Fault-Tolerant Real-Time Architecture for Elderly Care

DISSERTATION
zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften (Dr.-Ing.)

vorgelegt von
Dipl.-Inform. Michael-Christian Schmidt

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

Betreuer und erster Gutachter:
Prof. Dr.-Ing. habil. Roman Obermaisser
Universität Siegen

Zweiter Gutachter:
Prof. Dr.-Ing. habil. Marcin Grzegorzek
Universität zu Lübeck

Tag der mündlichen Prüfung: 08.12.2020

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

Abstract

The ongoing transition from traditional elderly care to the use of modern technologies from the field of Cyber Physical Systems (CPSs) results in new challenges for both industry and research. This shift is mainly motivated by the increasing share of elderly people in the population which is causing a notable shortage of nursing staff. With the availability of new technologies, the CPSs for elderly care are also enabling new fields of applications in the area of biomedicine and robotics. Use cases like the automatic injection of insulin and robotic assistance are prominent examples for these application fields. These new application fields impose new requirements on architectures in the field of elderly care, such as deterministic real-time behavior and dependability along with an open-world assumption in which dynamic changes within the composition of the system can occur at run-time. Likewise, the application of robotic systems in the field of elderly care introduces stringent real-time requirements to the whole CPS, affecting the integration of complex and heterogeneous sensors, the control of actuators and the communication network. Moreover, the application of fault-tolerance and mixed-criticality techniques is required to establish a dependable CPS that is able to tolerate faults in order to prevent dangerous situations for human life. Furthermore, CPSs have to encompass different integration levels like the local network and the Internet in order to support services from professional stakeholders like medical services from caregivers or a doctor.

The proposed architecture for elderly care takes into account the new emerging application fields in elderly care as well as the associated challenges, which are (1) real-time support, (2) dependability and (3) support of an open-world assumption while taking into account multiple integration levels and the heterogeneity of the underlying technologies. A review of state-of-the-art architectures for elderly care shows that there is no architecture available at present that meets all these challenges. The proposed architecture addresses this gap by taking advantage of a broad range of well-known technologies and standards from the state-of-the-art like ISO/IEEE 11073 and Time Sensitive Networking (TSN) while further introducing new concepts and technologies, such as fault containment among containers for high-critical applications as well as real-time container-to-container communication with latencies and jitter in the low microsecond range. A huge challenge is further to address the open-world assumption while providing real-time guarantees and fault-tolerance. In particular, this puts further requirements to the real-time system like the capability of a dynamic rescheduling of real-time resources like the real-time network. This is addressed by the introduction of a service for the dynamic rescheduling of real-time communication resources that takes care about topology and service management, scheduling, configuration building and distribution of communication schedules. By this way, changes within the physical model (e.g. a new network switch or end system) and the logical model (e.g. a new service) are supported at run-time of the system.

In the field of software architectures, the use of microservices has reached a strong technical maturity in recent years. The proposed architecture is embracing this trend and introduces platform services as microservices. Finally, several proof-of-concept implementations are presented and evaluated in different experiments ranging from a real scenario to experiments in a laboratory in order to show that the proposed architecture for elderly care is able to address the shift in traditional elderly care to the use of modern technologies from the field of CPSs.

Zusammenfassung

Der fortschreitende Wandel von einer traditionellen Altenpflege hin zum Einsatz moderner Technologien aus dem Bereich der Cyber Physical Systems (CPSs) führt zu neuen Herausforderungen für Industrie und Forschung. Dieser Wandel ist vor allem durch den zunehmenden Anteil älterer Menschen an der Bevölkerung begründet und führt bereits jetzt zu einem bemerkenswerten Mangel an Pflegepersonal. Mit der Verfügbarkeit neuer Technologien ermöglichen die CPSs auf den Gebiet der Altenpflege neue Anwendungsfelder im Bereich der Biomedizin und Robotik. Die automatische Verabreichung von Insulin oder die Roboterunterstützung im Bereich der Pflege sind prominente Beispiele für diese Anwendungsfelder. Diese neuen Anwendungsfelder stellen neue Anforderungen an Architekturen im Bereich der Altenpflege, wie z.B. deterministisches Echtzeitverhalten und Zuverlässigkeit. Diese müssen zusammen mit einer Open-World-Annahme gelöst werden, bei der dynamische Veränderungen innerhalb der Zusammensetzung des Systems zur Laufzeit auftreten können. Ebenso stellt der Einsatz von Robotersystemen im Bereich der Altenpflege strenge Echtzeitanforderungen an das gesamte CPS, welche sich auf die Integration komplexer und heterogener Sensoren, die Steuerung von Aktoren und das Kommunikationsnetzwerk auswirken. Darüber hinaus ist die Anwendung von Fehlertoleranz- und Mixed-Criticality-Techniken erforderlich, um gefährliche Situationen für den Menschen zu verhindern. Zudem muss das CPS verschiedene Integrationsebenen, wie das lokale Netzwerk und das Internet, berücksichtigten, um Dienstleistungen von professionellen Stakeholdern, wie medizinische Dienstleister oder Ärzte zu unterstützen.

Die in dieser Arbeit vorgeschlagene Architektur berücksichtigt die neu entstehenden Anwendungsfelder in der Altenpflege, sowie die damit verbundenen Herausforderungen, nämlich (1) Echtzeit-Unterstützung, (2) Zuverlässigkeit und (3) Unterstützung einer Open-World-Annahme unter Berücksichtigung mehrerer Integrationsebenen und der Heterogenität der zugrunde liegenden Technologien. Ein Überblick über Architekturen für die Altenpflege im Stand der Technik zeigt, dass es derzeit keine Architektur gibt, welche allen Herausforderungen gerecht wird. Die vorgeschlagene Architektur schließt diese Lücke, indem sie sich eine breite Vielfalt bekannter Technologien und Standards aus dem Stand der Technik wie ISO/IEEE 11073 und Time Sensitive Networking (TSN) zunutze macht und gleichzeitig neue Konzepte und Technologien einführt, wie z.B. Fehlereindämmung zwischen Containern für sicherheitskritische Anwendungen, sowie echtzeitfähige Container-zu-Container-Kommunikation mit Latenzen und Jitter im niedrigen Mikrosekundenbereich. Eine maßgebliche Herausforderung besteht weiterhin darin, die Open-World-Annahme zu unterstützen und gleichzeitig Echtzeitgarantien und Fehlertoleranz zu bieten. Dies stellt zusätzliche Anforderungen an das Echtzeitsystem, wie beispielsweise die Fähigkeit einer dynamischen Umplanung von Echtzeit-Ressourcen. Diesen Anforderungen wird bspw. durch die Einführung eines Dienstes für die dynamische Umplanung von Echtzeit-Kommunikationsressourcen Rechnung getragen, welcher für Topologie- und Dienstmanagement, Zeitplanung, Konfigurationsbildung und Verteilung von Kommunikationsplänen verantwortlich ist. Auf diese Weise werden Änderungen innerhalb des physikalischen Modells (z.B. ein neuer Netzwerk-Switch oder ein neues Endgerät) und des logischen Modells (z.B. ein neuer Dienst) zur Laufzeit des Systems unterstützt.

Im Bereich der Software-Architekturen hat der Einsatz von Microservices in den letzten Jahren eine nennenswerte technische Reife erreicht. Die vorgeschlagene Architektur greift diesen Trend auf und führt Plattformdienste als Microservices ein. Schließlich werden mehrere Proof-of-Concept-Implementierungen vorgestellt und in verschiedenen Experimenten, die von einem realen Szenario bis hin zu Experimenten im Labor reichen, evaluiert, um zu zeigen, dass die vorgeschlagene Architektur für die Altenpflege in der Lage ist, den Wandel in der traditionellen Altenpflege hin zur Nutzung moderner Technologien aus dem Bereich CPSs zu bewältigen.

Acknowledgements

The acquisition of new knowledge and skills has always been a personal goal of mine. Writing this dissertation is therefore a big step for me to get closer to this goal. The completion of this dissertation would not have been possible without the advice, feedback and encouragement of many persons. I would now like to take the opportunity to thank these people.

First, I would like to express my deepest appreciation to my supervisor, Prof. Dr.-Ing. habil. Roman Obermaisser, for allowing me to pursue my research interests and for the possibility to write this dissertation. I am deeply indebted for his strong encouragement and advice during all phases of this dissertation. Further, I would like to thank Prof. Dr.-Ing. habil. Marcin Grzegorzek for his confidence in my work, his valuable constructive feedback and commitment as second reviewer of this dissertation.

I must also thank Stefan Otterbach for our constructive discussions and his support in helping me to organize all the hardware and tools required for a successful implementation of this dissertation. I also wish to thank Tobias Pieper for our discussion about Linux kernel technologies and topics related to real-time communication and scheduling. Special thanks also to my colleagues Veit Wiese and Simon Meckel for their helpful comments and encouragement while writing this thesis. I would also like to extend my sincere thanks to Michael Bohn, who has been a mentor and a friend to me during my time in the industry and who taught me how to solve complex problems and conveyed to me my enthusiasm to get to the bottom of every problem. Thanks also to Vinicius Costa Gomes from Intel for his great support in all aspects around the time-aware priority shaper in Linux.

I would also like to thank my whole family and especially my wife and my sons for their understanding and patience, which cannot not be underestimated. Finally, I would like to express my deepest gratitude to my parents, who have always been by my side and have always supported me.

Contents

List of Figures

List of Tables

Acronyms

AAL Ambient Assisted Living

ABS Anti-lock Braking System

AFDX Avionics Full DupleX Switched Ethernet (ARINC 664)

AMQP Advanced Message Queuing Protocol

APDU Application Protocol Data Units

API Application Programming Interface

ARINC615A ARINC 615A-3

ATS Asynchronous Traffic Shaper

AVB Audio Video Bridging

BAG Bandwidth Allocation Gap

BE Best-Effort

Bluetooth LE Bluetooth Low Energy

CBAS Constant BAndwidth Server

CBS Credit Based Shaper

CGM Continious Glucose Monitor

COTS Commercial of The Shelf

CPS Cyber Physical System

CSP Clock Synchronization Protocol

DetNet Deterministic Networking

DIM Domain Information Model

DLP Data Loading Protocol

DNS Domain Name System

DNS-SD DNS Service Discovery

DUN Dial-up Networking Profile

EDF Earliest Deadline First

EEE Energy-Efficient Ethernet

ETF Earliest TxTime First

FCU Fault-Containment Unit

FIFO First In First Out

FMEA Failure Modes and Effects Analysis

FSM Finite State Machine

GATT Generic Attribute Profile

GCL Gate Control List

GPS Global Positioning System

gPTP generalized Precision Time Protocol

HDP Health Device Protocol

HFSC Hierarchical Fair Service Curves

HID Human Interface Device Profile

HTB Hierarchical Token Bucket

HTTP Hypertext Transfer Protocol

IACS Industrial Automation and Control System

IEEE 1588 IEEE Standard for a Precision Clock Synchronization

IEEE 802.1AS TSN - Timing and Synchronization

IETF Internet Engineering Task Force

IoT Internet of Things

IPC Inter-Process Communication

JNI Java Native Interface

JSON JavaScript Object Notation

JVM Java Virtual Machine

JWT JSON Web Tokens

L2CAP Logical Link Control and Adaptation Protocol

LAN Local Area Network

LOC Lines Of Code

LTC Linux Traffic Control

LXC Linux Containers

MAC Media Access Control

MCAP Multi-Channel Adaptation Protocol

MDS Medical Device System

MDSD Model-Driven Software Development

MINT Minimum Inter-Arrival Time

MPLS Multiprotocol Label Switching

MQTT Message Queuing Telemetry Transport

NIS Network Information Service

NTP Network Time Protocol

OASIS Advancing open standards for the information society

OSGi Open Services Gateway initiative

OSI Open Systems Interconnection

OSI model Open Systems Interconnection model

PID Process identifier

PM QoS Power Management (PM) Quality Of Service Interface

PTP Precision Time Protocol

QDISC queueing discipline

QoS Quality of Service

RabbitMQ	RabbitMQ Message Broker
RC	Rate-Constrained
RDMA	Remote Direct Memory Access
REST	Representational State Transfer
RMS	Root Mean Square
RPC	Remote Procedure Call
RT-Broker	Real-Time Broker
RT-Client	Real-Time Client
RTT	Round-Trip Time
SCP	Secure Copy
SKB	Linux Socket Buffer
SLP	Service Location Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SoC	System-on-a-Chip
SPP	Serial Port Profile
SSH	Secure Shell
STPA	System Theoretic Process Analysis
TAPRIO	Time-Aware Priority Shaper
TAS	Time-Aware Shaper
TBF	Token Bucket Filter
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TLS	Transport Layer Security
TMR	Triple-Modular Redundancy
ToS	Type of Service
TSN	Time Sensitive Networking

TT Time-Triggered

TT-615A-Loader TT-615A-Loader

TTE-Build TTE-Build for TTEthernet

TTE-Plan TTE-Plan for TTEthernet

UDP User Datagram Protocol

UML Unified Modeling Language

UPnP Universal Plug and Play

URL Uniform Resource Locator

UTS Unix Time Sharing

UUID Universally Unique IDentifier

VL Virtual Link

VLAN Virtual Local Area Network

WSDL Web Services Description Language

XML Extensible Markup Language

Zeroconf Zero-configuration networking

1 Introduction

Aging of the population is an ongoing trend visible in the increasing share of elderly people in the structure of the population. According to the aging report of the European Commission from 2015 [Eur15], by 2030 around 25% of the EU population will be over 65 and the amount of people aged from 65 to 80 will rise by nearly 40% between 2010 and 2030. This demographic change in combination with the wish of elderly people for a self-determined life and the extensive costs for the assistance and care for elderly people led to an upcoming interest in systems for elderly care and Ambient Assisted Living (AAL). This change is also leading to a shortage of skilled workers in the field of elderly care.

In order to provide assistance in activities of elderly people in their everyday life, systems for elderly care and AAL are typically realized as Cyber Physical System (CPS) [HAR14][RLSS10]. The comprehensive survey of Geisberger et. al even see CPS as the only way for future AAL and elderly care applications [GB12]. New application fields in the area of elderly care and AAL like the automatic injection [Dan20] of insulin or robotic assistance [LMA+19] put additional requirements to these CPSs, like real-time support, dependability and support of an open-world assumption. These requirements turn the underlying CPS into a real-time system which has to meet deadlines. These systems are denoted as safety-critical systems which must prevent critical failures where the system poses unacceptable risks to human health or to the environment. In addition, the ongoing trend towards multi-core processors and highly integrated System-on-a-Chips (SoCs) provides CPSs with the capability to run a large variety of services and applications at the same time. This requires the use of mixed-criticality techniques, since applications with high criticality are executed concurrently with applications with low criticality. In addition, techniques for spatial and temporal isolation are required to limit the impact of faults.

The new fields of application further require the integration of complex heterogeneous sensors and actuators, such as a glucose meter or an insulin pump. These sensors often have to be integrated into the system at runtime, taking into account the open-world assumption that dynamic changes to the system are possible at any time. One possible scenario here is the use of a CPS to allow automatic detection of an emergency situation such as the fall of an elderly person. Further, the integration of different types of sensors needs to offer a high degree of autonomy in order to provide easy handling by the elderly person or by a caregiver. In addition, dynamic setups must be supported with variable numbers of users while supporting multiple sensors per user at the same time. In order to support everyday life, systems for AAL must not focus on the apartment or house of the elderly people. Rather, a AAL system must also support external scenarios, such as situations in which elderly people go for a walk or visit a doctor's office. These distributed AAL solutions requires a highly adaptive and technology-independent architecture.

This thesis will provide an architecture for elderly care which will provide flexibility, reliability and technology-independence by introducing a fault-tolerant, distributed message-oriented system architecture using a modern microservices approach. In a microservices based architectures, applications are built as a suite of small services instead of one monolithic application. Each service is realized as a small autonomous component with its own life-cycle, making the services independently maintainable and deployable. In comparison to a monolithic architecture, this has several advantages, such as the facilitation of technology diversity, better maintainability as well as an improved scalability of the distributed system. The proposed architecture is embracing this trend and introduces platform services as microservices. This includes services for service orchestration and discovery, fault-tolerance, real-time communication, temporal and spatial partitioning of resources, clock synchronization, sensor integration, maintenance as well as a service for the dynamic reconfiguration of real-time communication resources.

1.1 Objectives

The main objective of this dissertation is the definition of an architecture for elderly care that supports reliable and safe applications in addition to typical applications in AAL and elderly care such as daily activity reports or exercises for mental training. A prominent example for a safety relevant application is the automatic injection of insulin. In particular, safety critical applications put further requirements on the underlying system such as the adoption of techniques for fault-tolerance and the establishment of real-time communication. Here, the support for fault-tolerance is of utmost importance since the injection of insulin is safety-critical and can lead to dangerous situations in case of a failure. For example, the injection of too less insulin can result in a hyperglycemia which further can lead to vascular diseases or even cardiac dysfunction. Therefore, the system must be able to tolerate faults, like design faults in the application, hardware faults and even interaction faults caused by the user. In addition, safety critical applications like robotic assistance with tight control loops require the establishment of real-time communication.

This work aims to provide an architecture that is flexible and easy to use by application developers and also provides maintenance interfaces for administrators. This is particularly important because the success of an architecture also depends on broad approval and acceptance. The presented architecture for elderly care will extend the state-of-the-art with innovative technologies needed to meet the new challenges of demographic change in society. With this regard, this thesis presents a completely new architecture based on microservices incorporating state-of-the-art technologies from industry and research.

In particular, the presented architecture addresses the following challenges:

Fault-Tolerance Taking systems for the automatic injection of insulin as an example, the application of fault-tolerance techniques is mandatory in order to prevent catastrophic system failures that can lead to dangerous situations for users of the system, which are in particular elderly people. Further, the nursing staff and relatives might also be affected by the system failure. Fault tolerance techniques are also needed to prevent malicious applications from monopolizing network traffic, which could e.g. lead to unstable control loops. For example, in the above example of automatic insulin injection, messages may be lost due to an unstable control loop between the Continious Glucose Monitor (CGM) (which measure blood glucose levels) and the insulin pump. In the worst case, this can result in too little insulin being delivered.

Real-Time Support Critical control loops require an underlying real-time system that can provided the execution of task within a bounded limit of time. Furthermore, CPS for robotic assistance are typically realized as a distributed system, where the underlying communication network must provide a successful message transport as well as bounded limits for latency and jitter.

Multiple Integration Levels Distributed systems comprise multiple distributed services running concurrently across different physical machines that are connected by a communication network located at different integration levels like the Local Area Network (LAN) or the Internet. Depending on the integration level, different types of devices are involved in the communication and different techniques have to be applied in order to establish and maintain the communication between the services.

Open-World Assumption Many use cases in the field of elderly care require the integration of heterogeneous sensors. A prominent example is the use case of an artificial pancreas which is addressed by the project OpenAPS [Dan20]. Here, different types of CGMs and insulin pumps have to be supported in order to provide a solution for an artificial pancreas that complies with the needs of a broad range of different users. Further, the integration of sensors often has to be conducted at run-time demanding a dynamic reconfiguration of the system where other components have to be informed about the new configuration. This adaptability requirement that comes with the open-world assumption is also a huge challenge for real-time systems since typical real-time systems are scheduled in advance and do not support changes in the composition of the system at run-time. In order support the open-world assumption, a dynamic reconfiguration of real-time resources is required.

Heterogeneity of the Underlying Technologies When integrating sensors, a main challenge is the technological heterogeneity. For example, in the field of elderly care and AAL, complex medicals sensors like a pulseoximeter should be supported as well as simple sensors like a weighting scale. Medical and health device communication standards can help to cope with this challenge.

