Distributed Co-Simulation Framework for Hardware- and Software-In-The-Loop Testing of Networked Embedded Real-Time Systems

DISSERTATION
zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften

vorgelegt von
Tobias Pieper, M. Sc.

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2020

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

Betreuer und erster Gutachter
Prof. Dr. Roman Obermaisser
Universität Siegen

Zweiter Gutachter
Prof. Dr. Uwe Brinkschulte
Johann Wolfgang Goethe-Universität Frankfurt

Vorsitzender der Promotionskommission
Prof. Dr. Kristof Van Laerhoven
Universität Siegen

Tag der mündlichen Prüfung: 05.10.2020

Abstract

Today's complex control systems such as trains, aircraft or cars are typically composed of multiple networked components which are developed by geographically distributed manufacturers. During their development process, integrating and testing the components are central steps. However, the manufacturers' locations complicate the process since the components must be shipped to a central place and intellectual property must be protected. Using a distributed co-simulation framework which supports Software- and Hardware-In-The-Loop (SIL/HIL) testing can solve these issues. It enables a virtual integration and testing of the components via the Internet and protects intellectual property if it operates on a network-centric abstraction level. In this case, it focuses on the data exchange between the components and knowledge about their internal implementation is not required.

In today's state-of-the-art, a framework that operates on a network-centric abstraction level and supports co-simulation, SIL and HIL testing together is not available yet. Besides that, HIL testing involves hardware devices with real-time requirements. Connecting those devices via public wide area networks such as the Internet, the accuracy of distributed HIL tests is limited by the determinism of the network's communication delays. The available frameworks are mainly based on Quality of Service mechanisms such as differentiated services. However, the communication cycles of the System Under Test (SUT) might be smaller than the guaranteed latencies which leads to deadline misses. Hence, delay-management mechanisms are required which ensure a timely forwarding of input data.

This thesis proposes a distributed co-simulation framework which operates on a network-centric abstraction level and supports the above mentioned techniques. It synchronizes the components of the SUT, coordinates their data exchange and includes fault-injection to validate the dependability. By providing a generic component interface, heterogeneous simulation tools and physical devices are supported. The main contributions of the thesis are two delay-management mechanisms based on state-estimation and speculative execution. The first mechanism forwards estimated inputs to the component if (I) received inputs are delayed or (II) as intermediary inputs. This reduces the number of communication activities inside the framework. The second mechanism divides a simulation setup into several subsets. Those subsets execute independent tasks in advance to forward data to real-time devices in time. Using the mechanisms, the framework is able to connect simulations, software algorithms and real hardware via public communication networks while maintaining the SUT's real-time requirements. Hence, there is no need to make all simulation models or physical prototypes centrally available.

The evaluation using a distributed control application demonstrates the scalability of the framework. The time to execute a simulation setup increases linearly with the simulated time and is bounded by the growth of the component's number in larger setups. Furthermore, the evaluation shows the advantages of the delay-management mechanisms for distributed real-time tests. After determining a proper real-time configuration of the simulation host, the state-estimation mechanism can be used for a timely forwarding of inputs to the components. Using intermediary packets improves the accuracy of distributed real-time tests and makes it independent from the network delays. While the speculative execution enables real-time tests locally and in Local Area Networks, networks with larger delays (e.g., the Internet) require less stringent temporal requirements of the SUT. From a performance point of view, both mechanisms achieve significant speedups depending on the setup's size, the network topology and the communication period.

Zusammenfassung

Heutige komplexe Steuerungssysteme wie Züge, Flugzeuge oder Autos bestehen in der Regel aus mehreren vernetzten Komponenten, die von geografisch verteilten Herstellern entwickelt werden. Während ihres Entwicklungsprozesses sind die Integration und das Testen der Komponenten zentrale Schritte. Die Standorte der Hersteller erschweren jedoch den Prozess, da die Komponenten an einen Ort geliefert werden müssen und Intellectual Property geschützt werden muss. Die Verwendung eines verteilten Co-Simulations-Frameworks, das Software- und Hardware-in-the-Loop (SIL/HIL) Tests unterstützt, kann diese Probleme lösen. Es ermöglicht eine virtuelle Integration sowie das Testen der Komponenten über das Internet und schützt Intellectual Property, wenn es auf einer netzwerkzentrischen Abstraktionsebene arbeitet. In diesem Fall konzentriert es sich auf den Datenaustausch zwischen den Komponenten und Kenntnisse über ihre interne Implementierung sind nicht erforderlich.

Nach dem heutigen Stand der Technik ist ein Framework, das auf einer netzwerkzentrischen Abstraktionsebene arbeitet und Co-Simulation, SIL- und HIL-Tests unterstützt, noch nicht verfügbar. Außerdem werden beim HIL-Test Hardware-Geräte mit Echtzeitanforderungen eingesetzt. Wenn diese Geräte über öffentliche Wide Area Networks wie das Internet miteinander verbunden werden, ist die Genauigkeit verteilter HIL-Tests durch den Determinismus der Netzwerklatenzen begrenzt. Die verfügbaren Frameworks basieren hauptsächlich auf Quality of Service Mechanismen wie z.B. Differentiated Services. Die Kommunikationszyklen des Systems Under Test (SUT) können jedoch kleiner als die garantierten Latenzen sein, was zu Deadline-Überschreitungen führt. Daher sind Mechanismen zur Verwaltung von Verzögerungen erforderlich, die eine rechtzeitige Weiterleitung der Eingangspakete gewährleisten.

Diese Dissertation schlägt ein verteiltes Co-Simulations-Framework vor, das auf einer netzwerkzentrischen Abstraktionsebene arbeitet und die oben genannten Techniken unterstützt. Es synchronisiert die Komponenten des SUT, koordiniert ihren Datenaustausch und beinhaltet Fehlerinjektionsmechanismen zur Validierung der Zuverlässigkeit. Durch die Bereitstellung einer generischen Komponentenschnittstelle werden heterogene Simulationswerkzeuge und physikalische Geräte unterstützt. Die wichtigsten Beiträge dieser Dissertation sind zwei Mechanismen zur Verwaltung von Verzögerungen, die auf State-Estimation und spekulativer Ausführung basieren. Der erste Mechanismus leitet vorhergesagte Pakete an die Komponente weiter, wenn (I) empfangene Pakete verzögert werden oder (II) als zusätzliche Pakete zwischen den Empfangenen. Dadurch wird der Nachrichtenaustausch innerhalb des Frameworks reduziert. Der zweite Mechanismus teilt die Simulation in mehrere Untergruppen auf. Diese Untergruppen führen unabhängige Tasks im Voraus aus, um Daten rechtzeitig an Echtzeitgeräte weiterzuleiten. Mit Hilfe der Mechanismen ist das Framework in der Lage, Simulationen, Software-Algorithmen und reale Hardware über öffentliche Kommunikationsnetze zu verbinden und gleichzeitig die Echtzeitanforderungen des SUTs zu erfüllen. Es ist daher nicht notwendig, alle Simulationsmodelle oder physikalische Prototypen zentral zur Verfügung zu stellen.

Die Auswertung mit einer verteilten Steuerungsanwendung demonstriert die Skalierbarkeit des Frameworks. Die Zeit für die Ausführung einer Simulation steigt linear mit der simulierten Zeit und ist durch das Wachstum der Anzahl der Komponenten in größeren Setups begrenzt. Darüber hinaus zeigt die Auswertung die Vorteile des Verzögerungsmanagements für verteilte Echtzeittests. Nach der Bestimmung einer geeigneten Echtzeitkonfiguration des Simulations-PCs kann der State-Estimation-Mechanismus für eine rechtzeitige Weiterleitung von Paketen an die Komponenten verwendet werden. Die Verwendung von zusätzlichen Paketen zwischen den Empfangenen verbessert die Genauigkeit von verteilten Echtzeittests und macht sie unabhängig von den Verzögerungen im Netzwerk. Während die spekulative Ausführung Echtzeittests lokal und in Local Area Networks ermöglicht, erfordern Netzwerke mit größeren Verzögerungen (z.B. das Internet) weniger strikte zeitliche Anforderungen an das SUT. Aus Performance-Sicht erreichen beide Mechanismen je nach Größe des Setups, der Netzwerktopologie und der Kommunikationsperiode erhebliche Speedups.

Danksagung

Diese Dissertation wurde während meiner Arbeit als Wissenschaftlicher Mitarbeiter am Lehrstuhl Embedded Systems der Universität Siegen erstellt. Die Grundlagen hierzu entstanden im Rahmen des europäischen Forschungsprojekts *Safe4RAIL*, welches durch das Horizon 2020 Programms für Forschung und Innovation der Europäischen Union unter Grant Agreement Nummer 730830 gefördert wurde.

Zuerst möchte ich mich ausdrücklich beim Inhaber des Lehrstuhls Embedded Systems, Professor Dr. Roman Obermaisser, bedanken. Dieser Dank gilt besonders für die Betreuung der Dissertation, die Anstellung als Wissenschaftlicher Mitarbeiter sowie das Vertrauen, mir die Leitung des Arbeitspakets *Virtual Placement in the Market* des Safe4-RAIL Projekts zu übertragen. Des Weiteren möchte ich mich bei Professor Dr. Uwe Brinkschulte für die Erstellung des Zweitgutachtens bedanken. Professor Dr. Kristof Van Laerhoven danke ich für die Leitung des Promotionsverfahren und Professor Dr. Roland Wismüller für seine Funktion als weiterer Prüfer.

Ein weiterer großer Dank gilt meinen Kollegen am Lehrstuhl Embedded Systems. Besonders bedanken möchte ich mich bei Hongjie Fang, Daniel Onwuchekwa und Maryam Pahlevan für die umfangreichen und regelmäßigen Diskussionen im Rahmen des Safe4-RAIL Projekts sowie bei Michael Schmidt für unsere gemeinsame Forschung über Echtzeit-Linux und seine Nutzung für Mixed-Criticality Systeme. Diese Forschung hat die Echtzeitkonfiguration der PCs erst möglich gemacht. Des Weiteren danke ich Stefan Otterbach für die Anschaffung und Bereitstellung der notwendigen Hard- und Software sowie allen Kollegen für die vielen fachlichen und persönlichen Gespräche. Darüber hinaus danke ich allen Partnern des Safe4RAIL Projekts für die vielen fachlichen Diskussionen und die gute Zusammenarbeit im Arbeitspaket.

Dr. Regina Fuchs danke ich aus zweierlei Hinsicht. Einerseits für das Korrekturlesen der Dissertation, andererseits für die vielen Stunden, die wir während meines Studiums an der Entwicklung einer Mess- und Regeleinrichtung für Adhäsionsmessungen mittels Nanoindenter gearbeitet haben. Während dieser Zeit konnte ich eine Menge an praktischer Erfahrung sammeln und sie hat maßgeblich zu der Entscheidung beigetragen, ein

Promotionsstudium zu absolvieren.

Abschließend bedanke ich mich bei meiner Familie und meinen Freunden für jegliche Unterstützung und Motivation, vor allem in Zeiten von Zweifel, schlechter Laune und Motivationslosigkeit. Ohne diese Unterstützung hätte ich mich niemals voll auf das Studium und die Promotion konzentrieren können. Vielen Dank!

Contents

List of Figures

List of Tables

Acronyms

AVB Audio/Video Bridging

DetNet Deterministic Networking

DiffServ Differentiated Services

ETB Ethernet Train Backbone

ECN Ethernet Consist Network

ECU Electronic Control Unit

FCR Fault-Containment Region

FMI Functional Mock-up Interface

FMU Functional Mock-up Unit

GALT Greatest Available Logical Time

HIL Hardware-In-The-Loop

HLA High Level Architecture

HMI Human Machine Interface

HVAC Heating, Ventilation and Air Conditioning

IntServ Integrated Services

LAN Local Area Network

MPLS Multiprotocol Label Switching

MTTF Mean Time To Failure

OMT Object Model Template

PID Proportional, Integral and Derivative

PTP Precision Time Protocol

PWM Pulse Width Modulation

QoS Quality of Service

RSVP Resource Reservation Protocol

RTAI Real-Time Application Interface

RTI Runtime Infrastruture

RTOS Real-Time Operating System

SIL Software-In-The-Loop

STL Standard Template Library

SUT System Under Test

TDMA Time Division Multiple Access

TSN Time Sensitive Networking

TSO Timestamped Order

TMR Triple Modular Redundancy

VPN Virtual Private Network

V&V Verification and Validation

WCET Worst-Case Execution Time

WCCOM Worst-Case Communication Delay

1. Introduction

1.1. Motivation

Nowadays, embedded systems can be found almost everywhere: in transportation, factory automation, smart buildings or grids, robotics and in health care [Mar18, p. 4ff.]. In 2011, about 98% of all microprocessors were embedded. Using sensors and actuators, they are able to interact with their environment and the connection via communication networks establishes large-scale distributed embedded real-time systems [aca11, p.5]. Typical examples of such complex systems are aircrafts [SP13], trains [JGSJS18] or a modern car. The latter can contain 100 connected Electronic Control Units (ECUs) which control several physical processes [WCAF15].

Typically, the connected control systems are developed by several, geographically distributed manufacturers. For trains as an example, the German ASC GmbH and Frauscher Sensortechnik GmbH provide different types of sensors, the Interautomation Deutschland GmbH concentrates on passenger WiFi or infotainment, Knorr-Bremse develops braking systems and the Konvekta AG is supplier for Heating, Ventilation and Air Conditioning (HVAC) systems [Int19]. Besides them, UniControls a.s. from the Czech Republic provides network units or I/O systems. Regarding the automotive domain, examples of suppliers are Robert Bosch GmbH (e.g., electronics), Denso Corp. (e.g., powertrain control), Continental AG (e.g., advanced driver assistance), Infineon Technologies AG (e.g., micro-controllers) or Brose Fahrzeugteile GmbH (e.g., mechatronic systems) [Cha18].

During the development process, the components are designed separately, integrated once they are available and tested afterwards. The final system is built incrementally by repeating the steps [BST10, p. 233-236]. In this process, verification and validation are major steps, which are even more important if the system under test is safety-critical. Common examples of such systems are trains, aircrafts and parts of a car. Without a valid safety argument, human life might be endangered while using those systems [SMS+13]. Besides this, validating a system early in the development process prevents issues in later stages [BCPS11]. However, the distributed locations of the different suppliers complicate the process. To integrate and validate the components, they must be shipped to a central place which introduces costs and causes delays. Furthermore, intellectual property must be protected [HLV06].

Using techniques such as distributed co-simulation, Software- and Hardware-In-The-Loop (SIL, HIL) testing can simplify the development process. They are common technologies which enable a repeatable and controllable component testing. Since many simulation tools are developed for dedicated purposes (e.g., Riverbed Modeler for communication networks and MATLAB for controlled plants), they are coupled to co-simulate complex systems [CDF$^+$14]. Coupling them via the Internet or Local Area Networks (LANs), it is further possible to connect geographically distributed tools. In SIL testing, models of control algorithms are replaced by their software implementation and validated against models of the rest of the system [BCPS11]. This technology supports an early identification of design faults. Furthermore, physical prototypes are not yet required which results in less costs and no risk of damage or accidents [BVP10]. The validation of physical system components is performed in HIL testing instead. If the tested hardware is a real-time device, its temporal requirements must also be satisfied during the validation process. In a setup distributed via the Internet, the network's best effort character introduces indeterministic communication delays into the test system. While there are no consequences on a System Under Test (SUT) without any real-time components (e.g., SIL setups), the delays might lead to deadline misses in real-time components of a HIL test. Hence, delay-management mechanisms are required which (I) detect the delays and stop the test and (II) cope with the unpredictability of the communication. Finally, fault-injection during HIL and SIL testing allows to investigate a system's behavior in case of faults. Since the injection can be performed in a deterministic way, it is a widely used and effective validation technique.

The difficulties caused by the manufacturers' distributed locations can be addressed if the systems under test are integrated via time-triggered real-time communication networks. In such networks, the instants of all communication activities are a priori known [EBK03]. Using the instants as events in a co-simulation helps to synchronize the time advance of distributed simulations. Furthermore, instants for message reception denote the deadlines for real-time hardware at which the packets have to be delivered. With this knowledge, it is possible to mitigate delays in the test system.

This thesis introduces a framework which validates the component integration on a network-centric abstraction level. Since it focuses on the data exchange via the linking network interfaces, the components' internal implementations need not be known during the integration and validation process. By distributing the framework across LANs and the Internet, the components can be validated at their manufacturers' sites during early development steps. In this way, shipping and integrating the components at a central place is not required and intellectual property can be protected. Furthermore, it facilitates the interplay between the manufacturers which expedites the development process and reduces costs.

As shown in this thesis, there is a lack of generic, network-centric frameworks which cover the entire validation process including co-simulation, SIL and HIL testing. If they do so, they suffer from suitable delay-management technologies which enable real-time tests via the Internet or do not consider the injection of faults.

1.2. Objectives and Contribution of this Thesis

The framework developed in this thesis aims on validating the behavior of subsystems in time-triggered networked control systems. Those systems are common for safety-critical domains such as avionics, railway or automotive, but can be found also in other areas. Their safety-critical character introduces research objectives which can be grouped into five different categories. They are defined in the following.

Distributed Co-Simulation Framework The first objective is the development of a co-simulation framework which is distributed via heterogeneous communication networks such as LANs or the Internet. In this way, it is possible to connect simulations between geographically distributed locations, for example between sites of different manufacturers. The distributed character and the transmission of potentially confidential data requires the inclusion of suitable security mechanisms. To validate the communication between the components, the framework operates on a network-centric abstraction level. Hence, the data exchanged between the simulations are the packets sent via the components' network interfaces. Using a scalable synchronization mechanism, the data exchange and the time advance of the simulations need to be coordinated in the correct temporal order. At last, the framework shall provide a generic interface to connect a large number of simulation tools.

Software- and Hardware-In-The-Loop testing Simulations are replaced gradually by software applications and the real hardware during the development process. Hence, the developed framework shall support Software- and Hardware-In-The-Loop testing. The latter involves the execution in real-time wherefore the framework must ensure a timely data exchange and the synchronization of the logical simulation time and the physical time of the components in the HIL test. Furthermore, an interface is required which is able to capture and forward network traffic.

Real-time simulation hosts Connecting real hardware requires a communication between the framework and the devices in real-time. Hence, also the connected simulation hosts on which the framework is running have to provide real-time guarantees. In this way, the packets can be forwarded to the hardware in time.

Delay-management for distributed real-time tests In addition to real-time execution of the simulation host, the data transfer inside the framework must be performed in time. However, heterogeneous communication networks such as the Internet introduce indeterministic communication jitter and packet loss. Even if QoS mechanisms are applied, it is not able to provide the level of determinism required by safety-critical applications. The distributed co-simulation framework shall provide mechanisms which cope with delays leading to deadline misses as follows: (I) a detection mechanism which stops the simulation and (II) mechanisms which ensure a timely input reception for real-time devices.

Fault-injection support Fault-injection is an effective way to validate the behavior of a system in the case of faults. Failures in components of a distributed SUT typically propagate to other subsystems via faulty messages. Hence, the framework shall support a mechanism to inject faults into the communication between the subsystems of the SUT. In this way, failures of the components can be mapped to the communication network. Since the communication may be safety-critical, the injection has to follow related standards.

Within this thesis, a distributed co-simulation framework is proposed which covers all these objectives. It operates on a network-centric abstraction level and supports Software- and Hardware-In-The-Loop testing. Furthermore, a fault-injection mechanism is included which covers all message errors defined in clause 7.4.11 of part two of the IEC 61508 standard for Functional safety of electrical/electronic/programmable electronic safety-related systems [iec11]. Using the standard, the faults are injected directly into the packets exchanged. The main contributions of this thesis are three delay-management technologies which ensure real-time testing via the Internet. The first approach uses state-estimation as a fall-back solution for a timely packet reception. By estimating intermediary packets, the second approach forwards packets to the components in time independent from the framework's communication delays. Finally, a third mechanism distributes the ensemble of components into subsets so that independent processes can execute speculatively in advance. In this way, Internet-introduced delays can be mitigated to maintain the deadlines in hard real-time devices.

1.3. Outline

The remainder of this thesis is organized as follows.

Chapter 2 starts with an overview about basic knowledge regarding topics covered in this work. At first, it denotes the characteristics of distributed real-time systems as the SUTs for which the distributed co-simulation framework is developed. Afterwards, an overview about a typical development and validation process is given followed by example technologies for validation. Those are distributed co-simulation, SIL and HIL testing and fault-injection. Since the framework exchanges potentially confidential data, the chapter closes with a discussion about available security mechanisms.

Chapter 3 continues with an analysis of the current state-of-the-art. In addition to related work regarding the topics introduced in the previous chapter, further technologies are presented. Those are mechanisms to transform Linux into a real-time operating system and to realize Quality of Service via the Internet. Additionally, state-estimation mechanisms and speculative execution as alternative delay-management techniques are depicted. Based on the state-of-the-art analysis, the research gap for this work is defined at the end of the chapter.

Chapter 4 introduces the developed network-centric co-simulation framework. It covers the overall architecture and the simulation bridges as the main components. In the subsequent sections, their functionality is described in detail with regard to SIL and HIL testing. This includes the configuration, synchronization and packet handling mechanisms together with the injection of faults. Finally, the application of VPNs to secure the communication within the framework is explained.

Chapter 5 covers the three delay-management techniques developed. Two of these techniques use state-estimation while the third one is based on speculative execution. They target on enabling distributed real-time tests via the Internet.

Chapter 6 focuses on the evaluation of the distributed co-simulation framework and the delay-management mechanisms. Before evaluation results are presented, a suitable configuration for a real-time Linux simulation host is determined. Furthermore, a fault-tolerant and scalable fan-control application is introduced as the SUT.

Chapter 7 concludes the thesis. In addition to a summary of the main contributions, it depicts potential aspects for future work.

2. Fundamentals

2.1. Real-Time Systems

There are several characteristics of distributed real-time systems the developed framework grounds on. This includes time and the synchronization of clocks, composability, component-based design and determinism. Besides this, real-time communication, real-time operating systems and the property of temporal accuracy are important aspects. All of them are introduced in the following.

2.1.1. Characteristics of Real-Time Systems

In real-time systems, the correctness of a task does not only depend on the correctness of the logical result, but also on its availability at a defined point in time [Sch05, p.40]. This point in time is called a deadline. If it is not met, there are different consequences depending on the type of the real-time system. There are soft, firm and hard deadlines. For soft deadlines, the result may be useful even if the deadline is missed. If the result looses its usefulness instead, the deadline is called firm. When missing a deadline results in catastrophic consequences on the system's environment, the deadline is called hard [Kop98b].

According to the above definitions, real-time systems can be classified as two types. Systems without any hard deadline are called soft real-time systems. If there is at least one hard deadline that has to be met, we speak about hard or safety-critical real-time systems. The latter have to guarantee a temporal behavior in all specified load and fault conditions [Kop11, p.3]. Hence, determinism is a very important aspect during design and test. Examples for soft and hard real-time systems are video and audio streaming where missing a deadline only results in reduced quality (soft), and the flight control system in an aircraft (hard) [Sch05, p.40]. If this system fails in meeting the deadline, it might cause the aircraft to crash. Hence, it is classified as a hard real-time system.

Computer systems may not provide their correct functionality throughout the entire operation. Avizienis et al. [ALRL04] provide a detailed overview about the reasons for those deviations and the resulting consequences in their paper. Typically, a fault causes an error in the state of the system. Categories of faults are for example development and operational faults, software and hardware faults, or natural and human-made ones. If the error causes the system's provided service to deviate from the specified one, one speaks about a failure of the system. Those failures can be classified in their domain (temporal, content), detectability, consistency or in their consequences. Without any countermeasures, a failure might propagate causing a fault in another subsystem. This may continue the chain and cause further errors and failures.

The concepts of faults, errors and failures are covered in dependability which includes reliability, maintainability, availability, safety and security [Mar06, p.2]. The first aspect, reliability, describes the probability that a system will not fail over a defined period of time under specified environmental conditions [Mul85]. If a failure occurs, maintainability represents the time interval which is required to repair the system. Both aspects influence the availability of a system. It is defined as the temporal fraction the system provides its service [ALR+01]. If the reliability is low, the system fails often and must be repaired. Similarly, the system cannot be used if it takes much time to repair it after a failure occurred. Standards such as DO-178C [do111], IEC 61508 [iec10] or ISO 26262 [ISO11] group critical failures into dedicated categories, for example the Safety Integrity Levels 1-4 in IEC 61508. The reliability against those modes is considered as the safety of a system [ALR+01]. Finally, security covers the aspects of confidentiality, integrity and authenticity of data. It is important to consider them, because faults could be introduced intentionally [ALRL04]. For example, the message errors listed in IEC 61508-2 clause 7.4.11 might be caused by malicious components or users.

Handling failures is an important requirement in safety-critical systems. Those systems are used for safety-critical purposes having a direct impact on their controlled environment [Mar06, p.2]. The associated Mean Time To Failure (MTTF) accounts to better than 10^9h [LH94]. Since normal hardware can achieve only an order of 10^4h up to 10^5h, fault-tolerance mechanisms are required [Kop11, p.272]. They use additional information (e.g., parity bits) or redundant hardware and software which detect and correct errors and failures [Sch05, p.182]. To realize this, the redundant components can be grouped into so-called Fault Containment Regions (FCRs). Those regions are defined as correctly operating subsystems unconcerned from any external fault. Using FCRs can prevent the propagation of faults to components outside the region, but the faults can manifest as erroneous data which is propagated. Hence, redundant FCRs are required to realize error containment [LH94]. A widely used concept is Triple Modular Redundancy (TMR). Here, a component is triplicated and a voting mechanism determines the correct value [LV62]. The tolerated faults of a system are defined in the fault hypothesis [HS95]. Faults that are not covered require a so-called never-give-up strategy which brings the system back into a safe state [Kop11, p.154]. Systems where such a state can be reached upon the occurrence of a failure are called fail-safe. However, there are systems like the flight control of an aircraft where safe states are impossible. Those systems must be fail-operational and provide a minimum level of service to prevent a catastrophe [KKS$^+$15].

Internally, the execution of tasks and the communication with other systems is initiated by two different control mechanisms. The first type is called event-triggered. Events are state changes in the controlled object or activities within the computer system. One example is the completion of a task [Obe11, p.20]. If such an event occurs, the CPU is activated by an interrupt and has to schedule the task which handles the event. This requires a dynamic scheduling mechanism [Kop11, p.17]. Event-triggered systems exclude events caused by the progress of time. The control mechanism based on this type of events is called time-triggered. Each task or communication activity is released by a predefined tick of a global clock. In a distributed real-time system, this requires a synchronized, global time [Kop98b]. Typically, time-triggered activities repeat periodically [EBK03].

The delay-management technologies developed in this thesis (cf. Chapter 5) concentrate on time-triggered systems. Without delay-management, event-triggered systems are also supported (cf. Chapter 4).

2.1.2. Time and Clock Synchronization

According to [Kop92], real-time can be modeled by a dense and totally ordered set of instants which are arranged on an directed time-line. In an ordered set, two instants p and r occur either simultaneously or mutually precede each other. To call the ordered set dense, there must be at least one instant q between p and r which is not equal to r.

The interval between the instants p and r is called duration. In real-time systems, events occur at instants of the time-line without having a duration [Kop07]. Since there is no order relation between events arising at the same instant, an event set is only partially ordered [Lam78]. To order them, their delivery and causal order are of interest. The latter is used in a sequence of alarms to identify the primary event. It is more than the temporal order since an event e_1 must not be the reason for an event e_2 which happens later. Otherwise, the temporal order is required since the subsequent event e_2 cannot be the cause of e_1 [Kop92]. The delivery order is not related to time and causality. It ensures only the same perceived order of a set of events between all nodes in a distributed system [Kop11, p. 53].

Time in real-time systems is measured using physical, digital clocks. They contain a counter which increments based on a periodic event. This event is caused by an oscillation mechanism and is called a micro-tick. An important characteristic of a clock is its granularity which is the duration between two successive micro-ticks. In time measurements, the granularity leads to a digitalization error wherefore it must be small. Measuring its value is only possible with a clock having a finer granularity. This can be reached with reference clocks. These clocks typically have a very small granularity and they are perfectly aligned with the international time standard. At runtime, an observer queries the state of the counter to create a time-stamp once an event occurs. They can be used to measure time intervals or to order events [Kop07].

Oscillators differ in quality and price wherefore clocks drift apart [KAH04]. The value of the clock drift represents the frequency ratio between a clock and its reference. It can be calculated by dividing the granularity of the clock z by the number of micro-ticks n^k in the reference clock during this interval. Subtracting the drift by one and taking the amount provides the drift rate ρ (cf. Equation 2.1). A perfect drift rate of 0 is impossible, but it can be bounded by a maximum rate. This value typically lies in the order of magnitude of 10^{-2} to 10^{-7}, but it can also be less if better oscillators are used [Kop11, p. 54f.].

$$\rho_i^k = \left| \frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k} - 1 \right| \tag{2.1}$$

Since every clock has a different drift rate, an ensemble of clocks must be synchronized. There are two possibilities to reach this. In internal synchronization, the precision of the clocks in the ensemble is bounded while external synchronization reduces the accuracy between a clock and a reference [KO87]. Both values are based on the offset between two clocks. It is determined using a reference clock which takes the time-stamps of one of the clocks' respective micro-ticks. The amount of their difference represents the offset. In a set of clocks, the precision denotes the maximum offset between all pairs of clocks during an interval of interest. Similarly, the accuracy represents the maximum offset with regard to the reference clock [Kop11, p. 56].

As clock drift arises continuously, internal synchronization must be repeated periodically after an interval R_{int}. This process is shown in Figure 2.1. The resulting offset at the end of the resynchronization is represented by the convergence function Φ while the drift offset Γ denotes the maximum deviation of two clocks. It depends on the specified drift rates of both clocks and the resynchronization interval. The synchronization mechanism must ensure that the sum of the convergence function and the drift offset is always smaller than the precision Π. Otherwise, a clock will leave the interval [KO87]. The convergence function further defines the correction of the clocks. State-correction adapts the current time value of a clock but leads to discontinuities in the time-base. Rate-correction prevents this issue as the clock speed is adjusted either by adapting the number of micro-ticks per macro-tick or by changing the oscillator voltage [Kop11, p. 72f.]. This mechanism can also be used in external synchronization. A time gateway receives the current time from an external server periodically and adjusts the other clocks in the system [KKMS95].

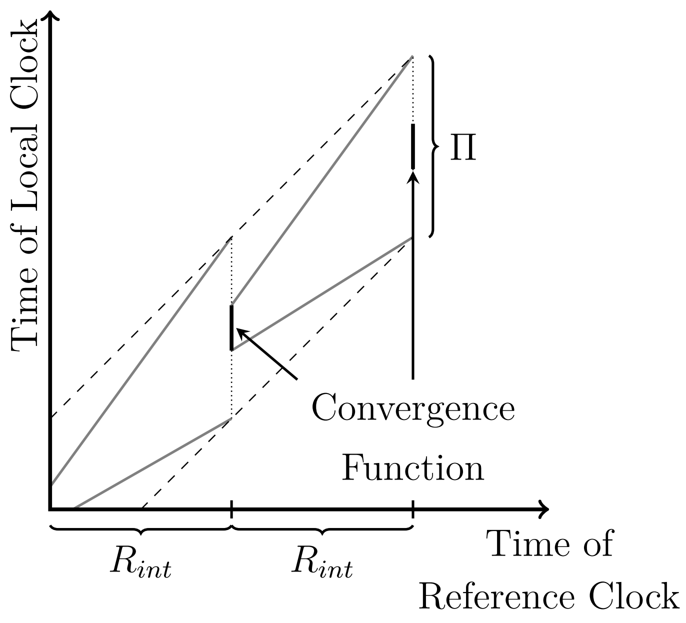


Figure 2.1.: Periodic resynchronization with regard to [KO87].

In distributed real-time systems, most nodes have an oscillator. Since it is not possible to synchronize those clocks perfectly, the notion of a global time was introduced. It is an approximated abstraction which uses selected micro-ticks of an internally synchronized clock set of precision Π. These ticks are called macro-ticks and the nodes use them to create a local implementation of the global time. The best achievable synchronization error is defined by the reasonableness condition. It bounds the maximum difference between global time-stamps of an event by at most one tick and the synchronization error by less than the granularity of a macro-tick [Kop98a]. As a consequence, it is not possible to reconstruct the temporal order of two events whose global time-stamps differ by one tick. To reconstruct them, the time-stamps must differ by at least two ticks and the clocks must satisfy the reasonableness condition. Furthermore, the true duration of an interval is bounded by the observed duration plus/minus twice the granularity of the global time. Only reducing the global time's granularity can improve this fundamental limitation [Kop11, p. 59f.].

The limitation explained before grounds in the usage of a dense time base. Here, events are entitled to occur at every instant of the time-line [Kop92]. However, it is possible to restrict their occurrence to active intervals of duration ϵ which have an interval of silence denoted as Δ in between. The resulting sparse time base is shown in Figure 2.2. Events in the same ϵ-interval are considered to occur simultaneously. Hence, a consistent temporal order can be reconstructed if the events occur in consecutive ϵ-intervals [Kop07] and Δ is greater than three times the granularity [Kop95]. The architecture must ensure that significant events (e.g., sending a message or observing the environment) occur only in ϵ-intervals. Otherwise, an agreement protocol must be used which requires additional resources (e.g., time, bandwidth, etc.) [KAGS05]. Another benefit of the Δ-intervals is the possibility to identify the state of the system clearly. The state represents all values of variables, etc. and separates future activities from the past [Kop11, p. 85]. During the Δ-intervals, there are no activities performed and the state can be determined. Using a sparse time base is one central aspect to reach determinism in distributed real-time systems [Kop07]. Since determinism in replicated components enables fault-tolerance, composability is introduced in the following.

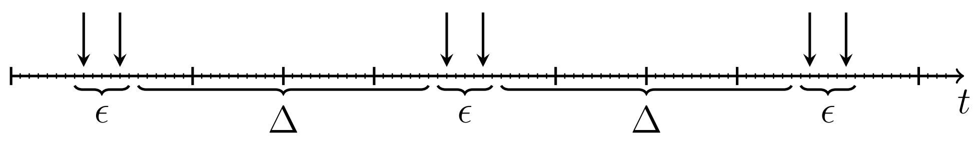


Figure 2.2.: Sparse time-base (adapted from [KAGS05, Kop07]).

2.1.3. Composability

Large systems can be built from prefabricated components that are connected via standardized interfaces. In this way, knowledge about a component's internal design or implementation is not required as long as the component's behavior at the interface is correct [Kop98a]. In real-time systems, the temporal correctness is as important as the correctness of computational results. Since there is no possibility to assign temporal capabilities to a software, it requires a virtual or real hardware unit for execution [Obe11, p. 28]. Figure 2.3 illustrates an example system which consists of two connected clusters, a physical plant and a Human Machine Interface (HMI). The different components and interfaces are introduced in the following.

In distributed computer systems, a component must be separated from the communication infrastructure. This infrastructure provides a unidirectional message exchange from a sending component as unicast (one receiver) or multicast (multiple receivers) during a defined time-interval. Since the message transfer is unidirectional, error propagation from a faulty receiver to a correct sender is avoided. This property is very important in fault-tolerant systems, similar to multicasting [Kop11, p. 80f.]. Multicasting supports a timely correct reception of a single message by multiple receivers. This is required for active redundancy and fault-tolerance [KAGS05].

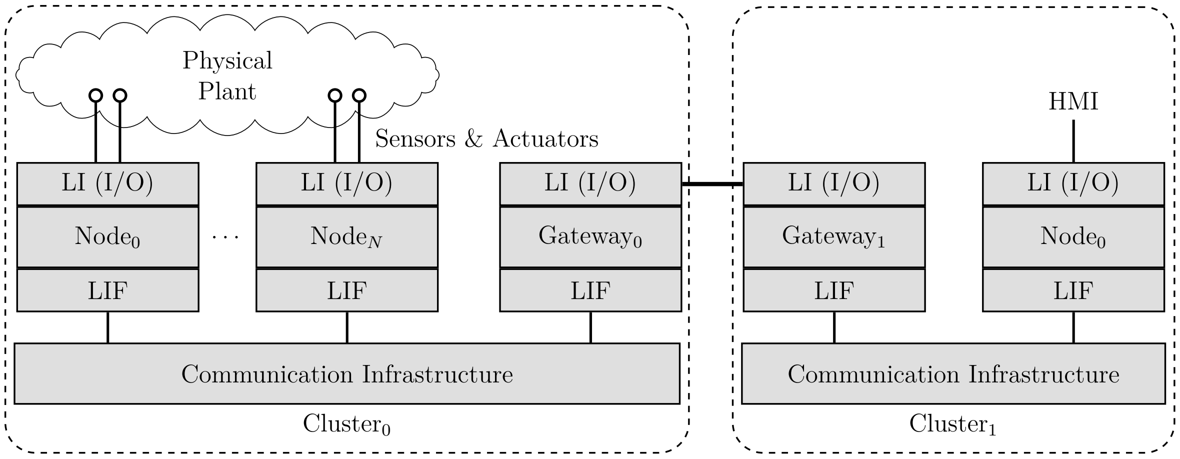


Figure 2.3.: Connected clusters (adapted from [Obe11, p. 7], [Kop11, p. 81]).

Messages exchanged via the communication infrastructure consist of a header, a data field and a trailer. The header contains delivery information like the destination address or how the message has to be handled. Besides this, the data field contains the message's content. Each message has to be delivered in its whole. Since the trailer contains checksums or electronic signatures, a corruption and the authenticity of the message can be checked. If it is not received completely or the trailer does not match, it is discarded. The transport delay of a message is given by the time interval between the instants for sending and receiving it. To prevent congestion of the receiver, it is possible to constrain the rate messages are sent with [Kop11, p. 88f.]. This is realized using the so called minimum inter-arrival time which has to elapse before another message instance can follow [AOA16].

A message can be either time-triggered or event-triggered. The latter type is sent with regard to an event that occurred in the system. It contains information about this event and it is unique, wherefore a receiver must process all messages to determine the new state [Kop99]. Error detection can only be performed by the sender as the receiver does not know whether an event occurred. Hence, it requires an explicit acknowledgement. As the sender must decide whether the transmission failed within a defined real-time interval, fault-tolerant systems cannot be built on systems that are not aware of the progression of real-time [Kop11, p. 91]. In contrast, time-triggered messages are sent periodically at a defined point in time and contain information about the system's state [KS03]. Using the temporal knowledge, congestion can be avoided and the receiver is able to detect errors. If the receive instant elapsed without any reception, a failure occurred in the sender or the message is lost in the communication infrastructure [Kop11, p. 91].

There are four types of interfaces that components provide [Kop11, p. 92]:

1. Linking Interface (LIF)
2. Technology-Independent Control Interface (TICI)
3. Technology-Dependent Debug Interface (TDDI)
4. Local Interface (LI)

The linking interface provides the specified service and connects the component to other components building a cluster (cf. Figure 2.3). It is message-based and must be independent from the internal implementation of the component. Otherwise, it is not possible to develop components independently [Kop02a]. The technology-independent control interface is used to configure and control the component. Arriving messages are handled by the hardware itself or by the operating system, but not by the application. In this way, the application and the overall complexity of the node can be simplified [Kop11, p. 94]. Via the technology-dependent debug interface it is possible to debug the component. It is intended for the component's developer who can observe the state of variables and it is irrelevant for the user [KS03]. Finally, the local interfaces connect the component with its environment [Kop02a] as shown in Figure 2.3. Besides sensors and actuators in the physical plant, this can be a human operator or another computer [Kop11, p. 95].

Components containing a local and a linking interface are called gateway components. At cluster level, only the timing and the semantic content of the local interface are important. Hence, its technology can be exchanged without any influence on the linking interface [Kop11, p. 95f.]. Gateway components link two different systems which may have varying data representations, data semantics or component interactions. Hence, a gateway has to adapt the different technologies used between the systems [Kop98a].

The integration of components into a cluster requires four principles. First, the component shall be developed independently. To reach this, it must define the specification of the linking interface precisely. This covers the interface data structures in the value and time domains and a conceptual interface model of the provided service. Second, the component's LIF must provide the specified service and the integration must not change the functionality. Besides this, the component is not allowed to interfere with communication activities of other components. Finally, a component must be replica deterministic to support fault-tolerance by replication [KO02]. With these principals, it is possible to integrate the components building clusters. Those clusters can be connected further via gateway components to develop large systems [Kop98a].

2.1.4. Determinism

According to Bunge [Bun17, p. 7], a machine is called deterministic if it runs regular and reproducible. As a consequence, determinism enables the prediction of future system states and outputs based on an initial state, the sequence of inputs and the progress of time. With this prediction, it is easier to understand the system's real-time behavior and testing is simplified. In addition, it enables fault-tolerance by replication [Kop08a].

A desired feature in a system with replicated components is replica determinism. If those components start in the same state and receive the same inputs, they produce the same outputs in approximately the same time. Using this property, it is possible to mask failures of a component in a properly designed fault-tolerant system [Kop95]. A failure occurred if one replicated component does not work according to its specification. In this case, the system requires time to correct the failure using the outputs of the correctly working components. How much time it takes must be deduced from the application dynamics and can be bounded by the precision of the global time in time-triggered systems. To ensure the correct behavior, it is important to prevent failures caused by the software. Only random, physical faults are allowed to occur [Obe11, p. 34].

Realizing (replica-)determinism in distributed real-time systems requires several aspects. First, all components must agree on a consistent initial state and they must receive inputs simultaneously [Obe11, p.33]. Similarly, an agreement is required on the inputs of the components. Due to digitalization errors, these inputs might differ in the temporal and value domain [Pol94]. Using a dense time base, events can be observed in a different order which leads to a significantly diverging behavior. A sparse time base prevents these errors and enables the simultaneous provision of inputs to all components [Kop07]. Furthermore, it facilitates the determination of a consistent state [Kop11, p. 128].

The second requirement covers certain computations. This includes arbitrary values such as random numbers or the protection of shared resources via non-deterministic synchronization mechanisms [Pol94]. Those mechanisms should be avoided. The same accounts for preemptive scheduling. Here, a task may be interrupted due to an external event leading to a deadline miss [Kop11, p. 129f.]. Non-preemptive scheduling prevents this issue. If it cannot be used, all possible task preemptions must be considered during the design [Obe11, p.37].

Finally, the instants when messages are delivered should be known. Using a time-triggered communication mechanism, all instants of the message delivery are defined in advance. In addition, the temporal order of all messages equals at sender and receiver and there are no arbitrary delays. Both aspects are further requirements [KAGS05].

2.1.5. Real-Time Communication

Components in real-time embedded systems exchange data via communication networks. The characteristics of real-time systems explained in Section 2.1.1 impose various requirements on those networks. This section presents the requirements and introduces typical traffic types afterwards.

The stability of feedback control loops depends on the time between reading a sensor value and performing an actuation. In networked control systems, this time is influenced by the network delays and their variation [KÖC+94] which is called jitter. Another big influence on the control's quality is the precision of the clock synchronization mechanism. Hence, communication delays and jitter should be minimized and the synchronization mechanism must provide a global time base with a proper precision [Kop11, p. 168]. Furthermore, real-time behavior of the communication system and known jitter are required to comply with the deadlines of the system [Kop00].

Further important properties are reliability and determinism. Reliability includes the transmission of messages via replicated communication channels [GK91] and robust encoding with error-detecting or even correcting codes. As retransmission in case of a failure increases the jitter significantly, it should be avoided. However, the communication system should detect the faulty behavior of components and report it [Kop11, p. 168ff.]. In addition, it should prevent the propagation of temporal faults between the components. To realize this, temporal firewalls can be used. They exploit common knowledge about the transmission instants to block faulty messages [Kop98b]. A deterministic message transfer covers a correct message order on all channels and replicated messages must arrive at simultaneous instants [Kop08b].

In distributed real-time systems, the standard communication type is a multicast as the same data is required often by multiple components. One example is fault-tolerance using replication. Hence, the communication network should support this type [GK91]. Due to frequent changes in the configuration, it must be possible to add components dynamically. This must not require changes and retesting of existing ones. Furthermore, a congestion of the available bandwidth must be avoided [Kop11, p. 171].

Communication channels are often shared resources as the amount of costs and weight due to wiring can be reduced. Multiplexing enables the transmission of multiple signals via the same channel [TW11, p. 125]. The technology considered in this thesis is Time Division Multiplexing, also called Time Division Multiple Access (TDMA). Every data stream receives the entire bandwidth periodically for a dedicated amount of time. The streams must be synchronized and small timing variations can be compensated using guard times [TW11, p. 135]. Time-triggered networks are based on this technology which provides a deterministic transmission, low latency and low jitter. Additionally, different traffic types such as time-triggered, rate-constrained and best-effort traffic can be sent via the same network [SLK+12]. Those types are depicted in Figure 2.4 and explained in the following.

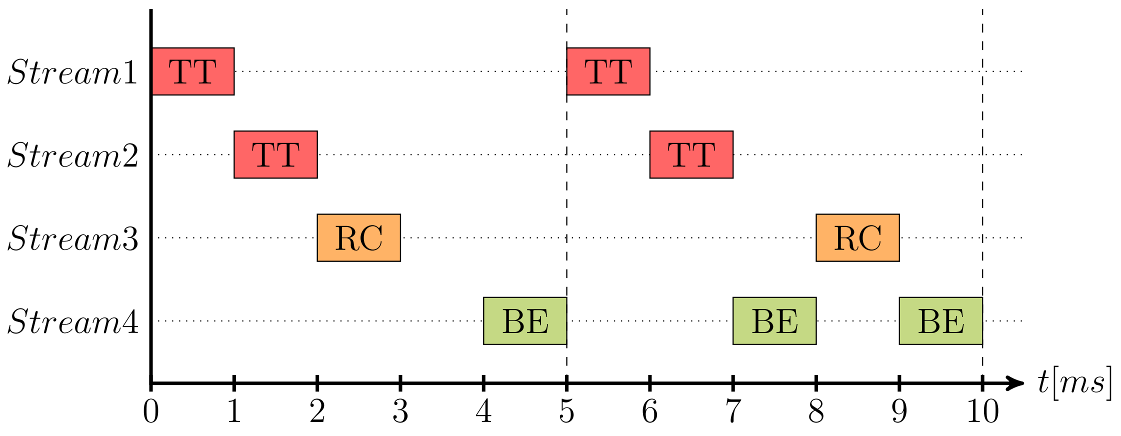


Figure 2.4.: TDMA with time-triggered, rate-constrained and best-effort data streams.

The first traffic type introduced is called Best Effort (BE, colored in green). Whenever a significant event occurs at the sender, the related output message is buffered until the communication system is ready to forward it. The receiver buffers the message in an input queue and consumes it later [Kop08b]. Both buffers can overflow if the network is congested or the sending rate is larger than the receiver's reception rate. These overflows have to be handled using an event-triggered protocol [Kop91]. Without any limitations, it is impossible to provide temporal guarantees. All senders may send messages at the same instant overloading the network. Countermeasures would be buffering, delaying packets at the sender's site or dropping packets. All of them are not suitable for real-time data, wherefore the following to traffic types were developed [Kop11, p. 178].

Rate-Constrained (RC) communication systems (colored in orange) guarantee a maximum bandwidth including a bounded transport latency and jitter. To provide the guarantees, the bandwidth parameters of each application are predefined. In contrast to time-triggered traffic, rate-constrained communication is not deterministic as it does not follow a synchronized time-base [Bel11]. It is possible that some systems exceed their bandwidth as long as other systems are not congested. In this case, the communication system sends the additional messages as best effort. However, the senders are forced to delay their packets if the network cannot handle the increased traffic. In this way, temporal error detection and the protection from babbling idiot can be provided. Typically, there is less traffic compared to the assumed peak [Kop11, p. 180f.].

Finally, Time-Triggered (TT) communication (red-colored streams) is based on an a priori known and periodically repeated communication schedule. Each time-triggered message is sent at a dedicated phase in every period. Using the known schedule, the communication system can assign the required resources to those messages wherefore they are sent in known intervals and without any collisions [Kop08b]. Furthermore, it provides determinism since a sparse time base ensures a consistent temporal order of all messages [Kop07]. The time base can be realized using a synchronized global time which bounds the communication jitter to its precision. Typically, it lies in the sub-microsecond range. An alternative to this approach is using the period of a single leading process. This process establishes a basic period from which all other cycles must be derived [Kop11, p. 184].

2.1.6. Real-Time Operating Systems

Operating systems are used to manage the resources of a computer system and to provide a simplified interface for developing applications. It is realized as an additional software layer on top of the bare hardware [TB15, p. 1].

The management of resources covers scheduling, access to memory and input/output devices, protection and synchronization of resources between tasks and interprocess communication. Herein, Real-Time Operating Systems (RTOSs) must take care of a predictable execution [SR04]. This is different from general-purpose operating systems which focus typically on a maximum throughput and fairness [Ler05]. Section 3.4 introduces approaches which try to provide real-time characteristics to standard operating systems. The aim is to exploit comforts from the standard OS like graphical user interfaces, file systems or standard APIs while providing real-time capabilities [Mar18, p. 203f.].

Resources in embedded systems are usually limited. Hence, it should be possible to remove services from the operating systems which are not required. The concept of a micro-kernel provides a configurable operating system according to the hardware resources and needs of the application. Typical elementary functionalities are interprocess communication, synchronization and a few functions for task management. The latter cover creation, activation, blocking and termination of tasks. All remaining functionalities can be implemented in additional modules based on the elementary services. Benefits are an improved usage of limited resources, scalability and portability. Furthermore, the time required to execute system calls is reduced and the micro-kernel is mostly preemptive. This improves timeliness and determinism. As a disadvantage, the short duration of system calls results in many context switches between user and kernel mode. Since less functions are executed in the protected kernel-mode, the protection of resources is decreased further [WB05, p.

346ff.]. However, some embedded systems are designed for a special purpose and tested to be reliable which is why protection is not always required [Mar18, p. 200].

A task is the sequential execution of program code controlled by the operating system. It contains functions and variables and can be divided into several threads [WB05, p. 350]. These threads share the task's resources and can be executed in parallel. In this way, they can communicate via shared memory instead of exchanging messages as interprocess communication. As a disadvantage, threads can interfere with each other which has to be prevented using synchronization mechanisms [Wil12, p. 5f.].

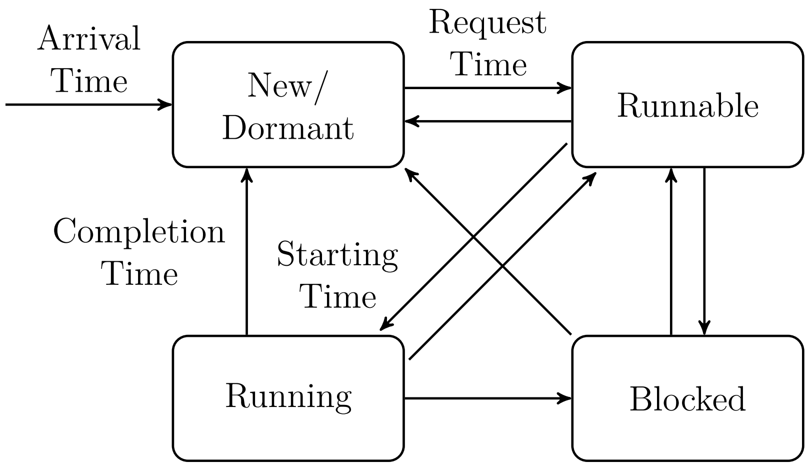


Figure 2.5.: Task states and temporal parameters (adapted from [WB05, p. 352]).

A task/thread can be new (dormant), ready to run, running or blocked. In the blocked state, tasks are waiting for an event to execute. This can be an input or the release of a resource by another task [Sta12, p. 117ff.]. Each task has the following temporal parameters. They are illustrated in Figure 2.5 together with the task states. At the arrival time, the task is created (new) but dormant. It becomes runnable at the request time and starts its execution once it is scheduled (starting time). The time difference between them is called reaction time or release jitter as it is variable. After termination, the task returns to the dormant state (completion time) and can be activated again during its next period (if the task is periodic) [WB05, p. 354ff.]. Important parameters for real-time tasks are the deadline and the execution time. The latter represents the time required to complete an action and is typically bounded by a value called Worst-Case Execution Time (WCET). A guaranteed WCET can be used to schedule tasks deterministically [AB04]. Finally, slack denotes the difference between the completion time and the deadline. It can be reused to schedule tasks with a lower priority dynamically [LB05]. Besides the already mentioned periodic tasks, there are aperiodic ones. If there is a minimum separation between their request times and if they have a hard deadline, aperiodic tasks are called sporadic [SSL89].

Scheduling mechanisms take care of assigning the CPU to runnable tasks. In an RTOS, they must guarantee to maintain all deadlines as long as the task-set is schedulable. This requirement must be analyzed by a scheduling analysis which also figures out if a suitable schedule exists and if it can be found [GR04]. There are different types of scheduling algorithms: dynamic/static and (non-)preemptive scheduling. A dynamic scheduler calculates the schedule and assigns the next task to the CPU at runtime introducing an overhead. In contrast, a static schedule is defined at design time and stored in a table. A dispatcher uses this table at runtime to assign the CPU to the tasks. In complex systems, static scheduling is often the only way to reach predictability. Using preemptive scheduling it is possible to suspend a task at runtime [Mar06, p. 128ff.]. Hence, it is used in case of long execution times or if high-priority tasks request their execution. Non-preemptive scheduling does not provide this capability. It is suitable for short execution times or if a task shall not be interrupted [Kop11, p. 240].

Supporting concurrent tasks causes the problem of synchronization. Once those tasks share common resources such as data or devices they are called dependent [WB05, p. 378f.]. Without any synchronization mechanisms, accessing the resources might lead to race conditions or deadlocks. Race conditions occur in situations where multiple tasks read and write the same data and the final result depends on the order of the read/write operations [CMS01]. Deadlocks occur when multiple tasks wait for resources that are held by each other instead [CMH83]. There are two concepts to solve this issue. The first is mutual exclusion where a resource is granted to one task exclusively until it releases the resource. The second solution is to synchronize the sequential order of accessing the resource [WB05, p. 379f.]. Both concepts can be realized using condition variables, monitors and semaphores. Condition variables allow blocking on a condition that has to be met while a monitor is a module providing mutually exclusive functions to access its variables. Semaphores are integer variables which provide atomic functions to increment/decrement the counter. Only if the value is greater than zero, access can be granted [Din89]. A binary semaphore is called a mutex [TB15, p. 132]. All these constructs require a context switch once a task tries to access a locked resource. Hence, they can lead to a loss of replica determinism [Kop11, p. 226]. Using a static schedule solves this issue as most concurrent activities can be preplanned and prevented [Kop02b].

Even if real-time communication and RTOSs can provide guarantees for delays, jitter and execution times, there are cases in which those guarantees are not sufficient to realize a proper accuracy of real-time data. State-estimation as one solution for this issue is presented in the next section.

2.1.7. Temporal Accuracy and State-Estimation

The usage of state-estimation is motivated by the problem of the temporal accuracy of real-time data. This data type is useful only during a limited time interval and the system would fail if the accuracy is not given. State-estimation is one possibility to extend the interval in which real-time data is valid [KESHO07].

Relevant state-variables in a computer system or in its environment are denoted as real-time entities [Kop98b]. Observing the value of an entity results in an atomic data set including the entity's name, the time of the observation and the value. The latter is the entity's state or an event [EBK03]. Since observations are also events, event information describes the difference of the old and the new state [Kop02a]. In a distributed system without global time base, determining the time of the observation can be impossible. The missing global time base makes the provided time-stamp meaningless (cf. Section 2.1.2) and the time of the message arrival cannot be taken instead. The reason is imprecision caused by network delays and the unknown communication jitter. This imprecision reduces the quality of the observation [Kop11, p. 113f.].

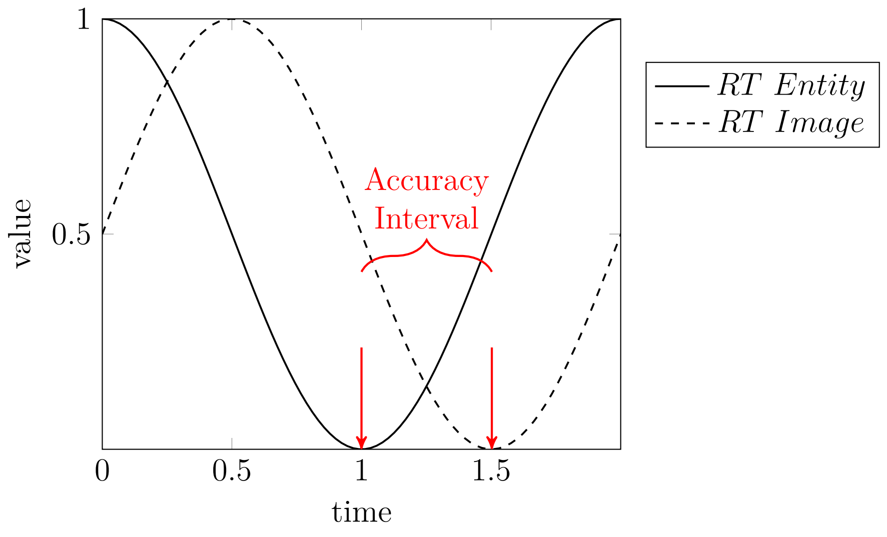


Figure 2.6.: Temporal delay and accuracy interval (adapted from [Kop11, p. 117]).

Real-time entities are represented by real-time images. Those images are valid only during a dedicated time interval and if the value is correct [Kop98b]. To construct them, it is possible to use observations of states and events or the mechanism of state-estimation [Kop11, p. 115f.]. The temporal relationship between entity and image is called temporal accuracy. Accuracy is given if the deviation between the value of the image and the entity is bounded for every instant during a so called accuracy interval. This interval represents an ordered set of instants at which the entity was observed and its length is given by the entity's dynamics. Due to transmission delays between the observing and the receiving nodes, the image lags behind its entity causing an error in the image. Figure 2.6 shows an accuracy interval of 0.5 time units between the real-time entity (solid line) and its image (dashed line) as an example. In a properly designed system, the worst-case error's order of magnitude is similar to the error in the measurement [Kop98c].

A phase-aligned transaction consists of a sender (observer) and a receiver (actuator) task. Its duration can be calculated by adding the WCETs of the tasks and the Worst-Case Communication Delay (WCCOM) required to exchange the data. All those values are known in a time-triggered system. The sum equals the time difference between observing and using a real-time image. If the required temporal accuracy interval is smaller than this duration, state-estimation is required to realize a sufficient accuracy [Kop11, p. 118].

Using state-estimation, the probable state of a real-time entity can be determined at a selected future instant. To reach this, an estimation model is executed periodically and the image is updated based on the result. One important aspect is a close agreement between entity and image at the instant when an output must be provided to the system's environment [KESHO07]. To build an adequate model, it must be possible to represent the entity's behavior by a known process. Otherwise, state-estimation cannot be used [Kop11, p. 121]. In some cases it is possible to use the first derivative of a continuous and differentiable equation while other cases require a more detailed mathematical model. However, state-estimation is a powerful technique to improve the accuracy between a real-time entity and its image if the model describes the system's behavior properly [Kop98c].

