly formed thread group has exactly the same input-output interface as the input component, $M_i$. The pattern defines the internal connections of this thread group. The input ports of $M'_i$ are connected to the input ports of $M_i,\text{syn}$ if they are relevant to the current pattern instantiation (identified by the property **PALS\_Properties::PALS\_Connection\_Id**), otherwise these input ports are connected with corresponding input ports of $M_i$. The outputs of $M_i$ are directly propagated to corresponding output ports of $M'_i$.

The pattern also annotates $M'_i$ with a number of AADL properties: **Period**, **Deadline**, **PALS\_Properties::PALS\_Output\_Time**, **Priority**, and **PALS\_Properties::PALS\_Id**. These properties capture the timing characteristics of the combined execution of these two components. We derive their values from the properties of $M_i$ and $M_i,\text{syn}$. In this case, **Period**, **Deadline**, **Priority** of $M'_i$ are set to the same values as $M_i$ since the main input component inside this new thread group does not change with this composition. However, the multi-rate synchronizer adds small computation overhead during the first execution of $M_i$ in a PALS clock period. We consider this overhead in the computation and output time. In this pattern, we primarily update the **PALS\_Properties::PALS\_Output\_Time**. In this composition, $M'_i,\text{PALS\_Output\_Time}$ is a time range, in which the minimum value of the range is equal to $min(M_i,\text{Output\_Time} + M_i,\text{syn},\text{Output\_Time})$ and the maximum value of the range is equal to $max(M_i,\text{Output\_Time} + M_i,\text{syn},\text{Output\_Time})$. Here $min(x)$ and $max(x)$ give the minimum and maximum value of a time range $x$, respectively.

In addition, $M'_i$ assigns two properties, **PALS\_Properties::PALS\_Id=‘GROUP\_ID’** and **PALS\_Properties::PALS\_Period=>T_{hp}**. Here, “GROUP\_ID” is the identifier of the logical synchronization group and $T_{hp}$ is the PALS clock period of this group.
5.3.4 Environment Input Synchronizer

The multi-rate PALS pattern extends the concept of \textit{environment input synchronizer} of the single-rate PALS system. The external component transmits its outputs asynchronously to an environment input synchronizer $M_{env}$. The environment input synchronizer then relays the external inputs to the input components of this pattern. Similar to the PALS pattern specification of Chapter 4, the environment input synchronizers define a property, \texttt{PALS\_Properties::PALS\_Synchronizer\_Type=>Environment\_Input\_Synchronizer}.

Unlike the single rate PALS system, the environment input synchronizer does not have to execute at the PALS clock period. It can execute at a faster rate in a multi-rate PALS system provided that the PALS clock period is perfectly divisible by its period. In this pattern, $M_{env}$ follows the timing constraints of Equation 5.1 and 5.2. With this solution, the external component can execute at any rate asynchronously with respect to the components of a given logical synchronization group.

5.3.5 Alternative Implementation

Some of the suggested implementations may have alternative modeling solution with equivalent results. For example, we have modeled the periodic processing of the multi-rate synchronizer as a thread in AADL. We choose the AADL thread representation as it clearly shows the timing relationship between the multi-rate synchronizer and the input component. Alternatively, we can model the multi-rate synchronizer as a subprogram that periodically executes at one of the dispatches of the input component. However, the exact timing relationship between the multi-rate synchronizer and the input component is not clearly expressed with a subprogram. We need to assume the timing relationship through additional AADL properties. In this case, we also have to consider the effect of the subprogram implementation in the pattern modeling and analysis.

5.3.6 Exemplar Model

Figure 5.4 gives simplified AADL diagrams of the pattern instantiations for rudder servo control synchronization and supervisory control synchronization. In this figure, we only show the distributed process elements of different subsystems and the threads inside them. We do not show the binding of these threads to the hardware components.

In the first synchronization in Figure 5.4a-c, two replicated rudder controller threads (RCT) receive
Servo control synchronization (PALS_Id=>'Rudder\_Control', PALS\_Period=>20ms).

(a) Input model.

(b) Multi-rate synchronizer (Syn) is added.

(c) Computations are grouped in a thread group.

Supervisory control synchronization (PALS_Id=>'Supervisory\_Control', PALS\_Period=>60ms).

(d) Input model.

(e) Multi-rate synchronizer (Syn) is added.

(f) Computations are grouped in a thread group.

Figure 5.4: Composition of multi-rate PALS pattern instances.
the sensor data from the rudder sensor thread (RST) and propagate their control commands to the actuator thread (RAT). They also exchange the heartbeat messages as part of the active-standby replication protocol. After the pattern is applied, multi-rate synchronizers, denoted by Syn, are added to the processes. These synchronizers affect only the input data that are relevant to this pattern instantiation. In this instantiation, the setpoint commands from the supervisory controllers are not directly involved. Therefore, the setpoint commands do not pass through the multi-rate synchronizer. After this, we create an AADL thread group component, such as RCT.Gr, at the servo controllers composing both RCT and Syn.

In the second synchronization in Figure 5.4d-f, we use the output model of the first instantiation as the input model for the supervisory control synchronization. In this case, two rudder controllers and an actuator controller receive the setpoint commands from the supervisor. The pattern instantiation follows the same rule as above without affecting the non-participating inputs of a component.

We provide an AADL code snippet of this hierarchical control system in Appendix D.

5.4 Compositional Analysis

We provide an analysis framework in OSATE to validate the assumptions and the structural specification of the multi-rate PALS pattern. This analysis framework extends the analysis of the single-rate PALS pattern. In the composition of many instantiations of the multi-rate PALS pattern, this analysis guarantees that the design does not have any error when we extend the output model of the prior instantiations as the input to next instantiation.

5.4.1 Analysis Procedure

Table 5.1 lists the main analysis rules that validates the structural specification, the timing and external input assumptions for different multi-rate PALS instantiations.

Explanation of the symbols: We consider that the top-level configuration, \( G = (Comp, Conns) \) consists of the set of all thread and thread group components, \( Comp = \{M_i\} \) and the set of all port connections, \( Conns = \bigcup_{i,j} Conns(M_i, M_j) \), where \( Conns(M_i, M_j) \) is the set of all port connections from component \( M_i \) to \( M_j \). The set of all port connections to a component \( M_j \) is also given by \( Conns(*, M_j) \). For each connection \( Cn \), we assume that \( Cn.DstPort \) provides information on
the destination port. We use some data enumerating functions. For example, \( PALS\_Id(G) \) gives all synchronization group identifiers in \( G \). \( PALS\_Period(id) \) is equal to the PALS clock period of the logical synchronization of \( id \). \( Comp^F(G, id) \) gives the set of components \( M_i \) that define \( PALS\_Id\Rightarrow id \). If \( id = \ast \), \( Comp^F(G, \ast) \) returns the set of components that define the \( PALS\_Id \) property. We use the term \( M_i.Property \) to give the values of an AADL property ‘Property’ of a component \( M_i \). As shown earlier, a thread group \( M_i \) is formed from a computation component and its multi-rate synchronizer. Here, we denote these subcomponents as \( M_i.Sync \) and \( M_i.Base \), respectively. In this system model, \( M_i, M_i.Sync \) and \( M_i.Base \) are members of the set \( Comp \).

**Pattern specification rules:** The rules MPALS_R1-MPALS_R7 are related to the scheduling and communication characteristics of the pattern instantiated multi-rate synchronizer and the newly formed thread group. For example, MPALS_R5-MPALS_R6 guarantee the condition that all messages related to a multi-rate PALS pattern instance flow through the multi-rate synchronizer. We analyze the data flow between the components and detect if the multi-rate synchronizer is bypassed. This is important since any violation of these constraints may potentially invalidate the logical synchronization guarantee. On the other hand, the rule MPALS_R7 shows that that if a connection belongs to a different logical synchronization group, then corresponding base component handles it. We eventually check if this connection is indeed managed by the appropriate multi-rate synchronizer of the target synchronization group.

**Timing assumptions:** The rules MPALS_R8 and MPALS_R9 describe the timing assumptions we have defined in Equation 5.1 and 5.2.

**Environment input assumptions:** The rules MPALS_R10 and MPALS_R11 validate the environment input assumptions. MPALS_R10 validates that environments inputs are originated from an environment input synchronizer and the environment input synchronizer follows the constraint of MPALS_R8 and MPALS_R9. MPALS_R11 shows that its outgoing connections contribute to a given logical synchronization group.
Sanity check of the multi-rate PALS specifications

**MPALS_R1.** At each PALS component $M_i$, PALS\_Period of $M_i$ must equal to Period of $M_i$.Sync.
\[
\forall id \in PALS\_Ids(G) \forall M_i \in \text{Comp}^F(G,id) \text{ PALS\_Period}(id) = (M_i.\text{Sync}).\text{Period}.
\]

**MPALS_R2.** At each PALS component $M_i$, Deadline of $M_i$ is set to Deadline of $M_i$.Base and Deadline of $M_i$.Base must be greater than that of $M_i$.Sync.
\[
\forall M_i \in \text{Comp}^F(G,\ast) \text{ M}_i.\text{Deadline} = (M_i.\text{Base}).\text{Deadline} \land (M_i.\text{Sync}).\text{Deadline} < (M_i.\text{Base}).\text{Deadline}.
\]

**MPALS_R3.** At each PALS component $M_i$, the minimum value of PALS\_Output\_Time of $M_i$ is set to the sum of the minimum value of PALS\_Output\_Time of $M_i$.Base and $M_i$.Sync. Similarly, the maximum value of PALS\_Output\_Time of $M_i$ is set to the sum of the maximum value of PALS\_Output\_Time of $M_i$.Base and $M_i$.Sync. Furthermore, the maximum value of PALS\_Output\_Time of $M_i$ must be smaller than the Deadline of $M_i$.
\[
\forall M_i \in \text{Comp}^F(G,\ast) \text{ min}(M_i.\text{PALS\_Output\_Time}) = \text{min}((M_i.\text{Base}).\text{PALS\_Output\_Time} + (M_i.\text{Sync}).\text{PALS\_Output\_Time}).
\]
\[
\forall M_i \in \text{Comp}^F(G,\ast) \text{ max}(M_i.\text{PALS\_Output\_Time}) = \text{max}((M_i.\text{Base}).\text{PALS\_Output\_Time} + (M_i.\text{Sync}).\text{PALS\_Output\_Time}).
\]
\[
\forall M_i \in \text{Comp}^F(G,\ast) \text{ max}(M_i.\text{PALS\_Output\_Time}) < M_i.\text{Deadline}
\]

**MPALS_R4.** $M_i$.Sync and $M_i$.Base must be be collocated and execute on the same processor core.
\[
\forall M_i \in \text{Comp}^F(G,\ast) (M_i.\text{Sync}).\text{Processor} = (M_i.\text{Base}).\text{Processor}.
\]

**MPALS_R5.** At each PALS component $M_i$, the connections received at the $M_i$.Sync must belong to the corresponding logical synchronization group. We validate this property by checking the PALS\_Connection\_Id property of the connections and the PALS\_Id property of $M_i$.
\[
\forall M_i \in \text{Comp}^F(G,\ast) \forall Cn \in \text{Conns}(\ast,M_i.\text{Sync}) \text{ Cn.\text{PALS\_Connection\_Id}} = M_i.\text{PALS\_Id}.
\]

**MPALS_R6.** At each PALS component $M_i$, the connections between the pair $(M_i, M_i$.Sync) must correspond to the connections between the pair $(M_i.\text{Sync}, M_i.\text{Base})$. We liberally use DstPort to show the equivalence between the ports of two components.
\[
\forall M_i \in \text{Comp}^F(G,\ast) \forall Cn \in \text{Conns}(\ast,M_i) \text{ Cn.\text{PALS\_Connection\_Id}} = M_i.\text{PALS\_Id} \rightarrow
(\exists Cn' \in \text{Conns}(M_i.\text{Sync},M_i.\text{Base}) \text{ Cn'.DstPort} = \text{Cn'.Observers} \land \exists Cn' \in \text{Conns}(M_i,M_i.\text{Sync}) \text{ Cn.DstPort} = \text{Cn'.DstPort}).
\]
At each PALS Component $M_i$, the connections to $M_i$ that have a different `PALS_Connection_Id` than its `PALS_Id` are directly passed to $M_i.Base$ instead of passing through the multi-rate synchronizer $M_i.Sync$.

$$\forall M_i \in Comp_F(G,*) \forall Cn \in Connns(*,M_i) \ Cn.PALS.Connection_Id \neq M_i.PALS.Id \rightarrow$$

$$\exists Cn' \in Connns(M_i,M_i.Base) Cn.DstPort = Cn'.DstPort.$$ 

**Timing assumptions**

**MPALS_R8.** The period of each PALS component $M_i$ must satisfy its period constraint, i.e.

$$\forall M_i \in Comp_F(G,*) M_i.Period > 2 \times G.Clock.Skew + \text{Max.Latency}_i + M_i.Deadline.$$ 

Here, $\text{Max.Latency}_i = \max(\{Cn.Latency \mid Cn \in Connns(M_i,Base,*))\}$. $G.Clock.Skew$ is the maximum clock skew in the configuration.

**MPALS_R9.** Each PALS component $M_i$ must satisfy the causality constraint.

$$\forall M_i \in Comp_F(G,*) \min(M_i.PALS.Output.Time) > \max(2 \times G.Clock.Skew - \text{Min.Latency}_i, 0).$$ 

Here, $\text{Min.Latency}_i = \min(\{Cn.Latency \mid Cn \in Connns(M_i,Base,*))\}$. 

**Environment input assumptions**

**MPALS_R10.** An environment input synchronizer must satisfy the constraints of MPALS_R8 and MPALS_R9 and `PALS_period` of an environment input synchronizer is divisible by its `Period`.

$$\forall M_i \in Comp_F(G,*) M_i.PALS_Synch.Type = Env.Input.Synch \rightarrow M_i.PALS.Period \% M_i.Period = 0$$

$$\land (M_i \text{ satisfies the predicates of MPALS_R8, MPALS_R9}).$$

**MPALS_R11.** All outgoing connections of an environment input synchronizer must have the same values for the `PALS_Connection_Id` property as its `PALS_Id`.

$$\forall M_i \in Comp_F(G,*) M_i.PALS_Synchronizer.Type = Environment.Input.Synchronizer \rightarrow$$

$$\forall Cn \in Connns(M_i,*) Cn.PALS.Connection_Id = M_i.PALS.Id.$$ 

Table 5.1: Multi-Rate PALS pattern rules.
CHAPTER 6
MIDDLEWARE DESIGN FOR PALS SYSTEM

In previous chapters, we have applied the PALS pattern in architecture models of hard real-time distributed systems. Users can use the PALS analysis framework to verify the correctness of the implementation models. In this thesis, we have further extended this research. With Dr. Cheolgi Kim, we have developed a distributed middleware, called PALSware, to enable robust implementation of the PALS computations in C++.

In PALSware, we address several practical challenges for guaranteeing the logical synchronization. First, the PALS pattern assumes that nodes are fail-stop. Existing safety-critical systems, such as avionics, support the fail-stop executions with redundant processor pairs [9, 36]. The outcomes of redundant processors are compared to detect a fault. However, the node-level fail-stop model does not guarantee the fail-stop semantics in distributed computing. For example, a node may send the same message sequentially to different nodes. If this node suddenly stops, only a subset of the receiving nodes may receive this message, which subsequently leads to inconsistent states. In this chapter, we discuss a fault-tolerant communication protocol to prevent this problem. We also use this protocol to guarantee atomicity in logically synchronous computations.

Second, the PALS pattern assumes bounded clock skew and bounded message transmission delay. Any violation of these timing assumptions may lead to inconsistent and unsafe operations. PALSware therefore enables run-time monitoring of the system parameters to detect a violation of the assumptions. For example, the middleware detects timing faults, such as unusually large clock skew, and converts them into the fail-stop model. Users can also extend the fault-management capabilities of the middleware and add application or architecture-specific fault managers.

The rest of this chapter gives an overview of the middleware architecture and the design considerations during its development. In Section 6.2, we discuss the basic PALS execution and communication model in PALSware. This execution and communication model is similar to the multi-rate PALS pattern of Chapter 5. In Section 6.3, we discuss a simple fault-tolerant commu-
communication protocol to guarantee consistent message communication. In Section 6.3, we extend the basic PALS communication model that guarantee atomicity in logically synchronous interactions. In Section 6.4, we discuss PALSware's solution to detect any violation of the timing assumptions. Later in Chapter 7, we present the experimental studies of different aspects of this middleware.

6.1 Middleware Architecture

Figure 6.1 shows the PALS system architecture with PALSware. It consists of three layers: infrastructure layer, middleware layer, and application layer. This layered architecture makes the middleware portable and extendable in different platforms. The application logic does not change as long as the pattern's assumptions are satisfied.

At the infrastructure layer, PALSware assumes a fault-tolerant clock synchronizer that enables high-precision synchronization of the distributed clocks [37, 38]. PALSware is not dependent on a specific clock synchronizer. It can support an off-the-shelf clock synchronizer, such as Precision Time Protocol (PTP) [39], as long as it satisfies the assumptions on the clocks. PALSware also assumes a fault-tolerant real-time network architecture, such as AFDX [40], which has redundant communication channels or sub-networks to ensure reliable transmission with bounded delay.

At the middleware layer, PALSware provides the services to execute the distributed tasks periodically based on the user-specified scheduling parameters such as period and priority. It also
provides the required communication services for logically synchronous interactions. In particular, task failures increase the system complexity. PALSware guarantees the logical synchronization even when a task fails in the middle of the computation.

At the application layer, application developers implement the application logic for the distributed tasks. Users can plug-in fault managers to detect a fault that may affect the logical synchronization and the application’s safety. For example, if the clock skew assumption is violated for some reason, the distributed tasks will not be logically synchronized within the $2\epsilon$ interval in global time. PALSware also supports fail-safe actions after the detection of a fault.

6.2 PALS Tasks and Communications in PALSware

PALSware assumes the task and communication model of the multi-rate PALS pattern. In this section, we give an overview of the implementations in PALSware.