Mixed-Criticality In today's Cyber-Physical Systems and smart devices, there is an ongoing trend towards multi-core processors and highly integrated SoC allowing to run a large variety of services and applications at the same time on a single device. This imposes multiple challenges for systems that run applications with different levels of criticality. These so called mixed-criticality systems have to provide mechanisms for spatial and temporal isolation of the system resources in order to run applications with a high criticality concurrently with applications of low criticality. This isolation is required since concurrent applications can interfere each other. E.g. a low critical application might occupy the whole CPU which can lead to a starvation of a high-critical task. Further, techniques for the temporal and spatial isolation can provide a solid basis for modular certification [LO17] and e.g. can further prevent the elevation of criticality in the system.

1.2 Contributions

This thesis offers several contributions which extend the state-of-the-art of architectures for elderly care and AAL applications:

Modern Microservice-Based System Architecture This thesis promotes a service oriented architecture based on a microservices approach. The introduction of a best-effort communication service based on a modern message-based communication protocol is the fundamental building block to overcome the heterogeneity of the underlying technologies.

Real-Time Support in an Open-World Scenario One main contribution is the support of real-time applications in an open-world scenario. Typically, real-time systems are scheduled in advance and therefore do not allow the integration of new applications and components like network switches and service gateways at runtime. The proposed solution in this thesis extends the state-of-the-art by supporting the dynamic reconfiguration of real-time communication resources and applications in order to comply with the dynamic nature of elderly care scenarios.

Support for Mixed-Criticality Applications A further contribution of this thesis is the temporal and spatial partitioning of critical resources in order to establish isolation among applications with different criticalities. This allows to run applications with high-criticality besides applications with low-criticality without any interference.

Sensor Integration with ISO/IEEE 11073 A major contribution of this work is the large variety of possibilities for integrating sensors. This is particularly achieved by the integration and extension of ISO/IEEE 11073, which is a standard for medical and health device communication. Most notably is the extension of the standard in order to work with a modern microservice based architecture and Bluetooth LE devices.

Reliable Real-Time Container-To-Container Communication This thesis further presents a hierarchical real-time communication approach which includes a solution for real-time communication between containers on the same service gateways and between containers on different service gateways.

1.3 Structure of this Thesis

This thesis is structured as follows. Since the definition of the presented architecture requires knowledge in a broad range of topics such as aspects of dependability, service orchestration, real time operating systems and networks, the introductory Chapter 2 describes comprehensively the necessary basis concepts and technologies, which are necessary to understand the later definitions and design of a fault-tolerant and real-time capable architecture for elderly care. The starting point is the introduction of real-time systems followed by the presentation of real-time communication systems and real-time operating systems. Further, the concept of dependability is introduced and different threats to dependability as well as means to achieve dependability are presented.

Afterwards, Chapter 3 introduces the key challenges which a fault-tolerant real-time architecture for elderly care has to address before presenting relevant state-of-the-art architectures for elderly care and AAL. The discussion about state-of-the-art architectures is followed by Chapter 4, which introduces the building blocks and the fundamental concepts of the fault-tolerant architecture presented in this thesis. This is supplemented by a requirements analysis of the underlying technologies. Afterwards, the different platform and system services of the presented architecture are introduced and the fault assumptions of this thesis are presented. The following Chapter 5 gives an overview about the services with the respective models and algorithms, which includes services for service orchestration and discovery, fault-tolerance, best-effort and real-time communication as well as clock synchronization and the dynamic reconfiguration of real-time communication resources. Subsequently, Chapter 6 describes the implementations that were established to evaluate the architecture. The purpose of Chapter 7 is then to present and discuss the evaluation using several scenarios and experimental setups. Finally, this thesis is concluded by Chapter 8 which summarizes the results and presents potential future work.

2 Basic Concepts and Technologies

The purpose of this introductory chapter is to explain the basic concepts and technologies that are used within this thesis. The presented concepts and technologies cover a broad range since the definition of an architecture for elderly care has to address a lot of different topics, such as aspects of configuration, service orchestration, operating systems and networks. Therefore, the methodology of this chapter is as follows: First, the definition and the basic ideas of the concept of the technology is introduced. Afterwards, the aspects of the introduced technology relevant to this work are highlighted and explained in detail. The starting point of this chapter is the introduction of real-time systems followed by the presentation of real-time communication systems and real-time operation systems. All of them are often used in combination to build distributed and dependable real-time systems where dependability refers to the attributes of a system regarding it's reliability, safety, maintainability, availability and security which are introduced in the next section of this chapter. Subsequently, medical and health device communication standards like ISO/IEEE 11073 and Bluetooth Health Device Protocol (HDP) are introduced. A further important technique for the architecture of this thesis is afterwards presented, which is the virtualization of hardware resources like CPU, memory and network resources. Finally, several architectural patterns like service discovery and Microservices are introduced and explained.

2.1 Real-Time Systems

This section will introduce the topic of real-time systems. Systems that perform safety-critical tasks are often real-time systems, such as the Anti-lock Braking System (ABS) in a car or the fly-by-wire control in an aircraft. Within real-time systems, the correctness of the behavior does not only depend on the logical result of the computations, but also on the physical instant in time at which the results are produced [Kop11a]. That means in particular that a real-time system has strict temporal requirements in addition to its functional requirements and results have to be available within a bounded period of time. The instants in time when results have to available are called deadlines (c.f. Section 2.1.1) and missing these deadlines can lead to catastrophic accidents or even the in loss of life of human people. Real-time systems that are distributed over a set of computing and/or sensor nodes are typically interconnected by a real-time communication network (c.f. Section 2.2).

2.1.1 Classification of Real-Time Systems

Real-time systems are classified by several attributes that describe their behavior in different situations.

One of these attributes is the temporal behavior of the system regarding deadlines where a deadline is the instant in time when a (computational) result must be available. If the result is useful even after the deadline has passed, the deadline is called a soft deadline. If severe consequences like the harm of a human life can result in missing the deadline, the deadline is called a hard deadline. Otherwise it is called a firm deadline.

Systems that are able to meet hard deadlines are called hard real-time systems. The design of hard real-time systems completely differs from the design of soft real-time systems like for example a video-streaming system. Hard real-time systems often have to provide a response time (e.g. for a computation) that is within the order of a milliseconds or less whereas soft real-time systems have a response time that is in the order of seconds. Another difference is the behavior of the system in peak-load scenarios. A hard real-time system has to provide predictable behavior even at the peak-load of a system. In contrast, in soft real-time systems a degraded performance is tolerable.

Real-time systems that carry out a safety-critical function are further classified regarding their behavior after a failure. This behavior has already be defined during the design phase of the system. If a safe state can be identified that the system can enter in case of a failure, the system is called fail-safe. A prominent example of a fail-safe system is a traffic light control. Whenever there is a failure in the system, the system can enter a safe state where all signals are changing to red which prevents accidents by preventing cars to enter the crossing accidentally. If no safe state can be identified that the system can enter in case of failure, the system has to preserve a minimum functionality. That means that the system must remain functional even in case of a failure. One such system is the flight control system in an aircraft, where the pilot must still be able to land the aircraft safely in case of a failure.

A further attribute that is used to classify real-time systems is the location of control. In *event-triggered* systems, decisions and actions are based on sporadic events like external triggers (e.g. interrupts in a micro-controller) or the arrival of a control message on the communication network. In a *time-triggered* real-time system, all activities are carried out at predefined periodic instants. The instants are determined by the clock ticks of a real-time clock that is under the internal control of the real-time system.

2.1.2 Temporal Requirements

As noted, real-time systems have to comply with (stringent) temporal requirements. These temporal requirements mostly depend on the requirements of the control loops executed by the real-time systems. However, temporal requirements can also arise from the user interface. Nevertheless, the time constraints in this case are not as tight as, for example, in the control system of a nuclear power plant. If a real-time system is used to control an object based on the information of a sensor another requirement comes into place: minimal latency jitter. Jitter denotes the variability in the control loop latency regarding its timeliness. The main problem here is that jitter by its definition is not deterministic and thus cannot be compensated automatically. Therefore latency jitter can lead to degradation of the service quality of the control loop or even to a complete failure of the control loop.

2.2 Real-Time Communication Networks

The focus of this section is real-time communication in Ethernet based networks. Real-time communication systems are required whenever the need for packet transmission has to happen within tight timing constraints like in industrial or automotive control applications. At present, real-time communication systems also become increasingly important in the field of bio-medicine. For example, consider robotic systems for surgery that already assist surgeons during their work. According to [KLK13], in the near future it will “be possible to perform surgical procedures that are limited only by available communication technologies even at extreme distances between the surgeon and the patient by computerized mediation of the surgeon’s actual hand motions to the surgical instruments affecting the patient’s tissue”.

The main requirements of real-time communication networks and the corresponding infrastructure are “low protocol latency with minimal jitter, the establishment of a global time base, fast error detection at the receiver and the need for temporal error containment by the communication system” itself [Kop11a]. Regarding this, prioritization of packets is not sufficient because the transmission of a frame with lower priority that is already in transmission will first be finished before a frame with a higher priority can be sent. This can lead at every hop to additional message transport delay and accumulates indeterministically to a high amount of time. Therefore, real-time communication networks are typically based on scheduled traffic. This section will introduce to the basic concepts of scheduled traffic and clock synchronization before continuing with the introduction of two state-of-the-art real-time communication protocols TTEthernet and Time Sensitive Networking (TSN). Further, the results of the Deterministic Networking (DetNet) task group of the Internet Engineering Task Force (IETF) relevant for this work are presented.

2.2.1 Scheduled Traffic

In real-time communication systems, low protocol latency alongside with minimal jitter is typically achieved by the establishment of scheduled traffic where packets are injected to the communication network at predefined instants in time. These communication networks are also often called time-triggered networks. Minimal protocol latency is required in order to keep the dead time of critical control loops as short as possible. The dead time of a control loop is the interval between the observation of a significant state variable and the beginning of the reaction of the system to the observed state variable. Jitter in the communication network affects the dead time of the control loop and reduces the quality of service if not being compensated correctly.

In time-triggered networks it has to be assured that the transmission of non-critical communication traffic has to be finished sufficiently far in advance to the time-critical traffic in order to not affect the time-critical traffic. The most common way is to establish a guarding band in advance to the transmission window of the time-critical traffic. Typically, the length of the guarding band is as long as it takes to transmit a frame with a maximum size. However, a more efficient approach is to evaluate the length of the packets in the queue that are going to be submitted next and choose a guarding band length accordingly.

The scheduling of time-triggered networks requires knowledge about the network structure and communication requirements of the services running on the end systems. This knowledge is typically represented in a network description document. The network description should abstract from the specific hardware and must define global properties like the topology of the network which includes all communication nodes (switches and end systems), the cabling and the properties of the communication links. Further, the network description may contain information about redundancy and fault-tolerance requirements of the network nodes. This network description is then used to create the communication schedule of the network. Ideally, the network description is available in a machine-readable markup language like Extensible Markup Language (XML) which can be automatically checked for syntactical (document is well-formed) and semantic (document is valid) correctness.

In order to prevent faulty communication nodes from sending packets at non-scheduled time points (e.g. in the event of a "babbling idiot" failure), time-triggered communication networks usually use guardians at ingress ports where the guardians just allow the passing of messages during the scheduled time points. This is possible because of the a-priori knowledge provided by the communication schedule. In contrast, this is not possible in event-driven communication networks such as standard Ethernet, because the location of control is not located within the communication network. Instead, the transmission of the packet lies within the control of the application or is triggered by external events.

2.2.2 Clock Synchronization

In real-time networks the communication activities of all nodes are often synchronized in order to coordinate the behavior of the communication system. This is achieved by the establishment of a global time base. However it is not sufficient to synchronize the clocks only at the startup of the communication network. Furthermore, clocks have to be synchronized at periodic intervals. That is because each physical clock has an individual drift rate. The drift rate also changes during the lifetime of the clock by aging of the crystal oscillator and is further influenced by changes in the environment like a change in the ambient temperature. Also the addition, removal or failure of communication nodes or changes in the network configuration (e.g. new time schedules) influences the resynchronization process.

In order to synchronize the clocks, several clock synchronization protocols are available like IEEE 1588 [IEE08b] and 802.1AS (Timing and Synchronization for Time-Sensitive Applications). The Precision Time Protocol (PTP) defined in IEEE 1588 provides precise synchronization of clocks in measurement and control systems in the sub-microsecond range with a high degree of accuracy and precision at low network and local clock computing resources requirements [IEE08b]. The clock synchronization is organized into a master-slave synchronization hierarchy where the clocks of the slaves are synchronized to the clocks of the masters. The clock at the top of the hierarchy is called the grandmaster clock and determines the reference clock for the whole communication system. The grandmaster clock itself can be for example synchronized with an external time source like the Global Positioning System (GPS). PTP further recommends that timestamps are generated as close as possible to the network interface (e.g. at the physical layer (PHY)), which helps to reduce the jitter in IEEE 1588 synchronized communication networks. Detailed information about PTP is covered by the book of John C. Eidson [Eid06].

The generalized Precision Time Protocol (gPTP) defined in 802.1AS (Timing and Synchronization for Time-Sensitive Applications) is based on IEEE 1588. However, there are several fundamental difference to IEEE 1588. The most crucial difference of gPTP is that it only differentiates between end stations and bridges while PTP differentiates between the different clock types in the system like ordinary, boundary or end-to-end transparent clocks. Another important difference is that gPTP systems can only communicate with other systems directly that implement gPTP. That also means that there can be no bridges in the system that are not time-aware according to gPTP like it is the case in IEEE 1588. Further differences are explained in the IEEE 1588 standard [IEE08b].

2.2.3 TTEthernet

TTEthernet implements the time-triggered communication paradigm by introducing deterministic communication with bounded latency to the switched Ethernet standard IEEE 802.3. Large parts of TTEthernet are standardized in SAE AS6802 (Society of Automotive Engineers) [AS611]. SAE AS6802 establishes deterministic communication by introducing synchronous time-triggered frame transport besides non time-triggered asynchronous frame transmission as it is the case for standard best-effort Ethernet frames. In addition to time-triggered synchronous traffic, the standard defines an algorithm for clock synchronization that allows the synchronization of the clocks down to a sub-microsecond precision. Further, TTEthernet introduces fault-tolerant communication by providing redundant channels at the switches. While end systems can also be standard Ethernet devices, all switches have to be switches that implement the TTEthernet standard. This is required in order to protect the communication system against end systems that are sending messages outside their specified time interval (babbling idiot failure).

TTEthernet classifies the network traffic into three different classes: Time-Triggered (TT), Rate-Constrained (RC) and Best-Effort (BE) [Obe11]. In TTEthernet, TT messages are sent at predefined instants of times and can be used when tight latencies with minimal and bounded jitter are required. Unlike TT messages, RC messages are not sent by a predefined schedule. Instead, sufficient bandwidth is allocated in order to provide upper bounds for latency and jitter [Obe11]. However, these bounds are larger than those which are provided by the TT message class. Finally, BE messages use the remaining bandwidth and do not provide any guarantees for delivery or timeliness. In order to resolve conflicts (e.g. when TT messages becomes ready while a RC message is already in transmission), TTEthernet provides three conflict resolving methods, namely preemption, timely blocking and shuffling. A comprehensive discussion about timely blocking techniques is provided by [AO17].

For uploading of schedules and configurations to the end-systems and switches, TTEthernet uses ARINC615A. ARINC 615A is an avionics standard that defines a Data Loading Protocol (DLP) which can be used to load software into network switches or end-systems that support the ARINC 615A standard. The latest version of the standard is ARINC 615A-3 which basically uses Trivial File Transfer Protocol (TFTP) as the underlying protocol. ARINC 615A defines three different modes of operation: (1) information retrieval, (2) data uploading and (3) data downloading. The first mode can be used to get information about the hardware and further to query the current configuration of the target. The second mode can be used to upload new software or configuration files to the target hardware. The last operation mode can be used for example to download files from the target hardware.

2.2.4 Time Sensitive Networking (TSN)

TSN is a family of standards that is still under development by the TSN task group which is a part of the IEEE 802.1 working group. The main goal of TSN is to achieve deterministic latency bounds within 802.1 networks which allows to run real-time applications with hard deadlines on top of these networks. The TSN task group has evolved from the former 802.1 Audio Video Bridging (AVB) task group whose goal was to improve the synchronization for switched Ethernet networks in combination with low-latency and high reliability. However, AVB was designed to achieve low latency in contrast to TSN which achieves deterministic latency bounds which are required for dependable real-time applications.

In order to achieve deterministic latency bounds, TSN introduces within the standard 802.1Qbv (Enhancements for Scheduled Traffic) the concept of the Time-Aware Shaper (TAS). The basic idea of the TAS defined in 802.1Qbv is to block non time-sensitive traffic in reserved time intervals to have idle time for sending time-sensitive traffic. The TAS allows the transmission of packets in an end station or a bridge by predefined time schedules (which is called scheduled traffic or protected traffic) and supports up to eight different traffic classes. According to this, TSN defines eight transmission queues per port where each transmission queue has one transmission gate that determines if a frame from the corresponding queue can be selected for transmission. The transmission queues are under the control of a Gate Control List (GCL) which contains a timely ordered list that defines the instant in time when a gate is open or closed for a particular queue. Figure 2.1

shows the relationship between GCL, transmission queues and transmission gates. The processing of the GCL leads to so called gate-close and gate-open events on every queue.

The instants at which the events occur can be determined from the sequence of the gate events in the GCL. Further, the GCL is processed in gating cycles which define the "period of time over which the sequence of operations in a GCL repeats" [IEE16a].

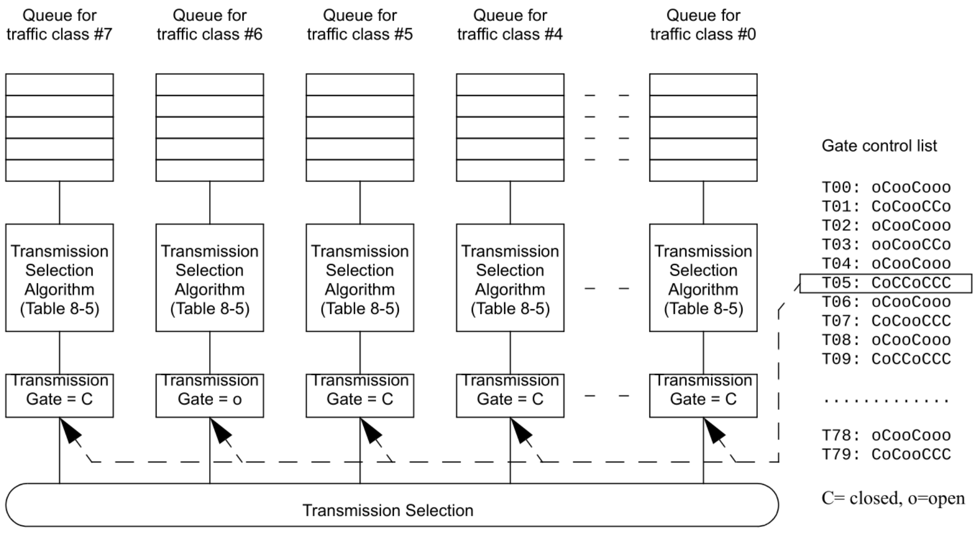


Figure 2.1: Transmission selection in TSN with Transmission Gates [IEE16a]

As already mentioned, the gate operations at the transmission gates are controlled by the GCL. However, the GCL and the distinct gate operations at each port are controlled by three different state machines, which are the cycle-timer state machine, the list-execute state machine and the List Config state machine. The cycle-timer state machine initiates the execution of the GCL and ensures that the gate cycle time for the associated port is maintained [IEE16a]. The list-execute state machine executes the gate operations defined in the GCL in sequence and ensures an appropriate delay between each operation. Finally, the list configuration state machine is responsible for all actions required when a new schedule is applied (e.g. updating the current schedule, stopping and starting of the other state machines). Further, the TAS allows to open more than one gate at the same time [IEE16a] by a schedule. In this case, the transmission selection algorithm will refer to a second scheduling algorithm like priority queuing or the Credit Based Shaper (CBS) from the TSN predecessor AVB.