2.2. System Design and Validation

After presenting characteristics of the developed systems, this section focuses on the development process itself. It covers the design of embedded real-time systems and their verification and validation.

2.2.1. Design of Real-Time Systems

The development process of real-time systems comprises several, incrementally traversed phases. Their order depends on the system development life-cycle used. Although there are several models available (e.g., V-, spiral or waterfall model), all of them describe similar phases [Mar18, p. 19ff.].

Typically, the development starts with an analysis of the customer's needs and a definition of requirements. From these requirements, the system architecture is derived. Regarding distributed real-time systems, it includes the definition of components, clusters and their interfaces. The next step in the process is the design of the components [Kop11, p. 264f.], their implementation and validation. Once the validated components are available, an incremental integration follows building the final system. This system is tested regarding the functional specification first, followed by a validation whether all requirements are fulfilled [BST10, p. 236]. In the following, two common design styles are presented. Those are model- and component-based design.

In model-based design, the functionalities of the system and the system's environment are realized as models first. A model is an abstraction of the developed functionality covering only relevant aspects. In many cases, it is possible to execute it on a computer as it contains sequential instructions. Such a model is called a simulation [FLV14, p. 16]. During the definition of the system's architecture, models describing different functionalities are integrated and the models are refined to be representable by a behavioral description. This refinement is continued in the implementation phase to obtain the related software which is adapted to the executing hardware platform [BST10, p. 234f.]. During the last years, model-based design has proven to reduce development costs and time. Furthermore, it increases the system's quality and models can be reused. Due to these reasons, model-based design is an attractive approach for embedded systems development [BKKS12].

Component-based design grounds on the concept of composability as described in Section 2.1.3. Independently developed and potentially existing components are integrated to build large systems via specified interfaces [KS03]. The components' functional and temporal behavior is well known since the requirements are derived top-down from the application functions. In this way, it is not required to provide information about the internal implementation of the components. During the design process, the capabilities of the components are developed bottom-up. It is necessary that the capabilities match the requirements, otherwise a new component has to be developed. In addition, the interface specification must be easy to understand and use [Obe11, p. 27f.].

2.2.2. Verification and Validation

Verification and Validation (V&V) ensures the conformance of a developed system with its specification and requirements within a certain phase of its life-cycle. To reach this, it utilizes reviews, static and dynamic analysis, testing and formal methods [Fis07, p. 3]. Since it is impossible to design tools which generate correctly working implementations, every design has to be verified. In addition, there is no technique available which solves all problems wherefore different techniques need to be combined [Mar06, p. 199f.].

Comparing the abstraction levels of the specification and the implementation, the difference is quite large [Mar06, p. 199]. As a consequence, V&V must be seen as a complementary process to design and performed during the entire development life-cycle. Once a development step has finished, V&V assesses the results analyzing different behaviors. By providing feedback regarding quality and issues, it is possible to figure out system failures or missing functionalities during early design stages [Fis07, p. 3ff.]. In addition, new versions of a system must be validated to determine if already tested functionalities are still working correctly [Lev00].

Although verification and validation are often mentioned together, there is a difference between both. Verification refers to consistency between the system specification and the developed system (called System Under Test (SUT)). Example techniques are (I) formal mathematical analysis and (II) model checking using finite models of a system [CW96]. In contrast, validation considers the user's intention instead of the system specification. While the user's intention describes the role of the system in an application context, the system specification defines this intention from the developer's point of view. Validation usually uses testing to examine correctness in the real world [Kop11, p. 292]. Further validation techniques are simulation and fault-injection.

Testing provides selected inputs to the system and compares the resulting outputs with the expected ones [Mar06, p. 201]. Based on induction, it is assumed that the system works correctly for all possible inputs. However, this probabilistic method requires testing durations in the order of magnitude of the MTTF. For safety-critical systems with an MTTF in the order of $10^9 h$, such tests cannot be performed [Kop11, p. 292f.]. To select a proper set of test cases and the related inputs, different approaches can be used. Examples are random tests using randomly selected inputs or functional testing. The latter approach uses input data leading to a correct or faulty system behavior [JMV04]. Herein, data used for fault-injection should cover all faults defined in the fault hypothesis. Besides the system's functionalities, the selected data should cover aspects of the code such as statements, branches or conditions [Kop11, p. 294f.].

A basic requirement of component-based design is the validation of components in the temporal and value domains in isolation and the maintenance of the correct service after their integration [Kop00]. The component developer can use the technology-independent control interface to parametrize the user scenarios and the technology-dependent debug interface to monitor the internal execution. After integrating the components, their validation must be repeated by the component user since the interactions of components might cause emergent behavior. If the components work correctly, the message exchange can be tested according to the specification of the linking interface [Kop11, p. 297f.]. Exploiting the capability of multicasting, it is possible to observe the message exchange without probe-effects on the components [Kop08b].

Components can be used further to embody the models of a plant or a control algorithm developed during model-based design [FLV14, p. 28]. These components can be linked in a simulation environment to validate the interactions and to tune the closed loop control parameters at early development stages. In later stages, the target application can be derived from the models using a generator software [FvHRK06]. Usually, the simulations operate on different time-scales than the target system. Hence, the phase relations of the message exchange in the simulation environment should be similar to the final implementation. This avoids design errors and improves the simulation's faithfulness [Kop11, p. 299].

Formal methods use mathematical techniques and logical analysis during the specification, design and verification stages [KZ10]. The process starts with a precise representation of system requirements in natural language. This representation is then transformed into a formal specification [EC98] which includes precise semantics and syntax. However, all assumptions, omissions and misconceptions introduced will remain in the model limiting the validity of the derived conclusions. Those are gained during the last step when the model is analyzed and the results are interpreted [Kop11, p. 299f.]. The benefit of formal models is the usage of a precise language without any ambiguities. It is more effective and inconsistencies, incomplete specifications or faults can be found early during the development process [Rus93, p. 39f.]. Furthermore, formal methods can be used to support the certification of functions in safety-critical systems [DBDC03]. However, the syntax and mathematical roots are intimidating wherefore formal methods are rarely used in practice [GD13]. Apart from that, a formal description cannot cover all analyzed system properties and the link between the informal user's intention and the specification is potentially missing [Kop11, p. 293].

2.3. Distributed Co-Simulation

Many models built during model-based design are executable wherefore they are called a simulation [FLV14, p.16]. The execution is performed in a simulation tool which is usually optimized for a respective domain. For example, Riverbed Modeler, OMNET and NS2 are widely used network simulators while control systems can be simulated in MATLAB. Since those tools might provide limited results if they are not used for their intended purpose, simulating a complex system with different components should not be performed by a single tool [HSB10]. In addition, porting components to another tool is time consuming and error prone [CDF+14].

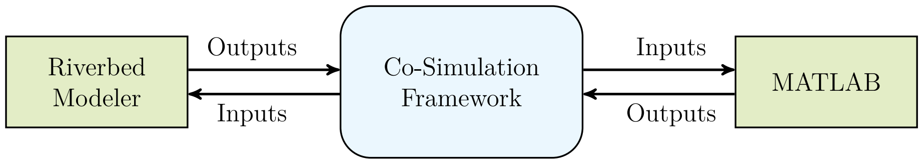


Figure 2.7.: Co-simulation framework.

To solve this disadvantage, co-simulation was introduced. In this technique, a framework couples specialized simulation tools which enables the simulation of complex systems [CDF+14]. Figure 2.7 shows an example where the Riverbed Modeler and MATLAB are connected. While each tool maintains its own time and state, the framework synchronizes them by coordinating their time advance and the exchange of data. Hence, the simulation tools do not communicate directly. In a co-simulation environment, the data exchange consists of shared variables, common design parameters and events. Events can be separated into two types. The first type are time events which are scheduled at a defined time. State events as a reaction on a state change represent the second type [FLV14, p. 18f.]. The simulation environment must ensure a communication mechanism providing a consistent and timely message delivery. A message is delivered consistently, if its timestamp equals the current instant or if it is smaller. In this way, a correct delivery order can be realized [CDF+14].

Complex systems typically consist of components with discrete and continuous dynamics. Discrete systems are digital hardware, embedded software or communication systems while physical processes or analogue circuits are examples for continuous dynamics [Liu98]. Accordingly, there are different simulation paradigms: continuous-time, discrete-time and discrete-event simulation. The systems' dynamics are represented by differential equations in continuous-time simulations. They define the transition between state variables which occur continuously based on the simulation time [Fuj00, p. 30]. In discrete-time simulation, the simulation time is discretized into equal time steps. The simulation time advances to the next multiple of the time step first and the state variables are updated afterwards. Since state changes do not only occur at those multiples but also in between, changes might be missed and a step has to be recomputed considering the event [BAR10]. Instead of using equal time steps, discrete-event simulations jump between the events of the system. They are stored in a chronologically ordered list which is traversed at runtime. The environment selects the next event at the correct instant, simulates the effect and adapts the simulation time using the time of the next event [Mis86].

To synchronize the time in a heterogeneous environment with continuous and discrete dynamics, a simple algorithm can be defined as follows. It is a step-wise algorithm in which the discrete tool defines the step granularity and both tools have the same simulation time at the beginning of a step. The discrete tool sends the size of the next time-step to the continuous tool which advances in time. It stops once an event occurred or if the defined duration is reached. Afterwards, it sends its internal time and the monitored variables back to the discrete tool which advances to the same time [FLV14, p. 19f.]. As shown in Section 3.2.1 there are many algorithms which are suited for real simulations.

Co-simulation must not be restricted to the execution on a single host as shown in Figure 2.8, again with the Riverbed Modeler and MATLAB as examples. Using a convenient framework, co-simulation can be spread across geographically distributed locations. Each tool is coupled with a local instance of the framework while these instances are connected via LANs or even the Internet. In this case, one speaks about distributed co-simulation which has several advantages. It allows project decentralization which enables the parallel execution of simulators on geographically distributed machines. As a consequence, it is possible to design and validate a system by teams located in different sites or countries and to share resources. In addition, simulator licenses and intellectual property can be managed. The latter enables the simulation of components without publishing their descriptions. One disadvantage is the overhead caused by the communication between the distributed hosts which might increase the simulation's execution time. This overhead depends on the network delays wherefore it is not present in local co-simulation. Nevertheless, local co-simulation cannot provide the advantages mentioned above [AMO+02].

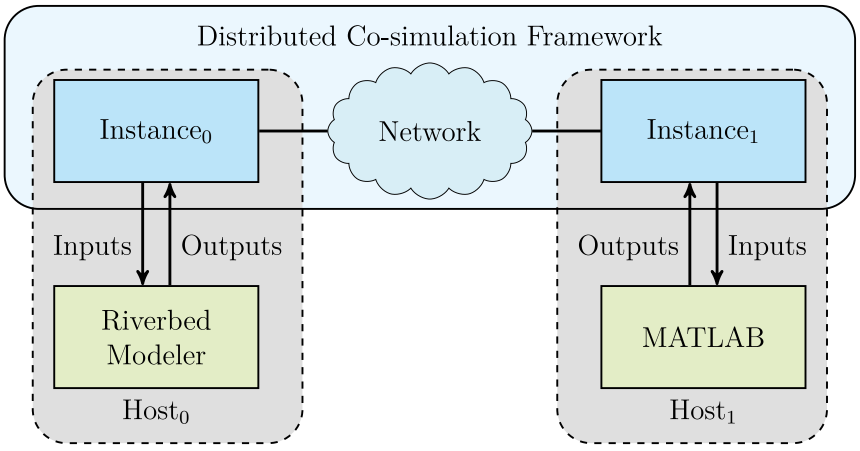


Figure 2.8.: Distributed co-simulation framework via a communication network.

The synchronization of distributed discrete event simulations can be realized by merging the local event lists to form a global one. This list is traversed by selecting the simulation with the smallest time-stamp of the next event and allowing it to execute. The other tools suspend their execution in the meantime. Once the simulation step has finished, data messages are exchanged and the next tool is scheduled for execution. If new events arise during the execution, they are added to the global event list and considered in the following [OAO15]. If time-stepped simulations are involved, several synchronization points can be predefined. At runtime, the tools execute until they reach such a point and exchange information afterwards [LVS$^+$12].

2.4. Software- and Hardware-In-The-Loop Testing

Although co-simulation enables the validation of complex systems at early design phases, it has several disadvantages. First, models are always abstractions of the developed systems. Even without abstractions, there is the risk of introducing errors or losing design details wherefore validating a model and the correctness of results is difficult. Furthermore, there must be a trade-off between a model's accuracy and the simulation time required to process the results. This issue makes large simulations or simulations with many design details impractical. Finally, the model's code is typically not reused for the final software increasing the development costs. Reasons are the design of simulation tools for a usage during the design phase (flexibility, low cost, etc.) and possibly different languages between the simulator and the software [DGK07].

To save development costs and time, hardware and software can be developed in parallel. As soon as prototypes of both parts are available, they are integrated and testing starts. However, there are many cases in which serious integration problems arise. The reason for these issues are side effects from the device's technological aspects. If hardware and software are developed independently, they cannot be considered [BCPS11].

Software-In-The-Loop (SIL) testing as illustrated by Figure 2.9 provides a solution to these issues. Here, the interactions between a software-implemented control algorithm and the model of a simulated plant are validated. Using SIL, it is possible to validate the final software directly during the design phase without developing a related model. This possibility saves time and costs. If the hardware is not available yet, a simulation provides different validation setups and repeatable tests. Furthermore, the environmental conditions of a simulated plant can be controlled which is not possible in field tests outside a lab [DGK07]. In this way, integration issues due to technological aspects can be eliminated early during the design [BCPS11]. However, there are also challenges remaining. Examples are scalability in large setups, required modifications in the software to interact with the simulator, breaking TCP connections between the software and a network simulation, or timing issues combining event- and time-triggered paradigms [DGK07].

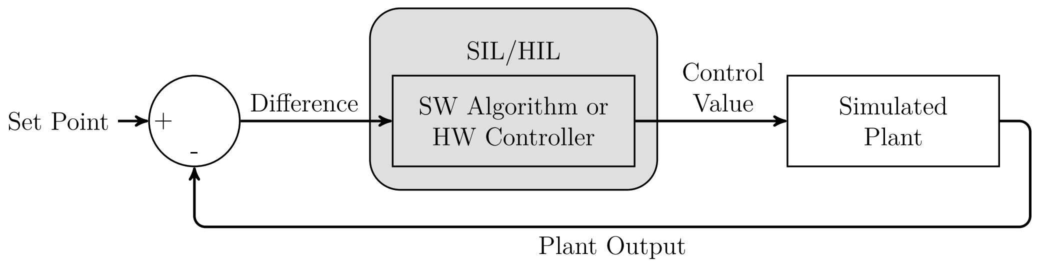


Figure 2.9.: Software- and Hardware-In-The-Loop testing.

While SIL validates the software against a simulated plant, Hardware-In-The-Loop (HIL) testing includes actual physical devices into the simulation loop (cf. Figure 2.9). These devices interact with the simulation models in real-time via input and output interfaces of the HIL test bench [LWFM07]. Applying this technique during the design phase has several advantages. It avoids errors in later steps, increases the system's reliability, provides efficiency [SY14] and improves the flexibility and rate of various tests. Furthermore, it prevents damages of equipment or human lives and reduces costs and development time [BCD+12]. If a model replaces its system and the remaining hardware and software components are implemented exactly, high implementation costs arise. Hence, a trade-off must be made between accuracy and the expenses of the hardware [SY14]. However, HIL also has a major issue in distributed real-time tests via the Internet. Since real-time hardware is connected, the execution must be performed in real-time also. In such setups, the Internet's delays, jitter and packet loss reduce the stability, accuracy and transparency of the test. Hence, the HIL framework must provide a mechanism which mitigates these issues [EBSF12]. Although both approaches have disadvantages, they are widely used in the validation and verification process.

2.5. Fault-Injection

Fault-injection is an effective way to validate the dependability of a system. This includes the tolerance mechanisms' effectiveness and the system's ability to detect errors, locate them and to recover. In addition, fault-injection provides feedback to the developer for improvements [PBC+96]. There are different targets where faults can be injected: hardware, software and simulations [NL11]. The related injection techniques are presented in the following.

Hardware-implemented fault-injection requires additional hardware to perform the injection. There are two different injection types depending on the faults and their location. Both cause faults by inducing voltage or current into the system. The first type uses direct contact via (I) active probes attached to pins or (II) sockets. While active probes may destroy the system due to an inappropriate amount of current, sockets are separated from the hardware. They are able to apply boolean operations on the signals like AND, OR or inverting. The second injection type does not have a direct contact but uses radiation or electromagnetic interference. Since it is difficult to determine the injection location and time exactly, these techniques cannot be controlled precisely [HTI97].

Software-implemented fault-injection provides a scalable and statistically verifiable means to analyze the behavior of the target system for different types of faults. In contrast to hardware-implemented fault-injection, the results are reproducible and do not cause damages in the hardware [SV05]. It is applicable for operating systems or applications without using additional hardware. To inject faults into the operating system, the fault-injector must be embedded into it. Otherwise, an additional layer can be added between the application and the operating system. At runtime, the injection is performed using different mechanisms. A timer may trigger an interrupt or an event or a condition may cause an exception. Both inject the fault afterwards. Alternatively, the code may be changed to cause the injection. However, these mechanisms have the following disadvantages. The injection target must be accessible by the software, the execution might be disturbed or the software might be changed. Furthermore, it has a limited time resolution whereby faults with a short latency might not be captured. To prevent the latter, software-implemented fault-injection should be combined with hardware monitoring [HTI97].

Finally, simulation-based fault-injection enables the observation of the behavior and the propagation of faults [STB97]. It is a versatile, controllable and simple approach which can be realized by simulator commands and modifications of simulators or models. The first technique provides commands in the simulator to manipulate signal values or variables in the model. It does not require model modifications but injecting a large number of faults is inconvenient. Modifications in the models are realized using saboteurs or mutants. Saboteurs are separate modules which manipulate inputs and outputs. Although the technique increases the complexity of the modules since multiple signals must be added, it is possible to implement more faults compared to simulator commands. Mutants are modified modules of the original target and replace it at runtime. They support many types of faults but have to be built in advance. Furthermore, additional memory space is needed for every mutant which is a disadvantage since their number in typical fault-injection experiments is large. The third simulation-based technique are simulator modifications. It simplifies the experiment and requires less simulation resources as deterministic or random injection can be performed easily. Furthermore, there is no need to change the original semantics of the target in an event-driven engine [NL11].

2.6. Communication and Data Security

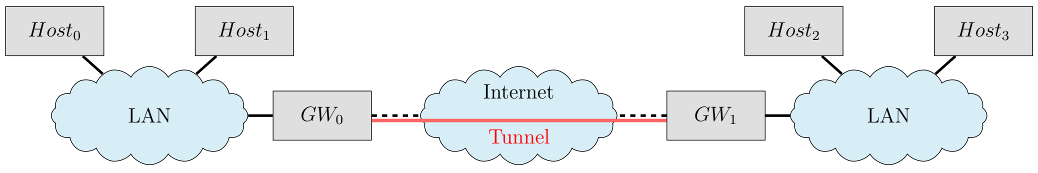
IT security is used to protect data and resources in computer systems from unauthorized access or manipulation. In contrast to safety, it covers the analysis of threats and the application of countermeasures against them. There are several protection goals defined such as confidentiality, integrity and authenticity [Eck08, p. 5f.]. During the tests, the framework exchanges data which may be legally protected by non-disclosure agreements or intellectual property. Hence, it is necessary to protect it technically from manipulation and unintended insights considering confidentiality, integrity and authenticity.

Symmetric or asymmetric encryption mechanisms ensure confidentiality and prevent unauthorized insights into the exchanged data. Both types transform an original into a cipher text using a key and an encryption function [PK79]. In symmetric mechanisms, this key is the same for encryption and decryption. Many known algorithms are implemented as block ciphers which transform n-bit blocks into the cipher text [AM12]. However, symmetric mechanisms have the disadvantage of depending on a secure distribution of the key between the communication partners. Asymmetric algorithms use different keys instead: a private key which is known only by the user and a public key which is open to everyone. The sender encrypts the data using the receiver's public key while the receiver is able to decrypt the data by means of his private key. Since it is computationally infeasible to derive the private key from the public key, the communication is confidential [TW11, p. 793f.] as long as the key is authentic [bsi19, p. 32]. Typical examples such as RSA or elliptic curves are based on the problem of factoring large numbers and computing discrete logarithms modulo a large prime [RSA78, Kob87]. However, the computations required for RSA are more complex compared to symmetric algorithms. Hence, combinations are commonly used where a symmetric key for the communication is exchanged via asymmetric means such as RSA [TW11, p. 796].

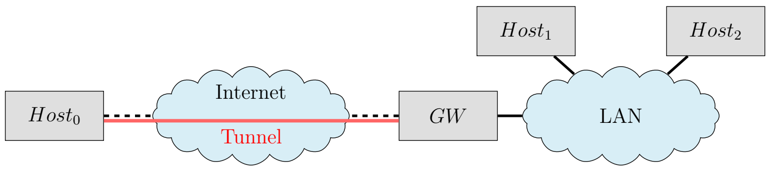
The integrity and authenticity of data can be protected using one-way hash functions and digital signatures. A one-way hash function maps an input into a value taken from a smaller set [Win84]. To be secure, the hash function must prevent collisions and it must be impossible to calculate the input from the hash [Mer89]. Furthermore, changing one bit must produce a very different output and the hash value must be sufficiently large [TW11, p. 801]. To ensure data integrity, the hash function is applied on the data and sent with it. The receiver can calculate his own hash value from the received data and compares it with the received hash. If the hashes are equal, the data should be the same [Eck08, p. 350]. Combining the hash function with a secret key, the authenticity of the sender can be ensured. Such a function is called Message Authentication Code

[BCK96]. Alternatively, digital signatures ensure the sender's authenticity, the data's integrity and non-repudiation. They are created using asymmetric mechanisms. The sender encrypts the message with his private key and the receiver can restore the original content by applying the sender's public key. While RSA provides both, encryption and signatures [RSA78], the Digital Signature Algorithm only provides the latter [NIS92].

In practice, the protection goals mentioned above can be realized using Virtual Private Networks (VPNs). A gateway in a private network encapsulates the original packet and sends it as payload in another packet [TW11, p. 821f.]. This technique is also called tunneling. In this tunnel, the encapsulated payload is secured using a protocol such as IPSec which provides a confidential and authentic data exchange [Eck08, p. 704f.]. A second gateway in the receiving private network decapsulates the packet to forward it. This VPN type is called Site-to-Site-VPN [Fri04] and shown in Figure 2.10a. While the Site-to-Site VPN connects different sites of a company, a Remote-Access-VPN establishes the connection between the company network and an end device [BH06] (cf. Figure 2.10b).



(a) Site-to-Site VPN



(b) Remote-Access VPN

Figure 2.10.: Different types of VPNs.

3. Related Work

3.1. Real-Time Communication Systems

During the last years, there was a growing interest in real-time communication via Ethernet. Hence, several technologies such as EtherCAT, PROFINET or AFDX were developed. Two further examples on which the distributed co-simulation framework focuses are described in this section. Those are TTEthernet and Time Sensitive Networking.

3.1.1. TTEthernet

TTEthernet is a deterministic, time-triggered extension of the Ethernet standard which is designed to support mixed-criticality systems [SB09]. The main objectives during the development process were the integration of time-triggered and event-triggered communication via the same Ethernet network, transparent synchronization, scalability and fault-tolerance [tte]. In November 2011, the technology was standardized in SAE AS6082 [sae11]. Due to the scope of the distributed co-simulation framework, this section concentrates on the three traffic classes provided by TTEthernet.

The first traffic class is the Time-Triggered (TT) message which has the highest priority. It is used to ensure determinism, tight latency and jitter [Obe11, p. 185f.]. To guarantee the timing, the clocks of the nodes in the TTEthernet network establish a network-wide synchronized time base [sae11, p. 9]. Each TT message is sent at a defined point in time which is determined by a period and an offset in this period [sae11, p. 14]. The length of the frame determines the required length of the related time-slot. Due to the integration of synchronized with unsynchronized traffic, messages may be delayed by a bounded interval. As a consequence, a so-called acceptance window has to be considered during which a TT message is received. It depends on the link latency and the precision of the global time-base [Obe11, p. 207f.].

Rate-Constrained (RC) traffic as the second class is used for less strict temporal requirements without synchronization to a global time-base [SBH$^+$09]. Using a sufficient bandwidth, delays and temporal deviations have defined limits and packet loss can be avoided [tte]. Furthermore, it ensures a data-flow in which successive messages are forwarded with a minimum offset [sae11, p. 10]. The bandwidth is configured using a transmission rate parameter. This parameter determines the maximum generation rate and the minimum time interval between two subsequent packets. However, the maximum time interval is not defined. Using a rate-enforcing algorithm, network switches ensure the configured frame rates. Since TT traffic has a higher priority, it might block RC messages. Including blank intervals in a sparse time-triggered schedule solves this issue [Obe11, p. 212f.].

Finally, Best-Effort (BE) messages represent the standard Ethernet traffic without any guarantees. It has a lower priority than the other classes wherefore it uses the remaining bandwidth [tte].

3.1.2. Time Sensitive Networking

Apart from TTEthernet, Time Sensitive Networking (TSN) is developed recently for reliable communication with low packet-loss and guaranteed latency [NTA^{+}18]. The TSN task group [tsn19] as part of the IEEE 802.1 working group advances the IEEE 802.1 Audio/Video Bridging (AVB) standards. Herein, the focus lies on AVB shortcomings regarding crucial requirements for industrial automation. Those are reduced latency and accurate determinism, independence from physical transmission rates, fault-tolerance without additional hardware and improved interoperability, safety and security [WSJ17].

A TSN network reserves defined bandwidth, buffering and scheduling resources for a traffic flow. This offers bounded latencies and prevents congestion loss. Furthermore, the packets of a flow can be sequenced and sent via one or more redundant network paths. Using sequence numbers, duplicated packets can be detected and eliminated. Together with zero congestion loss, this capability provides reliable packet delivery. To ensure real-time communication, the network devices and hosts are synchronized to an accuracy between $1\mu s$ and $10ns$ using a variant of the Precision Time Protocol (PTP), also known as IEEE Std. 1588. Finally, TSN allows to add and remove traffic flows while maintaining a proper transmission of the other flows. The remaining bandwidth can be assigned to standard best-effort data flows which work in their usual manner [Fin18].

In store-and-forward bridging, every packet is buffered in a switch before it is forwarded via its destination port [TW11, p. 356]. TSN defines a number of output queues for each port in which incoming packets are buffered once their destination is determined. To provide zero congestion loss, the buffer required in the worst case must be determined. This is possible because of the following two properties. First, the queuing algorithms which select the next packet to forward define their own schedules. Besides this, the input rate of a switch shall equal its output rate. To realize the latter, each TSN switch buffers a sufficient amount of packets for a TSN flow. In this way, there is a transmission opportunity even if a flow slows down or stops and a constant output rate can be provided for some duration of time. The number of packets to buffer depends on the duration during which both rates shall be maintained [Fin18]. There are different queuing algorithms available such as IEEE Std. 802.1Qbv, IEEE Std.802.1Qch and IEEE Std. P802.1Qcr. The Time-Aware Shaper defined in IEEE Std. 802.1Qbv even supports time-triggered traffic with deterministic ultra-low latency requirements [NTA+18]. In Qbv, slots can be defined which are used to forward scheduled traffic. During those time intervals, non-scheduled traffic is blocked and interference with it is prevented [DD15]. However, high priority traffic still might be blocked by unsynchronized low priority streams. To prevent this drawback of priority inversion, the IEEE Stds. 802.1Qbu and 802.3br where introduced which provide packet preemption [NTA+18].

3.2. Distributed Co-Simulation

As explained in Section 2.2.1, model-based design starts with developing models of the components and validating them using distributed co-simulation. This section introduces available co-simulation frameworks and two widely used simulation standards: the Functional Mock-up Interface (FMI) and the High Level Architecture (HLA).

3.2.1. Distributed Co-Simulation Frameworks

This section starts with frameworks for distributed co-simulation. It focuses mainly on the co-simulation of networked control systems, different synchronization mechanisms and the connection of simulation tools between distributed hosts.

In the current state-of-the-art, several works can be found which co-simulate networked control systems. Hasan et al. for example couple MATLAB/SIMULINK and OPNET [HYGY08, HYCY09] while Mkondweni et al. [MT13] realize data exchange between LabView and the NS-2 network simulator via UDP. Hence, this framework could also be used in a distributed setup. In contrast, the NMLab environment couples NS-2 with MATLAB via the Tcl interface of NS-2 [HSB10]. Two approaches using SystemC for the network simulation are provided by Quaglia et al. [QMBF12] and Zhang et al. [ZEK+14]. Similar to this thesis, the latter consider TTEthernet as the network technology used to connect the components. Instead, ten Berge et al. focus on fieldbuses in their work [MOB06]. Finally, the Pia simulator developed by Hines and Borriello [HB97] supports network-centric, distributed co-simulation of embedded systems including the protection of intellectual property.

Besides networked control systems, co-simulation is commonly used to couple power with network simulations. The FNCS framework [CDF+14] couples the simulators via a central broker. It supports three different algorithms for synchronization. A conservative approach uses the smallest next time step of the power grid simulators while the other two exploit speculative execution to speed up the performance. VPNET integrates the Virtual Test Bed (VTB) and OPNET [LMLD11]. A coordinator manages a global communication step time, starts and stops the tools and triggers the execution of a step. Together with defined sampling periods, the global simulation time defines the instants for communication. The middleware presented by Lai et al. [LSW+14] balances the requirements of accuracy and efficiency for adjusting the simulation step size. To realize this, it exploits information about communication delays and errors. Another tunable synchronization mechanism is used in GECO [LVS+12]. It is based on a global event list which is ordered according to the event time-stamps. A scheduler identifies the next event at runtime wherefore it is able to react on interaction requests without delays.

To enable distributed co-simulation, several works use TCP/UDP sockets. Bian et al. [BKP+15] connect the OPAL-RT (power) and OPNET (network) simulators and provide two synchronization mechanisms. In a non-real-time approach, the tools communicate via files while the real-time solution requires an interface for synchronization and data exchange. Another example is provided by Owda et al. [OAO15] who simulate networked multi-core chips using OPNET and Gem5. The tools are connected via local communication controllers which exchange data and synchronize them based on a global event calendar. Harding et al. [HGY07] couple MATLAB and OPNET using Java and threaded sockets while Amory et al. exploit Unix sockets. Their work co-simulates hardware and software via a simulator-independent backplane [AMO+02]. Finally, the FSKIT [KTS+15] is based on the Message Passing Interface (MPI). An asynchronous API enables overlapping of the communication with the execution activities and the synchronization is realized using three mechanisms. A conservative approach synchronizes all events while the second mechanism uses fixed-length intervals. The third technique is based on speculative execution instead.

3.2.2. The Functional Mock-Up Interface

As a tool-independent standard, the Functional Mock-up Interface (FMI) facilitates the co-simulation and the exchange of dynamic models between different simulation tools [BOA+11]. It is available in two versions (FMI 1.0 and 2.0) and supported by various tools already [Assb]. FMI provides two types of interfaces. While the first type enables the import and export of simulation models in other tools (FMI for model exchange), FMI for co-simulation realizes an environment for the co-simulation of models. Together with new features, version 2.0 integrates both interfaces in one standard which improves usability and performance [BOA+12].

Mathematical equations and events can be used to represent dynamic system models [BOA+11]. From a dynamic model, FMI for model exchange creates C-code which can be included by other simulation tools as an input/output block. In FMI for co-simulation, the different simulation tools are coupled by a master. Its purpose is the synchronization of the tools and the exchange of data at discrete communication points. Between these points, each simulation is executed individually [BOA+12]. Since there is no master algorithm defined in the standard [CLT+16], there are various works focusing on this topic. Examples are a deterministic solution [BBG+13], an extension of this work for co-simulating continuous and discrete dynamics [CLT+16] and performance improvements using different step sizes [SAC12].

Functional Mock-up Units (FMUs) are components implementing the FMI standard and can be shared via an archive. This archive contains information about the model and the definition of all environmental variables, the model code and additional data. As the model is described using equations and events, their implementation has to be included either as source code or as a binary. For co-simulation, there are functions to initialize the communication with the simulation tool, to compute a step and to exchange data. The additional data is optional and may represent documentation or included libraries [BOA+11].

FMI is used in various frameworks which focus on different simulation problems. Since cyber-physical systems couple continuous with discrete dynamics [LS10, p.1], their properties must be considered during simulation. Tripakis [Tri15] encodes subsystems with heterogeneous modeling formalisms. After creating related FMUs, they are co-simulated using the results from Broman et al. [BBG+13]. The latter focus on deterministic execution and develop memoryless FMUs which implement a rollback or step-size prediction mechanism. Together with a master algorithm that queries an FMU for the time of future events, determinism can be reached. This solution is exploited further as a master algorithm in the FIDE framework [CLT+16] implemented by Cremona et al. Finally, Elsheikh et al. combine different simulation tools and extend them by the flexible prototyping capabilities of Modelica using FMI [EAWP13]. In this way, they can simplify the development of cyber-physical systems.

3.2.3. The High Level Architecture

The IEEE Standard 1516-2010 [iee10b], also known as High Level Architecture (HLA), is a widely applicable simulation standard. It is based on distributing a simulation into subsystems which are called federates. The ensemble of all federates is called a federation [APE+13].

The HLA standard consists of several parts. First, the HLA Framework and Rules Specification (IEEE Std 1516-2010, [iee10b]) defines rules which guarantee a proper interaction between the federates and their responsibilities. An HLA Federate Interface Specification (IEEE Std 1516-2010.1, [iee10a]) represents the second part. It defines the services provided by the HLA's central component, the Runtime Infrastruture (RTI), including details about their implementation. The third component is the HLA Object Model Template (OMT) Specification (IEEE Std 1516-2010.2, [iee10c]). It defines a specification about the object models which intends to ensure a common data model for mutual interaction.

The HLA does not constrain what is represented, but the federates must document their object models using the OMT. In this way, the standard facilitates information sharing and reusability. Additionally, the HLA specifies interactions between the federates which are performed via the exchange of data [DFW98]. Since it is language and platform independent, the HLA further provides a solution for the most common interoperability problems [APE+13].

The standard provides six main categories of management and a number of support services. The most important ones are presented briefly in the following while a detailed description can be found in [iee10a].

Federation Management The services are used to connect a federate to the RTI, create federation executions and join them. Once the simulation execution has finished, a federate resigns from the federation, destroys it and disconnects from the RTI. Using synchronization points, the HLA provides a barrier synchronization for participating federates.

Declaration Management Data exchange is realized by attribute updates or interactions following a publish/subscribe pattern. While the first type represents updated characteristics of an object, the latter is data which is sent explicitly.

Object Management These services handle the data exchange at runtime. They are used to register object instances, discover them, send updates of object attributes and to exchange interactions.

Ownership Management The HLA supports to transfer the ownership of instance attributes between the federates. In this way, object instances can be modeled cooperatively across a federation.

Time Management Time management is one central element in this work. The HLA supports two different approaches for time management, a conservative and an optimistic one. The conservative approach enables a federation-wide synchronization so that messages are exchanged in a consistent order for all participating federates (cf. Section 4.3.1). Instead, the optimistic approach enables the transmission of messages without order consistency. Exploiting parallelism, it is able to increase the simulation's performance.

Data Distribution Management The services enable a more fine-granular data transfer considering instance attributes. Its purpose is to limit the exchange of irrelevant data by grouping instance attributes into regions.

Support Services Support services can be used to perform transformations between handles and names, evoke callbacks, set advisory switches or manipulate regions. Furthermore, there are services to get the message order type or name and to obtain the handles of interactions, object instances or attributes.

Since the HLA was introduced, a lot of work has been done to improve it and there are several frameworks available which use the standard. The first group of works focuses on enabling real-time simulations via the standard. For example, they use real-time operating systems and Quality of Service technologies [ZG04, CSSA14, BL05, GDKR16] or focus on providing real-time guarantees via the RTI [MFF04, CNS11]. In this way, it is also possible to establish HIL testing [JBN15]. To hide information between groups of federates, Cai et al. introduce hierarchical federations. In their work, a gateway or a federation proxy forwards data between these groups [CTG01]. A similar approach is used by Bréholée and Siron for scalability, security and interoperability [BS03]. Besides this, optimistic synchronization is able to increase the performance of simulations in specific scenarios [SQ06]. Hence, it is another common research objective [SQ12, SQ06, NSRM09, WTLG05]. Finally, there are approaches which simulate network-centric systems. While Rivera et al. [RTC+11] combine the HLA and high performance computing to investigate tactical edge applications, other authors distribute their simulations via the standard

[OIHVL04, BBZ+13, YZ12]. Those works use the OPNET tool for network simulations. The next section shows the possibility to combine the HLA with FMI. In this way, a simpler and more generic interface to simulation tools can be provided.

3.2.4. Combinations of FMI and HLA

As explained in Section 3.2.2, there is no master algorithm defined in the FMI standard and several authors concentrated on this topic. In this section, related works are presented which use the HLA as a master.

Awais et al. published two approaches. The first [APE+13] aims on the usability of components in various distributed environments which shows the possibility of integrating the standards. In [APM+13], they implement a heterogeneous platform which couples fixed time-stepped, continuous-time and discrete-event simulations. To reach this, they exploit the HLA's *TimeAdvanceRequest* service for the first two types while discrete simulations advance in time using the *NextMessageRequest* service. The work of Neema et al. [NGL+14] focuses on cyber-physical systems. By using different solvers and step sizes, discontinuities and non-linearity can be prevented in the simulation. Furthermore, distributing the simulations via the HLA and using it as a master algorithm enables flexibility in connecting different simulation types.

Finally, Garro and Falcone [GF15] analyze the combination from two perspectives, HLA for FMI (I) and FMI for HLA (II). To realize the first approach, they extend FMI to include the HLA by modifying the model description file and adding HLA related functionalities into the solvers. This improves performance and reduces development time and costs since the FMUs become reusable. The second approach uses either an adapter between the FMU and the RTI or a mediator to reduce dependencies and to lower the coupling. The adapter manages the FMU's lifecycle while the mediator realizes the communication between the FMUs and a federate. This federate uses the FMUs to simulate specific components.

3.3. Software- and Hardware-In-The-Loop Testing

After validating the models of a System Under Test, it is possible to replace them by software-implemented control algorithms and the final hardware. Both technologies are widely used in the automotive, railway and other domains. This section presents available frameworks for SIL and HIL testing.

In the automotive domain, the work of Yan et al. [YWL02] uses SIL and HIL testing for the development of chassis control systems. They implement a model of the vehicle dynamics in Matlab/Simulink/Stateflow and couple it with control algorithms or the production module. To realize this, they use the dSPACE Rapid Prototyping system and Targetlink. Mitts et al. investigate the benefits of SIL testing for the development of next generation engines and transmission systems. They provide a flexible SIL solution which is based on the CosiMate tool-chain [MLRK09]. Another SIL environment for developing transmission systems is implemented by Zoppi et al. based on Matlab/Simulink [ZCTV13]. Using NI TESTSTAND and NI LabVIEW, Kulkarni et al. introduce a HIL system for testing the after-treatment system in cars [KPB16]. Since these solutions suffer from specialized use-cases and missing interoperability, several projects were started which aim on solving the issues. The FMI standard, the Distributed Co-Simulation Protocol [KBB+18, BKD+19, KB18] or the ASAM standard family with the ASAM XIL API for SIL and HIL [asa17] are example results of these projects. They all target on simplifying the development and validation process.

Regarding the railway domain, Malvezzi et al. work on a HIL test rig to validate on-board subsystems which is based on dSPACE Controldesk [MMPP07]. Baccari et al. model the most relevant electromechanical components of the powertrain and couple them with the related electronic control units via real-time HIL [BCD+12]. A similar setup is presented by Pugi and Allotta who integrate actuation systems into rig design [PA12]. Facchinetti and Bruni propose a test bench for the interaction between a physical pantograph and a numerical model of a centenary. Comprising stagger effects in the contact wire, they reproduce the interaction in a 0-20 Hz frequency range [FB12]. While Verhille and Bouscaryol simulate the traction system of an automatic subway in [VBBH07a], they focus on an anti-slip control mechanism for traction systems in [VBBH07b]. Both works exploit HIL testing. Finally, the authors of [TKS99] concentrate on causality variations and the hybrid nature of discrete and continuous vehicle control systems and simulations.

Besides automotive and railway, SIL and HIL can be found in various other domains. ISIS+ is a SIL framework for unmanned aircraft simulation [RBP13] while Liu et al. examine a speed drive controller's sensitivity for power quality deviations using HIL [LSR05]. Huerta et al. apply power HIL to evaluate control strategies for power interfaces in a hierarchical, model-based approach [HGPM16]. Validating space equipment on the ground suffers from difficulties due to environmental and hardware limitations. To prevent these difficulties, Sun et al. propose a HIL setup connecting real controllers and virtual devices [SCM16]. Examples for distributed SIL and HIL setups can be found further. The tool SPADES exploits distributed SIL testing via Unix sockets to support machine learning in various applications of artificial intelligence [RR]. CEMTool is a distributed

SIL environment connecting the nodes via Ethernet. To provide real-time guarantees, a master-slave concept is used where a master synchronizes its slaves and exchanges data in fixed time-slots [KC99]. Another approach is introduced by Ersal et al. They analyze the negative effect of latency, jitter and packet loss in distributed HIL via the Internet to develop suitable countermeasures [EBSF12].

In heterogeneous setups, the logical time of non-real-time simulations must be synchronized with the physical time of real hardware. CORESIM [LFRE01] realizes interoperability using a lock-step and a time-slicing approach. Both algorithms have the disadvantage of freezing the real devices so that the simulations are able to catch up. Finally, HIL testing introduces real-time requirements on the tests wherefore the execution environment must provide the related guarantees. Hence, Lu et al. extend the Virtual Test Bed environment (VTB) by a real-time component (VTB-RT) [LWFM07]. Running the tool on a Linux system modified by the Real-Time Application Interface (RTAI) enables the execution of the tests in real-time.

3.4. Linux as Real-Time Operating System

Besides embedded RTOSs, there are approaches which extend standard OSs to provide real-time behavior. For example, the Linux Operating System does not support critical real-time applications since it cannot provide the required timing guarantees [RLB08]. This section focuses on approaches which transform Linux into an RTOS. Those are RTAI, Xenomai, Preemp-RT and the Linux deadline scheduler.

The Real-Time Application Interface (RTAI) is based on a real-time nanokernel which provides basic services such as scheduling or memory management. It was implemented first as a Real-Time Hardware Abstraction Layer (RTHAL) which captures and redirects hardware interrupts to a real-time handler or the Linux OS. To provide low response times and jitter to real-time applications, the interrupt handling is fully preemptive [ZCY06]. Later, the approach has been extended to the ADEOS nanokernel. It virtualizes the RTHAL functionalities into an event pipeline which hosts several domains. Each event such as an interrupt or a system call is dispatched to a domain according to domain priorities. To ensure a timely execution of real-time tasks, the highest priority is reserved for them [DM03]. The ADEOS approach is more complex, but the scheduling latencies are only slightly higher compared to the RTHAL. Hence, ADEOS is capable of replacing it [ZCY06].

A similar approach to RTAI is Xenomai. Since they originate from the same project, they share most concepts [BLM+08]. Both technologies are interfaces to real-time tasks which treat the Linux kernel as an idle task. While RTAI supports event handling using both, the RTHAL and the ADEOS mechanisms, Xenomai only uses the latter [KC13]. Apart from that, Xenomai allows the execution of real-time tasks in the user space which isolates their memory space from the remaining Linux tasks. The resulting memory protection is not available in RTAI where real-time tasks and Linux run in the same space. Disadvantages of this execution mode are increased latency and jitter due to the Linux system. However, performance evaluations have shown almost the same latencies in the real-time domain compared to RTAI and those in the Linux domain are still acceptable [SL06].

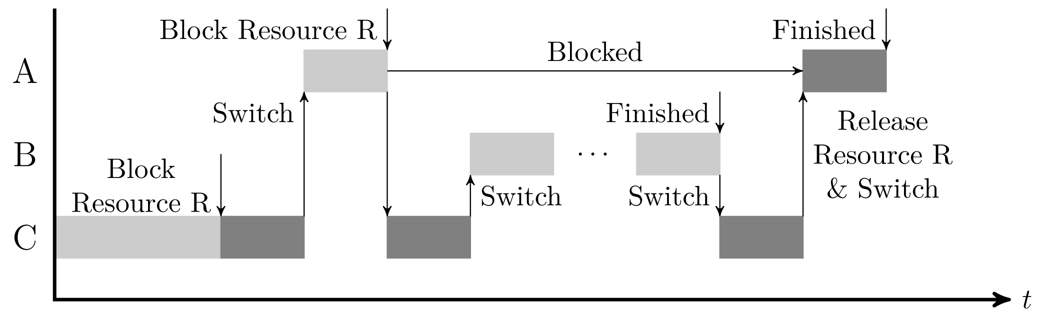


Figure 3.1.: Priority Inversion (adapted from [WB05, p. 390]).

Both mechanisms suffer from reduced usability since the execution in the kernel mode does not allow the usage of normal system calls. This also aggravates debugging [BLM+08]. Hence, researchers focus on turning Linux into an RTOS [Ber]. Instead of using an additional nanokernel, the PREEMT-RT patch makes the Linux kernel fully preemptive [LS11]. Herein, it focuses on reducing the maximum and average response time of a real-time tasks. To realize this, the patch changes different parts of the kernel in the following way. First, high resolution timers were introduced which have a resolution bound to hardware capabilities. In addition, threaded interrupt service routines handle most interrupts with some exceptions such as timers. By using a lower priority for the interrupt handler thread than for the real-time processes, low priority interrupts do not affect high priority threads. The third aspect is replacing almost all spin locks by mutexes. This enables preemption in critical kernel sections and prevents busy waiting for blocked ones. Finally, priority inversion as shown in Figure 3.1 is a normal issue where a low priority task C blocks a resource which is required by a high priority task A. If a medium priority task B now preempts C, A is also blocked for an undetermined time since it waits for the resource. By implementing priority inheritance, C inherits A's priority to release the resource. This enables A to execute [RVH07]. Performance evaluations have shown comparable results with RTAI and Xenomai [SL06] while the patch outperforms the standard Linux kernel heavily regarding the maximum wakeup latency [SL06, CB13, BGD12]. However, there are setups where heavy load causes high latencies [Hen09].

A PREEMPT-RT patched kernel schedules its tasks based on priorities, but it does not consider a task's deadline. This may be one reason for high latencies in heavy load setups since multiple tasks with the same priority might be scheduled at the same time. Lelli et al. provide a scheduling algorithm which is based on ensuring the deadlines of periodic tasks [LSAF16]. Based on resource reservations, the algorithm ensures temporal isolation between the tasks which allows real-time scheduling. This reservation guarantees a maximum execution time Q_i of a task during its period T_i. In case the task exceeds its maximum execution time, the algorithm throttles the task by preventing its execution until the next period starts. To decide which task to execute next, the scheduler uses the deadlines d_i and the remaining runtime q_i of each task and schedules them according to the Earliest Deadline First algorithm. This also accounts for multiprocessor scheduling. Since shared resources might block real-time tasks, the algorithm supports deadline inheritance similar to the priority inheritance of PREEMPT_RT. However, the guarantee of maintaining the deadlines is only provided if the overall utilization of all real-time tasks $\sum_{i=1}^{N} \frac{Q_i}{T_i}$ is below a certain threshold. This value is set to one for uniprocessor scheduling, but cannot reach a value similar to the number of cores in a multiprocessor setup. The authors' evaluation has shown the applicability of the mechanism for real-time applications even in high load scenarios until a utilization of 90% on a uniprocessor PC. In case of a multiprocessor, some deadline misses occur but the percentage is lower than for the alternative Linux schedulers SCHED_FIFO and SCHED_OTHER. The algorithm is available in the mainline kernel since version 3.14 and supports the combination with PREEMPT-RT.

3.5. Technologies for Real-Time Tests via the Internet

Supporting HIL testing via the Internet requires the transmission of packets in bounded time. However, this is hardly achievable due to the network's best effort character. This section presents technologies which may be used by the framework to mitigate insufficient network delays. It starts with Quality of Service mechanisms and continues with two technologies which come from different research topics. Those are state-estimation and speculative execution.

3.5.1. Quality of Service Protocols for the Internet

Due to its best effort packet transmission, the Internet is not suitable to provide Quality of Service (QoS) guarantees [MS99]. Once the network demands exceed its capacities, the provided service is degraded resulting in increased packet loss and jitter [PYPB02]. Some means to provide predictability are Integrated Services (IntServ), Differentiated Services (DiffServ), Multiprotocol Label Switching (MPLS) and Deterministic Networking (DetNet). They are presented in this section.

Integrated Services are based on the Resource Reservation Protocol (RSVP). Using this protocol, a path from the sender to the receiver is determined for the current data flow first. Afterwards, the receiver replies with the requested bandwidth. The reply is sent via the path and each router reserves the required amount of resources if possible [MS99]. In this way, the mechanism enables two service types. While the guaranteed service type ensures a bounded end-to-end queuing delay, controlled load services share the overall bandwidth between multiple flows. The provided service with almost no loss and delay can be compared to an underutilized network [ZOS00].

Similar to Integrated Services, Differentiated Services support two service types, one with absolute (premium service) and one with relative assurance (assured service) [ZOS00]. By forwarding packets according to assigned traffic classes, the mechanism works without any reservations [MS99]. Packets of a traffic class are marked using the Type Of Service (TOS) field in the IP header. The routers analyze the field and select the next packet to forward based on their priorities. Since the complexity of classifying traffic is shifted to the network boundary, the core router tasks remain simple. This enables the scalability of the mechanism [ZOS00]. Comparing the performance, Differentiated Services have shown lower end-to-end queuing delays. As a consequence, they can provide the QoS guarantees for high traffic better than Integrated Services [SM05].

MPLS is a routing mechanism where the header is settled between layer two and three. Among others, MPLS packets contain a label and a Class Of Service (COS) field. While the label is used to forward the packet in the routers, the COS field selects the router interface's service queue. This mechanism enables the definition of determined routes and increases switching speed and efficiency [ZOS00]. Combining MPLS with differentiated services, the mechanism can provide QoS to a data stream. In this case, the core routers use the label and the COS field to forward the packets [XN99].

DetNet is a novel approach which is complementary to TSN. It focuses on zero packet loss and a deterministically bounded latency and jitter on the routing layer. This enables a reliable and redundant communication across LANs. DetNet specifies four flow types which are based on the required end-to-end latency and packet loss and sent via established point-to-point links. To maintain the requirements, time synchronization of the network devices and control parameters is needed. However, the research in this topic is not finished yet and the approach is still under standardization [NTA+18]. Hence, this work focuses on other techniques to enable real-time tests via the Internet even if DetNet may be suitable in the future. Those technologies are presented in the following.

3.5.2. State-Estimation

State-estimation represents one technique to improve the accuracy of a real-time image. It can be used if the observation frequency of the real-time entity is insufficient or if there are large communication delays in a networked control system. This work uses state-estimation to provide inputs of components in time and to increase the accuracy of simulation results even if there are large communication delays in the test system (cf. Section 5.1).

Several works exploit state-estimation to solve problems in the power distribution domain. On the one hand, the works of Naka et al. [NGYF03], Baran and Kelley [BK94], Hübner et al. [HKH11] and Mathieu et al. [MKC12] estimate load in a power distribution system. The estimation of these values can be used to reduce measurement costs as a major cost factor in the domain [MKC12]. On the other hand, Xu and Abur [XA03] propose a state-estimation algorithm which can be exploited to determine controller settings of flexible AC transmission systems.

Runtime verification of an application determines whether an execution trace fulfills a desired logical formula using emitted events. The introduced monitoring overheads can be reduced by leaving out some of the events. A disadvantage of this technique is missing events which might indicate a faulty execution. To prevent this issue, Stoller et al. [SBS+11] fill the gaps using estimated events. In another work, Weiss et al. exploit state-estimation for Micro Aerial Vehicle navigation. They provide estimated control inputs for real-time processing and as a fall-back solution if tracking errors occur [WAL+13]. Stettinger et al. [SHBZ14], Benedikt and Hofer [BH13] address the compensation of coupling errors in co-simulation using extrapolation. Similar to state-estimation, extrapolation of a signal provides unknown inputs in a co-simulation setup with bidirectional dependencies. Using a correction signal, the introduced estimation errors are reduced further by Benedikt and Hofer [BH13]. The proposed state-estimation mechanism of this work is similar to the last four works mentioned.

3.5.3. Speculative Execution

The concept of speculative execution is widely used in today's processor architectures to improve their performance and resource utilization. Besides them, there are different works which exploit the mechanism in other domains.

Increasing the performance of modern processors is based on a parallel usage of their resources. However, value dependencies on outputs of subsequent code or predicate dependencies on conditional statements limit the possible parallelism. Using speculative execution, alternative paths are computed in parallel based on different inputs. Examples are subsequent iterations in loops or alternative paths in if-statements. Since outdated values might be used or false branches may be executed, there must be a mechanism to restore the initial state. Furthermore, there must be a mechanism which discards results from the operations that did not occur. Alternatively, the execution is performed on a copy of the state. At the end of the path, the execution continues with either the original state or the copy [YMS94].

To support speculative execution in high performance processors, Smith defines a technology called boosting. It supports instruction-level parallelism across conditional branches while it prevents negative effects of incorrect speculations on the program state [Smi92]. Gabbay uses value prediction to improve parallelism by handling value dependencies [Gab96]. While Sahu and Adl-Tabatbai target speculative execution across synchronization barriers [SAT07], Knauth et al. save and restore event counters in their work [KRI+12]. Besides this, Younis et al. improve the performance of hard real-time systems without affecting the WCET. Their approach is based on code transformation performed by the compiler [YMS94].

In today's state-of-the-art, speculative execution can also be found in other domains. Nightingale et al. improve the performance in distributed file systems [NCF05] whereas Mickens et al. fetch web page data speculatively [MEHL10,MHL+12]. Finally, speculative execution is used to execute a back-up of slow tasks on alternative hosts in the parallel computing framework MapReduce. In this way, a job's execution time can be decreased by up to 44% [DG08].

3.6. Network-Centric Fault-Injection Tools

According to functional safety standards like ISO 26262 or IEC 61508, fault-injection should be used to ensure a system's safety even in the case of faults, errors and failures

[SFV18]. This section introduces examples of fault-injection tools which focus on the communication between systems. Furthermore, common message errors as defined in IEC 61508-2 [iec11] and CENELEC EN 50159 [en510] are presented.

There are different frameworks available which allow the injection of communication faults. The first example is called DOCTOR [HSR95] and can be used to analyze the effects of message loss in distributed real-time systems. While NFTAPE corrupts bits in the physical layer of a Myrinet LAN link [SFB+00], VirtualWire covers any network protocol [DNC03]. As stated in Section 3.1.1, the TTEthernet protocol is used for safety- and mixed-criticality systems. Fejoz et al. [FRMN18] consider link failures and transmission errors to validate the clock synchronization mechanism. Besides this, Onwuchekwa et al. [OO18] propose a framework which analyzes the influence of a babbling idiot failure on the latency and jitter of the network. In another work, they use an FPGA-based cut-through paradigm to inject faults into individual TTEthernet traffic classes [OOF18]. Finally, sfiCAN [GBBP13] and the work of Roque et al. [RPPF16] are two examples for physical fault-injection into the CAN network.

The IEC 61508 standard series covers the functional safety of electrical/electronic/programmable electronic safety-related systems [iec10]. In this series, part two focuses on requirements for electrical/electronic/programmable electronic safety-related systems and lists different message errors in clause 7.4.11. These errors are defined in more detail in the CENELEC EN 50159 standard for safety-critical data transmission in railway communication systems [en510]. The definitions are listed below.

Delay: A Packet is received later than expected.

Replay: The same packet is received multiple times.

Omission: A packet is removed from the message stream.

Insertion: A packet is added to the message stream.

Resequencing: The packet order in the message stream is changed.

Corruption: Bytes in the packet are changed to a new value.

Manipulation: The sender address in the packet is changed, hence the packet's authenticity is corrupted. In IEC 61508-2, it is named as *masquerading*.

Each error can be caused by system faults (unintended) or by an attacker (intended). This work covers only system faults since intended ones require the application of security mechanisms in the SUT. As explained in Section 4.5, faults related to the basic message errors are injected into the packets forwarded by the framework.

3.7. Security Mechanisms

During the last decades, several cryptographic algorithms and security protocols were developed to realize the concepts described in Section 2.6. The German Federal Office for Information Security (BSI) provides a technical guideline with recommendations regarding cryptographic algorithms and key lengths [bsi19]. Based on this document, suitable algorithms are presented for the concepts mentioned in the following. Furthermore, OpenVPN as an example for Virtual Private Networks (VPNs) is presented.

To realize confidentiality, symmetric and asymmetric encryption algorithms can be used. The recommended symmetric block cipher is AES with key lengths of 128, 192 or 256 bit since its security was analyzed intensively [bsi19, p. 21]. For asymmetric algorithms, the complexity of problems in algorithmic number theory defines the security level. A recommended algorithm is RSA (key length $>$ 2000 bit) which is based on the factorization of large prime numbers. Alternatively, discrete logarithms in the finite field $\mathbb{F}_p^*$ or on elliptic curves can be used. One example algorithm for $\mathbb{F}_p^*$ is DLIES (key length $>$ 2000 bit) while ECIES (key length $>$ 250 bit) is an example for the latter [bsi19, p. 26f.].

Cryptographic hash functions are used to ensure authenticity and integrity of data. Recommended algorithms are the SHA-2 and SHA-3 families with hash lengths of 256, 384 and 512 bit [bsi19, p. 37f.] which can be used to generate an HMAC (Keyed-Hash Message Authentication Code) of a message. Other possibilities to authenticate data and the communication partner are digital signatures using RSA (key length $>$ 2000 bit) or DSA. Similar to DLIES and ECIES, the latter is based on discrete logarithms in $\mathbb{F}_p^*$ (key length $>$ 2000 bit) or on elliptic curves (key length $>$ 250 bit) [bsi19, p. 43ff.].

The algorithms mentioned above are used in VPNs to secure the communication between the connected hosts. OpenVPN is a widely used example. It is based on SSL (Secure Socket Layer) which secures the communication on top of the transport protocol, e.g. TCP [EH97]. OpenVPN supports both, site-to-site and remote access VPNs, and multiple clients can connect to a private VPN server. Furthermore, the security algorithms and key lengths can be selected by the user [INC]. In this way, it provides all the required security services.

3.8. Research Gap

Although this chapter presents a large number of related work, there is no solution available which covers all the research objectives defined in Section 1.2. The main goal of this thesis is the development of a distributed co-simulation framework which facilitates the validation and verification of time-triggered networked control systems. To realize this, it works on a network-centric abstraction level and supports techniques such as SIL/HIL testing and fault-injection. Furthermore, a delay-management mechanism is required which mitigates Internet-introduced communication latencies for distributed real-time tests. This section defines the research gap which is addressed in the following chapters.