6.2.1 Task Execution

In PALSware, each task $M_i$ executes periodically with a period of $T_i$. Following this model, a task $M_i$ dispatches at the local clock time $kT_i, \forall k \in \mathbb{N}$.

According to the multi-rate PALS pattern, the logically synchronous interactions happen at every hyper-period interval. The PALS clock period $T_{hp}$ is equal to the hyper-period or the least-common-multiple of the task periods.

PALSware has an abstract C++ class, called \textit{PALS_task}, for the execution of each task in this model. \textit{PALS_task} defines a real-time periodic thread in the user-space. For each task $M_i$, an application developer extends this class and defines the periodic logic of this task in the virtual function, called \textit{each_pals_period}. We use the scheduling parameter of the period and priority of this task as input variables to construct an instance of the \textit{PALS_task} class. For each task $M_i$, PALSware maintains a set of timestamps in this class:

- **\textit{dispatch_time}:** This is equal to the local clock time of the most recent dispatch event of $M_i$. If the current local clock time is in the interval $[kT_i, (k + 1)T_i)$, then \textit{dispatch_time} = $kT_i$, for some $k \in \mathbb{N}$.

- **\textit{PALS_base_time}:** This is equal to the local clock time of the most recent PALS clock event. If the current local clock time is in the interval $[jT_{hp}, (j + 1)T_{hp})$, then \textit{PALS_base_time} =
6.2.2 Task Startup

PALSware coordinates the start time of the logically synchronous tasks. In PALSware, a task executes its first logically synchronous computation at the beginning of a PALS clock period. Let the PALS clock period be \( T_{hp} \). In a multi-rate PALS pattern, \( T_{hp} = n_i T_i \), where \( n_i \geq 1 \). If the application binary is loaded at the local time interval \((jT_{hp}, (j + 1)T_{hp}]\), the task starts its logically synchronous execution at the local clock time \((j + 2)T_{hp}\). We wait for an extra \( T_{hp} \) interval, especially when the binary is loaded just before the local clock time \((j + 1)T_{hp}\). PALSware guarantees consistency at the cost of this initial delay. Otherwise, the receiving tasks would not always receive a fixed set of \( n_i \) messages in each PALS clock period according to the multi-rate PALS pattern.

6.2.3 Message Communication

Since there exists a non-zero clock skew in the actual system, we have to satisfy the causality constraint and ensure that a message is received in the same PALS clock period and consumed in the next PALS clock period. As shown in Section 2.2.2 and Section 5.2.1, each task must not deliver its messages too early to violate the PALS causality constraint. In these approaches, a task buffers its messages and uses an output timer to transmit the messages. The output timer has a timeout after an interval of \( \max(2\epsilon - \mu_i^{min}, 0) \) so that the messages are delivered consistently even with maximum clock skew and minimum message transmission delay.

However, there are some overheads in maintaining a timer in each task. A receiving task may not also start exactly at the expected dispatch time due to the scheduling of other higher-priority tasks in the same CPU. We need an explicit constraint on the input time that any message received after a task’s dispatch time are not processed in the current period.

PALSware uses a simpler solution to avoid the output timer. In this middleware, a source task transmits its messages without any additional output hold delay. Instead of buffering the messages at the source task, the middleware uses message timestamps and compares the timestamps to detect if the messages have arrived early. PALSware buffers the messages at the receiving tasks when they arrive early without requiring any timer.
In this approach, PALSware attaches a timestamp, \( x_1 \) in each message.\(^1\) If the current \texttt{PALS\_base\_time} of the source task \( M_s \) is \( jT_{hp} \), \( x_1 = (j + 1)T_{hp} \) for a PALS clock period of \( T_{hp} \). This indicates that the message is consumed in the PALS clock period \( j + 1 \) at the destination tasks.

The destination task then compares the timestamp \( x_1 \) of the pending messages with its current \texttt{PALS\_base\_time}, \( j'T_{hp} \) in the PALS clock period \( j' \). A message is accepted when \( x_1 = j'T_{hp} \). It is buffered when \( x_1 = (j' + 1)T_{hp} \). Otherwise, it is rejected. In non-faulty scenarios, a message is either accepted or buffered for at most 1 PALS clock period. The message is buffered in software with very minimal overhead. The buffered message is delivered at the next PALS clock period. The rejection of a message indicates that current clock skew at the source or the destination node violates the clock skew requirement of the PALS system. We address this timing fault in Section 6.5.

PALSware implements this timestamp-based message transmission and reception in two C++ classes: \texttt{TX\_PALS\_Port} and \texttt{RX\_PALS\_Port}. We define one object of \texttt{TX\_PALS\_Port} to send a message to an output port.\(^2\) Similarly, we use one object of \texttt{RX\_PALS\_Port} to read a message from an input port. These classes internally use system-dependent objects for unicast/multicast message communication. For example, we have developed a prototype of the middleware with POSIX libraries in Linux. We define two classes, \texttt{TX\_POSIX\_unicast\_port} and \texttt{RX\_POSIX\_unicast\_port}, for transmitting and receiving unicast UDP messages in this environment. In PALSware, these classes for message communications extend two abstract classes, called \texttt{TX\_port} and \texttt{RX\_port}, and implement the virtual functions, \texttt{send} and \texttt{recv}, respectively.

In Figure 6.2, we show the pseudocode of the \texttt{send} and \texttt{recv} functions of \texttt{TX\_PALS\_Port} and \texttt{RX\_PALS\_Port}. These objects also have a reference to the corresponding PALS\_task object to access the timing and scheduling parameters of a task such as period, PALS clock period, \texttt{PALS\_base\_time}, and \texttt{dispatch\_time}. The functions, \texttt{sysdep\_send} and \texttt{sysdep\_recv}, refer to the system-dependent services of message transmission and reception.

\textit{Note 1:} \texttt{TX\_PALS\_Port::send} and \texttt{RX\_PALS\_Port::recv} only guarantee that messages are processed at the next PALS clock period. The underlying network services must guarantee reliability. Later in Section 6.3, we show how to achieve consistent message communications in a real-time network architecture.

---

\(^1\)PALSware internally manages the timestamp information. Its use is transparent to the application logic.

\(^2\)The definition of the \texttt{port} is identical to the \texttt{event\_data\_port} in AADL. We only assume one-to-one or one-to-many port connections between the application tasks. In case of a many-to-one port connection, the application must define the order of messages that are received in the same port from multiple source tasks.
TX_PALS_Port::send(payload) {
    Let timestamp, x1 = PALS_base_time + PALS_clock_period;
    Append x1 and payload to form a message, msg;
    sysdep_send(msg);
}

RX_PALS_Port::recv(payload) {
    // Step 1: Process any previously received messages.
    // Let packet_container is a class variable to save packets, especially when they arrive early.
    // Each element of packet_container is a pair containing the timestamp and the payload.
    for (int i = 0; i < packet_container.size();)
        Read the i-th element (saved_x1, saved_payload) from packet_container;
        if (saved_x1 < PALS_base_time)
            Remove the i-th element from packet_containter as it is too old for the current period;
        else if (saved_x1 == PALS_base_time)
            Copy saved_payload to payload;
            Remove the i-th element from packet_containter;
            return;
        else { break; }
    }
    // Step 2: No more messages are queued. Read the messages from the network.
    while (true) {
        sysdep_recv(msg);
        if (msg is received) {
            Extract the timestamp (recv_x1) and the payload (recv_payload) from msg;
            if (recv_x1 == PALS_base_time)
                Copy recv_payload to payload;
                return;
            else if (recv_x1 == (PALS_base_time + PALS_clock_period)) {
                // That is, the message has arrived too early.
                Append (recv_x1, recv_payload) to packet_container;
            }
        }
        else { Copy NULL to payload;
                Return by throwing an exception that ‘nothing is received’;
            }
    }
}

Figure 6.2: Pseudocode of TX_PALS_Port::send and RX_PALS_Port::recv.

**Timing constraint.** The use of the timestamps simplifies the constraints specified in Section 5.2.1. In PALSware, we do not need the causality constraint explicitly. As long as the tasks are dispatched in the pre-defined global time interval according to the maximum clock skew of $\epsilon$, we only need the constraint on the task period. Similar to what we have in Equation 5.1, $T_i > 2\epsilon + \alpha_i^{max} + \mu_i^{max}$ for
each task $M_i$.

The middleware uses the timestamp of the messages to deliver them in the next PALS clock period similar to what we achieve with a multi-rate synchronizer. We still have to define a wrapper of RX_PALS_Port to filter messages based on the user-specified criteria such as last received message or a vector of all received messages.

6.2.4 Logical Synchronization Groups

In PALSware, a task may participate in more than one pattern instances to have logically synchronous interactions with different sets of components. We implement the concept of logical synchronization group in PALSware. To support the composition of these pattern instances, the middleware maintains a data structure for each logical synchronization group $G_i$ containing

- group’s identifier: $\text{GROUP}\_\text{ID}_i$,
- tasks of this group: \{${M_i,1}, \ldots , {M_i,n_i}$\},
- connections of this group,
- PALS clock period: $T_{hp}^i = lcm(T_{i,1}, \ldots , T_{i,n_i})$,
- current PALS_base_time of the group.

PALSware synchronizes a task’s start time with the PALS clock events of the groups in which it is a member. Thus, a task starts its logically synchronous computations at the local clock time $jT_{hp}$, where $T_{hp}$ is the least-common-multiple of the PALS clock period of these groups. Furthermore, for the message communication in a group, PALSware uses the corresponding PALS_base_time as the timestamp $x_1$.

In PALSware, application developers declare these pattern instances in a simple configuration file, currently described in the JSON format. Figure 6.3 illustrates a part of the configuration declaration. The configuration file declares the information of the logical synchronization groups (in the “pals_groups” block), the tasks (in the “components” block), and the connections (in the “connections” block). PALSware uses this configuration file to instantiate C++ objects for the tasks and the communication ports. In the future, this configuration file can be generated from the AADL models.
6.3 PALS Fault-Tolerant Communication Protocol

In this section, we specify the fault model of the PALS system. We discuss a possible distributed inconsistency when a task fails in the middle of its computation. We develop a fault-tolerant communication protocol to address this problem. We also use UPPAAL [41] for the model checking of this protocol. We discuss the UPPAAL model and the verification result in Section 7.3.

6.3.1 PALS Fault Model

Assumption 1. (Fail-stop nodes) The nodes in a PALS system are fail-stop. In this model, a faulty node fails by stopping its execution to minimize the fault propagation. A failed node does not send any extra message in the network.

Assumption 2. (Real-time reliable message communication) The PALS system uses a redundant real-time network architecture for reliable message communications. The default is the dual-redundant network architecture such as AFDX [40]. At most one of the two sub-networks may fail during operation. Both sub-networks have self-checking capability such as checksums to detect transmission errors. For increased reliability, a message may also be re-transmitted $k$ times, where $k$ is a known parameter for a given network. Since the probability of simultaneous errors in these sub-networks is very low, we assume that at least one of the two sub-networks deliver a unicast message to the receiver in bounded time.
6.3.2 Problem Description

Even with these assumptions, a node failure can cause inconsistency in a system. For example, a node may fail while a task is in the middle of its computation or message transmissions. As a result, different receivers may have partial and inconsistent views of the failed node. Consider an example configuration in Figure 6.4a. Here, outputs of the task $M_1$ are used as inputs in the tasks $M_2$ and $M_3$. In a normal condition, $M_1$ sends two messages $msg_1$ and $msg_2$ sequentially in period $j$. In this example, the failure of $M_1$ may manifest following conditions:

1. **Benign failure condition**: Figure 6.4b illustrates this condition. In this condition, $M_1$ fails after it has finished its execution and message transmission of period $j$. However, this failure does not cause any problem since the executions of $M_2$ and $M_3$ are consistent and both destination tasks receive all messages of $M_1$ before the next period $j + 1$. In this situation, they detect $M_1$’s failure at the period $j + 2$.

2. **Incomplete but consistent messages**: Figure 6.4c illustrates this condition. $M_1$ fails after
sending msg1 to both M2 and M3, but before sending msg2. In this case, even though M2 and M3 receive the messages consistently, they have partial view of M1’s outputs.

3. Inconsistent messages: In Figure 6.4d, M1 sends msg1 to both M2 and M3 successfully. However, it fails after sending msg2 to M3 but before sending to M2. As a result, M2 and M3 have inconsistent views of M1’s outputs.

In addition to the computation nodes, network devices can also fail. Based on the Assumption 2 of the PALS fault model, only one sub-network can be unavailable during the message transmission. The network can transmit individual unicast messages reliably in bounded time. However, a reliable unicast message communication is insufficient to prevent inconsistency. For example, the condition illustrated in Figure 6.4d happens when the network does not support reliable multicast message transmission.

6.3.3 A Simple Real-Time Reliable Multicast Service

In the PALS system, we use a simple communication protocol for real-time reliable multicast message transmission. The protocol satisfies the following property:

**Agreement.** Let a source task Ms transmits a message msg to a set of destination tasks MsD = {Md1, ..., Md_k}. In the PALS fault model, if a non-faulty destination task receives msg, then the other non-faulty tasks in MsD will also receive the message in the same period.

Based on this property, there are two acceptable failure scenarios in the PALS system: 1) the non-faulty destination tasks MsD receive all messages from the source task Ms if Ms does not fail in its period j, or 2) they receive an identical subset of messages from Ms if Ms fails. Figure 6.4b and 6.4c illustrate these acceptable scenarios. In the following, we discuss the related work and the implementation of this protocol.

**Related work.** Reliable multicast is a well-studied concept in the networking and distributed systems [42, 43, 44, 45, 46, 47, 48, 49, 50]. These protocols are however applicable for general-purpose computing systems. The assumption is that the network architecture provides a best-effort service for *eventual* message delivery. Thus, these protocols have to handle complex interaction scenarios, such as message omission and network partitions, to guarantee consistency within a
bounded time. On the other hand, in the real-time network architecture, at least one of the sub-networks deliver the messages in bounded time. Hence, we can implement a lightweight real-time multicast protocol with reduced complexity.

Furthermore, one of the main objectives of the existing protocols is to guarantee ordered delivery of the messages from multiple source tasks. For example, ISIS² [51] is a group communication service that delivers messages in different order: FIFO order using integer counter, causal order using Lamport’s vector clock, and total order using a form of lock. In addition, Kaashoek et al. [45] use a special node, called sequencer, to define the message order. Christian et al. [52] and Kopetz [53] use the synchronized local clocks as the timestamp to determine the message order and the delivery time. Abdelzaher et al. [54] use the token of a logical ring to define the order of events. In contrast, there is no need for additional ordering of the messages in a PALS system. The logically synchronous execution of the tasks, by default, provides the necessary order of the messages. The only requirement is to guarantee that the multicast of a message from a task $M_i$ happens within a maximum delay of $(T_i - \alpha_i^{max} - 2\epsilon)$.

There are two approaches to implement real-time reliable multicast protocols: network layer vs. middleware layer. A network layer multicast relies on either a linear bus or the multicast-aware network routers [53, 55, 56]. These protocols guarantee reliability based on the redundant transmission over duplicated sub-networks. On the other hand, middleware layer protocols are used when the network does not support the multicast. In this case, the middleware multicasts a message based on unicast transmissions.

Ideally, the PALS system can support both approaches for reliable multicast. In both approaches, one has to extend the abstract communication classes (TX_port, RX_port) of PALSware for the multicast service. The PALS communication model of Section 6.2.3 will then internally use these system-dependent multicast service.

**Implementation.** In this section, we discuss a simple implementation of a middleware protocol for real-time reliable multicast. The main idea of this implementation is based on the protocols proposed in [42, 43, 52]. In these protocols, the tasks are organized in a multicast group. During each multicast operation, a sender sequentially transmits a message to the receivers. Upon receiving a message, each receiver re-transmits to other receivers if it had not already received the same message. A task may crash any time during the multicast of a message. As long as at least one non-faulty receiver receives the message, other non-faulty receivers will also receive that message.
Figure 6.5: Use of $n$ fault-tolerant relay nodes for reliable multicast. In this figure, $n = 2$.

However, a naive approach requires $O(n^2)$ message delivery for each multicast message, where $n$ is the size of the multicast group. We use a simple extension. In this extension, we form a small multicast group of $n$ relay nodes, $\{R_1, R_2, \ldots, R_n\}$. A source PALS task transmits its message to these relay nodes. These relay nodes then apply the original multicast protocol to exchange the message within themselves. Once a non-faulty relay node has received a message, it first retransmits it to other relay nodes and then transmits to the destination PALS tasks. A destination task can receive at most $2n$ message from $n$ relays over two sub-networks. It selects only one message from the received messages. Figure 6.7 shows an example topology with two relay nodes. In this implementation, total number of unicast message transmissions is reduced when the number of relays is relatively smaller than the number of destination tasks.

We extend the PALS fault model with following assumptions about the relay nodes.

- **Assumption 1-1.** The relay nodes are also fail-stop.

- **Assumption 1-2.** At least one relay must be non-faulty or working during the operation of a real-time multicast operation. Thus, if this non-faulty relay receives a message from either another relay node or the source PALS task, it will propagate the message to the destination tasks in bounded time.

- **Assumption 2-1.** Each relay node is connected to both sub-networks. Based on the Assumption 2 of the PALS fault model, at least one of the sub-networks will deliver the messages to and from a relay within bounded time. We assume that maximum transmission delay to a relay node or from a relay node takes at most $\delta^{\text{max}}$ time.
We also assume that the processing of a message takes at most $\beta^{max}$ time at a relay. Both $\delta^{max}$ and $\beta^{max}$ are computed based on the real-time schedulability analysis of the relay nodes and the message communications.

```c
multicast_send(destinations, msg) {
    Send msg by using reliable unicast to the tasks in the set, destinations;
}

multicast_relay() {
    Wait for a msg from either a PALS task or other relays;
    if (msg has not been received before) {
        Update the message history of the output message port;
        multicast_send(relays, msg);
        multicast_send(destination_tasks, msg);
    }
}

TX_port_application_to_relays::send(msg) {
    multicast_send(relays, msg);
}

RX_port_relay_to_application::recv(msg) {
    Receive messages from the relays;
    Select one received message and copy into msg;
    Reject the remaining messages.
}
```

Figure 6.6: Pseudocode of multicast send and received operations.