However, the TAS from TSN only targets cyclic traffic and requires precise clock synchronization among all participants. For other traffic types besides cyclic traffic, TSN introduced the Asynchronous Traffic Shaper (ATS) in IEEE 802.1Qcr. In comparison to TAS, ATS does not require any clock synchronization or reference clock. A good insight about ATS is given in [ZBRY19]. A detailed comparison between TAS and ATS is provided by [NTA+19].

The upcoming TSN standard does not put any needs on end-systems that just receive scheduled traffic to implement the TAS defined in IEEE 802.1Qbv. However, if the node is part of a critical control loop, the clock of the node has to be synchronized with all other clocks. A prerequisite for using the TAS of TSN is to have all clocks synchronized by the substandard 802.1AS (Timing and Synchronization for Time-Sensitive Applications) that is explained in Section 2.2.2.

For reliability and error detection, TSN further introduces IEEE 802.1CB (Frame Replication and Elimination for Reliability). For that purpose, IEEE 802.1CB defines mechanisms for identification (e.g. by destination address or VLAN identifier) and replication of packets for redundant transmission (over multiple paths), identification of duplicate packets, and elimination of duplicate packets [IEE17].

Summarized, TTEthernet and TSN use the same concepts and are based on schedule traffic. However, in contrast to TTEthernet, which requires dedicated TTEthernet switches, TSN will be available in standard Ethernet hardware in the near future. A comprehensive comparison between TTEthernet and TSN can be found in [ZHLL18].

2.2.5 Deterministic Networking (DetNet)

The DetNet Working Group as part of the IETF focuses on deterministic networks with latency guarantees and ultra-low packet loss beyond LAN boundaries. This includes in particular "data paths that operate over layer 2 bridged and layer 3 routed segments, where such paths can provide bounds on latency, loss, and packet delay variation (jitter), and high reliability" [iet15]. DetNet operates at the IP layer and delivers service over lower-layer technologies such as Multiprotocol Label Switching (MPLS) and TSN as defined by IEEE 802.1 [FTVF19]. The envisioned use cases of DetNet show that there is a very high demand on such systems even beyond the "classical case of Industrial Automation and Control Systems (IACSs)" [Gro19].

The primary goals of DetNet are minimum and maximum end-to-end latency from source to destination with timely delivery and bounded jitter, bounded packet loss ratio and bounded out-of-order packet delivery [FTVF19]. In order to achieve these goals, the DetNet Working Group proposes three different techniques, which are *resource allocation*, *service protection* and *explicit routes*. The allocation of resources alongside the path of a DetNet flow includes for example the reservation of sufficient buffer space in the routers. This is very important to avoid buffer congestion which could lead to packet loss due to contention.

Further, resource allocation allows to establish a maximum end-to-end latency. In order to protect services, DetNet recommends several techniques to handle random media errors and equipment failures. One of these techniques is the establishment of redundancy by introducing packet replication and elimination for DetNet flows. Lastly, the paths that DetNet flows use are typically explicit routes which avoid the interruption of the flows by the convergence of routing or bridging protocols [FTVF19].

2.3 Real-Time Operating Systems

This section gives an overview of the basic concepts and fundamental properties of real-time operating systems. Unlike general purpose operating systems such as Linux or Windows, a real-time operating system is designed for real-time applications that place stringent requirements on timing. A Real-time operating system is often the basis for a real-time system (cf. Section 2.1).

Real-time operating systems are usually designed with the following capabilities: minimum interrupt latency, short critical regions and preemptive task scheduling [Wan17]. Interrupts in a real-time operating system are the most crucial part since in an interrupt-driven software system, a transient error on the interrupt line may upset the temporal control pattern of the complete node and may cause the violation of important deadlines [Kop11a]. Therefore, real-time operating systems must provide mechanisms for continuously monitoring the (minimal) time intervals between subsequent interrupts. Short critical regions such as access to shared data objects are another challenge for real-time operating systems, where a non-critical task could cause a critical task to fail if the critical regions are not left in time. Preemptive task scheduling where a non-critical task can be preempted by a critical task is the most frequently used instrument to ensure that deadlines (cf. Section 2.1.1) are met.

Regarding the temporal behavior of real-time operating system, Kopetz [Kop11a] provides a more precise definition: a real-time operating system is an operating-system where the worst-case administrative overhead regarding its temporal performance is known a priori. Based on this knowledge, the temporal behavior of the whole system can be determined. The exact knowledge of the temporal performance of the underlying operating system is required to implement predictable services that can execute a specific task within a bounded amount of time without missing its deadline. Further, “a real-time operating system must provide a predictable service to the application tasks such that the temporal properties of the complete software in a node can be statically analyzed” [Kop11a]. That is one of the reasons why real-time operating systems are typically based on a small kernel with a limited number of Lines Of Code (LOC). These lightweight real-time systems can be further distinguished regarding their kernel type. FreeRTOS [Ser17] for example uses a micro-kernel which means that all real-time applications are running in the user space with different priorities. Zephyr, which is a project from the Linux Foundation follows a different approach and uses a monolithic kernel where all applications are running in kernel mode besides kernel components like device drivers and power management.

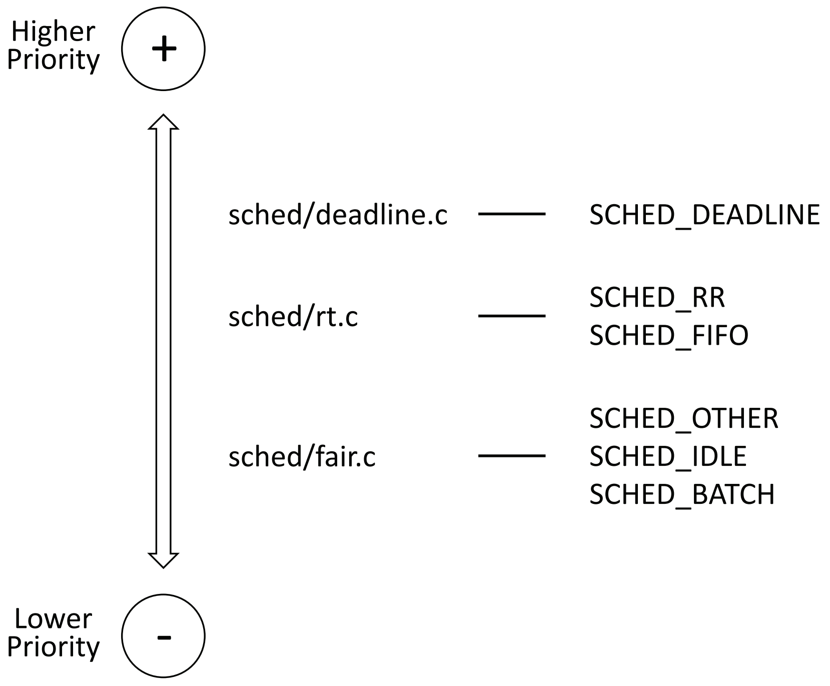


Figure 2.2: Hierachy of Linux Task Schedulers

However, there are also general purpose operating systems available that provide deterministic timely behavior. Taking Linux as an example, there are currently two competing solutions available. The first approach is a patchset called PREEMPT_RT which makes the Linux kernel preemptive for applications running in the user space. This reduces the affects of kernel tasks on real-time tasks. The second approach is SCHED_DEADLINE, which a deadline scheduler that is available since Linux kernel version 3.14. The SCHED-_DEADLINE scheduler adds a real-time policy to the kernel which has even a higher priority than the POSIX-compliant real-time fixed-priority scheduling policies SCHED_FIFO and SCHED_RR (cf. Figure 2.2). SCHED_DEADLINE allows predictable task scheduling by implementing both the Earliest Deadline First (EDF) and Constant BAndwidth Server (CBAS) algorithms. CBAS allows reservation based scheduling while EDF takes care that tasks with the earliest deadline gets executed first. Each real-time task can run for a maximum runtime Q_i every period T_i as long as the total utilization $\sum_{i=1}^{N} \frac{Q_i}{T_i}$ of all real-time tasks is below a certain threshold [LSAF16].

2.4 Dependability

This section introduces to dependability, which is a generic concept that describes the "ability of a system to avoid failures that are more frequent and more severe than it is acceptable" [ALRL04]. The dependability of a system is defined by its attributes which are reliability, availability, maintainability, integrity and safety. In recent scientific discussions the security of dependable systems gets a more prominent role. This adds a further attribute to dependability, which is confidentiality. After introducing these five attributes, this chapter will discuss the threats to dependability, which are faults, errors and failures. Subsequently, two special means of achieving reliability are explained, which are fault tolerance and fault prevention.

2.4.1 Dependability attributes

In the state-of-the-art dependability is determined by its attributes, which are reliability, availability, maintainability, integrity, safety and confidentiality. The first attribute **reliability** denotes the probability that a system will provide the required service for a specified amount of time, given that the system was fully operational at the beginning. The second attribute **availability** is a system property that measures the ability of the system to provide the correct service and is denoted by the ratio of correct and incorrect service. Next property is **maintainability**, which is a measure for the capability of the system to be kept or reset in an operational status. Further, **integrity** describes the freedom from improper changes within the system such as the corruption of data. The last attribute **confidentiality** comes into place when addressing the security of a system. Confidentiality describes the extent to which information is not disclosed without authorization. Having this in mind, **security** can be seen as a meta attribute and is defined a "composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of 1) availability for authorized actions only, 2) confidentiality, and 3) integrity with 'improper' meaning 'unauthorized' " [ALRL04]. The last attribute of dependability is **safety** which describes the reliability regarding critical failures where the system carries unacceptable risks to human health or to the environment.

2.4.2 Threats to Dependability

Avizienis et al. [ALRL04] defined the fundamental concepts of fault, error and failure in dependable systems and pinpointed them as the major threats to the dependability and security of the system. A **failure** of the system is an event that occurs when a service that the system delivers deviates from the intended behavior (correct service). The way in which the deviation from the correct service occurs and how the failure is manifesting can have different forms which are called failure modes. A failure is the result of an **error**, which is an unintended state of the system. The cause of an error is called a **fault**. Figure 2.3 shows the causal chain between fault, error and failure. Examples for faults are hardware faults (e.g. a hardware defect in a processor), natural faults (e.g. voltage peaks in the power supply through electromagnetic interference) and developments faults during the system development like race-conditions in the developed algorithms. A comprehensive taxonomy of faults, errors and failures can be found in [ALRL04].

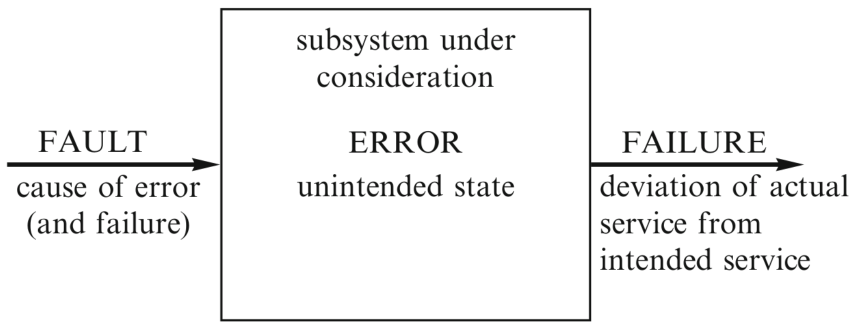

subsystem under consideration
FAULT
cause of error (and failure)
ERROR
unintended state
FAILURE
deviation of actual service from intended service

Figure 2.3: Threats to Dependability - Faults, Errors and Failures [Kop11a]

An error within one component can propagate to other components. In the worst case this can lead to a performance degradation or even a failure of the entire system. Also, a failure can cause faults in other components which in turn can lead to a failure as well. Therefore it is important to apply techniques for fault-containment.

2.4.3 Means to achieve Dependability

Dependability can be achieved by different means. **Fault prevention** can help to mitigate faults already at the beginning of the design of the system. A typical technique for fault prevention is encapsulation in object-oriented languages, where, for example, a mutator method checks the value for validity before the new value is assigned to a variable. Another fault prevention technique is Continuous Testing which is often combined with Continuous Integration. **Fault tolerance** is another mean to establish dependability. Fault tolerance targets at avoiding failures by first detecting errors and provide mechanisms to recover to a system state that has no error or fault (which could be activated again) [ALRL04]. For example, redundancy like Triple-Modular Redundancy (TMR) is a frequently used mean to detect an error. Further techniques can be used to recover from an error, such as rollback (return to a saved system state without error) and roll-forward (move forward to a system state without the error).

2.5 Medical and Health Device Communication Standards

Interoperability with a standard for connected health devices offers a great benefit because interoperability enables the integration of a compatible sensor with less effort at the user level. With this regard, this chapter introduces two important standards for connected health devices: ISO/IEEE 11073 and Bluetooth HDP.

2.5.1 ISO/IEEE 11073

The ISO/IEEE 11073 family of standards provides interoperability for “medical devices as well as for health and fitness devices which are used by professional healthcare organizations or by users at their home” [IEE12]. Figure 2.4 depicts an overview of the ISO/IEEE 11073 family of standards.

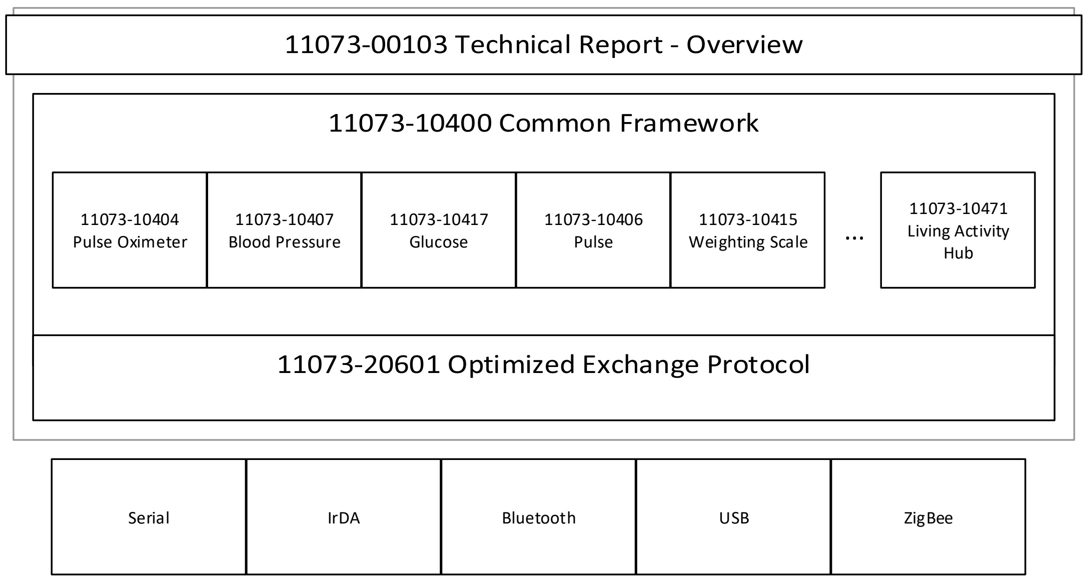

11073-00103 Technical Report - Overview
11073-10400 Common Framework
11073-10404 Pulse Oximeter
11073-10407 Blood Pressure
11073-10417 Glucose
11073-10406 Pulse
11073-10415 Weighting Scale
...
11073-10471 Living Activity Hub
11073-20601 Optimized Exchange Protocol
Serial
IrDA
Bluetooth
USB
ZigBee

Figure 2.4: IEEE 11073 Family of Standards - Overview

ISO/IEEE 11073 distinguishes between two device types: agents and managers. Agents are data sources, typically sensor devices like weighting scales or an ECG. Managers are data sinks, collecting the data from the agents. ISO/IEEE 11073 assumes limited computational power and limited energy for agents, therefore the communication model of ISO/IEEE 11073 puts the main burden for processing data on the managers. ISO/IEEE 11073 is transport agnostic, which means, that it can be used with almost any packet-based communication protocol like Bluetooth or ZigBee.

Device Specialization Standards In order to support a variety of sensor classes, the ISO/IEEE 11073 family of standards defines appropriate device specialization standards. The ISO/IEEE 11073-104xx device specialization standards are available for common device types like pulse oximeter (ISO/IEEE 11073-10404), glucose meter (ISO/IEEE 11073-10417) or insulin pump (ISO/IEEE 11073-10419).

Optimized Exchange Protocol One central part of ISO/IEEE is the Optimized Exchange Protocol, which is standardized in ISO/IEEE 11073-20601. This standard describes the structure and the behavior of agents and managers by defining three different models: a domain information model, a service model and a communication model.

The Domain Information Model (DIM) describes a set of atomic parts and shows how these are put together to form a larger element that might be used in a piece of software [IEE14]. The Medical Device System (MDS) is used to describe the structure of agent devices and is modeled as a set of objects where objects have a corresponding class and can have hierarchies. Objects in turn have attributes that contain for example measurement data. Depending on the corresponding class, attributes can be mandatory for objects. Figure 2.5 shows an exemplary DIM of a peak expiratory flow monitor modeled as a UML object diagram. The MDS object at the top represents the peak expiratory flow monitor itself which has four numeric objects *PEF (Peak Expiratory Flow)*, *FEV1 (Forced Expiratory Volume of a subject under forced conditions at 1 second)*, *FEV6 (Forced Expiratory Volume of a subject under forced conditions at 6 seconds)* and *Personal Best* besides one enumeration object *Reading Status*.

The service model defines the interaction with objects and their attributes. The interactions are modeled as data exchange services where agent and manager exchange messages using the Abstract Syntax Notation One (ASN.1) in order to define the data structures within the messages. The service model differs between association services and object access services. Association services are used to build up and release a logical connection whereas the object access services are used to access attributes defined in the MDS.

The communication model describes the communication between agents and managers by defining their corresponding state machines and valid interactions within each state. It also describes how the messages corresponding to each service are encoded and decoded to and from binary format to be transmitted over the transport channels [IEE12].

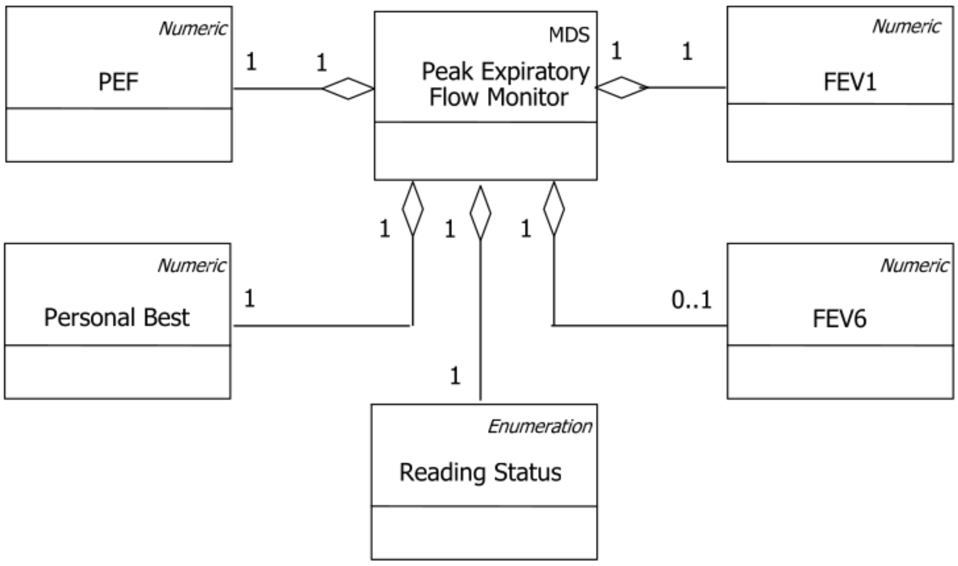


Figure 2.5: Peak expiratory flow monitor - Domain Information Model [IEE14]

State Machines Like most protocols, the ISO/IEEE 11073 session layer is controlled by a state machine. Most states and state transitions of the manager state machine are symmetrical to the elements described below for an agent. Figure 2.6 shows the state machine of an agent according to the Optimized Exchange Protocol.