The frameworks presented in Section 3.2 focus mainly on different synchronization mechanisms and distributed co-simulation. Even if some of them cover network simulations in smart grids, they do not provide a generic interface to various simulation tools. Furthermore, the support of Hardware- and Software-In-The-Loop testing is missing. The FMI solves these aspects by providing an interface which is implemented by several simulation tools already. In addition to co-simulation, it can also support Software-In-The-Loop testing if the functions to execute a simulation step are implemented accordingly. Since FMI for co-simulation does not define a mechanism that synchronizes the tools, a lot of work has been done in developing different master algorithms. One possibility is the usage of the HLA which supports distributed co-simulation but requires a significant amount of code for integration. This code must be hand-developed which is tedious and error-prone [NGL+14]. However, a lack of frameworks which focus on the distributed validation and verification of time-triggered networked control systems including HIL testing and fault-injection remains.

Regarding the HLA, there are several concepts that can be extended to support distributed real-time tests. If geographically distributed manufacturers are involved, the communication cycles of the SUT may be smaller than the latencies between the simulation hosts. As a consequence, they may lead to deadline misses even if QoS techniques are applied. Since most of the real-time solutions cited in Section 3.2.3 are based on this approach, another mechanism for delay-management is required. It must cope with large delays leading to deadline misses in the following way: (I) detect the delays and stop the simulation or (II) cope with the delays' unpredictability. Using optimistic synchronization as an alternative time management mechanism improves the simulation performance but requires state recovery to handle message causality errors. In real-time tests, state recovery cannot be used wherefore causality errors must be prevented. Besides this, a hierarchical federation can reduce the number of messages exchanged between the federates and the RTI for time management and data exchange. Instead of concentrating on security and scalability, knowledge about the time-triggered communication can be exploited to maintain the deadlines in distributed real-time tests.

Most of the frameworks focusing on HIL and SIL testing introduced in Section 3.3 are limited to either SIL or HIL. The only exceptions are the dSPACE Rapid Prototyping system and ASAM XIL. However, the dSPACE system is a central unit which does not provide distributed tests. The same accounts for the ASAM XIL standard which is further dedicated to the automotive domain and supports only the CAN bus as the communication network. As a consequence, a distributed co-simulation framework is required that integrates both, SIL and HIL testing on a network-centric abstraction level.

While Section 3.6 introduces different fault-injection tools for communication systems, Section 3.7 covers suitable security mechanisms. Since the main focus of this thesis lies on different topics, using a VPN between the simulation hosts provides a sufficient level of security. In contrast, the fault-injection techniques presented cannot be used because they represent stand-alone tools and do not support all message errors of IEC 61508-2 clause 7.4.11. Hence, a means must be included into the framework which is capable of injecting faults related to all errors covered by the standard.

In the subsequent chapters, the distributed co-simulation framework is presented in detail covering all the research objectives mentioned before. It is based on the HLA and includes FMI as a generic interface to the simulation tools. Another possibility for data exchange would have been the distributed co-simulation protocol [Assa]. Indeed, it does not define sufficient synchronization and delay-management mechanisms. A master can synchronize non-real-time slaves by sending commands to execute a simulation step or to exchange data. However, synchronization with real-time slaves is performed only implicitly by using a time synchronization protocol such as IEEE 1588 [dcp19] and synchronizing the simulation start [KSK+19]. Besides this, there are no mechanisms specified that guarantee consistency and a timely packet reception for hard real-time slaves. It is only possible to detect dropped packets or an inconsistent transmission order using sequence numbers [KB18]. Providing data more frequently can be reached using an extrapolation technique, but the referred approach of Stettinger et al. is designed only for continuous system dynamics [SHBZ14]. This work shall be used for discrete-event simulations instead. For these reasons and since the protocol was standardized just in 2019, it is not considered during the design phase.

4. Network-Centric Distributed Co-Simulation Framework supporting HIL and SIL

4.1. Architectural Overview of the Framework

Providing a means to validate and verify geographically distributed SUTs is one main goal of the distributed co-simulation framework. Based on the High Level Architecture (HLA) simulation standard, it is able to connect subsystems via heterogeneous communication networks such as the Internet or LANs. This section gives an overview about the framework's overall architecture and introduces the Simulation Bridges as its main components.

4.1.1. Overall Architecture

Typical networked embedded real-time systems can be described by the composited architecture introduced in Section 2.1.3. A main benefit of this architecture is the possibility to develop components independently and to integrate them into a cluster using standardized interfaces. Hence, the internal implementation must not be known, only the interface specification is required [Obe11, p.26 f.].

During the last years, the integration of components using the Ethernet technology became more common in industrial applications (using Ethernet IP, ProfiNet), avionics (using ARINC 664-p7) or railway applications (using Ethernet Train Backbone (ETB), Ethernet Consist Network (ECN)). All of them try to achieve end-to-end transmission guarantees using a certain degree of Quality of Service (QoS). Deterministic time-triggered communication can be reached using the TTEthernet protocol which enables the usage of Ethernet in safety-critical systems [Obe11, p.182]. Besides TTEthernet, there is an ongoing interest in standardizing deterministic real-time capabilities in the IEEE Standard 802.1 for Higher Layer LAN protocols. The technology used is Time Sensitive Networking

(TSN) [LPS16].

The distributed co-simulation framework instantiates a test system that can be used to validate composited SUTs. It operates on a network-centric abstraction level and assumes the integration of the components via time-triggered communication networks. The developed framework focuses on time-triggered systems which use TTEthernet and TSN for communication according to the ongoing interest in these protocols. Exploiting the related network interfaces, the framework collects messages from a sender and forwards them to the receiving component. To forward the data inside the framework, heterogeneous communication networks such as the Internet or LANs are used. In this way, it is possible to validate the components at the sites of geographically distributed manufacturers. The support of Software- and Hardware-In-The-Loop (SIL/HIL) testing enables the validation and verification of components throughout common development processes further.

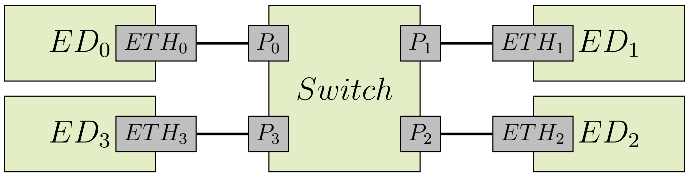


Figure 4.1.: Distributed SUT using Ethernet.

Figure 4.1 shows a distributed System Under Test (SUT) representing a simple, networked real-time system. It consists of four End Devices (ED) which are connected via a four-port Ethernet switch. Examples for such end devices can be sensors and actuators which interact with the system's environment or controllers. The latter process the sensor data to initiate an actuation. The gray boxes represent the network interfaces of the end devices (ETH_i) and the ports of the switch (P_i).

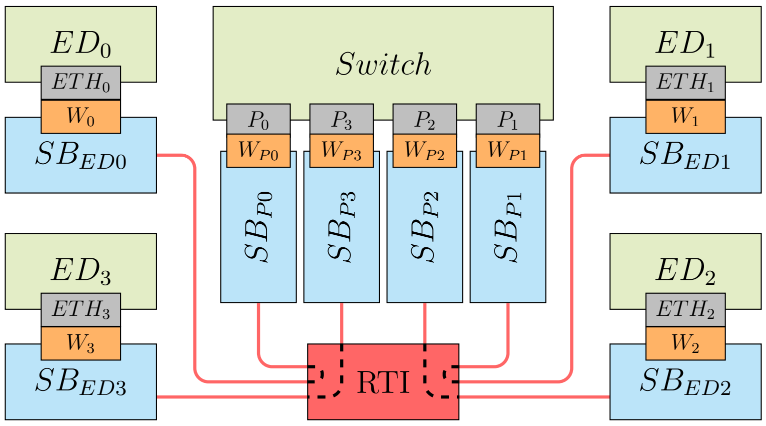


Figure 4.2.: Connecting the subsystems of an SUT using the distributed co-simulation framework.

The simulation framework operates on a network-centric abstraction level. Using it as shown in Figure 4.2, the black wires between ETH_i and P_i are replaced by two simulation bridges (blue boxes). To connect a device to a simulation bridge, the bridges provide an interface which is implemented in a Wrapper Module (orange boxes). The technology used to implement it depends on the properties of the simulation host on which the bridge is executed and its operating system. The connection is transparent which means that a bridge represents a device's communication partner. For example, an end device considers its bridge to be the switch it is connected to.

As the bridges operate on the OSI Data Link Layer, they capture the Ethernet frames sent by their connected devices and forward them to the receiving ones. To reach this, they use the HLA which is represented by its central component, the RTI. It is depicted as a red box while the logical links between a simulation bridge and the RTI are printed as red lines. In this example, there are only unicast links. To illustrate them in Figure 4.2, black dashed lines are added in the RTI box. However, the HLA also supports multicast data exchange through its publish-subscribe pattern. In this case, there would be multiple logical links between the bridges. Using the time management services of the HLA, the bridges ensure a packet transmission in the correct temporal order. In this way, they also synchronize the time advance of the simulations. Only if all packets for the related event are received, a time advance is granted and a simulation step is triggered.

While Figure 4.2 illustrates the connection of SUT subsystems from a logical point of view, Figure 4.3 presents a possible topology in a heterogeneous network. Its characteristics depend on the properties of the RTI used. The distributed co-simulation framework is based on an open source solution called *OpenRTI*. It uses TCP sockets for communication between the federates and supports a hierarchical RTI structure. Hence, it is possible to use it for simulations on a local host, in a cluster, across a LAN or via the Internet. Furthermore, the federates can be connected to local RTI instances, e.g. in a LAN or on a local host, which are then connected to a global instance on a higher level in the hierarchy. Connecting the federates to the global instance directly is further possible.

The topology shown in Figure 4.3 depicts the capabilities of the distributed co-simulation framework. The representation of end devices, simulation bridges, the RTI and its communication links is similar to the previous figure (cf. Figure 4.2). Each simulation bridge and the RTI instances are processes running on a simulation host. Those hosts are named as $Host_i$ and depicted as gray boxes. If a device is simulated or represents a software application, it is executed on the same host like its simulation bridge. Real hardware devices are connected to the host running the related bridge via an additional Ethernet interface instead. This connection is shown as a black, dotted line between ED_3 and SB_3.

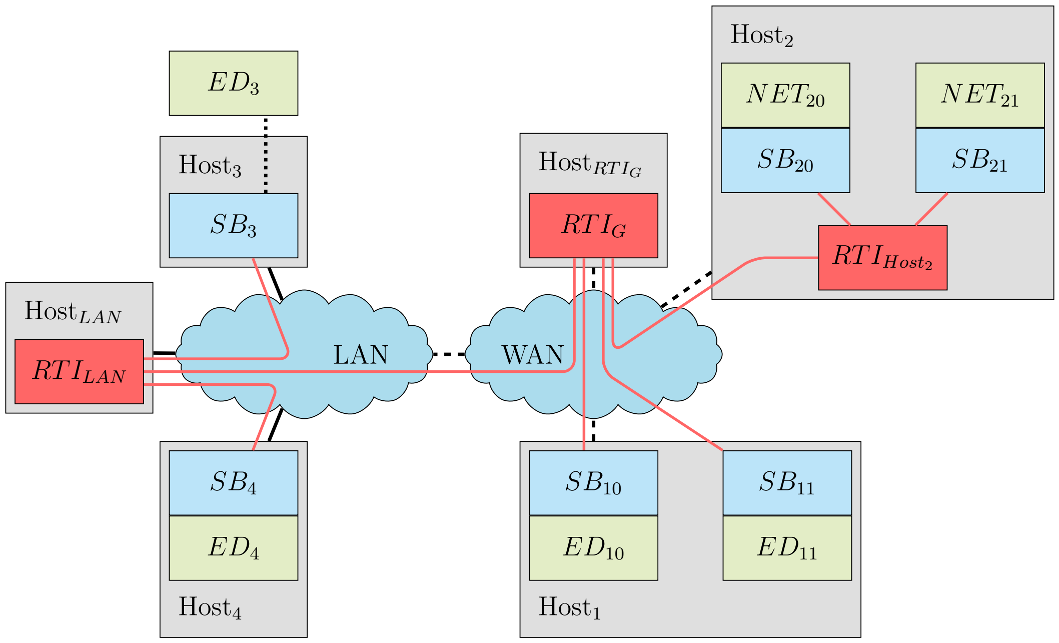


Figure 4.3.: Possible topology of the distributed co-simulation framework.

The simulation hosts in Figure 4.3 are connected via a heterogeneous communication network with a Local Area Network (LAN) and a Wide Area Network (WAN). The global RTI instance (RTI_G) is executed on $Host_{RTI_G}$ and there are two local RTI instances connected to it. One of them is running in the LAN (RTI_{LAN} on $Host_{LAN}$) while the other one is executed on $Host_2$ (RTI_{Host_2}). Besides the local RTI instances, there are two direct connections to RTI_G from SB_{10} and SB_{11}. They are both hosted by $Host_1$. $Host_2$ represents a host on which multiple network simulations are executed (NET_{20} and NET_{21}) with their own RTI. Each of them might simulate one or multiple TTEthernet or TSN switches, for example. In contrast, the end devices ED_3 and ED_4 are distributed via $Host_3$ and $Host_4$. They use the LAN to communicate with their local RTI instance. In the figure, physical connections between devices in the LAN are depicted as black, solid lines while dashed lines represent direct WAN connections.

Using a hierarchical RTI infrastructure can reduce communication load and speedup a simulation. This accounts for large simulation setups where devices mainly communicate locally in a LAN or on a host. Introducing a local RTI instance, those devices can communicate with it wherefore long network delays can be reduced and only messages related to remote devices have to be forwarded. As a single RTI instance would represent a bottleneck in such setups, scalability can be improved. However, this depends on the setup and the network delays between the devices. Synchronization data due to the HLA time management still has to be exchanged between all federates, wherefore an inappropriate topology may introduce further overheads.

4.1.2. Functionality of Simulation Bridges

After introducing the simulation bridges as the central components of the distributed co-simulation framework, this section focuses on their architecture. Figure 4.4 shows the bridges' building blocks and the data flow between the modules. Detailed descriptions for each module follow later in this document.

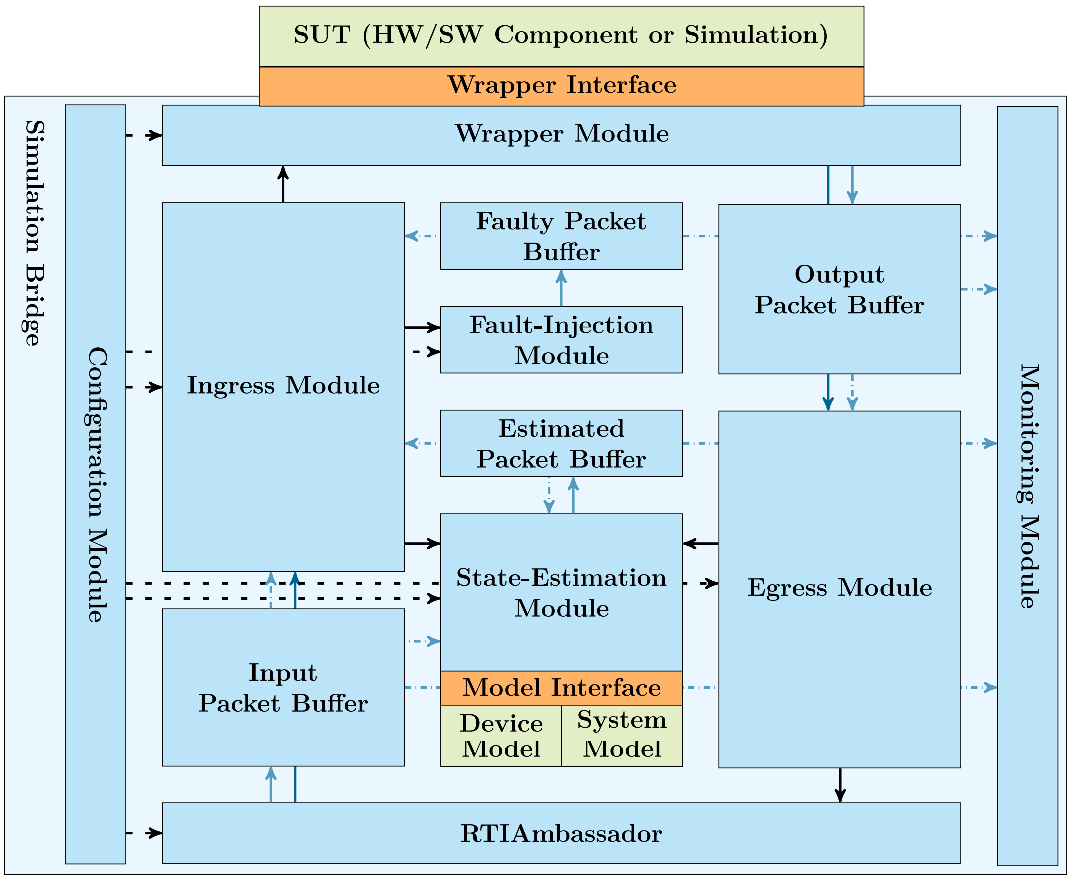


Figure 4.4.: Architecture of simulation bridges.

Each simulation bridge is connected to its device via a Wrapper Module. The wrapper forwards data received from other simulation bridges to the device and captures its output packets. Depending on the connected SUT and the implementation of the framework, different technologies may be used for the Wrapper Interface. Once the wrapper captures a packet, it encapsulates it into an HLA interaction including a time-stamp of the capture. The interaction class is determined based on the packet's MAC address. Afterwards, the wrapper inserts the packet into the Output Packet Buffer and signals the availability to the Egress Module.

To manage Internet-incurred delays, a mechanism based on state-estimation can be enabled. This mechanism estimates the next inputs of the device based on its captured outputs and a model of the remaining system under test. Once an output packet is inserted into the buffer, the wrapper notifies the Egress Module about its availability. The latter gets the packet from the buffer and forwards it to the State-Estimation Module. This module estimates the next inputs and inserts them into the Estimated Packet Buffer. After finishing the estimation, the output packet can be sent via the RTIAmbassador and the HLA. If the state-estimation mechanism is disabled, the packets are sent directly. The egress module further requests a time advance to the next event using the HLA's *NextMessageRequest* service. In this way, the simulation bridge is able to react to unscheduled events, e.g. the reception of an event-triggered message.

On the incoming site, input packets are received by the RTIAmbassador before a time advance is granted. The ambassador decapsulates the interaction parameters and creates an interaction object. This object is inserted into the Input Packet Buffer and the ambassador notifies the Ingress Module. The ingress module provides different functionalities depending on the configuration of the simulation bridge. Without using the state-estimation mechanism, the module checks whether the network delay exceeds a configurable time interval. For this, the instant of capturing the packet is subtracted from the time when the packet was received. If the threshold is exceeded, the user is notified and the packet is dropped. Otherwise, the ingress module forwards it to the fault-injection module.

Using the state-estimation mechanism, there are two possible functionalities. In the first case, the connected device is a real-time device. Here, the ingress module is time-triggered exploiting knowledge about the device's communication and application schedules. Each event is represented by its logical time. Adding it to the time of the simulation start results in the physical point in time when the event has to be scheduled. Once the event occurred, the ingress module checks if a packet is available in one of the buffers and forwards it after injecting faults. Afterwards, a simulation step follows. If a packet is delayed and cannot be found in the input packet buffer, the estimated packet is forwarded. In this way, it is possible to provide guarantees about the message reception to the device (cf. Section 5.1.2). The second case can be used further to reduce the communication overhead with the RTI and other devices which improves the simulation's performance. Here, a larger bridge communication period is defined when packets are exchanged with other bridges. In between, the simulation bridges estimate their devices' inputs and forward these packets at the correct point in time. This mechanism is described in Section 5.1.3. In both cases, the accuracy of the forwarded data must satisfy a configurable threshold. After receiving a packet, the state-estimation module calculates the difference between the received and the estimated values. If the difference exceeds the threshold, the user is notified and able to decide if the simulation must be stopped.

Fault-injection is provided based on the message errors listed in IEC 61508-2 clause 7.4.11 (cf. Section 3.6). The message errors are implemented in the Fault-Injection Module which injects the faults into input packets of the connected component. According to the architecture of the distributed co-simulation framework, the components of the SUT (switches, end devices, etc.) are considered as Fault-Containment Region (FCR). Hence, the injection implies the propagation of an error from a faulty sending component via a message. The required parameters for each fault can be defined in the configuration data based on the logical time of the received packet. At runtime, the fault-injection module checks whether there is a fault available related to the current time. If this check evaluates to true and the packet is not omitted, the fault is injected. The injection is described in detail in Section 4.5. Delayed and replayed packets are inserted into the Faulty Packet Buffer and a new event is created for the related time. While resequencing can be implemented by delaying the packets by a different time interval, corruption and manipulation can be performed directly on the packet. Once the injection is finished and the packet is not omitted, the ingress module can trigger the wrapper module to forward the packet to the device.

By creating new events for the delayed and replayed packets, they are also considered during the time synchronization process. After receiving a *TimeAdvanceGrant*, the RTI-Ambassador notifies the ingress module. As there might be delayed, replayed or inserted packets available, this module checks the faulty packet buffer also, forwards available packets and triggers a simulation step via the wrapper module.

Since the simulation bridges provide different functionalities, a configuration mechanism is required which sets the related parameters. To reach this, the bridges contain a Configuration Module. Before the simulation starts, this module reads the configuration data and sets the related parameters in the other modules. Hereby, it performs a check on integrity and coherency already. The data includes, e.g., information about the devices, their schedules, faults to be injected into the communication or the RTI connections. To support an automated simulation execution, it is possible to start a new simulation run once the previous one has finished. In this case, the execution is repeated using new configuration data. After finishing the simulation, it is further necessary to access the buffers and check if the functionality of the SUT was correct. Hence, a Monitoring Module has access to the functions of each buffer which allow to access every packet inserted. Gained information may be the content of each packet, the time-stamps of the packet capture and the reception or information about faults. The monitoring module writes this data into files which can be checked by the user once the simulation has terminated.

4.2. Configuration and Simulation Execution

To configure the framework, two configuration models are required. The first model is related to the HLA and defines the object model according to the IEEE standard 1516.2-2010 [iee10c]. The second model is used to configure the simulation bridges.

The main parts of the Object Model Template (OMT) are the federates' object classes including the related attributes and the exchanged interaction classes with their parameters. In addition, the template contains properties such as dimensions, transportation types or update rates, among others. This information is used to configure the RTI via a Federation Object Model (FOM). Since only interactions are exchanged between the simulation bridges, solely those properties are described in the following.

By default, each interaction class is derived from the *HLAinteractionRoot* class building a hierarchy. The only child class of the root interaction is called *EthernetInteraction* and represents the Ethernet frames exchanged by the framework. If the framework shall support another Layer 2 protocol in the future, a related interaction class can be added in this hierarchy level. The parameters of the *EthernetInteraction* are the packet, its size and the egress time-stamp in microseconds. The latter represents the time when the packet was captured by the simulation bridge. The parameters are defined as child elements together with the child interaction classes. In this case, the child classes are related to the interactions exchanged between the simulation bridges which depend on the SUT.

Figure 4.5 introduces the configuration and initialization process of the simulation bridges. Similar to the FOM, they are configured via a model. Its detailed definition is given in Appendix A. At first, the connection with the RTI is configured and established in the *RTI Connection* state. Afterwards, the configuration of the remaining modules follows in the *Configuration* state. The configuration module starts with reading the configuration data and checking its integrity. Each simulation bridge can be connected to one or more federations for the application data exchange. Additionally, they are all members of a federation to exchange synchronization meta data and control commands. If the model is correct, the RTIAmbassador for each federation is configured including information such as the RTI's IP address or the FOM. In the next step, the simulation bridges establish the connections to the RTI instances and join the related federation executions. Those federation executions are created by the first bridge requesting the creation. Afterwards, the ambassadors start a callback handler which receives the RTI replies. In this way, the configuration effort can be reduced since some members require data provided by the RTI. If one step failed, the other calls are unrolled in the *Disconnect* state and the simulation bridge terminates printing a related error message.

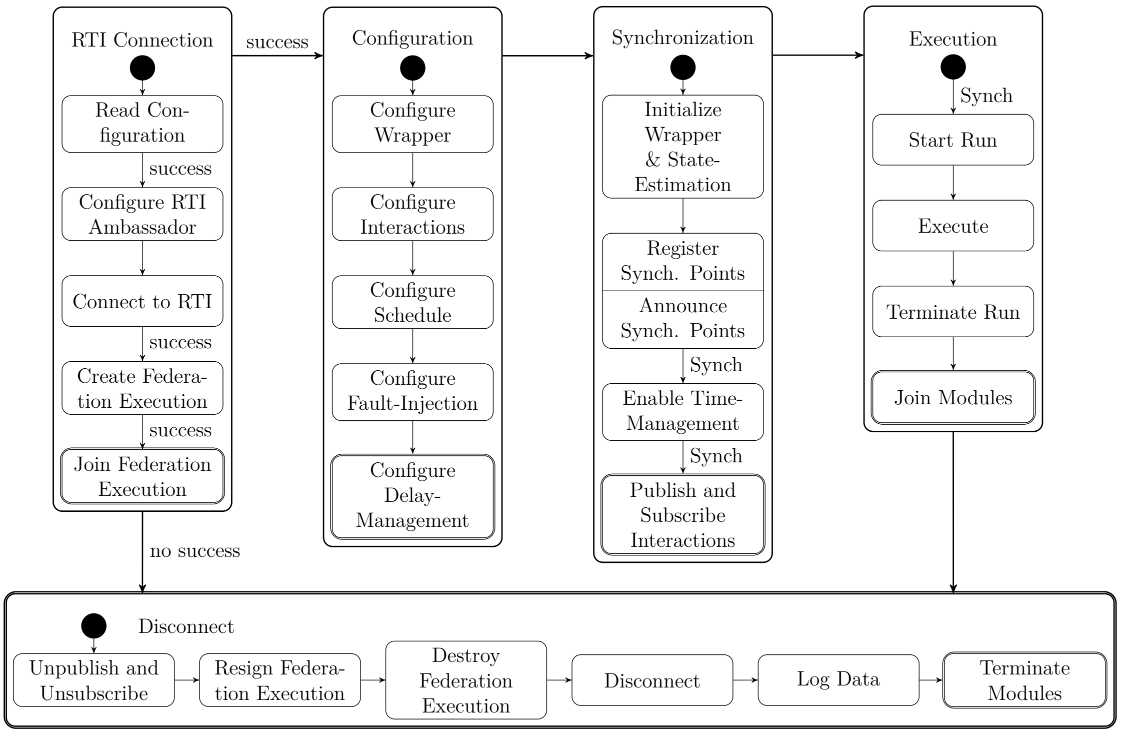


Figure 4.5.: Simulation bridge configuration and simulation execution.

If the federation execution is joined successfully, the *Configuration* state follows. First, the configuration module determines the wrapper type and sets its parameters. Afterwards, the module configures the interactions the simulation bridge publishes or subscribes to in the ambassadors. For each interaction, an *EthernetInteraction* object is created which is related to the interaction class in the FOM. In addition to the interactions where the destination is defined, there are two default interactions for sending and receiving. Once an *EthernetInteraction* object is added to the ambassador, the ambassador is able to request the handles for each parameter defined in the FOM from the RTI.

Task
ID
Period
Offset in Period
WCET
Real-Time Offset
Period (BCS)
Offset in Period (BCS)
Input Message IDs
Output Message IDs

Message
ID
Period
Offset in Period
Length
Real-Time Offset
Period (BCS)
Offset in Period (BCS)
Type

Figure 4.6.: Parameters of tasks and messages.

The following activities are related to the definition of a time-triggered task and message schedule. Tasks and messages are represented as classes (cf. Figure 4.6). Their members are their IDs, their period and the offset in this period, the length (for messages) and WCET (for tasks), a real-time offset (used for estimation) and period and offset used for a Bridge Communication Schedule (BCS). In addition, messages have a type (sending or receiving) and tasks have references to their input and output message IDs. Based on this information, a schedule can be calculated which is used for synchronization (cf. Section 4.3), fault-injection (cf. Section 4.5), state-estimation (cf. Section 5.1) and speculative execution (cf. Section 5.2).

Table 4.1.: Configuration parameters of message errors.

Error Mode	Parameters
All Errors	Message ID, Logical Time
Omission	
Corruption & Manipulation	Number of Changed Bytes, Changed Indexes, Changed Values, Recalculate Checksum
Replay	Replay Instants
Delay & Resequencing	Delay
Insertion	Source IP, Destination IP, Transport Protocol, Source Port, Destination Port, Data

Besides the schedule, the fault-injection module requires information about each fault. The parameters are summarized in Table 4.1 following the error modes of IEC 61508-2 clause 7.4.11. They are set in the related message objects. To identify the faults at runtime, the fault-injection module requires the message's ID and the logical time of the instant at which the fault has to be injected. These instants are stored in fault-schedules for each error mode. In case of omission, this data is sufficient whereas the other message errors require additional information. Since manipulation is a specification of the corruption error, they can be modeled in the same way using the same parameters. Those are the indexes of the bytes to change in the packet, their number and the modified values. Additionally, a boolean value is provided which states whether the packet's CRC checksum has to be recalculated or not. Another list is provided for the replay error mode. Each instant in this list represents the logical time at which the replay occurs. Resequencing two or more packets can be implemented exploiting the delay error. Both modes use an integer value which represents the delay interval for each packet. Finally, the simulation bridges support the insertion of additional packets. The previous errors use the time-stamp to identify the message at runtime. Instead, packets for this mode are created and inserted into the faulty packet buffer directly during the configuration.

Furthermore, another reception event is added in the schedule. Creating a packet requires the source and destination ports, IP and MAC addresses. In addition, the content of the packet and the transport protocol are provided. The remaining packet content can be calculated automatically.

To realize delay-management for data exchange via the Internet, there are three possibilities. The first is a simple check whether the delay is larger than a configurable threshold (*maxDelay*). Using state-estimation is the second possibility. In this case, two models are used to estimate the next inputs (system model) and to compare the received with the estimated contents (device model). The configuration data contains information about the models and the interface which is used to connect them to the state-estimation module. Speculative execution as the third possibility is configured by defining the federations the simulation bridge is connected to.

The wrapper module is initialized in the *Synchronization* state after finishing the configuration. Additionally, the state-estimation module is started if it is enabled. Before the simulation bridges can continue with the publish/subscribe process, the HLA time management must be enabled. If there are simulation bridges connected to multiple federates, deadlocks occur if the time management is enabled in one bridge and another bridge joins a federation execution. Using synchronization points solves this issue. A synchronization master is the last federate connected to the control federation. The master is responsible to register the synchronization points while the other bridges are waiting for the points' announcement. Since they are aware of the points due to the configuration file, they do not continue their execution before all points are announced. Once the announcement has finished, the points establish three synchronization barriers which synchronize the federates. After they reached the first barrier, the federates can enable the time management (*EnableTimeManagement*). The second barrier is used to synchronize the publish/subscribe process (*PublishSubscribe*) while the last barrier starts the simulation execution (*RunSimulation*) in the *Execution* state.

Depending on the configuration, the ingress and egress modules start in parallel using one of three different functions. The first pair is related to a connected device without real-time requirements and disabled state-estimation (*run*). Using the state-estimation mechanism, the function *runEstimation* is used. If the device is a real-time component, the modules start the function *runRT*. In all cases, the ingress and egress modules take a time-stamp which is used to calculate the duration of the simulation execution and as a starting reference for real-time activities. Once the simulation execution has finished, all parallel modules are joined and they obtain the related time-stamps for the end of the run. Furthermore, the simulation bridges calculate the simulation duration and the ambassadors stop the callback handlers.

After stopping the callback handlers, the interactions are unpublished and unsubscribed in the *Disconnect* state. The RTIAmbassadors disconnect from the RTI and the last bridge deletes the federation execution. Apart from that, the wrapper and the state-estimation modules are shutdown and the monitoring module logs information about the run. This includes the simulation duration, pass-through times of the simulation bridges and the ingress/egress modules and their scheduling delays among others.

4.3. Time Synchronization

This section focuses on the synchronization of simulation bridges in the distributed co-simulation framework. They provide two different modes of execution which are based on the HLA time management. The first mode synchronizes the bridges based on a logical simulation time while the second mode supports the execution in real-time. Before these algorithms are introduced, an explanation of the time management services is given including a detailed example.

4.3.1. Time Management in the High Level Architecture

To realize time synchronization between the simulation bridges, the framework exploits the time management services provided by the HLA. The explanation given in this section is based on Chapter 8 of the HLA standard's federate interface specification [iee10a].

Time management is used to realize a consistent delivery of messages during the simulation execution. Points along a time axis can delineate a federate's logical time if it assigns time-stamps to messages. While executing, a federate can advance to a logical time which is greater than or equal to its current time. Each time advance can be constrained by other federates which ensures a correct and causally ordered data exchange.

Updates of attributes or the transfer of interactions are both considered as messages. Including time-stamps into the messages, a federation-wide Timestamped Order (TSO) can be established. By default, a joining federate does not use the time-management. This state can remain as long as the federate does not require coordination with other federates. Otherwise, it has to enable the time-management using the *EnableTimeRegulation* and *EnableTimeConstrained* services. A time-regulating federate is able to send messages including time-stamps while federates have to be time-constrained to receive them.

The RTI establishes the TSO by placing a bound called Greatest Available Logical Time (GALT) on each time-constrained federate. It represents the greatest logical time to which those federates can advance without receiving a message in their past. The GALT is calculated individually for each federate using their current logical time, the time advance requests and a value called lookahead. The lookahead has to be set by time-regulating federates once they request to become regulating. During execution, they are only allowed to send messages with a time-stamp which is greater than or equal to their current requested time plus the lookahead. Only if the requested time is smaller than the GALT value, the RTI can guarantee that the federate will not receive messages in the past. If the federate tries to advance beyond the bound, it has to wait until the GALT has increased based on the time advances of the regulating federates.

Federates which intend to advance in time must use the *TimeAdvanceRequest* or *NextMessageRequest* services including the requested time. Once the GALT value fulfills the constraint explained above, the RTI replies with a *TimeAdvanceGrant* whereby the granted time included depends on the request used. A *TimeAdvanceRequest* is granted with the requested time in every case and the RTI forwards all TSO messages with a time-stamp smaller or equal to it. Hence, it is used for time-stepped federates where the time-step is known and there are no messages sent in between. Using a *NextMessageRequest*, the RTI may grant a time advance to an earlier time than the requested one if a message is received at this point. As this service enables a federate to react on previously unknown events, it is used for event-stepped federates.

The following scenario illustrates the time management services on a simple discrete-event based application. This application consists of two end devices (ED) and one network simulation (NET). For example, one end device (ED_0) could host a sensor and an actuator while the other one (ED_1) could represent a related controller. ED_0 periodically sends a message M_0 to ED_1 containing the current sensor value. Based on it, the controller calculates the control input for the actuator and replies with M_1. The actuator on ED_0 can now set the received control input performing an actuation. In this example, data is sent via the network simulation in both directions. To be able to react on receiving events, time advances are requested using the *NextMessageRequest* service.

In each figure of the scenario (cf. Figures 4.7 to 4.12), there are three time-lines representing the logical times (LT) of each federate: the two end devices as well as the network simulation. The messages (events) are denoted as arrows. While sent messages are printed in dark blue, the received ones are painted light blue. One can see that each sent message has a related received message at the same logical time. At tick 2 in each period, ED_0 sends its message to the network simulation (NET). After emulating the communication behavior, the network simulation forwards the message to ED_1 at tick 4. ED_1 calculates the control input and sends it to the network simulation at tick 6. Again, this message is forwarded to the receiving federate ED_0 two ticks later. At tick 10, the period has finished and the next period starts. While the green spots denote the current logical times of each federate, the red ones represent the requested times for a time advance. Because of the lookahead, the requested times are always one tick before the next sent message. At last, the GALT is printed as solid black line from the initial time until the black circle. The GALT value represents the logical time until when it is guaranteed that the federate will not receive any TSO message and is calculated individually for each federate.

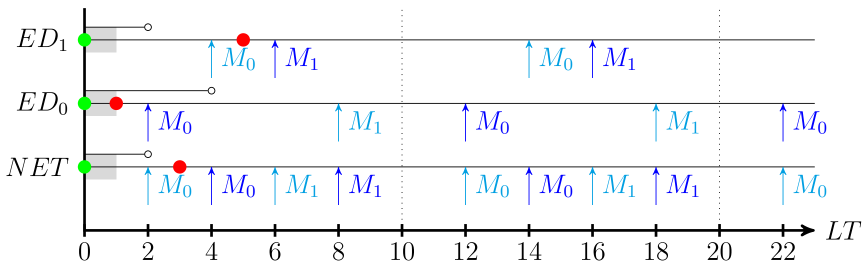


Figure 4.7.: Initial step in the time management algorithm.

Figure 4.7 shows the initial step of the time management algorithm for the example use-case. The logical times for each federate are 0 and they all request to advance until they plan to send the next message (respecting the lookahead). Hence, ED_1 guarantees not to send any message until tick 6, ED_0 until tick 2 and NET until tick 4. Based on those guarantees, the GALT values are calculated as follows: for ED_1 and NET, the value is set to 2 since then they might receive a message from ED_0. For ED_0, the GALT equals 4 because of NET.

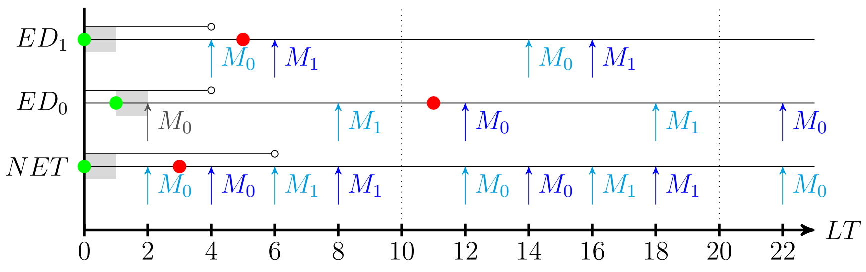


Figure 4.8.: First step in the algorithm, time advance of ED_0.

In the first step shown in Figure 4.8, the RTI can grant a time advance for ED_0 to logical time 1. The federate calculates the new sensor value and sends it in M_0 with time-stamp 2. In the figure, the color of event M_0 is changed to gray to outline that this event is processed already. Federate ED_0 has no further message to send in this period, so it can request to advance until tick 11 when the next sensor value needs to be sent.

Since NET has subscribed to message M_0, the RTI grants an advance to tick 2 and forwards the message to the federate (cf. Figure 4.9). Using the input, NET emulates the network behavior and sends M_0 to the receiving federate ED_1 with the assigned time-stamp 4. Initially, M_0 should have been emulated at tick 3. As the emulation is performed already after receiving the message, the time advance to this tick is obsolete. An alternative usage of the time management would be to just receive M_0 at tick 2 and to request another time advance to tick 3 where the event is processed. In this example, the first version is assumed which is why NET already requests an advance to tick 7.

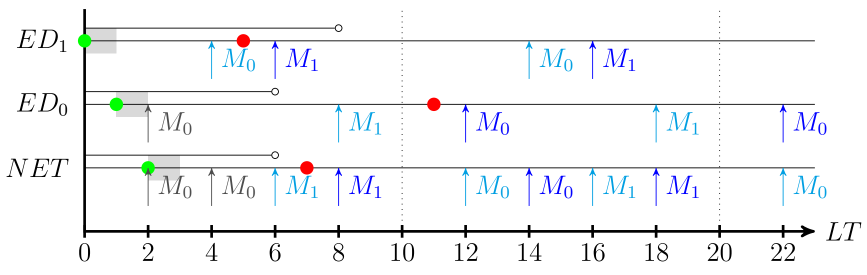


Figure 4.9.: Time advance of NET as the second step.

Similar to the previous steps, ED_1 is granted to advance to tick 4 where it receives message M_0. Afterwards, it sends M_1 containing the calculated control input to the network simulation with time-stamp 6. Its next time advance request is for tick 15 in the next period (cf. Figure 4.10).

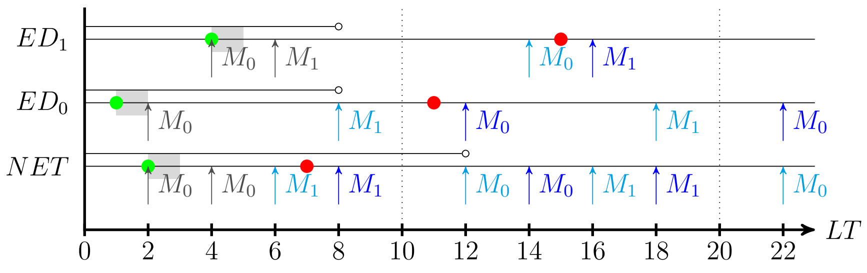


Figure 4.10.: Third step showing the time advance of ED_1.

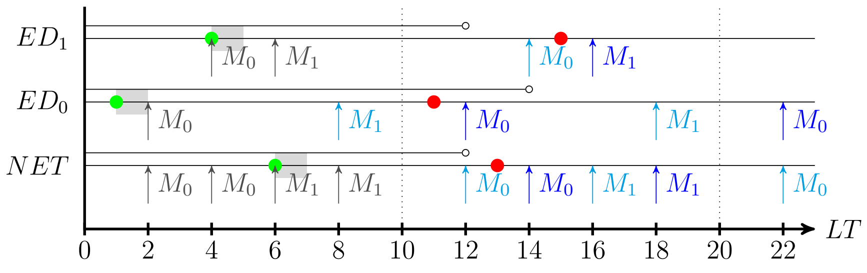


Figure 4.11.: Fourth step in the time management algorithm, time advance of NET.

In the fourth step (cf. Figure 4.11), the network simulation receives message M_1 at tick 6 and forwards it with time-stamp 8 after simulating the network behavior. The federate further requests to advance until tick 13.

Message M_1 is received by ED_0 at tick 8. After setting the control value and calculating the next sensor value, the latter is sent in M_0 with time-stamp 12 for the next period. The current status of each federate as shown in Figure 4.12 is similar to step one. Hence, there is a cyclic behavior from now on.

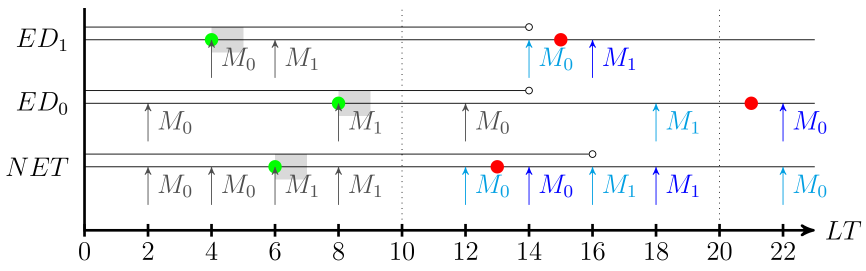


Figure 4.12.: Fifth step with time advance of ED_0.

The figures above show the contrast of the *NextMessageRequest* (NMR) service compared to *TimeAdvanceRequest* (TAR). Using TAR, the RTI would grant only time advances to the requested logical times and transmit the messages when it grants the advance. In this way, the federate is able to react on messages only when the advance is granted and not before. Hence, it is not possible to model a system where the federate's behavior may depend on events such as Best Effort traffic in TTEthernet. The NMR service solves this problem since the RTI also grants time advances to logical times of received messages if they occur earlier than the requested time. However, this may cause multiple invocations of the NMR service by the federate to advance in time.

Based on this example, the next section illustrates the basic time synchronization mechanism of the simulation bridges which uses a logical simulation time.

4.3.2. Synchronization of Simulation Bridges

During normal simulation execution, the bridges jump between events defined in an application and communication schedule. This schedule is calculated based on information about the SUT's time-triggered messages and tasks provided in the configuration data. For each message and task, the schedule determines the number of events through dividing the duration of the simulation execution by the task's or message's period. Afterwards, it calculates the instants considering the offset in each period and stores them together with scheduling information. Regarding the normal execution mode, the important scheduling information is the event's type (message or task) and a reference to the related task or message object. This object stores further information such as the parameters for fault-injection. The remaining scheduling information is used for the other modes.

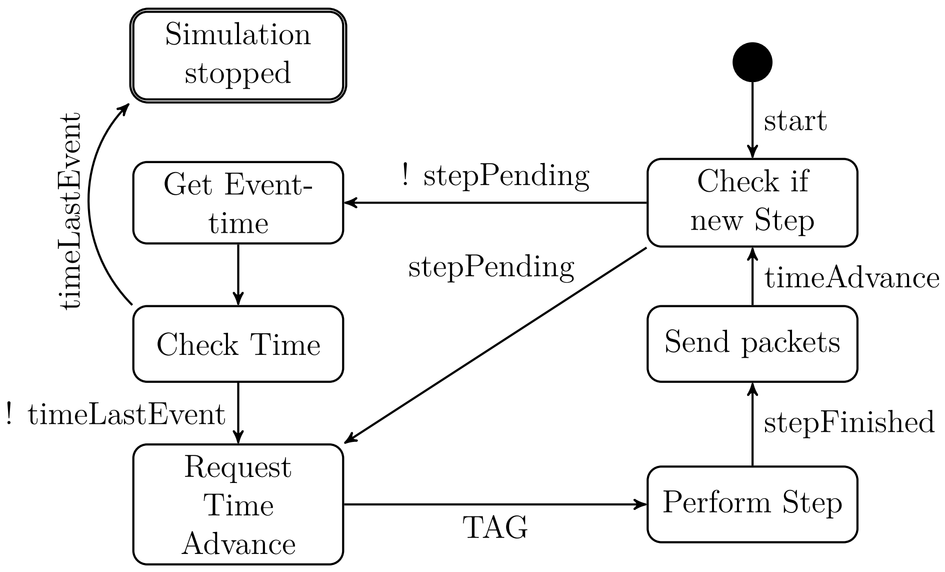


Figure 4.13.: Main loop in the ingress module.

Figure 4.13 illustrates the synchronization algorithm. At the beginning, the egress module checks if a step needs to be performed. This is signaled by the flag *stepPending*. To support unknown events in addition to the communication and application schedule, the simulation bridges use the *NextMessageRequest* service. Hence, it is possible that the RTI grants a time advance to a time before the requested event. One example is the reception of an event-triggered message which causes the execution of a task sending additional messages. In this case, there might be pending internal or additional, unknown events from other devices and the request must be repeated. Otherwise, the RTIAmbassador obtains the next event from the schedule and checks its time. The last event in the schedule equals the time of the simulation's end. If this value was returned, all events are processed and the ambassador terminates the simulation. If not, the module sends the *NextMessageRequest*.

Before the RTI grants the time advance to the simulation bridge, it sends all subscribed interactions for this event. The simulation bridges forward them to the connected device as described in Section 4.4. Once the *TimeAdvanceGrant* is received (*TAG*), the ingress module triggers the execution of the next step. Its end is represented by the flag *stepFinished* which enables the RTIAmbassador to send the available outputs. Afterwards, the next time advance is triggered and the loop repeats.

4.3.3. Synchronization in Real-Time

Compared to the basic synchronization mechanism, the execution in real-time for HIL testing is more complex, because its continuous advance must be considered further. This section introduces the related synchronization mechanism. For a proper functionality, it requires the execution on a simulation host which is configured to run real-time applications as described in Section 6.2.2. This includes a synchronized global time for the simulation bridges and the connected devices. The logical times in the HLA are synchronized globally according to the time management. Furthermore, the algorithm expects the availability of input packets in time. If the network connecting the bridges introduces large delays, the management techniques introduced in Chapter 5 have to be applied.

Algorithm 1 Real-time execution in the ingress module

```
 1: procedure RunRT
 2:     set bool terminated to false;
 3:     declare packetPtr;
 4:     while not terminated do
 5:         wait for next activation;
 6:         take time-stamp and compare it with the time of the next scheduled event;
 7:         if scheduled event occured then
 8:             if TT-packet found in inputPacketBuffer then
 9:                 set packetPtr to input packet;
10:             else if TT-packet found in faultyPacketBuffer then
11:                 set packetPtr to faulty packet;
12:             end if
13:         else if non-scheduled event occured then
14:             if ET-packet found in inputPacketBuffer then
15:                 set packetPtr to input packet;
16:             end if
17:         end if
18:         if packet found then
19:             inject fault into packet if configured;
20:             if not (omission or delay failure) then
21:                 forward packet of packetPtr;
22:             end if
23:         end if
24:     end while
25: end procedure
```

Algorithm 1 illustrates the real-time operation mode of the simulation bridges which is implemented in the ingress module. Detecting the correct point in time to forward a packet requires the information provided by the device's application and communication schedule. The logical times in the schedule represent the logical duration from the simulation start to the time of the event. At runtime, this duration is added to the time-stamp of the simulation start to map the logical times to physical points in time. The other way around, each physical time interval is represented by a logical time similar to a sparse time base. The resolution of the mapping is based on the temporal requirements of the SUT. For example, a logical duration of 1 may represent $1ms$. This resolution also represents the period with which the simulation bridge is activated. After each activation, the bridge takes a time-stamp and compares it with the physical time of the next scheduled event (Line 5-6). If the activation time-stamp is equal or larger (due to wake-up delays) than the physical time of the event, a scheduled event has occurred. In this case, the module forwards available time-triggered packets (Lines 7-12). Since the ingress module aims on forwarding data from other devices, the input packet buffer is checked first (Line 8). If there is no packet found, delayed, inserted or replayed packets may be available. Hence, the faulty packet buffer follows (Line 10). In between the scheduled RT-events, it is possible to select available event-triggered messages for the forwarding (Lines 13-16). If a packet was found, the injection of faults is triggered and non-omitted packets are forwarded to the device (Lines 18-23).

Algorithm 2 Advance in time for real-time execution for the egress module

```
procedure AdvanceInTimeRTegress(int64 logicalTimeEvent)
    set logicalTimeFinishedEvent to logicalTimeEvent;
    if logicalTimeFinishedEvent == logicalTimeLastEvent then
        set rtSimulationFinished to true;
        if simulationFinished then
            terminate simulation and return;
        end if
    end if
    if grantedTime == logicalTimeFinishedEvent then
        get next event from schedule;
        if event found then
            call NextMessageRequest;
        else
            terminate simulation and return;
        end if
    end if
end procedure
```

During real-time execution, the logical time of the HLA federation must be synchronized with the physical time of the device. This makes the synchronization with the RTI more complex than in the normal execution mode. While the basic time management message exchange with the other simulation bridges remains the same, the progress of real-time must be considered further. On the one hand, the simulation bridge must wait for the device to finish an execution step and to provide all outputs (egress module). On the other hand, the simulation bridge cannot request a time advance if the step is finished, but the RTIAmbassador is still in the time advancing state.

Algorithm 2 depicts the function *AdvanceInTimeRTegress* of the RTIAmbassador which is called once the simulation bridge has sent all outputs for the device's current task. It requires the logical time of the current, finished event and updates a member variable accordingly (Line 2). Afterwards, it compares the time of the finished with the time of the last event. A match of both time-stamps signals the end of the real-time execution which is represented by setting *rtSimulationFinished* to true (Line 4). If all events are granted by the RTI, the simulation has also finished (*simulationFinished*) and the simulation bridge can terminate (Line 6). Otherwise, the function compares the granted time with the time of the finished event. Only if the granted time is equal, a request to the current event has been granted by the RTI and the device has finished the related task. The RTIAmbassador can obtain the next event from the application schedule and request a time advance if the event exists (Lines 9-12). A non-existing event represents an error since the last event of the simulation is available in the schedule and the function checks this case at its beginning. Hence, the simulation bridge also terminates (Line 14).

Algorithm 3 Advance in time for real-time execution in the RTIAmbassador

```
 1: procedure ADVANCEINTIMERTINGRESS
 2:     if grantedTime == logicalTimeLastEvent then
 3:         set simulationFinished to true;
 4:         if rtSimulationFinished then
 5:             terminate simulation and return;
 6:         end if
 7:     end if
 8:     if grantedTime $\leq$ logicalTimeFinishedEvent or rtSimulationFinished then
 9:         get next event from schedule;
10:         if event exists then
11:             call NextMessageRequest;
12:         else
13:             set simulationFinished to true;
14:             if rtSimulationFinished then
15:                 terminate simulation and return;
16:             end if
17:         end if
18:     end if
19: end procedure
```

Algorithm 3 is used to request a time advance from the RTIAmbassador to handle the case if the HLA time synchronization is slower than the real device. During normal operation, a *TimeAdvanceGrant* message triggers the execution of a simulation step. Since the real device runs independently from the simulation bridge, such a trigger is not necessary. As a consequence, the RTIAmbassador can call the function *AdvanceInTimeRTingress* directly once the bridge receives a *TimeAdvanceGrant*. The main differences to Algorithm 2 are the conditions under which the next time advance can be requested and the simulation bridge can terminate. A termination is possible if the currently granted time equals the time of the last event and the real-time execution has finished also (Lines 2-7). If the latter is not true, the other components were faster than real-time and the simulation bridge has to wait until all events are finished by the real device. In this case, no further *NextMessageRequests* are sent. To advance in time, the time management algorithm must be processed as fast as the real-time execution or slower. This is represented by a granted time which is smaller or equal to the currently finished event. This condition is sufficient for the normal execution but may cause a deadlock if the real-time execution has finished already. Hence, a request is also sent in this case (Lines 8-11).

4.4. Packet Handling in the Simulation Bridges

While Section 4.1.2 gives an overview about how packets pass the simulation bridges at runtime, this section focuses on a detailed explanation about handling the packets. It covers the packet capture in the wrapper module of the sending simulation bridges and the forwarding process in the receiving ones.

How packets are captured depends on the connection of the device to the simulation bridge in the wrapper module and the implementation of the wrapper interface. Captured packets are stored in objects of the type *EthernetInteractionInstance*. Besides the packet itself, these objects contain meta data about it such as the logical time, egress and ingress time-stamps or injected faults. Once a packet is available, the wrapper function *handlePacket* creates the interaction instance object and sets the egress time-stamp which represents the time the packet was captured. Afterwards, the packet is analyzed to determine the related interaction class, the packet's size, its logical send-time or if the packet is the last one for the current simulation step. To obtain the latter two values, an additional protocol header can be added to each packet. This header can be placed between the transport (e.g., TCP/UDP) and the application protocol (e.g., TRDP [iec15]) or on top of the latter. It is not required if the temporal information is contained in the other headers already. The same applies if the device and the bridge are timely synchronized and the communication schedule is known. The packet size can be calculated by exploiting the length values in the Ethernet and IP headers. In Ethernet networks, the destination

MAC address determines the receiving network device. Besides communication with one device (unicast), it is also possible to perform multicasts (a group of receivers) or broadcasts (all devices in a network). The simulation framework provides two mechanisms to map the destination MAC address to an HLA interaction. First, an interaction class can be defined which is dedicated to a unique destination address. The second possibility is the usage of a default interaction class which is destined to any address. At runtime, the wrapper analyzes the address and sets the interaction class of the interaction instance object accordingly. After copying the packet content into this object, it can be inserted into the output packet buffer and the egress module is notified.

The information if the captured packet is the last one of the current step is required to determine when a time advance request can be sent. If the simulation bridges only supported time-triggered traffic, all inputs and outputs of the device were known and a time advance request could be performed once they are captured. However, technologies such as TTEthernet also support best-effort traffic and a simulation step might produce multiple outputs sent at different times. The simulation bridge must forward all packets and request a time advance afterwards. Without knowledge about the number of outputs, the simulation bridge can advance only to the time of the latest packet captured. Otherwise, packets may be missed or the TSO might be violated. In case of large network delays or a rare number of output packets, the synchronization delays caused by waiting for the next packet might introduce severe overheads.

Once the egress module is notified, it queries the available packet from the buffer and triggers the estimation of next inputs if the state-estimation mechanism is enabled. A detailed description of the mechanism can be found in Section 5.1.2 for guaranteeing a timely packet arrival and in Section 5.1.3 for an increased accuracy. If the current event is denoted as an output event, the egress module calls the RTIAmbassador to forward the packet. The ambassador retrieves the interaction class, its RTI handle and the handles of its parameters from the interaction instance object and encapsulates the parameters into a parameter handle value map. This map is sent to the RTI together with the interaction handle and the send time if the latter is valid. Afterwards, a time-stamp is stored in the interaction instance object representing the time the packet was sent. Once all packets are forwarded, a *NextMessageRequest* is sent to the RTI. Depending on the simulation bridge configuration, there are different constraints when this message is sent. These constraints are explained in the related sections for the synchronization mechanisms (cf. Sections 4.3.2 and 4.3.3) and the usage of the state-estimation mechanism (cf. Sections 5.1.2 and 5.1.3).

At the incoming site, the RTIAmbassador decapsulates the parameters from the received interaction and saves a time-stamp of the reception. After creating and filling an interaction instance object, it inserts the latter into the input packet buffer and notifies the ingress module. Once the packet is forwarded, the wrapper module stores the current physical time as the ingress forward time-stamp. Together with the ingress time-stamp representing the time of the reception, the pass-through time of the packet in the simulation bridge can be calculated.

4.5. Fault-Injection based on IEC 61508-2

As mentioned in Section 4.1.2, the distributed co-simulation framework supports fault-injection to validate the dependability of the SUT. Following the IEC 61508-2 standard, the injection is performed on incoming packets from other devices at runtime.

After selecting the packet to forward, the ingress module triggers the fault-injection module to inject an omission, corruption, delay or replay fault according to Algorithm 4. The fault-injection module obtains the first entry of each fault-schedule and compares it with the current logical time. In case the time matches for one or more faults, a related boolean flag is set (Lines 2-7). This flag determines the injection of the related fault and is checked in Line 8. If one flag is set, the fault-injection module obtains the message object of the currently received interaction from the application schedule (Line 9). As mentioned in Section 4.2, this object contains the parameters for the injection. An error in the configuration leads to a missing object and the function returns true. As a consequence, the packet is forwarded without any injection. Otherwise, the module checks for packet omission and if the current time is available in the related fault list in the message object. In case both checks are successful, the packet is omitted by setting a related flag in the interaction instance object and returning false (Lines 10 to 13).

The remaining corruption, replay and delay errors can be configured to occur alone or in combination. In every case, the fault information is obtained from the message object (Lines 16, 20 and 27) and the event is removed from the fault-schedule after injecting the fault (Lines 17, 24 and 30). In between, the injection is performed as follows. To corrupt a packet (Line 17), the interaction instance object retrieves the parameters, makes a copy of the packet and changes the faulty bytes to their new value. In case of replay, the replay instants are added to the schedule and new packets are inserted into the faulty packet buffer. By adding the events to the application schedule in the correct order, they are automatically considered during the time advance process. Since the buffers store references to the interaction instance objects, the reference to the original instance is copied. Furthermore, a flag is set in the related object which signals the performed injection of the fault (Lines 21-24). Similarly, a copy of a delayed packet is added to the faulty packet buffer for its new time of reception. To determine the instant, the fault-injection module obtains the delay and adds it to the current time. Furthermore, this instant is inserted into the application schedule and the delay interval is set in the related member variable of the interaction instance object (Lines 27 to 30).