Figure 6.6 gives the pseudocode of this multicast protocol. For example, a basic multicast operation with sequential transmission to a group of destinations is given in the function, `multicast_send`. The main logic of this relay node is `multicast_relay`. A PALS task uses the functions `TX_port_application_to_relays::send` and `RX_port_relay_to_application::recv` to send and receive the multicast message.

In this service, each relay maintains a data structure to detect if a message has been previously observed. In this implementation, a message from a connection, equivalently an output port of a source task, is uniquely identified by the connection identifier and a timestamp. Each relay node keeps a history of the timestamp of the last message received from a connection.

There are two approaches to define the timestamp in this protocol. Firstly, one can use a counter as the timestamp that increments from 0-to-N, where N is the maximum value of this counter. The counter value wraps around after N. Each multicast message in a connection from a source task has a counter value. The source task increments the counter at every multicast operation in a connection. A message is considered `new` at the relay node when its counter is one greater than the
saved value at the relay node. When a source task restarts, it sends a special \textit{start} message with $-1$ as the counter value to reset the saved value at the relay.\footnote{The start message may be piggybacked in the first message from the restarted node.} A restarted relay initializes the counter value as $-1$. The restarted relay accepts any message as long as the message counter is positive.

We discuss the UPPAAL model of this design in Section 7.3. Alternatively, the timestamp can also be based on the local clock time of the source node. Since we assume that the local clocks are monotonically increasing, the relay nodes can easily detect whether a message is new by comparing the timestamps.

\textbf{Lemma 1.} \textit{This protocol guarantees that the worst-case end-to-end delay of the successful multicast message transmission is $\delta_{\text{max}} + n(\beta_{\text{max}} + \delta_{\text{max}})$ for $n > 1$ relay nodes.} The time is measured from the time when the source task finishes its computation of the function “multicast\_send” to when the message arrives at the final non-faulty destination node.

\textbf{Proof.} The end-to-end delay in a successful multicast message transmission has three segments: 1) the message transmission delay from the source to the relays, 2) the delay within the multicast group of relays to reach the first non-faulty relay node, and 3) the delay from a non-faulty relay to a non-faulty destination task. The worst-case delays at the first and the third segments are simple. Since the network architecture guarantees a reliable message transmission to and from the relay node (Assumption 2-1), these segments are given as $\delta_{\text{max}}$ and $\beta_{\text{max}} + \delta_{\text{max}}$, respectively.

Christian el al. [52] discuss the scenario at which the worst-case delay occurs in a multicast group. In this group of relay nodes, the worst-case delay occurs when only one relay node remains non-faulty, and the remaining source and relay nodes fail during the same multicast operation. Let the messages are sequentially transmitted to relays $R_1$ to $R_n$. In this worst-case scenario, the nodes fail immediately after sending the message to the next relay in the list. Thus, the source task fails after it transmits the message to $R_1$. Similarly, the relay $R_k$ fails after receiving the message and transmitting the message to the relay $R_{k+1}$, for $k = 1 \ldots n - 1$. As a result, it takes at most $(n - 1)(\beta_{\text{max}} + \delta_{\text{max}})$ time to deliver the message to the final non-faulty relay node from the first relay. We illustrate this worst-case scenario in Figure 6.7 for three relay nodes. In this case, the source task and the relay nodes fail in a sequence until the message reaches the final non-faulty relay node.

It therefore follows that the worst-case end-to-end delay of a successful multicast message tran-
mission is $\delta^{max} + n(\beta^{max} + \delta^{max})$ in this protocol. ■

Figure 6.7: Worst-case message flow in the reliable multicast protocol for $n = 3$ relays. Gray ellipses represent faulty nodes that fail after one message transmission.

Lemma 2. When a source task $M_s$ delivers a message $msg$ to a set of destination tasks $M_s^D = \{M_{d1}, \ldots, M_{dk}\}$, if a non-faulty destination task receives $msg$, then the other non-faulty tasks in $M_s^D$ will also receive it.

Proof. Without any loss of generality, we assume that there are two non-faulty destination tasks $M_{d1}$ and $M_{d2}$, and $n$ relay nodes $\{R_1, \ldots, R_n\}$. We prove this lemma by contradiction. We assume that $M_{d1}$ receives a message $msg$ from a source task $M_s$, but $M_{d2}$ does not receive $msg$.

Let $M_{d1}$ receives $msg$ from a relay node $R_k$, for some $k$ (1 $\leq$ $k$ $\leq$ $n$). Thus, $R_k$ must be non-faulty at the time of sending $msg$ to $M_{d1}$. Since the relay nodes are fail-stop, $R_k$ must have failed before transmitting the message to $M_{d2}$ on both sub-networks (Assumption 1-1). Otherwise, $M_{d2}$ would have received the message based on the assumption of the real-time network architecture (Assumption 2-1).

In the proposed logic of the relay node in multicast_relay, a relay node propagates the message to other relays prior to sending to the destination tasks. Since there are at least one non-faulty relay nodes during the multicast operation (Assumption 1-2), they must receive this message $msg$ during the current multicast operation from $R_k$. Subsequently, at least one non-faulty relay nodes will propagate the message to both destination tasks $M_{d1}$ and $M_{d2}$. However, this contradicts our initial hypothesis. Thus, the lemma holds. ■

Proof of agreement. The proof of the above-mentioned agreement property follows from Lemma 1 and Lemma 2. Let the source task $M_s$ transmits the message $msg$ in its period $j$. If the message
arrives at a non-faulty relay node during the multicast operation, then the non-faulty destination
tasks receive $msg$ in a bounded time of $\delta_{max} + n(\beta_{max} + \delta_{max})$. Thus, by defining the maximum
message transmission delay $\mu_{s_{max}}$ of the source task $M_s$ as $\delta_{max} + n(\beta_{max} + \delta_{max})$ and using $\mu_{s_{max}}$ to
derive the bound on the source task’s period $T_s$, we guarantee that either all non-faulty destination
tasks receive $msg$ in the same period $j$ before the next dispatch of the source task or none of them
receive it. ■

6.4 Atomicity of Logically Synchronous Computations

The multicast protocol of Section 6.3.3 ensures consistency but does not guarantee atomicity.
Destination tasks may receive only a subset of the output messages because of the failure of a
source node. In the section, we extend the PALS communication model to guarantee atomicity of
the task execution during a period. This extension provides consistent information about the failed
state of a source task to the destination tasks. The destination tasks may use this information to
discard any of the partially received messages so that either all or no messages from a source task
is received during a period. Thus, a task failure can only be benign as shown in Figure 6.4b.

6.4.1 Protocol Implementation

End-marker message. In this protocol, PALSSware propagates a special end-marker message to
the destination tasks of $M_s$ at the end of $M_s$’s computation in each period. The end-marker mes-
gee has a timestamp, $x_{end} = kT_s$, denoting the completion of the period $k$ at $M_s$. PALSSware uses
the real-time reliable multicast service to send this message.

Processing of application messages. Let the destination tasks of a source task $M_s$ is given as
a set $M_s^D$. In this extension, the middleware appends another timestamp $x_2$ to each message from
a PALSS task. The timestamp $x_2$ is equal to the current dispatch_time of the source task. Let the
task $M_s$ transmits the message in its period $k' = n_s j + k$ i.e. the $k^{th}$ execution in the PALSS clock
period $j$. Thus, $x_2 = k'T_s$.

---

4End-markers are also commonly used when a large message is packetized. Either the last packet has an explicit
end-marker bit or the payload length is used as the end-marker. This proposed solution is different from end-marker
of the packets. Our solution is applied when a source task transmits separate messages in a period. In this approach,
we do not form a large packet by bundling all messages as the messages may be transmitted to different sets of
targets.
In this protocol, a destination task $M_d \in M^D_s$ compares the timestamp $x_2$ of an application message with the timestamp of the last received end-marker message. $M_d$ processes a message from $M_s$ sent at its period $k'$ only if an end-marker message with a timestamp, $x_{\text{end}} \geq k'T_s$ has been received.\(^5\) A message is delivered to the application when the conditions on both timestamps $(x_1, x_2)$ are satisfied. ($x_1$ is defined in Section 6.2.3.)

![Operation sequence diagram](image)

**Operation sequence.** Figure 6.8 illustrates the operation sequences of a task $M_s$ in non-faulty conditions. During each periodic execution, $M_s$ reads the end-markers of other tasks from the reliable multicast service in the system. It decides on the failed state of other tasks based on the received end-marker messages. PALSware then executes the fault detection logic of the user-specified fault manager (Section 6.5). If $M_s$ has no internal fault, PALSware executes the task logic defined in the function $\text{each}_pals\_period$ of the class $\text{PALS\_task}$. At the end of the task logic, PALSware propagates the end-marker message of current period to the multicast service, which then multicasts to the destination tasks of $M_i$.

In PALSware, we use a wrapper class, called $\text{PALS\_comm\_client}$, to manage the PALS communication objects, $\text{TX\_PALS\_Port}$ and $\text{RX\_PALS\_Port}$. It also transmits the end-marker message through the real-time multicast service. Application tasks use two functions, $\text{pals\_send}$ (connect-

---

\(^5\)A simple optimization is possible for the end-marker messages in a task participating in a multi-rate PALS pattern. Given the above mentioned logic, only the final end-marker message from the last execution in a PALS clock period is required to be transmitted in this fail-stop model.
tion_name, payload) and pals_recv(connection name, payload), to send and receive application messages. In the function pals_recv, the parameters payload is used as a call-by-reference parameter to return the received message from this function. We also have a variant of pals_recv to receive messages only when the corresponding source task has an atomic execution in the period of the message transmission. This function is given as pals_atomic_recv(connection name, payload). In these functions, we use connection_name as the input identifier for a connection from a source task to a group of destination tasks. We provide the pseudocode of pals_send and pals_atomic_recv in Figure 6.9. The only difference between pals_recv and pals_atomic_recv is that pals_recv does not compare the timestamps to detect the atomic execution of the source task.

```
PALS_comm_client::pals_send(connection_name, payload) {
    x_2 = dispatch_time;
    Create a message, msg by appending x_2 and payload;
    tx_port = pointer to a TX_PALS_port object for a given connection, named connection_name;
    tx_port->send(msg);
}

PALS_comm_client::pals_atomic_recv(connection_name, payload) {
    x2_end = timestamp of the last received end-marker message;
    rx_port = pointer to a RX_PALS_port object for a given connection, named connection_name;
    rx_port->recv(msg);
    Extract the timestamp, x_2 of the received message, msg;
    if (x2_end >= x_2) {
        // That is, the source did not fail in the period of transmission.
        Copy the rest of the message to payload;
        return the length of the payload;
    }
    else {
        // Inform users of the non-atomic execution of the source task.
        return -1;
    }
}
```

Figure 6.9: Pseudocode of the communication wrapper class, PALS_comm_client.

6.4.2 Protocol Property

We now discuss the atomicity property of this protocol. The atomicity of this protocol is obvious when we use the real-time reliable multicast service for all messages, including the end-marker message. Based on the agreement property of Section 6.3.3, the non-faulty destination tasks receive either all messages from the source task $M_s$ if $M_s$ does not fail in its period $j$, or an identical subset of messages from $M_s$ if $M_s$ fails. If an end-marker is received, the destination tasks treat $M_s$ as non-
faulty or working and subsequently deliver the received application messages from $M_s$. Otherwise, the destination tasks reject any of the partially received messages from $M_s$.

We can however achieve a simpler approach to achieve atomicity. In this section, we prove a sufficient condition based on only the real-time reliable multicast of the end-marker message. The remaining application messages can be transmitted sequentially to the destination tasks by using the available unicast message transmission of the real-time network architecture.

**Atomicity.** A real-time reliable multicast of the end-marker message is sufficient to guarantee atomicity in the proposed protocol under the assumptions of the PALS fault model.

**Proof.** The proof is divided into two steps. In the first step, we show that if a destination task $M_d$ receives an end-marker message of period $j$ from $M_s$, then it must have received all of $M_s$’s application messages of period $j$. In the second step, we show that the destination tasks agree on the received end-marker message.

The first step of this proof is based on the assumptions of the PALS fault model, and the ordered transmission of the application messages and the end-marker message. Let $M_s$ outputs a set of application messages $\{msg_1, \ldots, msg_k\}$ and the end-marker message $end_s$ in period $j$. In the proposed protocol, the middleware transmits $end_s$ after transmitting the application messages. By the assumption of the real-time network architecture, even if $M_s$ transmits a message $msg_j$ sequentially to the destination tasks, all non-faulty destination tasks receive it as long as $M_s$ does not fail in the middle of this transmission. We assume that the nodes are fail-stop. Hence, if $M_s$ does not fail prior to transmitting $end_s$, the non-faulty destination tasks must receive the previous application messages. Furthermore, the period of the source task $M_s$ is sufficiently large to guarantee that a message arrives in the same period at the destination tasks. In the multi-rate PALS system, the worst-case multicast message delay of these messages, including the end-marker message, from the source task $M_s$ is less than $(T_s - \alpha_{max}^s - 2\epsilon)$. Thus, it is obvious that if a non-faulty destination task $M_d$ receives $end_s$, it must have received other messages $\{msg_1, \ldots, msg_k\}$ from $M_s$ in the same period.

The second step of this proof directly follows from the agreement property of the real-time reliable multicast protocol such as the protocol of Section 6.3.3. In case of a real-time reliable multicast of an end-marker message, if a non-faulty destination task receives the end-marker message from the source task $M_s$, the other non-faulty destination tasks also receive it in the same period. This
implies that the non-faulty destination tasks agree on the failed state of $M_s$ based on the availability of the end-marker message.

Based on these steps, when the end-marker message from the source task $M_s$ is available at the non-faulty destination tasks, these tasks consume all application messages from $M_s$ sent in the same period as the end-marker message. Otherwise, these tasks decide consistently that $M_s$ has failed to transmit the end-marker message successfully and reject any received messages from $M_s$.

6.5 Fault Managers in PALSware

PALSware supports integration of fault managers to detect any software fault that can adversely affect the PALSw system requirements. In PALSware, we define a generic interface for the fault managers, called AbstractFaultManager. It has a virtual function, called check_pals_faults. PALSw_task invokes this function to check the assumptions in each period. Users can extend this interface to define application-specific fault managers and plug into PALSware. If the fault manager detects a fault, PALSw_task does not execute the task logic. It rather invokes a user-defined function, called fail_safe, to have an application-specific graceful termination of the task.

While many of the faults are generic and have system-wide detectors, a small class of timing faults is critical to the correctness of the logical synchronization. For example, PALSware can integrate with existing clock synchronizers [57, 58, 34] to synchronize the local clocks. These clock synchronizers nowadays achieve a clock skew in the order of microseconds in a local area network. Such precision is sufficient for many real-time control applications. However, there are certain operational conditions, such as system initializations or faulty kernels, in which the clock skew requirement may be violated. In this section, we demonstrate some of these fault scenarios and discuss a fault manager that detects the clock synchronization error locally.

6.5.1 Clock Synchronization Error

Clock synchronization is a well-studied concept in distributed systems community [34, 35, 59, 60, 61]. Since the hardware clock oscillators do not progress precisely at the same rate, the clocks of the distributed nodes eventually drift apart. The distributed nodes must therefore synchronize their local clocks to satisfy the PALSw timing requirements. In a PALSw system, each node executes
a clock synchronizer process to synchronize the local clocks with respect to a reference clock server. The clock synchronizer periodically computes the clock offset or clock skew with respect to the reference clock server and adjusts the local clock time to minimize the timing error.\textsuperscript{6}

We identify two integration problems when the clock synchronizer and the PALS application execute as independent processes in the operating system. We discuss these problems in the context of two clock synchronizers. The first clock synchronizer is our prototype implementation of the Christian’s algorithm [34]. We refer it as ClockSyncProto in this section. In this clock synchronizer, nodes interact in a master-slave mode. Each node runs a clock synchronizer process that periodically communicates with reference (master) clock servers to synchronize the local clock.

The second clock synchronizer is an open-source implementation of IEEE-1588 Precision Time Protocol (PTP) [57, 62]. PTP synchronizes the distributed clocks in a broadcast mode. In this mode, a reference clock server periodically broadcasts its clock information to other nodes. Each node runs a PTP daemon. This daemon then synchronizes the local clock based on the reference clock information.

**Problem 1:** A PALS application has a startup dependency on the clock synchronizer. During the node startup, the local clocks usually have a large clock offset or clock skew. In this condition, a clock synchronizer typically resets the clock to match with the reference clock. Despite the initial reset, it may still take several minutes to reduce the clock synchronization error below an acceptable range. As a result, the execution of a PALS task may be delayed after the node restarts. Figure 6.10 demonstrates this phenomenon for both ClockSyncProto and PTP. For example, in our experiments, it takes 3-6 minutes to reduce the clock offset from 0.19 seconds (approximately) to a value in the order of hundreds of microseconds.

**Problem 2:** Both clock synchronizer and PALS application have some shared dependency. Both of them depend on the local clock time maintained at the operating system. A clock synchronizer uses the local clock time to attach the timestamps to its messages and measure the delay to the reference clock server. On the other hand, PALSware uses the local clock time for the PALS clock events and task executions. Thus, any fault affecting the local clock time, such as in the form of time loss or faster clock, also affects these processes.

\textsuperscript{6}The clock offset is an equivalent measurement of the clock synchronization error as the clock skew defined in Section 2.1. For a local clock time $c(t)$ at the global clock time $t$, the true clock offset is given as $c(t) - t$. With respect to the reference clock server’s clock time $c_{ref}(t)$, the clock offset is $c(t) - c_{ref}(t)$. We discuss the relationship between the clock offset and the clock skew in Appendix F.
For illustration purpose, we use a simple kernel module to emulate the effect of time loss in an operating system. When we load this module, it disables the timer interrupt and implements a busy loop before re-enabling the interrupt. Since the operating system does not update the clock time during this interval, it loses some time from its measurement.\(^7\) This results in a spike of about 4-5ms in the clock offset. As a result, the tasks in the node may no longer be logically synchronized with the tasks in other nodes. Figure 6.11 demonstrates this experimental result for both clock synchronizers.