The basic states for the agent are:

- *Disconnected.* When an agent is turned on for the first time, the state of the agent is Disconnected. This means that there is no connection between the agent and the manager. An agent can return to the disconnected state after a connection has been established if the connection was intentionally terminated or unintentionally disconnected.
- *Connected.* When a connection is established between the agent and the manager, the agent changes its state to Connected and remains in this state as long as the connection exists. The agent then changes to the Unassociated state.
- *Unassociated.* The agent is in this state if it has no application layer association with a manager. This can be caused by the fact that a new connection has just been set up, the manager rejects an association, or a member of the active association

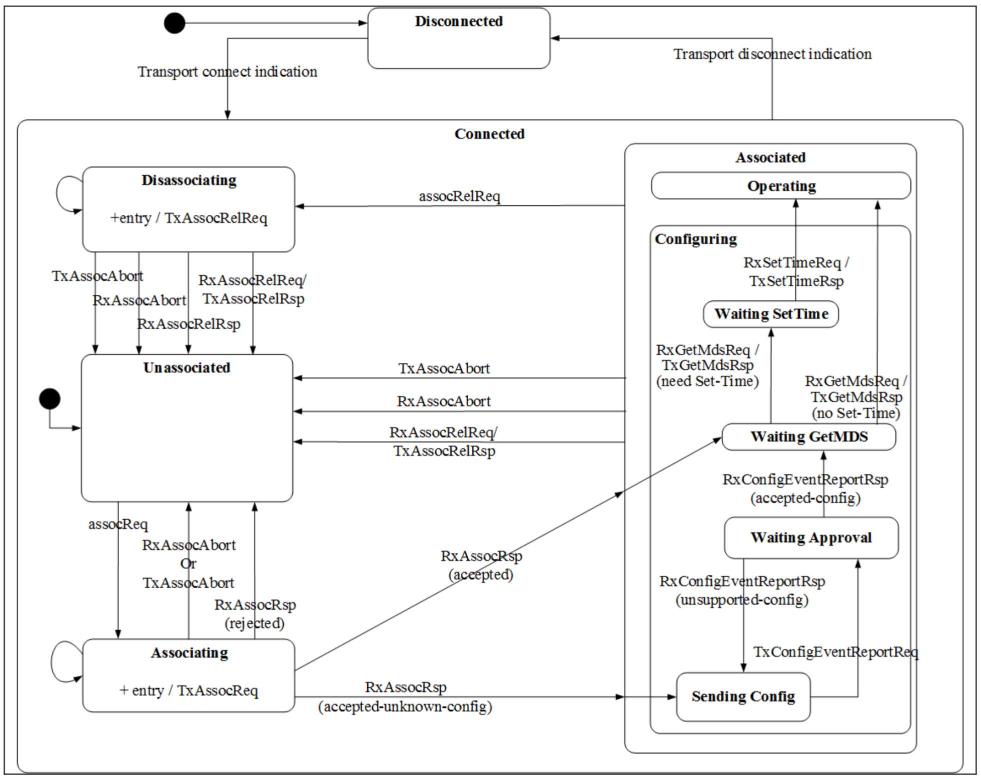

Disconnected
Transport connect indication
Transport disconnect indication
Connected
Associated
Operating
Disassociating
+entry / TxAssocRelReq
assocRelReq
Configuring
RxSetTimeReq / TxSetTimeRsp
Waiting SetTime
TxAssocAbort
RxAssocAbort
RxAssocRelReq/ TxAssocRelRsp
RxAssocRelRsp
Unassociated
TxAssocAbort
RxAssocAbort
RxAssocRelReq/ TxAssocRelRsp
RxGetMdsReq / TxGetMdsRsp (need Set-Time)
RxGetMdsReq / TxGetMdsRsp (no Set-Time)
Waiting GetMDS
RxConfigEventReportRsp (accepted-config)
Waiting Approval
assocReq
RxAssocAbort Or TxAssocAbort
RxAssocRsp (rejected)
RxAssocRsp (accepted)
RxConfigEventReportRsp (unsupported-config)
TxConfigEventReportReq
Associating
+ entry / TxAssocReq
RxAssocRsp (accepted-unknown-config)
Sending Config

Figure 2.6: Peak expiratory flow monitor - Agent State Machine [IEE10]

terminates it. The agent remains in the unassociated state until it starts associating with the manager.

- *Associating.* When an agent starts an association with the manager, the agent switches to the associating state and sends a connection request to the manager. If an association fails, an agent can request a new association with new association parameters.
- *Associated.* If the manager determines that the agent and the manager are using shared versions and protocols, the association is accepted. In doing so, the manager sends an association response to the agent. When the agent receives this message, it switches to the associated state and remains in this state until the agent sends or receives a release or termination request for association. The initial sub-state depends on whether the manager has responded to the association request with an indication that the configuration of the agent is recognized or not.
- *Operating.* If a manager already knows the configuration of an agent, the agent switches to the operating state. If the configuration is not known, the configuration is transferred alternatively.

- *Configuring.* If the manager does not know the configuration of the agent, it first accepts the association, but then reports to the agent that the configuration must be transferred. The agent remains in the configuring state until the agent transfers the configuration information and the manager confirms the configuration.
- *Disassociating.* When the agent determines that it should release the current association, the agent switches to the disassociating state and sends a request to the manager to release the association. In the event of a timeout, the agent sends an abort request and switches to the unassociated state.

2.5.2 Bluetooth HDP

Bluetooth HDP is Bluetooth profile that aims to ensure interoperability between connected health devices based on Bluetooth. The motivation of the Medical Devices Workgroup (MED WG) as part of the Bluetooth Bluetooth Special Interest Group (Bluetooth SIG) was to establish an interoperable wireless standard that replaces the variety of existing proprietary data protocols and formats. Formerly, “there was not even agreement over the best profile to base these on” [WG09]. Until HDP, the mostly used profile was the Serial Port Profile (SPP) which only emulates a standard RS-232 serial cable. However, other Bluetooth profiles like the Dial-up Networking Profile (DUN), the Fax Profile (FAX) and the Human Interface Device Profile (HID) were also in use.

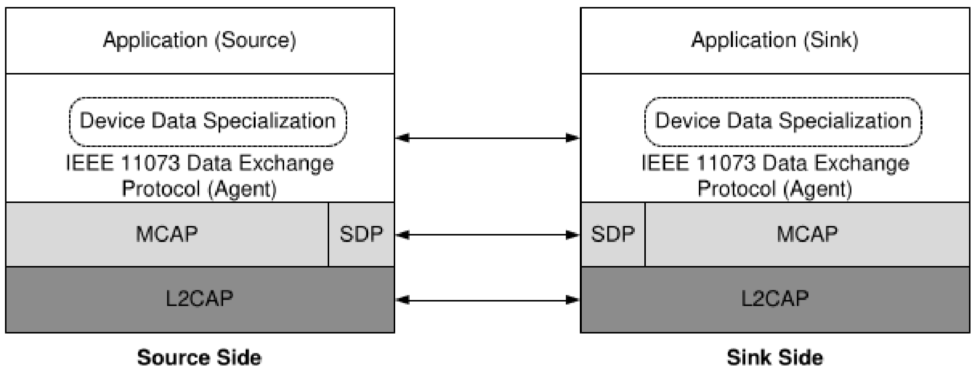

Application (Source)
Device Data Specialization
IEEE 11073 Data Exchange Protocol (Agent)
MCAP
SDP
L2CAP
Source Side
Application (Sink)
Device Data Specialization
IEEE 11073 Data Exchange Protocol (Agent)
SDP
MCAP
L2CAP
Sink Side

Figure 2.7: Bluetooth HDP Protocl layers [WG12]

At the top layer, HDP uses the already introduced ISO/IEEE 11073-20601 Optimized Exchange Protocol in association with ISO/IEEE 11073-104xx device specialization specifications in order to provide application level interoperability. Figure 2.7 shows the complete protocol stack of HDP. At the lower layers, HDP uses the Multi-Channel Adaptation Protocol (MCAP) in combination with the Logical Link Control and Adaptation Protocol (L2CAP) to establish the communication channels. Precisely, the MCAP is a

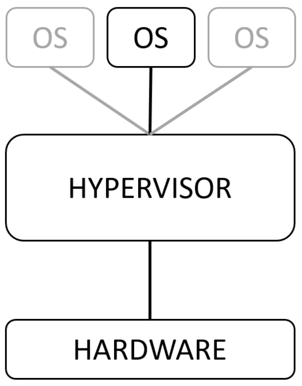
L2CAP-based protocol that provides a Control Channel to create and manage a set of Data Channels [WG08]. However, HDP adds further requirements to L2CAP such as the Enhanced Retransmission Mode and Streaming Mode. The Enhanced Retransmission Mode intends to set up reliable L2CAP channels while the Streaming Mode adds support for streaming applications. Further, HDP enforces authentication and encryption for all communication links. The HDP profile also provides optional clock synchronization by using the Clock Synchronization Protocol (CSP). CSP is a feature of the MCAP protocol and allows to synchronize the clocks of two or more devices with a high accuracy in the range of microseconds [WG08].

2.6 Virtualization Techniques

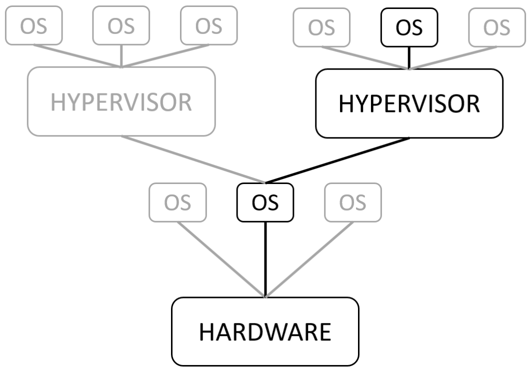
Mixed-Critical systems are typically designed using a bottom up strategy where the foundation of the system is based on a safety hypervisor at the lowest level dividing the system's resources into so called partitions. This provides full spatial and temporal isolation among the partitions allowing to run safety-critical applications together with non-critical applications on the same platform. The strict isolation among the partitions prevents any interference from non-critical applications towards critical applications.

In contrast to hypervisors with full hardware virtualization (type-1 hypervisor) that run directly on the hardware, hypervisors at OS-level (type-2 hypervisor) use technologies and mechanisms from the operating system (OS) kernel to provide virtualization for the resources of the host. Nowadays, OS-level based hypervisors are very commonly used in cloud environments for the virtualization of resources. In cloud infrastructures, container-based virtualization techniques have been proven to be a solid solution for building up highly scalable, flexible and distributed applications. Containers are easy to use and ideally suited for the packaging and deployment of distributed software. Applications based on containers are extremely scalable by simple setting up multiple instances of a container and distributing the work load among the instances.

However, today's OS-level based hypervisors cannot provide complete spatial and temporal isolation in order to build up a mixed-criticality system. Though, first approaches have already been done to overcome these limitations. One of these approaches is MemGuard, a memory bandwidth reservation system for efficient performance isolation in multi-core platforms [YYP+13] providing spatial and temporal isolation for the memory resources. Another approach was done by Luca Abeni et. al who proposed a real-time deadline-based scheduling policy to provide temporal scheduling guarantees to different co-located containers [ABC19].



(a) Baremetal Hypervisor



(b) Hosted Hypervisor

Figure 2.8: Hypervisor-Types: Baremetal vs. Hosted Hypervisor

Hypervisors allow to run multiple so called virtual machines on one host and are typically classified to two types: type-1 (bare-metal hypervisor) and type-2 (hosted or OS-level hypervisor) as shown in figure 2.8b. Type-1 hypervisors can provide temporal and spatial isolation. However, they are expensive and not efficient regarding the underlying resources (e.g. free resources of one partition cannot be used for another partition). In contrast to type-1 hypervisors, which run directly on the host hardware, type-2 hypervisors run on an common operation system. An example for a type-2 hypervisor is Linux Containers (LXC), which uses features of the Linux kernel like namespaces (e.g. pid, network and user) and control groups (cgroups) to build up lightweight containers.

An interesting approach is implemented by the Apache Mesos project [APA20], which can also use Linux containers to isolate tasks. Apache Mesos is implemented in C++ and runs on Linux, Solaris and OS X by using the libprocess library developed by the University of California, Berkeley. However, it currently only supports the isolation of CPU cores and memory but it is planned to support network and I/O isolation in Linux in the future [HKZ+11].

2.7 Architectural Patterns

2.7.1 Service Oriented Architectures

Many modern software architecture designs are driven by a service oriented architectural approach. Instead of one complex monolithic application, the software is designed as an ensemble of services where each service fulfills a precisely specified task within this ensemble. Services are loosely coupled by using message based communication, encouraging an easy re-usability of services and good maintainability of the overall system. Typically, a message broker is used for the dissemination of messages between the services. The standard communication model of most message brokers is Publish-Subscribe, where a message is distributed from the origin service (Publisher) to one or multiple receiving services (Subscriber). Figure 2.9 shows the main concept of the Publish-Subscribe communication paradigm.

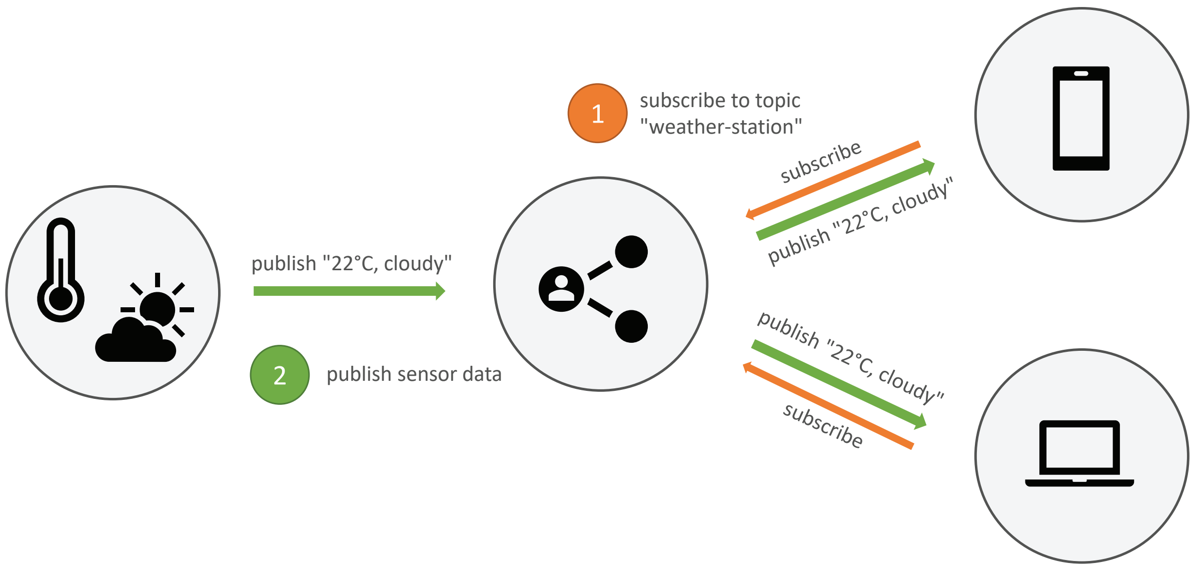


Figure 2.9: Publish-Subscribe Paradigm

A broker may support further communication models like Remote Procedure Calls (RPCs) or e.g. can be used for load balancing among multiple instances of one service. Complex services are generally realized by the composition of services using predefined service contracts, similar to interface definitions in programming languages. By this abstraction, no internal knowledge about the implementation is required to use the service.

2.7.2 Microservices

Microservices are an approach for designing and building applications as a suite of small services instead of building a monolithic software application. Figure 2.10 illustrates the difference between the two approaches.

Services are realized as small autonomous components with their own life-cycle, which makes them independently maintainable and deployable. Instead of in-memory function calls, services communicate with lightweight mechanisms using protocols like AMQP or Message Queuing Telemetry Transport (MQTT) for message exchange and remote pro-

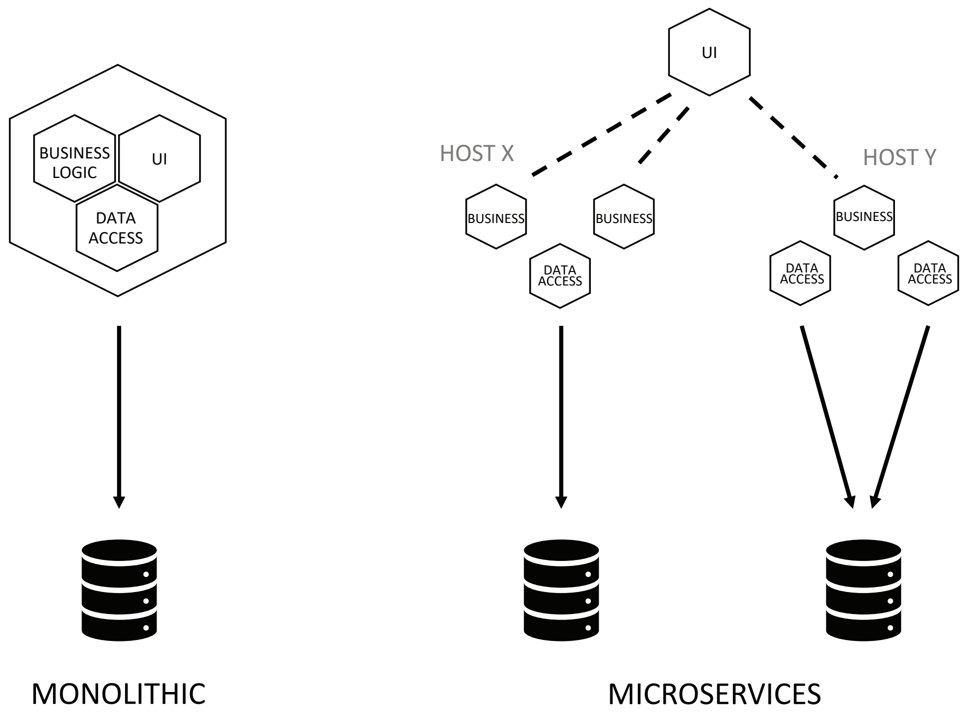


Figure 2.10: Monolithic Architecture vs. Microservices Architecture

cedure calls. This also facilitates technology diversity where for example services can be implemented in different programming languages, choosing a programming language which fits best for the service. In typical client-server architectures the program logic is located on a webserver with a single monolithic application that serves the requests of the clients. In scenarios with many clients and high load, typically load balancers are used for distributing the load in order to provide scalability to the application. Though, this implies the scaling of the whole application rather than just scaling the parts of the application that require greater resources. Applications that are based on a Microservices approach do not have this downside. Services that need more resources are just realized by creating multiple instances of the service and distributing the workload among the instances located either on one node (scaling up) or on multiple nodes (scaling out).