Algorithm 4 Fault-injection for omission, corruption, delay and replay errors

```
 1: procedure InjectFaults(int64 currentTime, EthernetInteractionInstance eii)
 2:     for all fault-event schedules do
 3:         get first event from fault-schedule and compare with current time;
 4:         if time matches then
 5:             set event-flag of related event to true;
 6:         end if
 7:     end for
 8:     if corruption, delay, omission or replay event-flag set then
 9:         get message object for current time from schedule;
10:         if message found and omission event then
11:             if logical event-time found in omission fault list of message object then
12:                 set packetOmitted in eii and return false;
13:             end if
14:         else if message found and no omission event then
15:             if corruption event then
16:                 get corruption fault from fault list in message object;
17:                 corrupt payload in eii and remove event from fault-event list;
18:             end if
19:             if replay event then
20:                 get replay fault from fault list in message object;
21:                 for all replay instants do
22:                     add interaction to faulty packet buffer and event to schedule;
23:                 end for
24:                 set replayed member in eii and remove event from fault-event list;
25:             end if
26:             if delay event then
27:                 get delay fault from fault list in message object;
28:                 set delay member in eii and add new event to schedule;
29:                 insert packet into faulty packet buffer for current time plus delay;
30:                 remove event from fault-event list and return false;
31:             end if
32:         end if
33:     end if
34:     return true;
35: end procedure
```

Using the return value, the fault-injection module indicates whether the packet has to be forwarded to the wrapper module or not. The first case is covered by returning true while false signals the omission or delay of the packet. Manipulation, resequencing and insertion are not mentioned explicitly. As explained in Section 4.2, manipulation can be mapped to corruption while delaying one or more packets results in resequencing. Apart from that, the insertion of additional packets is performed during the configuration process.

4.6. Software- and Hardware-In-The-Loop Testing

In the previous sections, the handling of packets, the synchronization of simulation bridges and the injection of faults are discussed. This section illustrates their combination to support Software- and Hardware-In-The-Loop (SIL, HIL) testing. SIL represents the default operation mode of the simulation bridges and is used for co-simulation or the connection of software-implemented control algorithms. HIL is used to connect real hardware instead.

To synchronize the simulation bridges for SIL with non-real-time devices, they use the normal execution mode according to Section 4.3.2 automatically. In this mode, a time advance to time t signals the connected device to execute until the step starting at t has finished. Afterwards, its outputs are exchanged. If there are no packets available, the device returns signaling that outputs are pending and the simulation bridge requests a time advance to the next event.

Real-time execution via the distributed co-simulation framework uses the synchronization mechanism described in Section 4.3.3. This mechanism expects a timely packet arrival which must be ensured in two ways. Either the communication cycles in the device's application schedule are large enough so that all data is received in time or a delay-management mechanism according to Chapter 5 must be applied. At runtime, the bridges capture and forward packets according to Section 4.4. If real-time devices shall be synchronized using the HLA time-management, a mechanism is required between the bridge and the device which triggers the end of an execution step. A real device can be connected to the simulation host via a communication mechanism such as TTEthernet or TSN. In this case, temporal synchronization is used and the simulation bridge can determine the end of a step by exploiting information provided by the application schedule. Herein, all output messages of a step are stored including their scheduled send-time. Alternatively, explicit synchronization by exchanging control packets can be used.

Fault-injection is supported automatically for each operational mode as soon as there are faults configured. In this case, the injection of insertion faults is performed during the configuration of the simulation bridge (cf. Section 4.2) while the other faults are injected at runtime as described in Section 4.5.

4.7. Secure Communication

The main purpose of using the distributed co-simulation framework is the validation and verification of distributed embedded systems via the Internet. Herein, the protection of confidential data and intellectual property are central aspects. The Internet is an open network which does not provide any security services. Hence, it is required to apply mechanisms which ensure the confidentiality, integrity and authenticity of the transmitted data.

This thesis focuses mainly on delay-management technologies to support real-time communication via the Internet. Security is not a central topic, but one important aspect in future validation and verification processes. Hence, this section suggests the usage of a Virtual Private Network (VPN) to provide the required security mechanisms. As explained in Section 2.6, VPNs provide confidentiality, integrity and authenticity of data in public communication networks. Using them, it is not required to implement any security mechanisms in the simulation bridges because the communication between the hosts is secured already. However, the encryption of data introduces additional delays, wherefore it is the user's responsibility to decide if those mechanisms are required or not.

At the University of Siegen, the open source solution OpenVPN is used to connect hosts from other networks with the network of the university. It is a widely used technology for secure and scalable communication with downloads by millions of people and companies [INC]. Hence, it is also used in this thesis if communication via the Internet is required.

5. Delay-Management in Distributed Real-Time Tests via the Internet

5.1. Delay-Management using State-Estimation

The state-estimation mechanism estimates future inputs for the connected device based on a model of the remaining system under test, previous device outputs and its communication schedule. Furthermore, received inputs are compared with the estimated ones to ensure a valid accuracy of the estimation. There are two possibilities to provide the inputs to the device. In the first, estimation is used as a fall-back solution if packets are delayed. In the second possibility, a subset of the events in the application and communication schedule is used to exchange data with the other bridges. In between, intermediary estimated packets are forwarded. Both possibilities expect a properly configured real-time simulation host.

5.1.1. State-Estimation Models

As introduced in the overall architecture of the simulation bridges, there are two models connected to the state-estimation module. While the system model is used to estimate future inputs, the comparison between received and estimated ones is performed in the device model. Since the models used depend on the SUT, they must be provided by the system developer. Using a generic interface such as FMI to connect them provides a variety of supported simulation tools as depicted in Section 3.2.2. In the following, the interaction between both models and the state-estimation module is presented.

Once a packet has been received and there is enough time left before the next real-time event occurs, the ingress module triggers the comparison with the estimated data. In this way, the simulation bridges prevent delays in real-time activities caused by the state-estimation module. However, the check might not be performed directly. Performing the check also if the received packet is forwarded ensures a valid accuracy of future estimation results. To compare the inputs, the state-estimation module searches the interactions of the reception time in the two buffers and forwards them to the device model. This model decapsulates the inputs and compares their difference with a configurable error threshold during the execution of its step. As the step size, the interval to the previous check is used. If the result lies within the configured bounds, the model returns with status *ok* while status *error* signals a difference which is too large. In this case, the simulation bridge notifies the user and terminates.

Before captured packets are sent to the other simulation bridges, the egress module triggers the state-estimation module to estimate the future device inputs. In this way, the simulation bridge ensures an implicit synchronization between the egress and the ingress module since both access the RTIAmbassador. As a consequence, potential delays caused by sending the packet do not influence the availability of the estimated packet negatively. Furthermore, estimated packets are available before the received ones which simplifies the comparison algorithm in the device model.

The state-estimation module communicates with the system model similar to the wrapper module (cf. Section 4.4). As simulation step size, the interval between the last and the next packet reception time is chosen. After obtaining and analyzing the estimated packets with regard to the interaction class and the reception time, the latter is used by the state-estimation module to insert the packet into the estimated packet buffer. Since it is further possible that inputs have to be estimated for the first simulation step, an initial estimation is required. This estimation is triggered during the initialization process of the system model. At run-time, the ingress module forwards either estimated or received inputs depending on the estimation mechanism used.

5.1.2. State-Estimation as Fall-Back Solution for Timely Packet Reception

The application of state-estimation as a fall-back solution to guarantee a timely packet reception is the first estimation mechanism introduced. It extends the real-time synchronization described in Section 4.3.3 and is used mainly in simulation bridges connected to real-time devices. However, applying it for non-real-time components is also possible to reduce the number of estimations in the real-time devices.

The extension for this state-estimation mechanism (cf. Algorithm 5) is applied on Algorithm 1 introduced in Section 4.3.3. While the real-time synchronization mechanism checks only the input and faulty packet buffers for available packets (Lines 11-14), the extension includes also the estimated packet buffer (Lines 15-16). Since the state-estimation module provides an input for every reception event, this mechanism can be used as a fall-back solution if packets are delayed due to the network latencies. In this case, an estimated packet is forwarded instead of the received one. Once a packet was received, the RTIAmbassador signals its availability by adding a *RECEIVE_EVENT* in the ingress module. This event type is checked after all available real-time events are processed to prevent delays. If the check was successful, the state-estimation module performs the comparison between received and estimated inputs (Lines 24-25). The remainder of the real-time synchronization algorithm remains similar to Algorithms 2 and 3 which realize the time advance.

Algorithm 5 State-estimation in ingress module for real-time execution

```
 1: procedure RunRT
 2:     set bools terminated, estimationFinished to false;
 3:     declare packetPtr;
 4:     while not terminated do
 5:         if event in application schedule then
 6:             wait until time of next application schedule event;
 7:         else
 8:             wait for non-RT-event;
 9:         end if
10:         if RT-event occured then
11:             if packet found in inputPacketBuffer then
12:                 set packetPtr to input packet;
13:             else if packet found in faultyPacketBuffer then
14:                 set packetPtr to faulty packet;
15:             else if packet found in estimatedPacketBuffer then
16:                 set packetPtr to estimated packet;
17:             end if
18:             if packet found then
19:                 inject fault into packet if configured;
20:                 if not (omission or delay failure) then
21:                     forward packet of packetPtr;
22:                 end if
23:             end if
24:         else if RECEIVE_EVENT then
25:             check estimated input;
26:         end if
27:     end while
28: end procedure
```

Besides using state-estimation to ensure a timely packet reception in real-time tasks, it is also possible to use it for non-real-time devices. Figure 5.1 shows a use-case with two non- and one real-time device where large network delays cause a deadline miss in the latter. While the boxes in the figure denote tasks, sent messages are represented as upwards pointing arrows and received messages as descending ones. Message M_0 is sent by device $ED1$ after $20ms$ and received by $ED2$ after $40ms$, wherefore the worst case communication delay ($WCCOM$) accounts for $20ms$. For M_1, the instants are $50ms$ (sending) and $75ms$ (receiving), resulting in a delay of $WCCOM = 25ms$. However, the network connecting the simulation bridges introduces a communication latency of $t_{COMM} = 45ms$ which delays the task in $ED2$ by $25ms$. Even if the (simulated) execution of the tasks finishes after $5ms$ and the communication delay is shorter than the scheduled WCCOM, the deadline of the real-time task at $75ms$ is missed.

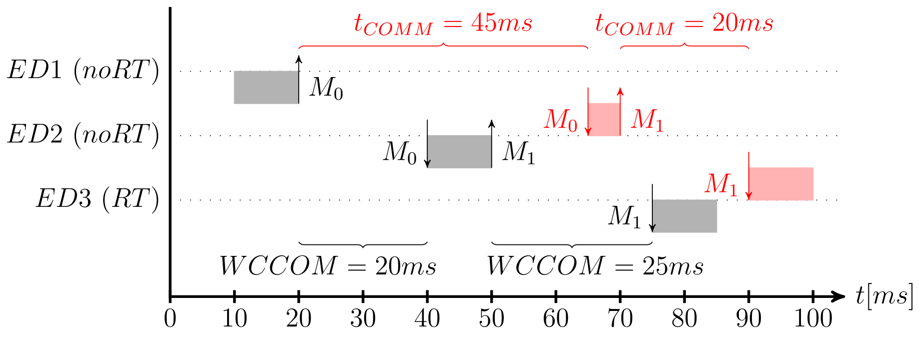


Figure 5.1.: Communication delays causing a retarded execution of a real-time task.

Using state-estimation also in non-real-time tasks solves this issue. In this mode of operation, activities in a non-real-time device are triggered in real-time. Additionally, an offset can be defined in the configuration file which allows to prepone a task's execution (*rtOffset*). In this way, it is possible to mitigate larger communication delays than scheduled by the WCCOM. With regard to Figure 5.1, the execution of the task on $ED2$ could be triggered at $40ms$ using an estimated M_0. Since the real communication delay ($t_{COMM} = 20ms$) is smaller than the WCCOM, the message would be received in time. If the latency would account for a larger value (e.g., $t_{COMM} = 30ms$), an offset of more than $5ms$ could ensure a timely packet reception in the real-time device. To define this offset, different parameters must be considered. Those are the:

1. Network delay between simulation bridge and device
2. Execution times of the simulation bridges (incoming and outgoing) and the RTI
3. Latency of transmitting the interaction including the data via the network
4. Time synchronization overhead introduced by the HLA time management

The first parameter in this list depends on the device and its connection to the simulation bridge's wrapper interface. In case of a real device, the delay depends on the network link and the library which captures the packet (e.g., PCAP). If the device is simulated and loaded as a shared library (e.g., using FMI), there is a small latency as the simulation bridge performs only a function call. This latency and the execution times of the bridges and the RTI depend on the simulation hosts on which they are running. All parameters and the network delay required to transfer an interaction have to be determined by the user of the framework. Compared to the communication latency of a single interaction, the synchronization overhead is much larger. The number of messages exchanged during the time synchronization algorithm depends on the number of federates in the federation. The message number has to be multiplied with the network delay to determine the communication costs of the synchronization. Accumulating all four parameters and subtracting the result from the WCCOM provides the offset that has to be configured. However, ascertaining them can be difficult especially if there are no QoS services available.

5.1.3. Increased Accuracy using Estimated Intermediary Packets

The previously described mechanism is useful if the communication slots of the SUT's schedule are in the same order of magnitude like the delays of the simulation bridges' network. However, there might be SUTs with a dense schedule for which the network latencies are too large. As a consequence, there would be many estimation activities without forwarding any received packet. This section introduces a mechanism where the number of synchronization messages exchanged between the simulation bridges is reduced. In between, intermediary, estimated packets are forwarded to the device. Adapting the communication activities to the network delays allows a timely packet reception in the real devices while maintaining a sufficient accuracy.

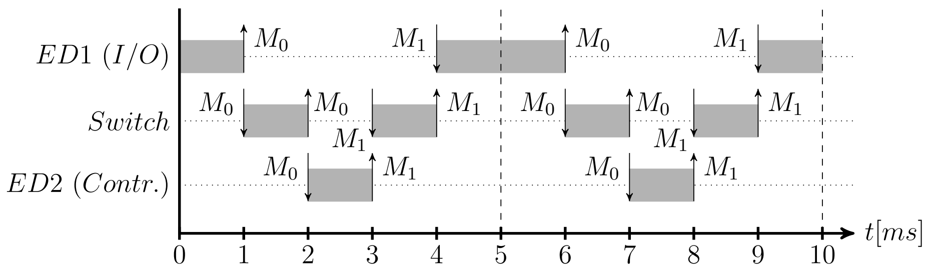


Figure 5.2.: Dense application schedule of two end devices connected via a switch.

Figure 5.2 shows an example with a dense schedule as mentioned above. An end device hosting a sensor and an actuator ($ED1$) is connected to a controller ($ED2$) via a switch. The application schedule is time-triggered with a period of $5ms$. At offset $1ms$ in each period, the sensor collects the state of the environment and sends it to the switch in message M_0. After simulating the network latency of $1ms$, the switch forwards the packet to the controller (offset $2ms$) which replies with a control value in M_1 after $3ms$. Similar to M_0, M_1 passes the switch and reaches $ED1$ after $4ms$ which performs the actuation.

To reduce the number of interactions exchanged, a bridge communication schedule is introduced which represents a subset of the SUT's application and communication schedule. Within a selectable bridge communication period, this schedule determines the instants at which the simulation bridges communicate. The idea is illustrated in Figure 5.3 where four types of messages are shown. At first, black arrows pointing upwards represent interactions which contain packets captured from the connected device. After using them for the estimation, they are forwarded to the receiving bridges. The related received messages are printed as red arrows pointing down. In contrast, the remaining two types are not exchanged between the bridges. Captured packets that are used only to estimate future inputs are printed in blue while the estimated inputs are represented by cyan-colored arrows. Which messages are exchanged can be defined by the user of the framework and the related events are stored in the bridge communication schedule. In this way, the mechanism is adaptable to the SUT and the network delays ensuring a timely packet reception by the real hardware.

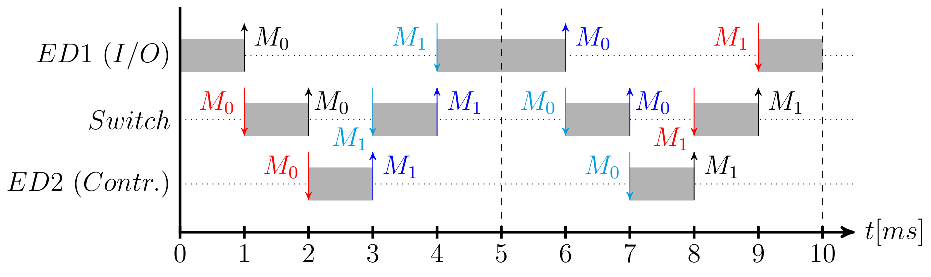


Figure 5.3.: Definition of events in bridge communication schedule.

An example for using the state-estimation algorithm based on a suitable bridge communication schedule is depicted in Figure 5.4. An interval of $10ms$ is defined as bridge communication period which is twice the period of the application schedule. In this case, $ED1$ is a device running in real-time. The switch as a simulation and the controller as a software application only have an implicit relation to the real-time based on the schedule. At the beginning of the period, the I/O-device sends the sensor result to the switch at $1ms$. Assuming a network delay (*delay-net*) of $1.5ms$, the switch receives the message after $2.5ms$. Since it is simulated, the switch processes the message fast and forwards it to the controller after $2.75ms$. Due to the same network delay, this device receives the input after $4.25ms$. In the meantime, the simulation bridge connected to the I/O-device forwarded an estimated input at instant $4ms$ and the switch was able to simulate until event 7. Since the next event in its application schedule is also a reception event of the bridge communication schedule, the switch has to wait for the controller. Similarly, the controller can process its remaining events of the bridge communication period and it sends the final control output to the switch after $5.5ms$. While the message is transferred, the simulation bridge of the I/O-device uses its output at $6ms$ to estimate the next input. However, the switch is able to forward its output in time at $8.75ms$ due to the properly selected schedule and the I/O-bridge can forward the received packet after $9ms$.

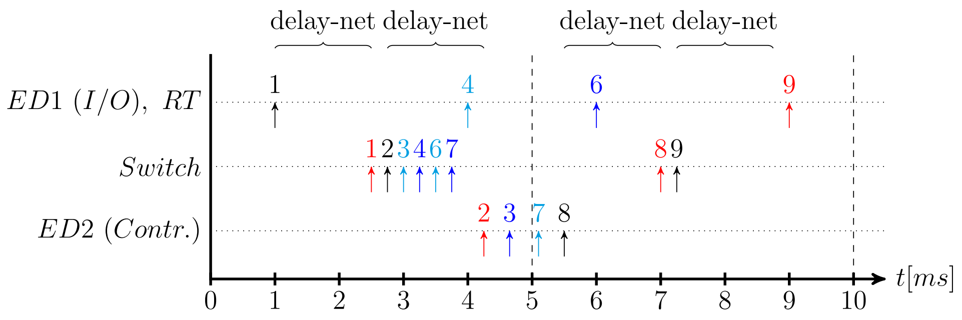


Figure 5.4.: Application of state-estimation providing intermediary packets.

To implement a real-time simulation bridge for this mode of operation, the functions defined in Section 4.3.3 can be reused with some adaptations. These refer to Line 10 of Algorithm 2 (*AdvanceInTimeRTegress*) and Line 9 in Algorithm 3 (*AdvanceInTimeRT-ingress*). Instead of obtaining the instant of the event for the next time advance from the application schedule, both functions retrieve it from the bridge communication schedule. Similarly, the egress module checks whether the current send time is available in the bridge communication schedule before it forwards an interaction. This does not account only for real-time execution, but also for the estimation of intermediary packets.

Algorithm 6 Advance in time for SIL in the RTIAmbassador

```
 1: procedure AdvanceInTime(int64 timeEvent)
 2:     if not timeAdvanceGranted then
 3:         wait for TimeAdvanceGrant;
 4:     end if
 5:     set timeAdvanceGranted to false and get next application event;
 6:     if event found in bridge communication schedule then
 7:         call next message request;
 8:     else if event found in application schedule then
 9:         set timeAdvanceGranted to true;
10:         add ESTIMATION_EVENT in ingress module
11:     else if all events of application schedule processed then
12:         terminate;
13:     end if
14: end procedure
```

During real-time execution, the Algorithms 2 and 3 have to consider the progress of real-time in parallel to the time advance procedure. Instead, the RTIAmbassador has to handle only the different schedules during non-real-time execution. The related procedure is depicted in Algorithm 6. To prevent multiple time advance requests, a flag *timeAdvanceGranted* is used. While a request is processed, the flag is set to false (Line 5) and another thread calling the procedure has to wait (Lines 2-4). Afterwards, the ambassador obtains the next application event and checks if it is available in the bridge communication schedule. In this case, a *NextMessageRequest* can be sent to the RTI to advance in time (Lines 6-7). If the event is only present in the application schedule, the RTIAmbassador can set the flag *timeAdvanceGranted* to true and trigger an estimation event in the ingress module (Lines 8-10). In case of a *NextMessageRequest*, the flag is set during the *TimeAdvanceGrant*. Finally, it is also possible that all events are processed already. This case signals the end of the simulation and the bridge terminates (Line 12).

Algorithm 7 State-estimation for intermediary packets in SIL

```
 1: procedure RunEstimation
 2:     set bools terminated, estimationFinished to false;
 3:     declare packetPtr;
 4:     while not terminated do
 5:         wait for event;
 6:         if ESTIMATION_EVENT then
 7:             if faulty packet available for current step then
 8:                 forward packet;
 9:             else
10:                 get estimated packet and perform fault-injection;
11:                 if not (omission or delay failure) then
12:                     forward packet;
13:                 end if
14:             end if
15:             perform simulation step;
16:         else if RECEIVE_EVENT then
17:             get received packet and perform fault-injection;
18:             if not (omission or delay failure) then
19:                 forward packet;
20:             end if
21:         else if TAG_EVENT then
22:             if faulty packet available for current step then
23:                 forward packet;
24:             end if
25:             perform simulation step;
26:         end if
27:     end while
28: end procedure
```

Similar to the normal and the real-time execution, a dedicated function is defined for the ingress module to run the estimation of intermediary packets. The related algorithm is presented in Algorithm 7. At the beginning of the main loop, the module waits for one of the following events. An *ESTIMATION_EVENT* is triggered by the function *AdvanceInTime*. In this case, the ingress module searches for faulty or estimated packets to forward (Lines 7-14). As shown in Line 10, faults can also be injected into estimated packets before forwarding them. A *RECEIVE_EVENT* denotes the reception of a packet from another device instead. Similar to an estimated packet, it is forwarded once the injection of faults has finished (Lines 16-20). Finally, a *TAG_EVENT* is added in case of a *TimeAdvanceGrant*. After forwarding available faulty packets, the ingress module triggers a simulation step in the device (Lines 21-26). Once the step has finished, the egress module handles the packets as described in Section 4.4. The only difference is an additional check if the current send-time is available in the bridge communication schedule. In this case, the transmission is scheduled and the egress module can forward the interaction.

5.2. Delay-Management using Speculative Execution

Speculative execution was developed for processors to increase the utilization of resources by speculatively executing commands in advance. This work provides a mechanism which is able to distribute a federation into several independent subsets. In this way, independent tasks can be processed in advance to provide real-time data in time. Furthermore, the synchronization effort can be reduced which speeds up the overall simulation execution. The reason is a reduced number of messages that are exchanged between the simulation bridges and the RTI. In this section, the overall concept and the extensions of the simulation bridges to route data between different federations are described.

5.2.1. Concept of Speculative Execution

Complex distributed embedded real-time systems may comprise different types of components or functionalities with varying temporal dynamics. One example can be a brake together with a Human Machine Interface (HMI) system to control it. While the brakes have to respond fast on inputs such as an emergency stop, user inputs via the HMI system are slow. Hence, the latter can be executed with larger cycle times compared to the brake control. If these components are integrated via the same network, all communication activities need to be validated during the development process. In this case, a component with small cycle times performs much more activities than another component having long cycles. As a consequence, the latter might be blocked gratuitously waiting for the permission to execute.

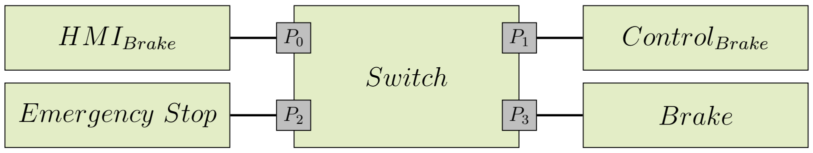


Figure 5.5.: Distributed SUT with different cycle times.

Figure 5.5 shows an example SUT with a brake, the related controller, an emergency stop component and an HMI. The components are connected via a switch and the related schedule is illustrated in Figure 5.6. In this example, the brake, its controller and the emergency stop component operate with a cycle time of $20ms$ while the HMI has a cycle time of $100ms$ due to human response times. At the beginning of each period, the brake reacts on the previously received control input and the emergency stop signal and sends the current speed to the brake controller afterwards. Based on the received input and the current destination speed, the controller determines the braking force and sends it to the brake. The emergency stop component reacts on related sensor inputs and sends a stop signal to the brake if required. During normal operation, a keep-alive signal is sent instead. Since the emergency stop component does not receive any inputs from the brake and its controller, it can be considered to be independent. Finally, the HMI component receives the current speed, it outputs the value and the operator is able to brake. A related message is sent to the brake controller around $80ms$ at the end of the HMI's period. The controller is responsible to adapt the braking force afterwards.

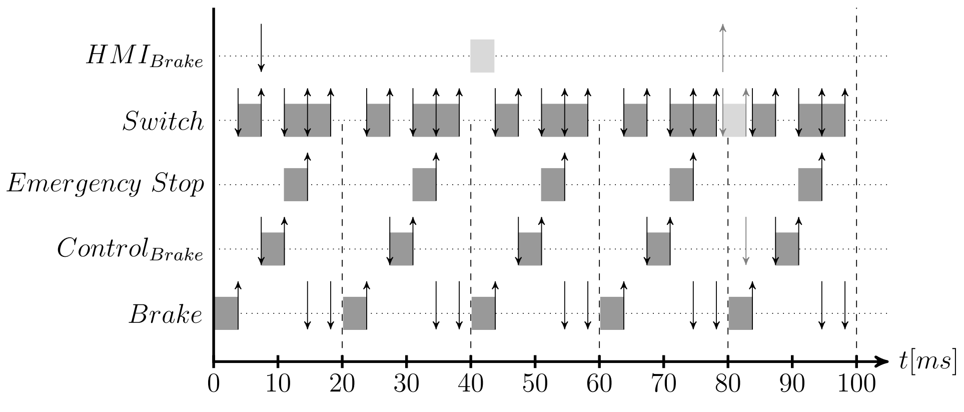


Figure 5.6.: Schedule of example SUT.

In the example above, there are two cases for which a speculative execution is useful. First, the emergency stop controller has its own sensors and does not depend on inputs of the brake or its controller. Second, the HMI component has a larger cycle time. In this cycle, the HMI requires the brake's output from the first period and the HMI's output is used by the brake controller during its last period. In between, there are no interactions wherefore the HLA's time management would block the HMI's simulation. The concept of speculative execution prevents this drawback by splitting the federation into independent subsets. These subsets can be executed in parallel and synchronization between them is required only if there are messages exchanged beyond the subset boundaries. In this way, independent non-real-time components can be executed earlier than scheduled since they are not blocked waiting for non-related *TimeAdvanceGrant* messages. As a consequence, there is more slack available to transfer real-time data between the simulation bridges in time.

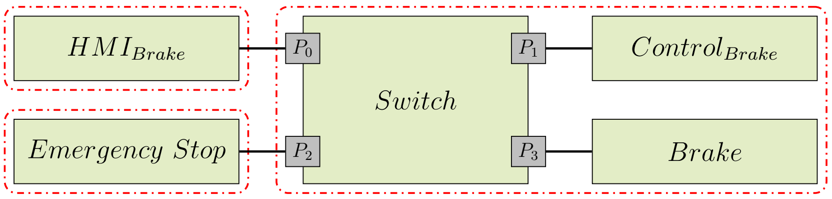


Figure 5.7.: Different subsets of SUT.

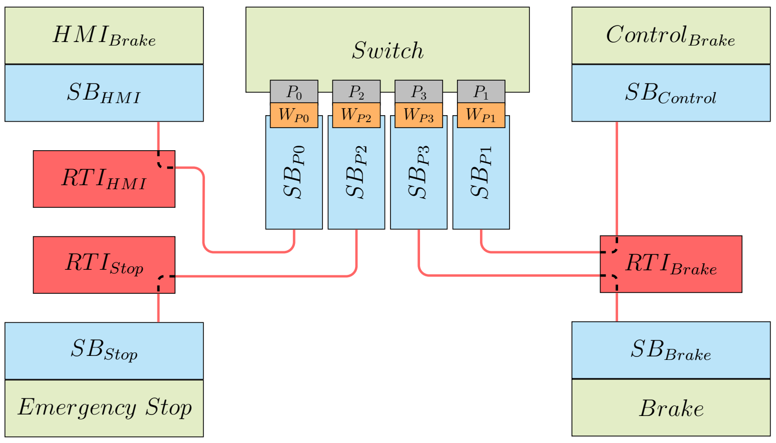


Figure 5.8.: Connecting the subsystems of the SUT using simulation bridges.

Figure 5.7 depicts the different subsets for this example. The brake, its controller and the switch represent the first subset which might be available as real hardware in a HIL setup. The second subset is the emergency stop component while the HMI represents the last one. To exchange data between the different subsets, the related links are connected via simulation bridges in different federations. As shown in Figure 5.8, there are three RTI components: one for the link between the switch and the HMI, one for the emergency stop component and one RTI which connects the brake and its controller with the switch.

Considering the HMI and the emergency stop component to be non-real-time components, their early execution is illustrated by Figure 5.9. Since the emergency stop task uses its own inputs, the execution of its task instances can be performed directly at the beginning of the simulation. Assuming that the third instance of the task causes an emergency brake, a slack of about 43ms is created between sending the message and the scheduled send time. Similarly, the HMI task can be executed once the input is available which results in a slack of about 68ms.

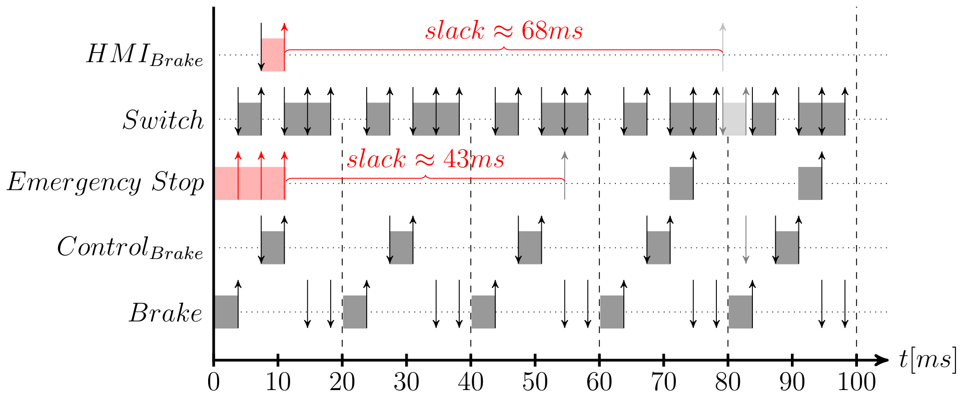


Figure 5.9.: Adapted schedule of example SUT with early execution of independent tasks.

As a main advantage of this mechanism over state-estimation, an estimation model is no longer required. However, there can be limitations in the accuracy of the tests since all synchronization and communication messages have to be exchanged via the network connecting the simulation bridges. The related delays must be considered while defining the subsets.

5.2.2. Extending Simulation Bridges to Simulation Gateways

Using the HLA time management, all simulation bridges connected to the same HLA federation can ensure a timely correct order of all events in the global communication and task schedule. As a consequence, a component with multiple simulation bridges connected must not be aware of synchronizing its inputs. One example for such a component is the switch in the braking example of Section 5.2.1. Once the simulation bridges are connected to different, independent federations, a global event synchronization is not guaranteed anymore. Hence, a new type of simulation bridges has to be introduced which realizes the synchronization between multiple federations. This type is called a simulation gateway and described in this section.

Figure 5.10 shows the architecture of the simulation gateways. To connect a component to multiple federations, the following two changes are needed compared to the simulation bridges described in Section 4.1.2. According to the network-centric character of the simulation framework, a pair of simulation bridges replaces one network link of the SUT. Hence, they are only capable of connecting one device and a component with multiple network interfaces requires multiple bridges. Using the speculative execution, the federations can be divided only at components with multiple network interfaces such as switches or routers. To ensure a correct synchronization between all its inputs, such a component needs to be connected to a single simulation gateway. As a consequence, the gateway must be extended to provide multiple wrapper interfaces. Those interfaces are managed by the wrapper module. Similarly, connecting a gateway to multiple federations requires a correlating number of RTIAmbassadors. They are aggregated by the RTIAmbassador Module. To minimize the number of changes, the wrapper interface and the RTIAmbassador are retained as designed for the simulation bridges. The changes regarding synchronization and data exchange are performed in the RTIAmbassador, the wrapper module, the ingress module and the schedule instead.

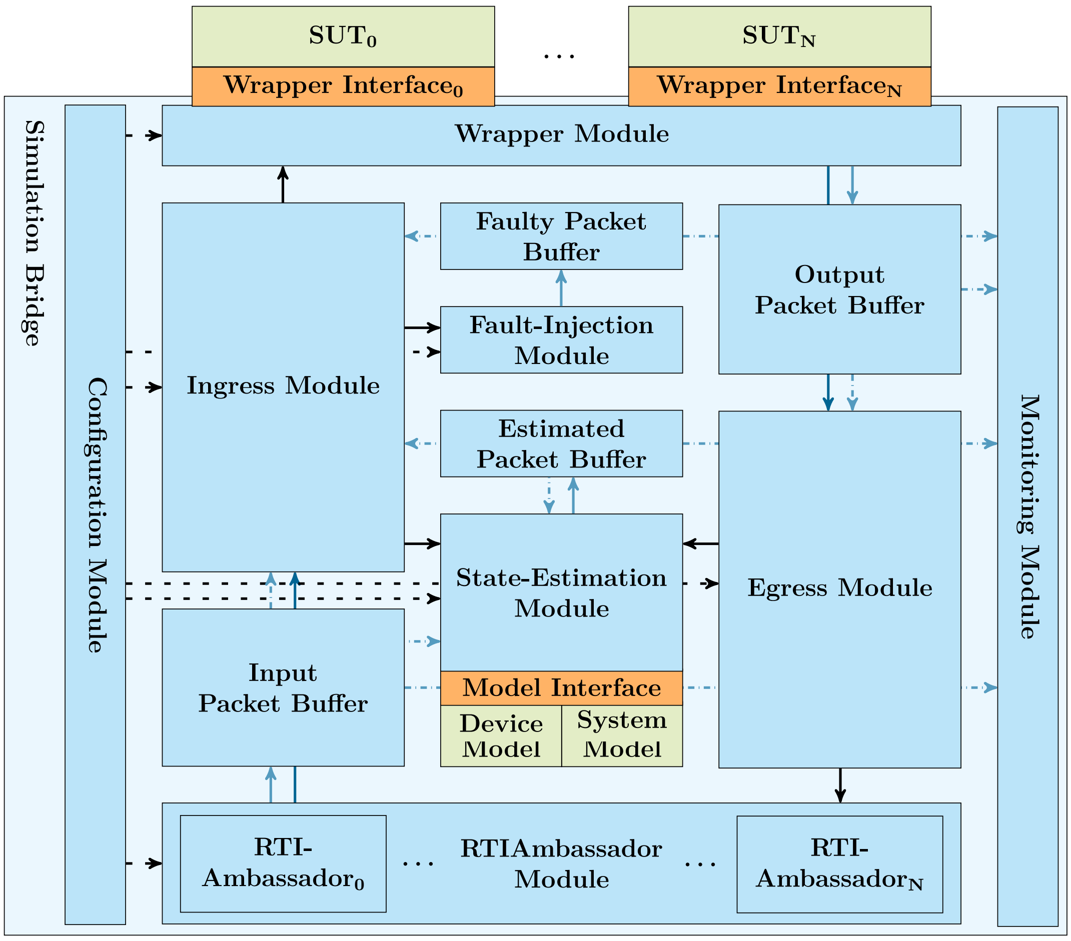


Figure 5.10.: Architecture of simulation gateways.

Since the synchronization mechanism is based on an event calendar, the schedule is extended by a local schedule for each ambassador. This local schedule contains all task and communication events related to the connected federation. Events affecting multiple federations are added to all associated local schedules. For example, ambassador A receives a packet at time t in its federation. If the packet has to be forwarded via ambassador B, a task event is added for time t in B's local schedule. Merging the local schedules to a global event list enables an overall synchronization of the events. In addition to the logical time, each event is associated with a list of required input packets and references to the ambassadors receiving a *TimeAdvanceGrant*. Those references are used by the RTIAmbassador module to send the next time advance requests once the step has finished. The ingress module selects the next event from the schedule and checks if all input requirements are fulfilled. Afterwards, the ingress module forwards the packets and triggers a simulation step as described before.

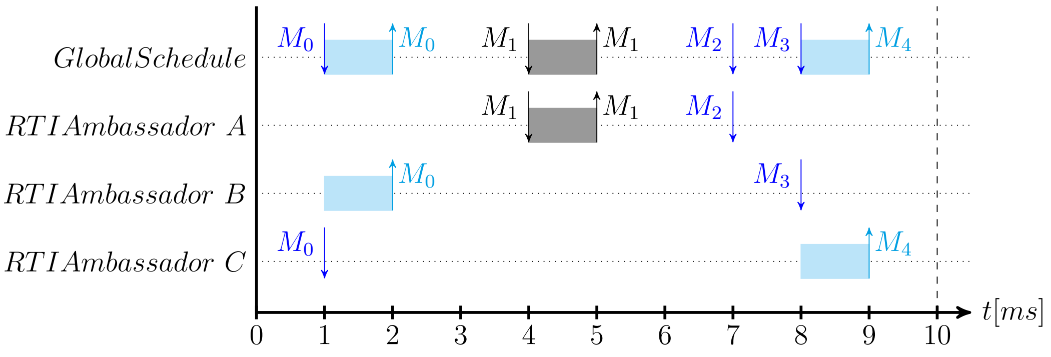


Figure 5.11.: Local and global event schedules.

The synchronization mechanism including the local and global event schedules is illustrated in Figure 5.11. It depicts four schedules, one for each federation (RTIAmbassadors A to C) and the global schedule. At instant 1, RTIAmbassador C receives message M_0 which has to be forwarded via RTIAmbassador B at instant 2. Hence, there is one reception event available in RTIAmbassador C and one task event in B at this time. To synchronize the federations, the ingress module blocks the gateway until both, M_0 and the *TimeAdvanceGrant* for the task are received. The next event in the global schedule is the reception of message M_1 at instant 4. Since the message is forwarded via RTIAmbassador A which also receives the message, there is no additional event needed. Finally, the event at instant 8 represents a task with multiple inputs (M_4, M_5 and M_6) affecting all three federations. Again, the task can only be executed once both messages and the related *TimeAdvanceGrants* are received. The ingress module blocks the gateway accordingly.

Forwarding the packets to the correct SUT requires information about the related wrapper interfaces. This knowledge is provided by adding a list of interface references to the packets' interaction classes. The wrapper module checks the list and forwards the packet to all receivers. The same accounts for the RTIAmbassadors to which the packet is sent on the egress site. This list is checked by the RTIAmbassador module instead. After forwarding the outputs, the RTIAmbassador module requests a time advance in all RTIAmbassadors associated with the current event. Once all local events are processed, the RTIAmbassador terminates and notifies the RTIAmbassador module. This module maintains the status of all RTIAmbassadors and terminates the gateway after processing all events in the global schedule.

6. Evaluation using a Fault-Tolerant Fan-Control Application

6.1. System Under Test for Framework Evaluation

To evaluate the framework, a distributed fan-control application is used as an SUT. It is based on a fan component, a PID controller, a voter and a network simulation. These components are connected in different setups to demonstrate the framework's scalability and the applicability of the developed fault-injection and delay-management mechanisms. This section introduces the different setups and the implementation of the components as FMUs and on real hardware.

6.1.1. Fan-Control Application

The components of the distributed fan-control application are a fan, a PID controller, a voter and a switch which links the components. They are combined in the following five setups starting with the fan and one PID controller. The second setup follows the concept of triple modular redundancy where the PID controller is triplicated and a voter is introduced. In the third setup, the voter is triplicated further while the fourth and fifth setup duplicate and triplicate Setup 3. The components are connected via Ethernet and communicate according to a periodic, time-triggered schedule. By increasing the number of components, the setups shall demonstrate the scalability of the framework.

Setup 1: Fan-Control Without Redundancy

The first setup without any redundancy is illustrated in Figure 6.1. In this case, there is only one PID controller which sends its control outputs to the fan directly. The fan closes the loop by sending the current speed with a period of $10ms$. In addition, the fan sends a new speed set point to the PID controller every $10s$.

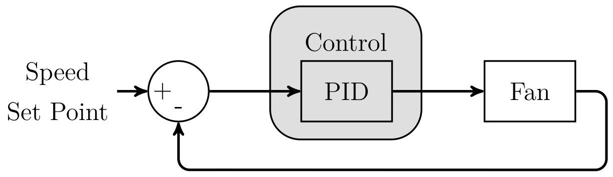


Figure 6.1.: Fan control application without redundancy.

Figure 6.2 introduces the time-triggered schedule for the first setup. While the x-axis denotes the period's time in ms, the components are shown on the y-axis. The dashed line at time $10ms$ represents the end of the period which repeats afterwards. In every period, an offset is assigned to a task's execution or the transmission of a message. At offset $1ms$, the fan starts with sampling the current speed and sending it to the PID controller in the message *SensorRPM*. In every setup, the worst-case communication delay of the network links is set to $1ms$. It is simulated by the network simulation which receives the message at $2ms$ and forwards it to the PID controller at $3ms$. After receiving the value, the PID controller uses the difference between the current and the reference speed as input for the PID algorithm ($3ms$). The result of this computation is sent back to the fan at offset $4ms$ and received by the fan at $5ms$. The new reference speed (*DestinationRPM*) is sent at the end of the $10s$ period (offset $0ms$). Since the period is different from the control loop, the messages and the simulation task are colored in light gray.

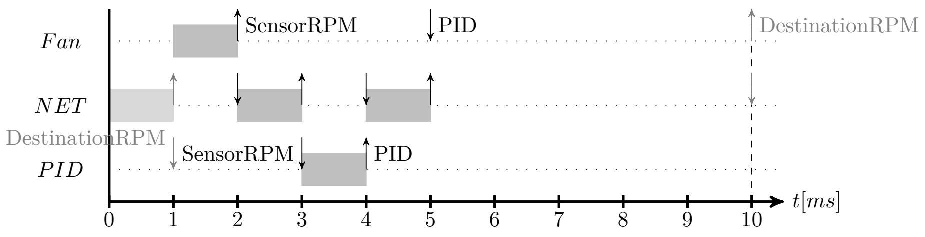


Figure 6.2.: Schedules in DUT without redundancy for one period of 10ms.

Setup 2: Fan-Control With Triplicated PID Controller

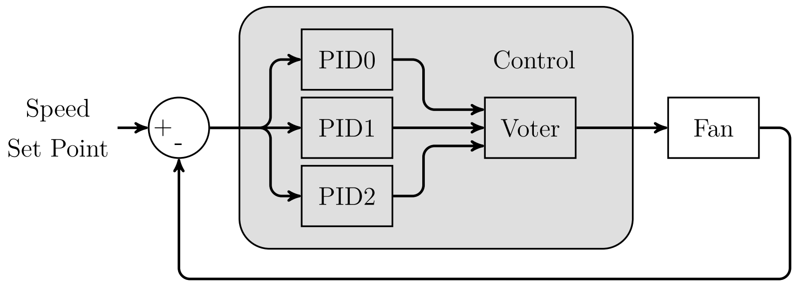


Figure 6.3.: Fan control application with triplicated PID controller.

In the second setup, triple modular redundancy is used to mask the failure of one PID controller. To determine the correct control value that can be sent to the fan, a voter is required. Figure 6.3 illustrates the closed loop of the control algorithm.

The schedule for this setup is shown in Figure 6.4 where the transmission of the *DestinationRPM* every $10s$ remains equal. The same accounts for the fan's execution at offset $1ms$ and the transmission of the current speed at $2ms$. Since there are multiple receivers for this value, the message can be sent as a multicast. The controllers execute the PID algorithm after receiving the current speed ($3ms$) and send the result to the voter afterwards. At the same time, it is possible to transmit only one message via the same link. Hence, the outputs are sent in a sequence at $4ms$ (*PID0*), $5ms$ (*PID1*) and $6ms$ (*PID2*). According to the triple modular redundancy concept, the voter compares the three inputs at $7ms$. Based on a majority decision, it is able to detect a failure of one PID controller if the other two outputs are equal. In case of three different outputs, it is not possible to determine the correct value and the voter puts the system into a safe state. To reach this, it forwards a control value which causes the fan to run at its maximum speed. However, this mechanism has disadvantages if two controllers fail providing the same output. Due to the majority decision, a faulty value is forwarded in this case. Using different PID implementations can prevent this issue. The voting result is sent to the fan (message *PID*) at offset $8ms$ and received $1ms$ later.

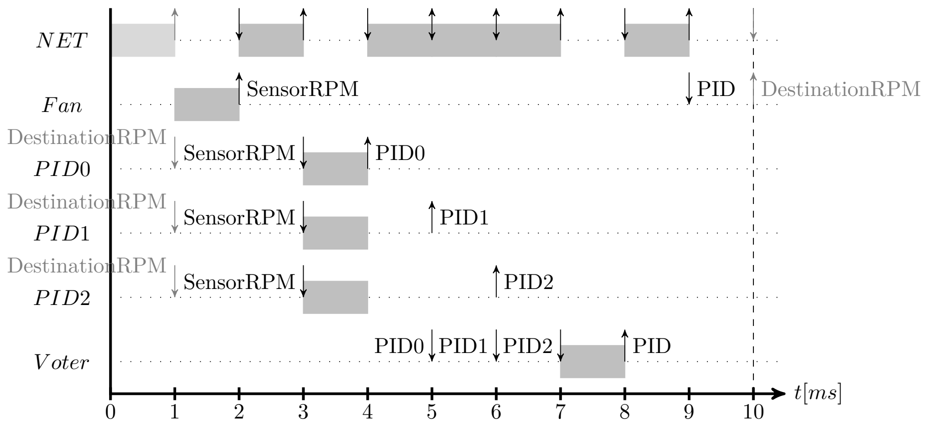


Figure 6.4.: Schedule in DUT with triplicated PID controller for one period of 10ms.

Setup 3: Fan-Control With Triplicated PID Controller and Voter

The third setup extends the triple modular redundancy concept to the voters. In this way, it is further possible to mask failures in one of these components. The resulting closed control loop is shown in Figure 6.5. On the one hand, the PID controllers must send their results to all voters. On the other hand, the fan task must also include a voting mechanism which determines the correct control input. Similar to the previous setups, the fan is responsible to send new destination speed set points. The temporal parameters remain equal for this message type compared to Setup 2.

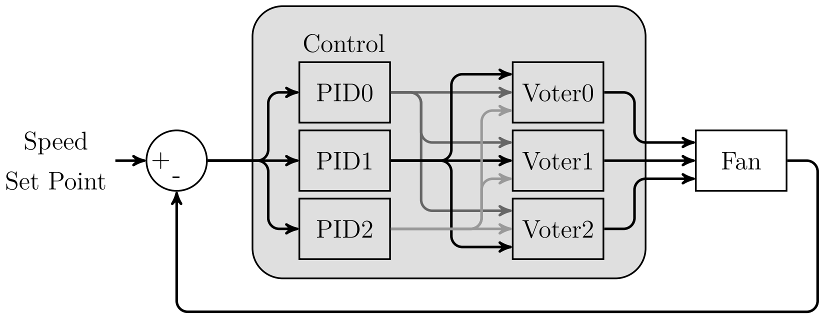


Figure 6.5.: Fan control application with triplicated PID controller and voter.

Figure 6.5 presents the schedule for this setup. To maintain a period of 10ms, the fan has to execute at offset 0ms wherefore the current speed is sent as a multicast at 1ms. This message is received by the PID controllers after 2ms which send the related control inputs in a sequence at 3ms, 4ms and 5ms. Since there are multiple receivers for these messages, they can also be sent as multicasts. The voters execute at offset 6ms and transmit the outputs at *7ms* (*V0*), 8ms (*V1*) and 9ms (*V2*). Before the fan is able to adapt and sample the speed at the beginning of the next period, it must execute a voting on these inputs. The algorithm is the same like the one of the voters.

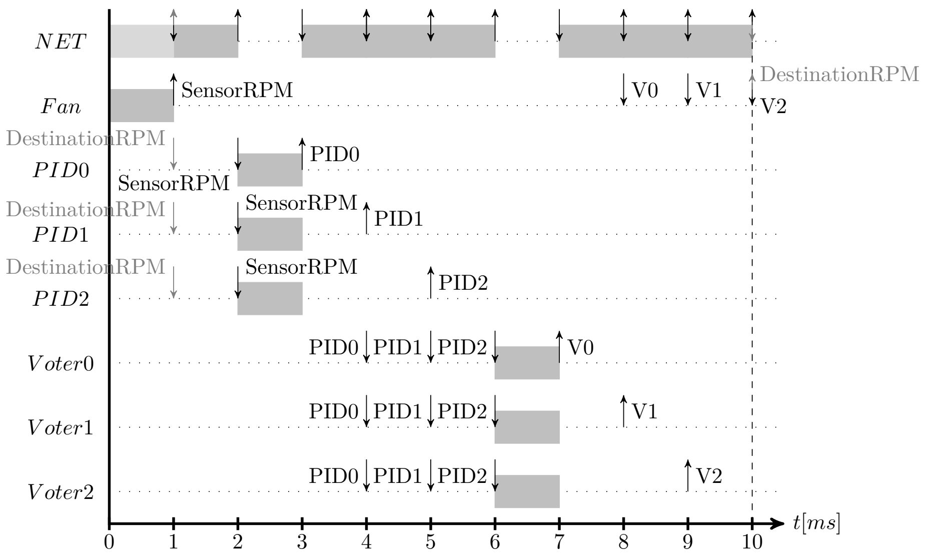


Figure 6.6.: Schedule in DUT with triplicated PID controller and voter.

Setup 4: Duplicated Fan-Control Based on Setup 3

In this setup, the components of Setup 3 are duplicated representing two parallel applications (cf. Figure 6.7). While both fans are connected to the central switch (*Switch1*), the PID controllers and voters of both applications are linked via *Switch0* and *Switch2*, respectively. To simplify the figure and the schedule, the fan, PID and voter instances are summarized in one box. Index i of the fan indicates the application index which is either 0 (red messages) or 1 (blue arrows). Index j denotes the PID and voter indexes similar to the previous setups (0 to 2).

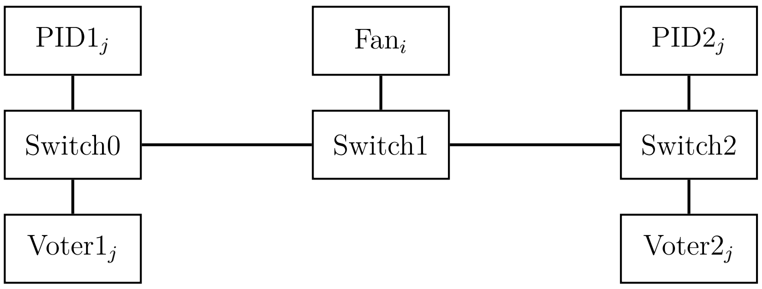


Figure 6.7.: Duplicated fan control application based on Setup 3.

To exchange all data in one period, it is extended to $20ms$. The fan instances execute at $1ms$ sending their outputs at $2ms$ and $3ms$, respectively. After passing the switches with $1ms$ delay, the PID controllers receive the values at $4ms$ and $5ms$. While the exchange of the PID outputs is similar to the previous setup, the voter outputs are sent with $2ms$ delay in between. In this way, the message schedule is simplified. For the same reason, the destination inputs are sent at the beginning of the period instead of the end.

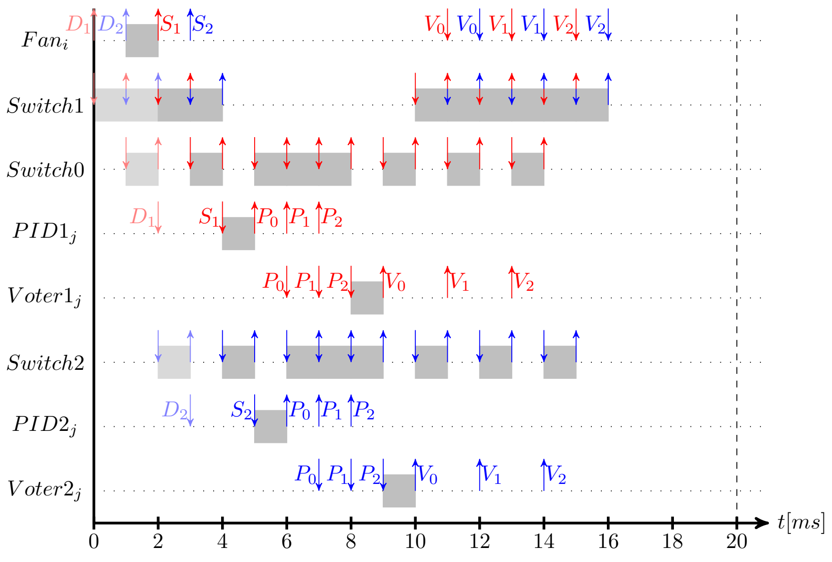


Figure 6.8.: Schedule of duplicated fan control application.

Setup 5: Triplicated Fan-Control Based on Setup 3

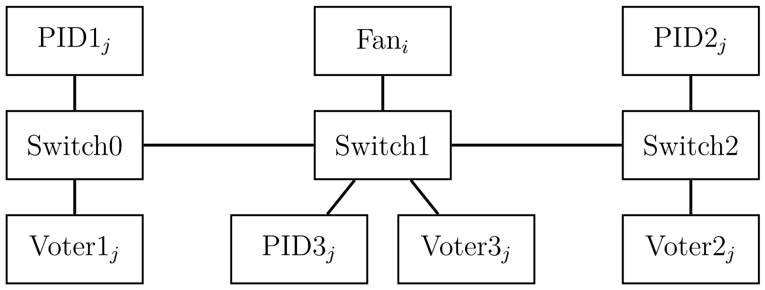


Figure 6.9.: Triplicated fan control application based on Setup 3.

Another set of instances is added in Setup 5 so that there are three parallel fan control applications. As shown in Figure 6.9, the additional components are added to *Switch1* and their messages are colored in cyan. The related schedule is similar to the one of Setup 4 with some modifications (cf. Figure 6.10). On the one hand, all task instances are delayed by $1ms$ wherefore the fans start at $3ms$ now. On the other hand, the gap between the voter outputs is increased to $3ms$ to include the additional outputs of the third application set.

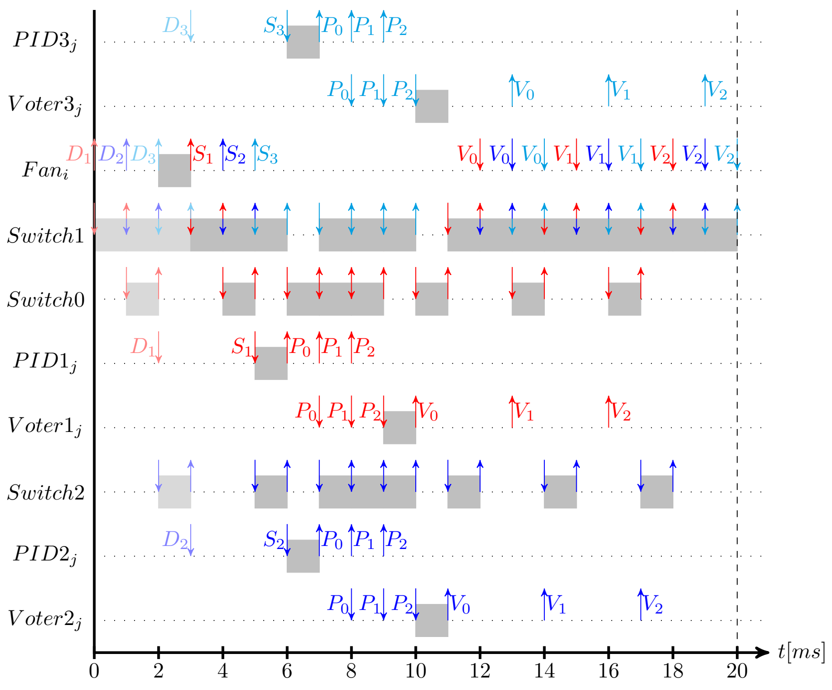


Figure 6.10.: Schedule of triplicated fan control application.

6.1.2. FMUs of Fan, PID Controllers and Voter

This section starts with a description of the PID controllers, followed by the voting algorithm. The fan is implemented in two ways: as a simulation model and on real hardware. The implementations of the model is presented afterwards whereas the implementation for the real hardware follows in the next section. Finally, the algorithms are combined to construct the state-estimation models.