Note: In this section, we do not address the faults of the reference clock servers. Appendix F

\(^7\)Note that the timer interrupt should not be disabled for a long period of time in actual systems. It is not often allowed to disable the timer interrupt at all. Yet, a faulty device driver, a buggy kernel or even a simple memory bit-flip can affect the local clock time to result in similar behavior.
gives a generic concept of the arbitrary faults in these servers. We believe that existing research results may be applied in these problems [63, 38, 37].

6.5.2 Experiments with a Timing Fault Manager

We have developed a fault manager to monitor the clock synchronization state and detect the above mentioned timing errors in PALSware. The fault manager uses two fault detection logic. The first solution is supported on any architecture but requires a simple extension of existing clock synchronizers. On the other hand, the second solution is architecture-specific and requires processor counters for fine-grained timing information.

In the first solution, the clock synchronizer updates the current clock offset or clock skew estimate periodically in a pre-defined shared memory. The fault manager in PALSware reads this published value at every period. If the current clock skew is greater than the expected maximum clock skew, then it is not safe to start or execute the PALS tasks. We currently use this solution to detect the clock synchronization error during the task startup. PALSware does not start the tasks until the clock skew stabilizes below the expected bound.

In this technique, suppose that the clock synchronizer process notifies its clock skew estimate at a maximum interval of R and PALSware reads this value at a minimum interval of T. Let \( t_s \) be the most recent local clock time when the clock skew estimate is updated and \( t_p \) be the most recent local clock time after \( t_s \), i.e. \( t_p \geq t_s \), when PALSware reads this value. We define two conditions for this monitoring:

1. \( (t_p - t_s) \leq R \): It means that there is a recent update of the clock skew from the clock synchronizer process.

2. \( (t_p - t_s) > R \): It means that the clock skew estimate is stale. Either the clock synchronizer has crashed or is making no progress. In this condition, PALSware, by default, stops the execution of the task. Note that it may be possible to continue the task execution for an additional interval. However, this needs to consider the maximum clock drift rate and last clock skew estimate. If the last clock skew estimate is \( \Delta \) and the maximum clock drift rate is \( \rho \), then the computation can safely progress until a future time \( t \) as long as \(|\Delta| + (t - t_s)\rho < \epsilon\). Beyond the time \( t \), we cannot guarantee the PALS clock skew requirement without reinitializing the clock synchronizer.
However, the clock synchronizers typically compute the new clock skew value at the time of resynchronization. Traditionally, the clock resynchronization happens in a very slow rate, often in the order of seconds or minutes. As a result, even if the clock is bad, or there are bugs in the underlying operating system, the fault manager cannot detect the fault until the next resynchronization time.

We address this problem in the second fault detection technique in which we use high-resolution timers available in the processor. For example, x86 processors currently support a 64-bit counter, called Time Stamp Counter (TSC) in each CPU. Its value is incremented at every pulse of the hardware clock oscillator. Since TSC is not subject to the interrupt, it is less prone to any time loss. System-specific fault detectors in PALSware can read these counters and measure the time between two consecutive PALS clock events or two successive periodic dispatches. If the current measurement deviates significantly from the regular execution, the fault manager notifies the middleware of a possible timing fault. In this approach, the exact accuracy of the processor counter, such as TSC, is not necessary to detect any timing error. We can afford some inaccuracies as long as we know a reasonable bound on the TSC measurement of the intervals.

**Experimental results.** In this study, we compare the effectiveness of these two fault detection techniques based on the time-loss error. We inject the fault by loading the above-mentioned kernel module multiple times in the target node. In order to evaluate for multiple fault injections, we let the PALS task continue even after the fault manager detects the timing error. Figure 6.12 gives the experimental results of this fault injection experiment for both ClockSyncProto and PTP.

In our experiment, these clock synchronizers notify the clock offset or clock skew value at every 2s interval. For the purpose of demonstration, we test with a PALS task with a period of 100ms. We observe that there are sudden changes in the measurements of clock offsets and TSC measurements after we load the kernel module. The fault manager can easily notice the anomalous behavior based on the prior knowledge of the maximum clock skew and the bounds on the TSC measurements of the task period.

As shown in Figure 6.12, the fault manager based on the high-resolution processor counter, such as TSC, performs better in this experiment. Figure 6.12c gives the exact period number at which the fault manager detects the error in this experiment. Since the clock synchronizer updates the current clock synchronization error at a slower rate than the task period, there can be many intermediate executions that are unaware of the timing fault in the system. In this configuration,
we may decide to reduce the update interval or the clock resynchronization interval. However, this approach increases the network and processing overhead.

Experiments with ClockSyncProto to detect timing error

Experiments with PTP daemon to detect timing error

(a) ClockSyncProto

(b) Precision Time Protocol (PTP) daemon

<table>
<thead>
<tr>
<th>Clock synchronizer</th>
<th>Fault #</th>
<th>Offset</th>
<th>TSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClockSyncProto</td>
<td>1</td>
<td>542</td>
<td>538</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2025</td>
<td>2006</td>
</tr>
<tr>
<td>PTP daemon</td>
<td>1</td>
<td>753</td>
<td>752</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2893</td>
<td>2893</td>
</tr>
</tbody>
</table>

(c) The task periods at which the fault manager detects the error.

Figure 6.12: Monitoring of clock offset (or skew) update and TSC to detect timing error.
CHAPTER 7
EXPERIMENTAL STUDY

This chapter discusses a number of experimental studies on various aspects of the PALS middleware, PALSware. In Section 7.1, we present an experimental study of the middleware in a distributed control system of an inverted pendulum. In this section, we also discuss a scenario of how we use PALSware’s guarantee of logical synchronization to solve a performance problem, in addition to the distributed consensus. In Section 7.2, we discuss a fault-injection framework that we use to test the PALS applications. In Section 7.3, we describe a UPPAAL model of the PALS fault-tolerant communication protocol.

7.1 Case Study: A Dual-Redundant Control System

7.1.1 Description

The inverted pendulum is a linear-motion servo-plant consisting of a movable cart and a free turning rod attached to the cart [64]. The servo control of the inverted pendulum is based on a feedback controller that reads the measurements of the cart position and the rod angle. The controller then controls the cart movement on a steel shaft to maintain the rod in its upright position. The failure in this control system happens when the rod falls over.

Figure 7.1 shows the setup of our experiment on 4 distributed machines. We have two redundant servo controllers: Side1 and Side2. The pendulum is connected with the I/O machine that executes both sensing and actuation logic. Both servo controllers execute the same feedback control logic. They also collaborate in an active-standby configuration. The coordination of these two controllers is similar to our model of Section 4.2 except that we do not use the input variables: side1FullyAvailable, side2FullyAvailable, side1Failed, and side2Failed. We implicitly assume that the subsystems are always available and the controller tasks are crashes in case of a fault. The user can issue a command from the user interface machine to flip the active-standby
status of a controller similar to manualSelection of Section 4.2.

In the experiment, there are two logical synchronization groups. The servo control synchronization happens between the tasks of the I/O, Side1 and Side2 machines at a PALS clock period of 20ms. On the other hand, the user command synchronization happens between the tasks of the Side1, Side2 and user interface machines at a PALS clock period of 40ms. In our implementation, there is one task per machine in this implementation. Both Side1 and Side2 execute a single task for both servo control and active-standby coordination that execute a period of 20ms. The user interface runs an environment input synchronizer to propagate the user command logically synchronously at a period of 40ms.

We use PALSware to guarantee consistent inputs at the servo controllers. Such consistency is crucial for the inverted pendulum. For example, when there is no active controller, we do not normally execute a control command. As a result, we can visually observe the jittery movement or the failure of the inverted pendulum, when the controllers are inconsistent and do not agree on who is active.

7.1.2 C++ Implementation

Figure 7.2 and 7.3 give the code snippet of the periodic tasks in Side1 and Side2. We particularly show the active-standby configuration logic in these two figures. Here, Side1_task and Side2_task are two child classes of the PALS_task class that implement the logically synchronous computations in these servo controllers. We define the periodic computations in the each_pals_period function.
bool Side1_task::each_pals_period() {
    
    // 1. Define task-specific default values for messages.
    int8_t side1 = NO_MSG;
    int8_t side2 = NO_MSG;
    bool user_cmd = false;

    // 2. Receive previous period’s data, given that the source has not crashed.
    comm_client->pals_recv("side1_status", &side1, 1);  
    comm_client->pals_recv("side2_status", &side2, 1);  
    comm_client->pals_recv("user_cmd", &user_cmd, 1);  

    // 3. Decide on which side is active, based received information from
    // Side1. Side2, and the user.
    next_side1_state = active_standby_logic(side1, side2, user_selection);
   // 4. Send current period’s state.
    comm_client->pals_send("side1_status", &next_side1_state, 1);
    ...
}

(a) The each_pals_period function.

int8_t Side1_task::active_standby_logic(int8_t side1, int8_t side2, bool user_cmd) {
    // If both sides have same status, side1 becomes ACTIVE
    if(side1 == side2) {
        return ACTIVE;
    }

    // If side2 is alive, but side1 has just woken up. Therefore, side1 starts as STANDBY.
    else if (side1 == NO_MSG && side2 != NO_MSG) {
        return STANDBY;
    }

    // If side2 is down, but side1 is running. Thus, side1 becomes ACTIVE.
    else if (side1 != NO_MSG && side2 == NO_MSG) {
        return ACTIVE;
    }

    // When both are alive, a new user command flips the active/standby status.
    if (user_cmd) {
        if (side1 == ACTIVE) return STANDBY;
        else return ACTIVE;
    }

    return side1;
}

(b) The active_standby_logic function.

Figure 7.2: Code snippet of Side1’s each_pals_period.
bool Side2_task::each_pals_period()
{
    
    // 1. Define task-specific default values for messages.
    #define at 1
    #define at tие 1
    #define at 1
    int8_t side1 = NO_MSG;
    int8_t side2 = NO_MSG;
    bool user_cmd = false;

    // 2. Receive previous period's data, given that the source has not crashed.
    comm_client->pals_recv("side1_status", &side1, 1);
    comm_client->pals_recv("side2_status", &side2, 1);
    comm_client->pals_recv("user_cmd", &user_cmd, 1);

    // 3. Decide on which side is active, based received information from
    // Side1, Side2, and the user.
    next_side2_state = active_standby_logic(side1, side2, user_selection);

    // 4. Send current period's state.
    comm_client->pals_send("side2_status", &next_side2_state, 1);
    ...
    return true;
}

(a) The each_pals_period function.

int8_t Side2_task::active_standby_logic(int8_t side1, int8_t side2, bool user_cmd)
{
    // If both sides have same status, side2 becomes STANDBY
    if (side1 == side2) {
        return STANDBY;
    }

    // If side1 is alive, but side2 has just woken up. Therefore, side2 starts as STANDBY.
    else if (side2 == NO_MSG & side1 != NO_MSG) {
        return STANDBY;
    }

    // If side1 is down, but side2 is running. Thus, side2 becomes ACTIVE.
    else if (side2 != NO_MSG & side1 == NO_MSG) {
        return ACTIVE;
    }

    // When both are alive, a new user command flips the active/standby status.
    if (user_cmd) {
        if (side2 == ACTIVE) return STANDBY;
        else return ACTIVE;
    }
    return side2;
}

(b) The active_standby_logic function.

Figure 7.3: Code snippet of Side2's each_pals_period.
For the simplicity of presentation, we define separate classes for these servo controllers. Both classes use the same feedback control logic. The only difference between these classes is in the initialization of the active-standby configuration. For example, when both Side1 and Side2 start at the same period, we break the tie in favor of Side1. In such case, Side1 starts as the active controller, where as Side2 starts as the standby controller.

In PALSware, we use a PALS_comm_client object, called comm_client, for the message communication in each PALS task. If the source task fails or the input is not available, we use a default value for the input data port. Note that even though the connections may belong to separate PALS patterns, the interfaces remain same in PALS_comm_client. PALSware hides the pattern managements using the system configuration discussed in Section 6.2.4. PALSware guarantees identical input views of the previous period’s state and the user command at the servo controllers. Based on this guarantee, these controllers remain consistent and perform identical state transitions. For example, in the active_standby_logic function, the controllers can flip their active/standby status in the same period when they are both alive and there is a corresponding user command. Without the PALS pattern guarantee in PALSware, it would not have been possible without acknowledgment messages or a complex interaction protocol.

### 7.1.3 Experimental Setup and Result

<table>
<thead>
<tr>
<th>Machine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side1</td>
<td>OS: Linux 2.6.28-11-generic&lt;br&gt;Dual core processor. Intel(R) CPU T2300 1.66GHz&lt;br&gt;Total memory: 500160 kB&lt;br&gt;Cache size: 2048 KB</td>
</tr>
<tr>
<td>Side2</td>
<td>OS: Linux 2.6.31-14-generic&lt;br&gt;Single core processor. Pentium II 349.123 MHz&lt;br&gt;Total memory: 250716 kB&lt;br&gt;Cache size: 512 KB</td>
</tr>
</tbody>
</table>

Table 7.1: System configuration.

In this experiment, the four machines are connected via Ethernet switches. Table 7.1 gives the configuration of these machines. We can assume this system as a pseudo real-time system since the network is small and other tasks or communications have not significantly interfered our experiments. In this case study, we use ClockSyncProto to synchronize the clocks of these machines. The clock skew is less than 0.5ms in normal configuration. We discuss the clock synchronizer’s operation in Section 6.5 and Appendix F.
Later in Figure 7.7a, we give a boxplot of the response times of the application tasks. The I/O machine’s task has the largest response time because of its processor speed with respect to other machines. Furthermore, the I/O machine has a single core. Thus, the response time is also increased since we have to run both clock synchronizer and the I/O task in the same core. On the other hand, we run these tasks in different core in other machines.\footnote{The I/O board of the inverted pendulum uses an ISA card. We an old I/O machine that supports ISA.}

![Variation of cart’s position and rod’s angle](image)

Figure 7.4: Inverted pendulum response. Fault injections are marked as blue circles.

We ran this experiment for maximum 5hrs. The pendulum remained stable and balanced during this time. Figure 7.4 gives a sample plot of the cart’s position and the rod’s angle. Here, we manually killed the active controller (Side1) at period 11076. Side2 became active at period 11077. We then restarted Side1 at period 11985, and finally killed the active controller (Side2) at period 12037. There was however a slight perturbation after the failure of the active controller (shown by the blue circles). This is expected in the synchronous design. When the active controller fails, there is no active control command for a period. Even though the standby becomes active in the next period, missing a control command for a period affects the pendulum’s response for a brief interval. We therefore amend the design based on PALSware’s guarantee that the active controller may be unavailable for a limited number of periods (in this case, one period). We now use the last active controller’s command when no one is active. This solution improves the pendulum’s response during the failure of the active controller. Figure 7.5 shows the pendulum’s response after this fix. In the amended solution, we do not observe any jittery movement in the control of this
inverted pendulum.

Figure 7.5: Inverted pendulum response after our fix to handle the failure of the active controller. Fault injections are marked as blue circles.

**Note:** In the worst-case, it may take up to 2 periods before this active-standby design can have a new active controller after the failure of the current active controller. The worst-case scenario happens when the active controller fails in the same period as the other controller has restarted. In our design, the restarted controller may take up to 2 periods before becoming active, in case the other controller is failed. Two periods for a restarted controller is required to overcome any confusion when both controllers start at the same time.

### 7.2 Validation of Agreement and Atomicity

Distributed testing is simple with PALSware. Since PALSware delivers an application’s messages in the order of the PALS clock events, we can easily validate the global state properties. To test a distributed algorithm in PALSware, we only need a validator task that receives the messages from the target PALS tasks and execute a validation logic on the global state in each PALS clock period.

In this work, we extend this concept to test the agreement and atomicity properties mentioned in Section 6.3 and 6.4. We implement a testing framework to inject faults in the code of a PALS task and validate these properties even when the task fails in the middle of its computation. Figure 7.6 illustrates this framework. There are 4 components in this framework: a fault injection driver, a
Figure 7.6: Fault injection test setup.

target PALS task $M_s$ interacting with other distributed tasks $M_d$, a set of replicated fault monitors and a validator task.

The fault injection driver and the target PALS task $M_s$ execute in the same machine. The fault injection driver injects a fault in $M_s$. Currently, we kill the task randomly during its computation. In this framework, the fault monitors act as dummy destination tasks to monitor $M_s$’s state. A fault monitor therefore executes identically at the same rate as a destination task of $M_s$. For example, in case of the servo controller, we emulate the I/O task to be a fault monitor. Since the fault monitors are assumed to be non-faulty during the testing, we only need two redundant copies of a fault monitor to detect any inconsistency. These fault monitors receive the end-marker and application messages from the target task. In each period, the fault monitors notify the state of the received end-marker and application messages to the validator. The validator then compares these messages to detect any inconsistency due to a bug or an implementation error.

This framework is currently developed in Linux. In order to control the fault injection point, the fault injection driver uses the *ptrace* system call of Linux to monitor the system call invocations from $M_s$. The fault injection driver then randomly kills the target task after a number of system calls based on a given MTTF (*mean time to failure*) parameter. In this context, MTTF is the statistical mean on the number of system calls after which $M_s$ fails in a fail-stop manner.

---

2We thank Daniel Chen, Kuan-Yu Tseng, and Cuong Pham of CSL, UIUC for their help with the fault injection driver. We use some part of their tool, called *pfi*, to build our own fault injection driver.
Performance overhead. The fault injection experiments increase the task response time due to the overhead of ptrace and interruption of the system calls. Figure 7.7 compares the response times of the application tasks of Section 7.1 with and without fault injection experiments. The boxplot shows that the upper bound of the response time of the I/O task increases about 80%. We also notice some outliers during the fault-injection experiments. These outliers result in some false positives if we do not adjust the task periods. For testing purposes, we increase the task periods by a constant factor to reduce the number of false positives. In these cases, the code for the I/O tasks also have to be replaced with a dummy code since the adjusted control periods are not appropriate for the physical control of a device. Furthermore, this framework only validates the consistency of the middleware, not the application semantics.

![Graph showing response time with and without fault injection](image)

(a) Response time (without fault injection).

(b) Response time (with fault injection).