2.7.3 Message Oriented Communication

In service oriented architectures often Representational State Transfer (REST) is used for the communication between services. However REST is based on a synchronous communication model where one service has to wait the response of another service. This is not ideal since it brings in dependencies among services. A better approach is to use message oriented communication based on the Publish-Subscribe paradigm where message queues replace the direct synchronous calls. This avoids the mentioned dependencies between services because requests to other services can be sent in a fire and forget manner.

AMQP

Advanced Message Queuing Protocol (AMQP) is a binary message protocol which is standardized (ISO/IEC 19464:2014 and OASIS) and operates in the OSI model layer 7. Figure 2.11 shows AMQP in the OSI model. AMQP is a message protocol that uses the concept of a message broker for receiving messages from Publishers and routing them to Subscribers. AMQP is commonly used in version 0.9.1, although there is already version 1.0 available. The following concepts refer to version 0.9.1 where messages are sent by the publishers to AMQP entities called exchanges. There are different types of exchanges available that define how messages are routed in the Message Broker. An exchange type that is relevant for this work is the topic exchange, which routes messages based on the routing key in the message and routing schemes (also called bindings) to zero or more further AMQP entities called queues. The general purpose of queues is to store messages before they are consumed by the consumers. A queue has to be declared by the consumer and the consumer has to subscribe to the queue before message can be forwarded to the consumer. Queues can be configured to persist even after the (last) consumer has cancelled the subscription to that queue. This allows for example to update or replace consumers in a hot plug and play manner where no data is lost during the absence of the consumer. Topic exchanges further allow to use patterns as routing schemes, which enables a broad range of possible use cases for this type of exchange, such as distributing data to different consumers depending on their type (e.g. sensor or service) or multicast routing of messages.

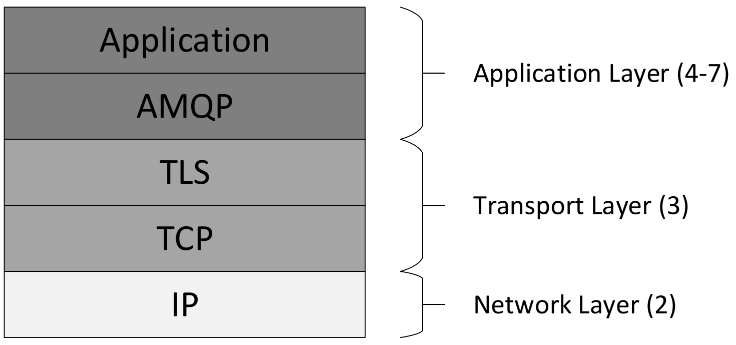


Figure 2.11: AMQP in the OSI model

A state-of-the-art message broker for AMQP is RabbitMQ Message Broker (RabbitMQ) which is an open-source message broker that can be used to let heterogeneous applications communicate and interact via a common message protocol. Additionally, RabbitMQ supports the distribution of work load (load balancing) by allowing to queue messages and distribute them automatically to multiple workers. In this way, architectures and platforms built on top of RabbitMQ can easily scale with an increasing amount of users during their lifetime.

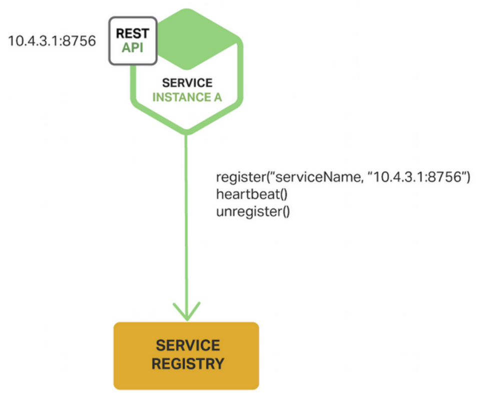
MQTT

MQTT is a lightweight publish/subscribe message protocol, particularly designed for the limitations of Machine to Machine (M2M) communication and IoT devices where a small footprint is required and the usage of network is generally desired to be infrequent due to restricted power resources. In 2014, MQTT was standardized as an OASIS (Advancing open standards for the information society) standard [OAS14] and further became an ISO standard [ISO16] in 2016. The MQTT protocol requires a central message broker for the message exchange. The broker can handle multiple clients at the same time where clients can be publishers and/or subscribers. Messages in MQTT are published in message exchanges called topics. Clients can publish messages to a topic or subscribe to topics in order to get messages that they are interested in. Topics can be arranged hierarchically (e.g. *sensors/person_id/heart_rate*), which allows a very dynamic filtering on topics. Applying wild card filters is possible as well, e.g. *sensors/#/heart_rate* would deliver the heart rate measurements of all users. MQTT allows three different QoS metrics for message delivery. With QoS level 0 the broker will deliver a message once, without confirmation just like a best-effort approach without any guaranty that the message will be delivered at all. QoS level 1 assures that the message is delivered at least once, though the message may be delivered more than once (which must be handled by the subscribers). At Quality of Service (QoS) level 3 MQTT delivers the message exactly once.

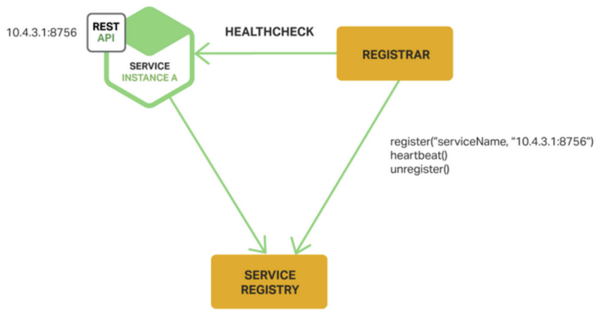
2.7.4 Service Orchestration and Discovery

In dynamic and distributed systems where services and devices (e.g. sensors) can enter and leave at run-time, it is of utmost importance that this change within the system is detected by the system. This can be achieved by two techniques, which are Service Discovery and Service Registration (cf. Figure 2.12).

Service Registration is an active step that is taken by a service itself where the service typically contacts a central Service Registry in order to announce its service details and service contact points. This technique is also called Service Self-Registration. In contrast to this, Service Discovery is an active process by the system. Here, new services are



(a) Service Self-Registration



(b) Third-Party Service Registration

Figure 2.12: Service Self-Registration vs. Third-Party Service Registration [Ric15]

announced to the Service Registry by a service registrar. In order to detect changes within the system (e.g. new service), a service registrar for example listens to events or actively scan the system. This technique is also called Third-Party Service Registration. In order to discover new services, a service registrar may use different Service Discovery Protocols. Examples of Service Discovery Protocols are the Service Location Protocol (SLP), Domain Name System (DNS) Service Discovery (DNS-SD) and Zero-configuration networking (Zeroconf).

Coordinating tasks and processes in a distributed architecture is a challenging topic. This particularly counts when the coordinating of tasks has to be managed in a fault-tolerant way. With this regard, ZooKeeper from the Apache Foundation can help to establish a fault-tolerant service orchestration. ZooKeeper is an open-source software project that targets the coordination of large distributed systems with high availability. Therefore it provides a centralized coordination service besides other features like group management or leader election. ZooKeeper facilitates the coordination of services by using a shared hierarchical namespace of nodes which is similar to a UNIX filesystem. ZooKeeper nodes can be either *persistent* or *ephemeral*. Once created, persistent nodes will exist until they are explicitly deleted. Persistent nodes are well suited for storing data which should be shared among all other components inside the system. In contrast, the lifetime of ephemeral nodes ends when the component that has created the node disconnects from Zookeeper. An important aspect of ephemeral nodes is, in contrary to the persistent nodes, the inability of having child nodes [HKJR10]. An exemplary namespace hierarchy is provided in Section 6.1.

3 Architectural Challenges and State Of The Art

A fault-tolerant real-time architecture for elderly care has to address several key challenges. A major challenge is the support of high-criticality applications which implies the application of techniques for fault-tolerance and partitioning of (hardware) resources. Especially medical applications with tight control loops require a deterministic real-time behavior of the system, which in particular puts challenges on the provisioning of computational and communication (e.g. network, memory) resources. Further challenges are the heterogeneity of the underlying technologies and the different integration levels at which the architecture is applied.

After introducing the different challenges, this chapter will present different state-of-the-art architectures for elderly care and AAL. The presented architectures are evaluated regarding the presented architectural challenges and advantages as well as disadvantages are pointed out. A further motivation of the presented state-of-the art analysis is to pinpoint research gaps that have to be addressed. Some parts of this chapter were published in [SO18].

3.1 Fault-Tolerance

As noted in Section 2.4.3, the main target of fault tolerance as a mean for dependability is the avoidance of failures. Fault tolerance is of utmost importance for safety-critical systems, where even the failure of a single component can lead to catastrophic accidents or even the loss of life of human people. In order to avoid failures, fault tolerance provides mechanisms to recover to a system state that has no error or fault. However, this requires first an autonomous detection of errors by the system itself. In order to be able to detect errors autonomously, the system must have knowledge about the intended state or behavior of the system as well as knowledge about the possible errors that can occur. An example for error detection is a bus guardian in a time-triggered real-time network that uses a priori knowledge to detect temporal failures of a component.

The design of any fault-tolerant system starts with the precise definition of a fault-hypothesis, which defines what types of errors have to be tolerated by the system [Kop11b]. By this way, the fault-hypothesis segregates faults into two domains: normal faults (faults that have to be tolerated by the fault-tolerant system) and rare faults (faults that are not considered by the fault-hypothesis). Figure 3.1 shows an overview about the state space of a fault tolerant system. As shown, faults that are covered by the fault-hypothesis (normal faults) lead to a system state that can be recovered by fault tolerance techniques to a correct state of the system. Faults that are outside the fault-hypothesis (rare faults) can for example be addressed by a never-give-up (NGU) strategy. A never-give-up strategy could for example result in a fast restart of a complete system, if the never-give-up strategy assumes transient faults in case that two components fail simultaneously.

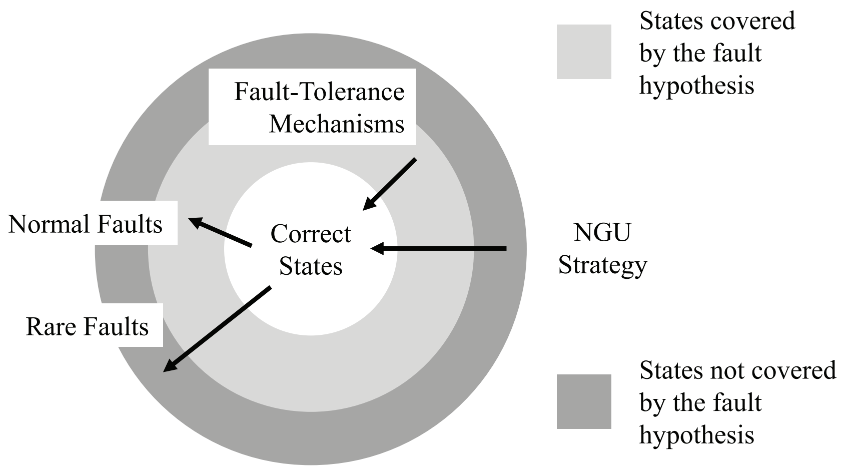

Fault-Tolerance Mechanisms
Normal Faults
Correct States
NGU Strategy
Rare Faults
States covered by the fault hypothesis
States not covered by the fault hypothesis

Figure 3.1: State space of a fault-tolerant system [Kop11b]

The fault-hypothesis enables to mask all faults that are within the fault-hypothesis and allows to apply fault tolerance techniques that are able bring the system back into a correct operation state. Therefore the fault-hypothesis must specify the Fault-Containment Units (FCUs) and the corresponding failure modes with their associated failure rates. A FCU precisely describes a unit of failure and must be defined in a way that the FCU will fail independently from other FCUs. For example, components that share the same power source will fail simultaneously if the power source has a failure. Here it is obvious that these components cannot be part of two separate FCUs and must be allocated in the same FCU. The independence of FCUs is a critical issue, which means that they have to be engineered in a way that a fault cannot cause more than one FCU to fail due to a common cause. The fault-hypothesis "is a central part in any safety-relevant system and provides the foundation for the design, implementation and test of the fault-tolerance mechanisms" [Obe11]. A failure mode describes the way in which the incorrect behavior of a faulty service deviates from its correct service. [ALRL04] provides a detailed taxonomy of failure modes and characterize them by four different viewpoints: failure domain, detectability, consistency and consequences. Figure 3.2 provides an overview about possible failure modes from the domain viewpoint. The failure rate of a service with a given failure mode is denoted as the probability that the service will fail with this failure mode in a given interval of time.

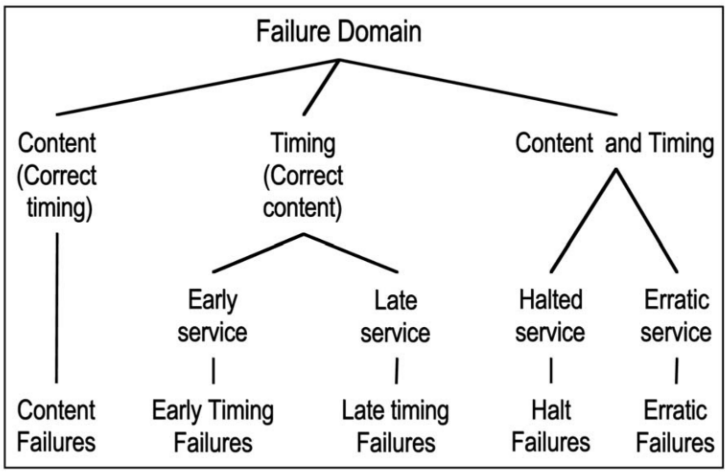


Figure 3.2: Failure modes with respect to the failure domain viewpoint [ALRL04]

A common mean to establish a fault hypothesis is the Failure Modes and Effects Analysis (FMEA). The FMEA is a bottom up process that helps to identify possible failure causes and failure modes with their effects on other system functions [DIN06]. Further, error detection methods are planned within the FMEA process. Another technique to analyze possible hazards of a system is the System Theoretic Process Analysis (STPA). In contrast to FMEA, STPA is a top-down approach which is especially tailored for safety-related system analysis and it uses a functional control diagram to model a system [Lev11]. A detailed comparison between FMEA and STPA is provided by [SBFH19].

Current state-of-the-art architectures for elderly care are missing a fault-hypothesis which also includes a solid fault-tolerance concept. As noted, in a system without fault-tolerance, the failure of a single component can affect the safety of the whole system and may even lead to failure of the complete system which in turn can lead to dangerous situations for human life. Further, fault-tolerance techniques must be implemented in order to prevent for example fault propagation. This thesis will extend the state-of-the-art by establishing a fault hypothesis within the presented architecture. This includes the definition of failure modes and failure rates of the established FCUs. In addition, fault-tolerance techniques like active redundancy and fault recovery building on the fault-hypothesis will be introduced.

Besides means for tolerating hardware failures, there are also means to achieve fault-tolerance at the level of the software. Typical fault-tolerance policies that are applied are timeout, retry or fallback. For example, the timeout policy defines a period of time during which the service must respond. If a service has a timeout, the retry policy can define how often the involvement of the service should be retried until a fallback can happen to a different service that can handle the request. In the state-of-the-art, there are already frameworks available that implement software fault-tolerance policies like Eclipse MicroProfile Fault Tolerance [Ecl20]. Eclipse MicroProfile Fault Tolerance is part of the Eclipse MicroProfile project which is an open forum to optimize JAVA for a microservices architecture with a goal of standardization [Ecl19].

3.2 Real-Time Support

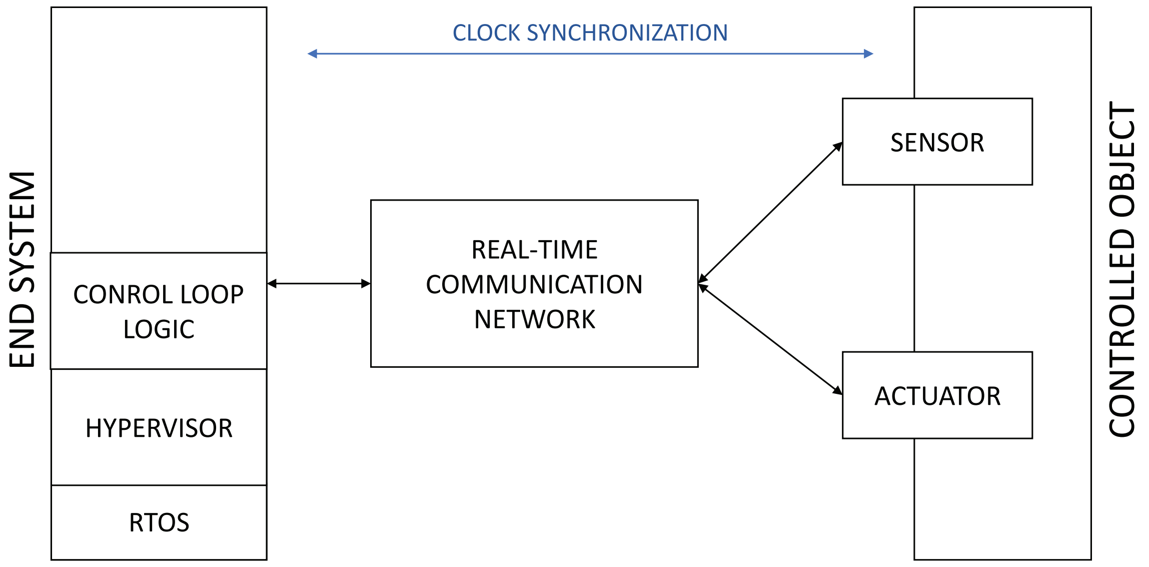


Figure 3.3: Distributed real-time system - Example

Having a look at elderly care scenarios with telemedicine in case of an emergency of an elderly person, medical control loops enforce hard real-time requirements both for end-systems and communication networks in order to be reliable and stable. Regarding communication networks, this requires a deterministic real-time network with bounded message-transport latency and jitter. Time-triggered networks such as TTEthernet [KAGS05] or the evolving Time Sensitive Networking (TSN) [IEE16b] standard are well suited to satisfy the required timing constraints and dependability. The stringent requirements on timely behavior counts as well for the end-systems that run the critical services or that provide critical sensor data. Further, all clocks have to be synchronized to a global time base in order to have a consistent view about time and about the state of the system.

Figure 3.3 shows an example of a distributed real-time control system. The depicted system consists of a real-time network that connects an actuator and a sensor to an end-system which runs the control logic. A possible application scenario for the depicted control loop could be a system where the controlled object is a boiler with water and the controller (end-system that runs the control logic) has to maintain the temperature of the water with a heater (actuator) within a small range around a specific temperature. The stability of the control loop depends on the dead time (cf. Section 2.2.1) of the control loop and has direct impacts on the temperature of the water in the boiler. The open-world assumption introduced in Section 3.3 motivated the requirement that components can enter and leave the system at run-time of the system. This implies a dynamic reconfiguration of the system which counts especially for the computational and communication resources in real-time scenarios.

3.3 Open-World Assumption

Many use cases that apply for architectures in the field of elderly care require the integration of heterogeneous sensors like wearables or stationary health sensors. Often, this integration has to be conducted at run-time demanding a dynamic reconfiguration of the system where other components have to be informed about the new configuration. Besides the hardware components like sensors, this counts for applications and services as well. This demand for a dynamic reconfiguration requires a flexible software architecture that allows the integration and removal of components at run-time and avoids to put any burden on the elderly people or nursing staff regarding the integration procedure.