While the network model simply adapts the transmission time-stamps encoded into the packets by the configured delays, the other FMUs are more complicated. Proportional, Integral and Derivative (PID) controllers consist of proportional, integral and derivative parts. The proportional part amplifies the input signal by a given constant [ZR17, p. 106]. Using only the proportional part, a difference to the reference set point remains. This difference can be reduced using an integral part [ZR17, p. 116] while adding a derivative gain enables the controller to react faster on disturbances [ZR17, p. 127]. Equation 6.1 depicts the superposition of the three gains as a continuous mathematical equation.

$$y\left(t\right) = K_{PID}\left[e\left(t\right) + \frac{1}{T_N}\int_0^t e\left(t\right)dt + T_V\frac{de}{dt}\right] \tag{6.1}$$

The implementation of a digital controller requires the transformation of Equation 6.1 into the discrete domain as depicted in Equation 6.2. Calculating the integral part can be realized by accumulating the inputs (e_i) weighted by the integral gain ($1/T_N$). The difference between the current (e_k) and the last input (e_{k-1}) multiplied by the derivative gain (T_V) and divided by the sampling time (Δt) represents the derivative part.

$$\begin{aligned} y_k &= K_{PID}\left[e_k + \frac{1}{T_N}\sum_{i=0}^{k-1} e_i\Delta t + T_V\frac{e_k - e_{k-1}}{\Delta t}\right] \\ &= K_{PID}\left[e_k + \frac{1}{T_N}e_{k-1}\Delta t + \frac{1}{T_N}\sum_{i=0}^{k-2} e_i\Delta t + T_V\frac{e_k - e_{k-1}}{\Delta t}\right] \end{aligned} \tag{6.2}$$

$$y_{k-1} = K_{PID}\left[e_{k-1} + \frac{1}{T_N}\sum_{i=0}^{k-2} e_i\Delta t + T_V\frac{e_{k-1} + e_{k-2}}{\Delta t}\right] \tag{6.3}$$

Accumulating the weighted inputs can be prevented if the sum is removed from the equation. This can be realized using a recursive algorithm where the portion of the current step is added to the previous output. Hence, the sum in Equation 6.2 is split into two parts. The first part represents the portion for the previous value while the second part is the accumulated sum until e_{k-2}. The same sum can be found in the calculation for the previous output y_{k-1} as shown in Equation 6.3. By calculating the difference between the current and the previous output, the sum can be eliminated. This difference represents the portion for the current input and is depicted in Equation 6.4.

$$y_k - y_{k-1} = K_{PID}\left[(e_k - e_{k-1}) + \frac{1}{T_N}e_{k-1}\Delta t + T_V\frac{e_k - 2e_{k-1} - e_{k-2}}{\Delta t}\right] \tag{6.4}$$

After reordering all parts, Equation 6.5 shows the resulting formula on which the discrete PID algorithm is based. It depends only on the current and the previous two inputs (e_i), the proportional gain, the reset time T_N, the derivative time T_V and the sampling time Δt. If the system can guarantee a constant sampling time, all factors can be calculated in advance which simplifies the execution.

$$y_k = y_{k-1} + K_{PID}\left[\left(1 + \frac{T_V}{\Delta t}\right)e_k - \left(1 - \frac{\Delta t}{T_N} + 2\frac{T_V}{\Delta t}\right)e_{k-1} + \frac{T_V}{\Delta t}e_{k-2}\right] \tag{6.5}$$

Algorithm 8 Implementation of the PID algorithm

```
1: a0 = kp + (kd / deltaT);
2: a1 = -kp + (ki * deltaT) + (2 * kd / deltaT);
3: a2 = kd / deltaT;
4: procedure RunPID
5:     ek = referenceSpeed - currentSpeed;
6:     yk = yk_1 + (a0 * ek) + (a1 * ek_1) + (a2 * ek_2);
7:     if yk > MAX_Y then
8:         yk = MAX_Y;
9:     end if
10:    if yk < MIN_Y then
11:        yk = MIN_Y;
12:    end if
13:    ek_2 = ek_1;
14:    ek_1 = ek;
15:    yk_1 = yk;
16:    encapsulate data into message;
17:    send message;
18: end procedure
```

Algorithm 8 illustrates the implementation of the PID control algorithm. Assuming a constant sampling time, the constant factors of Equation 6.5 are calculated in advance in Lines 1-3. In Line 6, they are multiplied with the current and previous inputs to calculate the current PID control value. The current input is the difference between the reference and the current speed (Line 5). A comparison with the maximum (*MAX_Y*) and minimum (*MIN_Y*) control outputs (Lines 7-12) ensures a bounded fan speed. After updating the previous output and inputs (Lines 13-15), the PID output is encapsulated into a message and sent as an Ethernet frame (Lines 16-17). If a constant sampling time cannot be guaranteed, the factors have to be calculated during each function call instead.

Once the voter received all inputs from the PID controllers, it can perform the majority election according to Algorithm 9. During the election, the different combinations of the PID inputs are checked to determine two equal values. This value is set as the PWM output afterwards. The algorithm starts with checking the equality of all inputs in Line 2 which is given as long as there is no failure. Afterwards, the equality checks of *PID0* and *PID1* (Line 4), *PID0* and *PID2* (Line 6) and *PID1* and *PID2* (Line 8) follow. Finally, Lines 10-11 represent the case where all inputs differ. Since there is no chance to select the correct output, a value of *PWM_MAX* is set which causes the fan to run at its maximum speed. Similar to the PID controllers, the output is encapsulated into a message and sent as an Ethernet frame in Lines 13-14.

Algorithm 9 Implementation of the voting algorithm

```
 1: procedure RunVoting
 2:     if PID0 == PID1 and PID1 == PID2 then
 3:         PWMout = PID0;
 4:     else if PID0 == PID1 and PID0 != PID2 then
 5:         PWMout = PID0;
 6:     else if PID0 != PID1 and PID0 == PID2 then
 7:         PWMout = PID0;
 8:     else if PID0 != PID1 and PID1 == PID2 then
 9:         PWMout = PID1;
10:     else if PID0 != PID1 and PID0 != PID2 and PID1 != PID2 then
11:         PWMout = MAX_PWM;
12:     end if
13:     encapsulate data into message;
14:     send message;
15: end procedure
```

The last device FMU described in this section is an Intel E97379-001 fan which can be described by the superposition of two signals. Both signals can be modeled as first-order delay elements. The first one represents a basic speed according to Equation 6.6. It raises from $0RPM$ to $1000RPM$ during the first $3.5s$ and remains stable afterwards. The second portion is controllable using a Pulse Width Modulation (PWM) and represented by Equation 6.7. At runtime, it is multiplied with the PWM input received from the voter and added to the base speed. This results in the final speed of the fan.

$$a\left(t\right) = 10 * \left(1 - e^{\frac{-t}{0.46}}\right) \;\; and \; G\left(s\right) = \frac{10}{1 + 0.46s} \tag{6.6}$$

$$a\left(t\right) = 10 * \left(1 - e^{\frac{-t}{0.87}}\right) \;\; and \; G\left(s\right) = \frac{10}{1 + 0.87s} \tag{6.7}$$

Besides the device FMUs, the models for the evaluation of the state-estimation mechanism must be defined. All device models calculate the difference between the received and the estimated input and compare it with the maximum error set during the configuration process. If the error is too large, the FMU returns with status *fmi_status_error* while a valid estimation returns *fmi_status_ok*. To realize the system model, the algorithms described above can be reused. Each system model must provide the same inputs like the device from which the data is received. The resulting packets are inserted into the estimated packet buffer at the scheduled reception time. Hence, the system model of the devices consist of the following algorithms. The fan requires the PWM control value provided by the PID controllers. Since TMR is not needed for the estimation, only the PID algorithm is implemented and the voter is neglected. The inputs of the PID controllers are the current fan speed instead which is provided by Equations 6.6 and 6.7. In addition, they receive new reference values from the fan. Those values are encoded in an array and the model creates the new destination's packet after the period elapsed. The same accounts for the fan model where the index of the reference speed array is updated after this time. Implementing the voter's system model is more complicated. The voter's inputs are the three PID outputs while it forwards one of these packets to the fan. Hence, the system model has to calculate the fan speed and to execute the PID algorithm based on it. Furthermore, it must triplicate the output by constructing three packets for the scheduled reception times of the PID inputs. The other system models have to create only one input.

6.1.3. Implementation of Fan on Zynq-XC7Z010 Hardware

After introducing the FMUs of the PID controllers, the voting algorithm and the fan model in the previous section, this section focuses on the implementation of the latter on real hardware. A Digilent ZYBO Z7 development board is used as a target device. It is based on the Zynq-XC7Z010 variant of the Xilinx All Programmable System-on-Chip architecture which integrates the Xilinx 7 FPGA logic with an ARM Cortex-A9 processor [Xilb].

The fan control algorithm is executed on the ARM processor using version 10.1.1 of FreeRTOS. FreeRTOS is an open source real-time kernel which is available for different platforms including the Zynq architecture [Bar]. Furthermore, it supports lwIP, a light-weight TCP/IP stack implementation for embedded systems [DW]. While Algorithm 10 introduces the initialization of the algorithm, Figures 6.11 and 6.12 depict the threads for fan and execution control at runtime.

Algorithm 10 Initialization of the fan control algorithm.

```
 1: procedure MAIN
 2:     start MainThread;
 3:     start scheduler;
 4:     while 1 do
 5:         run forever;
 6:     end while
 7: end procedure
 8:
 9: procedure MAINTHREAD
10:     initialize lwIP;
11:     create and start NetworkThread;
12:     delete task and return;
13: end procedure
14:
15: procedure NETWORKTHREAD
16:     initialize and activate network interface;
17:     start lwIP packet reception thread;
18:     start fan control thread;
19:     start execution control thread;
20:     delete task and return;
21: end procedure
```

To setup the fan and execution control threads, a thread hierarchy of 4 levels is required. The procedures of Algorithm 10 are listed according to this hierarchy. The main function is the root of the tree which starts the main thread and the FreeRTOS scheduler (Lines 2-3). Afterwards, it starts a loop which runs forever (Lines 4-6). The main thread is responsible to initialize lwIP (Line 10) and to start the network thread (Line 11). The latter is related to the network interface used for the communication with the simulation host. In this way, the application supports the communication via multiple network interfaces in parallel. Each network thread needs to initialize and activate the interface (Line 16) and to start a thread which handles the reception of packets (Line 17). Afterwards, the application threads can be started as the fourth level and the leafs of the thread hierarchy. The first thread (Line 18) controls the fan while the second one is used to exchange control commands with the simulation host (Line 19). Hereby, the distributed co-simulation framework is able to start and stop the execution in synchronization with the remaining simulations or real devices.

Figure 6.11 shows the execution control thread. It starts with the creation of a UDP socket for the communication with the simulation bridge in the initialization state. Assuming a usage of the fan control application in a train, the socket for application data is bound to port 17224 which is assigned to the TRDP protocol. With regard to this protocol, the control socket is bound to port 17225. Once the initialization is complete, the thread enters its main loop. In the waiting states, the thread checks periodically if a packet is available and if the command equals the one required to advance. Since the thread continues its execution while the fan control thread is running, a continuous polling might cause deadline misses. Hence, it is suspended for a short period between fetching the inputs. For the evaluation in this chapter, a period of $10ms$ is used with an offset of $5ms$. In this way, the check is performed when there is no event scheduled for the fan control. After receiving the correct command, the state machine advances to the start and stop states. Herein, a shared variable is set to signal the related command to the fan control thread.

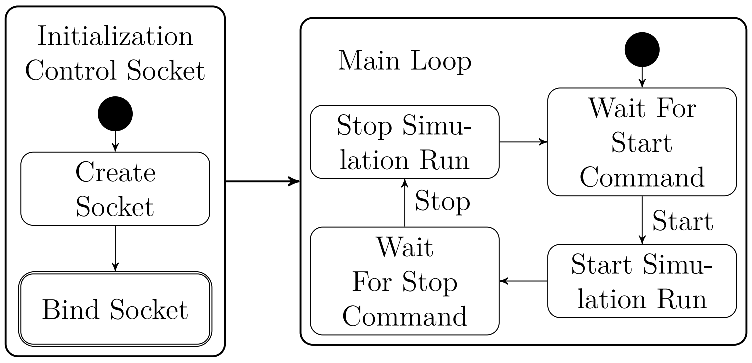


Figure 6.11.: Execution control thread.

The state machine of the fan control thread is shown in Figure 6.12. Similar to the previous one, it starts with the initialization of a UDP socket for the exchange of application data. In contrast to the control commands, the fan sends its speed outputs to the PID controllers as a multicast. Hence, the thread requests an IGMP membership for the related multicast IP address afterwards. After finishing the initialization, the thread waits for the command to start a simulation run. Once the command is received, the run is initialized by setting the start time to the current FreeRTOS tick. Furthermore, the first offsets of the schedule are initialized with regard to the start time. In the loop, the thread is suspended between the different offsets according to the schedule. To ensure a valid synchronization with the simulation host and a timely packet reception, a period of $100ms$ is used. At offset $10ms$, the thread determines the fan speed and encapsulates the value into a message. This message is sent at offset $20ms$ before the thread waits for the reception of a PWM output value. After receiving the value at offset $80ms$, the thread adapts the fan speed and the loop repeats. The simulation bridge stops the run by sending a stop command. When it is received, the member variables storing the current speed and the PWM output are reset and the thread waits for the next run.

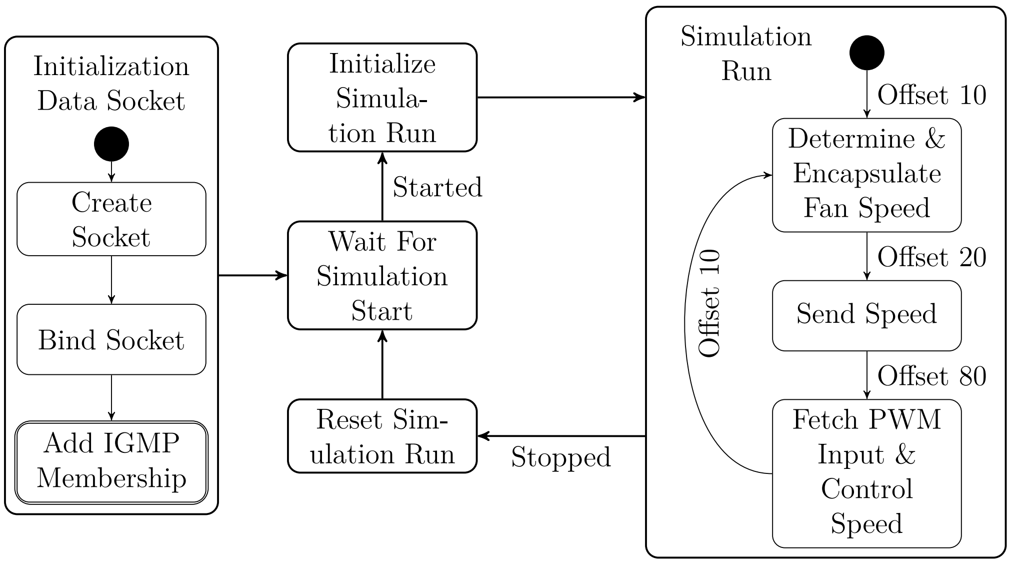

Initialization Data Socket
Create Socket
Bind Socket
Add IGMP Membership
Initialize Simulation Run
Started
Wait For Simulation Start
Reset Simulation Run
Stopped
Simulation Run
Offset 10
Determine & Encapsulate Fan Speed
Offset 20
Send Speed
Offset 80
Fetch PWM Input & Control Speed
Offset 10

Figure 6.12.: Fan control thread.

The implementation of the algorithm does not provide any fault-tolerance. It is only possible to detect delayed packets if there is no packet available at the scheduled reception time. In this case, a counter is increased. Its final value is printed while resetting the run and can be checked by the user. Besides that, the thread always fetches the next packet from the buffer. If a packet is delayed, replayed or inserted, the subsequent packets are also delayed by one period. The consequences of this implementation are used to demonstrate the fault-injection for HIL testing in Section 6.3.

6.2. Implementation of the Distributed Co-Simulation Framework on Linux

In the previous two chapters, different algorithms for packet forwarding, synchronization and delay management in the distributed co-simulation framework were presented. Based on them, this section depicts implementation details of the framework's simulation bridges on a Linux host. Furthermore, a suitable configuration of a Linux PC for real-time execution is determined. Herein, different mechanisms provided by Linux are validated with regard to real-time performance.

6.2.1. Implementation of Simulation Bridges

The simulation bridges as the main components of the distributed co-simulation framework are implemented as a proof-of-concept in C++ and can be executed on a Linux PC. They use several techniques provided by Linux such as the deadline scheduler for real-time execution, the locking of memory pages or the PCAP library for capturing network traffic. Furthermore, platform-independent libraries such as the Standard Template Library (STL) of C++, FMI or the TinyXML2 parser are used.

To interface a device with a simulation bridge, there are two possibilities which depend on the device's representation. Simulated devices and software applications are connected via FMI. Besides FMI, a software application can be connected further using the Linux packet capture library PCAP. This library is also used for real hardware devices. Both possibilities require parameters that are set by the configuration module. In case of FMI, those parameters are the path to the FMU and a directory to extract it. The code of the FMU is loaded as a shared library. PCAP requires the name of the Ethernet interface and the MAC and IP addresses of the interface and the connected device instead. To read the parameters, the configuration module uses the TinyXML2 parser [Tho].

Capturing packets via FMI is realized using interface variables. They can be set and queried using setter and getter functions for the different FMI data types. Each variable is stored in an array of its type and the index in this array is stored as a constant. This constant can be used as a value reference in the setter and getter functions. A step is executed by calling the *doStep*-function. If no packets are available, *pending* signals the correct execution of the step without any outputs. Otherwise, the function returns with status *ok*. In this case, the FMU provides the number of packets sent, the logical send time for each packet and the packets themselves. The packet size has to be calculated using information provided by the headers. Captured packets are represented as C char arrays independent from the wrapper type. FMI wraps those arrays into strings and forwards them using a related setter function. However, typical packet headers contain bits which are interpreted as the terminating '\0' character. In this case, parts of the packet are lost wherefore a new setter function *setPacket* is implemented. This function passes only a pointer to the FMU which refers to the beginning of the packet. In this way, the entire packet is provided and the number of copy activities is reduced further.

If PCAP is used instead, the wrapper module starts a new thread which queries packets from the network interface periodically using *pcap_next_ex*. In contrast to FMI, the PCAP functions do not provide a means to determine the number of packets sent in one step. The same accounts for the logical send time. Hence, an additional header can be added in the payload of the transport protocol. It contains the send time as a 64 Bit unsigned integer and a char denoting if the packet is the last one sent in the current step. If no packet will be received for the current step, the connected device can send an empty packet with ether-type $0x06FF$ or a TCP/UDP packet on port 17225. Both values are not registered for any technology yet. After receiving such a packet, the bridge sends the request to advance in time without forwarding any output. To forward packets, the wrapper uses the function *pcap_inject*. In case there is no packet available for the current step, a control packet sent via port 17225 can trigger the execution if required.

To enable a parallel capture and forwarding of packets, the ingress and egress modules run in different threads. Hence, the packet buffers are implemented as monitors based on Hoare and Brinch-Hansen [TB15, p.137]. The functions to insert and to query packets are protected by mutexes and condition variables wherefore only one thread can access the buffer at runtime. Each interaction instance object is created as a shared pointer and inserted into an STL vector as a pair with its logical send time. Referencing the objects as a shared pointer prevents unnecessary copying at runtime and deletes the object automatically once all references on it are removed. In contrast, the input packet buffer uses a map which stores pairs of the event time and the shared pointer to the interaction instance object. The simulation bridges exploit the map's implementation as a tree and the simple search of its elements using the logical times as keys. Since an Ethernet network interface can receive only one packet at the same time, a map prevents the availability of multiple packets for the same event further. Similarly, the events of the schedule are stored in a set of the STL to prevent duplications.

As explained in Section 3.4, Linux provides the deadline scheduler which is able to guarantee a defined bandwidth for periodic tasks since kernel 3.14. If real-time execution is required, the scheduler is activated during the initialization of the simulation bridges. Furthermore, the *mlock* mechanism is activated to prevent a removal of pages from the memory. At runtime, the simulation bridge process is scheduled every millisecond which represents the period of the deadline scheduler. The WCET and the deadline can be configured manually depending on the system requirements. In addition to the scheduler and *mlock*, Linux provides further technologies which support real-time execution or influence it negatively. Those technologies are analyzed in the next section.

6.2.2. Configuration of Linux as Real-Time Host

Once real-time components are connected to the simulation bridges, the simulation host must also support a real-time execution. Otherwise, it is not possible to guarantee a timely reception of packets to the real-time device. In this section, different setups are examined to determine a sufficient real-time configuration of a general purpose Linux PC under normal load conditions.

Real-time applications can be influenced by other tasks if they share the same resources. Examples are the CPU including hyper-threading, memory or hard drives. Although hyper-threading provides multiple logical processors to the operating system, the tasks finally use the same physical execution units [KM03]. Besides these resources, interrupts can also affect the execution since the handler might block the real-time task.

Table 6.1 introduces six setups which combine the following mechanisms. As mentioned in the previous section, the Linux deadline scheduler is used. This scheduler guarantees to maintain the deadlines of real-time tasks as long as the overall utilization of all real-time tasks $\sum_{i=1}^{N} \frac{Q_i}{T_i}$ is below a certain threshold [LSAF16]. In the setups for this evaluation, two different WCETs are examined ($300\mu s$ and $800\mu s$). Using different CPU frequency governors, the Linux operating system can adapt the frequency of the CPU cores. The governors considered in the second column are *powersave* to save energy or *performance* which provides the highest possible frequency [PS06]. In the subsequent columns, hyper-threading, USB interrupts, the logging of data to the hard drive and the locking of pages in the memory are either enabled (check-marks) or disabled (x-marks). Setup A starts with a WCET of $800\mu s$ and the *powersave* governor. Furthermore, hyper-threading, USB interrupts and logging are enabled while pages are not locked in the memory. In Setup B, the governor is changed to *performance* while hyper-threading and USB interrupts are disabled. Setup C changes the WCET to $300\mu s$, Setup D disables data logging to the hard drive and the *mlock* mechanism to lock memory pages is enabled in Setup E. Finally, Setup F uses the *powersave* governor again and re-enables hyper-threading and USB interrupts. Each alteration compared to the previous setup is denoted as a gray cell.

Table 6.1.: Configuration of real-time host in different test cases.

	Schedule WCET	Frequency Governor	Hyper-Threading	USB	Log HDD	mlock
A	$800\mu s$	powersave	✓	✓	✓	✗
B	$800\mu s$	performance	✗	✗	✓	✗
C	$300\mu s$	performance	✗	✗	✓	✗
D	$300\mu s$	performance	✗	✗	✗	✗
E	$300\mu s$	performance	✗	✗	✗	✓
F	$300\mu s$	powersave	✓	✓	✗	✓

The evaluation is performed using Setup 2 of the fan-control application (cf. Section 6.1.1) considering the fan as the real-time component. A Dell PC with an Intel Core i7-8700 CPU (6 cores, 12 threads and 4.60 GHz maximum speed) and 16 GB RAM is used as the simulation host. Table 6.2 and Table 6.3 illustrate the performance results of the fan's simulation bridge. In Table 6.2, the numbers of erroneous events and missed packets is compared to the total number of events/packets. Table 6.3 shows the scheduling delay of the ingress thread and the duration to process the ingress and egress events. To determine the values, counters are added to the ingress module which count the number of forwarded received and estimated packets before the deadline has passed. Furthermore, the ingress and egress threads take a time-stamp once they are scheduled and when the event is processed. Missed events can be determined by comparing these time-stamps with the scheduled time-stamps of the events.

Table 6.2.: Missed events and packet loss.

	Events			Packets	
	Total	Errors	Fraction	Forwarded	Missed
A	2000000	66	0.033 [‰]	9999.43	0.57
B	2000000	38	0.019 [‰]	9999.74	0.26
C	2000000	56	0.028 [‰]	9999.58	0.42
D	2000000	2	0.001 [‰]	9999.98	0.02
E	2000000	0	0 [‰]	10000.00	0
F	2000000	0	0 [‰]	10000.00	0

Table 6.2 shows that the deadline scheduler on its own is insufficient to provide the required real-time guarantees as long as data is logged and memory pages can be swapped out. In Setup A, a fraction of 0.033‰ events is not scheduled in time. This fraction can be decreased to 0.019‰ in Setup B where the frequency governor is changed to *performance* and hyper-threading and USB interrupts are disabled. Reducing the WCET in Setup C has a negative effect since the guaranteed bandwidth of the real-time task is reduced strongly. This results in an increased fraction of 0.028‰. The activation timestamps of the threads and their durations show that each error is caused by a previous job that exceeded its WCET. The scheduler then throttles the task and continues its execution during the next period if there is no bandwidth to reclaim from other tasks. If multiple periods are affected, this might delay the execution of the next event. Logging data to the hard drive has a strong influence on this behavior since the related C++ function *fprintf* is indeterministic. Disabling it in Setup D reduces the number of missed events to 2 which corresponds to a fraction of 0.001‰. These two misses can be eliminated further by locking all memory page in the RAM in Setup E. Finally, Setup F shows the independence of a valid real-time configuration from the frequency governor, hyper-threading and USB interrupts. Even if those mechanisms are re-enabled, all events can be processed correctly. These results are strongly related to the average number of timely

forwarded packets. In Setup A, 0.57 packets are missed, 0.26 packets in Setup B and 0.24 packets in Setup C. The decreased number of missed events in Setup D is reflected further by only missing 0.02 packets. Finally, all packets can be forwarded in time once the events are processed correctly (Setups E and F).

As mentioned before, analyzing the time-stamps of the erroneous events shows large scheduling delays and processing durations. These results are reflected by the maximum scheduling delays and ingress durations for Setups A to C in Table 6.3. The maximum durations of the ingress threads range from $3040\mu s$ to $6287\mu s$ and exceed the bridge's WCET strongly. As a consequence, the execution of the subsequent events is delayed by $2124\mu s$ up to $4967\mu s$. However, the minimum delays of $0.00\mu s$ and the average value of up to $1.30\mu s$ show the performance of the scheduler in the absence of erroneous events. The same accounts for the minimum and average durations of the ingress and egress threads. For every setup, they are in the same order of magnitude and remain under the simulation bridge's scheduling period. After removing the logging from the ingress thread, the maximum scheduling delays can be bounded by a value of $65\mu s$ and the maximum durations of the ingress threads are lower than $600\mu s$. Although these durations are sufficient to process all events, they exceed the configured WCET of $300\mu s$. This might lead to deadline misses in heavy load situations. However, the WCET of the simulation bridge can be increased up to a value which is still smaller than the period of $1000\mu s$. Adding the maximum durations results in an upper bound of $765\mu s$ in Setup F which is still smaller than this period.

Table 6.3.: Duration of ingress and egress threads and ingress scheduling delays.

	Delay Ingress [μs]			Duration Ingress [μs]			Duration Egress [μs]		
	Min	Max	Average	Min	Max	Average	Min	Max	Average
A	0.00	4967	1.30	3.41	6287	9.08	20.21	256	26.70
B	0.00	2124	0.50	1.93	3040	4.44	21.83	175	27.33
C	0.00	3534	0.92	1.88	4570	4.54	21.46	168	27.31
D	0.00	24	0.04	2.00	412	3.69	20.05	223	25.44
E	0.00	29	0.03	2.00	407	3.78	18.96	239	25.20
F	0.00	65	0.15	3.80	595	8.39	18.86	170	25.10

To summarize the results of this section, it is possible to run real-time tests on a general purpose Linux without any packet loss. A sufficient configuration for normal load scenarios includes the Linux deadline scheduler and the *mlock*-mechanism to prevent the swapping of memory pages from the RAM. Furthermore, the logging of data to the hard drive must be disabled. In the next section, this setup is used to validate the influence of the fault-injection mechanism on a HIL setup of the fan-control application.

6.3. Influence of Fault-Injection on Fan-Control

This section aims on demonstrating the functionality of the fault-injection mechanism by manipulating the messages exchanged in Setup 2 of the distributed fan-control application. As described in Section 6.1, the application uses a fault-tolerance mechanism based on triple modular redundancy. It is able to detect faults in the three PID outputs wherefore the injection should target one of the remaining messages. Those are the destination speed messages exchanged during longer simulation runs for SIL testing and the fan's PWM control inputs in HIL testing.

6.3.1. Fault-Injection in SIL Testing

During longer simulation runs, the fan sends new destination speeds between $1000rpm$ and $2000rpm$ to the PID controllers every $10s$. These messages are manipulated in the following.

Figure 6.13 shows the distribution of the application subsystems via the distributed co-simulation framework. Each subsystem is executed on a separate host and a LAN is used to connect the nodes. The tests are performed as SIL where the fan is executed as a simulation model while the PID controllers and the voter are implemented as software applications. As PID parameters, the following values are used: the proportional gain is set to 0.0925, the integral gain equals 1.0 whereas the derivative gain remains 0.0. According to the application schedule described in Section 6.1.1, a sampling time of $1ms$ is used.

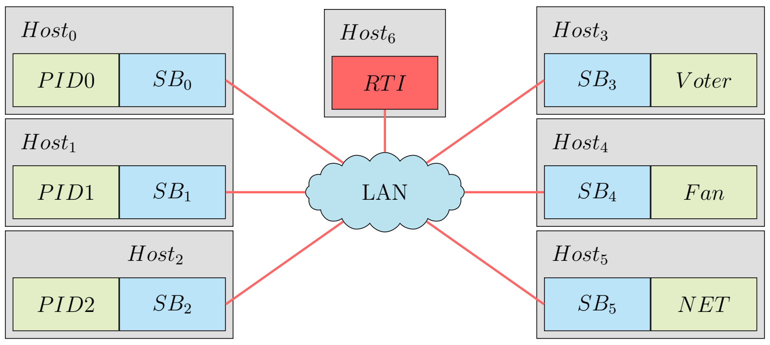


Figure 6.13.: Topology to demonstrate the fault-injection mechanism in a SIL test.

Based on IEC 61508-2 clause 7.4.11, Section 3.6 introduces seven message errors: delay, replay, omission, insertion, resequencing, corruption and manipulation. Since manipulation can be mapped to a corruption of the packet's sender address, it is not considered in the following. For the remaining errors, a simulation run of 50s is executed and faults are injected into dedicated destination speed messages.

Table 6.4.: Configuration of message errors.

Error	Message ID	Time	Parameters	Figure

Corruption	DestinationRPM	10001ms	numChangedBytes="4"
			changedIndexes="73,74,75,76"
			changedValues="1,3,5,0"
			recalculateChecksum="true"
				6.14

Delay	DestinationRPM	10001ms	delay="8000"	6.14

Insertion	DestinationRPM	15001ms	srcIP="172.16.1.10"
			destIP="224.0.0.2"
			transportProtocol="UDP"
			srcPort="17224"
			destPort="17224"
			data="DestinationRPM 1700"
				6.14

Omission	DestinationRPM	10001ms		6.15
Resequencing	DestinationRPM	10001ms	delay="15000"	6.15
Replay	DestinationRPM	10001ms	replayed="25001"	6.15

Table 6.4 denotes the fault parameters for each message error according to the framework's configuration model. Since the fault-injection targets only on the destination speed messages, the message ID is always set to *DestinationRPM*. The first error depicted is the corruption of the destination speed received at 10001ms. Its value is changed from 1600*rpm* to 1350*rpm*, wherefore four bits starting at byte index 73 in the packet must be adapted. Besides that, the packet's checksum is recalculated to prevent packet dropping in the destination devices. In the second run, the message of instant 10001ms is delayed by 8000ms while an additional packet for a destination of 1700*rpm* is inserted after 15001ms in the third run. This packet contains the IP addresses of the fan (172.16.1.10) and the destination speed multicast message (224.0.0.2). As the transport protocol, UDP via source and destination port 17224 is selected. The packet's data payload includes the message ID and the related speed. For omission, only the time of the injection at 10001ms is required while resequencing the packets can be realized by delaying the message at instant 10001ms by 15000ms. Finally, the destination speed message received at 10001ms is replayed at instant 25001ms. To improve the readability of the results, the fan speed graphs are printed in two Figures. The related number for each error is depicted in the last column. Both Figures plot the simulated time in seconds on the x-axis and the fan speed in revolutions per minute (rpm) on the y-axis.

Figure 6.14 illustrates the corruption, delay and insertion errors. To enable a comparison with the initial signal, the fan speed without any fault is printed further in black. The destination speed messages are sent with a period of $10s$. This interval is large enough so that the speed can reach the value as long as there is no error. According to the corruption of the destination speed message received at $10001ms$ ($10s = 10000ms$ period, $1ms$ offset), the fan accelerates only to a speed of $1350rpm$ instead of rising to $1600rpm$ (green graph). Since the difference to the subsequent destination of $1200rpm$ is smaller compared to the normal execution, this destination is reached faster afterwards. Due to the inserted delay of $8s$, the fan is not able to reach its destination of $1600rpm$ in the red graph. However, the next destination ($1200rpm$) can be reached faster similar to corruption. A converse effect can be observed in case of insertion (blue graph) where an additional destination of $1700rpm$ is received after $15001ms$. Although the first $5s$ interval ($10s$ to $15s$) is too short to reach the destination of $1600rpm$, the remaining difference is small enough to achieve the inserted destination. In consequence of the larger value, the fan requires more time to slow down to $1200rpm$. Since there is no further fault-injection performed in each case, the signal recovers after $30s$.

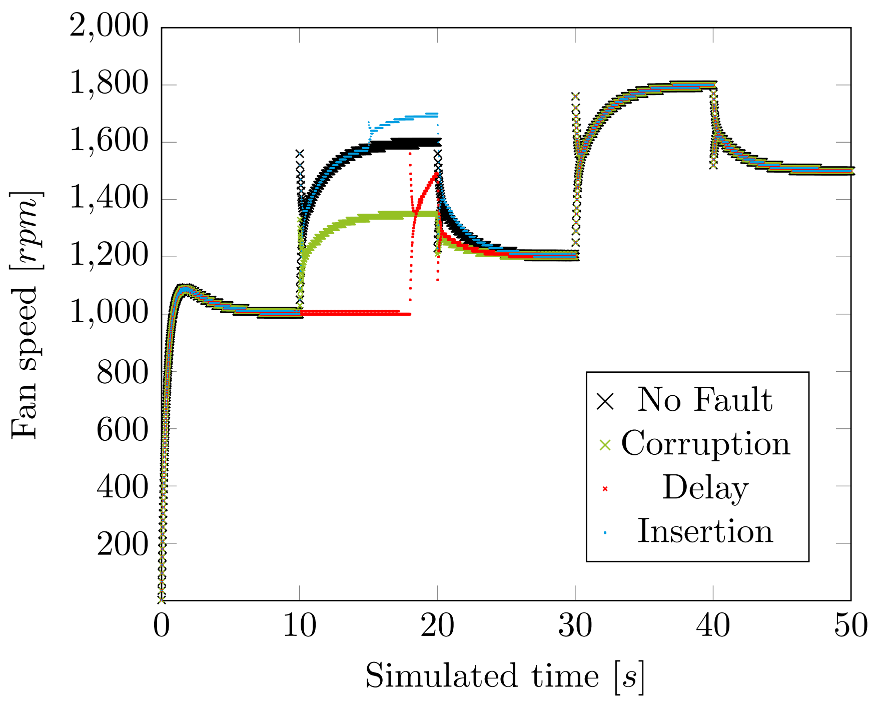


Figure 6.14.: Fan speed during normal execution, corruption, delay and insertion.

In Figure 6.15, the remaining message errors (omission, resequencing and replay) and the non-error case (black graph) are shown. The omission of the $1600rpm$ destination speed message at $10001ms$ is printed in green. In consequence of the fault, there is no activity between 10s and 20s and the speed rises to a value of 1200rpm once the next destination is sent. Delaying the value of 1600rpm by 15s results in resequencing with the 1200rpm destination. Hence, there is no activity between 10s and 20s while the fan accelerates in two steps between 20s and 30s (red graph). In both accelerations, the difference between the destinations can be nearly compensated and only small errors remain. Since the subsequent difference to 1800rpm after 30s is smaller compared to the non-error case, the value can be reached faster. Finally, the blue graph illustrates the replay of the 1600rpm destination after 25s. Until this time, the signal equals the one of the non-error case while it is almost similar to resequencing afterwards. The related difference in the interval of 25s to 30s is caused by the remaining errors in the speed signals at 25s. After 25s, there is no further fault-injection and all graphs are equal again after 40s.

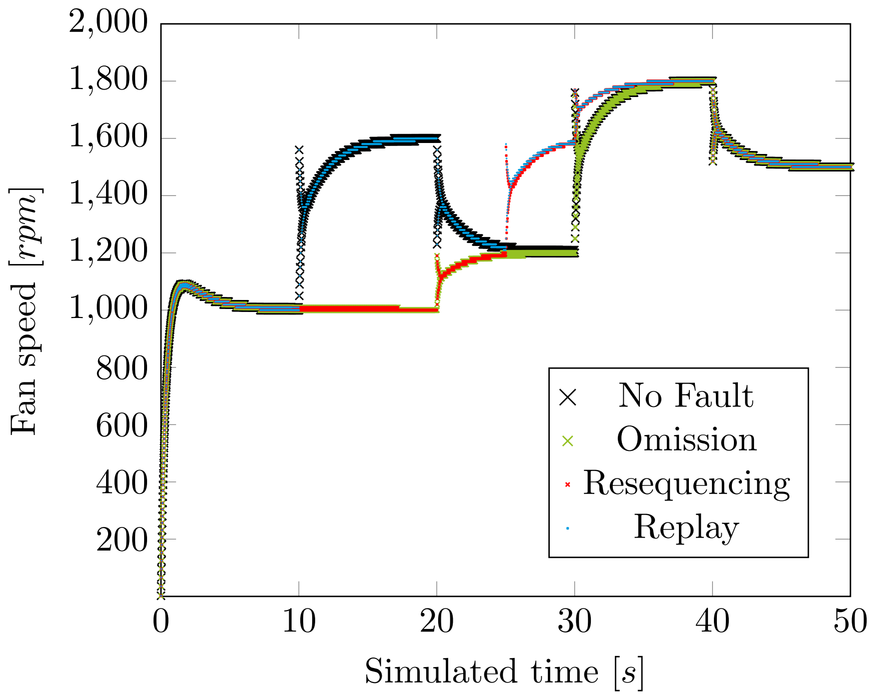


Figure 6.15.: Fan speed during normal execution, omission, resequencing and replay.

6.3.2. Fault-Injection in HIL Testing

According to Section 6.1.3, the fan control algorithm is executed on a Digilent ZYBO Z7 board during the HIL tests performed in this section. In the following, the PWM control inputs received from the voter are manipulated and the consequences on the fan speed are observed.

The distribution of the application subsystems via the distributed co-simulation framework is similar to the SIL tests from the previous section. Each subsystem except the fan is executed as a simulation on separate hosts. Only the fan subsystem runs on a ZYBO board as real hardware which is connected to $Host_4$ via an additional Ethernet interface on the host. To capture and forward the data, the simulation bridges use the Linux PCAP library as the wrapper interface. Figure 6.16 shows the resulting topology. With Vivado 2019.1, Xilinx introduced a TSN subsystem for the Zynq-7000 architecture of the ZYBO board [Xila]. However, the documentation of the subsystem is protected and was not available at the time of writing. Hence, the synchronization between the board and the simulation host is limited to the host's TSN driver and its egress time-slots. To prevent synchronization issues, the fan schedule presented in Figure 6.4 is scaled by a factor of 10 resulting in a period of $100ms$ and the related offsets. Consequently, the PID parameters are changed to 0.7 (proportional gain), 1.0 (integral gain) and a sampling time of $100ms$. The derivative gain of 0.0 remains.

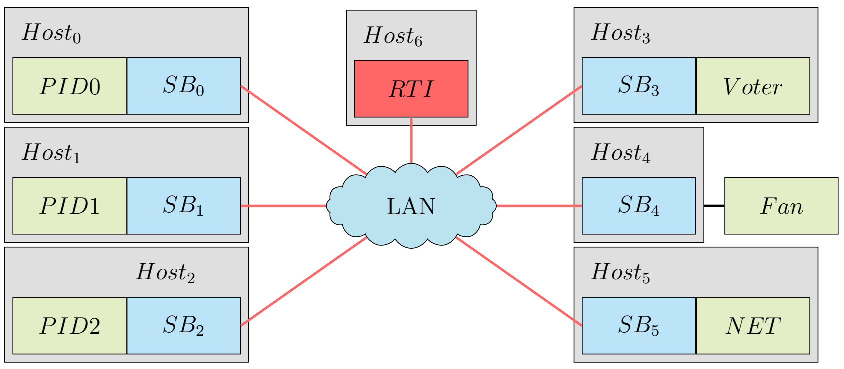


Figure 6.16.: Topology to demonstrate the fault-injection mechanism in a HIL test.

The fault parameters for the message errors are presented in Table 6.5 which is structured like Table 6.4 from the previous section. Each fault is injected into the PWM inputs of the fan. For corruption, the input received after $180ms$ is selected and the first two bytes of the value (indexes 62 and 63) are changed to 0. The checksum is recalculated to prevent packet dropping. The delay error targets the message received at $280ms$ and delays it by $50ms$ while a packet is inserted after $230ms$ during the third run. This packet is destined to the multicast address 224.0.0.6 and has an input value of 0. The remaining parameters are similar to the SIL test since the packet is sent from the fan (IP address 172.16.1.10) via UDP port 17224 for TRDP. While the packet received after $80ms$ is dropped to simulate omission, resequencing changes the packet sequence of $980ms$ (I), $1080ms$ (II), $1180ms$ (III) and $1280ms$ (IV). According to the delays injected, the resulting sequence is $1080ms$ (II), $1200ms$ (I), $1280ms$ (IV) and $1330ms$ (III). The roman numbers denote the initial order. Finally, the packet received after $1380ms$ is replayed after $3030ms$.

Table 6.5.: Configuration of message errors.

Error	Message ID	Time	Parameters	Figures

Corruption	PWM	180ms	numChangedBytes="2"
			changedIndexes="62,63"
			changedValues="0,0"
			recalculateChecksum="true"
				6.17 & 6.23

Delay	PWM	280ms	delay="50"	6.18 & 6.23

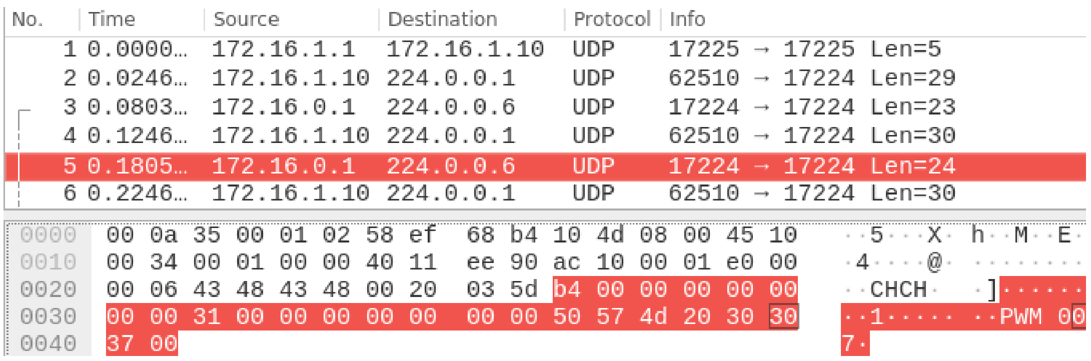
Insertion	PWM	230ms	srcIP="172.16.1.5"
			destIP="224.0.0.6"
			transportProtocol="UDP"
			srcPort="17224"
			destPort="17224"
			data="PWM 0"
				6.19 & 6.23

Omission	PWM	80ms		6.22 & 6.24

Resequencing	PWM
	PWM
		980ms
		1180ms
			delay="220"
			delay="150"
				6.21 & 6.24

Replay	PWM	1380ms	replayed="3030"	6.20 & 6.24

To demonstrate the correctness of the injection, Figures 6.17 to 6.22 show Wireshark packet captures of the related simulation runs. In each figure, the first packet represents the simulation start command sent from the simulation bridge to the ZYBO board while the remaining packets show the application data. Red colored packets denote the ones affected by the fault-injection as elucidated in the following.

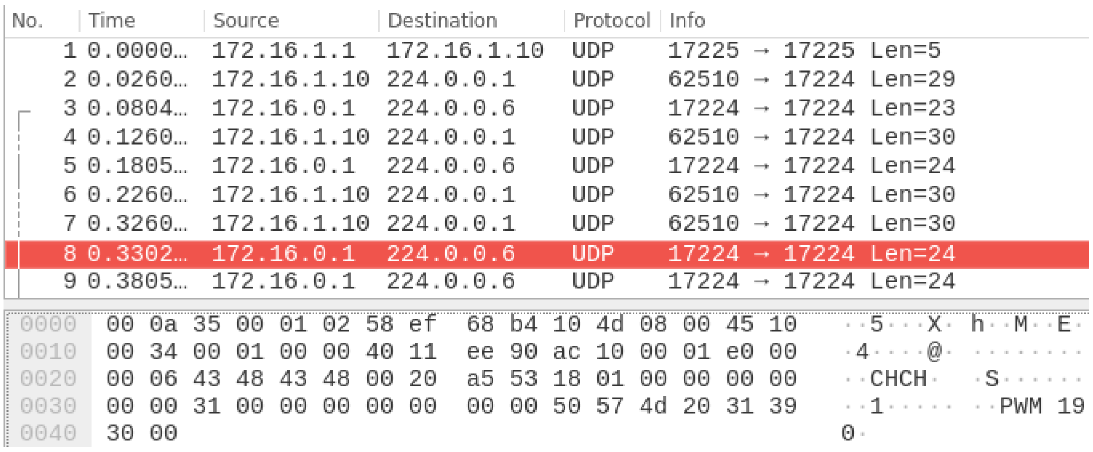


```
No.  Time       Source       Destination  Protocol  Info
  1 0.0000…  172.16.1.1   172.16.1.10  UDP       17225 → 17225 Len=5
  2 0.0246…  172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=29
  3 0.0803…  172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=23
  4 0.1246…  172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=30
  5 0.1805…  172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=24
  6 0.2246…  172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=30

0000  00 0a 35 00 01 02 58 ef  68 b4 10 4d 08 00 45 10   ··5···X· h··M··E·
0010  00 34 00 01 00 00 40 11  ee 90 ac 10 00 01 e0 00   ·4····@· ········
0020  00 06 43 48 43 48 00 20  03 5d b4 00 00 00 00 00   ··CHCH·  ·]······
0030  00 00 31 00 00 00 00 00  00 00 50 57 4d 20 30 30   ··1·····  ··PWM 00
0040  37 00                                              7·
```

Figure 6.17.: Packet capture with corruption error.

The corruption error affects packet number five as shown in Figure 6.17. According to the configuration, this packet is received 180ms after simulation start and the first two bytes of the PWM value are changed to 0.

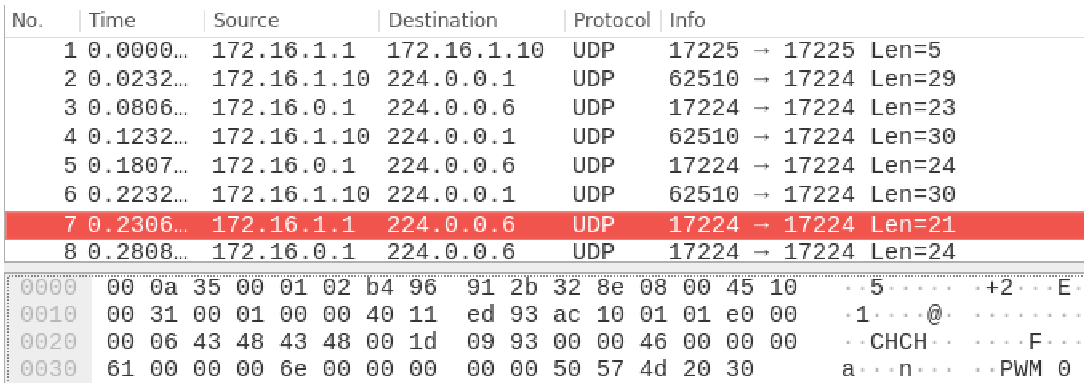
Figure 6.18 presents the delay error. The target packet is scheduled 280ms after simulation start. Applying the delay of $50ms$ results in a forwarding time at $0.3302s$. Figure 6.18 shows that there is no packet available between $0.2260s$ and $0.3260s$ (the fan outputs). Hence, packet number eight represents the delayed control input. As shown, there is an offset of $6ms$ in the time-stamps of the packets sent by the board. It is caused by the communication delays and a small offset between receiving the start command and the actual execution start.



```
No. | Time      | Source      | Destination | Protocol | Info
  1 0.0000…  172.16.1.1   172.16.1.10  UDP    17225 → 17225 Len=5
  2 0.0260…  172.16.1.10  224.0.0.1    UDP    62510 → 17224 Len=29
  3 0.0804…  172.16.0.1   224.0.0.6    UDP    17224 → 17224 Len=23
  4 0.1260…  172.16.1.10  224.0.0.1    UDP    62510 → 17224 Len=30
  5 0.1805…  172.16.0.1   224.0.0.6    UDP    17224 → 17224 Len=24
  6 0.2260…  172.16.1.10  224.0.0.1    UDP    62510 → 17224 Len=30
  7 0.3260…  172.16.1.10  224.0.0.1    UDP    62510 → 17224 Len=30
  8 0.3302…  172.16.0.1   224.0.0.6    UDP    17224 → 17224 Len=24
  9 0.3805…  172.16.0.1   224.0.0.6    UDP    17224 → 17224 Len=24

0000  00 0a 35 00 01 02 58 ef  68 b4 10 4d 08 00 45 10   ··5···X· h··M··E·
0010  00 34 00 01 00 00 40 11  ee 90 ac 10 00 01 e0 00   ·4····@· ········
0020  00 06 43 48 43 48 00 20  a5 53 18 01 00 00 00 00   ··CHCH·  ·S······
0030  00 00 31 00 00 00 00 00  00 00 50 57 4d 20 31 39   ··1·····  ··PWM 19
0040  30 00                                              0·
```

Figure 6.18.: Packet capture with delay error.

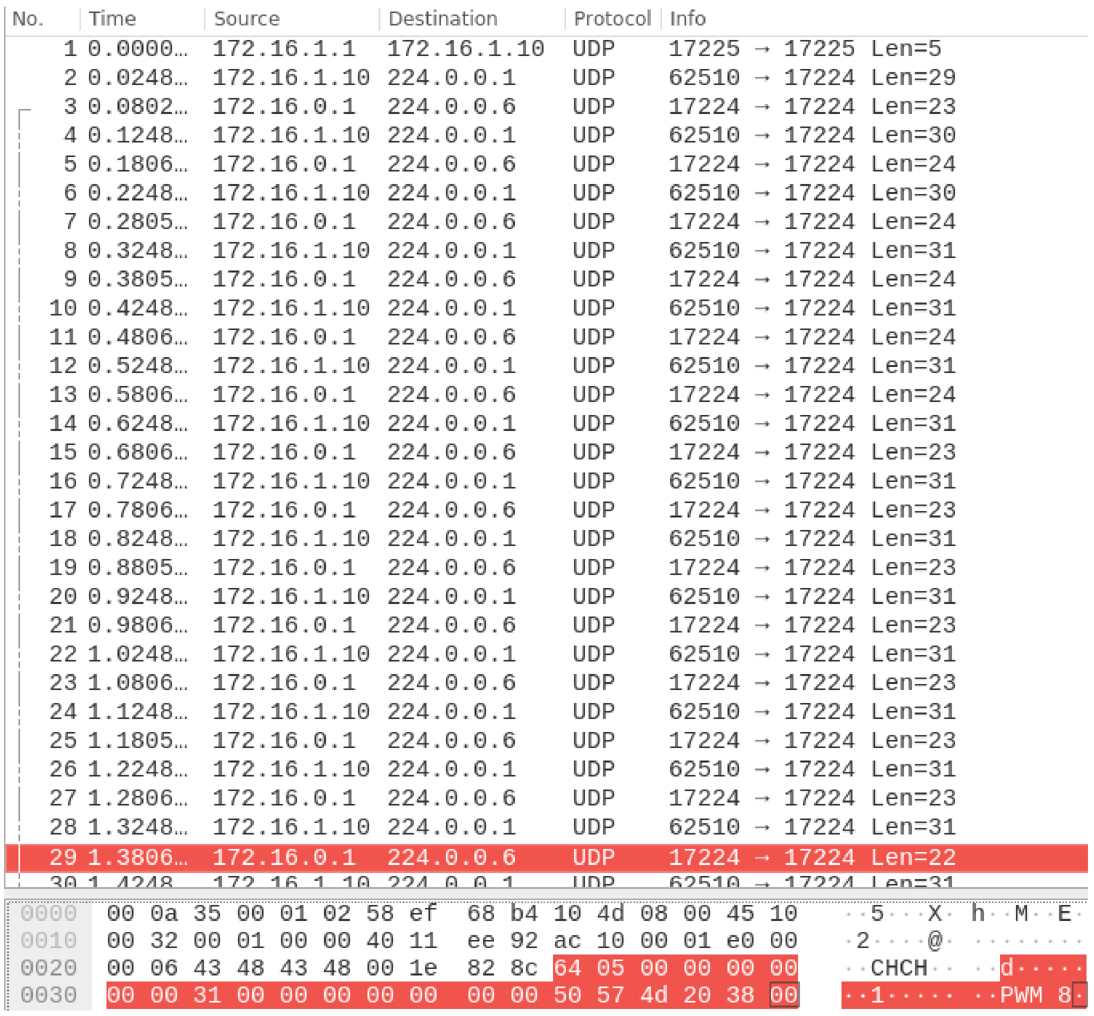
For insertion, an additional packet is added to the data stream at $230ms$. The packet capture for this message error is shown in Figure 6.19. The injection instant is related to packet seven captured at $0.2306s$. Its content is as specified in the configuration with the source IP address of the voter, the destination address of the PWM multicast and ports 17224. Furthermore, a PWM value of 0 is used.



```
No. | Time      | Source      | Destination | Protocol | Info
  1 0.0000…  172.16.1.1   172.16.1.10  UDP    17225 → 17225 Len=5
  2 0.0232…  172.16.1.10  224.0.0.1    UDP    62510 → 17224 Len=29
  3 0.0806…  172.16.0.1   224.0.0.6    UDP    17224 → 17224 Len=23
  4 0.1232…  172.16.1.10  224.0.0.1    UDP    62510 → 17224 Len=30
  5 0.1807…  172.16.0.1   224.0.0.6    UDP    17224 → 17224 Len=24
  6 0.2232…  172.16.1.10  224.0.0.1    UDP    62510 → 17224 Len=30
  7 0.2306…  172.16.1.1   224.0.0.6    UDP    17224 → 17224 Len=21
  8 0.2808…  172.16.0.1   224.0.0.6    UDP    17224 → 17224 Len=24

0000  00 0a 35 00 01 02 b4 96  91 2b 32 8e 08 00 45 10   ··5·····  ·+2···E·
0010  00 31 00 01 00 00 40 11  ed 93 ac 10 01 01 e0 00   ·1····@· ········
0020  00 06 43 48 43 48 00 1d  09 93 00 00 46 00 00 00   ··CHCH·· ····F···
0030  61 00 00 00 6e 00 00 00  00 00 50 57 4d 20 30      a···n··· ··PWM 0
```

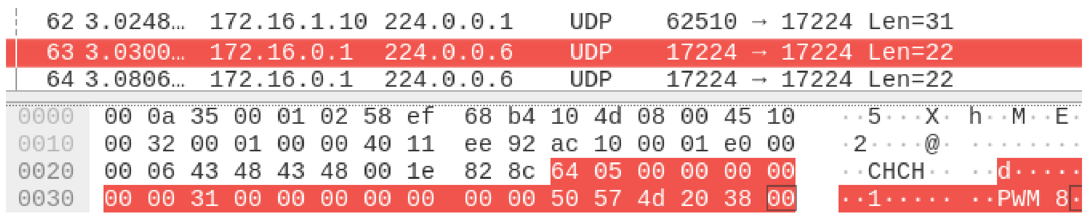
Figure 6.19.: Packet capture with insertion error.

Since there is a duration of $1650ms$ including 34 packets between the packet to replay and the scheduled replay instant, the capture is split into Figures 6.20a and 6.20b. The packet to replay is scheduled after $1380ms$ and replayed after $3030ms$. The related absolute times are $1.3806s$ and $3.0300s$. Comparing the contents of both packets, they are fully equal including the PWM input of 8. Furthermore, the subsequent packets 63 (3.0300s) and 64 (3.0806s) are both sent from the simulation host to the ZYBO board. This signals the insertion of an additional packet into the data stream.



```
No.   Time       Source       Destination  Protocol  Info
  1 0.0000…  172.16.1.1   172.16.1.10  UDP   17225 → 17225 Len=5
  2 0.0248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=29
  3 0.0802…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
  4 0.1248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=30
  5 0.1806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=24
  6 0.2248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=30
  7 0.2805…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=24
  8 0.3248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
  9 0.3805…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=24
 10 0.4248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 11 0.4806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=24
 12 0.5248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 13 0.5806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=24
 14 0.6248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 15 0.6806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
 16 0.7248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 17 0.7806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
 18 0.8248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 19 0.8805…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
 20 0.9248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 21 0.9806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
 22 1.0248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 23 1.0806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
 24 1.1248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 25 1.1805…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
 26 1.2248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 27 1.2806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=23
 28 1.3248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 29 1.3806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=22
 30 1.4248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31

0000  00 0a 35 00 01 02 58 ef  68 b4 10 4d 08 00 45 10   ··5···X· h··M··E·
0010  00 32 00 01 00 00 40 11  ee 92 ac 10 00 01 e0 00   ·2····@· ········
0020  00 06 43 48 43 48 00 1e  82 8c 64 05 00 00 00 00   ··CHCH·· ··d·····
0030  00 00 31 00 00 00 00 00  00 00 50 57 4d 20 38 00   ··1····· ··PWM 8·
```

(a) Packet capture showing packet to replay.



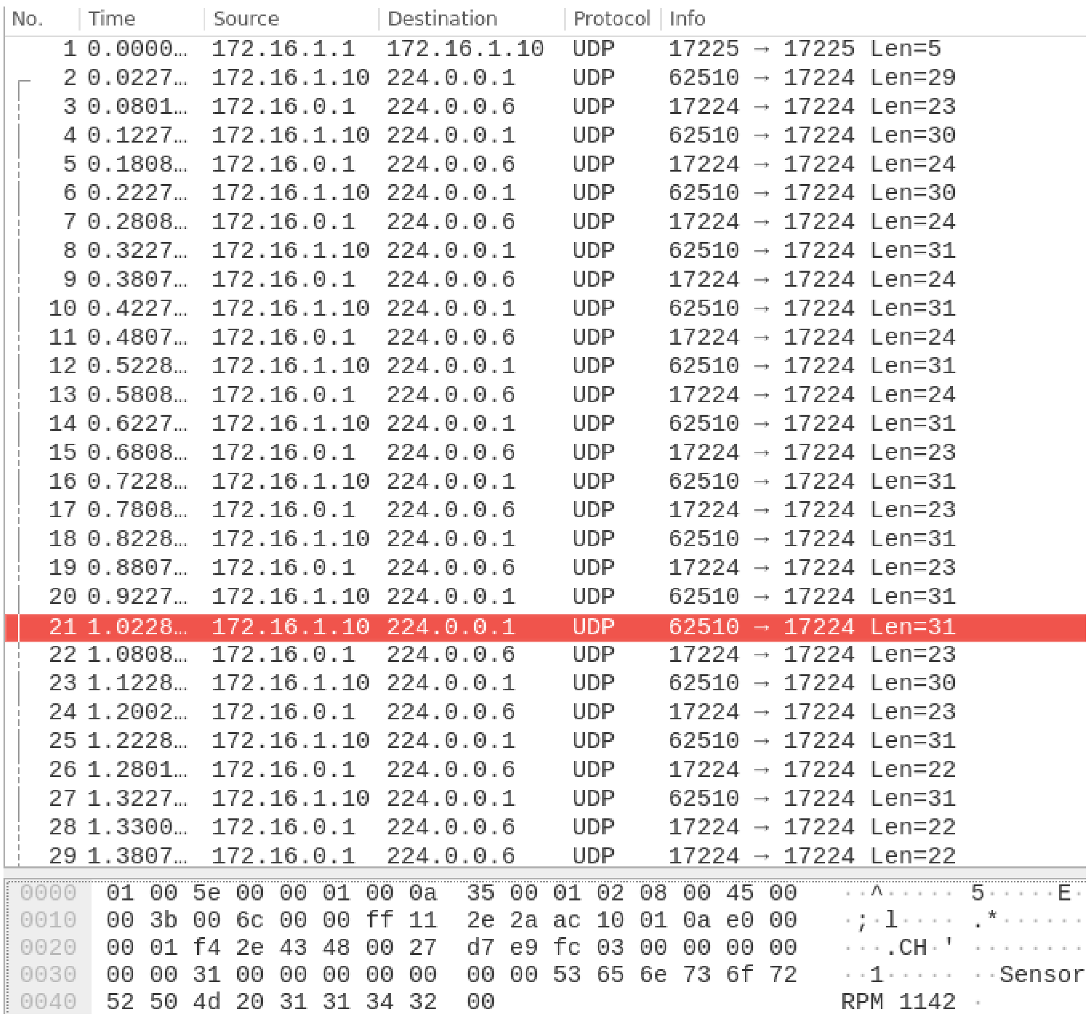
```
 62 3.0248…  172.16.1.10  224.0.0.1    UDP   62510 → 17224 Len=31
 63 3.0300…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=22
 64 3.0806…  172.16.0.1   224.0.0.6    UDP   17224 → 17224 Len=22

0000  00 0a 35 00 01 02 58 ef  68 b4 10 4d 08 00 45 10   ··5···X· h··M··E·
0010  00 32 00 01 00 00 40 11  ee 92 ac 10 00 01 e0 00   ·2····@· ········
0020  00 06 43 48 43 48 00 1e  82 8c 64 05 00 00 00 00   ··CHCH·· ··d·····
0030  00 00 31 00 00 00 00 00  00 00 50 57 4d 20 38 00   ··1····· ··PWM 8·
```

(b) Packet capture with replayed packet.