Figure 7.7: Comparison of response time with and without fault injections in the inverted pendulum control system of Section 7.1.

### 7.3 Verification of PALS Fault-Tolerant Communication Protocol

In this section, we describe the UPPAAL model of the fault-tolerant communication protocol of Section 6.3.3. In the UPPAAL model, we currently model the reliable multicast from a single source task. The source task multicasts its messages to a group of distributed relays. These relays then exchanges the source message within themselves. Once a relay receives a new message either from the source task or remaining relays, it multicasts the message to other relays and the destination tasks. In this protocol, each message has a counter. During each multicast operation, the counter increments by one. Thus, the relays can know whether an incoming message is new based on its
counter value and the counter of the last received message.

In this model, the source and the relays can fail non-deterministically any time during the multicast operation. We only add a constraint on the number of non-faulty relays. In each multicast operation, there must be at least one non-faulty relay to deliver the message to the destination tasks.

The rest of this section gives a brief overview of the components of this model and verification of the agreement property. We compare the model checking results for different size of the relay group. In this model, we do not explicitly model the destination tasks. Instead, we use an output variable for each relay. We compare these output variables to evaluate the agreement property.

7.3.1 Brief Overview of UPPAAL

UPPAAL [41] is model checker for real-time embedded systems. It models a system as a network of timed automata. A timed automaton is a finite-state machine with clock variables. One can define the automaton with a finite number of locations and transitions between the locations. UPPAAL models the state of the modeled system as the current locations of the timed automata, their clock conditions and other user-defined variables. Each automaton instance in UPPAAL is referred to as a process. Users use a process template to declare the locations and transitions of an automaton. The process templates are parameterized. One can instantiate a process by defining the values of the parameter.

In UPPAAL, each transition (edge) between two locations of an automaton defines 4 optional labels: select, guard, sync, update. The select label allows to select a value non-deterministically from a range such as an integer range and apply the value in other labels of the transition. The guard defines an expression that must be true to execute the transition. Two processes in UPPAAL can have synchronized transitions based on a synchronization label defined in sync. UPPAAL uses the notion of channel to synchronize the processes. The sync label of a transition uses the expression of type c? or c! for a given channel c. A process with the sync label of c! synchronizes with another process with the sync label of c? as long as the receiving process’s guard expression is true. The automaton can also update its variables based on the expression of the update label. UPPAAL also defines the notion of committed and urgent locations. In the model, these locations are marked with C and U, respectively. These locations are used to have restrictive executions. For example, time does not progress in a system when a process is in these locations. In case of the committed
locations, the next transition in the system must be from a committed location.

7.3.2 Description of the Model

There are 4 process templates in our model of this protocol:

- **Driver**: We have a process template, called Driver, to coordinate the interactions during a multicast operation. We instantiate this process template and define a process, called MainDriver. In our model, the driver also plays a partial role for the source task. During each multicast operation, it increments the message’s counter value prior to the message transmission to the relays.

- **MulticastSrcMsg**: We model a process template, called MulticastSrcMsg, to transmit a source task’s message to the relays. This process sequentially transmits this message to the relays. It also models the failure of the source task during the multicast operation. The source task can fail before, after or in the middle of the computations of this process.

- **Relay**: The operations of the relays are divided into two process templates in our model: Relay and MulticastRelayMsg. In the Relay process, we synchronize the relay’s message receipt from the source task through the process MulticastSrcMsg and from other relays through the process template MulticastRelayMsg. Once a relay receives a new message, it propagates the message to other relays. In UPPAAL, we instantiate the process template Relay for all relays in the system.

- **MulticastRelayMsg**: We model the message transmission from a relay to its neighboring relays through this process template. We also model the non-deterministic failure of a relay node during the sequential message transmissions in this process. We instantiate this process template for all relays in the system.

We now provide a detailed description of these process templates.

**Driver process.** Figure 7.8 gives the UPPAAL representation of this process template. We use a single clock clk in this process. The multicast operation in this model spans across one clock step of this process. At each clock step, this process increments the message counter and transmits a message on behalf of the source task. According to the proposed protocol, the counter increments
from 0 to $\text{MAX\_VAL}-1$. ($\text{MAX\_VAL} = 18$ in the model.) When the source task fails, we set the counter value to $\text{FAILED\_VAL}(-2)$. When the source task recovers, we also reset this counter to $\text{RESET\_VAL}(-1)$. This counter is set in the function, set_new_val during the transition $L1\rightarrow L2$.

The driver then generates an event on the channel $\text{relay\_reset}$. This event non-deterministically resets or restarts any of the previously failed relay nodes. In this way, we can control the execution of the relays across different clock steps. In the transition $L3\rightarrow L4$, the driver triggers an event on the channel $\text{src\_msg}$ to initiate the process of multicasting the source task’s message to the relays. This process waits at the location $L4$ until the operation of the multicast operation completes. Note that $L4$ is not a committed location in our model. Thus, UPPAAL will handle the transitions from committed locations in other processes prior to executing a transition at this process.

**MulticastSrcMsg process.** Figure 7.9 gives the UPPAAL representation of this process template. After the synchronization on the $\text{src\_msg}$ channel, this process sequentially sends an event to each relay node by using the channel $\text{src\_msgs}[i]$, $\forall i \in \text{MAX\_RELAYS}$. We use the variable $\text{MAX\_RELAYS}$ to give the total number of relays in the system. We also model the failure of the source

---

3In actual implementation, this wait is implicit when the period interval is sufficiently large.
task. The process may move from the location **MULTICAST_LOC** to **INIT** non-deterministically and set the message counter to **FAILED_VAL** to simulate the failure of the source task.

![Figure 7.10: Relay process template.](image)

**Relay process.** Figure 7.10 gives the UPPAAL representation of this process template. We synchronize the processes Relay and MulticastSrcMsg using the channel `src_msgs[i]`, where `i` is the index of the relay node. In this synchronization, each relay checks if the input message’s counter (given by `src_val`) is acceptable or not. Each relay saves the previously observed counter value in a variable `relay_vals[i]`. As mentioned above, a relay accepts a new message if the new message counter is equal to `RESET_VAL` or `(relay_vals[i]+1) % MAX_VAL`. In the model checking, we find that there is a third condition to accept a message. When the failed relay restarts, it updates its internal variable for the previously observed counter value to `RESET_VAL`. In this condition, an input message with a positive counter is also accepted. We give the function `accept_src_val` that checks a message from the source task.

```cpp
bool accept_src_val(val_t val) {
    if (val == FAILED_VAL || relay_vals[id] == FAILED_VAL) return false;
    if (relay_vals[id] == RESET_VAL && val != FAILED_VAL) return true;
    return (val == RESET_VAL || val == (relay_vals[id] + 1) % MAX_VAL);
}
```

In case of a reset message, the relay does not propagate the message to other relays. Otherwise, a relay triggers an event on the channel `mult_relay_msg` to initiate a multicast operation to other relays (shown by the transition L2→L1).
Similarly, when a relay receives a forwarded message from other relays, it first checks if the message is new. In case of a new message, the relay forwards the message to other relays (shown by the transition \( L3 \rightarrow L1 \)). In the following, we give the code snippet of the function `accept_relay_val`, which checks an incoming message from another relay.

```c
bool accept_relay_val(val_t val) {
    if (val < 0 || relay_vals[id] == FAILED_VAL) return false;
    if (relay_vals[id] == RESET_VAL && val != FAILED_VAL) return true;
    return (val == (relay_vals[id] + 1) % MAX_VAL);
}
```

In both transitions, we update `relay_vals[i]` with the new incoming message’s counter in the function `update_val`.

Figure 7.11: MulticastRelayMsg process template.

**MulticastRelayMsg process.** Figure 7.11 gives the UPPAAL representation of this process template. We manage a 2-dimensional array of channels to transmit an event to other relays. We also model the non-deterministic failure of the relay node during the multicast operation. We only take a transition to inject a fault (denoted by the transition \( MULTICAST\_LOC \rightarrow FAULT\_LOC \)) when there are other non-faulty relays. At this transition, we set `relay_msgs[i]` to `FAILED_VAL`. The fault persists until the driver process generates an event `relay_reset`. At this event, we non-deterministically restart the relay node by resetting its saved counter `relay_msgs[i]` to `RESET_VAL`. Note that as long as the value is set to `FAILED_VAL`, the relay does not accept any new value.

In order to propagate the relay’s content to the receivers, we use an array, called `recv_vals`. Each relay of index \( i \) updates `recv_vals[i]` with `relay_vals[i]` after if it has successfully transmitted the message to other relays.
7.3.3 Verification Results

In order to store the input message’s counter during the multicast operation, we use a UPPAAL meta variable, called meta_src_val.\textsuperscript{4} meta_src_val is set to src_val during the transition L1→L2 at Driver so that the failure of the source task does not affect this meta variable.

Based on the recv_vals array, we can easily verify the agreement property when a multicast operation completes. In our model, we define this completion at the location L1 of the Driver process. In this model, we use the following safety property to verify the agreement among the relay nodes as well as the destination tasks that use the message from the relay nodes.

**Property 1.** If one of the relay nodes accepts the source task’s message during the multicast operation, other relay nodes also accept the same message unless they have failed.

\[
\text{A[} (\text{MainDriver.L1} \&\& \exists (i:\text{int}[0,\text{MAX_RELAYS}-1]) \text{recv_vals}[i] == \text{MainDriver.meta_src_val}) \equiv \text{(forall}(i:\text{int}[0,\text{MAX_RELAYS}-1]) \text{recv_vals}[i] == \text{MainDriver.meta_src_val} ||
\]

\[
\text{recv_vals}[i] == \text{FAILED.VAL})\]

In this experiment, UPPAAL (version 4.0.13) explores a total of 18284 states to verify this property for two relays with default settings such as breadth-first search exploration and conservative space optimization. UPPAAL explores a total of 422902 states in case of three relays. It explores a total of 12062277 states for four relays.

\textsuperscript{4}In UPPAAL, meta variables are not part of the system state. They may be used to perform special calculations without increasing the state space.
CHAPTER 8
RELATED WORK

8.1 Formal Software Engineering

Since the famous book on design patterns, called “Design Patterns: Elements of Reusable Object-Oriented Software” [12], many design patterns have been proposed for different domains. There also exist many patterns on fault-tolerance, real-time computing and networking of the cyber-physical systems [65, 66, 67]. A pattern can be viewed as a design template of the solution to a generic problem. Although useful in enabling the software reuse, the standard practice of the design patterns is not sufficient for cyber-physical systems. These patterns are usually documented in informal languages. Thus, correct instantiations often depend on users’ expertise and interpretation of the application’s context. There have also been many efforts on formal modeling of design patterns [68, 69, 70, 71, 72, 73, 74]. Similar to the PALS pattern, these works also intend to avoid implementation ambiguities by using domain-specific languages and structural analyses. However, these works do not address the formal verification of the target applications. In contrast, we apply the PALS system to reduce both design and verification complexities of real-time distributed applications.

In this work, we use AADL to model and analyze the PALS systems. Synchronous design languages and tools, such as Simulink [75], SCADE [76], and Lustre [77], are also widely used in the cyber-physical system development. However, these languages are only applied in the modeling, simulation and analysis of the software components. Furthermore, the software components in a synchronous design language are originally intended to be only centralized and driven by a global clock. As a result, these techniques, by default, lack the support for architectural-level analysis of distributed software components. Several works have proposed solutions to simulate the asynchronous behavior of the distributed software in a synchronous design language [78, 79]. These works simulate the nondeterministic asynchronous behavior by having sporadic execution
of processes and controlled delivery of the messages. While these techniques are useful in modeling asynchronous software components, we still need complexity-reducing techniques to deal with combinatorial event interleaving and complex interactions in a distributed application. In this respect, the PALS system complements these languages and tools. One can model and verify the logically synchronous behavior of a PALS system in a synchronous design language as we have done in Synchronous AADL and AADL Behavior Annex. In a similar fashion, we then need a correctness-preserving transformation of the synchronous models to execute them on the physically asynchronous architecture.

There are other architectural modeling languages such as SysML [80] and MARTE [81]. These languages and AADL have similar capabilities for system modeling but at different levels of flexibility and expressiveness. The PALS pattern may also be defined in these languages. We note that there are active collaborations between these language communities to develop transformation tools for these languages. Thus, in the future, it may also be possible to translate the PALS system in AADL to other modeling languages.

8.2 Distributed Consensus Algorithms

Distributed consensus is a fundamental concept in distributed systems and theory. Virtual synchronization is one of the early solutions for distributed consensus. Birman and Joseph [82] first introduced the process group abstraction to achieve virtual synchronization for event-triggered computations. This virtual synchrony model guarantees that the behavior of the replicated processes is indistinguishable from the behavior of a single reference process on a non-faulty node. ISIS [83] and its new version ISIS² [51] are two middleware that achieve the virtual synchrony with a group communication service. The group communication service maintains a list of active processes and notifies the process join or crash events, known as view change events. These middleware synchronize the view change events and the application messages in such a way that distributed processes remain consistent.

Horus is another system supporting virtual synchrony [84]. Guo et al. [85] give a lightweight version of this implementation. Pereira et al. [86] use application-level semantics to relax some strong consistency requirements of virtual synchrony. However, these techniques do not provide hard real-time guarantees or timing bound of when a synchronization is completed. Real-time versions of these communication services have been proposed in [87, 88]. For example, Abdelzaher
et al. [88] provide a multicast and membership service for real-time process groups organized in a logical ring. When an application needs to send real-time messages, it presents the message with timing constraints to an admission controller to perform online schedulability analysis. Real-time messages that can be scheduled are admitted. Otherwise, they are rejected.

There are several key differences between the PALS system and the virtual synchronization model implemented in these group communication services. Firstly, these services provide a different level of abstraction to the application developers than the PALS system. These services are primarily used as the network-layer services for reliable and consistent message communications in a group of computations. One can synchronize the computations at individual events such as application messages, membership or view change events with these services. Thus, from an application’s perspective, it makes more sense to use these services with event-triggered or aperiodic computations. Otherwise, the application has to provide the necessary mechanisms for the timed processing of the events. On the other hand, the PALS system achieves a time-triggered, logical synchronization of real-time computations. It coordinates both computations and message communications in an application such that the messages generated during two consecutive PALS clock events are processed consistently at the receiving tasks. Thus, the computations are synchronized at the PALS clock events. Based on the clock synchronization algorithms, these synchronization events happen periodically within a fixed global time interval across the nodes.

Secondly, the group communication services, by default, guarantee reliable multicast of individual application messages. As discussed in Section 6.3.3, even though we can use these services for consistent message communications, the PALS system requires only a minimal protocol for real-time reliable multicast communication. These existing works have more complexity than what we need. For example, the PALS system does not require the ordered delivery of individual messages in a period as long as they are received in separate application ports. The tasks in the PALS system only care whether the messages are delivered reliably or not.

Thirdly, the group communication services bundle various fault-tolerance mechanisms such as group membership, state transfer upon initialization. These are useful for many applications. In this thesis, we do not address these mechanisms in the PALS system. We separate our logical synchronization mechanism from these fault-tolerance mechanisms so that designers can implement the right fault-tolerance mechanisms to meet different reliability requirements of the applications. Implementation of these mechanisms should not be difficult in the PALS system. For example, we
can easily determine the non-faulty computations in a group based on the end-marker events.

There are other well-known consensus algorithms, such as Lamport’s Paxos algorithm [89] and Chandra-Toueg’s algorithm [43]. These consensus algorithms are widely used in distributed transactions, distributed locking protocols, and leader election [90, 91]. These consensus algorithms however do not provide hard real-time guarantees. They assume a globally asynchronous system, which does not provide any bound on message transmission delay, clock drift rate, and execution speed. We note that there is a famous theory on the impossibility of distributed consensus by Fischer et al. [92]. This theory suggests that no algorithm can always reach consensus in a bounded time in this model of asynchronous system, even with a single process crash failure. The main reasoning is that processes cannot correctly differentiate a slow process from a crashed one without a bound on the end-to-end delay. These consensus algorithms circumvents this impossibility of consensus based on concepts such as failure detectors and quorum consistency [93]. Since the PALS pattern assumes a bound on message transmission delay, clock drift rate, clock skew and response time, this theory does not apply in the work of this thesis. In contrast to these algorithms, the PALS pattern can achieve consistency in real-time with significant reduction in complexity.

8.3 Synchronous Lockstep Execution

Our work is also related to the works done by other researchers to implement synchronous model onto different asynchronous architectures such as Loosely Time-Triggered Architecture [19, 94] and Asynchronous Bounded Delay (ABD) network [95, 96]. Tripakis et al. [19] deals as ours with the problem of mapping a synchronous computation on an asynchronous architecture. In their approach, they consider a loosely timed-triggered architecture (LTTA). The mapping is achieved through an intermediate finite FIFO platform (FFP) layer. Although correctness is achieved in spite of unpredictable network delays and clock skews, these approaches do not provide the hard real-time guarantee required for synchronization and consistent views in cyber-physical systems. Furthermore, this work does not handle any failure and multi-rate computations.

The architectural assumptions of the PALS system is also related to those of the ABD network. An ABD network primarily assumes that the message transmission delay is bounded. Chou et al. [95] and Tel et al. [97] give similar protocols to simulate a globally synchronous design on an ABD network with bounded clock drift rate. These works define the logical synchronization period in terms of the round intervals for different network topologies, where each round interval
gives an upper bound on the message transmission delay. However, these works do not assume that a fault-tolerant clock synchronizer synchronizes the local clocks. As a result, these protocols require complex reinitialization procedure to correct the clock drift errors. For example, after a certain number of rounds, these protocols reset the clocks based on the multicast of special “start” messages. In these approaches, the real-time periodic computations of a cyber-physical system may be discontinuous during this reinitialization procedure. Furthermore, none of these works discuss node failure, reliable message communication, and multi-rate computations.

Awerbuch [18] gives three protocols for achieving logical synchronization: $\alpha$ synchronizer, $\beta$ synchronizer, $\gamma$ synchronizer. These synchronizers generate local tick events to execute the synchronous logic, similar to our approach. However, these synchronizers either depend on the acknowledgment messages or a leader node to prevent the arrival of past messages after a tick event. As a result, these solutions require longer synchronization periods and have high overhead to maintain a verifiable leader election logic with respect to failure and other asynchronous events.