In modern state-of-the-art architectures for distributed architectures, service orchestration and discovery (cf. Section 2.7.4) are used to track changes within the system regarding services and applications. This typically aims at minimizing or even avoiding manual administrative efforts regarding the configuration and management of services. Zhu et. al provide a short overview about existing protocols for service discovery in pervasive environments and compare them by introducing a taxonomy for the classification of service discovery designs [ZMN05]. Thus, existing architectures are typically restricted to the local domain like the Local Area Network and do not provide functionality for the integration of services at Internet level (cf. Section 3.4). The integration of heterogeneous devices like wearables or stationary health sensors requires similar techniques or can even be realized with protocols for service discovery. However, this requires a more complex logic in the devices which are now required to implement these service discovery protocols. In order to reduce the complexity and the overhead for the implementation in the devices, it is important for modern architectures, to rely on communication standards like Bluetooth HDP or IEEE/ISO 11073 (cf. Section 2.5).

The adaptability which is a requirement that comes with the open-world assumption is a huge challenge for real-time systems. Typically, systems with hard real-time requirements have a fixed network topology and are scheduled in advance. Predefined schedules provide determinism and good analysis facilities. This means that there is no possibility to integrate new components and to reconfigure the schedule at run-time. However, systems with predefined schedules do not cope with the requirements for a dynamic change of the system composition. For example, medical applications may require a dynamic integration of new devices at run-time. Thus, the openness of the system, where devices and applications have to be dynamically integrated, implies for example the dynamic scheduling of network traffic where it is not feasible to prepare communication schedules beforehand.

Further, manual reconfiguration of the system by humans is error-prone and time consuming. To avoid errors, which may result in later failures of the system, the reconfiguration should be done automatically by the system itself. Self-adaptation of real-time systems has already been addressed in prior work [HO13][BSP05][PZ12][RP05]. Though, the self-adaptation focuses today on the reconfiguration of resources within a fixed network topology. Currently, there is no solution in the state-of-the art for a self-adaptable real-time system, that considers changes within the network composition. Owda et. al [OAOD14] introduced the concept of a dynamic time-triggered platform based on TTEthernet where the bandwidth of pre-defined Virtual Links (VL) is allocated dynamically. However this concept does not support the introduction of new devices in the network. Thus, the system is restricted to the dynamic integration of new software components. This thesis will extend the state-of-the-art by introducing an authority for real-time communication resources that allows the dynamic integration of new software components and network nodes at run-time of the system.

3.4 Integration Levels

The definition of a distributed system used within this thesis comprises multiple (distributed) services running concurrently across different physical machines that are connected by a communication network located at different integration levels like LAN or the

Internet. Figure 3.4 shows an overview about the integration levels and typical devices located at that level. Depending on the integration level, different types of devices are involved in the communication and different techniques have to be applied in order to establish and maintain the communication between the services.

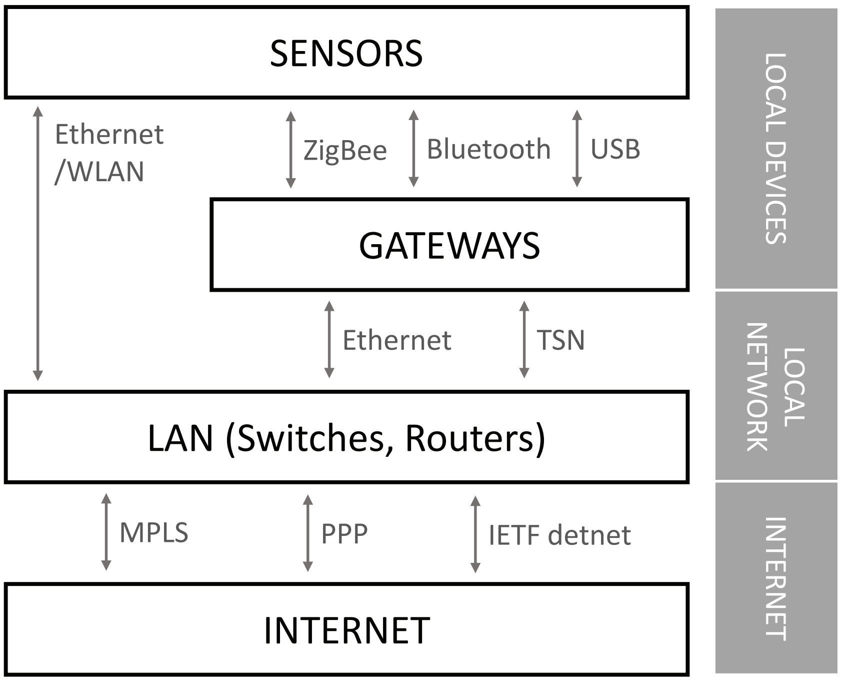


Figure 3.4: Architectural challenges - Multiple integration levels

Local Devices As shown in the figure, sensors and local gateways represent the lowest level of the local devices. Sensors could be stationary or mobile and are typically connected to a local gateway device which provides support during the integration process, e.g. in order to minimize the administrative burden on the user. This is of especial interest, since in the envisioned target field of elderly care, a typical user might be an elderly person or a member of the nursing staff. In the most optimum way, a sensor integration should be conducted automatically without any actions required by the user. Besides the sensors, this integration level comprises applications and services running on the gateway device as well. Services running on the same gateway can find each other for example by using different service discovery techniques, which are discussed in Section 2.7.4. The same applies to sensors, which can be integrated into the system in a similar manner or by using communication standards like Bluetooth HDP or IEEE/ISO 11073 (cf. Section 2.5).

Local Network (LAN) At the level of the Local Area Network (LAN), systems like gateways are typically connected with IEEE 802.1 based communication technologies like Ethernet or Wireless Local Area Network (WLAN). At this integration level, the discovery of gateway devices has to be addressed in order to comply with the requirements for distributed application scenarios that support the mobility of elderly people. This is of special interest when thinking of a use case in a nursing home where a wireless sensor reports continuously vital functions of an elderly person which is not restricted to resting in bed. The depicted scenario includes gateway devices which are distributed over the rooms of the nursing home. In order to support the mobility of the elderly person, a handover of the sensor has to be conducted whenever the person is leaving the coverage of one gateway device and enters the scope of another device. Apparently, this requires techniques for discovery and peering of gateway devices to conduct a seamless handover of the sensor device.

Internet Integration at the level of the Internet is especially important for supporting services from professional stakeholders like informational or medical services from care-givers or a doctor. Typically these service providers are not located in the same LAN as the system of the caretaker or the elderly person. At the Internet integration level, common Service Discovery protocols like DNS-SD or SLP cannot be applied.

3.5 Heterogeneity of the Underlying Technologies

Many use cases for elderly care and AAL require the integration of sensors. Therefore an architecture for elderly care has to support techniques to access and disseminate the data of the sensors to interested applications like a health monitoring application. When integrating sensors, the main challenge is the technological heterogeneity of sensors. For example, in the field of elderly care and AAL, complex medicals sensors like a pulse-oximeter should be supported as well as simple sensors like a weighting scale. Further, the integration of different types of sensors needs to offer a high degree of autonomy in order to enable easy handling by the elderly person or the nursing staff. In addition, dynamic setups must be supported with variable numbers of users and support for multiple sensors per user at the same time. At last, it should be possible to disable a certain sensor for example by the command of an elderly person or the nursing staff in order to comply with privacy needs.

In order to deal with the heterogeneity of sensors on the one side and provide an easy integration for applications of service providers on the other side, the support of medical and health device communication standards is of utmost importance (cf. Section 2.5). Regarding this, an architecture for AAL should for example support the ISO/IEEE 11073 family of standards which provides interoperability for medical devices as well as for health and fitness devices. A further standard that should be supported is Bluetooth HDP in order to comply with modern wireless health devices which mostly are based on Bluetooth as the wireless communication technology. Further, there is an ongoing trend for health and fitness devices to store sensor data in cloud based environments [DLNW13]. Due to this fact, an architecture for elderly should also be able to integrate cloud services in order to also support this type of sensor integration.

In addition, standards should be applied to the underlying communication technologies as well. Here, modern message protocols like AMQP or MQTT can facilitate the integration of components from different service providers. Applying message oriented communication also tackles with the challenge of heterogeneous programming languages by providing an Application Programming Interface (API) for the most common programming languages. This allows to use the most appropriate programming languages for a specific problem or application. Furthermore, it is very helpful to use message oriented communication based on the publish-subscribe paradigm, where a message is distributed from the origin service (publisher) to one or multiple receiving services (subscriber). Taking again AMQP as an example, the publish-subscribe communication paradigm is even the fundamental basis of the standard itself. A comprehensive overview about message oriented communication is provided in Section 2.7.3. With a carefully designed hierarchical topic namespace, new applications and sensors can be integrated into the system seamlessly without any action at the subscribers. This smart binding in AMQP is possible due to topic wildcards where e.g. an asterisk matches any single word and a hash matches zero or more words. For example, applications can subscribe with the topic *sensors.kitchen.** to receive the data from all sensors in the kitchen. Likewise, the topic *sensors.#* would address all sensors.

Message oriented communication protocols like AMQP have further advantages beyond choosing the appropriate programming language for a given task. One advantage is the possibility to spread services over the entire network which allows a very flexible distributed system. A second advantage is the possibility of queuing messages within the message broker which allows an update or replacement of services even at run-time.

3.6 Mixed-Criticality

Having a look at today's Cyber-Physical Systems and smart devices, there is an ongoing trend towards multi-core processors and highly integrated SoCs allowing to run a large variety of services and applications at the same time on a single device. This imposes multiple challenges for systems that run applications with different levels of criticality. These so called mixed-criticality systems have to provide mechanisms for spatial and temporal isolation of the system resources in order to run applications with a high criticality concurrently with applications of low criticality. Mixed-criticality applications integrate subsystems with different criticalities on a common platform. Herein, a criticality describes a component's assurance level towards failures [BD19]. Integrating those subsystems on a common platform imposes requirements on using shared resources so that non safety-critical subsystems do not interfere with safety-critical ones. Weber et al. [OW14] give a profound analysis about these requirements. The next paragraphs summarize the most important points of their analysis.

Partitioning A platform running a mixed-criticality system must provide an execution environment for each subsystem including the corresponding resources. Such an environment is called partition and can be a hypervisor partition, a CPU in a multi-core processor or a node in a distributed system. Using partitioning, the impact of faults can be limited to a component. Hence, every component can be considered as FCU and interference with other components is only possible by faulty inputs, not via shared resources. The primary goal of partitioning is to simplify certification as the subsystems can be certified independently without having to certify the entire system. This modular approach is supported by different certification standards, e.g. IEC-61508 and DO-297. There are two types of partitioning. While spatial partitioning separates multiple available resources into subsets (e.g., one CPU in a multi-core processor) or areas (e.g., memory), temporal partitioning assigns time-slots in which access to a shared resource is granted. In both types, different resources must be considered. Examples are processor cores, memory, communication and I/O resources.

Temporal Requirements Safety-critical subsystems are typically hard real-time applications. In those, the correctness of a computation depends not only on the correctness of the result, but also on its availability in bounded time. Even in fault scenarios or high load, the system must guarantee a response which regards to a computation's deadline. Hence, the worst-case behavior must be considered in the analysis of temporal aspects such as communication and computation delays. Another important temporal aspect is jitter which is the difference between minimum and maximum delays. High jitter brings additional inaccuracy in computations. However, low jitter enables the compensation of known delays as long as temporal partitioning can give guarantees and determinism on resource access.

Heterogeneity In addition to safety, mixed-criticality applications have different requirements on timing and models of computation. There are firm, soft, hard and non real-time tasks and applications which communicate via shared memory or exchange of messages. Although shared memory communication can simplify the data exchange, it results in temporal unpredictability. Accessing the memory is not pre-planned, wherefore concurrent access must be solved dynamically. This accounts further for memory hierarchies. In contrast, message passing avoids overheads for coherence protocols and is superior in the typical cases of applications with a high communication/computation ratio. All in all, a mixed-criticality platform must consider all these types of heterogeneity to support a wide range of applications.

Adaptability At run-time, there is always the possibility of foreseen and unforeseen changes in operational and environmental conditions. Mixed-criticality platforms require mechanisms to detect such changes and to react on them. Typically, safety-critical subsystems switch between system-wide modes which represent statically defined scheduling tables. In contrast, non safety-critical subsystems require a higher degree of flexibility. The reconfiguration mechanism must be able to reconfigure the system safely, without unintended interference between the subsystems. Furthermore, reconfiguration must finish in bounded time avoiding intermediate configurations. Here, some subsystems remain in an outdated configuration state.

3.7 State-of-the-art Architectures for Elderly Care

This section gives an overview about relevant state-of-the-art architectures for elderly care and AAL. All the presented architectures are implemented by research project as collaborative work of academia and industry. However, the list of the presented architectures does not reflect the tremendous number of architectures developed in the field of AAL and elderly care. A comprehensive survey of state-of-the-art AAL and healthcare frameworks, platforms, standards and quality attributes was conducted by Memon et. al [MWP^{+}14]. However, the survey did not focus on aspects like fault tolerance or non-functional properties like timing requirements of the applications.

3.7.1 Relevant State-of-the-art Architectures

SOPRANO The SOPRANO (Service oriented programmable smart environments for older Europeans) project [WSK08] aimed at the development of an open platform for AAL solutions by creating an ontology-based architecture on top of Open Services Gateway initiative (OSGi). SOPRANO was an Integrated Project in the European Commission's 6th Framework Programme and aimed at enabling older Europeans to lead a more independent life in their well-known environment, to integrate people with functional impairments into social life and so to retain their dignity [SOP08]. The main technological goal of SOPRANO was the SOPRANO Ambient Middleware (SAM), which should be used in every household to enable the users to interact with different sensors and actors. The project targeted at including sensor for detecting smoke, measuring temperature, detecting door status and tracking of users by radar or RFID. Potential actuators were "speech synthesizers, digital TVs with avatars, device regulators (for switching devices on/off or modifying their behavior), emergency callers to a central, touch screen devices and more" [WSK08].

As the SOPRANO architecture is based on OSGi, it thus inherits all benefits and drawbacks of OSGi. A great benefit of OSGi is its modularity that comes with the support of so called bundles. A bundle encapsulates a certain functionality and provides this functionality via services to other bundles. Bundles can be dynamically integrated into the system and have different life cycles during their stay in the system. The life cycle management allows to replace, remove or add bundles at any time. This is for example very helpful for the maintenance of bundles (e.g. installing an update required due to a bugfix) because it allows to replace a bundle during the runtime of the system in a hot plug and play manner. Further, OSGi allows to have different versions of bundles coexisting in the system at the same time. OSGi also introduces a service registry that can be used by other bundles to be responsive to the addition or removal of services. A further benefit of using OSGi is the wide range of already included OSGi services like services for data logging, administration of users and preferences or automatic detection of devices (e.g. by Universal Plug and Play (UPnP)) [OSG20].

OSGi offers great benefits with the concepts of runtime adaptability, versioning and modularity in the form of bundles. However, there are several challenges that arises due to adaptability and versioning in combination with JAVA. Here, a big challenge are stale references that lead to memory leaks. Stale references result when services from uninstalled modules (or modules replaced with a newer version) are still referenced by active code [GD08][HPMS11][RWDD09]. A further drawback of OSGi is the nature of OSGi itself, which is based on JAVA. This restricts all modules and services to be implemented in JAVA or to use Java Native Interface (JNI) in order to invoke C/C++ libraries. Further, bundles are restricted to a single host where the Java Virtual Machine (JVM) and the OSGi framework is running. Some approaches like R-OSGI [RAR07] try to overcome this issue by implementing another middleware layer on top of OSGi. Likewise, there are currently no built-in fault tolerance techniques available in OSGi. However, there exist some research activities like FT-OSGI [TCR09] and [AOS06] that propose extensions to OSGi in order to achieve fault-tolerance at service level. However, fault tolerance at the service level is not sufficient to provide a reliable basis for critical applications.

PERSONA The main goal of the PERSONA (PERceptive Spaces prOmoting iNdependent Aging) project was to find and develop AAL services for "social inclusion, for support in daily life activities, for early risk detection, for personal protection from health and environmental risks, for support in mobility and displacements" [COR20c]. The developed AAL services are executed on the PERSONA technical platform which is based on OSGi [TFRF10]. Thus, it has the same architectural benefits and drawbacks as discussed within the SOPRANO architecture. To achieve extensibility of the system in an ad-hoc fashion it was designed to use different buses for communication instead of using strictly defined interfaces.

universAAL / ReAAL The main goal of the universAAL project [HMH+11] was to consolidate the results of existing initiatives and to provide an open standardized platform for an economically development of AAL solutions. The OSGi approach from the universAAL project brings the same benefits and drawbacks as discussed within the SOPRANO architecture. In order to reduce the technical and programmatic burden of OSGi for the developers of AAL applications, several development tools were implemented within universAAL. A further goal was to implement an "application store, called uStore, through which developers, service providers and end users can offer and obtain AAL applications" [UNI20a]. A further outcome of universAAL was a ontological model for AAL applications, which was used to share knowledge about resources in the universAAL architecture. Relevant for this work is the fact that the universAAL project introduced the concept of scheduled traffic to OSGi in order to comply with needs for critical and time-sensitive applications [OAOD14]. After the lifetime of the project, the development of the universAAL platform was continued as non-profit software platform for the Internet of Things (IoT), called universAAL IoT [UNI20b].

The main objective of the EU-funded ReAAL project was a pilot study of reference AAL service implementations involving over 7000 users in seven European countries. The service implementations were based on the universAAL platform [REA20]. However, it turned out that the universAAL platform has to be revised with several further releases in order to include the features and needed by the pilot and to fix bugs [REA08]. The documentation of the universAAL platform also needed to be revised, and the developers in ReAAL needed a significant amount of time to become familiar with the complexity of the universAAL platform. One of the consequences of this was that the first pilots developed in ReAAL used only a small subset of functions of the universAAL platform.

OASIS The EU funded project OASIS (Open Architecture for Accessible Services Integration and Standardisation) aimed at introducing an “innovative, ontology-driven, open reference architecture and platform in order to facilitate interoperability, seamless connectivity and sharing of content between different services and ontologies in all application domains relevant to applications for the elderly” [COR20b]. The main target area were applications for independant living, such as a nutritional advisor, an activity coach or health monitoring. The design of the user interface followed a user-centered design in order to achieve usability and acceptance as one of the main goals of OASIS. Within the project, several tools were developed to work with ontologies. E.g. a tool called Concept Anchoring and Alignment Tool (CAAT) was implemented to align Simple Object Access Protocol (SOAP) based services with the service ontologies used in the project by using the Web Services Description Language (WSDL) of the services.

MPOWER MPOWER (Middleware Platform for eMPOWERing cognitive disabled and elderly) was an EU funded project had four overall technical goals: The MPOWER architecture, the MPOWER middleware, the Model-Driven Software Development (MDSD) healthcare framework and the MPOWER UML extensions. The MPOWER architecture was designed as Service Oriented Architecture (SOA) and was implemented by using the SOA reference architecture from IBM [MPO08a]. The MPOWER middleware implemented reference services for three different categories, specifically communication services (e.g. service for sending notifications to users), information services (e.g. calendar, medication list), management services (e.g. managing user rights), security services (e.g. authentication and authorization of users) and sensor services (e.g. adding a sensor). The reference services were realized as web services implemented in JAVA running on a Java Sun Application Server. The MDSD healthcare framework added guidelines for the development process of services and their configuration. Finally, the MPOWER UML extensions extended the MPOWER middleware, e.g. by providing an UML profile for services [WSM09].