Figure 6.20.: Wireshark packet captures with replay error.

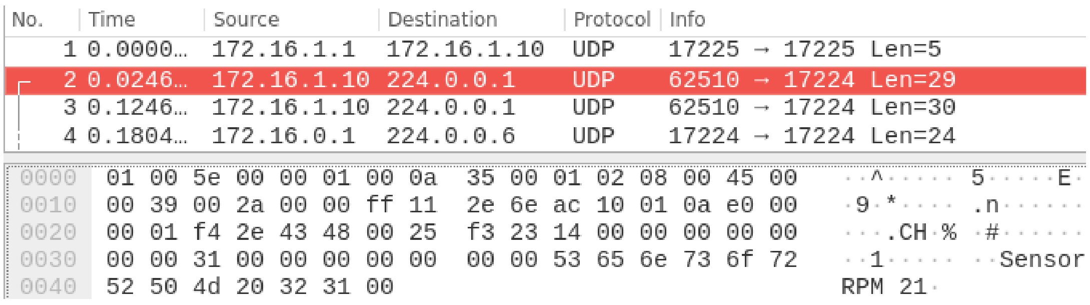
Figure 6.21 illustrates the packet capture with the resequencing error. In this case, the packets affected from the fault-injection are scheduled at 980ms (I), 1080ms (II), 1180ms (III) and 1280ms (IV). The delay of 220ms results in a forwarding time at 1.2002s for the first packet (packet 24). In addition to this packet, packet III is delayed by 150ms

resulting in a forwarding time of 1.3300s (packet 28). As a consequence, the capture shows a missing PWM input between packets 20 and 21 and an additional one between 27 and 29. According to the delays injected, the PWM input sequence is changed further.



```
No.  Time      Source       Destination  Protocol  Info
   1 0.0000…   172.16.1.1   172.16.1.10  UDP       17225 → 17225 Len=5
   2 0.0227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=29
   3 0.0801…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=23
   4 0.1227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=30
   5 0.1808…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=24
   6 0.2227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=30
   7 0.2808…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=24
   8 0.3227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
   9 0.3807…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=24
  10 0.4227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  11 0.4807…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=24
  12 0.5228…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  13 0.5808…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=24
  14 0.6227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  15 0.6808…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=23
  16 0.7228…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  17 0.7808…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=23
  18 0.8228…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  19 0.8807…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=23
  20 0.9227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  21 1.0228…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  22 1.0808…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=23
  23 1.1228…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=30
  24 1.2002…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=23
  25 1.2228…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  26 1.2801…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=22
  27 1.3227…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=31
  28 1.3300…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=22
  29 1.3807…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=22

0000  01 00 5e 00 00 01 00 0a  35 00 01 02 08 00 45 00   ··^·····5·····E·
0010  00 3b 00 6c 00 00 ff 11  2e 2a ac 10 01 0a e0 00   ·;·l····.*······
0020  00 01 f4 2e 43 48 00 27  d7 e9 fc 03 00 00 00 00   ···.CH·'········
0030  00 00 31 00 00 00 00 00  00 00 53 65 6e 73 6f 72   ··1·······Sensor
0040  52 50 4d 20 31 31 34 32  00                        RPM 1142·
```

Figure 6.21.: Packet capture with resequencing error.



```
No.  Time      Source       Destination  Protocol  Info
   1 0.0000…   172.16.1.1   172.16.1.10  UDP       17225 → 17225 Len=5
   2 0.0246…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=29
   3 0.1246…   172.16.1.10  224.0.0.1    UDP       62510 → 17224 Len=30
   4 0.1804…   172.16.0.1   224.0.0.6    UDP       17224 → 17224 Len=24

0000  01 00 5e 00 00 01 00 0a  35 00 01 02 08 00 45 00   ··^·····5·····E·
0010  00 39 00 2a 00 00 ff 11  2e 6e ac 10 01 0a e0 00   ·9·*····.n······
0020  00 01 f4 2e 43 48 00 25  f3 23 14 00 00 00 00 00   ···.CH·%·#······
0030  00 00 31 00 00 00 00 00  00 00 53 65 6e 73 6f 72   ··1·······Sensor
0040  52 50 4d 20 32 31 00                               RPM 21·
```

Figure 6.22.: Packet capture with omission error.

Finally, the packet selected for omission is received at 80ms. As shown in the packet capture of Figure 6.22, there is no packet forwarded between the sensor outputs at 0.0246s and 0.1246s wherefore the omission was successful.

Similar to the previous section, the presentation of the speed results is split into two figures. Figure 6.23 starts with the first three errors (corruption, delay and insertion) as depicted in the last column of Table 6.5. In addition to the these cases, the initial fan speed without any error is printed in black. The simulated time of $5s$ is plotted in seconds on the x-axis while the y-axis shows the fan speed in rpm. Absent from any fault, there is an overshoot up to $1400rpm$ during the first second. Afterwards, the speed reaches the destination of $1000rpm$ without further oscillation. As explained in Section 6.1.2, the fan speed can be modeled as two parts, a base speed and a proportion influenced by the PWM outputs. Corrupting the input scheduled at $180ms$ results in a decreased portion of the PWM part wherefore the speed is lower at this instant (green graph). The PID controllers compensate this fault with larger PWM outputs resulting in an increased overshoot. After around $0.8s$, the signal is recovered. Caused by the delay error, there is no actual packet available at $280ms$ (red graph). In contrast, the insertion of a 0 control input occasions a negative peak at $310ms$ (cyan graph). The behavior until the end of the execution is similar for both cases. Due to the missing fault-tolerance, the fan algorithm uses a control input which is destined for the previous period. This results in larger overshoots and oscillations afterwards until $2.5s$.

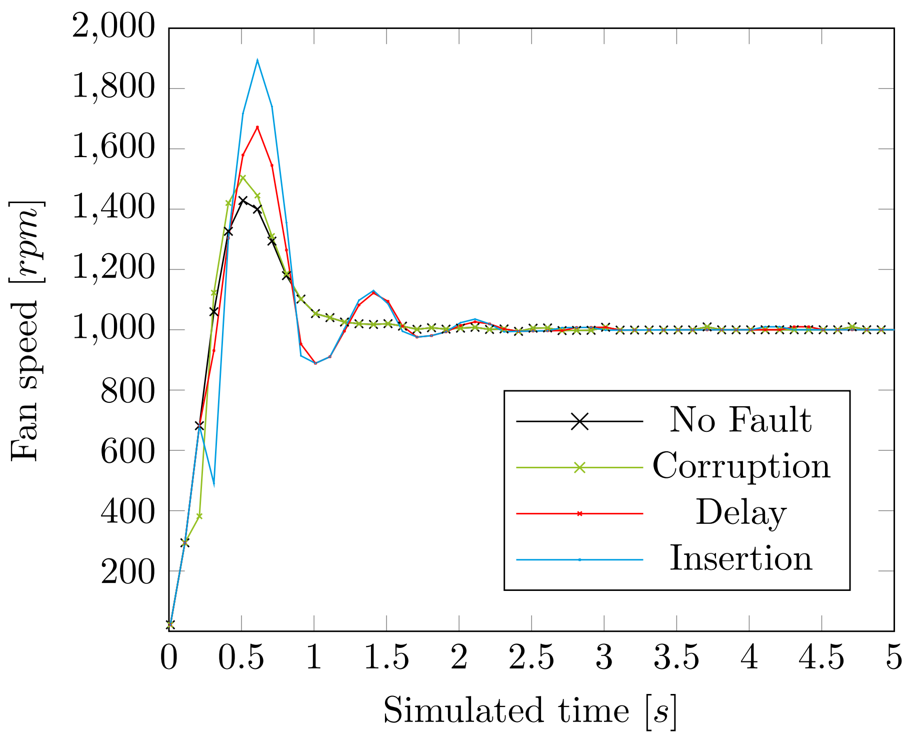


Figure 6.23.: HIL fan speed during normal execution, corruption, delay and insertion.

Figure 6.24 continues with the remaining errors. Again, the fault-free case is plotted additionally in black. Omitting the packet scheduled at 180ms shows a similar signal like its corruption (green graph). However, the difference to the speed signal without any error is much smaller. Changing the sequence of the PWM inputs results in an oscillation with several additional peaks. This case is shown in red. Finally, the signal in the replay case is not changed until the additional packet is inserted. According to the inserted value of 8, there is an additional peak in the speed signal at 3030ms (cyan graph). Afterwards, the signal recovers directly without further changes since the speed signal was stable already.

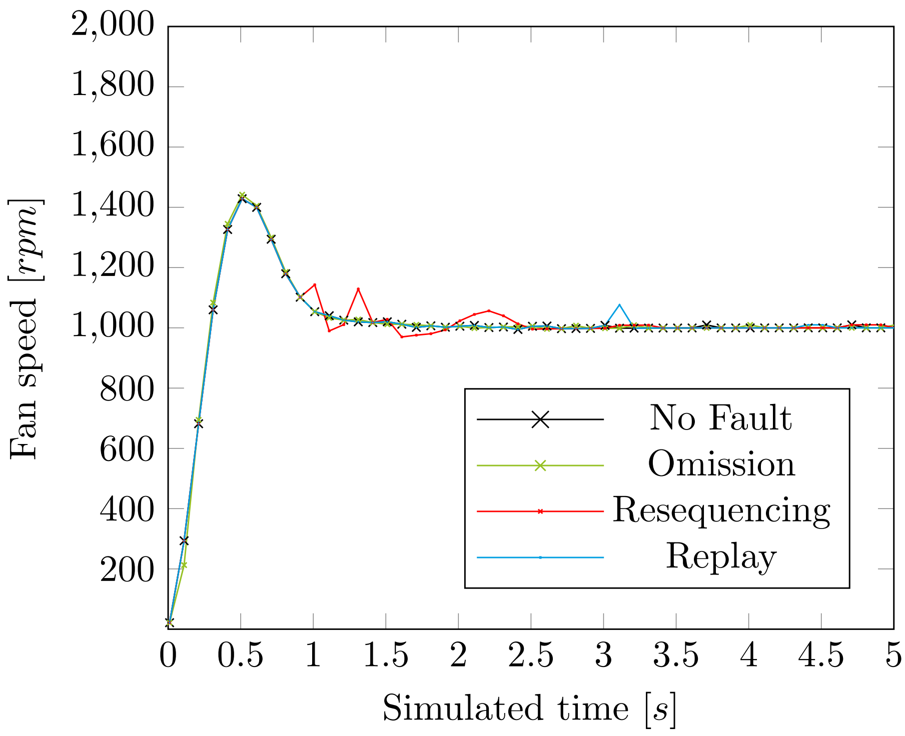


Figure 6.24.: HIL fan speed during normal execution, omission, resequencing and replay.

6.4. Framework Scalability

After demonstrating the fault-injection mechanism for SIL and HIL testing in the previous section, this section focuses on the scalability of the distributed co-simulation framework. To show this property, the setups introduced in Section 6.1.1 are executed on different topologies and the results are analyzed focusing on the following aspects. First, the number of messages exchanged during the synchronization mechanism of the RTI is determined and an equation to estimate this number is provided. Afterwards, the simulated times of the different setups are correlated to the runtime of the tests.

6.4.1. Synchronization Data Exchange

The simulation bridges in this framework are based on the *OpenRTI* implementation of the HLA. In this section, the most important message classes used by the synchronization algorithm are introduced. Furthermore an upper bound of the message numbers is defined based on the number of federates, the number of events to process every period, the number of data messages exchanged and the simulation duration. To determine this formula, the data exchange of the simulation bridges is analyzed using the RTI's source code and Wireshark. The message types are:

NMC	Next Message Commit
TAC	Time Advance Commit
CLBTSRM	Commit Lower Bound Time Stamp Response Message
LBNMR	Locked By Next Message Request

To advance in time, the simulation bridges send a *NextMessageRequest* which is represented by the first message type, *Next Message Commit*. It includes the time of the requested event and a commit ID which is used to identify the request. In addition to the next message commit, the simulation bridge sends a *Time Advance Commit* which contains the bridge's current logical time. Both messages are sent as broadcasts and the receiving bridges reply with a *Commit Lower Bound Time Stamp Response Message*. This unicast-response acknowledges the reception of the next message commit and its ID. In the following, the simulation bridges exchange time advance commits with increasing time-stamps. If a bridge received time advance commits from all other bridges with time-stamps larger than its requested time, a time advance is granted and a step can be computed. Afterwards, the output data is sent and the next event is requested using a further next message commit.

The time-stamps in the TAC message increase usually based on the lookahead defined. However, the simulation bridges also omit logical times if all requested times are larger (not smaller or equal) than the sum of the current time and the lookahead. This results in a significant speedup of the simulation duration which is exploited by the state-estimation mechanism. Figure 6.25 shows an example based on three components. The logical time of the simulation is depicted on the x-axis. While the NMC messages of the components are colored in red, the TAC messages are printed in blue. The figure illustrates the following situation. ED2 and the switch have sent a TMC with instant 5 to ED1 which performs the step for instant 4. After finishing the step, it sends the NMC for the next period and the TMCs for instants 5 and 6. The switch receives the messages before ED2 and also sends the TMC for 6. If ED2 receives the messages from both components before it replies, it is aware about the requested and the current times. According to the related guarantee of not receiving any data message, it can send a TMC for instant 9 without considering the intermediary instants. The same accounts for the other components wherefore a time advance is granted to the switch directly. Figure 6.25 shows that the TAC messages are sent for each time-stamp in the NMCs from each component. Furthermore, the messages are sent for at most the subsequent two instants after the requested times. Hence, the number of TAC messages can be bounded although an exact number depends on the order of the synchronization data exchange.

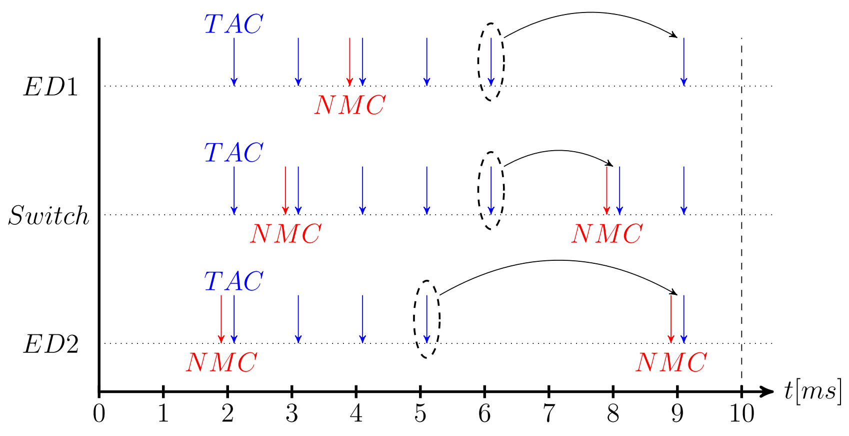


Figure 6.25.: Time advance in sparse schedule.

The locked by *NextMessageRequest* signals the waiting for a different commit ID. Whether it is sent or not depends on an alternating boolean. Its value is set to *false* if (I) the federate is not time constrained, (II) a *TimeAdvanceRequest* is used or (III) if the federate is waiting for a commit ID which differs from the currently received one. If none of these cases occurred, the boolean is set to *true*. A new commit ID is received in a commit lower bound time stamp response message after sending a next message commit. This update is idempotent. Similar to the TAC, the number of LBNM requests also depends on the order of the synchronization messages received. Hence, a determinate number cannot be given but it is possible to bound it as explained below.

Based on the above analysis, the following assumption can be made about the number of messages a federate sends. Herein, messages with multiple receivers are considered as one multicast message. The set $N_{e,c}$ represents the set of events e for which the component c sends an NMC message. Accumulating the amount of events for each component ($|N_{e,c}|$) results in the number of total events n_{TE}. Besides this, the set N_{DE} represents the disjoint set of all sets $N_{e,c}$ and can be used to determine the set N_{TAC}. The latter is constructed by taking all events $e \in N_{DE}$, adding the additional instants $(e+1)$ *mod* P and $(e+2)$ *mod* P (P is the period) and calculating the disjunction. To determine the number of CLBTSR messages (n_{CR}), the amount of $N_{e,c}$ must be subtracted from the number of total events. Finally, the number of LBNM requests (n_{LR}) is bounded by the number of disjoint events multiplied by two. Table 6.6 summarizes the sets and numbers based on Figure 6.25.

Table 6.6.: Numbers and sets of synchronization data exchange.

Value	Computation	ED1	Switch	ED2
$N_{e,c}$		$\{4\}$	$\{3,8\}$	$\{2,9\}$
n_{TE}	$\sum(\lvert N_{e,c}\rvert)$	5		
N_{DE}	$\bigcup N_{e,c}$	$\{2,3,4,8,9\}$		
N_{TAC}	$\bigcup_{e\in N_{e,c}} \{e,(e+1)\ mod\ P,(e+2)\ mod\ P\}$	$\{2,3,4,5,6,8,9,0,1\}$		
n_{CR}	$n_{TE}-\lvert N_{e,c}\rvert$	4	3	3
n_{LR}	$2\lvert N_{DE}\rvert$	10		

The total number of messages is the sum of the number of NMC, TAC, CLBTSRM, LBNMR and data messages per period as shown in Equation 6.8. The analysis shows that the number of messages per period depends mainly on the number of events in the setup. Table 6.7 varies this number according to the setups defined in Section 6.1.1. These theoretical results are compared to real tests in the following sections. While Section 6.4.2 varies the number of federates and the simulated time, a variation of the period length and the number of events in it is considered in Section 6.5.2. Since the simulation bridges are implemented to support a reaction on event-triggered messages, the lookahead remains equal with a value of $1ms$. This value is reflected further by the period of the deadline scheduler as explained in Section 6.2.1.

$$\begin{aligned}
\sum \frac{msgs_c}{P} &= |N_{e,c}| + |N_{TAC}| + n_{CR} + n_{LR} + n_{Data,c}\\
\sum msgs_c &= \frac{ST}{P}\left(|N_{e,c}| + |N_{TAC}| + n_{TE} - |N_{e,c}| + 2|N_{DE}| + n_{Data,c}\right) \qquad (6.8)\\
\sum msgs_c &= \frac{ST}{P}\left(|N_{TAC}| + n_{TE} + 2|N_{DE}| + n_{Data,c}\right)
\end{aligned}$$

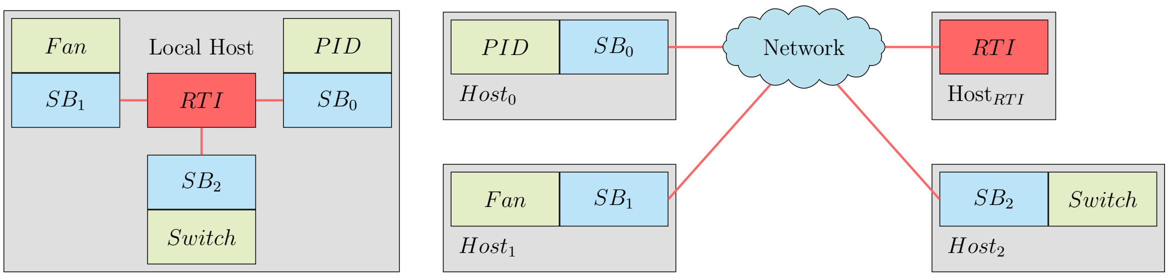
Table 6.7 denotes the number of events and messages for the different setups based on Equation 6.8. The numbers of the total events, the disjoint events and the TAC messages are equal for every component whereas the output data messages vary. All values are derived from the schedules introduced in Section 6.1.1. In Setups 2 to 5, there are multiple instances of the components. Since each instance sends the same number of messages, they are summarized in one line. The lines colored in gray denote the gradients the number of messages and components increase with in the different setups. Herein, the message gradient is calculated for a sequential and a parallel data transmission. All gradients are determined by dividing the values for the larger setups by the one of Setup 1. Besides that, the period is doubled in Setups 4 and 5 ($20ms$ instead of $10ms$) which is why the message gradients are divided by 2 in these cases. The gradients for a sequential transmission are much larger than those of the number of components but the absolute component numbers lie in the same order of magnitude. This property is shown in Figure B.1. Considering a fully parallel execution of the components, the gradients are smaller instead. This case is represented by the average number of messages of all components. Consequently, both gradients imply the scalability of the framework which is evaluated in detail in the following sections.

Table 6.7.: Number of events and messages per period in the different setups.

Component	Type	Setup 1	Setup 2	Setup 3	Setup 4	Setup 5
	n_{TE}	5	13	22	54	77
	$\|N_{DE}\|$	5	9	10	16	19
	$\|N_{TAC}\|$	7	10	10	18	20
	P	10	10	10	20	20
Fan_i	n_{Data}	1	1	1	1	1
	$msgs/P$	23	42	53	105	136
PID_i	n_{Data}	1	1	1	1	1
	$msgs/P$	23	42	53	105	136
$Voter_i$	n_{Data}	1	1	1	1	1
	$msgs/P$		42	53	105	136
$Switch1$	n_{Data}	3	5	7	8	15
	$msgs/P$	24	46	60	112	150
$Switch0/2$	n_{Data}				7	7
	$msgs/P$				111	142
$\sum msgs/P$		70	256	431	1804	3290
$Gradient_{msgs}$	$sequential$	1	3.657	6.157	12.886	23.500
$Gradient_{msgs}$	$parallel$	1	1.829	2.309	2.274	2.938
$N_{Components}$		3	6	8	17	24
$Gradient_{comp}$		1	2	2.667	5.667	8

6.4.2. Simulation Runtime and Simulated Time

To evaluate the scalability of the distributed co-simulation framework, the SUTs are allocated to the simulation hosts using three different topologies. These topologies are shown in Figure 6.26. The figure is based on Setup 1 which includes a fan, its PID controller and a switch connecting the components (cf. Section 6.1.1). In the local topology (cf. Figure 6.26a), all components and the RTI are executed on the same host and communicate via local TCP sockets. Topologies two and three are summarized in Figure 6.26b. The components are executed on different hosts and the communication is realized via a LAN or the Internet. In the latter case, a VPN is required further to protect confidential data. Both possibilities are represented by the *Network*-cloud. The other setups introduced in Section 6.1.1 are distributed in a similar way.



(a) Local setup.

(b) Communication via LAN or the Internet.

Figure 6.26.: Different topologies used during the evaluation based on Setup 1.

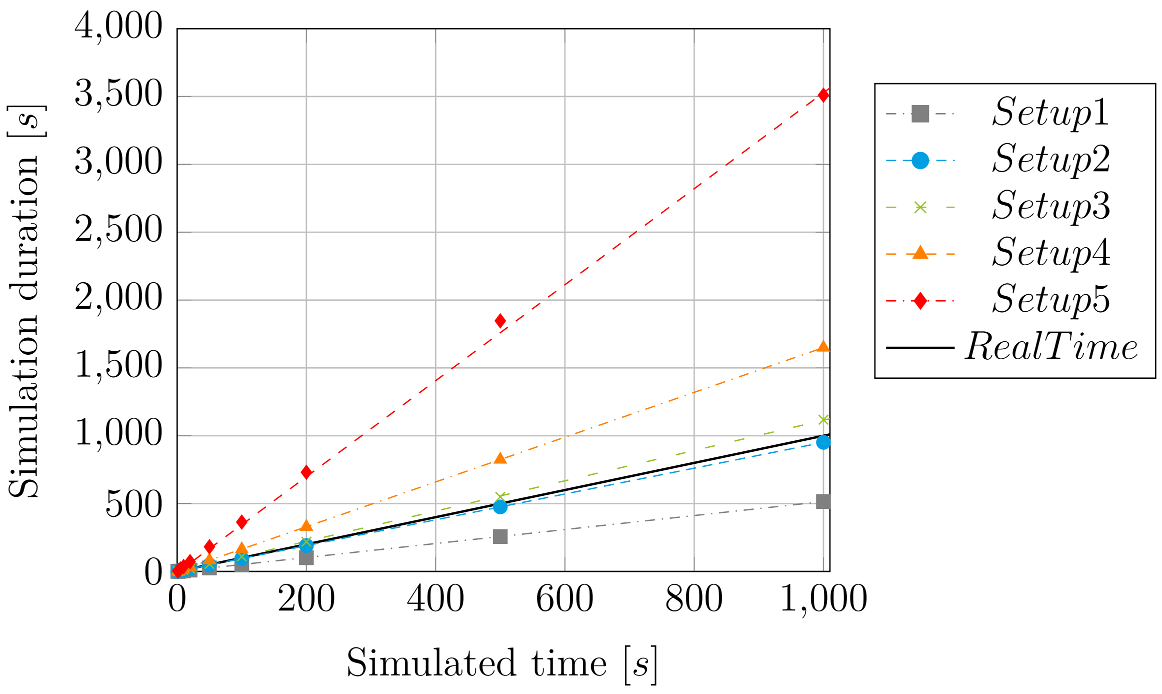


Figure 6.27.: Simulation durations for SIL in a local topology.

Using the different framework topologies, the simulated time is correlated to the time required to execute the test. In each test, ten simulation durations are chosen. Those are $1s$, $2s$, $5s$, $10s$, $20s$, $50s$, $100s$, $200s$, $500s$ and $1000s$. Figure 6.27 starts with the local setup where each case is executed 100 times. The simulated times are placed on the x-axis while the y-axis shows the average simulation duration of the setup. Both times are given in seconds and a black solid line represents the real-time case where the simulated time equals the simulation duration. For each setup, there is a linear correlation between the simulated time and the simulation duration which reflects Equation 6.8. The related regression slopes are analyzed later in this section. However, only the first two setups with three and six components can be executed in real-time. The average runtime of Setups 3, 4 and 5 with eight, 17 and 24 components is slower instead.

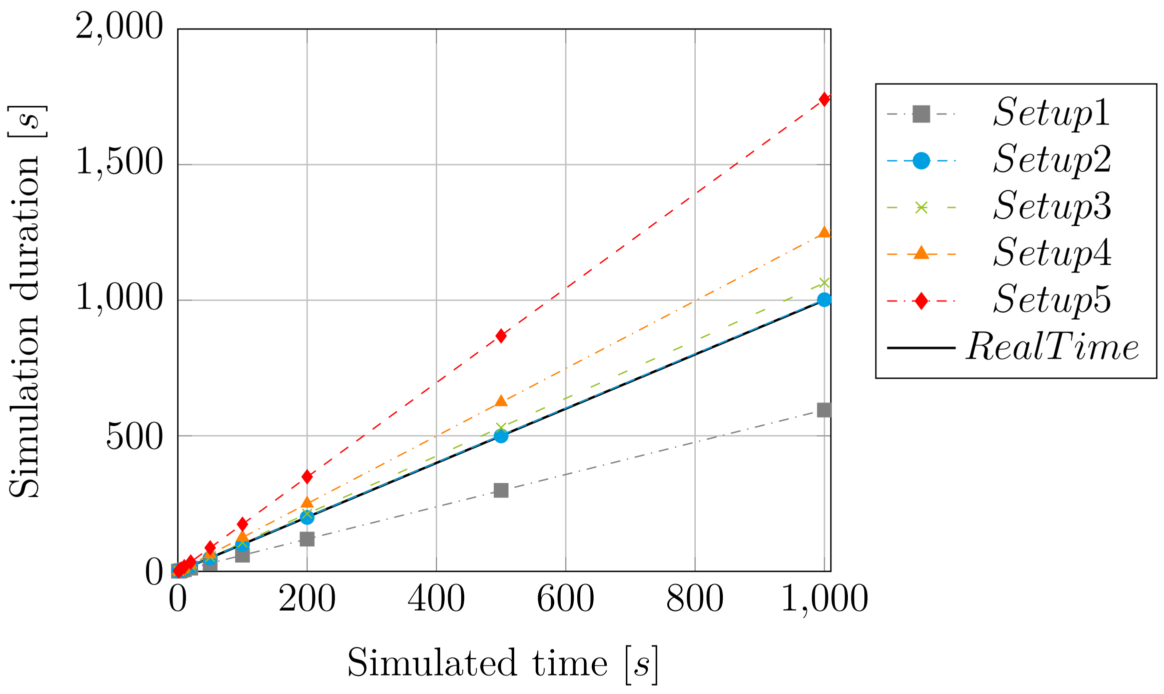


Figure 6.28.: Simulation durations for SIL in a LAN setup.

Figure 6.28 continues with the simulation durations of the LAN-topology for 50 runs. Although the communication delays are increased because of the network, the average durations of Setups 1, 2 and 3 are similar to those of the local topology. Besides this, the durations for Setups 4 and 5 are significantly smaller. The reason for this effect is the distribution of the setup across two simulation hosts and the execution of the RTI on a a third PC. As a consequence, more processes run in parallel. Considering Setup 5 as an example, the number of processes per host is reduced to twelve instead of 24. Another influence on the simulation execution is the kernel of the operating system. Whereas the simulation hosts use a real-time patched kernel, the kernel of the RTI PC runs the standard kernel. Since the latter is optimized for throughput and performance, the RTI's execution on this PC is faster than using the real-time kernel. This speeds up the execution further.

The influence of the communication delays on the simulation duration is shown if the Internet is used for the communication. The results of this topology are illustrated in Figure 6.29. In every setup, the simulation durations are much slower than in the previous topologies and far from the real-time case. For example, the durations for Setup 2 in the local topology are close to real-time whereas a simulation of 1000s via the Internet requires 41751s. However, the order of the graphs is similar to the number of components again and the durations increase linearly. The graph with the shortest duration refers to Setup 1, followed by Setups 2 and 3. According to the long simulation durations in this case, the tests for this topology are executed only 10 times. Furthermore, only the first three setups are considered.

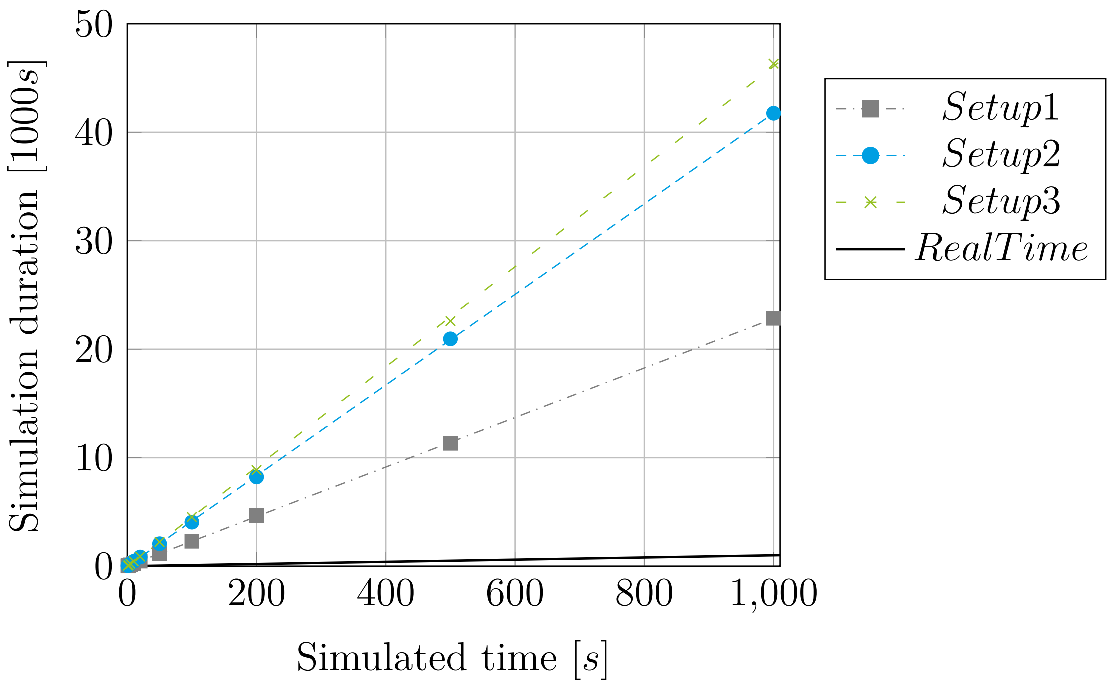


Figure 6.29.: Simulation durations for SIL in a setup with communication via the Internet.

Each of the previous figures (cf. Figures 6.27 to 6.29) shows a linear correlation between the simulated time and the simulation duration. The related regression slopes are depicted in Table 6.8 and correlated to the number of federations and messages per setup. Their numbers and gradients are printed in the first lines as introduced in the previous section. The subsequent lines of Table 6.8 denote the regressions slopes for each setup in the Local, LAN and Internet topologies and their rise compared to Setup 1. Starting with the local topology, the regression slope of Setup 1 accounts for 0.515. In the other setups, the slope is increased to values of 0.951 (Setup 2), 1.116 (Setup 3), 1.1650 (Setup 4) and 3.537 (Setup 5). The related gradients account for 1.847 (Setup 2), 2.167 (Setup 3), 3.204 (Setup 4) and 6.868 (Setup 5). Compared to the factors the number of components increases with, the slopes rise with smaller values. The same accounts for the message gradient of a sequential transmission. However, the parallel gradients are smaller for larger setups. Furthermore, there is no linear relationship as Setup 5 shows a slope which is 2.144 times larger than the one of Setup 4 (1.650 and 3.537). Considering the usage of a Dell PC with an Intel Core i7-8700 CPU (six CPU cores and twelve hardware threads), this rise can be explained with the number of concurrent processes that need to be executed. In Setup 5, six of the PID controllers and voters run in parallel, respectively. Each process further consists of three concurrent threads: a thread for the communication with the RTI, one for the ingress and one for the egress parts of the simulation bridges. In total, the number of threads exceeds the number of hardware threads wherefore the processes block each other. This causes increasing simulation durations in the larger setups.

Table 6.8.: Correlation between slopes of average runtime, number of components and number of messages.

Topology		Setup 1	Setup 2	Setup 3	Setup 4	Setup 5
Components	Number	3	6	8	17	24
	Gradient	1	2	2.667	5.667	8
Messages	$\sum msgs/P$	70	256	431	1804	3290
	Gradient seq.	1	3.657	6.157	12.886	23.500
	Gradient par.	1	1.829	2.309	2.274	2.938
Local	Regression Slope	0.515	0.951	1.116	1.650	3.537
	Gradient	1	1.847	2.167	3.204	6.868
LAN	Regression Slope	0.595	1.002	1.064	1.246	1.740
	Gradient	1	1.684	1.788	2.094	2.924
Internet	Regression Slope	22.820	41.781	46.169		
	Gradient	1	1.831	2.023		

In the LAN topology, the simulation durations are similar or shorter than on a single host as explained before. Starting with a slope of 0.595 in Setup 1, the values increase to 1.002 by a factor of 1.684 (Setup 2), to 1.064 (Setup 3, factor 1.788), to 1.246 (Setup 4, factor 2.094) and to 1.740 in Setup 5 (factor 2.924). This rise is smaller than the one of the components and also smaller than the message gradients for a parallel execution. As a consequence, a distribution via multiple PCs in a LAN provides scalable simulation durations even if the communication latencies are slightly longer than on a local host.

According to the long simulation durations when the Internet is used to communicate, the regression slopes are much larger than in the other topologies. Hence, only the first three setups are executed in this case. The related slopes of the regression lines account for 22.820 (Setup 1) 41.781 (Setup 2) and 46.169 (Setup3). The rise of these slopes is similar to those in the local setup which shows the main influence of the communication delays on the simulation duration. The number of messages exchanged during the synchronization process remains in the same order of magnitude even if the delays are increased.

The results presented in this section show the scalability of the framework in different use-cases. This covers a linear relationship between simulated time and runtime of the setups and proportional durations considering an increasing number of components. Furthermore, the upper bound of the number of messages provided by the previous section is validated by the results. In the next section, the state-estimation mechanism is evaluated with regard to real-time tests via the Internet and the achievable performance speedup.

6.5. Quality Improvement of Distributed Co-Simulation using State-Estimation

After analyzing the temporal characteristics of the distributed co-simulation framework, this section concentrates on the state-estimation mechanism as the first delay-management technique. On the one hand, it shows the improvements of using the mechanism for real-time tests via the Internet. Herein, the section covers the possibility to guarantee a timely packet reception and an improved simulation accuracy if intermediary packets are forwarded. On the other hand, a performance speedup in non-real-time tests is demonstrated by comparing the temporal characteristics of using different communication periods with those of Section 6.4.

6.5.1. Timely Packet Reception for Real-Time Devices

To evaluate the state-estimation mechanism, Setup 2 with a triplicated PID controller is used as described in Section 6.3.2. The DUT is distributed across six PCs and the fan is executed on real hardware. Since the device works in real-time, $Host_4$ has to be configured according to Section 6.2.2.

Before the benefits of using the state-estimation mechanism are shown, Figure 6.30 denotes the fan speed in a simulation run without any delay-management mechanism. The speed is depicted on the y-axis while the x-axis illustrates the simulated time. As the fan is a real-time device, the execution stops after the selected duration of $100s$. The sampling time is set to $100ms$ and each graph represents the result for a different communication period: $100ms$ (black), $200ms$ (green), $500ms$ (red) and $1000ms$ (cyan). To connect the simulation hosts, a VPN is established via the university's eduroam network. This results in network delays of approximately $8ms$ between the hosts and the RTI for each synchronization message. Since the state-estimation is disabled, these latencies lead to delayed packets for the communication periods of $200ms$ and $500ms$. As a consequence, the fan control is instable due to oscillations. If the communication period is set to

1000ms, there is enough time left in every period to receive all packets in time. Hence, there are no oscillations but the speed's accuracy is decreased compared to the black graph. In this graph, state-estimation is enabled to provide a reference signal.

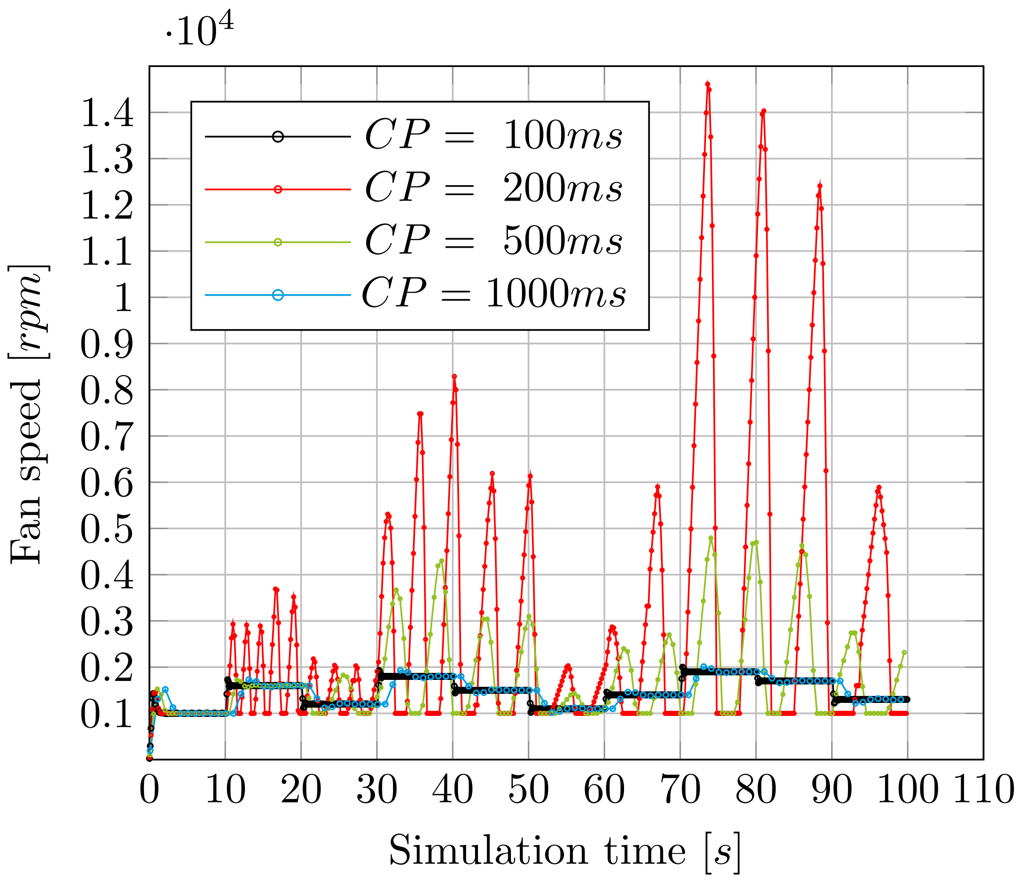


Figure 6.30.: Fan speed without state-estimation and increasing communication periods.

Figure 6.31 demonstrates the usage of the state-estimation mechanism to guarantee a timely packet reception. In this run, the control loop's parameters are equal and the same communication periods are used like before. Those periods represent setups in which the network delays lead to a decreased number of packets that can be transmitted in time. Even if packets are delayed in this use-case, the state-estimation can provide an estimated input to the fan wherefore there are no oscillations in the signal. However, its accuracy is decreased for longer periods. Since the time interval between subsequent packets is increased, the step response requires more time. As a consequence, the fan reaches its steady-state value later. As an example, the step from 1000rpm to 1600rpm finishes after approximately 7s if the communication period is set to 1000ms. In contrast, a period of 100ms reduces this time interval to 0.7s although a similar number of communication and computation cycles is executed.

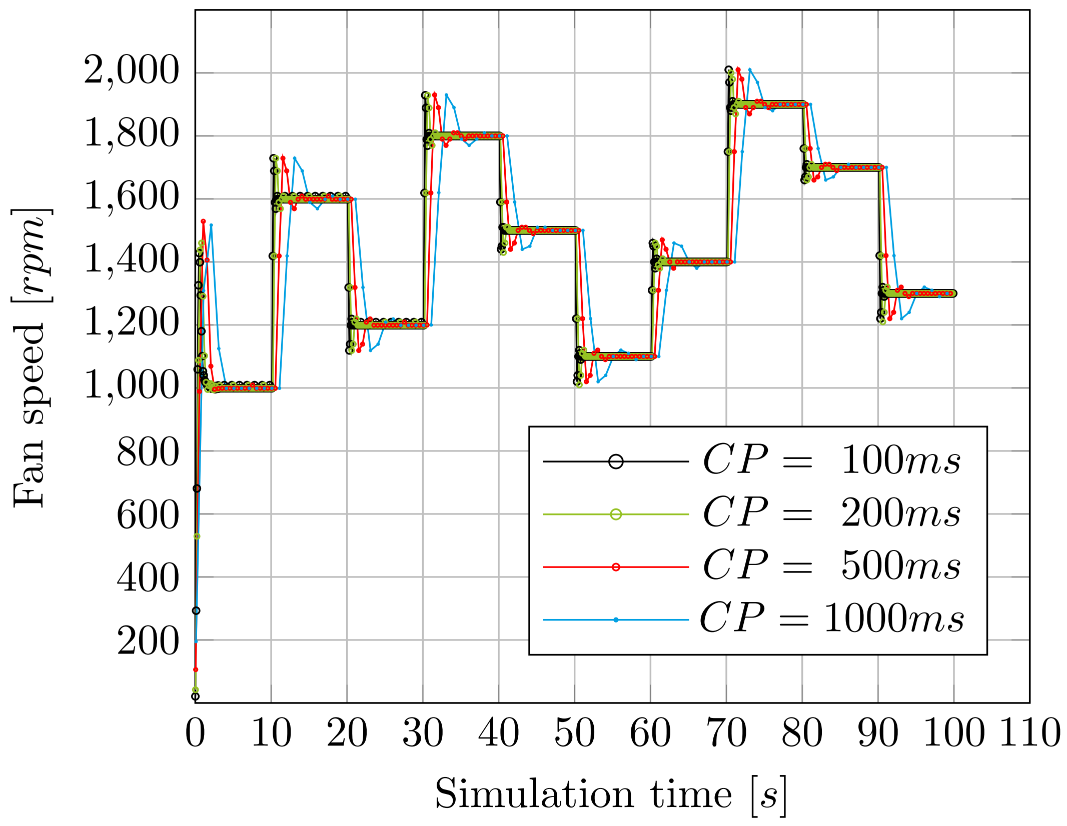


Figure 6.31.: State-estimation enabled to guarantee a timely packet reception.

Finally, Figure 6.32 illustrates the fan speed if the state-estimation mechanism is applied to forward intermediary packets between the received ones. Hence, the simulation bridge can forward an input to the device for every scheduled event independent from the network latencies. This results in similar speed signals for every communication period. As a consequence, the simulation accuracy is limited only by the system model's quality, the bridge's execution time and the communication delays between the bridge and the connected device.

To sum up, the results presented show the importance of using a delay-management mechanism for distributed real-time tests via the Internet. In a more realistic setup, the network delays are even longer than the $8ms$ measured for this topology. For example, pinging a DNS server located in New York, USA (e.g., 141.155.0.68) from Siegen requires approximately $100ms$. Furthermore, there can be more components connected to the federation which results in more messages exchanged between the simulation bridges and the RTI (cf. Section 6.4). The test results without state-estimation would be even worse in such a setup as the accuracy is deteriorated strongly.

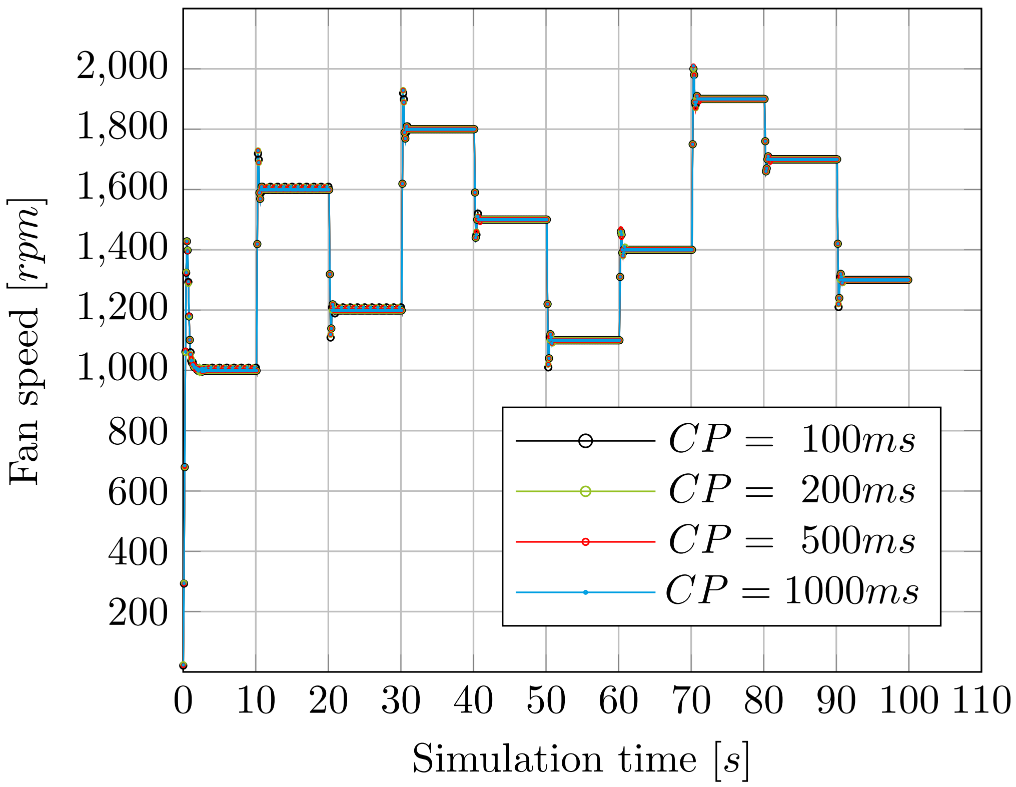


Figure 6.32.: State-estimation enabled to forward intermediary packets.

6.5.2. Performance Speedup using State-Estimation

Using longer periods for the data exchange between the simulation bridges, the state-estimation mechanism can mitigate Internet-introduced delays. In this way, it supports real-time tests via the network as shown in the previous section. These periods can be used further to speed up non-real-time simulations. Hence, this section focuses on the achievable speedup using different communication periods.

Figures 6.33 and 6.34 depict the simulation durations of Setup 1 in the local topology for different simulated times and communication periods. The durations for the simulated times from 1s to 100s are shown in Figure 6.33 while Figure 6.34 prints the remaining values for 200s to 1000s. Splitting the times into two figures improves the readability of the results. For the same reason, Figure 6.33 shows two different scales for the times of 1s to 5s and 10s to 100s, respectively. Both scales are separated using a dashed line. The durations for the different times and communication periods are shown as a bar chart. Herein, the gray bars represent the durations without state-estimation where the communication period corresponds to 10ms. In the black graphs, a period of 100ms is used while the periods of 200ms, 500ms and 1000ms are colored in green, red and cyan.

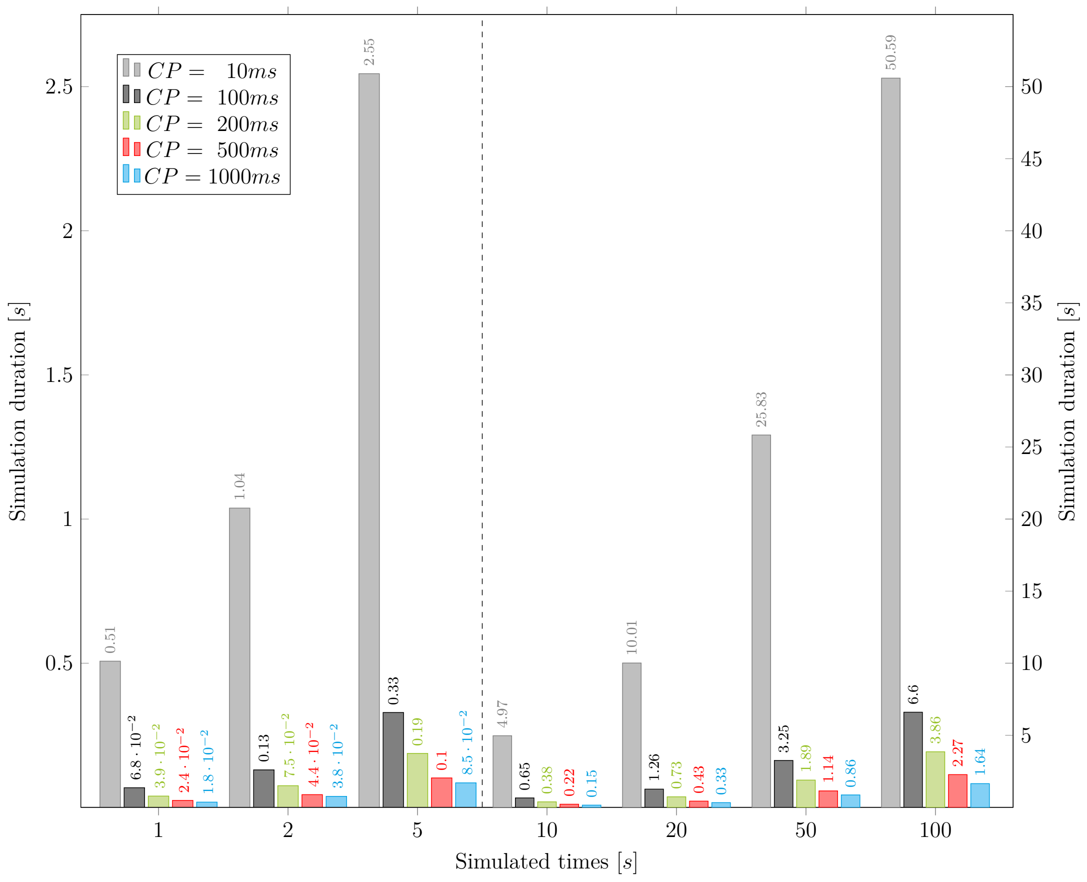


Figure 6.33.: Speedup using state-estimation for Setup 1 (1s to 100s).

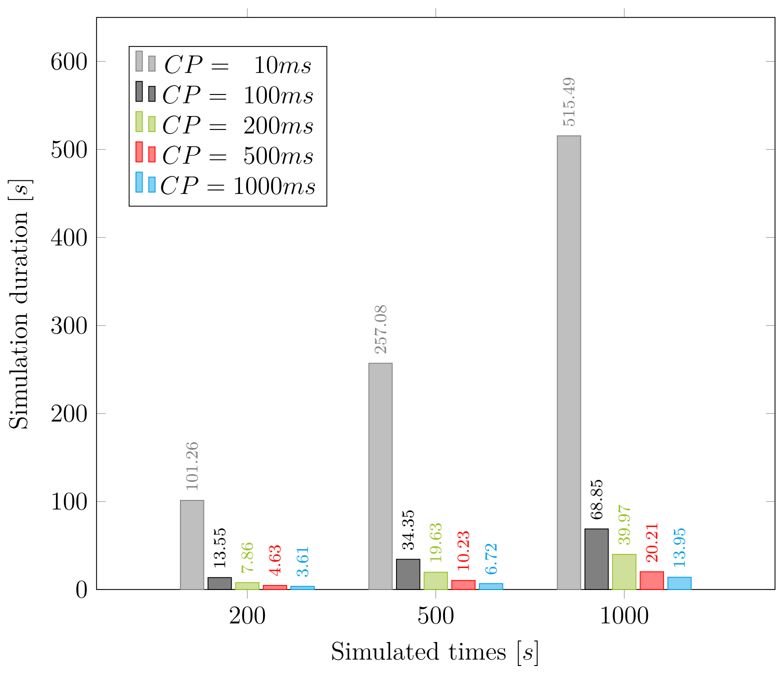


Figure 6.34.: Speedup using state-estimation for Setup 1 (200s to 1000s).

Both figures show a large speedup if the state-estimation mechanism is enabled. The largest gain can be reached with a communication period of $100ms$ compared to a period of $10ms$. Using longer periods, a speedup is still possible but the factor compared to the next smaller period is much smaller. For example, a simulated time of $100s$ is executed in $50.59s$. Using a communication period of $100ms$, the duration is reduced to $6.6s$ by a factor of 7.665. Scaling the period further to $1000ms$, the duration accounts to $1.64s$ which corresponds to a speedup of 4.024. These values reflect the approximation of Section 6.4.1 as the number of messages per period is independent from the period's length. This results in severe speedups for longer periods. In the following, the speedups for the different topologies and communication periods are analyzed in more detail.

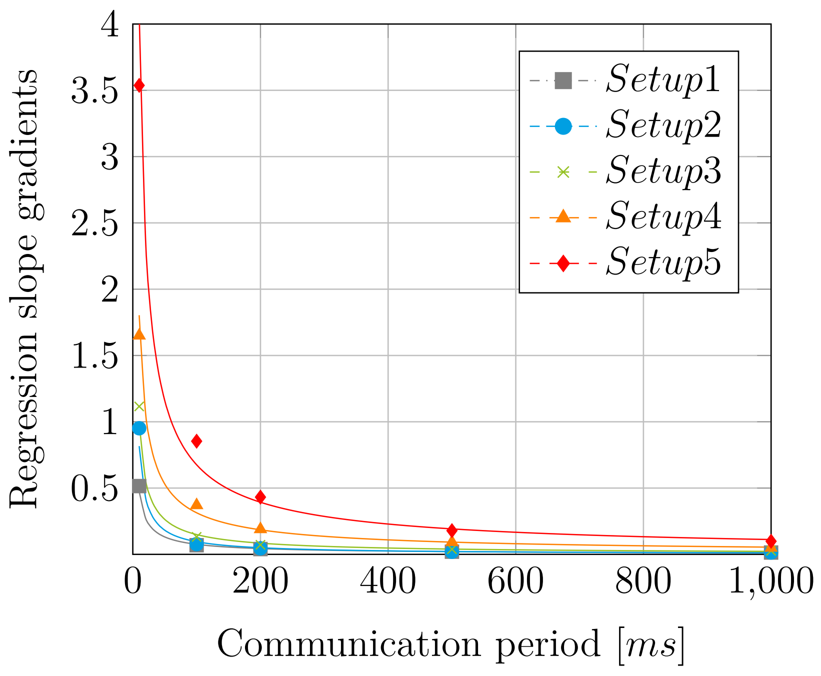


Figure 6.35.: Speedup using state-estimation in the local topology.

Starting with the local topology, Figure 6.35 illustrates the correlation between the communication periods and the regression slope gradients of the simulation durations. The periods are shown in ms on the x-axis of the chart while the gradients are printed on the y-axis. For each communication period and setup, the simulation durations are determined and a linear regression is calculated similar to Section 6.4.2. The resulting gradients are plotted against the communication periods and a regression is calculated. In contrast to the linear relationship between the simulated times and the time required to execute the test, there is a potential correlation in this case. The detailed parameters for the plots can be found in Appendix D including the equations for the regression graphs. Focusing on the local topology, the graphs for Setup 1 and Setup 2 are quite similar whereas the gap to the larger setups increases. This gap represents the increasing slopes shown in Figure

6.27. As implied by Figures 6.33 and 6.34, the most significant gain can be reached with communication periods between 10ms and 100ms for Setup 1 and Setup 2. For Setup 3, the gain can be extended to a period of 200ms whereas Setup 4 and Setup 5 show an approximating speedup for periods longer than 500ms. The reason for the reduced gains of longer periods is the increasing overhead of computing the intermediary packets.