Rushby [98] also gives a round-based synchronous execution in a time-triggered architecture. In this synchronous model, each synchronous round or period has two phases: communication and computation. The computation phase begins only after the communication phase has finished. Only the PALS system pattern (with single-rate) is closely related to this model. This work, however, does not support multi-rate executions. There is also a difference with respect to our approach. The PALS pattern does not require the computation phases to complete prior to sending messages. A task can send its messages while it is in the computation phase, which can reduce the required PALS clock period.

8.4 Time-Triggered Architecture

Time-Triggered Architecture (TTA) is one of the earliest system architectures that introduced distributed real-time clock sources for maintaining consistency [99]. The core functions of TTA are implemented in custom network architecture, such as TTP/C [53] and TTEthernet [100], for reliable message communications. In both TTP/C and TTEthernet, the nodes communicate according to a pre-specified Time Division Multiple Access (TDMA) schedule. Hence, every node knows exact message transmission time and has a time window for receiving each incoming message. The hardware in these architectures, such as network guardian and network switch, also maintains the message schedule and detects a faulty node when a message is not received in the allowable time.
window [101]. Correctness of these solutions requires a tight clock synchronization of all nodes, including the network switches. Thus, these network architectures also implement a fault-tolerant clock synchronization algorithm in the hardware [102, 103].

The existing capabilities of these network architectures are sufficient to satisfy the PALS architectural assumptions of bounded end-to-end delay and clock skew. Hence, it is possible to implement the PALS pattern in a system architecture with these time-triggered network architectures.

However, TTA has its own distinctive characteristics that make the implementation of distributed consistency and logical synchronization different from the proposed approach in this thesis. In TTA, the distributed consistency is based on the concept of sparse timebase [104]. In the sparse timebase, TTA controls the send instants of the message transmissions. Messages are transmitted with sufficient time difference so that other nodes can agree on the order of the messages based on the local timestamps of message transmissions. In order to define the timestamps, TTA defines a logical clock whose granularity (i.e. the tick duration) is a function of the maximum clock skew. Based on this clock, TTA can ensure that “the temporal order of events can be recovered from their timestamps, if the difference between their timestamps is equal to or greater than 2 ticks.” [105, p. 62].

While this model provides a simple approach to define the temporal order of messages in TTA, we need more efforts to coordinate the distributed interactions. In particular, TTA does not consider the task response time and the message transmission delay in its logical clock. Therefore, despite the control on message transmissions, variations of these system parameters can increase the verification complexity of the distributed algorithms in TTA.

Steiner and Rushby have recently proposed an extension of the sparse timebase to implement the globally synchronous model in TTA [106]. This approach requires an additional timing layer on top of the original logical clock of TTA so that the synchronization period is sufficiently long to allow for the task response time and the message transmission delay. A PALS clock period in TTA then becomes equivalent to a fixed integer-multiple of the logical clock of TTA. A small caveat: Since the granularity of the logical clock of TTA depends on the maximum clock skew, the PALS clock period may be slightly larger than the expected one.

We also note that the PALS pattern does not have to know the global message schedule of the time-triggered network architectures for the logical synchronization. The pattern only uses the system’s performance parameters to abstract away the underlying network architecture. Thus,
the use of the PALS pattern is similar in any real-time network architecture, whether it is time-triggered or event-triggered. Applications can be reused with minimal overhead when the network architecture is upgraded or modified.

8.5 Other Related Work

8.5.1 Real-time Networking Middleware

Distributed middleware, such as real-time CORBA [107], web-services, publish-subscribe middleware [108], and PALSware provide a virtualized platform for distributed tasks to collaborate. However, the level of abstractions provided by these middlewares are quite different. Real-time CORBA, web services, and publish-subscribe middleware require the developers to be explicitly aware of the asynchronous nature of the distributed nodes. Therefore, the applications on top of these middleware layers should be carefully designed and verified to provide consistency under such asynchronous environments. In contrast, PALSware is a middleware for logically synchronous computations that hides the physically asynchronous clocks and simplifies the distributed algorithms.

8.5.2 Fault-Tolerant System Design

Fault-tolerance is a major design criteria for safety-critical systems [109]. Various techniques have been proposed in different application domains. Triple modular redundancy and pair-pair redundancy are widely used to mask single point failure [110]. Sha et al. proposed a simplex architecture to separate the concerns of effectiveness and reliability for command and control systems [111]. Applications in a process group also use membership algorithms to have a consistent picture of the members’ state in the presence of various faults such as crash failure, message omission, and Byzantine faults [53, 52]. This thesis does not provide a specific fault-tolerance mechanism. However, we have demonstrated that the fault-tolerant solutions can be much simpler with the PALS pattern.

Fault injection is a widely studied mechanism to test fault-tolerance functionalities for both hardware and software [112, 113, 114]. These fault injection tools are capable of injecting a wide range processor, memory, and communication faults and collecting performance and dependability measurements. In this work, we consider a simple fault-injection driver to test the basic fail-stop model using SIGKILL. Future extension may integrate with low-level fault injectors in the future.
8.5.3 Formal Verification of Distributed Algorithms

Formal verifications of distributed consensus algorithms have been investigated in the past [115, 116]. These works show some feasibility of the model checking of distributed consensus algorithms in physically asynchronous architecture that only guarantee bounded message transmission delay. Such architecture is commonly known as partially synchronous distributed systems. These works however do not provide any generic solution for achieving a scalable verification in distributed systems. Researchers have also verified other distributed algorithms, such as distributed convergence, in this architecture. For example, Chandy et al. [117] transform a shared memory architecture to verify the distributed convergence problem in a partially synchronous distributed system. However, the architectural assumptions of the shared memory architecture and the partially synchronous architecture are different from the PALS system. In contrast to these architectures, the PALS system assumes real-time bounds on the various system parameters. Thus, not only that the PALS system reduces the possible non-deterministic interaction scenarios, but also it enables the use of the equivalent globally synchronous design.

In recent years, researchers have also explored the model checking of distributed software [118, 119, 120, 121]. These works consider various heuristics for efficient state-space explorations such as random walk, bounded search, and dynamic partial-order reduction. Despite these optimizations, model checking of distributed algorithms is still extremely difficult beyond a certain model size and complexity.

8.5.4 Other Works on PALS System

In [122], Bae et al. extend the multi-rate PALS pattern. The authors give a mathematical definition of the pattern and proves the bisimulation between a synchronous system and a multi-rate distributed system. This work is complementary to our approach. We emphasize on the engineering aspect of the pattern and apply model-based techniques to analyze the system specification.

In [106], Steiner and Rushby also discuss a correction of the PALS timing constraints by using local clock time measurements and clock drift rate. They normalize the global time based measurements with respect to the maximum clock drift rate $\rho$. For example, they use local clock time measurements of $\mu^{\text{min}}(1 - \rho), \mu^{\text{max}}(1 + \rho)$ for message transmission delays instead of $\mu^{\text{min}}, \mu^{\text{max}}$. In our proposal, we do not have to explicitly handle the clock drift rate. When defining the timing constraints of the PALS system, we always analyze with respect to the global time interval.
that defines the earliest and the latest global time of an event based on the maximum clock skew. Therefore, their corrections are not necessary unless one considers the PALS timing constraints purely in terms of the local clock time. In case of the local clock time, one can normalize the system parameters with respect to clock drift rate as suggested by Steiner and Rushby [106].

Rockwell Collins META toolset [123] is closely related to the AADL framework proposed in this thesis. The META toolset was developed during a DARPA-funded program [11] in which we also collaborated. This toolset supports design transformations, compositional verification, and static analysis for various architectural patterns. With respect to the PALS pattern, designers can instantiate the PALS design specification and validate the assumptions in this toolset. Both our framework and the META toolset perform similar architectural analysis for the PALS pattern. In contrast to the META toolset, we also support architectural analyses for the Synchronous AADL model and the multi-rate PALS model. Our framework also generates the synchronous model from the PALS model.
Components in cyber-physical systems require consistent views and actions in real-time to guarantee safety and correctness. In this thesis, we propose a complexity-reducing architectural pattern and its middleware implementation for achieving consistency in these systems. Our proposed solutions guarantee a logically synchronous abstraction for the real-time distributed computations. In this approach, we can reduce the amount of efforts spent for the distributed system design and verification. Engineers need to design and verify only the simple globally synchronous model as long as the system architecture satisfies the pattern’s assumptions.

Beyond the scope of this thesis, there are several open challenges in the research of the PALS system:

- We currently validate the middleware with a simple fault injection framework. We still have to test the middleware more rigorously to meet the certification requirements of the safety-critical systems. Our group is currently collaborating with the researchers of Software Engineering Institute (SEI), Carnegie Mellon University. We plan to use a software model checker, such as CBMC [124], to verify the PALS applications and this middleware.

- We also have to support the formal verification of the multi-rate PALS applications. We have to extend the Synchronous AADL specification for the multi-rate computations and translate the AADL model to a model checking language for formal verification.

- In many information processing applications, the distributed computations operate in the pipeline over multiple nodes. In these applications, we need to synchronize the components that execute at different phases of the pipeline. We have developed a simple extension of the PALS pattern, in which a component executes in one of the pipeline phases and interact with the neighboring tasks of the following phases [27]. We have to analyze the integration of this extension and the proposed patterns of this thesis.
Based on our experience with the PALS pattern, we believe that formal design patterns can be effective in reducing the complexities of a complex system. For a scalable formal analysis, a system can be designed by composing these design patterns. In this composition, each pattern will provide the necessary abstractions to simplify the design of the subsequent steps. This design style is also suitable for the assume-guarantee compositional reasoning [125, 126]. One can formalize the assumptions to define valid pattern instances so that the pattern guarantees can be preserved in the subsequent design steps. We believe that more research still needs to be done to integrate these patterns in existing formal verification frameworks.
AADL is a modeling language for the cyber-physical system architecture. There are several benefits of using AADL in the system engineering. It helps engineers maintain a logical mapping between the design and the final implementation throughout the development process. During the system integration, engineers can also perform various architectural analyses on the AADL models to validate the design requirements and avoid many integration problems.

In this chapter, we give a brief introduction of AADL. Readers who are familiar with AADL can skip this chapter. For a better understanding of AADL, we recommend the technical reports by Feiler et al. [127] and the AADL standard released by Society of Automotive Engineers (SAE) [24].

A.1 AADL Components and Connections

Engineers can specify both hardware and software components of a system architecture in AADL. In the following, we briefly describe the constructs used to model these components:

1. The software components are specified by the AADL constructs: thread, thread group, process, subprogram, data, etc. A thread represents an executable software component. A thread executes inside a protected address-space modeled by the process construct. A thread can be either periodic or aperiodic. A thread group provides a logical collection of threads, data and other thread groups in a process. A subprogram models a source code function. In AADL, a data construct represents a data type used in the source code.

2. The hardware components are specified by the AADL constructs: processor, memory, device, bus, etc. A processor construct models the processing hardware that executes the threads. A memory construct models the storage for code and data. A device construct is used to model any hardware device such as sensors and actuators. The processor, memory, and devices can be interconnected by a bus construct.
AADL also defines a composite construct, called system to have a hierarchical organization of the hardware and the software components of a subsystem.

In addition to the structural specification, one can specify the behavior of threads and subprograms using an AADL annex\(^1\), called Behavior Annex [29]. AADL Behavior Annex defines states, variables, and state transitions of a thread or a subprogram.

AADL defines different connection semantics for software interactions: data port connection, event data port connection, and event port connection. These connections connect two ports. A port defines a message interface of a software component. These connections have different queuing behavior. For example, when two threads communicate messages using an event data port connection, the messages can be queued at the destination thread. On the other hand, when the threads use a data port connection, the port has a single buffer at the destination thread. The messages are allowed to be overwritten in this buffer. AADL uses an event port connection to send events or notifications to other threads. An aperiodic thread may be dispatched at the arrival of an event at an event port. An event data port connection can also be used to send an event along with a data.

The AADL models have both textual and graphical representations. Figure A.1 gives the graphical representation of the above-mentioned AADL constructs.

![AADL modeling constructs](image)

Figure A.1: AADL modeling constructs.

\(^1\)An annex allows a user to extend the AADL language with specialized notations and modeling capabilities.
A.1.1 Component type and implementation:

In AADL, a component is defined by its type and implementation specification.

**Component type.** The type declares a component’s visible characteristics including its name, features (e.g. ports), properties, and an (optional) extend clause with the parent component’s type.

We give an example of a process’s type declaration in Figure A.2. In this figure, the process type, called **ComputationA**, has two features given by two ports. The internal components of this process communicate with the process’s external components through these ports. The declaration of a port includes its name, connection type, direction, and data type. For example, **msgOut** is an output event data port of **ComputationA**. **ComputationA** transmits an Integer message through this port to other components.

In AADL, one can additionally declare the characteristics or parameters of a component in the AADL property annotations. For example, in this example, we define the period of the computations inside **ComputationA** by using a pre-defined AADL property, called **Period**, which has a value of 20ms. Any periodic thread that executes inside this process has this default rate unless the thread defines its own period property.

![Figure A.2: The type declaration of an example process, ComputationA.](image)

**Component implementation.** An implementation, on the other hand, declares the internal structure of a component, which include its identity, subcomponents, and interactions between the subcomponents. Other declarations include the implementation properties, any refinement of the inherited features and subcomponents, etc.

We give an example process implementation, called **ComputationA.impl**, in Figure A.3. The implementation is identified as `{component type name}.impl`. Thus, **ComputationA.impl** is an implementation of the type **ComputationA**. The internal structure of this example implementation contains two thread subcomponents: **thread1** (thread implementation **ThreadX.impl**)
process implementation ComputationA.impl
subcomponents	hread1: thread ThreadX.impl;
thread2: thread ThreadY.impl;
connections
port msgIn -> thread1.msgIn1;
port thread1.msgOut1 -> thread2.msgIn2;
port thread2.msgOut2 -> msgOut;
properties
Priority => 10 applies to thread1;
Priority => 12 applies to thread2;
end ComputationA.impl;

thread ThreadX
features
msgIn1: in data port Base_Types::Integer;
msgOut1: out event data port Base_Types::Integer;
properties
Dispatch_Protocol => Periodic;
end ThreadX;

thread implementation ThreadX.impl
end ThreadX.impl;

thread ThreadY
features
msgIn2: in event data port Base_Types::Integer;
msgOut2: out event data port Base_Types::Integer;
properties
Dispatch_Protocol => Periodic;
end ThreadY;

thread implementation ThreadY.impl
end ThreadY.impl;

(b) AADL text.

Figure A.3: The implementation declaration of an example process, ComputationA.impl.

and thread2 (thread implementation ThreadY.impl). ComputationA.impl describes the interconnections between these threads and connections with its ports. For example, any incoming data to this process is directly delivered to thread1. Then, the output of thread1 is passed to thread2, which eventually generates the final output of this process. The process implementation abstracts these internal details from the external components. Similar to the type declaration, an implementation may define additional properties or override the property values declared in the component type declaration. In this example, we define the priorities of the thread subcomponents in ComputationA.impl by using the AADL property, called Priority.
APPENDIX B

AADL PROPERTY SETS FOR PALS SYSTEM

B.1 Synchronous AADL Property Set

property set SynchAADL is
    Synchronous: inherit aadlboolean applies to (system, process, thread group, thread);
    SyncPeriod: inherit Time applies to (system, process, thread group, thread);
    Deterministic: aadlboolean applies to (thread);
    IsEnvironment: aadlboolean applies to (thread);
    InputConstraints: list of aadlstring applies to (thread);
end SynchAADL;

B.2 PALS System AADL Property Set

property set PALS_Properties is
    PALS_Id : aadlstring applies to (system, process, thread group, thread);
    PALS_Period : Time applies to (system, process, thread group, thread);
    PALS_Connection_Id : aadlstring applies to (connection, port);
    PALS_Output_Time: Time_Range => 0 ns .. 0 ns applies to (system, process, thread group, thread);
    PALS_Synchronizer_Type: enumeration (Multi_Rate_Synchronizer, Environment_Input_Synchronizer,
                                         Environment_Output_Synchronizer, NOT_SYNCHRONIZER) => NOT_SYNCHRONIZER
                                         applies to (system, process, thread);
    PALS_Base_Component: aadlboolean => false applies to (thread, thread group);
    PALS_Implementation_Component: aadlboolean => false applies to (thread group);
    Computation: PALS_Properties::Supported_Computation applies to (thread, thread group, process);
    Supported_Computation: type enumeration (Multi_Rate_Base_Computation, Synchronous_Computation,
                                             Output_Delay_Computation, Pals_Event_Generator);
    Multi_Rate_Synchronizer_Operation : PALS_Properties::Supported_Synchronizer_Operation =>
                                            Last_Message_Only applies to (port, thread);
    Supported_Synchronizer_Operation: type enumeration (Last_Message_Only);
end PALS_Properties;
This chapter provides the complete Synchronous AADL model of the active-standby system, mentioned in Section 3.3.2. There are 4 AADL packages in this system: **MainModule**, **Side1**, **Side2**, and **Environment**. **MainModule** describes the top-level (main) system. The other packages describe the system-process-thread hierarchy of the two sides and the environment. We use AADLv2 to model this system.