The envisioned overall goal was to establish the MPOWER service platform that should be used in smart homes and care centers. For the communication among the different platform instances MPOWER proposed to use HL-7 (Health Level Seven) messages. HL-7 is set of international standards operating at Open Systems Interconnection (OSI) layer 7 for transferring sensitive medical data between various medical and healthcare providers like hospitals and nursing homes. Regarding the integration of different sensors, the MPOWER middleware proposed the introduction of the Frame Sensor Adapter (FSA) which main goal is was to overcome the communication burdens introduces by the usage of different sensor protocols. After the integration of the sensor, the sensor could be used like a normal service [MPO08b].

GENESYS The objective of the GENESYS (GENeric Embedded SYStem) project was the implementation of a cross domain reference architecture for embedded systems suited for different applications domains [COR20a], such as industry, automotive and avionics. Starting point of the GENESYS architecture was the analysis of the requirements and constraints documented by the ARTEMIS strategic research agenda. In particular, the following three challenges have driven the development of GENESYS [OK09]: complexity management, robustness and energy efficiency. A major contribution of the GENESYS architecture is the introduction to the notion of components. A component is a self-contained module that makes its local services available at the Linking Interface (LIF) to other components. According to the underlying definition of a time aware system on which GENESYS is based [GIJ$^+$03], the LIF specification explicitly includes non-functional requirements, such as time requirements of services. GENESYS further discussed the problems that arises with the progression of time in distributed systems, like the drifting apart of clocks due to the different drift rates of the clocks. In order to cope with these challenges, GENESYS proposed the usage of a sparse time base in combination with clock synchronization. Also, GENESYS discussed the presence of faults (e.g. design faults and hardware faults) and the resulting impacts on the whole system.

3.7.2 Discussion

In this section, the presented state-of-the-art architectures are discussed with regard to the previously introduced challenges for an architecture for elderly care that has to support real-time applications in a fault-tolerant manner. The presented architectures from the projects SOPRANO, PERSONA and universAAL are based on OSGi which offers great benefits with the already discussed concepts of runtime adaptability, versioning and modularity. However, the adaptability of the SOPRANO and the PERSONA architecture is limited to the dynamic integration of new bundles, services and sensors. Further these two architectures do not consider temporal requirements of (real-time) applications. universAAL goes a step further by introducing the possibility of a dynamic reconfiguration of real-time communication resources during the run-time of the system [OAOD14]. However, the dynamic adaptability is restricted to the available real-time communication resources that were available at the start of the system. The approach of universAAL does not consider the introduction (or removal) of new real-time communication hardware at run-time. Thus, it also cannot adopt to the loss of resources e.g. due to a failure. A further drawback of the presented OSGi architecture is the limitation to the integration level of local devices. Though, there are as mentioned some research approaches like R-OSGI [RAR07] that try to overcome this issue.

	SOPRANO	*PERSONA*	*universAAL / ReAAL*	*OASIS*	*MPOWER*	*GENESYS*
Fault-Hypothesis with Fault-Tolerance Techniques						X
Support for distributed Real-Time Applications			X			X
Open-World Assumption	X	X	X	X	X	
Multiple Integration Levels						
Heterogeneity of the underlying technologies					X	
Support for Mixed-Criticality						X

Table 3.1: State-of-the-art - Overview

The architectures resulting from the projects OASIS and MPOWER both use the concept of web services to establish the functionality of the system. While the architecture of the project OASIS heavily used ontologies for services, the MPOWER architecture used a model-based approach based on UML for building services. However, web services typically are based on REST or SOAP as interface for the communication with other services.

Since both techniques are synchronous communication techniques (cf. Section 2.7.3), using these techniques lead to dependencies among services which makes it extremely difficult to scale services for performance (e.g. if a service has to be duplicated in case of a load balancing scenario) or for fault-tolerance reasons. The architecture from the GENESYS project is farther more advanced than the previously presented architecture since it already considers timing-requirements of services and the occurrence of faults in the system. In order to achieve fault-tolerance, the GENESYS architecture introduces well-defined FCUs. However, the definition of the FCUs is missing assumptions about failure modes and failure frequencies. Further, the GENESYS architecture does not provide concepts about the temporal and spatial isolation of services in order to provide a solid platform for the concurrent execution of applications with different levels of criticality (mixed-criticality system). Summarized, none of the presented architectures provides all required characteristics for the establishment of distributed fault-tolerant real-time applications in the field of elderly care.

The following aspects are not considered at all or only considered partly by the presented architectures. First, there is no solution available that supports a dynamic reconfiguration of computational and communication resources in real-time scenarios. Second, temporal and spatial isolation of computational and communication resources for the support of mixed-criticality is only partly addressed. The support for concurrent applications with different criticality requires full temporal and spatial isolation as presented in this thesis. Table 3.1 shows an overview about the presented architectures regarding the support for the previously introduced challenges for a fault-tolerant real-time architecture for elderly care.

4 Fault-Tolerant Real-Time System Architecture

This chapter will introduce the fundamental concepts of the fault-tolerant architecture presented in this thesis. As the previous chapter has shown, there are several challenges that have to be tackled in order to establish a fault-tolerant and real-time capable architecture for elderly care. Some of these challenges put requirements on the underlying platform. This applies in particular to the real-time support of the presented architecture, which e.g. requires an underlying real-time operating system and by reason of the distributed character of the presented architecture also requires a real-time communication network. Referring to this, this chapter will first introduce the requirements towards the underlying platforms before presenting the overall system structure and the main building blocks of the intended fault-tolerant real-time architecture. As a next step, this chapter will show which services have to be provided for applications running on the resulting platform. This includes e.g. service for real-time communication, service orchestration and clock synchronization. Another goal of this chapter is to introduce the fault assumptions that the presented architecture is based on. This includes in particular the establishment of a fault-hypothesis and the definition of fault-containment regions (FCRs) for the presented architecture. Some parts of this chapter were published in [SO18].

4.1 Requirements towards the underlying platforms

In order to comply with the architectural challenges that were introduced in Chapter 3, the underlying platform has to fulfill several requirements. These requirements are presented in the following.

4.1.1 Real-Time Operating System

The real-time support of the presented architecture requires first of all an operating system where the temporal behavior is known a priori and the execution of tasks is guaranteed to happen within a bounded period of time. These guarantees can be provided by a real-time operating system. The basic concepts and properties of real-time operating systems are introduced in Section 2.3. Further, the presented architecture requires high level services which must be provided by the underlying operating system such as networking, multitasking or support for different programming languages. Further, it should be possible to leverage existing tools, communities and to provide compatibility for already existing applications. Due to these reasons the presented architecture is based on a fully featured operating system like Linux or Windows Embedded. However, these operating systems were originally not designed to provide deterministic behavior for real-time applications. Though, there are several approaches that can be applied to achieve support for real-time applications. For Linux for example there are two potential techniques that can be applied to make Linux real-time capable, namely PREEMPT_RT, which is a patchset that makes the Linux kernel preemptive and SCHED_DEADLINE, which is a deadline scheduling policy for scheduling tasks in Linux.

4.1.2 Real-Time Networking

Besides the execution of real-time tasks, the operating system has to supply real-time communication traffic between different nodes in the network. Here, time-triggered communication protocols and standards have to be supported by the operating-system and the networking hardware that provides the communication services. Here, the upcoming family of TSN standards for real-time communication in Ethernet based networks is one feasible solution. Although this family of standards is still in draft by the IEEE (see also Section 2.2.4), there are already implementations available for some of the TSN standards. In Linux for example, as with Linux Kernel version 5.2 there is a queueing discipline named Time-Aware Priority Shaper (TAPRIO) (cf. Section 6.2) that implements the time-aware scheduler defined in TSN enhancements for scheduled traffic. An additional queueing discipline implements the offloading of the packet transmission to the network hardware, which allows fine-grained control over the transmission time of Ethernet packets. However, the hardware offloading has to be supported by the network hardware. For example, the network card i210 from Intel is one of the networking cards that support packet offloading besides support for clock synchronization protocols like the IEEE standard for Precision Clock Synchronization and TSN Synchronization I(IEEE 802.1 AS).

4.1.3 Wireless Connectivity

A further requirement towards the underlying platform is wireless connectivity which allows a seamless and easy-to-use integration of wireless sensors and mobile medical devices into the system. Wireless connectivity is of particular importance here, because there is an ongoing trend in the market towards wireless devices. This applies in particular for the consumer market but the trend is also visible in the market for professional medical and healthcare devices. Bluetooth is the targeted wireless protocol in the presented architecture since it is supported by the two medical device communication standards IEEE 11073 and Bluetooth HDP.

4.2 System Structure and Building Blocks

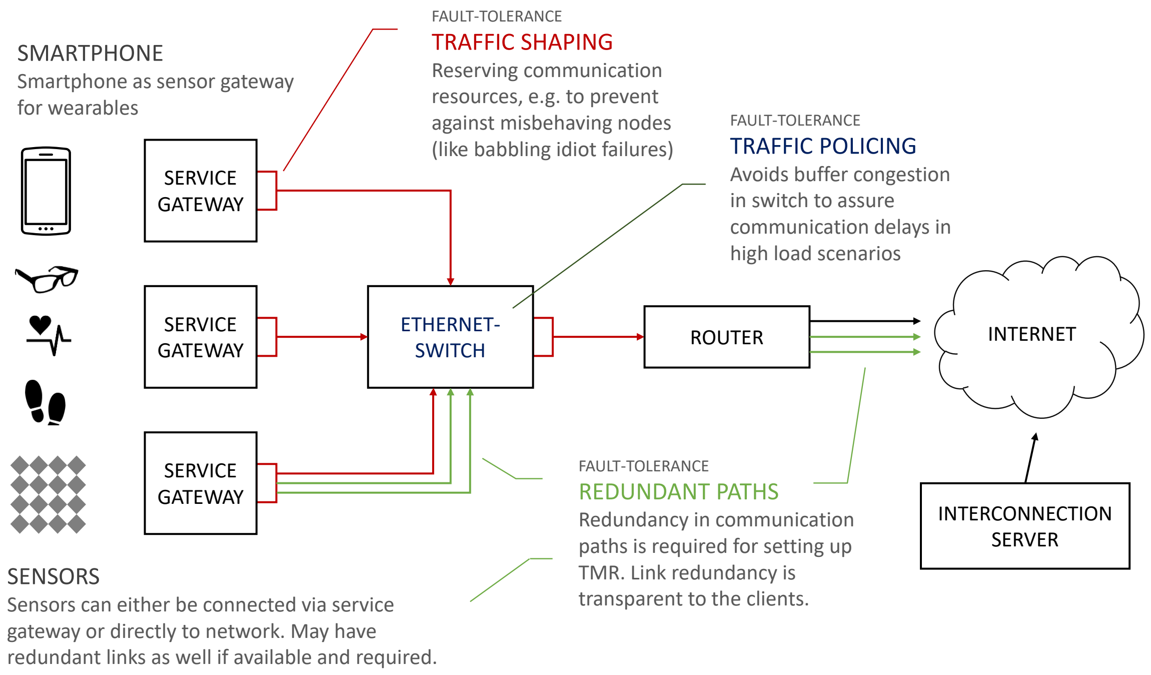

SMARTPHONE
Smartphone as sensor gateway for wearables
FAULT-TOLERANCE
TRAFFIC SHAPING
Reserving communication resources, e.g. to prevent against misbehaving nodes (like babbling idiot failures)
FAULT-TOLERANCE
TRAFFIC POLICING
Avoids buffer congestion in switch to assure communication delays in high load scenarios
SERVICE GATEWAY
SERVICE GATEWAY
SERVICE GATEWAY
ETHERNET-SWITCH
ROUTER
INTERNET
INTERCONNECTION SERVER
FAULT-TOLERANCE
REDUNDANT PATHS
Redundancy in communication paths is required for setting up TMR. Link redundancy is transparent to the clients.
SENSORS
Sensors can either be connected via service gateway or directly to network. May have redundant links as well if available and required.

Figure 4.1: Overview of the proposed fault-tolerant system architecture

The following section introduces the structure of the system model including the main parts of the proposed architecture. A high-level overview is given in figure 4.1. As depicted, the system model comprises local devices like sensors and smart devices (e.g. smartphone or tablet). These devices are either connected directly to the communication network or are connected to a service gateway that runs the platform and system services of the presented architecture. This includes a common set of core platform services for connectivity, message dissemination, sensor integration and service discovery. As noted, the ecosystem may also include smartphones which can play several roles. At first place, a smartphone can act as a sensor gateway in order to connect wearable devices, e.g. for personal health applications. At second place, the smartphone can act as an end-user interface, for example to configure services or to show alert messages. Further, the local ecosystem may also comprise one or more network enabled stationary sensors, like the commercial product SensFloor®. SensFloor is an intelligent floor underlay which is capable to detect a fall of a person or can be used for gait analysis [STS+13].

As the figure also shows, the presented architecture applies several techniques for fault-tolerance at the communication network and the service gateways. For example, redundant paths are used to comply with use cases that required a highly reliable communication between different nodes. Further, the architecture applies techniques for traffic policing and traffic shaping in order to detect temporal failures of the network nodes.

The depicted system model of local interconnected ecosystems would already be sufficient for small scale AAL solutions, e.g. for a private or retirement home. For example, think about a solution that provides fall detection and automatically alerts the nursing staff in case of an accident. Though, integrating professional services like information or medical services from caregivers or a doctor should also be supported by an architecture for elderly care. Therefore, the system model supports the interconnection of local services and sensor ecosystems at Internet level by implementing a central interconnection server. By this way, a doctor's office can use the services and the sensor ecosystem as well in order to provide professional medical services to the elderly people.

4.2.1 Service Gateway

The service gateway is the central component of the service and sensor ecosystem. It hosts the runtime environment with the platform and system services, which are introduced in Section 4.3. It is called service gateway because it will act as a gateway on the one side for professionals to provide their health services. On the other side elderly persons or their relatives will use the gateway to discover and employ the services provided by the professionals. For privacy and data protection reasons, the architecture stipulates one gateway per person.

4.2.2 Smartphone and Sensors

At present there is an ongoing trend towards wearable devices for personal health applications. Some of these devices can be of great interest for AAL use cases. For example, the E4 wristband provides sensors for photoplethysmography and electrodermal activity to monitor physiological signals like heart rate variability or to detect stress or excitement of a person [Emp15]. Another wearable to be promising for AAL use cases is the Jins Meme eyewear that for example can be used for cognitive activity recognition [LSG18]. It has a three-dimensional electrooculography sensor which can be used for example to recognize activities like reading a newspaper or to detect emotional states of a person. Wearables are typically connected to a smartphone by a wireless communication technology like Bluetooth or Bluetooth Low Energy in order to configure the wearable and to display and monitor the sensor data. For this reason, the system model comprises an smartphone, which will act as a sensor gateway in order to connect wearable devices and other sensors as well. Further, the smartphone can be used as an end-user interface, for example to configure services or the service gateway, to show alert messages and to find and apply for professional health care services. Besides the ability to integrate sensors with the use of the smartphone as a sensor gateway, stationary network enabled sensors can also be integrated into the ecosystem by using the platform services of the service gateway. Section 6.4 will provide detailed information about the integration of sensors into the architecture.

4.2.3 Real-Time Network

The service gateways and stationary sensors are interconnected by an Ethernet switch. In order to comply with the need for distributed real-time applications, the network hardware has to support a standard for real-time communication like TTEthernet or TSN. Both techniques are based on Ethernet and are ideally suited for the intended application scenario due to the compatibility to a vast amount of available hardware and software on the market. Further, the TSN technologies will be included in COTS (Commercial off-the-shelf) Ethernet network hardware in the near future. This provides real-time communication hardware at a very low price compared to expensive industrial real-time communication technologies like Profinet or Ethercat. Additionally, Ethernet allows a very flexible communication infrastructure reaching from a simple star topology to even hierarchical star topologies.

4.2.4 Interconnection Server

In order to use professional services like information or medical services from caregivers or a doctor within the local ecosystems, the system model introduces an interconnection server. The interconnection server will allow to register public services from stakeholders like professional caregivers. These services then can be discovered and used by the local service gateways. Further, the interconnection server will allow to connect the local ecosystem of a user with the ecosystem of other users. This is for example desirable when a relative wants to take care for an elderly person or wants to be alarmed in case of an accident or fall of the elderly person. In order to connect with the interconnection server, an additional router is required within the network topology.

4.3 Platform and System Services

This section elaborates about the services that have to be provided in order to run elderly care applications with fault-tolerance and real-time requirements. Figure 4.2 shows an overview about the services grouped into core services and platform services. Further shown are the services that have to be provided by the underlying technologies that were introduced in Section 4.1. In the following sections each service is introduced and the background is described that makes the respective service necessary for the presented architecture.

4.3.1 Service Orchestration and Discovery

Service orchestration and discovery is important to keep the knowledge about services within the system up-to-date. For example, the system should be able to detect new or leaving services automatically. Likewise, services should be able to register themselves in order to use all other services within the system and deregister themselves in order to leave the system in a ordered way. For this it is necessary that the system provides the following basic services.

Service Discovery As already mentioned, the system must be capable of detecting new services that enter the system. This is especially interesting for services that only send information or data to the system like a temperature sensor or an alarm button. For this purpose, different service discovery protocols are already available. One potential solution is Avahi, which is available for Linux and BSD based operating systems. Avahi

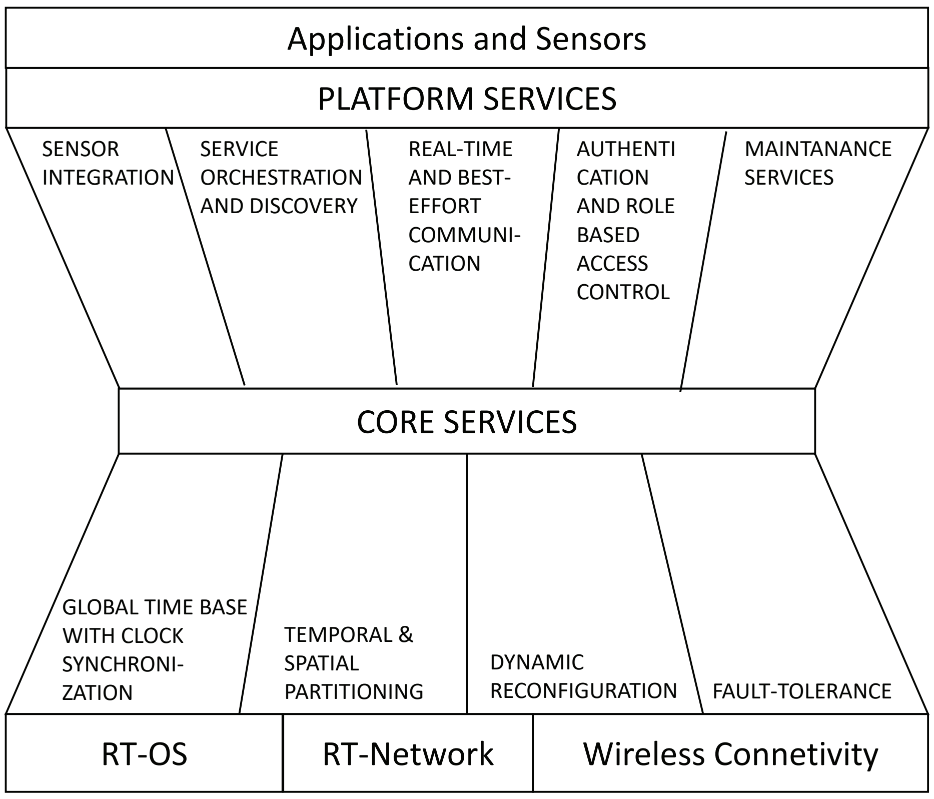

Applications and Sensors
PLATFORM SERVICES
SENSOR INTEGRATION
SERVICE ORCHESTRATION AND DISCOVERY
REAL-TIME AND BEST-EFFORT COMMUNI-CATION
AUTHENTI CATION AND ROLE BASED ACCESS CONTROL
MAINTANANCE SERVICES
CORE SERVICES
GLOBAL TIME BASE WITH CLOCK SYNCHRONI-ZATION
TEMPORAL & SPATIAL PARTITIONING
DYNAMIC RECONFIGURATION
FAULT-TOLERANCE
RT-OS
RT-Network
Wireless Connetivity

Figure 4.2: Service Model

is a free implementation of DNS Service Discovery (DNS-SD RFC 6763) over Multicast DNS (mDNS RFC 6762). An Avahi compatible protocol implementation is Bonjour, which is available for Apple MacOS based systems. Details on the mentioned protocols can be found in Section 2.7.4.