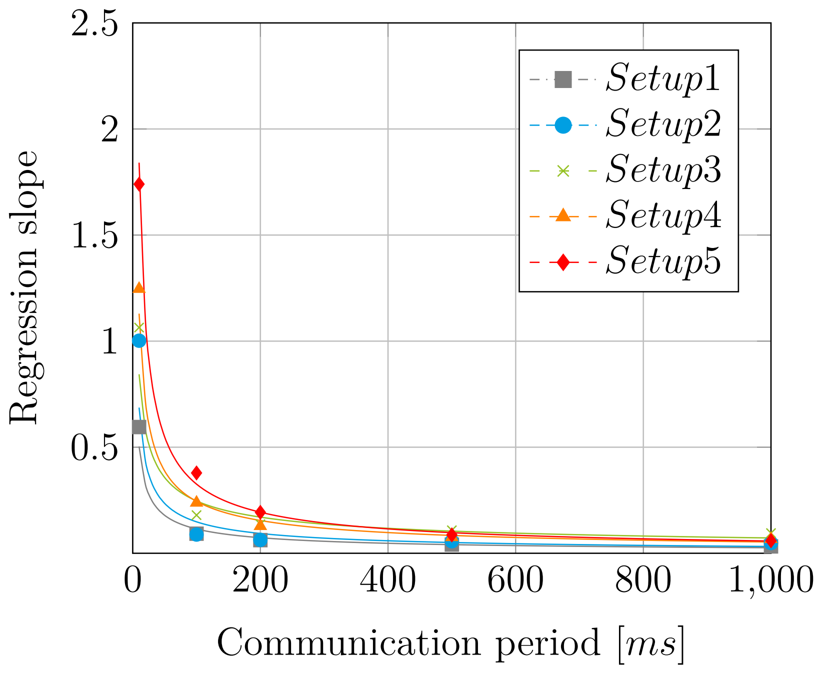


Figure 6.36.: Speedup using state-estimation in a LAN.

The speedups for the LAN topology are depicted in Figure 6.36. Although the potential relationship between the communication periods and the slopes remains, there are larger variations between the points and the regression graphs of Setup 2 and Setup 3. For the periods greater than 100ms, the coordinates of Setup 2 are similar to those of Setup 1 and Setup 3 conforms to Setup 4. Similar to the local topology, the largest gain for Setup 1 and Setup 2 can be found between periods of 10ms and 100ms. For Setup 3 and Setup 4, this bound can be moved to a period of 200ms whereas Setup 5 still benefits from a period of 500ms.

Finally, Figure 6.37 shows the results during the Internet tests. The graph of Setup 3 almost equals the one of Setup 2 and the distance between the graphs reflects the results of Figure 6.29. However, a communication period of 500ms can be used to achieve a perceptible speedup in all setups. This characteristic is contrary to the previous topologies but can be explained with the significant network delays the Internet introduces. For these tests, the delays of a ping via the Internet lie around 25ms whereas the LAN introduces delays less than 1ms. As a consequence, the reduced number of messages exchanged has a large influence on the execution's performance.

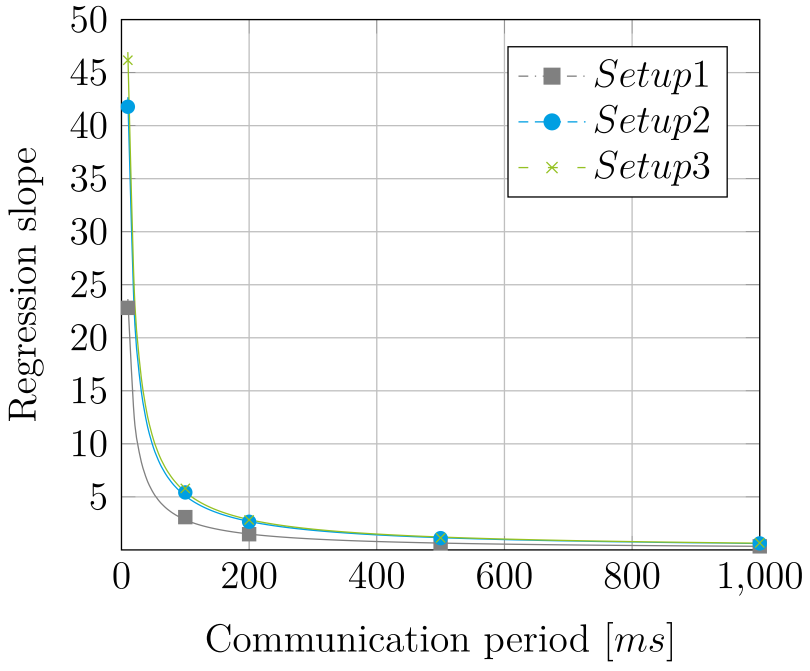


Figure 6.37.: Speedup using state-estimation via the Internet.

The majority of the presented results shows speedups up to a communication period of 500ms. Hence, Table 6.9 summarizes the related speedups for the different topologies compared to a period of 10ms. Since there are considerable variations between the values, a general conclusion cannot be drawn. However, a larger number of components also results in larger speedups in most cases. In the local topology, the speedups range from 17.779 to 28.755 with an outlier in Setup 2 (49.280). Regarding the LAN, there are variations between 14.548 to 20.069. Again, there is an outlier with Setup 3 (9.815) where the speedup is much smaller than in the other cases. These outliers are not available in the Internet tests where the results range from 36.425 to 40.620. Considering the median values, speedups of up to 25 (Local), 16 (LAN) and 37 (Internet) are achievable.

Table 6.9.: Achievable speedups for a communication period of 500ms.

Setup	Local	LAN	Internet
Setup 1	25.495	14.548	36.425
Setup 2	49.280	17.641	37.668
Setup 3	28.755	9.815	40.620
Setup 4	17.779	16.438	
Setup 5	19.771	20.069	

On the opposite side, these significant speedups reflect the influence of an increasing number of events and messages on the simulation durations. While there is a linear relationship between the durations, the simulated times and the number of federates per federation, the durations grow exponentially for large numbers of events. However, periods of $10ms$ are mapped to 10 ticks of the RTI and the deadline scheduler requires periods of $1ms$ to ensure a timely execution (cf. Section 6.2.2) As long as the periods of the SUT are in this order of magnitude, reasonable results can be achieved using the distributed co-simulation framework.

6.6. Performance Speedup using Speculative Execution

The concept of the speculative execution is the distribution of the setup into independent federations which run in parallel. This section focuses on the achievable speedup of the mechanism which does not require additional estimation models.

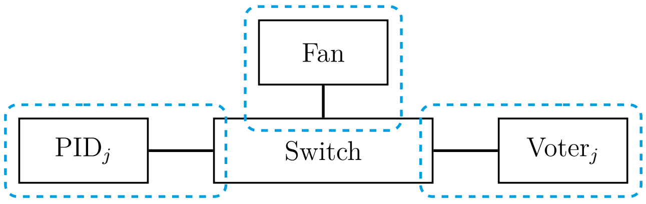


Figure 6.38.: Subsets of Setup 3.

During these tests, the setups are split into federations according to their sizes and topologies. To forward data between the federations, the switches are connected to simulation gateways. Figure 6.38 illustrates the federations for Setup 3 where the triplicated components are summarized by one box and an index j. Since there are only two components available in Setup 1 (fan and PID controller), this setup is distributed into two federations and the voters are removed. The other two setups are divided into three federations which include the fan, the PID controllers and the voters. In the figure, the federations are shown using cyan-colored dashed lines.

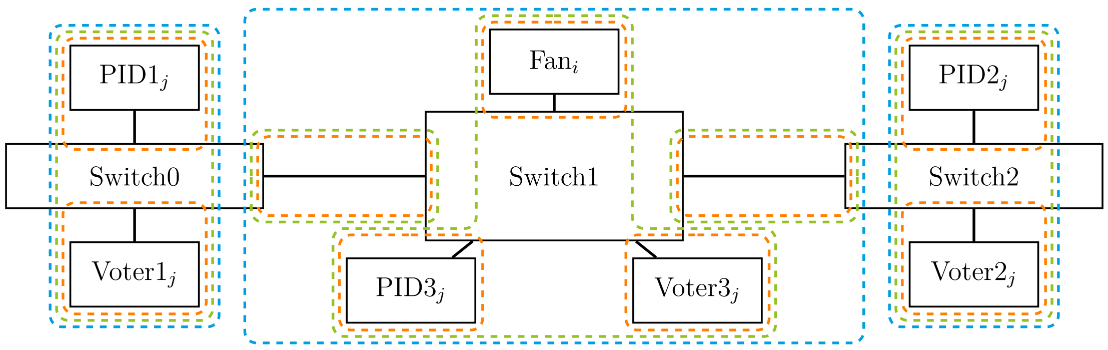


Figure 6.39.: Subsets of Setup 5.

Figure 6.39 covers the federations of Setup 5 which can be divided using three cases. The first case covers the three federations colored in cyan. Each federation includes the components connected to the three switches while the bridges of *Switch0* and *Switch2* operate as simulation gateways. The green colored federations cover the second case with five subsets. Herein, *Switch1* is connected to a third simulation gateway and the links between the switches are moved into dedicated federations. These dedicated federations remain in the final case with nine federations which is colored in orange. The other federations are split further so that each group of similar components (fans, PID controllers and voters) are located in separate sets. Since the PID controllers and voters of the third fan-control application are not available in Setup 4, these federations are omitted resulting in seven subsets. The remaining cases are similar to Setup 5.

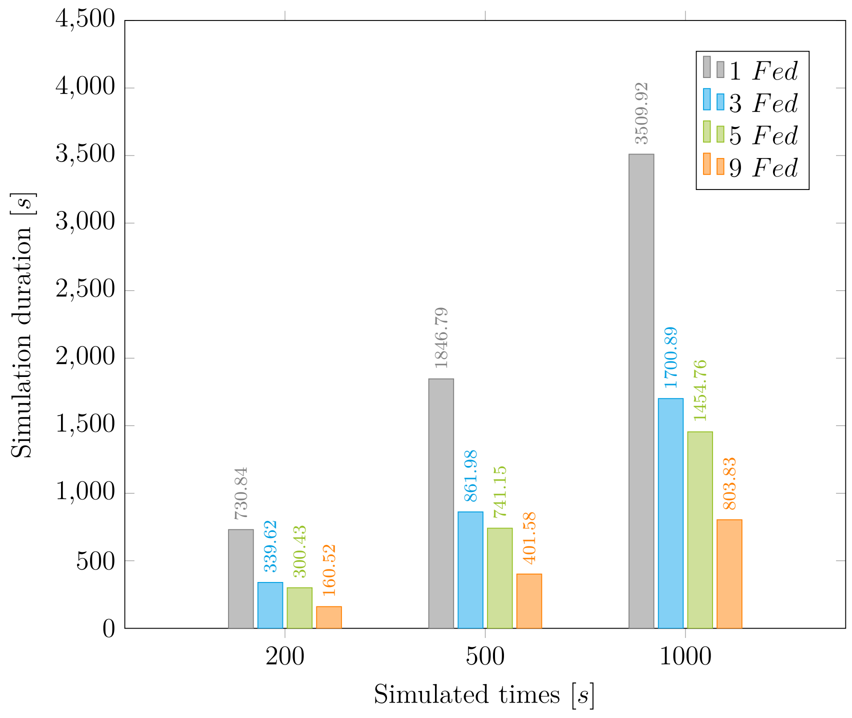


Figure 6.40.: Speedup for Setup 5 using speculative execution (200s to 1000s).

Since Setup 5 supports the largest number of different federations, its local execution is selected to demonstrate the achievable speedup using the speculative execution. The different simulation durations for each case are shown in Figure 6.40 and Figure 6.41 including the case without any distribution. This case is shown as a gray bar. The other bars are colored with regard to Figure 6.39. While the simulated times are plotted on the x-axis, the y-axis denotes the simulation durations. To improve the readability, Figure 6.41 uses two different scales for the simulated times, one for 1s to 5s and one for 10s to 100s. The results show a speedup of more than two if the Setup is distributed via three federations (cyan) compared to the initial case. Similar results can be seen for the cases with five and nine federations whereas the difference between three and five federations is quite small.

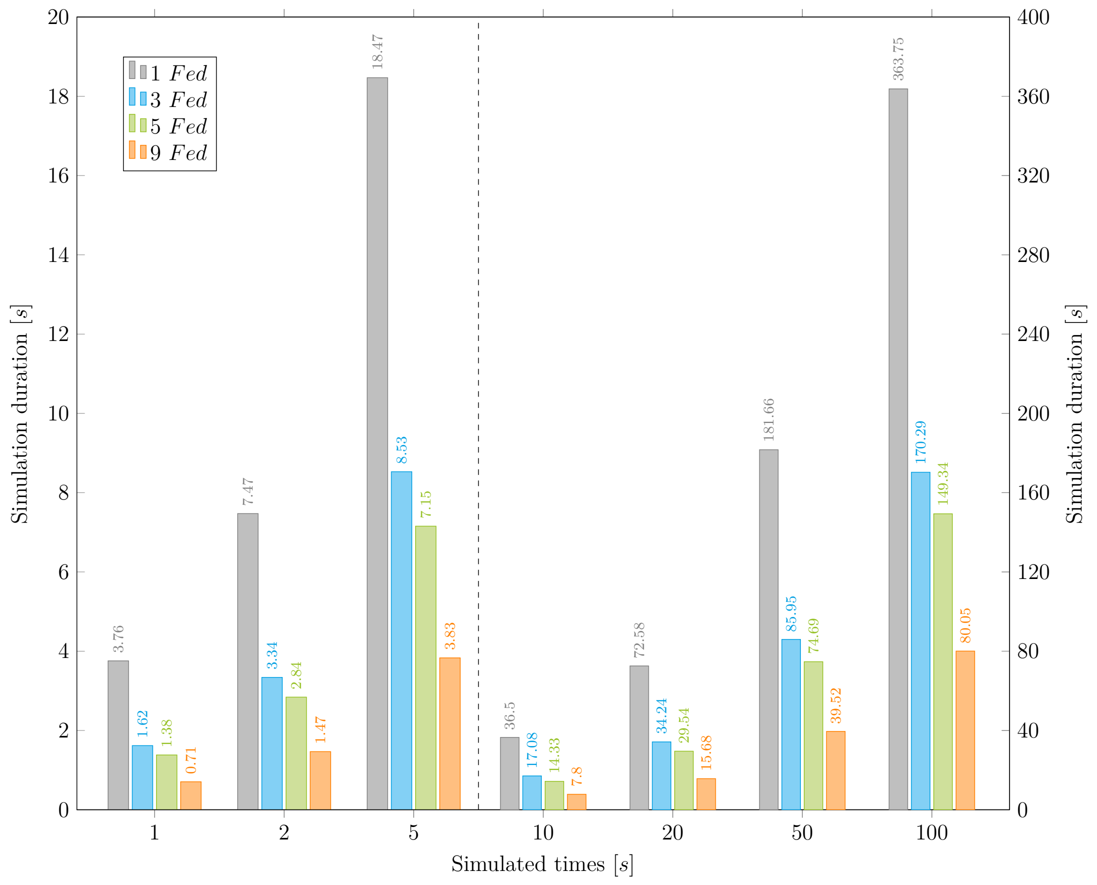


Figure 6.41.: Speedup for Setup 5 using speculative execution (1s to 100s).

Detailed speedup values for all setups in the local and LAN topologies are depicted in Table 6.10. For Setup 1 and Setup 2, the speedups on a local host are smaller than in the LAN whereas the results are reverse for Setup 3 to Setup 5. The largest gain in the local topology can be reached in Setup 5 (4.399). Here, the reduced number of components per federation leads to a significant reduction of the number of synchronization messages exchanged. As a consequence, the simulation can be executed faster. In Setup 1, this effect is minimal since there are only two federations with two components and both federations must be synchronized. This results in a speedup of 1.077. As shown in Figure 6.40 and Figure 6.41, a similar effect can be seen between the usage of three and five federations in Setup 4 and Setup 5. In these cases, the links between the switches are moved to separate federations, but the number of messages exchanged is small. Hence, a large speedup is not achievable.

Using a LAN to distribute the components, the slopes and the resulting speedups differ between the setups. While the slopes of Setup 2, Setup 3 and Setup 4 (three and five federations) are similar to those of the local tests, using seven federations in Setup 4 results in a larger slope. For Setup 1 and Setup 5, the slopes are smaller instead. These results can be explained again with the distribution of the components between the hosts and the resulting faster execution (cf. Section 6.4.2). Herein, Setup 4 must be contemplated as an outlier. The distribution is also the reason for the smaller speedups compared to a local execution since the reference results for one federation are smaller already. Apart from that, smaller federations still perform faster than larger ones. This can be seen in the results of Setup 4 and Setup 5, but also for Setup 1 and Setup 2.

Table 6.10.: Speedups for local and LAN topologies.

		Local		LAN	
Setup	Federations	Slope	Speedup	Slope	Speedup
Setup 1	1	0.515	1	0.595	1
	2	0.478	1.077	0.394	1.510
Setup 2	1	0.951	1	1.002	1
	3	0.651	1.461	0.620	1.616
Setup 3	1	1.116	1	1.064	1
	3	0.720	1.550	0.745	1.428
Setup 4	1	1.650	1	1.246	1
	3	0.701	2.354	0.702	1.775
	5	0.627	2.632	0.686	1.816
	7	0.430	3.837	0.542	2.299
Setup 5	1	3.537	1	1.740	1
	3	1.705	2.074	1.023	1.701
	5	1.459	2.424	0.804	2.164
	9	0.804	4.399	0.564	3.085

In real-time tests, the simulated time equals the simulation duration resulting in a slope of 1. To support such tests, the slope of the simulation durations must be smaller than this value, otherwise messages are received too late. Using the speculative execution on a local host, almost all cases show slopes smaller than this value. The only exceptions are the cases with three and five federations of Setup 5. Distributing the setups via a LAN, the slopes are also smaller if Setup 5 is distributed via five federations. These examples show the possibility to use the speculative execution for real-time tests if a proper configuration can be found. This configuration depends on the size of the test, the independence of the different components and the delays of the network connecting the simulation bridges.

To speedup the simulation further, the RTI can be located close to the components with the largest amount of traffic. While the influence of the location on the duration is low for small network delays, the tests via the Internet show larger differences. In Table 6.11, the column *Local RTI* denotes the cases where the RTI instances are placed on the same hosts like the fan, PID and voter components. The other column *Remote RTI* represents the runs where the RTI instances are executed on the network host. Setup 1 shows the case where only two simulation bridges are connected to the same federation. In this case, both bridges exchange the same number of messages wherefore there is no difference between the slopes. The speedups for this setup lie around a value of 2.2 while the other setups achieve larger speedups for the local case. Here, the values are almost similar with 2.359 (Setup 2) and 2.432 (Setup 3). If the RTIs are located at the network simulation, more message experience a longer network delay which results in a speedup of 1.865 for Setup 2. For the same reason, the speedup is reduced to a value of 1.554 in Setup 3. Here, the number of remote components is even increased compared to Setup 2.

Table 6.11.: Speedups for distributed simulations via the Internet.

		Local RTI		Remote RTI	
Setup	Federations	Slope	Speedup	Slope	Speedup
Setup 1	1	22.820	1	22.820	1
	3	10.208	2.236	10.420	2.190
Setup 2	1	41.781	1	41.781	1
	3	17.713	2.359	22.399	1.865
Setup 3	1	46.169	1	46.169	1
	3	18.987	2.432	29.702	1.554

All in all, the results for the Internet topology strengthen the characteristics determined for the local topology and the LAN. During the synchronization algorithm, the location of the RTI plays a major role for the performance of the execution. Hence, it is important to determine the number of messages exchanged when configuring the speculative execution. As shown before, a valid upper bound is given by Equation 6.8 introduced in Section 6.4.1. A real-time execution is not possible in this case and longer communication periods than the selected $10ms$ are still required.

7. Conclusion and Future Work

7.1. Summary and Contribution

Aircrafts, trains and cars are common example for today's large-scale distributed real-time systems. These systems are composed of several components which are developed and integrated within an incremental process. In this process, validation and verification are important steps, especially if the system under test is safety-critical. However, the typical inclusion of several specialized but geographically distributed manufacturers aggravates the development. Components must be shipped to a central place for the integration and validation procedure and intellectual property must be protected [HLV06].

Distributed co-simulation, Software- and Hardware-In-The-Loop (SIL, HIL) testing can be used to simplify the process. They are able to couple simulation tools, software-implemented control algorithm and real hardware devices via heterogeneous communication networks such as LANs or the Internet. Another widely used validation technique is fault-injection which allows to investigate the system behavior in case of faults. A distributed co-simulation framework supporting these mechanisms enables the integration and testing of distributed components at early development steps. However, HIL testing involves hardware components with real-time requirements. The indeterministic network delays introduced by the Internet have an adverse effect on the quality of the tests as deadlines might be missed. As a consequence, mechanisms must be included into the distributed co-simulation framework which manage these delays.

This thesis proposes a framework which enables distributed co-simulation, SIL and HIL testing via heterogeneous communication networks (cf. Chapter 4). It operates on a network-centric abstraction level and provides a generic interface to support a large variety of simulation tools, software applications and real hardware. Using the HLA simulation standard, the framework provides a synchronization mechanism which coordinates the time advance between non-real-time components. An additional mechanism synchronizes the advance of the logical simulation time with the physical time of real hardware devices. Besides this, the framework supports the injection of faults into the communication of the

SUT. This mechanism covers the error modes introduced in Section 3.6 which are based on IEC 61508-2 clause 7.4.11.

To support real-time HIL testing via the Internet, two delay-management technologies based on state-estimation and speculative execution are provided (cf. Chapter 5). The state-estimation mechanism can be used (I) to guarantee a timely forwarding of received or estimated data to the devices (cf. Section 5.1.2) and (II) to reduce the RTI message exchange by extending the communication periods and forwarding intermediary, estimated packets (cf. Section 5.1.3). Alternatively, the speculative execution distributes the SUT into independent subsets (cf. Section 5.2). These subsets are executed in parallel as separate federations and data is routed via gateways. In this way, real-time data can be provided in time and the synchronization effort of the RTI can be reduced. Both mechanisms exploit the available knowledge of time-triggered schedules where the instants of all communication activities are a-priori known [EBK03].

The framework is evaluated based on a distributed, fault-tolerant fan-control application. This application consists of four types of subsystems which are combined in five setups (cf. Section 6.1). The framework itself is implemented as a proof-of-concept for the Linux operating system. Using its deadline scheduler and the locking of memory in the RAM, the simulation hosts can be configured to schedule the framework in real-time (cf. Section 6.2.2). Furthermore, the usage of a TSN driver and a suitable network interface card enables a timely communication with the device. These technologies are used in Section 6.3 to validate the fault-injection mechanism for HIL and SIL. During these tests, faults covering all error modes of CENELEC EN 50159 are injected into the destination speed messages sent from the fan (SIL) and into the PWM values this component receives (HIL). The manipulations of the resulting fan speed are shown by plotting the signal and using Wireshark packet captures of the data transfer.

To demonstrate the scalability of the framework, the five fan-control setups consist of an increasing number of components. Furthermore, multiple simulation durations from $1s$ to $1000s$ are defined and the framework is executed in different topologies: locally, in a LAN and via the Internet. The results presented in Section 6.4 show a linear relationship between the simulated time and the resulting simulation duration. This accounts for all topologies and the longer durations of the Internet tests arise from the increasing network delays. Another scalability parameter is the number of components in a setup. Comparing the slopes of the simulation durations with the setup's scaling factor, the factor represents an upper bound for the durations' slopes. Hence, the synchronization mechanism also scales for larger setups. This is reflected further by Equation 6.8 which determines the number of messages exchanged via the RTI (cf. Section 6.4.1). In this equation, all parameters are linear.

Finally, the delay-management mechanisms are evaluated with regard to the capability of supporting real-time tests in heterogeneous communication networks. The state-estimation mechanism can be used to provide received or estimated inputs to the device in time independent from the network delays. Furthermore, the forwarding of intermediary packets can increase the accuracy of the tests to be limited only by three aspects. Those are (I) the system model's quality, (II) the computation times of the simulation bridges and (III) the communication delays between a bridge and the connected device (cf. Section 6.5.1). As a disadvantage, the creation of a valid model might be challenging.

The speculative execution represents an alternative mechanism. As it is based on smaller federations with a downscaled RTI message exchange, an additional estimation model is not required. However, the results of Section 6.6 show the mechanism's dependence on the communication network and the schedule of the SUT to support a real-time execution. For example, an application with periods of $10ms$ cannot be executed in real-time if the simulation bridges communicate via the Internet. Using a LAN or local host instead, the achievable speedup is sufficient to run real-time tests with some exceptions in the larger setups.

From a performance point of view, the state-estimation provides significantly better results compared to the speculative execution. The resulting speedup depends on the setup, the topology and the communication period (cf. Section 6.5.2). For example, a value of 9 is the minimum speedup for periods of $500ms$. To select a valid communication period, a trade-off must be made between the simulation's accuracy and the speedup required. Considering all topologies and setups using the speculative execution, the speedup reaches a value of at most 4.4 (cf. Section 6.6). All in all, both mechanisms are able to enhance the quality and the performance of the tests. Detailed improvements depend on the SUT and its schedule in both cases.

7.2. Future Work

The scope of this thesis focuses on time-triggered distributed real-time systems which communicate via the Ethernet protocol and its extensions such as TSN or TTEthernet. Hence, there are different topics that could be addressed in future research projects. These topics are introduced briefly in this section.

Alternative protocols

According to the current interest to support real-time communication via Ethernet, this protocol and its extensions were selected for this thesis. However, there are several alternative technologies used in distributed (real-time) systems. Examples are the CAN [iso15] or FlexRay [iso13] communication systems from the automotive domain, Wireless LAN technologies from the IEEE 802.11 standard family [iee16] or Wireless Personal Area Networks defined in 802.15 [oEEE] (e.g., Bluetooth, ZigBee). Supporting additional protocols, the application field of the distributed co-simulation framework can be extended to further domains such as industrial or medical systems.

Support for Rate-Constrained and Best Effort traffic in state-estimation and speculative execution

Currently, the state-estimation and speculative execution technologies are based on the a-priori knowledge of time-triggered real-time schedules. However, protocols such as TTEthernet support additional traffic types like rate-constrained and best-effort traffic. Although both traffic types are supported by the distributed co-simulation framework, the delay-management mechanisms can only recognize large delays. This is caused by the uncertain transmission times. In future work, an algorithm can be developed which predicts these times using machine learning and artificial intelligence. Such an algorithm enables the support for state-estimation at runtime and could be used at design time to configure the federations for the speculative execution.

Adaptation and dynamic creation of system model for state-estimation

As shown in this thesis, the state-estimation mechanism is a valid technique to support distributed real-time tests via the Internet including real hardware. Furthermore, a significant speedup can be reached if only non-real-time components are involved. However, its main drawback is the creation of a valid system model. If the model is inaccurate, the difference between the estimated and received inputs can become too large and the simulation has to stop. Using a rollback mechanism could bring the system model and the outputs back into a valid range, but this is not applicable for real hardware. Those devices run in parallel to the simulation bridges and they typically interact with their environment. Rolling back the environment is difficult or even impossible wherefore it is not supported by the current version of the state-estimation. Besides that, the system model needs to be adaptable, otherwise the same error will occur later in the simulation. In the future, artificial intelligence might be used to adapt the system model at runtime before the error occurs. Furthermore, it might be used during the execution of co-simulation setups to create a valid system model for later development stages.

Speculative execution using optimistic synchronization without rollback

In addition to the time-management algorithm used for this work, the HLA supports another algorithm based on optimistic synchronization. This synchronization mechanism allows to request the transmission of future interactions with the risk of missing messages that are not available yet in the RTI. Exploiting the knowledge of a time-triggered schedule, the simulation bridges can use these services to advance in time even if they are constrained by other, independent components. The available knowledge about the schedule can also prevent the necessity of a rollback mechanism. Such a mechanism is required predominantly by works of the current state-of-the-art using optimistic synchronization.

Further optimization

Finally, there are some aspects which concern the implementation of the framework. These aspects cover the support for additional operating systems such as Windows or the implementation on real hardware. Using real hardware, the simulation bridge could be connected to a device similar to a switch in a network. This reduces the delays introduced by the simulation host and solves issues with real-time capabilities of general purpose PCs. Besides that, there are several commercial alternatives to the OpenRTI which is used for this work. Examples are the MAK-RTI or the Pitch pRTI. While the MAK RTI promises a high-performance solution for small and large federations [VT], the Pitch pRTI is optimized for a local, cloud-based or virtual deployment [Tec]. In future work, those alternatives should be evaluated according to their usage in several other projects.

A. Configuration Model

This annex introduces the configuration model of the distributed co-simulation framework. The element *DCSF* is the origin and must contain four sub-elements: *Federations*, *Federates*, *SynchronizationPoints* and *SimulationExecution* (cf. Figure A.1). While the latter defines only the start and stop times of the simulation execution, the other elements have more attributes. These attributes are introduced in the following.

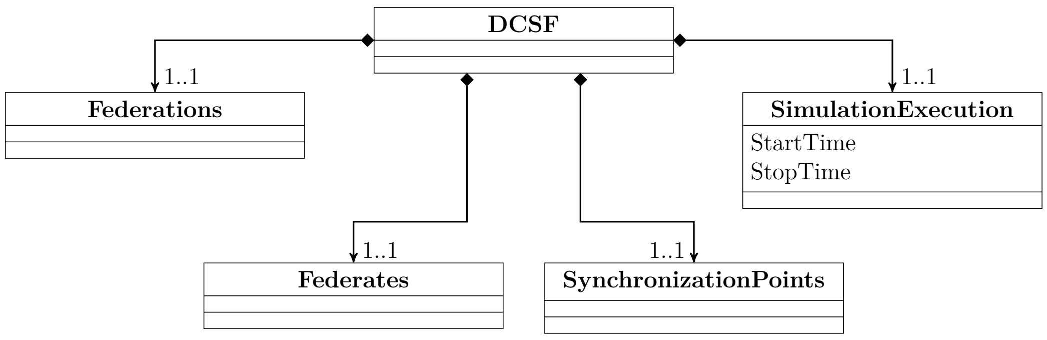


Figure A.1.: Main elements of the configuration file.

In a simulation setup, there might be multiple federations representing a subset of the setup. The related attributes are depicted in Figure A.2.

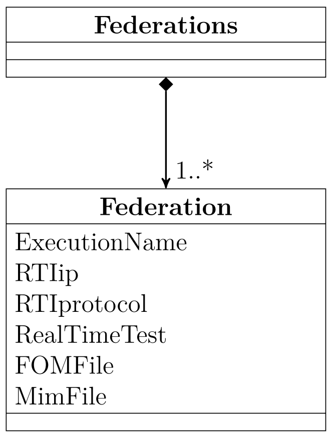


Figure A.2.: Attributes of the federations.

The attributes of the different federates (simulation bridges or gateways) in a simulation setup are shown in Figure A.3. Each federate can be located in one or more federations publishing or subscribing to multiple interactions. For each of these federations, a task and message schedule is defined. Besides this, the federate includes one or more wrapper interfaces with the related type. The elements *FaultInjection*, *DelayManagement* and *StateEstimation* must be available exactly once.

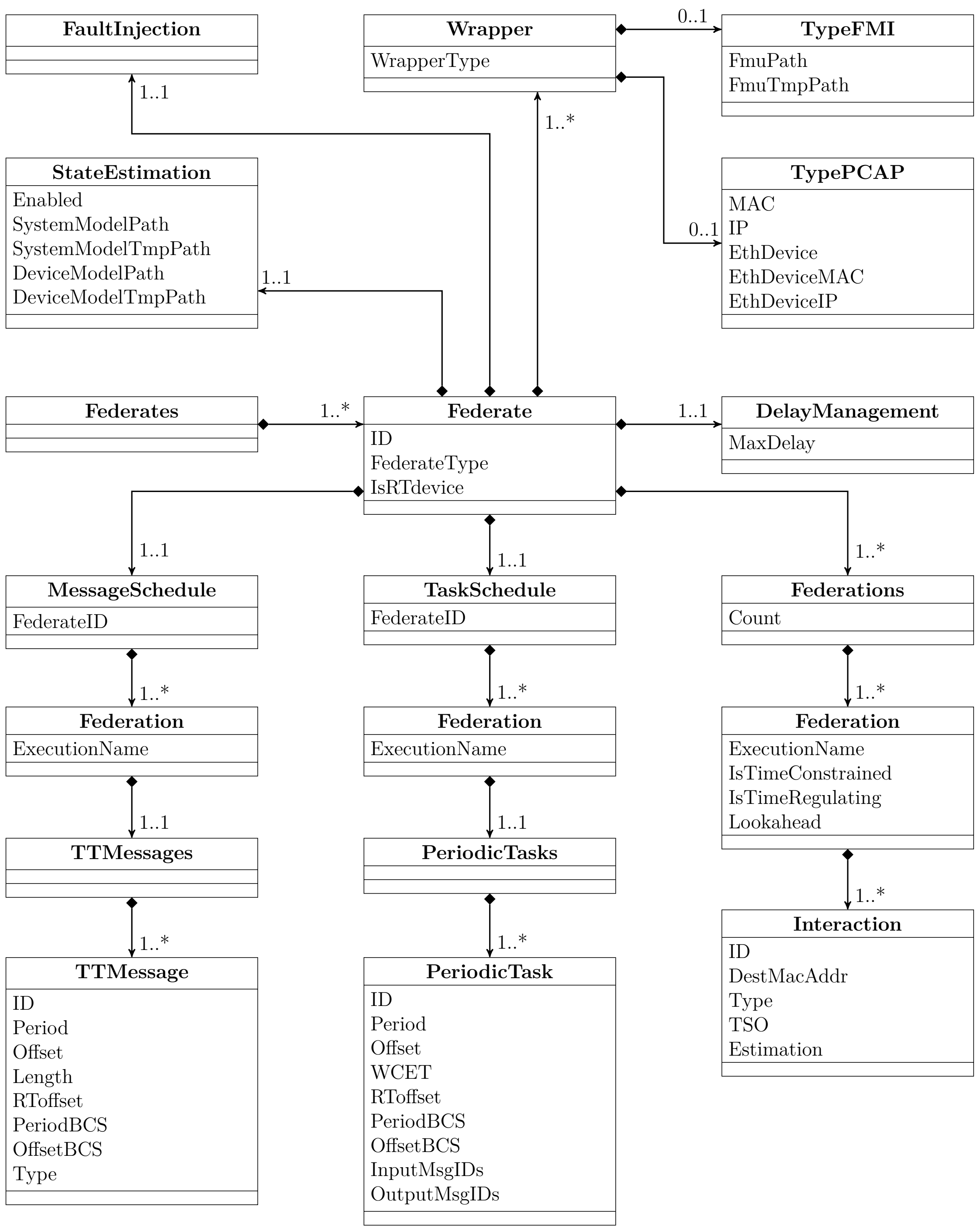


Figure A.3.: Attributes of the federates.

Figure A.4 illustrates the attributes of the different message error modes for the fault-injection. Each mode is optional in the configuration model.

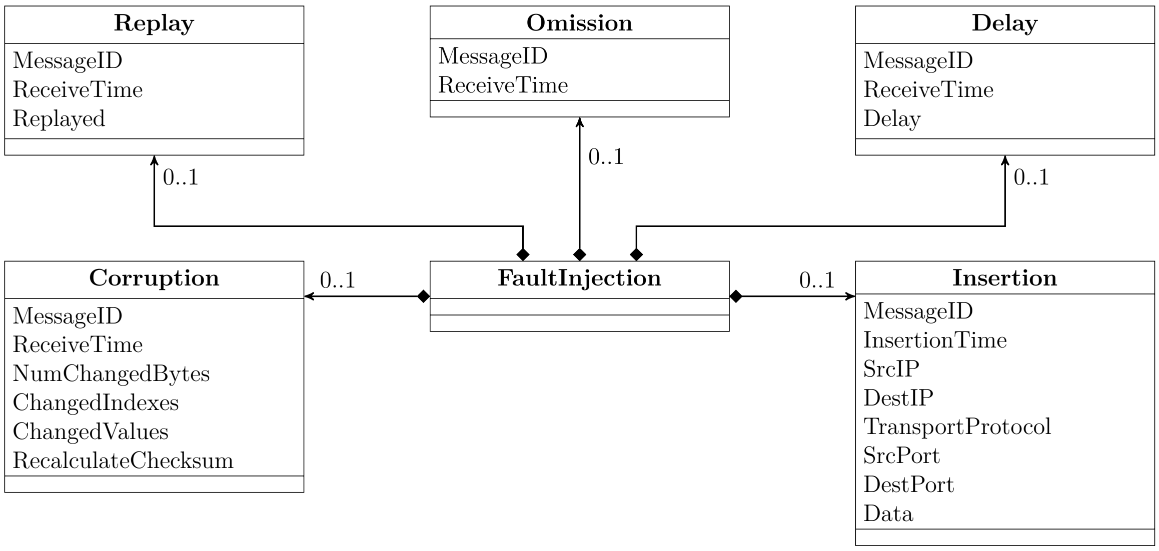


Figure A.4.: Attributes of error modes for the fault-injection.

Finally, Figure A.5 shows the synchronization points used during the initialization of the simulation bridges. Each synchronization point is announced by a dedicated host called *SynchMaster* while the registering federates are defined in the attribute *ManageFederate*.

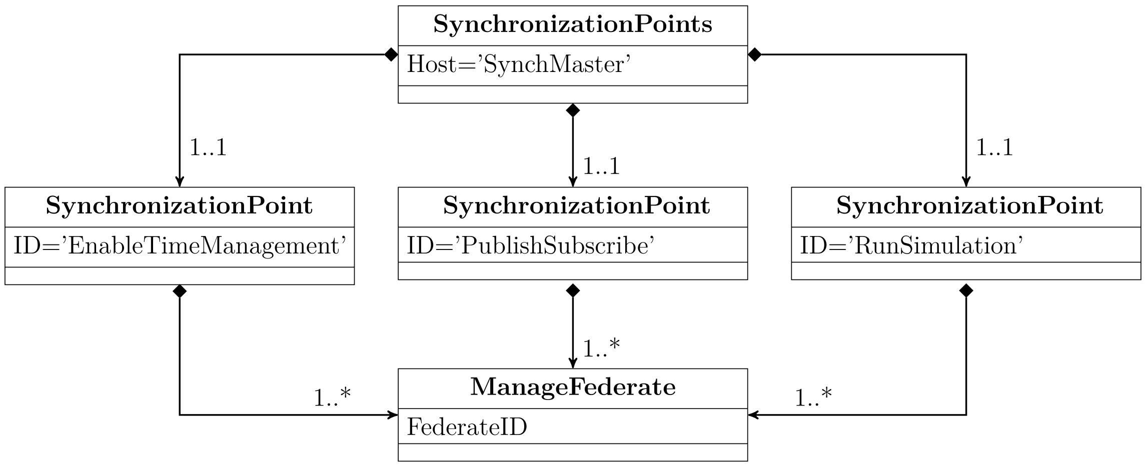


Figure A.5.: Attributes of synchronization points.

B. Synchronization Messages

This annex shows the linear relationship between the number of components and the number of synchronization messages per period. It further depicts the reduction of the message number using longer communication periods.

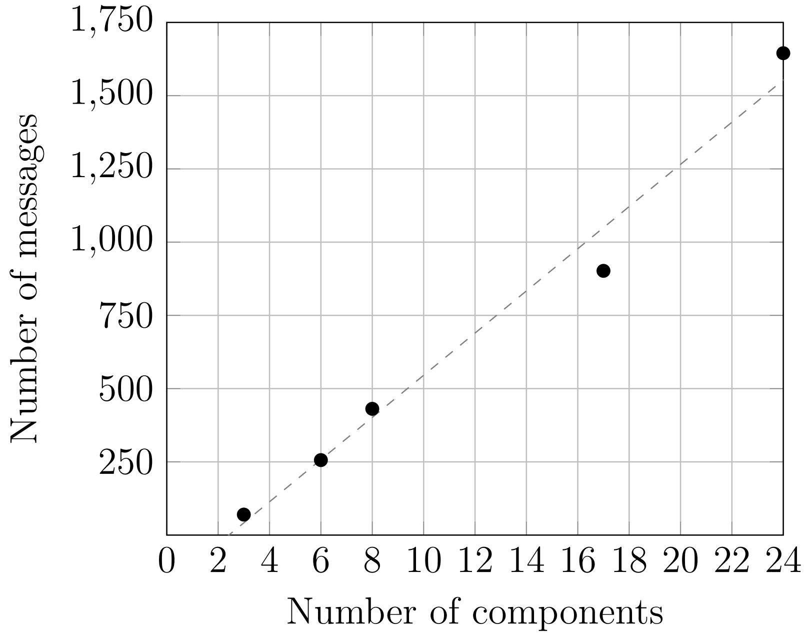


Figure B.1.: Number of synchronization data messages per period.

Table B.1.: Number of messages for a simulated time of 1s.

CP	Setup 1	Setup 2	Setup 3	Setup 4	Setup 5
10	7000	25600	43100	180400	329000
100	700	2560	4310	18040	32900
200	350	1280	2155	9020	16450
500	140	512	862	3608	6580
1000	70	256	431	1804	3290

Figure B.1 plots the number of components on the x-axis while the y-axis depicts the number of synchronization messages per period. The values are derived from Table 6.7 and show the linear relationship. As explained in Section 6.4.1, the number of messages per period is multiplied with the fraction of the simulated time and the period to determine the total number of messages in the run. In Table B.1, this number is determined for a simulated time of 1s with regard to the communication periods chosen. The results show an exponential reduction which is reflected by the achievable speedups using the state-estimation mechanism (cf. Section 6.5.2).

C. Simulation Durations without Delay-Management

In this annex, the simulation durations and their deviations are illustrated for the different setups and topologies without considering any delay-management mechanism. It is related to Section 6.4.2.

Table C.1.: Simulation durations and deviations for Setup 1.

Simulated	Local results $[s]$		LAN results $[s]$		Internet results $[s]$	
Time	Duration	Deviation	Duration	Deviation	Duration	Deviation
1	0.507	0.071	0.605	0.024	23.016	0.326
2	1.038	0.136	1.214	0.040	46.489	0.732
5	2.545	0.313	3.000	0.063	117.759	0.962
10	4.968	0.447	5.973	0.107	235.874	2.797
20	10.012	0.948	11.970	0.241	469.749	7.833
50	25.831	2.423	29.816	0.530	1151.577	35.883
100	50.591	4.158	59.779	0.857	2297.814	79.523
200	101.262	9.338	119.155	1.888	4662.179	171.339
500	257.083	24.471	298.832	4.897	11332.691	565.943
1000	515.487	48.952	595.004	8.586	22857.437	829.183

Table C.2.: Simulation durations and deviations for Setup 2.

Simulated	Local results $[s]$		LAN results $[s]$		Internet results $[s]$	
Time	Duration	Deviation	Duration	Deviation	Duration	Deviation
1	0.999	0.104	1.011	0.032	44.082	2.148
2	1.958	0.161	1.989	0.051	88.828	4.497
5	4.849	0.355	4.952	0.097	221.435	10.878
10	9.638	0.754	9.830	0.148	442.336	17.187
20	19.215	1.280	19.795	0.307	839.284	28.747
50	47.754	3.186	49.467	0.709	2064.156	97.710
100	95.354	6.163	99.390	1.380	4058.514	125.191
200	190.756	12.755	198.903	2.645	8223.501	245.490
500	475.533	32.151	499.589	5.633	20955.291	616.151
1000	951.296	63.899	1001.938	9.453	41751.530	630.075

Table C.3.: Simulation durations and deviations for Setup 3.

Simulated	Local results [s]		LAN results [s]		Internet results [s]	
Time	Duration	Deviation	Duration	Deviation	Duration	Deviation
1	1.125	0.120	1.105	0.038	43.088	0.997
2	2.244	0.197	2.204	0.059	86.146	2.214
5	5.531	0.419	5.405	0.117	217.808	5.679
10	10.952	0.858	10.679	0.125	437.430	12.449
20	21.760	1.653	21.357	0.248	882.488	30.820
50	54.737	3.546	53.008	0.525	2222.672	68.768
100	109.661	6.776	106.523	1.097	4530.210	142.667
200	219.091	14.427	212.723	2.940	8864.005	306.378
500	549.608	33.986	530.033	5.362	22590.943	1792.854
1000	1118.347	57.368	1064.416	7.239	46319.824	1887.877

Table C.4.: Simulation durations and deviations for Setup 4.

Simulated	Local results [s]		LAN results [s]	
Time	Duration	Deviation	Duration	Deviation
1	1.613	0.038	1.247	0.043
2	3.230	0.109	2.539	0.096
5	8.138	0.346	6.457	0.181
10	16.546	0.559	12.502	0.334
20	32.931	1.183	25.114	0.578
50	82.272	2.837	62.848	1.275
100	164.000	5.739	125.919	2.257
200	328.928	9.858	250.283	4.120
500	824.432	15.359	624.070	8.810
1000	1649.943	29.419	1246.363	7.304

Table C.5.: Simulation durations and deviations for Setup 5.

Simulated	Local results [s]		LAN results [s]	
Time	Duration	Deviation	Duration	Deviation
1	3.757	0.300	1.756	0.012
2	7.472	0.480	3.496	0.025
5	18.469	0.714	8.729	0.051
10	36.497	1.410	17.476	0.110
20	72.575	2.788	34.910	0.247
50	181.657	5.922	86.945	0.858
100	363.746	9.513	173.996	2.116
200	730.843	11.681	348.883	2.444
500	1846.789	36.999	868.292	8.052
1000	3509.915	352.563	1740.511	12.389

D. Speedup Using State-Estimation

This appendix shows the regression slopes, the simulation durations and their deviations for the different setups. It further depicts the power functions which illustrate the resulting simulation speedups. The results refer to Section 6.5.

Table D.1.: Regression slopes of Setups 1, 2 and 3 for the different topologies and communication periods.

	Setup 1			Setup 2			Setup 3		
CP	Local	LAN	Internet	Local	LAN	Internet	Local	LAN	Internet
10	0.515	0.595	22.820	0.951	1.002	41.781	1.116	1.064	46.169
100	0.069	0.092	3.093	0.077	0.088	5.419	0.132	0.180	5.800
200	0.040	0.062	1.480	0.046	0.063	2.654	0.072	0.137	2.826
500	0.020	0.041	0.627	0.019	0.057	1.109	0.039	0.108	1.137
1000	0.014	0.032	0.336	0.014	0.048	0.614	0.027	0.095	0.657

Table D.2.: Local and LAN regression slopes of Setups 4 and 5 for different communication periods.

	Setup 4		Setup 5	
CP	Local	LAN	Local	LAN
10	1.650	1.246	3.537	1.740
100	0.371	0.238	0.854	0.379
200	0.190	0.128	0.432	0.193
500	0.093	0.076	0.179	0.087
1000	0.050	0.067	0.099	0.059

Table D.3.: Power functions for regression of speedups.

	Local		LAN		Internet	
Setup	Base	Exponent	Base	Exponent	Base	Exponent
1	2.973	-0.797	2.214	-0.643	197.190	-0.921
2	6.985	-0.932	3.183	-0.666	357.360	-0.923
3	6.577	-0.822	2.885	-0.534	403.140	-0.934
4	10.382	-0.760	5.212	-0.664		
5	24.635	-0.781	10.401	-0.752		

Table D.4.: Simulation durations and deviations for Setup 1.

		Local results [s]		LAN results [s]		Internet results [s]	
CP	ST	Duration	Deviation	Duration	Deviation	Duration	Deviation
100	1	0.068	0.013	0.098	0.007	2.983	0.078
	2	0.130	0.022	0.191	0.011	6.051	0.062
	5	0.329	0.059	0.460	0.021	15.440	0.132
	10	0.653	0.121	0.945	0.050	30.908	0.182
	20	1.262	0.183	1.842	0.065	62.114	0.472
	50	3.251	0.378	4.588	0.151	155.213	0.951
	100	6.599	0.852	9.201	0.318	312.306	2.194
	200	13.545	1.889	18.311	0.473	622.672	6.277
	500	34.350	4.150	46.014	1.263	1690.499	269.045
	1000	68.845	7.957	92.323	1.990	3037.643	179.391
200	1	0.039	0.011	0.062	0.006	1.386	0.024
	2	0.075	0.015	0.125	0.009	2.866	0.032
	5	0.187	0.038	0.310	0.015	7.298	0.044
	10	0.382	0.081	0.618	0.030	14.679	0.144
	20	0.731	0.111	1.223	0.045	29.451	0.228
	50	1.889	0.369	3.093	0.134	73.737	0.323
	100	3.855	0.616	6.152	0.214	147.180	0.454
	200	7.858	1.356	12.372	0.450	294.941	1.311
	500	19.625	2.961	30.970	0.837	738.289	1.884
	1000	39.971	7.169	61.832	2.079	1480.746	7.213
500	1	0.024	0.009	0.040	0.005	0.538	0.013
	2	0.044	0.018	0.081	0.006	1.166	0.030
	5	0.102	0.037	0.200	0.010	3.031	0.065
	10	0.215	0.077	0.401	0.015	6.098	0.077
	20	0.430	0.139	0.813	0.019	12.395	0.228
	50	1.141	0.400	2.021	0.055	30.847	0.283
	100	2.270	0.713	4.053	0.078	62.167	0.617
	200	4.631	1.556	8.068	0.176	124.335	1.119
	500	10.228	1.938	20.432	0.292	311.500	2.751
	1000	20.210	3.913	40.897	0.679	626.969	7.538
1000	1	0.018	0.009	0.031	0.005	0.250	0.015
	2	0.038	0.017	0.063	0.005	0.587	0.027
	5	0.085	0.038	0.155	0.007	1.578	0.024
	10	0.152	0.058	0.310	0.012	3.279	0.089
	20	0.326	0.131	0.624	0.017	6.645	0.198
	50	0.861	0.341	1.580	0.025	16.636	0.127
	100	1.644	0.578	3.164	0.047	33.371	0.336
	200	3.614	1.487	6.346	0.084	67.015	0.864
	500	6.715	0.775	16.050	0.168	168.196	1.575
	1000	13.951	2.364	32.269	0.322	335.203	2.540

Table D.5.: Simulation durations and deviations for Setup 2.

CP	ST	Local results $[s]$		LAN results $[s]$		Internet results $[s]$	
		Duration	Deviation	Duration	Deviation	Duration	Deviation
100	1	0.178	0.043	0.102	0.020	5.133	0.083
	2	0.360	0.053	0.194	0.031	10.529	0.093
	5	0.833	0.186	0.468	0.056	27.014	0.965
	10	1.604	0.411	0.907	0.111	53.460	0.487
	20	3.125	0.806	1.796	0.183	106.862	0.246
	50	7.173	2.097	4.474	0.427	269.436	1.310
	100	11.302	3.511	8.886	0.614	543.326	4.499
	200	15.460	7.826	17.848	1.083	1101.497	8.194
	500	38.972	19.194	43.958	1.586	2775.096	29.897
	1000	78.645	38.514	88.381	3.414	5393.089	151.262
200	1	0.111	0.026	0.073	0.017	2.543	0.039
	2	0.226	0.068	0.152	0.031	5.292	0.056
	5	0.604	0.094	0.345	0.077	13.737	0.276
	10	1.259	0.192	0.682	0.131	27.503	0.269
	20	2.594	0.440	1.370	0.258	55.167	0.512
	50	5.322	1.938	3.250	0.492	137.854	0.989
	100	7.081	3.209	6.528	0.970	275.672	2.095
	200	8.508	3.726	12.825	1.620	570.008	13.221
	500	22.643	10.440	31.567	3.286	1375.723	24.218
	1000	47.135	22.578	63.117	6.621	2637.267	43.819
500	1	0.069	0.019	0.051	0.010	0.834	0.013
	2	0.138	0.043	0.112	0.018	1.909	0.038
	5	0.417	0.047	0.286	0.043	5.138	0.054
	10	0.857	0.135	0.588	0.081	10.700	0.118
	20	1.656	0.376	1.205	0.153	21.482	0.166
	50	3.041	1.784	2.919	0.280	54.376	0.417
	100	4.397	2.363	5.772	0.417	109.796	0.604
	200	4.034	0.786	11.475	0.742	218.463	1.759
	500	10.053	2.079	28.657	2.034	549.115	2.725
	1000	20.296	4.069	56.711	3.663	1110.605	5.903
1000	1	0.037	0.020	0.038	0.008	0.331	0.012
	2	0.089	0.041	0.083	0.012	0.925	0.024
	5	0.326	0.039	0.231	0.022	2.717	0.038
	10	0.644	0.171	0.468	0.048	5.723	0.152
	20	1.303	0.451	0.937	0.047	11.692	0.117
	50	2.447	1.567	2.379	0.144	29.612	0.265
	100	3.284	1.645	4.742	0.374	59.482	0.262
	200	3.075	0.334	9.583	0.502	119.898	0.776
	500	7.465	0.962	24.025	1.442	301.268	2.303
	1000	15.059	1.906	48.457	1.707	615.550	9.760

Table D.6.: Simulation durations and deviations for Setup 3.

		Local results $[s]$		LAN results $[s]$		Internet results $[s]$	
CP	ST	Duration	Deviation	Duration	Deviation	Duration	Deviation
100	1	0.183	0.025	0.208	0.013	5.450	0.067
	2	0.338	0.050	0.416	0.027	11.167	0.092
	5	0.770	0.108	1.024	0.073	28.506	0.827
	10	1.474	0.239	2.045	0.120	57.440	0.336
	20	2.927	0.484	3.975	0.252	115.426	0.950
	50	7.064	1.225	9.938	0.500	291.301	1.297
	100	13.764	2.625	19.533	0.982	641.017	100.549
	200	26.857	5.759	38.191	1.949	1148.486	26.583
	500	66.091	13.081	91.761	5.328	2798.674	45.354
	1000	131.906	25.350	179.562	11.794	5845.029	101.171
200	1	0.107	0.023	0.145	0.011	4.065	0.231
	2	0.224	0.029	0.295	0.023	8.916	0.404
	5	0.516	0.088	0.752	0.044	22.904	0.738
	10	0.963	0.175	1.485	0.124	36.103	5.650
	20	1.805	0.307	2.928	0.227	60.802	1.132
	50	4.328	0.802	7.376	0.408	149.208	1.281
	100	8.186	1.949	14.524	1.029	302.307	2.126
	200	14.720	3.638	28.599	2.174	552.212	4.791
	500	36.478	8.647	70.380	5.603	1396.253	23.388
	1000	72.404	16.309	136.309	12.239	2841.560	74.613
500	1	0.067	0.019	0.096	0.009	0.960	0.037
	2	0.138	0.025	0.212	0.013	2.216	0.061
	5	0.372	0.062	0.542	0.046	6.203	0.774
	10	0.715	0.139	1.095	0.066	12.192	0.195
	20	1.342	0.235	2.224	0.142	24.520	0.245
	50	3.215	0.753	5.469	0.363	61.864	0.458
	100	5.279	1.488	11.251	0.674	123.741	0.756
	200	7.943	1.986	22.332	1.474	231.856	3.025
	500	19.830	4.648	55.844	2.562	572.934	8.769
	1000	39.207	8.535	107.859	7.291	1137.368	10.425
1000	1	0.041	0.014	0.075	0.009	0.371	0.008
	2	0.110	0.026	0.177	0.009	1.059	0.019
	5	0.312	0.061	0.459	0.029	3.049	0.049
	10	0.602	0.116	0.938	0.083	6.466	0.067
	200	1.035	0.214	1.896	0.182	13.220	0.148
	500	2.812	0.518	4.795	0.357	33.444	0.189
	100	4.637	1.679	9.641	0.900	66.723	0.365
	200	5.172	0.700	19.682	1.250	133.418	0.774
	500	13.499	2.095	48.565	3.568	333.113	2.244
	1000	27.594	4.309	94.649	6.591	655.566	7.593

Table D.7.: Simulation durations and deviations for Setup 4.

		Local results [s]		LAN results [s]	
CP	ST	Duration	Deviation	Duration	Deviation
100	1	0.333	0.039	0.272	0.007
	2	0.655	0.076	0.519	0.015
	5	1.680	0.130	1.246	0.032
	10	3.586	0.127	2.462	0.054
	20	7.321	0.211	4.863	0.102
	50	18.417	0.555	12.147	0.231
	100	36.824	1.044	24.137	0.566
	200	73.898	2.081	48.178	1.021
	500	185.234	5.174	120.152	2.548
	1000	370.380	10.422	239.118	6.841
200	1	0.177	0.020	0.168	0.007
	2	0.345	0.037	0.301	0.012
	5	0.846	0.089	0.704	0.018
	10	1.732	0.131	1.364	0.039
	20	3.661	0.163	2.667	0.083
	50	9.381	0.304	6.590	0.182
	100	18.875	0.526	13.161	0.357
	200	37.764	1.135	26.388	0.859
	500	94.548	2.798	65.021	2.061
	1000	189.556	5.570	127.837	5.089
500	1	0.126	0.357	0.119	0.013
	2	0.169	0.008	0.219	0.013
	5	0.415	0.010	0.493	0.037
	10	0.807	0.017	0.944	0.077
	20	1.687	0.080	1.740	0.147
	50	4.526	0.354	4.027	0.328
	100	9.162	0.747	8.332	0.738
	200	18.440	1.499	16.287	1.461
	500	46.378	3.884	39.079	2.616
	1000	92.695	7.580	75.602	6.865
1000	1	0.054	0.007	0.079	0.012
	2	0.107	0.012	0.175	0.022
	5	0.235	0.017	0.426	0.043
	10	0.454	0.031	0.864	0.086
	20	0.887	0.067	1.628	0.200
	50	2.385	0.037	3.851	0.456
	100	4.859	0.075	7.519	0.671
	200	9.818	0.153	14.405	1.341
	500	24.681	0.374	34.960	3.347
	1000	49.463	0.778	66.649	7.628

Table D.8.: Simulation durations and deviations for Setup 5.

		Local results [s]		LAN results [s]	
CP	ST	Duration	Deviation	Duration	Deviation
100	1	0.687	0.113	0.392	0.011
	2	1.512	0.078	0.768	0.019
	5	4.117	0.046	1.897	0.055
	10	8.394	0.056	3.777	0.091
	20	16.895	0.127	7.594	0.148
	50	42.484	0.280	19.063	0.344
	100	85.054	0.598	37.896	0.782
	200	170.224	1.139	75.253	1.568
	500	426.127	2.809	189.667	3.809
	1000	853.475	5.368	378.451	6.138
200	1	0.336	0.033	0.214	0.007
	2	0.679	0.076	0.410	0.009
	5	1.947	0.087	0.988	0.022
	10	4.173	0.040	1.955	0.039
	20	8.481	0.066	3.856	0.098
	50	21.417	0.209	9.663	0.183
	100	42.895	0.384	19.396	0.386
	200	86.097	0.975	38.465	0.700
	500	215.569	2.422	96.229	1.977
	1000	431.475	4.735	192.852	3.150
500	1	0.133	0.009	0.115	0.007
	20	0.277	0.024	0.204	0.008
	5	0.706	0.074	0.473	0.011
	10	1.585	0.073	0.918	0.024
	20	3.422	0.024	1.790	0.055
	50	8.802	0.095	4.411	0.104
	100	17.727	0.215	8.819	0.250
	200	35.605	0.449	17.604	0.437
	500	89.209	1.234	43.894	0.911
	1000	178.813	2.549	86.562	2.155
1000	1	0.065	0.004	0.070	0.007
	2	0.152	0.008	0.163	0.010
	5	0.396	0.025	0.360	0.018
	10	0.790	0.069	0.676	0.033
	20	1.764	0.079	1.292	0.071
	50	4.767	0.029	3.138	0.159
	100	9.691	0.073	6.239	0.272
	200	19.575	0.345	12.320	0.578
	500	49.069	0.616	30.400	1.258
	1000	98.471	1.054	58.874	3.443

E. Speedup Using Speculative Execution

Finally, this annex depicts the simulation durations and their deviations using the speculative execution mechanism. From these results, the speedups in Section 6.6 are derived.