### C.1 Package: Main Module

```plaintext
package MainModule
public
    with Side1;
    with Side2;
    with Environment;
    with SynchAADL;

system ActiveStandbySystem
end ActiveStandbySystem;

system implementation ActiveStandbySystem.impl
subcomponents
    sideOne: system Side1::Side1.impl;
    sideTwo: system Side2::Side2.impl;
    env: system Environment::Environment.impl;
connections
    C1: port sideOne.side1ActiveSide -> sideTwo.side1ActiveSide;
    C2: port sideTwo.side2ActiveSide -> sideOne.side2ActiveSide;
    C3: port env.side1FullyAvail -> sideOne.side1FullyAvail;
    C4: port env.side1FullyAvail -> sideTwo.side1FullyAvail;
    C5: port env.side2FullyAvail -> sideOne.side2FullyAvail;
    C6: port env.side2FullyAvail -> sideTwo.side2FullyAvail;
    C7: port env.manualSelection -> sideOne.manualSelection;
    C8: port env.manualSelection -> sideTwo.manualSelection;
    C9: port env.side1Failed -> sideOne.side1Failed;
    C10: port env.side2Failed -> sideTwo.side2Failed;
```
properties
  SynchAADL::Synchronous ⇒ true;
  SynchAADL::syncPeriod ⇒ 2 Ms;
  Dispatch_Protocol ⇒ Periodic applies to sideOne.sideProcess.sideThread;
  Dispatch_Protocol ⇒ Periodic applies to sideTwo.sideProcess.sideThread;
  SynchAADL::Deterministic ⇒ true applies to sideOne.sideProcess.sideThread;
  SynchAADL::IsEnvironment ⇒ false applies to sideOne.sideProcess.sideThread;
  SynchAADL::Deterministic ⇒ true applies to sideTwo.sideProcess.sideThread;
  SynchAADL::IsEnvironment ⇒ false applies to sideTwo.sideProcess.sideThread;
  Period ⇒ 2 ms;
  Timing ⇒ Delayed applies to C1;
  Timing ⇒ Delayed applies to C2;
  Timing ⇒ Immediate applies to C3;
  Timing ⇒ Immediate applies to C4;
  Timing ⇒ Immediate applies to C5;
  Timing ⇒ Immediate applies to C6;
  Timing ⇒ Immediate applies to C7;
  Timing ⇒ Immediate applies to C8;
  Timing ⇒ Immediate applies to C9;
  Timing ⇒ Immediate applies to C10;
end ActiveStandbySystem.impl;
end MainModule;

C.2 Package: Side1

package Side1
public
  with Base_Types;
  with Data_Model;
  with SynchAADL;

system Side1
features
  side1FullyAvail: in data port Base_Types::Boolean;
  side2FullyAvail: in data port Base_Types::Boolean;
  side2ActiveSide: in data port Base_Types::Integer;
  manualSelection: in data port Base_Types::Boolean;
  side1Failed: in data port Base_Types::Boolean;
  side1ActiveSide: out data port Base_Types::Integer;
end Side1;

system implementation Side1.impl
subcomponents
  sideProcess: process Side1Process.impl;
connections
  port side1FullyAvail -> sideProcess.side1FullyAvail;
  port side2FullyAvail -> sideProcess.side2FullyAvail;
port side2ActiveSide -> sideProcess.side2ActiveSide;
port manualSelection -> sideProcess.manualSelection;
port side1Failed -> sideProcess.side1Failed;
port sideProcess.side1ActiveSide -> side1ActiveSide;
end Side1.impl;

process Side1Process
features
side1FullyAvail: in data port Base_Types::Boolean;
side2FullyAvail: in data port Base_Types::Boolean;
side2ActiveSide: in data port Base_Types::Integer;
manualSelection: in data port Base_Types::Boolean;
side1Failed: in data port Base_Types::Boolean;
side1ActiveSide: out data port Base_Types::Integer;
end Side1Process;

process implementation Side1Process.impl
subcomponents
    sideThread: thread Side1Thread.impl;
connections
    port side1FullyAvail -> sideThread.side1FullyAvail;
    port side2FullyAvail -> sideThread.side2FullyAvail;
    port side2ActiveSide -> sideThread.side2ActiveSide;
    port manualSelection -> sideThread.manualSelection;
    port sideThread.side1ActiveSide -> side1ActiveSide;
    port side1Failed -> sideThread.side1Failed;
end Side1Process.impl;

thread Side1Thread
features
    Dispatch: in event port;
    Complete: out event port;
    side1FullyAvail: in data port Base_Types::Boolean;
    side2FullyAvail: in data port Base_Types::Boolean;
    side2ActiveSide: in data port Base_Types::Integer;
    manualSelection: in data port Base_Types::Boolean;
    side1Failed: in data port Base_Types::Boolean;
    side1ActiveSide: out data port Base_Types::Integer;
properties
    Required_Connection ⇒ false applies to Dispatch;
    Required_Connection ⇒ false applies to Complete;
    Required_Connection ⇒ false applies to side1FullyAvail;
    Data_Model::Initial_Value ⇒ ("true") applies to side1FullyAvail;
    Required_Connection ⇒ false applies to side2FullyAvail;
    Data_Model::Initial_Value ⇒ ("true") applies to side2FullyAvail;
    Required_Connection ⇒ false applies to side1Failed;
    Data_Model::Initial_Value ⇒ ("false") applies to side1Failed;
end Side1Thread;
-- final state needs to be handled in Maude.
-- data types.

thread implementation Side1Thread.impl

annex behavior_specification {

variables
prevSide2ActiveSide : Base_Types::Integer;
prevmanualSelection : Base_Types::Boolean;
states
preInit : initial complete final state;
initState, side1FailedState, side2FailedState, side1WaitState, side1ActiveState,
side2ActiveState : complete state;
initState_tmp, side1FailedState_tmp, side2FailedState_tmp, side1WaitState_tmp,
side1ActiveState_tmp, side2ActiveState_tmp : state;

transitions
preInit -[ on dispatch ]-> initState {
  prevSide2ActiveSide := 0;
  prevmanualSelection := false
};
initState -[ on dispatch ]-> initState_tmp;
initState_tmp -[ side1Failed = true ]-> side1FailedState {
  side1ActiveSide := 0;
  prevSide2ActiveSide := 0;
  prevmanualSelection := false
};
initState_tmp -[ side1Failed = false ]-> side2FailedState {
  side1ActiveSide := 1;
  prevSide2ActiveSide := 0;
  prevmanualSelection := false
};
side1FailedState- [ on dispatch ]-> side1FailedState_tmp;
side1FailedState_tmp -[ side1Failed = false and side2ActiveSide = 0 ]-> side2FailedState {
  side1ActiveSide := 1;
  prevSide2ActiveSide := side2ActiveSide;
  prevmanualSelection := manualSelection
};
side1FailedState_tmp -[ side1Failed = false and side2ActiveSide != 0 ]-> side1WaitState {
  side1ActiveSide := 1;
  prevSide2ActiveSide := side2ActiveSide;
  prevmanualSelection := manualSelection
};
side1FailedState_tmp -[ side1Failed = true ]-> side1FailedState {
  side1ActiveSide := 0;
  prevSide2ActiveSide := side2ActiveSide;
  prevmanualSelection := manualSelection
};
side2FailedState- [ on dispatch ]-> side2FailedState_tmp;
side2FailedState_tmp -[ side1Failed = false and side2ActiveSide = 0 ]-> side2FailedState {
  side1ActiveSide := 1;
  prevSide2ActiveSide := side2ActiveSide;
  prevmanualSelection := manualSelection
};
side2FailedState_tmp -[ side1Failed = false and side2ActiveSide != 0 ]-> side1WaitState {
  side1ActiveSide := 1;
  prevSide2ActiveSide := side2ActiveSide;
  prevmanualSelection := manualSelection
};
side2FailedState_tmp -[ side1Failed = true ]-> side2FailedState {
  side1ActiveSide := 0;
  prevSide2ActiveSide := side2ActiveSide;
  prevmanualSelection := manualSelection
};
C.3 Package: Side2

package Side2
public
with Base_Types;
with Data_Model;
with SynchAADL;

system Side2
features
    side1FullyAvail: in data port Base_Types::Boolean;
    side2FullyAvail: in data port Base_Types::Boolean;
    side1ActiveSide: in data port Base_Types::Integer;
    manualSelection: in data port Base_Types::Boolean;
    side2Failed: in data port Base_Types::Boolean;
    side2ActiveSide: out data port Base_Types::Integer;
end Side2;

system implementation Side2.impl
subcomponents
    sideProcess: process Side2Process.impl;
connections
    port side1FullyAvail -> sideProcess.side1FullyAvail;
    port side2FullyAvail -> sideProcess.side2FullyAvail;
    port side1ActiveSide -> sideProcess.side1ActiveSide;
    port manualSelection -> sideProcess.manualSelection;
    port side2Failed -> sideProcess.side2Failed;
    port sideProcess.side2ActiveSide -> side2ActiveSide;
end Side2.impl;
process Side2Process
features
side1FullyAvail: in data port Base_Types::Boolean;
side2FullyAvail: in data port Base_Types::Boolean;
side1ActiveSide: in data port Base_Types::Integer;
manualSelection: in data port Base_Types::Boolean;
side2Failed: in data port Base_Types::Boolean;
side2ActiveSide: out data port Base_Types::Integer;
end Side2Process;

process implementation Side2Process.impl
subcomponents
sideThread: thread Side2Thread.impl;
connections
port side1FullyAvail -> sideThread.side1FullyAvail;
port side2FullyAvail -> sideThread.side2FullyAvail;
port side1ActiveSide -> sideThread.side1ActiveSide;
port manualSelection -> sideThread.manualSelection;
port side2Failed -> sideThread.side2Failed;
port sideThread.side2ActiveSide -> side2ActiveSide;
end Side2Process.impl;

thread Side2Thread
features
Dispatch: in event port;
Complete: out event port;
side1FullyAvail: in data port Base_Types::Boolean;
side2FullyAvail: in data port Base_Types::Boolean;
side1ActiveSide: in data port Base_Types::Integer;
side2Failed: in data port Base_Types::Boolean;
side2ActiveSide: out data port Base_Types::Integer;
manualSelection: in data port Base_Types::Boolean;
properties
Required_Connection => false applies to Dispatch;
Required_Connection => false applies to Complete;
Required_Connection => false applies to side1FullyAvail;
Data_Model::Initial_Value => ("true") applies to side1FullyAvail;
Required_Connection => false applies to side2FullyAvail;
Data_Model::Initial_Value => ("true") applies to side2FullyAvail;
Required_Connection => false applies to side2Failed;
Data_Model::Initial_Value => ("false") applies to side2Failed;
end Side2Thread;

thread implementation Side2Thread.impl
annex behavior_specification {**
variables
prevSide1ActiveSide: Base_Types::Integer;
}
prevmanualSelection: Base_Types::Boolean;

states
preInit : initial complete final state;
initState, side1FailedState, side2FailedState,
side2WaitState, side1ActiveState,
side2ActiveState : complete state;

transitions
preInit -[ on dispatch ]-> initState {
    prevSide1ActiveSide := 0;
    prevmanualSelection := false};
initState -[ on dispatch ]-> initState_tmp;

initState_tmp -[ side2Failed = true ]-> side2FailedState {
    side2ActiveSide := 0;
    prevSide1ActiveSide := 0;
    prevmanualSelection := false};
initState_tmp -[ side2Failed = false ]-> side1FailedState {
    side2ActiveSide := 2;
    prevSide1ActiveSide := 0;
    prevmanualSelection := false};

side2FailedState -[ on dispatch ]-> side2FailedState_tmp;
side2FailedState_tmp -[ side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
    side2ActiveSide := 2;
    prevSide1ActiveSide := side1ActiveSide;
    prevmanualSelection := manualSelection};
side2FailedState_tmp -[ side2Failed = false and side1ActiveSide != 0 ]-> side2WaitState {
    side2ActiveSide := 1;
    prevSide1ActiveSide := side1ActiveSide;
    prevmanualSelection := manualSelection};
side2FailedState_tmp -[ side2Failed = true ]-> side2FailedState {
    side2ActiveSide := 0;
    prevSide1ActiveSide := side1ActiveSide;
    prevmanualSelection := manualSelection};

side1FailedState -[ on dispatch ]-> side1FailedState_tmp;
side1FailedState_tmp -[ side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
    side2ActiveSide := 2;
    prevSide1ActiveSide := side1ActiveSide;
    prevmanualSelection := manualSelection};
side1FailedState_tmp -[ side2Failed = false and side1ActiveSide != 0 ]-> side2WaitState {
    side2ActiveSide := 1;
    prevSide1ActiveSide := side1ActiveSide;
    prevmanualSelection := manualSelection};
side1FailedState_tmp -[ side2Failed = true ]-> side2FailedState {
side2ActiveSide := 0;
prevSide1ActiveSide := side1ActiveSide;
prevmanualSelection := manualSelection;

side2WaitState - [on dispatch ]-> side2WaitState_tmp;
side2WaitState_tmp - [ side2Failed = false and side1ActiveSide != 0 ]-> side1ActiveState {
  side2ActiveSide := 1;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}

side2WaitState_tmp - [ side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}

side2WaitState_tmp - [ side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}

side1ActiveState - [on dispatch ]-> side1ActiveState_tmp;
side1ActiveState_tmp - [ side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}

side1ActiveState_tmp - [ side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}

side1ActiveState_tmp - [ side2Failed = false and side1ActiveSide != 0 and
(side2FullyAvail = false or side1FullyAvail = true and (prevmanualSelection = true or manualSelection = false)) ]-> side1ActiveState {
  side2ActiveSide := 1;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}

side2ActiveState - [on dispatch ]-> side2ActiveState_tmp;
side2ActiveState_tmp - [ side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}

side2ActiveState_tmp - [ side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;
}
C.4 Package: Environment

package Environment
public
with SynchAADL;
with Base_Types;

system Environment
features
    side1FullyAvail: out data port Base_Types::Boolean;
    side2FullyAvail: out data port Base_Types::Boolean;
    manualSelection: out data port Base_Types::Boolean;
    side1Failed: out data port Base_Types::Boolean;
    side2Failed: out data port Base_Types::Boolean;
end Environment;

system implementation Environment.impl
subcomponents
    envProcess: process EnvironmentProcess.impl;
connections
    C1: port envProcess.side1FullyAvail -> side1FullyAvail;
    C4: port envProcess.side1Failed -> side1Failed;
    C5: port envProcess.side2Failed -> side2Failed;
end Environment.impl;

process EnvironmentProcess
features
    side1FullyAvail: out data port Base_Types::Boolean;
process implementation EnvironmentProcess.impl
subcomponents
  envThread: thread EnvironmentThread.impl;
connections
  C1: port envThread.side1FullyAvail -> side1FullyAvail;
  C2: port envThread.side2FullyAvail -> side2FullyAvail;
  C3: port envThread.manualSelection -> manualSelection;
  C4: port envThread.side1Failed -> side1Failed;
  C5: port envThread.side2Failed -> side2Failed;
end EnvironmentProcess.impl;

thread EnvironmentThread
features
  side1FullyAvail: out data port Base_Types::Boolean;
  side2FullyAvail: out data port Base_Types::Boolean;
  manualSelection: out data port Base_Types::Boolean;
  side1Failed: out data port Base_Types::Boolean;
  side2Failed: out data port Base_Types::Boolean;
end EnvironmentThread;

thread implementation EnvironmentThread.impl
properties
  -- Uninitialized values will be chosen randomly
  SynchAADL::InputConstraints => ("not (s1F and s2F)");
  SynchAADL::IsEnvironment => true;
  Dispatch_Protocol => Periodic;
annex behavior_specification {**
  variables
    s1FA: Base_Types::Boolean;
    s2FA: Base_Types::Boolean;
    mS: Base_Types::Boolean;
    s1F: Base_Types::Boolean;
    s2F: Base_Types::Boolean;
  states
    preInit : initial complete final state;
    s0 : complete state;
  transitions
    preInit -[ on dispatch ]-> s0 {
      s1FA := true;
      s2FA := true;
      mS := false;
      s1F := false;
    }
s2F := false;

s0 -{ on dispatch }-> s0 {
    side1FullyAvail := s1FA;
    side2FullyAvail := s2FA;
    manualSelection := m5;
    side1Failed := s1F;
    side2Failed := s2F};
In this chapter, we give an AADL code snippet of an example of the multi-rate PALS pattern. We apply this pattern in the supervisory control synchronization of a hierarchical control system discussed in Chapter 5.

![Diagram](image)

Figure D.1: Application of the multi-rate PALS pattern on the hierarchical control system.

The top-level AADL diagram of this example system is presented in Figure D.1a. This system consists of a supervisory controller subsystem **SCS**, an aileron servo controller subsystem **ACS**, and a rudder servo controller subsystem **RCS**. In Figure D.1b, we show the internal structure of the system component **RCS**, which contains two rudder servo controllers **RCS1** and **RCS2**, an actuator **RA**, and a sensor **RS**. Similar to our discussion in Chapter 5, both rudder servo controllers receive the setpoint commands, **SpRD1** and **SpRD2**, from the supervisory controllers and the sampled data **RSD** from the sensor. Both **RCS1** and **RCS2** exchange their heartbeat status **Status** for the redundancy management. Output of the rudder servo controllers, **RCD1** and **RCD2**, are passed to the supervisory controller and the actuator. Only the active controller’s output is used to control the actuator.
The architectural models of the pattern instance are shown in Figure D.2a, D.2b, and D.2c. Here we add a multi-rate synchronizer, called \texttt{Synch\_Supervisory\_Control}. The multi-rate synchronizer only affects the data flow of \texttt{SpRD1} and \texttt{SpRD2} that are used in the supervisory control synchronization. The multi-rate synchronizer guarantees that the setpoint commands are processed consistently at the servo controllers in every hyper-period interval. We then form a new thread group with this multi-rate synchronizer and replace the original component in the process element with the new thread group.