Table E.1.: Simulation durations and deviations for Setup 1 (Local and LAN).

	Simulated	Local results $[s]$		LAN results $[s]$	
Federations	Time	Duration	Deviation	Duration	Deviation
2	1	0.461	0.072	0.425	0.042
2	2	0.890	0.118	0.862	0.085
2	5	2.169	0.253	2.064	0.174
2	10	4.391	0.539	4.125	0.355
2	20	8.733	0.814	8.066	0.633
2	50	22.084	2.167	20.171	1.451
2	100	45.920	3.969	39.040	1.567
2	200	93.369	7.155	79.474	4.514
2	500	237.302	16.287	196.774	8.582
2	1000	478.356	28.579	394.673	17.260

Table E.2.: Simulation durations and deviations for Setup 1 via the Internet.

	Simulated	Results local RTI $[s]$		Results remote RTI $[s]$	
Federations	Time	Duration	Deviation	Duration	Deviation
2	1	9.997	0.128	10.151	0.304
2	2	20.053	0.274	20.474	0.267
2	5	51.105	0.549	55.198	0.675
2	10	102.578	1.922	123.688	7.956
2	20	204.564	2.311	278.380	30.500
2	50	512.693	4.562	541.067	8.928
2	100	1029.854	17.888	1046.211	13.144
2	200	2103.796	59.256	2019.613	20.112
2	500	5266.874	201.924	4989.317	36.846
2	1000	10145.756	289.171	10528.737	260.282

Table E.3.: Simulation durations and deviations for Setup 2 (Local and LAN).

	Simulated	Local results [s]		LAN results [s]	
Federations	Time	Duration	Deviation	Duration	Deviation
3	1	0.803	0.110	0.675	0.071
3	2	1.567	0.117	1.354	0.089
3	5	3.863	0.603	3.174	0.283
3	10	7.842	1.204	6.186	0.341
3	20	12.201	1.067	12.303	0.577
3	50	31.540	3.415	31.061	1.185
3	100	63.289	5.414	61.536	2.296
3	200	128.276	10.079	123.280	3.647
3	500	323.980	21.625	307.791	9.024
3	1000	651.085	34.372	621.245	17.599

Table E.4.: Simulation durations and deviations for Setup 2 via the Internet.

	Simulated	Results local RTI [s]		Results remote RTI [s]	
Federations	Time	Duration	Deviation	Duration	Deviation
3	1	17.761	0.182	23.962	0.911
3	2	35.843	0.291	47.492	0.564
3	5	89.146	0.540	120.423	1.514
3	10	177.566	1.409	238.533	2.204
3	20	355.213	2.508	446.255	22.204
3	50	896.553	3.945	1065.972	5.252
3	100	1835.921	32.947	2113.100	2.361
3	200	3668.204	116.814	4367.162	99.786
3	500	8807.984	268.222	11248.975	369.584
3	1000	17739.610	550.607	22364.392	1202.987

Table E.5.: Simulation durations and deviations for Setup 3 (Local and LAN).

	Simulated	Local results [s]		LAN results [s]	
Federations	Time	Duration	Deviation	Duration	Deviation
3	1	0.803	0.061	0.795	0.050
3	2	1.616	0.093	1.595	0.103
3	5	3.818	0.268	3.816	0.200
3	10	7.756	0.623	7.644	0.280
3	20	14.209	1.301	15.093	0.559
3	50	35.684	3.300	37.893	1.286
3	100	72.064	6.696	75.427	2.293
3	200	144.737	14.154	150.632	3.707
3	500	359.477	31.400	372.424	7.420
3	1000	720.618	59.759	745.607	14.009

Table E.6.: Simulation durations and deviations for Setup 3 via the Internet.

	Simulated	Results local RTI [s]		Results remote RTI [s]	
Federations	Time	Duration	Deviation	Duration	Deviation
3	1	19.028	0.094	27.384	0.266
3	2	37.870	0.320	55.977	0.194
3	5	95.503	0.623	138.704	1.231
3	10	191.324	2.580	275.607	0.878
3	20	378.819	2.361	556.704	3.113
3	50	954.609	16.373	1408.824	15.355
3	100	1945.225	50.815	2890.253	10.889
3	200	3958.592	163.626	5718.621	160.893
3	500	9467.390	431.471	14516.948	1087.374
3	1000	19001.124	1068.624	29795.943	505.008

Table E.7.: Simulation durations and deviations for Setup 4.

	Simulated	Local results $[s]$		LAN results $[s]$	
Federations	Time	Duration	Deviation	Duration	Deviation
3	1	0.671	0.092	0.809	0.043
3	2	1.337	0.116	1.585	0.094
3	5	3.471	0.214	3.801	0.163
3	10	6.983	0.426	7.348	0.306
3	20	13.943	0.821	14.446	0.494
3	50	34.914	2.064	35.555	1.148
3	100	69.977	4.149	70.492	2.186
3	200	139.884	7.896	139.584	4.408
3	500	350.159	19.844	347.216	9.259
3	1000	701.178	40.919	685.264	14.600
5	1	0.589	0.075	0.893	0.075
5	2	1.187	0.133	1.802	0.132
5	5	3.065	0.260	4.171	0.370
5	10	6.184	0.458	8.185	0.612
5	20	12.439	0.867	16.081	1.014
5	50	31.185	2.040	39.083	2.823
5	100	62.359	4.090	76.413	5.554
5	200	125.022	8.118	148.453	9.275
5	500	312.822	20.601	357.806	17.702
5	1000	627.000	40.586	701.488	25.998
7	1	0.409	0.042	0.779	0.029
7	2	0.807	0.060	1.538	0.085
7	5	2.056	0.137	3.690	0.259
7	10	4.175	0.248	7.399	0.358
7	20	8.354	0.476	14.259	1.100
7	50	21.123	1.356	34.363	2.931
7	100	42.135	2.461	66.216	5.450
7	200	84.063	4.327	125.609	10.748
7	500	212.043	12.595	287.558	25.010
7	1000	430.562	19.806	538.992	39.122

Table E.8.: Simulation durations and deviations for Setup 5.

	Simulated	Local results [s]		LAN results [s]	
Federations	Time	Duration	Deviation	Duration	Deviation
3	1	1.618	0.045	1.070	0.128
3	2	3.339	0.037	2.032	0.144
3	5	8.528	0.123	5.106	0.339
3	10	17.076	0.289	10.011	0.718
3	20	34.240	0.539	19.975	1.149
3	50	85.945	1.659	51.205	3.750
3	100	170.289	2.632	99.739	5.510
3	200	339.616	4.422	206.120	16.362
3	500	861.981	13.481	499.368	19.303
3	1000	1700.891	32.626	1026.970	42.100
5	1	1.382	0.047	0.955	0.088
5	2	2.841	0.045	1.831	0.191
5	5	7.154	0.108	4.437	0.419
5	10	14.334	0.202	8.785	0.814
5	20	29.538	1.133	17.814	1.388
5	50	74.693	3.286	44.550	3.688
5	100	149.340	6.183	87.921	6.052
5	200	300.427	13.533	170.258	10.702
5	500	741.147	26.425	413.376	20.531
5	1000	1454.757	133.422	801.574	16.430
9	1	0.706	0.039	0.795	0.043
9	2	1.466	0.052	1.499	0.114
9	5	3.832	0.123	3.723	0.311
9	10	7.803	0.238	7.216	0.517
9	20	15.683	0.580	13.505	0.983
9	50	39.520	1.222	32.164	2.264
9	100	80.046	1.738	63.932	4.053
9	200	160.519	3.405	122.233	6.999
9	500	401.576	8.753	285.712	21.503
9	1000	803.829	17.130	565.571	38.590

F. Publications and Copyright Information

This thesis uses content which was published in the following publications:

T. Pieper and R. Obermaisser, "Distributed co-simulation for software-in-the-loop testing of networked railway systems," in *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, Budva, 2018, pp. 1-5. ©2018 IEEE.

T. Pieper and R. Obermaisser, "Network-Centric Co-Simulation Framework for Software-In-the-Loop Testing of Geographically Distributed Simulation Components," in *2018 IEEE International Conference on Computational Science and Engineering (CSE)*, Bucharest, 2018, pp. 104-111. ©2018 IEEE.

T. Pieper and R. Obermaisser, "Fault-Injection for Software-in-the-Loop Testing of Networked Railway Systems," in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, Budva, Montenegro, 2019, pp. 1-4. ©2019 IEEE.

T. Pieper and R. Obermaisser, "State-Estimation for Delay-Management in Distributed Real-Time Co-Simulation via the Internet," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, Helsinki, Finland, 2019, pp. 939-946. ©2019 IEEE.

T. Pieper and R. Obermaisser, "State-Estimation for Increased Accuracy in Software- and Hardware-In-The-Loop testing via the Internet," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Zaragoza, Spain, 2019, pp. 298-305. ©2019 IEEE.

From these publications, the following figures and tables are used in a similar or in an adapted version:

References

[AB04] L. Abeni and G. Buttazzo, “Resource Reservation in Dynamic Real-Time Systems,” *Real-Time Systems*, vol. 27, no. 2, pp. 123–167, 2004.

[aca11] acatech (Ed.), *Cyber-Physical Systems. Driving force for innovation in mobility, health, energy and production*, ser. acatech POSITION. Heidelberg u.a.: Springer Verlag, 2011.

[ALR$^+$01] A. Avizienis, J.-C. Laprie, B. Randell *et al.*, *Fundamental concepts of dependability.* University of Newcastle upon Tyne, Computing Science, 2001.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[AM12] M. Agrawal and P. Mishra, “A comparative survey on symmetric key encryption techniques,” *International Journal on Computer Science and Engineering*, vol. 4, no. 5, p. 877, 2012.

[AMO$^+$02] A. Amory, F. Moraes, L. Oliveira, N. Calazans, and F. Hessel, “A heterogeneous and distributed co-simulation environment,” in *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design.* IEEE, 2002, pp. 115–120.

[AOA16] H. Ahmadian, R. Obermaisser, and M. Abuteir, “Time-triggered and rate-constrained on-chip communication in mixed-criticality systems,” in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC).* IEEE, 2016, pp. 117–124.

[APE$^+$13] M. U. Awais, P. Palensky, A. Elsheikh, E. Widl, and S. Matthias, “The high level architecture RTI as a master to the functional mock-up interface components,” in *2013 International Conference on Computing, Networking and Communications (ICNC).* IEEE, 2013, pp. 315–320.

[APM$^+$13] M. U. Awais, P. Palensky, W. Mueller, E. Widl, and A. Elsheikh, “Distributed hybrid simulation using the HLA and the Functional Mock-up Interface,” *Industrial Electronics Society, IECON*, pp. 7564–7569, 2013.

[asa17] “ASAM AE XIL Generic Simulator Interface,” Association for Standardisation of Automation and Measuring Systems, Tech. Rep., Jun 2017.

[Assa] M. Association, “DCP Homepage,” Website, online available at https://dcp-standard.org/; accessed on 19.11.2019.

[Assb] ——, “FMI Homepage,” Website, online available at https://fmi-standard.org/; accessed on 11.10.2019.

[Bar] R. Barry, “The FreeRTOS Kernel,” Website, online available at https://www.freertos.org/; accessed on 22.10.2019.

[BAR10] A. Buss and A. Al Rowaei, “A comparison of the accuracy of discrete event and discrete time,” in *Proceedings of the Winter Simulation Conference.* Winter Simulation Conference, 2010, pp. 1468–1477.

[BBG+13] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, “Determinate composition of FMUs for co-simulation,” in *Proceedings of the Eleventh ACM International Conference on Embedded Software.* IEEE Press, 2013, p. 2.

[BBZ+13] R. Bottura, D. Babazadeh, K. Zhu, A. Borghetti, L. Nordström, and C. A. Nucci, “SITL and HLA Co-simulation Platforms: Tools for Analysis of the Integrated ICT and Electric Power System,” in *Eurocon 2013.* IEEE, 2013, pp. 918–925.

[BCD+12] S. Baccari, G. Cammeo, C. Dufour, L. Iannelli, V. Mungiguerra, M. Porzio, G. Reale, and F. Vasca, “Real-time hardware-in-the-loop in railway: simulations for testing control software of electromechanical train components,” in *Railway Safety, Reliability, and Security: Technologies and Systems Engineering.* IGI Global, 2012, pp. 221–248.

[BCK96] M. Bellare, R. Canetti, and H. Krawczyk, “Keying hash functions for message authentication,” in *Annual international cryptology conference.* Springer, 1996, pp. 1–15.

[BCPS11] C. Bonivento, M. Cacciari, A. Paoli, and M. Sartini, “Rapid prototyping of automated manufacturing systems by software-in-the-loop simulation,” in *2011 Chinese Control and Decision Conference (CCDC).* IEEE, 2011, pp. 3968–3973.

[Bel11] L. L. Bello, “The case for Ethernet in Automotive Communications,” *ACM SIGBED Review*, vol. 8, no. 4, pp. 7–15, 2011.

[Ber] R. Bermbach, “Echtzeitfähigkeit mit dem Linux RT-Preempt Patch in FPGA-basierten Prozessorsystemen,” online available at https://www.ostfalia.de/cms/de/pws/bermbach/.content/documents/Forschungsbericht-WF-4_SS11_Be.pdf; accessed on 11.10.2019.

[BGD12] R. Beamonte, F. Giraldeau, and M. Dagenais, “High performance tracing tools for multicore linux hard real-time systems,” in *Proceedings of the 14th Real-Time Linux Workshop.* OSADL, 2012.

[BH06] M. Badra and I. Hajjeh, “Enabling VPN and secure remote access using TLS protocol,” in *2006 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications.* IEEE, 2006, pp. 308–314.

[BH13] M. Benedikt and A. Hofer, “Guidelines for the Application of a Coupling

Method for Non-iterative Co-simulation," in *2013 8th EUROSIM Congress on Modelling and Simulation.* IEEE, 2013, pp. 244–249.

[BK94] M. E. Baran and A. W. Kelley, "State estimation for real-time monitoring of distribution systems," *IEEE Transactions on Power Systems*, vol. 9, no. 3, pp. 1601–1609, 1994.

[BKD+19] P. Baumann, M. Krammer, M. Driussi, L. Mikelsons, J. Zehetner, W. Mair, and D. Schramm, "Using the Distributed Co-Simulation Protocol for a Mixed Real-Virtual Prototype," in *2019 IEEE International Conference on Mechatronics (ICM)*, vol. 1. IEEE, 2019, pp. 440–445.

[BKKS12] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz, "What is the benefit of a model-based design of embedded software systems in the car industry?" in *Emerging Technologies for the Evolution and Maintenance of Software Models.* IGI Global, 2012, pp. 343–369.

[BKP+15] D. Bian, M. Kuzlu, M. Pipattanasomporn, S. Rahman, and Y. Wu, "Real-time Co-simulation Platform using OPAL-RT and OPNET for Analyzing Smart Grid Performance," in *2015 IEEE Power & Energy Society General Meeting.* IEEE, 2015, pp. 1–5.

[BL05] A. Boukerche and K. Lu, "Design and performance evaluation of a real-time RTI infrastructure for large-scale distributed simulations," in *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications.* IEEE, 2005, pp. 203–212.

[BLM+08] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application," *IEEE Transactions on Nuclear Science*, vol. 55, no. 1, pp. 435–439, 2008.

[BOA+11] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauß, M. Monteiro, T. Neidhold, D. Neumerkel *et al.*, "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," in *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, no. 063. Linköping University Electronic Press, 2011, pp. 105–114.

[BOA+12] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel *et al.*, "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models," in *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, no. 076. Linköping University Electronic Press, 2012, pp. 173–184.

[BS03] B. Bréholée and P. Siron, "Design and Implementation of a HLA Inter-federation Bridge," 2003.

[bsi19] "BSI – Technical Guideline: Cryptographic Mechanisms: Recommendations

and Key Lengths," Federal Office for Information Security, Tech. Rep., Feb 2019.

[BST10] K. Berns, B. Schürmann, and M. Trapp, *Eingebettete Systeme - Systemgrundlagen und Entwicklung eingebetteter Software.* Vieweg, 2010.

[Bun17] M. Bunge, *Causality and modern science.* Routledge, 2017.

[BVP10] J. Bélanger, P. Venne, and J.-N. Paquin, "The what, where and why of real-time simulation," *Planet Rt*, vol. 1, no. 1, pp. 25–29, 2010.

[CB13] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS RT," in *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications.* SYSGO AG, 2013, pp. 19–29.

[CDF^{+}14] S. Ciraci, J. Daily, J. Fuller, A. Fisher, L. Marinovici, and K. Agarwal, "FNCS: A framework for power system and communication networks co-simulation," in *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative.* Society for Computer Simulation International, 2014, p. 36.

[Cha18] L. Chappell, "North America, Europe and the world top suppliers," *Automotive News*, Jun 2018.

[CLT^{+}16] F. Cremona, M. Lohstroh, S. Tripakis, C. Brooks, and E. A. Lee, "FIDE – An FMI Integrated Development Environment," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing.* ACM, 2016, pp. 1759–1766.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed Deadlock Detection," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 2, pp. 144–156, 1983.

[CMS01] S. Carr, J. Mayo, and C.-K. Shene, "Race Conditions: A Case Study," *Journal of Computing Sciences in Colleges*, vol. 17, no. 1, pp. 90–105, 2001.

[CNS11] J.-B. Chaudron, E. Noulard, and P. Siron, "Design and modeling techniques for real-time RTI time management," in *Simulation Interoperability Workshop*, 2011, p. 32.

[CSSA14] J.-B. Chaudron, D. Saussié, P. Siron, and M. Adelantado, "Real-time distributed simulations in an HLA framework: Application to aircraft simulation," *Simulation*, vol. 90, no. 6, pp. 627–643, 2014.

[CTG01] W. Cai, S. J. Turner, and B. P. Gan, "Hierarchical Federations: An Architecture for Information Hiding," in *Proceedings of the fifteenth workshop on Parallel and distributed simulation.* IEEE Computer Society, 2001, pp. 67–74.

[CW96] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[DBDC03] S. Dellacherie, L. Burgaud, and P. Di Crescenzo, “Improve - HDL - a DO-254 formal property checker used for design and verification of avionics protocol controllers,” in *The 22nd Digital Avionics Systems Conference, 2003 (DASC'03)*, vol. 1. IEEE, 2003, pp. 1–A.

[dcp19] “DCP Specification Document, Version 1.0,” Modelica Association Project DCP, Linköping, Sweden, Tech. Rep., 2019, online available at https://dcp-standard.org/assets/specification/DCP_Specification_v1.0.pdf; accessed on 19.11.2019.

[DD15] G. A. Ditzel and P. Didier, “Time Sensitive Network (TSN) Protocols and use in EtherNet/IP Systems,” in *2015 ODVA Industry Conference & 17th Annual Meeting*, 2015.

[DFW98] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly, “The DoD high level architecture: an update,” in *Simulation Conference Proceedings, 1998. Winter*, vol. 1. IEEE, 1998, pp. 797–804.

[DG08] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[DGK07] S. Demers, P. Gopalakrishnan, and L. Kant, “A Generic Solution to Software-in-the-Loop,” in *MILCOM 2007 - IEEE Military Communications Conference*, Oct 2007, pp. 1–6.

[Din89] A. Dinning, “A survey of synchronization methods for parallel computers,” *Computer*, vol. 22, no. 7, pp. 66–77, 1989.

[DM03] L. Dozio and P. Mantegazza, “Linux Real Time Application Interface (RTAI) in low cost high performance motion control,” *Motion Control*, vol. 2003, no. 1, pp. 1–15, 2003.

[DNC03] P. De, A. Neogi, and T.-c. Chiueh, “Virtualwire: A fault injection and analysis tool for network protocols,” in *Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003.* IEEE, 2003, pp. 214–221.

[do111] “Software Considerations in Airborne Systems and Equipment Certification,” RTCA & EUROCAE, Standard DO-178C / ED-12C, Dec 2011.

[DW] A. Dunkels and L. Woestenberg, “Lightweight IP stack,” Website, online available at https://www.nongnu.org/lwip/2_1_x/index.html; accessed on 22.10.2019.

[EAWP13] A. Elsheikh, M. U. Awais, E. Widl, and P. Palensky, “Modelica-Enabled Rapid Prototyping of Cyber-Physical Energy Systems Via The Functional Mockup Interface,” in *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES).* IEEE, 2013, pp. 1–6.

[EBK03] W. Elmenreich, G. Bauer, and H. Kopetz, *The time-triggered paradigm.* Citeseer, 2003.

[EBSF12] T. Ersal, M. Brudnak, J. L. Stein, and H. K. Fathy, “Statistical Transparency Analysis in Internet-Distributed Hardware-in-the-Loop Simulation,” *IEEE/ASME transactions on mechatronics*, vol. 17, no. 2, pp. 228–238, 2012.

[EC98] S. Easterbrook and J. Callahan, “Formal methods for verification and validation of partial specifications: A case study,” *Journal of Systems and Software*, vol. 40, no. 3, pp. 199–210, 1998.

[Eck08] C. Eckert, *IT-Sicherheit: Konzepte - Verfahren - Protokolle.* Oldenbourg Wissenschaftsverlag GmbH, 2008.

[EH97] T. Elgamal and K. E. Hickman, “Secure socket layer application program apparatus and method,” Aug. 12 1997, uS Patent 5,657,390.

[en510] “Railway applications - Communication, signalling and processing systems - Safety-related communication in transmission systems,” European Committee for Electrotechnical Standardization, Norm CENELEC EN50159-2010, Sep 2010.

[FB12] A. Facchinetti and S. Bruni, “Hardware-in-the-loop hybrid simulation of pantograph–catenary interaction,” *Journal of Sound and Vibration*, vol. 331, no. 12, pp. 2783–2797, 2012.

[Fin18] N. Finn, “Introduction to Time-Sensitive Networking,” *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 22–28, JUNE 2018.

[Fis07] M. S. Fisher, *Software verification and validation: An engineering and scientific approach.* Springer Science+Business Media, LLC, Jan 2007.

[FLV14] J. Fitzgerald, P. G. Larsen, and M. Verhoef, *Collaborative Design for Embedded Systems: Co-modelling and Co-simulation*, ser. SpringerLink: Bücher. Springer Publishing Company, Incorporated, 2014. [Online]. Available: https://books.google.de/books?id=4ka3BAAAQBAJ

[Fri04] R. Friend, “Making the gigabit IPsec VPN architecture secure,” *Computer*, vol. 37, no. 6, pp. 54–60, 2004.

[FRMN18] L. Fejoz, B. Regnier, P. Miramont, and N. Navet, “Simulation-based fault injection as a verification oracle for the engineering of time-triggered Ethernet networks,” in *Proceedings of Embedded Real-Time Software and Systems (ERTS 2018)*, 2018.

[Fuj00] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, ser. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., 2000.

[FvHRK06] H. Fuhrmann, R. von Hanxleden, J. Rennhack, and J. Koch, “Model-based system design of time-triggered architectures-avionics case study,” in *2006 ieee/aiaa 25TH Digital Avionics Systems Conference.* IEEE, 2006, pp. 1–12.

[Gab96] F. Gabbay, *Speculative execution based on value prediction.* Technion-IIT, Department of Electrical Engineering, 1996.

[GBBP13] D. Gessner, M. Barranco, A. Ballesteros, and J. Proenza, “SfiCAN: A star-based physical fault-injection infrastructure for CAN networks,” *IEEE Transactions on Vehicular Technology*, vol. 63, no. 3, pp. 1335–1349, 2013.

[GD13] R. Gore and S. Diallo, “The need for usable formal methods in verification and validation,” in *2013 Winter Simulations Conference (WSC).* IEEE, 2013, pp. 1257–1268.

[GDKR16] T. Gerlach, U. Durak, A. Knüppel, and T. Rambau, “Running High Level Architecture in Real-Time for Flight Simulator Integration,” in *AIAA Modeling and Simulation Technologies Conference*, 2016, p. 4130.

[GF15] A. Garro and A. Falcone, “On the integration of HLA and FMI for supporting interoperability and reusability in distributed simulation,” in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium.* Society for Computer Simulation International, 2015, pp. 9–16.

[GK91] G. Grünsteidl and H. Kopetz, “A Reliable Multicast Protocol for Distributed Real-Time Systems,” *IFAC Proceedings Volumes*, vol. 24, no. 2, pp. 19–23, 1991.

[GR04] J. Goossens and P. Richard, “Overview of real-time scheduling problems,” in *Euro Workshop on Project Management and Scheduling.* Citeseer, 2004.

[HB97] K. Hines and G. Borriello, “Selective focus as a means of improving geographically distributed embedded system co-simulation,” in *Proceedings of the 8th IEEE International Workshop on Rapid System Prototyping Shortening the Path from Specification to Prototype.* IEEE, 1997, pp. 58–62.

[Hen09] L. Henriques, “Threaded IRQs on Linux PREEMPT-RT,” in *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications.* Citeseer, 2009, pp. 23–32.

[HGPM16] F. Huerta, J. K. Gruber, M. Prodanovic, and P. Matatagui, “Power-hardware-in-the-loop test beds: evaluation tools for grid integration of distributed energy resources,” *IEEE Industry Applications Magazine*, vol. 22, no. 2, pp. 18–26, 2016.

[HGY07] C. Harding, A. Griffiths, and H. Yu, “An Interface between MATLAB and OPNET to Allow Simulation of WNCS with MANETs,” in *2007 IEEE International Conference on Networking, Sensing and Control.* IEEE, 2007, pp. 711–716.

[HKH11] C. Hübner, M. Khattabi, and C. Huth, “State Estimation in Niederspannungsnetzen mit hohem Anteil dezentraler Erzeugung,” in *ETG Congress, Würzburg*, 2011.

[HLV06] H. R. Hertzfeld, A. N. Link, and N. S. Vonortas, “Intellectual property protection mechanisms in research partnerships,” *Research Policies, Special Issue on Property and the pursuit of knowledge: IPR issues affecting scientific research*, vol. 35, no. 6, pp. 825–838, 2006.

[HS95] M. M. Hugue and R. C. Scalzo, “Specifying Fault Tolerance in Large Complex Computing Systems,” in *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’95).* IEEE, 1995, pp. 369–372.

[HSB10] O. Heimlich, R. Sailer, and L. Budzisz, “Nmlab: A co-simulation framework for matlab and ns-2,” in *2010 Second International Conference on Advances in System Simulation.* IEEE, 2010, pp. 152–157.

[HSR95] S. Han, K. G. Shin, and H. A. Rosenberg, “Doctor: An integrated software fault injection environment for distributed real-time systems,” in *Proceedings of the International Computer Performance and Dependability Symposium.* IEEE, 1995, pp. 204–213.

[HTI97] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault injection techniques and tools,” *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[HYCY09] M. S. Hasan, H. Yu, A. Carrington, and T. Yang, “Co-simulation of wireless networked control systems over mobile ad hoc network using SIMULINK and OPNET,” *IET communications*, vol. 3, no. 8, pp. 1297–1310, 2009.

[HYGY08] M. Hasan, H. Yu, A. Griffiths, and T. Yang, “Co-simulation framework for Networked Control Systems over multi-hop mobile ad-hoc networks,” in *The 17th IFAC World Congress, the International Federation of Automatic Control, Seoul, Korea.*, 2008.

[iec10] “Functional safety of electrical/electronic/programmable electronic safety-related systems,” International Electrotechnical Commission, Standard IEC 61508:2010, Apr 2010.

[iec11] “Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems ,” International Electrotechnical Commission, Standard IEC 61508-2:2010, Feb 2011.

[iec15] “Electronic railway equipment – Train communication network (TCN) – Part 2-3: TCN communication profile,” International Electrotechnical Commission, Standard IEC 61375-2-3, Jul 2015.

[iee10a] “IEEE standard for modeling and simulation (M&S) High Level Architecture (HLA)– Federate Interface Specification,” Institute of Electrical and Electronics Engineers, Standard IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000), Aug 2010.

[iee10b] “IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules,” Institute of Electrical and Electronics Engineers, Standard IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000), Aug 2010.

[iee10c] “IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Object Model Template (OMT) Specification,” Institute of

Electrical and Electronics Engineers, Standard IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000), Aug 2010.

[iee16] "IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," Standard IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012), Dec 2016.

[INC] O. INC., "VPN Software Solutions & Services For Business | OpenVPN," Website, online available at https://openvpn.net/; accessed on 18.10.2019.

[Int19] Interdisziplinärer Forschungsverbaund BAHNTECHNIK e.V., "Firmen-Verzeichnis Bahn- und Verkehrstechnik," 2019, online available at http://www.bahntechnik-firmen.info/firmenverzeichnis.pdf; accessed on 12.10.2019.

[ISO11] "Road vehicles – Functional safety," International Standardization Organization, Norm ISO 26262, 2011.

[iso13] "Road vehicles — FlexRay communications system — Part 1: General information and use case definition," International Standardization Organization, Norm ISO 17458-1:2013, 2013.

[iso15] "Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling," International Standardization Organization, Norm ISO 11898-1:2015, 2015.

[JBN15] J. C. V. S. Junior, A. V. Brito, and T. P. Nascimento, "Testing Real-Time Embedded Systems with Hardware-in-the-Loop Simulation Using High Level Architecture," in *2015 Brazilian symposium on computing systems engineering (SBESC)*. IEEE, 2015, pp. 142–147.

[JGSJS18] M. Jakovljevic, A. Geven, N. Simanic-John, and D. M. Saatci, "Next-Gen Train Control / Management (TCMS) Architectures: "Drive-By-Data" System Integration Approach," in *ERTS 2018*, ser. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, Jan 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02156252

[JMV04] N. Juristo, A. M. Moreno, and S. Vegas, "Reviewing 25 years of testing technique experiments," *Empirical Software Engineering*, vol. 9, no. 1-2, pp. 7–44, 2004.

[KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The Time-Triggered Ethernet (TTE) Design," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE, 2005, pp. 22–33.

[KAH04] H. Kopetz, A. Ademaj, and A. Hanzlikm, "Integration of Internal and External Clock Synchronization by the Combination of Clock-State and Clock-Rate Correction in Fault-Tolerant Distributed Systems," in *25th IEEE International Real-Time Systems Symposium*. IEEE, 2004, pp. 415–425.

[KB18] M. Krammer and M. Benedikt, “Master for Simulation Control using the Distributed Co-Simulation Protocol,” in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 2018, pp. 329–334.

[KBB+18] M. Krammer, M. Benedikt, T. Blochwitz, K. Alekeish, N. Amringer, C. Kater, S. Materne, R. Ruvalcaba, K. Schuch, J. Zehetner *et al.*, “The distributed co-simulation protocol for the integration of real-time systems and simulation environments,” in *Proceedings of the 50th Computer Simulation Conference*. Society for Computer Simulation International, 2018, p. 1.

[KC99] W. H. Kwon and S.-G. Choi, “Real-time distributed software-in-the-loop simulation for distributed control systems,” in *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*. IEEE, 1999, pp. 115–119.

[KC13] J. H. Koh and B. W. Choi, “Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions,” *International Journal of Control and Automation*, vol. 6, no. 1, pp. 235–246, 2013.

[KESHO07] H. Kopetz, C. El-Salloum, B. Huber, and R. Obermaisser, “Periodic Finite-State Machines,” in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’07)*. IEEE, 2007, pp. 10–20.

[KKMS95] H. Kopetz, A. Kruger, D. Millinger, and A. Schedl, “A Synchronization Strategy for a Time-Triggered Multicluster Real-Time System,” in *Proceedings of the 14th Symposium on Reliable Distributed Systems*. IEEE, 1995, pp. 154–161.

[KKS+15] A. Kohn, M. Käßmeyer, R. Schneider, A. Roger, C. Stellwag, and A. Herkersdorf, “Fail-operational in safety-related automotive multi-core systems,” in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2015, pp. 1–4.

[KM03] D. Koufaty and D. T. Marr, “Hyperthreading technology in the netburst microarchitecture,” *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[KO87] H. Kopetz and W. Ochsenreiter, “Clock Synchronization in Distributed Real-Time Systems,” *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 933–940, 1987.

[KO02] H. Kopetz and R. Obermaisser, “Temporal composability [real-time embedded systems],” *Computing & Control Engineering Journal*, vol. 13, no. 4, pp. 156–162, 2002.

[Kob87] N. Koblitz, “Elliptic curve cryptosystems,” *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[KÖC+94] R. Krtolica, Ü. Özgüner, H. Chan, H. Göktas, J. Winkelman, and M. Liubakka, “Stability of linear feedback systems with random communication delays,” *International Journal of Control*, vol. 59, no. 4, pp. 925–953, 1994.

[Kop91] H. Kopetz, "Event-Triggered versus Time-Triggered Real-Time Systems," in *Operating Systems of the 90s and Beyond.* Springer, 1991, pp. 86–101.

[Kop92] ——, "Sparse Time versus Dense Time in Distributed Real-Time Systems," in *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems.* IEEE, 1992, pp. 460–467.

[Kop95] ——, "Why Time-Triggered Architectures will Succeed in Large Hard Real-Time Systems," in *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems.* IEEE, 1995, pp. 2–9.

[Kop98a] ——, "Component-Based Design of Large Distributed Real-Time Systems," *Control Engineering Practice*, vol. 6, no. 1, pp. 53–60, 1998.

[Kop98b] ——, "The Time-Triggered Architecture," *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, pp. 22–29, 1998.

[Kop98c] ——, "The Time-Triggered Model of Computation," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279).* IEEE, 1998, pp. 168–177.

[Kop99] ——, "Elementary versus Composite Interfaces in Distributed Real-Time Systems," in *Proceedings of the Fourth International Symposium on Autonomous Decentralized Systems.-Integration of Heterogeneous Systems-.* IEEE, 1999, pp. 26–33.

[Kop00] ——, "Software Engineering for Real-Time: A Roadmap," in *Proceedings of the Conference on the Future of Software Engineering.* Citeseer, 2000, pp. 201–211.

[Kop02a] ——, "On the Specification of Linking Interfaces in Distributed Real-Time Systems," *Unpublished draft, Technische Universität Wien, Institut für Technische Informatik, Treitlstr*, pp. 1–3, 2002.

[Kop02b] ——, "Time-triggered real-time computing," *IFAC Proceedings Volumes*, vol. 35, no. 1, pp. 59–70, 2002.

[Kop07] ——, "Why do we need a Sparse Global Time-Base in Dependable Real-time Systems?" in *2007 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication.* IEEE, 2007, pp. 13–17.

[Kop08a] ——, "The Complexity Challenge in Embedded System Design," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC).* IEEE, 2008, pp. 3–12.

[Kop08b] ——, "The Rationale for Time-Triggered Ethernet," in *2008 Real-Time Systems Symposium.* IEEE, 2008, pp. 3–11.

[Kop11] ——, *Real-Time Systems - Design Principles for Distributed Embedded Applications*, 2nd ed., ser. Real-Time Systems Series. Berlin Heidelberg:

Springer Science & Business Media, 2011. [Online]. Available: https://books.google.de/books?id=oJZsvEawlAMC

[KPB16] R. M. Kulkarni, R. P. Patil, and C. R. Bhukya, “Hardware–In–Loop Test Bench Based Failure Mode Effects Test Automation,” *International Journal Of Innovative Research In Electrical, Electronics, Instrumentation And Control Engineering*, vol. 4, no. 6, 2016.

[KRI$^+$12] L. A. Knauth, R. Rajwar, P. J. Irelan, M. G. Dixon, and K. K. Lai, “Method, apparatus, and system for speculative execution event counter checkpointing and restoring,” Sep 2012, US Patent App. 13365,104.

[KS03] H. Kopetz and N. Suri, “Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces,” in *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003.* IEEE, 2003, pp. 51–60.

[KSK$^+$19] M. Krammer, K. Schuch, C. Kater, K. Alekeish, T. Blochwitz, S. Materne, A. Soppa, and M. Benedikt, “Standardized Integration of Real-Time and Non-Real-Time Systems: The Distributed Co-Simulation Protocol,” in *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, no. 157. Linköping University Electronic Press, 2019.

[KTS$^+$15] B. M. Kelley, P. Top, S. G. Smith, C. S. Woodward, and L. Min, “A federated simulation toolkit for electric power grid and communication network co-simulation,” in *2015 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES).* IEEE, 2015, pp. 1–6.

[KZ10] A. J. Kornecki and J. Zalewski, “Hardware certification for real-time safety-critical systems: State of the art,” *Annual Reviews in Control*, vol. 34, no. 1, pp. 163–174, 2010.

[Lam78] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[LB05] C. Lin and S. A. Brandt, “Improving soft real-time performance through better slack reclaiming,” in *26th IEEE International Real-Time Systems Symposium (RTSS’05).* IEEE, 2005, pp. 12–pp.

[Ler05] P. N. Leroux, “RTOS versus GPOS,” *Embedded Computing Design*, 2005.

[Lev00] N. G. Leveson, “Intent specifications: An approach to building human-centered specifications,” *IEEE Transactions on software engineering*, vol. 26, no. 1, pp. 15–35, 2000.

[LFRE01] D. M. Lane, G. J. Falconer, G. Randall, and I. Edwards, “Interoperability and Synchronisation of Distributed Hardware-in-the-Loop Simulation for Underwater Robot Development: Issues and Experiments,” in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation (ICRA)*, vol. 1. IEEE, 2001, pp. 909–914.

[LH94] J. H. Lala and R. E. Harper, “Architectural principles for safety-critical real-time applications,” *Proceedings of the IEEE*, vol. 82, no. 1, pp. 25–40, 1994.

[Liu98] J. Liu, “Continuous Time and Mixed-Signal Simulation in Ptolemy II,” in *Dept. of EECS, University of California, Berkeley, CA.* Citeseer, 1998.

[LMLD11] W. Li, A. Monti, M. Luo, and R. A. Dougal, “VPNET: A co-simulation framework for analyzing communication channel effects on power systems,” in *2011 IEEE Electric Ship Technologies Symposium.* IEEE, 2011, pp. 143–149.

[LPS16] S. M. Laursen, P. Pop, and W. Steiner, “Routing optimization of AVB streams in TSN networks,” *ACM Sigbed Review*, vol. 13, no. 4, pp. 43–48, 2016.

[LS10] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*, 1st ed. Lee and Seshia, 2010. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/794.html

[LS11] N. Litayem and S. B. Saoud, “Impact of the linux real-time enhancements on the system performances for multi-core intel architectures,” *International Journal of Computer Applications*, vol. 17, no. 3, pp. 17–23, 2011.

[LSAF16] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, “Deadline scheduling in the Linux kernel,” *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.

[LSR05] Y. Liu, M. Steurer, and P. Ribeiro, “A Novel Approach to Power Quality Assessment: Real Time Hardware-in-the-Loop Test Bed,” *IEEE transactions on power delivery*, vol. 20, no. 2, pp. 1200–1201, 2005.

[LSW$^+$14] L. L. Lai, C. Shum, L. Wang, W. Lau, N. Tse, H. Chung, K. Tsang, and F. Xu, “Design a co-simulation platform for power system and communication network,” in *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC).* IEEE, 2014, pp. 3036–3041.

[LV62] R. E. Lyons and W. Vanderkulk, “The Use of Triple-Modular Redundancy to Improve Computer Reliability,” *IBM journal of research and development*, vol. 6, no. 2, pp. 200–209, 1962.

[LVS$^+$12] H. Lin, S. S. Veda, S. S. Shukla, L. Mili, and J. Thorp, “GECO: Global event-driven co-simulation framework for interconnected power system and communication network,” *IEEE Transactions on Smart Grid*, vol. 3, no. 3, pp. 1444–1456, 2012.

[LWFM07] B. Lu, X. Wu, H. Figueroa, and A. Monti, “A Low-Cost Real-Time Hardware-in-the-Loop Testing Approach of Power Electronics Controls,” *IEEE Transactions on Industrial Electronics*, vol. 54, no. 2, pp. 919–931, 2007.

[Mar06] P. Marwedel, *Embedded System Design*, 1st ed. Dordrecht, Netherlands: Springer, 2006.

[Mar18] ——, *Embedded System Design*, 3rd ed. Springer International Publishing AG, 2018.

[MEHL10] J. W. Mickens, J. Elson, J. Howell, and J. R. Lorch, "Crom: Faster Web Browsing Using Speculative Execution," in *NSDI*, vol. 10, 2010, pp. 9–9.

[Mer89] R. C. Merkle, "One way hash functions and DES," in *Conference on the Theory and Application of Cryptology.* Springer, 1989, pp. 428–446.

[MFF04] T. McLean, R. Fujimoto, and B. Fitzgibbons, "Middleware for real-time distributed simulations," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 15, pp. 1483–1501, 2004.

[MHL$^+$12] J. W. Mickens, J. R. Howell, J. R. Lorch, J. E. Elson, and E. B. Nightingale, "Network application performance enhancement using speculative execution," Mar 2012, US Patent 8,140,646.

[Mis86] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 39–65, 1986.

[MKC12] J. L. Mathieu, S. Koch, and D. S. Callaway, "State estimation and control of electric loads to manage real-time energy imbalance," *IEEE Transactions on Power Systems*, vol. 28, no. 1, pp. 430–440, 2012.

[MLRK09] K. J. Mitts, K. Lang, T. Roudier, and D. L. Kiskis, "Using a co-simulation framework to enable software-in-the-loop powertrain system development," SAE Technical Paper, Tech. Rep., 2009.

[MMPP07] M. Malvezzi, E. Meli, S. Papini, and L. Pugi, "Parametric models of railway systems for real-time applications," *Multibody dynamics*, 2007.

[MOB06] H. Matthijs, B. Orlic, and J. F. Broenink, "Co-Simulation of Networked Embedded Control Systems, a CSP-like process-oriented approach," in *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control.* IEEE, 2006, pp. 434–439.

[MS99] I. Mahadevan and K. M. Sivalingam, "Quality of service architectures for wireless networks: IntServ and DiffServ models," in *Proceedings of the Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99).* IEEE, 1999, pp. 420–425.

[MT13] N. S. Mkondoweni and R. Tzoneva, "LabNS2-co-simulation, co-emulation, and real-time control toolkit for investigation of network induced time delays and packet loss in networked control systems," 2013.

[Mul85] M. Mulazzani, "Reliability versus safety," *IFAC Proceedings Volumes*, vol. 18, no. 12, pp. 141–146, 1985.

[NCF05] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," in *ACM SIGOPS operating systems review*, vol. 39, no. 5. ACM, 2005, pp. 191–205.

[NGL$^+$14] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar, "Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems," in *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, no. 096. Linköping University Electronic Press, 2014, pp. 235–245.

[NGYF03] S. Naka, T. Genji, T. Yura, and Y. Fukuyama, "A hybrid particle swarm optimization for distribution state estimation," *IEEE Transactions on Power systems*, vol. 18, no. 1, pp. 60–68, 2003.

[NIS92] C. NIST, "The Digital Signature Standard," *Commun. ACM*, vol. 35, no. 7, pp. 36–40, Jul. 1992. [Online]. Available: http://doi.acm.org/10.1145/129902.129904

[NL11] J. Na and D. Lee, "Simulated fault injection using simulator modification technique," *ETRI Journal*, vol. 33, no. 1, pp. 50–59, 2011.

[NSRM09] N. Naumann, B. Schünemann, I. Radusch, and C. Meinel, "Improving v2x simulation performance with optimistic synchronization," in *2009 IEEE Asia-Pacific Services Computing Conference (APSCC).* IEEE, 2009, pp. 52–57.

[NTA$^+$18] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. ElBakoury, "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 88–145, 2018.

[OAO15] Z. Owda, M. Abuteir, and R. Obermaisser, "Co-simulation framework for networked multi-core chips with interleaving discrete event simulation tools," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA).* IEEE, 2015, pp. 1–8.

[Obe11] R. Obermaisser, *Time-Triggered Communication*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2011.

[oEEE] T. I. of Electrical and I. Electronics Engineers, "IEEE 802.15 Working Group for Wireless Specialty Networks (WSN) ," Website, online available at http://www.ieee802.org/15/; accessed on 05.03.2020.

[OIHVL04] F. J. Oppel III, B. Hart, and B. P. Van Leeuwen, "High Level Architecture (HLA) Federation with Umbra and OPNET Federates," Sandia National Laboratories, Tech. Rep., 2004.

[OO18] D. Onwuchekwa and R. Obermaisser, "Fault Injection Framework for Assessing Fault Containment of TTEthernet Against Babbling Idiot Failures," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS).* IEEE, 2018, pp. 1–6.

[OOF18] D. Onwuchekwa, R. Obermaisser, and J.-J. Foo, "Fault Injection Framework for Time Triggered Ethernet," in *2018 7th Transport and Research Arena (TRA).* Zenodo, Apr 2018.

[PA12] L. Pugi and B. Allotta, “Hardware-in-the-loop testing of on-board subsystems: Some case studies and applications,” in *Railway safety, reliability, and security: technologies and systems engineering.* IGI Global, 2012, pp. 249–280.

[PBC+96] P. Prinetto, A. Benso, F. Corno, M. Rebaudengo, M. Sonza Reorda, A. Amendola, L. Impagliazzo, and P. Marmo, “Fault behavior observation of a microprocessor system through a VHDL simulation-based fault injection experiment,” in *Proceedings of the conference on European design automation.* IEEE Computer Society Press, 1996, pp. 536–541.

[PK79] G. J. Popek and C. S. Kline, “Encryption and secure computer networks,” *ACM Computing Surveys (CSUR)*, vol. 11, no. 4, pp. 331–356, 1979.

[Pol94] S. Poledna, “Replica determinism in distributed real-time systems: A brief survey,” *Real-Time Systems*, vol. 6, no. 3, pp. 289–316, 1994.

[PS06] V. Pallipadi and A. Starikovskiy, “The ondemand governor,” in *Proceedings of the Linux Symposium*, vol. 2, no. 00216, 2006, pp. 215–230.

[PYPB02] A. Ponnappan, L. Yang, R. Pillai, and P. Braun, “A policy based QoS management system for the IntServ/DiffServ based Internet,” in *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks.* IEEE, 2002, pp. 159–168.

[QMBF12] D. Quaglia, R. Muradore, R. Bragantini, and P. Fiorini, “A SystemC/Matlab co-simulation tool for networked control systems,” *Simulation Modelling Practice and Theory*, vol. 23, pp. 71–86, 2012.

[RBP13] P. Royo, C. Barrado, and E. Pastor, “Isis+: A software-in-the-loop unmanned aircraft system simulator for nonsegregated airspace,” *Journal of Aerospace Information Systems*, vol. 10, no. 11, pp. 530–543, 2013.

[RLB08] P. Regnier, G. Lima, and L. Barreto, “Evaluation of interrupt handling timeliness in real-time linux operating systems,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 52–63, 2008.

[RPPF16] A. S. Roque, D. Pohren, C. E. Pereira, and E. P. Freitas, “Communication analysis in CAN networks under EFT injection,” in *2016 IEEE International Conference on Automatica (ICA-ACCA).* IEEE, 2016, pp. 1–6.

[RR] P. F. Riley and G. F. Riley, “Spades—A Distributed Agent Simulation Environment With Software-In-The-Loop Execution,” in *Proceedings of the 2003 Winter Simulation Conference S. Chick, PJ Sánchez, D. Ferrin, and DJ Morrice, eds.*

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[RTC+11] B. Rivera, J. A. Tufarolo, G. Comparetto, V. Lakshminarayan, M. Mirhakkak, E. H. Page, N. Schult, and D. Yoo, “An HLA-Based Approach to Quantify Achievable Performance for Tactical Edge Applications," ARMY RESEARCH LAB ADELPHI MD, Tech. Rep., 2011.

[Rus93] J. Rushby, *Formal methods and the certification of critical systems.* SRI International, Computer Science Laboratory, 1993.

[RVH07] S. Rostedt and D. V Hart, "Internals of the RT Patch," Jan 2007.

[SAC12] T. Schierz, M. Arnold, and C. Clauß, "Co-simulation with communication step size control in an FMI compatible master algorithm," in *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, no. 076. Linköping University Electronic Press, 2012, pp. 205–214.

[sae11] "Time-Triggered Ethernet," SAE Aerospace, Tech. Rep. SAE AS6802, Nov 2011.

[SAT07] B. Sahu and A.-R. Adl-Tabatabai, "Speculative execution past a barrier," Jun 2007, US Patent App. 11305,506.

[SB09] W. Steiner and G. Bauer, "TTEthernet: Time-triggered Services for Ethernet Networks," in *Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th*, 2009, p. 1.

[SBH$^+$09] W. Steiner, G. Bauer, B. Hall, M. Paulitsch, and S. Varadarajan, "TTEthernet Dataflow Concept," in *2009 Eighth IEEE International Symposium on Network Computing and Applications.* IEEE, 2009, pp. 319–322.

[SBS$^+$11] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok, "Runtime verification with state estimation," in *International conference on runtime verification.* Springer, 2011, pp. 193–207.

[Sch05] P. Scholz, *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*, ser. Xpert.press. Berlin: Springer, 2005.

[SCM16] W. Sun, X. Cai, and Q. Meng, "Testing flight software on the ground: Introducing the hardware-in-the-loop simulation method to the Alpha Magnetic Spectrometer on the International Space Station," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 815, pp. 83–90, 2016.

[SFB$^+$00] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS 2000).* IEEE, 2000, pp. 91–100.

[SFV18] B. Sangchoolie, P. Folkesson, and J. Vinter, "A study of the interplay between safety and security using model-implemented fault injection," in *2018 14th European Dependable Computing Conference (EDCC).* IEEE, 2018, pp. 41–48.

[SHBZ14] G. Stettinger, M. Horn, M. Benedikt, and J. Zehetner, “A Model-Based Approach for Prediction-Based Interconnection of Dynamic Systems,” in *53rd IEEE Conference on Decision and Control.* IEEE, 2014, pp. 3286–3291.

[SL06] C. Scordino and G. Lipari, “Linux and real-time: Current approaches and future opportunities,” in *IEEE Internafional Congress ANIPLA*, 2006.

[SLK+12] T. Steinbach, H.-T. Lim, F. Korf, T. C. Schmidt, D. Herrscher, and A. Wolisz, “Tomorrow’s In-Car Interconnect? A Competitive Evaluation of IEEE 802.1 AVB and Time-Triggered Ethernet (AS6802),” in *2012 IEEE Vehicular Technology Conference (VTC Fall).* IEEE, 2012, pp. 1–5.

[SM05] S. Shioda and K. Mase, “Performance comparison between IntServ-based and DiffServ-based networks,” in *IEEE Global Telecommunications Conference, 2005 (GLOBECOM’05)*, vol. 1. IEEE, 2005, pp. 6–pp.

[Smi92] M. D. Smith, “Support for speculative execution in high-performance processors,” Ph.D. dissertation, Citeseer, 1992.

[SMS+13] E. Shafei, I. Moawad, H. Sallam, Z. Taha, and M. Aref, “A Methodology for Safety Critical Software Systems Planning,” in *WSEAS European Computing Conference [Internet]. Dubrovnik, Croatia*, 2013, pp. 173–8.

[SP13] K. Sampigethaya and R. Poovendran, “Aviation Cyber–Physical Systems: Foundations for Future Aircraft and Air Transport,” *Proceedings of the IEEE*, vol. 101, no. 8, pp. 1834–1855, Aug 2013.

[SQ06] A. Santoro and F. Quaglia, “Transparent State Management for Optimistic Synchronization in the High Level Architecture,” *Simulation*, vol. 82, no. 1, pp. 5–20, 2006.

[SQ12] ——, “Transparent Optimistic Synchronization in the High-Level Architecture via Time-Management Conversion,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 22, no. 4, p. 21, 2012.

[SR04] J. A. Stankovic and R. Rajkumar, “Real-Time Operating Systems,” *Real-Time Systems*, vol. 28, no. 2-3, pp. 237–253, 2004.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky, “Aperiodic task scheduling for Hard-Real-Time systems,” *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[Sta12] W. Stallings, *Operating systems: internals and design principles.* Boston: Prentice Hall,, 2012.

[STB97] V. Sieh, O. Tschache, and F. Balbach, “VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions,” in *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, 1997 (FTCS-27), Digest of Papers.* IEEE, 1997, pp. 32–36.

[SV05] B.-H. Schlingloff and S. Vulinovic, “Zuverlässigkeitsprüfung eingebetteter Steuergeräte mit modellgetriebener Fehlerinjektion,” 2005.

[SY14] P. Sarhadi and S. Yousefpour, “State of the art: hardware in the loop modeling and simulation with its applications in design, development and implementation of system and control software,” *International Journal of Dynamics and Control*, vol. 3, Jan 2014.

[TB15] A. S. Tanenbaum and H. Bos, *Modern Operating Systems.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2015.

[Tec] P. Technologies, “Pitch pRTI – a Certified HLA RTI,” Website, online available at http://pitchtechnologies.com/products/prti/; accessed on 05.03.2020.

[Tho] L. Thomason, “GitHub - TinyXML-2 parser,” Website, online available at https://github.com/leethomason/tinyxml2; accessed on 19.10.2019.

[TKS99] P. Terwiesch, T. Keller, and E. Scheiben, “Rail vehicle control system integration testing using digital hardware-in-the-loop simulation,” *IEEE Transactions on Control Systems Technology*, vol. 7, no. 3, pp. 352–362, 1999.

[Tri15] S. Tripakis, “Bridging the semantic gap between heterogeneous modeling formalisms and FMI,” in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).* IEEE, 2015, pp. 60–69.

[tsn19] “Time-Sensitive Networking (TSN) Task Group,” Aug 2019. [Online]. Available: https://1.ieee802.org/tsn/

[tte] “Time-Triggered Ethernet technical whitepaper,” TTTech Computertechnik AG, Tech. Rep., online available at https://www.tttech.com/wp-content/uploads/TTTech_TTEthernet_Technical-Whitepaper.pdf; accessed on 22.11.2019.

[TW11] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. Prentice Hall, 2011.

[VBBH07a] J. Verhille, A. Bouscayrol, P. Barre, and J. Hautier, “Hardware-in-the-loop simulation of the traction system of an automatic subway,” in *2007 European Conference on Power Electronics and Applications.* IEEE, 2007, pp. 1–9.

[VBBH07b] ——, “Validation of anti-slip control for traction system using Hardware-In-the-Loop simulation,” in *2007 IEEE Vehicle Power and Propulsion Conference.* IEEE, 2007, pp. 440–447.

[VT] VT MAK, “MAK RTI,” Website, online available at https://www.mak.com/products/link/mak-rti; accessed on 05.03.2020.

[WAL$^+$13] S. Weiss, M. W. Achtelik, S. Lynen, M. C. Achtelik, L. Kneip, M. Chli, and R. Siegwart, “Monocular vision for long-term micro aerial vehicle state estimation: A compendium,” *Journal of Field Robotics*, vol. 30, no. 5, pp. 803–831, 2013.

[WB05] H. Wörn and U. Brinkschulte, *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen*, ser. eXamen.press. Springer Berlin Heidelberg

New York, 2005. [Online]. Available: https://books.google.de/books?id=buVfb0IMxjkC

[WCAF15] J. Wan, A. Canedo, and M. A. Al Faruque, "Functional model-based design methodology for automotive cyber-physical systems," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2028–2039, 2015.

[Wil12] A. Williams, *C++ concurrency in action: practical multithreading*. Manning Publ., 2012.

[Win84] R. S. Winternitz, "A secure one-way hash function built from DES," in *1984 IEEE Symposium on Security and Privacy*. IEEE, 1984, pp. 88–88.

[WSJ17] M. Wollschlaeger, T. Sauter, and J. Jasperneite, "The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0," *IEEE industrial electronics magazine*, vol. 11, no. 1, pp. 17–27, 2017.

[WTLG05] X. Wang, S. J. Turner, M. Y. H. Low, and B. P. Gan, "Optimistic Synchronization in HLA-Based Distributed Simulation," *Simulation*, vol. 81, no. 4, pp. 279–291, 2005.

[XA03] B. Xu and A. Abur, "State estimation of systems with embedded FACTS devices," in *Proceedings of the IEEE Bologna Power Tech Conference*, vol. 1. IEEE, 2003, pp. 5–pp.

[Xila] Xilinx, "100M/1G TSN Subsystem," Website, online available at https://www.xilinx.com/products/intellectual-property/1gtsn.html; accessed on 04.03.2020.

[Xilb] ——, "Digilent ZYBO," Website, online available at https://www.xilinx.com/support/university/boards-portfolio/xup-boards/DigilentZYBO.html; accessed on 22.10.2019.

[XN99] X. Xiao and L. M. Ni, "Internet QoS: a big picture," *IEEE network*, vol. 13, no. 2, pp. 8–18, 1999.

[YMS94] M. F. Younis, T. J. Marlowe, and A. D. Stoyenko, "Compiler Transformations for Speculative Execution in a Real-Time System," in *RTSS*, 1994, pp. 109–117.

[YWL02] Q.-Z. Yan, J. M. Williams, and J. Li, "Chassis control system development using simulation: software in the loop, rapid prototyping, and hardware in the loop," *SAE Transactions*, pp. 1734–1744, 2002.

[YZ12] Y. Yiqun and P. Zong, "Distributed Interactive Simulation for Iridium Communication System Based on HLA and OPNET," in *Proceedings of 2010 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010 no. 2)*, 2012.

[ZCTV13] M. Zoppi, C. Cervone, G. Tiso, and F. Vasca, "Software in the loop model and decoupling control for dual clutch automotive transmissions," in *3rd*

International Conference on Systems and Control. IEEE, 2013, pp. 349–354.

[ZCY06] G. Zhang, L. Chen, and A. Yao, "Study and Comparison of the RTHAL-based and ADEOS-based RTAI Real-time Solutions for Linux," in *First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06)*, vol. 2. IEEE, 2006, pp. 771–775.

[ZEK$^+$14] Z. Zhang, E. Eyisi, X. Koutsoukos, J. Porter, G. Karsai, and J. Sztipanovits, "A co-simulation framework for design of time-triggered automotive cyber physical systems," *Simulation modelling practice and theory*, vol. 43, pp. 16–33, 2014.

[ZG04] H. Zhao and N. D. Georganas, "HLA real-time extension," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 15, pp. 1503–1525, 2004.

[ZOS00] W. Zhao, D. Olshefski, and H. G. Schulzrinne, "Internet quality of service: An overview," Tech. Rep., 2000.

[ZR17] S. Zacher and M. Reuter, *Regelungstechnik für Ingenieure.* Springer Fachmedien Wiesbaden GmbH, 2017.