Figure D.2: An application of the multi-rate PALS pattern in a hierarchical control system.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig_d2.png}
\caption{A computation component RCT inside the process element of RCS1 (before the pattern is applied).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig_d2b.png}
\caption{A thread group, named RCT\_Group\_Supervisory\_Control, is created with a multi-rate synchronizer and RCT.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig_d2c.png}
\caption{RCT is replaced with RCT\_Group\_Supervisory\_Control after the application of the pattern.}
\end{figure}

\section{Input Model}

In this section, we give the input AADL model of the multi-rate PALS pattern for supervisory control synchronization. In this model, \texttt{Rudder\_Control\_Threads\_servo} is the main in-
put component. It has already been formed after the pattern instantiation of the group, “Rudder_Control”. In this model, we apply the multi-rate PALS pattern for the connections that define PALS_Connection_Id=>"Supervisory_Control” in the process component.

process Rudder_Control_Process
  features
  RSD: in event data port;
  SpRD1: in event data port;
  SpRD2: in event data port;
  RCD: out event data port;
  Status: out event data port;
  Other_Status: in event data port;
end Rudder_Control_Process;

-- Input process implementation.
process implementation Rudder_Control_Process.old
  subcomponents
  RCT: thread group Rudder_Control_Threads.servo;
  connections
  C1: port SpRD1 -> RCT.SpRD2;
  C2: port SpRD2 -> RCT.SpRD2;
  C3: port RCT.Status -> Status;
  C4: port RSD -> RCT.RSD;
  C5: port Other_Status -> RCT.Other_Status;
  C6: port RCT.RCD -> RCD;
  properties
    -- PALS_Connection_Id is used to define the logically synchronous interactions.
    PALS_Properties::PALS_Connection_Id => "Supervisory_Control" applies to C1;
    PALS_Properties::PALS_Connection_Id => "Supervisory_Control" applies to C2;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C3;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C4;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C5;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C6;
    PALS_Properties::Computation => Multi_Rate_Base_Computation applies to RCT;
end Rudder_Control_Process.old;

thread group Rudder_Control_Threads
  features
  RSD: in event data port;
  SpRD1: in event data port;
  SpRD2: in event data port;
  RCD: out event data port;
  Status: out event data port;
  Other_Status: in event data port;
end Rudder_Control_Threads;

-- Input computation component.
thread group implementation Rudder_Control_Threads.servo
subcomponents
  RC: thread Rudder_ServoControl_Thread.impl;
  Synch_Servo: thread Rudder_Servo_SynchThread.impl;
connections
  port RSD -> Synch_Servo.RSD_in;
  port Synch_Servo.RSD_out -> RC.RSD;
  port Other_Status -> Synch_Servo.Other_Status_in;
  port Synch_Servo.Other_Status_out -> RC.Other_Status;
  port RC.RCD -> RCD;
  port RC.Status -> Status;
  port SpRD1 -> RC.SpRD1;
  port SpRD2 -> RC.SpRD2;
properties
  PALS_Properties::PALS_Id => "Rudder_Servo_Control";
  PALS_Properties::PALS_Period => 30 Ms;
  PALS_Properties::PALS_Output_Time => 10 Ms .. 11 Ms;
  Period => 30 Ms;
  Deadline => 24 Ms;
  Priority => 20;
end Rudder_Control_Threads.servo;

thread Rudder_ServoControl_Thread
  features
    RSD: in event data port;
    SpRD1: in event data port;
    SpRD2: in event data port;
    RCD: out event data port;
    Status: out event data port;
    Other_Status: in event data port;
end Rudder_ServoControl_Thread;

thread implementation Rudder_ServoControl_Thread.impl
  properties
    PALS_Properties::PALS_Base_Component => true;
    ...
end Rudder_ServoControl_Thread.impl;

thread Rudder_Servo_SynchThread
  features
    RSD_in: in event data port;
    RSD_out: out event data port;
    Other_Status_in: in event data port;
    Other_Status_out: out event data port;
end Rudder_Servo_SynchThread;

thread implementation Rudder_Servo_SynchThread.impl
  properties
    ...

D.2 Output Model

After the application of this pattern, we create a thread group `Rudder_Control_Threads.supv` and add it as a subcomponent of the process implementation `Rudder_Control_Process.new`. This thread group contains the input thread group and the instantiated multi-rate synchronizer. In the output model, we also define the required timing, scheduling and PALS properties in these instantiated components. In the following, we give the code snippet of the output model:

```plaintext
process implementation Rudder_Control_Process.new
  subcomponents
    RCT_Group_Supervisory_Control: thread group Rudder_Control_Threads.supv;
  connections
    C1: port SpRD1 -> RCT_Group_Supervisory_Control.SpRD2;
    C2: port SpRD2 -> RCT_Group_Supervisory_Control.SpRD2;
    C3: port RCT_Group_Supervisory_Control.Status -> Status;
    C4: port RSD -> RCT_Group_Supervisory_Control.RSD;
    C5: port Other_Status -> RCT_Group_Supervisory_Control.Other_Status;
    C6: port RCT_Group_Supervisory_Control.RCD -> RCD;
  properties
    PALS_Properties::PALS_Connection_Id => "Supervisory_Control" applies to C1;
    PALS_Properties::PALS_Connection_Id => "Supervisory_Control" applies to C2;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C3;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C4;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C5;
    PALS_Properties::PALS_Connection_Id => "Rudder_Control" applies to C6;
end Rudder_Control_Process.new;

thread group implementation Rudder_Control_Threads.supv
  subcomponents
    RCT: thread group Rudder_Control_Threads.servo;
    Synch_Supervisory_Control: thread Rudder_Supervisor_SynchThread.impl;
  connections
    port RSD -> RCT.RSD;
    port RCT.RCD -> RCD;
    port RCT.Status -> Status;
    port Other_Status -> RCT.Other_Status;
    port SpRD1 -> Synch_Supervisory_Control.SpRD1_in;
    port Synch_Supervisory_Control.SpRD1_out -> RCT.SpRD1;
    port SpRD2 -> Synch_Supervisory_Control.SpRD2_in;
    port Synch_Supervisory_Control.SpRD2_out -> RCT.SpRD2;
  properties
    -- Properties of this newly formed thread group.
```

138
PALS_Properties::PALS_Id => "Supervisory_Control";
PALS_Properties::PALS_Period => 120ms;
PALS_Properties::PALS_Output_Time => 8 Ms .. 12 Ms;
Period => 30 Ms;
Deadline => 24 Ms;
Priority => 20;
PALS_Properties::Computation => Multi_Rate_Base_Computation applies to RCT;
PALS_Properties::PALS_Synchronizer_Type => Multi_Rate_Synchronizer applies to Synch_Supervisory_Control;
end Rudder_Control_Threads.supv;

-- New multi-rate synchronizer.
thread Rudder_Supervisor_SynchThread
  features
    SpRD1_in: in event data port;
    SpRD1_out: out event data port;
    SpRD2_in: in event data port;
    SpRD2_out: out event data port;
  end Rudder_Supervisor_SynchThread;

thread implementation Rudder_Supervisor_SynchThread.impl
  properties
    Dispatch_Protocol => Periodic;
    PALS_Properties::PALS_Output_Time => 8 Ms .. 9 Ms;
    Dispatch_Protocol => Periodic;
    Period => 120 Ms;
    Deadline => 10 Ms;
    Priority => 30;
  end Rudder_Supervisor_SynchThread.impl;
In this chapter, we give an AADL model of the multicast relay nodes of the fault-tolerant communication protocol of Section 6.3. As discussed in the protocol description, the source task transmits a message to all relays in the system. The relay nodes then exchange the source message within themselves and deliver to the destination tasks.

The first part of this model is to implement the multicast group of relay nodes. We can easily implement the protocol in AADL by defining components for a given number of relay nodes and the interconnections with the PALS tasks. Figure E.1 gives a graphical representation of the AADL model with two relay nodes (Relay1, Relay2) and two PALS tasks (Task1, Task2). In the original PALS model (Figure E.1a), these PALS tasks send and receive one data in every period. We form an AADL system of type MulticastSystem surrounding the relays (Figure E.1b). Each relay node implements an aperiodic thread to act on the incoming source message or the message from other relays during each multicast operation unless it has failed.

However, a naive implementation for a given number of relays is not sufficient. We must design the model to support its extensions. Especially, a designer must be able to add or remove any relay node from the configuration with minimal changes to the AADL model. In AADLv1, making a simple change like this one is not easy. One has to redefine the interfaces and connections of each relay node and its surrounding hierarchical components.

Fortunately, we can now have a simpler design solution in AADLv2. For subcomponents of identical type and similar connection patterns, AADLv2 now supports two features: array of components and a property called Connection_Pattern that declares the connection pattern to/from these components. For example, in this protocol, we define the relay nodes as an array of process element in MulticastSystem. For a connection from a source PALS task to array of relay nodes, we use the property value One_To_All for the Connection_Pattern property, which intuitively gives that the multicast nature of the connection. Similarly, for the connections from the array of relay nodes.
relay nodes to a destination task, we set the value of Connection_Pattern to All_To_One. For the connections between the relay nodes, we resort to a simpler model. Instead of defining event
data ports for each application message, we define a pair of one input port and one output port for each relay, given by \texttt{fromOtherRelays} and \texttt{toOtherRelays}, respectively. We assume that the message communications between two relays are generic and happen through a single event data port. Both input and output even data ports have the same data type and the size is equal to the maximum of the size of all application messages. We also set the property of \texttt{Connection\_Pattern} for the connections between these relay nodes as \texttt{All\_To\_All}. This helps us model the fact that each relay node communicates all relay nodes, especially when it receives a new message from either the source PALS task or other relay nodes.

Figure E.2 gives the AADL code snippet with an example of array of relay nodes. We define an abstract AADL component, called \texttt{AbstractRelay}, containing the ports: \texttt{fromOtherRelays} and \texttt{toOtherRelays}. The thread and the process components of the relay node can extend this abstract component. For example, we define a process component \texttt{Relay}, with ports for application messages from two PALS tasks. We then use this process to form an array of relay nodes and define the connections in a system implementation, called \texttt{MulticastSystem\_basic}. Later on, we integrate this system component with the PALS tasks to form the final implementation model such as \texttt{ExampleWithRelays\_impl}.

\textit{Note:} In order to support the event-triggered dispatch, we use an \textit{event data port} at the relay nodes. On the other hand, the use of the event data port at the PALS task is optional. It depends on the incoming data rate and the task period. For example, a single-rate PALS system assumes a \textit{data port} for the message communication. In such case, we model the connection end at the PALS tasks with a data port, while the other connection end at the relay nodes still use an event data port. AADLv2 supports different port categories at the connection ends. Such differentiation only affects the queuing behavior at the connection ends. In this scenario at the single-rate PALS system, the data port’s content at a receiving PALS task is overwritten by the incoming data from the relay nodes. The PALS fault model assumes that the source and the relay nodes are fail-stop. Since the proposed protocol guarantees consistency, an overwrite at an input port does not affect the correctness of the application. The receiving PALS tasks process identical messages from a non-faulty source task.
abstract AbstractRelay
  features
  fromOtherRelays: in event data port; toOtherRelays: out event data port;
end AbstractRelay;

process Relay extends AbstractRelay
  features
  inData1: in event data port; inData2: in event data port;
  outData1: out event data port; outData2: out event data port;
end Relay;

process implementation Relay.impl
  subcomponents
    task : thread RelayThread.impl;
  connections
    port inData1 -> task.inData1;
    port inData2 -> task.inData2;
    port task.outData1 -> outData1;
    port task.outData2 -> outData2;
    port task.toOtherRelays -> toOtherRelays;
    port fromOtherRelays -> task.fromOtherRelays;
end Relay.impl;

system MulticastSystem
  features
  inData1: in event data port; inData2: in event data port;
  outData1: out event data port; outData2: out event data port;
end MulticastSystem;

system implementation MulticastSystem.basic
  subcomponents
    allrelays : process Relay [10];
  connections
    inData1ToRelays: port inData1 -> allrelays.inData1
      { Connection_Pattern => ((One_To_All)); };
    inData2ToRelays: port inData2 -> allrelays.inData2
      { Connection_Pattern => ((One_To_All)); };
    relaysToOutData1: port allrelays.outData1 -> outData1
      { Connection_Pattern => ((All_To_One)); };
    relaysToOutData2: port allrelays.outData2 -> outData2
      { Connection_Pattern => ((All_To_One)); };
    CntoOtherRelays: port allrelays.toOtherRelays -> allRelays.fromOtherRelays
      { Connection_Pattern => ((All_To_All)); };
end MulticastSystem.basic;

system implementation ExampleWithRelays.impl
  subcomponents
    task1 : process Node::Task1.impl;
    task2 : process Node::Task2.impl;
    multicastRelays : system MulticastSystem.basic;
  connections
    task1ToRelays: port task1.outData1 -> multicastRelays.inData1;
    task2ToRelays: port task2.outData2 -> multicastRelays.inData2;
    relaysToTask1: port multicastRelays.outData2 -> task1.inData2;
    relaysToTask2: port multicastRelays.outData1 -> task2.inData1;
end ExampleWithRelays.impl;

Figure E.2: AADL model with array of relay nodes.
In this chapter, we give an overview of the basic operations of the clock synchronization algorithms. Generally, the nodes in a clock synchronization algorithm are categorized into two groups: clock masters (reference clock servers) and clock slaves. The masters have access to high-precision reference clocks such as atomic clock, GPS. On the other hand, the slaves use regular clocks. Each node executes a clock synchronizer process. The clock synchronizer periodically computes an estimate of the clock offset $\theta$ based on the round-trip delay $\delta$ between a master and a slave. The slave then adjusts its local clock by using these estimates.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{clock synchronize.png}
\caption{Clock synchronization modes.}
\end{figure}

**Basic operations.** There are two common operational modes, in which the slaves compute the clock offset and the round-trip delay: master-slave mode and broadcast mode. Figure F.2 illustrates these two approaches. The operational mode in a clock synchronizer depends on the network architecture. If the network does not support broadcasts, then the master-slave mode is applied.
In the master-slave mode, a slave node periodically sends a message to a master node requesting the master’s local clock time. Let the slave sends the request at its local clock time $t_1$, and the master receives the message at its local clock time $t_2$. The master replies a message to the slave with the time $t_2$ in its packet. Let the reply message is sent at time $t_3$, and the slave receives the reply at time $t_4$. Based on these 4 timestamps, the slave node computes an estimate of the clock offset relative to the master node [61]:

$$\delta = [(t_4 - t_1) - (t_3 - t_2)];$$

$$\theta = [(t_2 - t_1) + (t_3 - t_4)]/2$$

In the broadcast mode, a master node initiates the message communications. The master broadcasts a sync message at a regular interval to the slave nodes. Suppose that the master node broadcasts the sync message at its local clock time $t_1$, and a slave node receives this message at its local clock time $t_2$. In order to share the send time $t_1$ with the slave nodes, the master node usually broadcasts another (follow-up) message with the time $t_1$ in the packet. In order to estimate the network delay and the offset, the slave nodes infrequently transmits a delay-request message to the master at time $t_5$. The master receives the delay-request message at time $t_6$. In a similar approach as the master-slave mode, the master replies back to the slave with the time $t_6$ in the (delay-response) packet. Thus, when the slave receives this reply at time $t_7$, it becomes aware of 4 relevant timestamps: $t_1, t_2, t_5, t_6$. Based on these timestamps, the slave node computes the clock offset and the round-trip time in the following way: [128].

$$\delta = [(t_6 - t_1) - (t_5 - t_2)];$$

$$\theta = -[(t_2 - t_1) + (t_5 - t_6)]/2$$

The performance of a clock synchronization algorithm depends on many factors. Primarily, the uncertainty in the measurement of the network latency affects the performance. The clock synchronizers often assume symmetric communication channels between the master and the slave nodes. Thus, the asymmetric delay in the communication channels affect the estimation of the one-way delay from the round-trip delay. Furthermore, the computations of the send and receive timestamps are not always accurate, especially when they are done in the software. Inaccuracies
in these computations also affect the performance.

**Byzantine fault tolerance.** The performance of the clock synchronizers also depends on the quality and correctness of the clock of the master node. Existing research works consider different fault models of the master clocks. One common form of fault-tolerance considered in the literature is the Byzantine fault tolerance [129].

In this fault model, the master node can give an arbitrary (wrong) clock time to the slaves. In order to handle the arbitrary clock failure, a slave node estimates its relative clock offset with respect to N master nodes. It uses a “fault-tolerant average”, also called a *convergence function*, on these estimates [63]. For example, a fault-tolerant average of N clock offsets is equal to the average of \((N - 2f)\) clock offsets, after discarding \(f\) highest and \(f\) lowest clock offsets. Here, \(N \geq 2f + 1\) where \(f\) is the maximum number of faulty master nodes.

![Graph showing worst-case clock offset conditions for clocks \(c_1\) and \(c_2\) at global time \(t\). The dashed lines give the slope for worst-case clock drift rates.](image)

Here, \(t_1 = t - \Delta \cdot (1 + \rho)\)
\& \(t_2 = t + \Delta \cdot (1 + \rho)\)

**Relating the clock offset and the clock skew.** The performance of the clock synchronization algorithms is generally defined by the *maximum clock offset* with respect to the global time. Let the maximum clock offset is \(\Delta\), that is \(|c(t) - t| \leq \Delta\) for all global time \(t\).

In the PALS system, we assume that the *clock skew* is bounded. Let the maximum clock skew
is $\epsilon$. If $c(t) = x$, then the global time $t$ happens in the global time interval $[x - \epsilon, x + \epsilon]$.

We can see that there is a subtle difference in the definition of clock skew and clock offset. While the *clock offset* gives the time difference between the local clock time and the current global time, the *clock skew* gives the interval between the current global time $t$ and the global time at which the local clock time is equal to $t$.

We can however easily define the clock skew in terms of the clock offset. The relation between these two parameters is based on the *maximum clock drift rate*. If the maximum clock drift rate is $\rho$, then $\epsilon = (1 + \rho)\Delta$. The maximum clock drift rate is defined below:

$$1 - \rho \leq \frac{c(t_1) - c(t_2)}{t_1 - t_2} \leq 1 + \rho$$

To prove this relationship, consider a worst-case scenario in which any two clocks $c_1$ and $c_2$ have the worst-case relative clock offset at the global time $t$. Suppose $c_1(t)$ and $c_2(t)$ are at the worst condition and equal to $t + \Delta$ and $t - \Delta$, respectively. Figure F.2 illustrates this scenario at the points $x_1$ and $x_2$, respectively.

Let the clocks $c_1$ and $c_2$ are equal to $t$ at the global time $t_1$ and $t_2$, respectively. Thus, if the maximum clock drift rate is $\rho$, then $c_1(t_1) = t$ can happen at an earliest global time \( t - \Delta/(1 - \rho) = t - \Delta(1 + \rho) \) based on the smallest slope $(1 - \rho)$. Similarly, $c_2(t_2) = t$ can happen at a latest global time $t + \Delta(1 + \rho)$. Thus, it easily follows that in the worst-case scenario, $\epsilon = \Delta(1 + \rho)$. 

147
REFERENCES


[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