Service Orchestration Knowledge of the available services in the system and their corresponding status is especially of importance for real-time communication services introduced in Section 4.3.3. However, other services need to find other required services within the system. Furthermore, services should be informed when other services enter or leave the system. One possible scenario is a service A that can provide a better service quality with an additional service B that was not available at startup but became available later during the runtime of the service A. In a service-oriented architecture that uses message-oriented communication based on protocols like AMQP in the presented architecture, changes to the system regarding the service composition can be announce on an dedicated message exchange that all service have to subscribe to. However, this message exchange has to be fed with messages by some authority that has the knowledge of the services and their status within the system. Depending on the integration level, there are different solutions for the service orchestration available. Examples are Apache ZooKeeper [APA16], HashiCorp Consul [Has20a] and Netflix Eureka [Net20], which are all open-source and available under public licenses that also allows the use in commercial products. Netflix Eureka for example is especially of interest when services should be orchestrated within the integration level of the Internet.

4.3.2 Fault-Tolerance

Safety-critical services as well as services that have to guarantee a high level of availability require fault-tolerance techniques like active redundancy or a membership service. For services that have to provide a high level of availability, there should be a service authority that can relay failed requests from faulty or failed services to healthy services or nodes. To prevent a single point of failure, this authority should be replicated and distributed across multiple nodes. Furthermore, the service authority should be able to start new instances of a service if the requests cannot be handled by the existing instances of the service. Likewise, the system should support the load-balancing of requests to multiple instances of the services in order to ensure the quality of service of the involved service.

4.3.3 Real-Time Communication

Real-time communication is required whenever distributed safety-critical applications have to be executed. Real-time communication provides deterministic bounds to message transport latency. This is of utmost importance for example in medical control loops in which no human shall be injured or hurt. It must also be assured that the services using the real-time communication are not interfered or even stopped by other services. Moreover, the communication activities of the services have to be scheduled and executed within strict timing constraints. By this regard, the real-time communication services must be executed with the highest priority in an preemptive system (like Linux with PREEMPT-RT, Windows Embedded, QNX) or within a dedicated period of time (Linux with SCHED_DEADLINE task scheduler).

4.3.4 Best-Effort Communication

Best-effort communication services are required for services that are not safety-critical but can also be required by services that fulfill a safety-critical function, e.g. for registration of the service towards the service orchestration authority in the system. Best-effort communication is not restricted to one protocol (TCP, UDP, AMQP etc.) or message transport technology (Ethernet, Bluetooth etc.). Best-effort communication by its wording itself means that the successful message transport cannot be guaranteed, like it is the case with TCP/IP communication in the Internet where packets can get lost during their way through the Internet. This can have multiple reasons like buffer congestion in a router that has to forward the packet to the next hop or the final destination of the packet. However, best-effort communication is unreliable in terms of message transport latency because it relies on an event-driven paradigm where message can be sent at any time. In contrast to this, real-time communication often relies on a time-triggered paradigm with a fixed assignment of time-slots for each communication partner which results in a predictive behavior at the cost of a lower utilization of the communication resources. An example for best-effort communication is message-oriented communication provided by a message broker like RabbitMQ or Apache Kafka. Here in this open scenario where publisher and subscriber can enter and leave at any time, it is impossible to provide temporal guarantees regarding message latency and jitter.

4.3.5 Temporal and Spatial Partitioning

Temporal and spatial partitioning protects safety-critical applications from interference by other applications in the system. For the approach introduced in this thesis, two types of partitioning are of special interest. While spatial partitioning separates multiple available resources into subsets (e.g., one CPU in a multi-core processor) or areas (e.g., memory), temporal partitioning assigns time-slots in which access to a shared resource is granted. In both types, different resources must be considered. Examples are processor cores, memory, communication and I/O resources. Services for partitioning must be provided at the lowest level of the system. A feasible solution to achieve partitioning at a low level is to use a (certified) baremetal hypervisor. The basic concepts of hypervisors and virtualization techniques are introduced in Section 2.6. However, the operating system itself can also offer services and techniques for partitioning. Linux for example introduced the SCHED_DEADLINE scheduler with Kernel version 4.13 which allows the temporal partitioning of CPU time where each task scheduled under SCHED_DEADLINE is associated with a time budget Q (runtime) and a period P.

4.3.6 Authentication and Role Based Access Control

Sensors used in AAL and elderly care applications produce sensitive and private data. It is of high importance that access to this data is controlled and restricted to authorized persons. This requires techniques for authentication and access control. By this means, a role-based access control can help to minimize the configuration effort by using predefined and user defined roles with associated rights. For example, there should be roles for the owner, caregivers, relatives and doctors. Furthermore, the potential (hierarchical) access rights should be defined in such a way that the complexity is low, but sufficient to support all use cases. This counts especially for the data stored in the service gateways. Such data comprises historical sensor data that enables a system to provide for example weekly or monthy reports about activities or exceptional events.

4.3.7 Clock Synchronization

Clock synchronization is needed in order to have an overall correct temporal order of events in a distributed system. This is because without time synchronization, timestamps from different nodes cannot be linked together. This applies e.g. for timestamps that are assigned to sensor data or data that is stored in a database. Incorrect temporal order can lead to faulty behavior of the whole system. Furthermore, the temporal order is of high importance in distributed safety-critical real-time systems for the establishment of real-time communication. This requires the synchronization of all clocks within the distributed system. Here, protocols like IEEE Standard for a Precision Clock Synchronization (IEEE 1588) and TSN - Timing and Synchronization (IEEE 802.1AS) can be used to synchronize clocks via the communication network. Afterwards the synchronized network clock can be used to synchronize the clock of the operating system which then is used as the reference clock by all services running on the operating system. An implementation of IEEE 1588 (and parts of IEEE 802.1AS) is available from the LinuxPTP project [Ric18].

4.3.8 Sensor Integration

When integrating sensors, the main challenge is their heterogeneity e.g. regarding the used communication technology (wireless vs. wirebound) or protocol (standardized vs. proprietary). In order to lower this burden, an architecture for elderly care and AAL should support communication standards like ISO/IEEE 11073 and Bluetooth HDP (cf. Section 2.5). Furthermore, the platform should provide a mechanism to disable a certain sensor for example by the command of an elderly person or the nursing staff in order to comply with privacy needs.

Similar to services, a system has to keep knowledge about which sensors are connected to the system and has to keep track of changes in order to always have up-to-date knowledge about the system's state regarding sensors. This includes information about the properties of the sensor like which type of data that is delivered by the sensor or in which interval the data is delivered, which is especially interesting for real-time applications and scheduled traffic (cf. Section 2.2).

4.3.9 Dynamic Reconfiguration

The open-world assumption introduced in Section 3.3 where components can enter and leave during the runtime of the system implies the need for (autonomous) dynamic reconfiguration of the system. This counts especially for the computational resources as well as for the communication resources in real-time scenarios. There are further challenges that have to be tackled when using a real-time communication network. First, the system has to maintain detailed information about the network topology and further has to keep this view always up-to-date. Second, it must adapt to changes within the communication network and the composition of services that use the communication resources. Third, it must be capable of carrying out a re-scheduling of all communication resources and further be able to distribute the new schedules among all nodes.

4.3.10 Maintenance Services

Besides the services that provide the functionality of the system, further services are required in order to monitor the status of the system. This includes for example the status of registered services and the load of the underlying operating system regarding CPU or memory usage. Further, there should be a possibility to trace and to analyze the logfiles of the system and the running services in order to get early indicators of problems and potential failure risks. A further service should be available to maintain the system itself. This is for example necessary to update or replace services of the platform. Taking a local device and Linux as an example, parts of this maintenance service could be realized using SSH or SCP. At the integration level of the local network or the internet, different solutions like container-orchestration systems are available. Examples for state-of-the-art solutions are Kubernetes and Docker Swarm.

4.4 Fault Assumption

Within distributed systems, fault-tolerance is important to prevent that a single failure of one component can lead to a catastrophic system failure, which may lead to dangerous situations for human life. As introduced in Section 2.4.2, a failure is the deviation of the system's behavior from its specification. This failure is the result of an error, which is an unintended state of the system caused by faults. In order to detect errors, knowledge about the intended behavior of the system is required. This knowledge can be either provided a priori to error detection mechanisms or for example can be derived by the comparison of the results from redundant entities.

Due to these reasons, the presented system model includes a fault-hypothesis where all FCUs for hardware including their failure modes and failure rates are defined. This section will first introduce the fault-hypothesis following the fault-tolerance techniques which are used within the presented system model.

4.4.1 Fault-Hypothesis

During the design of a fault tolerant system, the fault-hypothesis is the most important document that states precisely which faults have to be tolerated by the system. It further identifies the units of failure and the FCUs. The fault-hypothesis enables a system to mask all faults that are within the fault-hypothesis and allows to apply fault tolerance techniques in order to bring the system back into a correct operation state.

The presented system model assumes, that during a specific time interval the failure of a single FCU must be tolerated (Single Fault Hypothesis). If more than one FCU of the same type becomes faulty then more than one fault is occurring. For hardware faults, each computing node (e.g. service gateway, sensors and smartphone) is regarded as a single FCU since they are physically at a distance and it is realistic that they therefore will fail independently. Further, each communication link is regarded as a FCU. By this way, the presented fault-hypothesis clusters the FCUs into two domains: (i) network nodes and (ii) communication links. Regarding software faults, each container (cf. Section 5.5) used for the temporal and spatial isolation of applications can be denoted as a software FCU. The following paragraphs explain the failure modes and failure rates of the FCUs in detail.

4.4.2 Failure Modes

Service Gateways For the computational nodes the failure mode is assumed to be arbitrary, where users may have a different view about the state of the node in case of a failure. This inconsistent behavior is one of the most difficult failure modes to handle. Additionally, it is assumed that a faulty network node will remain faulty until a fault-tolerance mechanism brings the FCU back into a correct mode of operation (permanent failure).

Communication Links Regarding the communication links in the local network domain, Ethernet frames can be dropped or invalidated (transient failure), e.g. due to electromagnetic interferences. Additionally, a communication link can become unavailable and remains in this faulty state (permanent failure).

Sensors Modern health and medical sensors are typically implemented as sensor nodes. This means that the sensor itself is equipped with a CPU or microcontroller, which is responsible for accessing the raw sensor and processing its data, as well as communicating with an optional device such as a smartphone or service gateway. Therefore, the failure mode of the sensors is assumed to be arbitrary, as with the service gateways.

Containers Within the proposed architecture, lightweight hypervisor-based containers are used for the spatial and temporal isolation of applications in order to limit the impact of faults. In this respect, faults that occur within one container do not affect other applications in different containers. Nevertheless, the fault can propagate within the container and lead to an inconsistent behavior of the entire container. It is therefore assumed that the failure mode of a container is arbitrary.

4.4.3 Failure Rate Assumptions

For our system architecture, the failure rate assumptions established in [OP06] are assumed. Here, it is important to distinguish between permanent and transient failures when considering the failure rate assumptions of hardware FCUs. Generally, transient failures are much more common than permanent hardware failures. For transient failures, the failure rate of FCUs with respect to hardware faults is about 1.000-1.000.000 FIT (Failures in Time, where 1 FIT is one failure per billion (10^{-9}) hours) and strongly depends on the environment. For permanent hardware failures, the failure rate is about 10-100 FIT.

4.4.4 Fault-Tolerance Techniques

The following section introduces the components and techniques that establish the fault-tolerance in the presented architecture.

Triple Modular Redundancy Arbitrary failures of the computational nodes can be masked by applying Triple Modular Redundancy (TMR). Such a TMR configuration results in fault-tolerant units (FTU), which consist of three timely synchronized deterministic

FCUs where a voter is added to every FCU. Further, FCUs must to be connected by two independent deterministic communication systems in order to tolerate a failure in the communication links. Provided that a fault-tolerant global time base is available, a TMR configuration can tolerate any arbitrary failure.

Rate-Constraint Communication In order to detect temporal failures of the network nodes, our approach applies rate constrained communication similar to Avionics Full DupleX Switched Ethernet (ARINC 664) (AFDX) (Arinc 664 Part 7) [ARI09]. Rate constrained communication establishes a priori knowledge about the temporal activity of the FCUs which makes it possible to detect failures by comparing the activity pattern of the FCU with the a priori knowledge. Additionally, rate constrained communication guarantees a minimum bandwidth for each communication channel, assuring an upper bound for both jitter and delay. The establishment of rate constrained communication with standard 802.1 Ethernet requires several techniques in order to avoid buffer congestion in the switches and to establish the communication links. Two of the applied techniques are traffic shaping and policing.

Traffic Shaping: Traffic shaping is required to bound the traffic sent to the network and thus avoid buffer congestion in the switches. According to Vila-Carbó et. al, even the switch delay is bounded, if all nodes limit their traffic [VCTMHO10]. Additionally, traffic shaping is used to provide bandwidth reservation required for setting up the virtual links with bounded transmission delay and jitter.

Traffic Policing: Traffic policing is required to avoid buffer congestion in the switches in high load scenarios, where even high priority packets of the virtual links may be dropped due to insufficient buffer. As within the approach of Vila-Carbo et. al we apply non-preemptive priority scheduling within the switches. Their empirical results showed further, that this approach can even be viable for the establishment of real-time communication in distributed real-time systems with standard 802.1 Ethernet. Thus, for traffic policing Commercial of The Shelf (COTS) switches are required that support priority scheduling according to IEEE 802.1p (Traffic Class Expediting) or the ToS (Type of Service) field in the IP Header (e.g. both are supported by COTS switches such as the GS1910-24 Gigabit switch from ZyXEL).

Active Redundancy (Redundant Paths)

As introduced, redundancy in the communication links is required for setting up the TMR. Redundancy is realized within the nodes by deploying two independent network interfaces which result in two independent communication links. Additionally, additional switches are deployed for the redundant paths.

Membership Service

If one fault-containment region fails, the information about the failure must be reported to all other FCUs with low latency. This is within the responsibility of the membership service which was introduced by Katz et. al [KLR97]. Additionally, the membership service is responsible for the "consistent activation of a never-give-up (NGU) strategy in case the fault-hypothesis is violated" [Kop11b].

Fault-Tolerant Global Time Base

A fault-tolerant global time base has to be implemented in a redundant way by introducing two synchronized reference clocks where the time can for example be derived by an external clock source like Network Time Protocol (NTP) or GPS. Further, precise clock synchronization can be achieved by protocols like the IEEE standard for a precision clock synchronization protocol [IEE08a].

Figure 4.3 shows the fault-model of this thesis. This UML model visualizes the correlations between the fault-hypothesis, the fault-tolerance techniques and the FCUs. At the top of the model we can see that the primary relationship is dependability (cf. Section 2.4). The model further introduces means, threats and attributes of dependability.

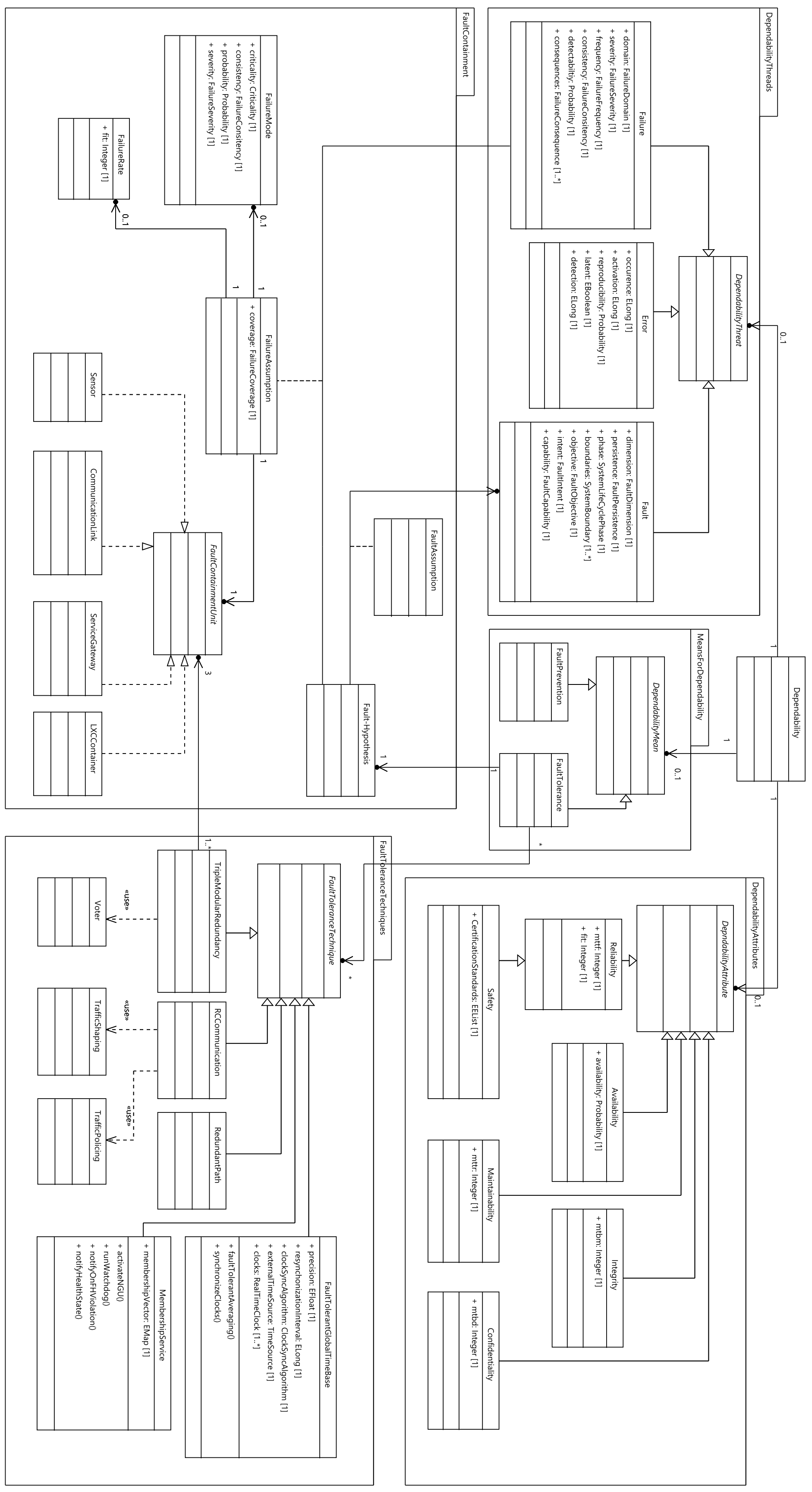


Figure 4.3: Fault Model

5 Platform Services

Based on the overall architecture and the identified services as introduced in the previous chapter, this section introduces the models and algorithms of the services.

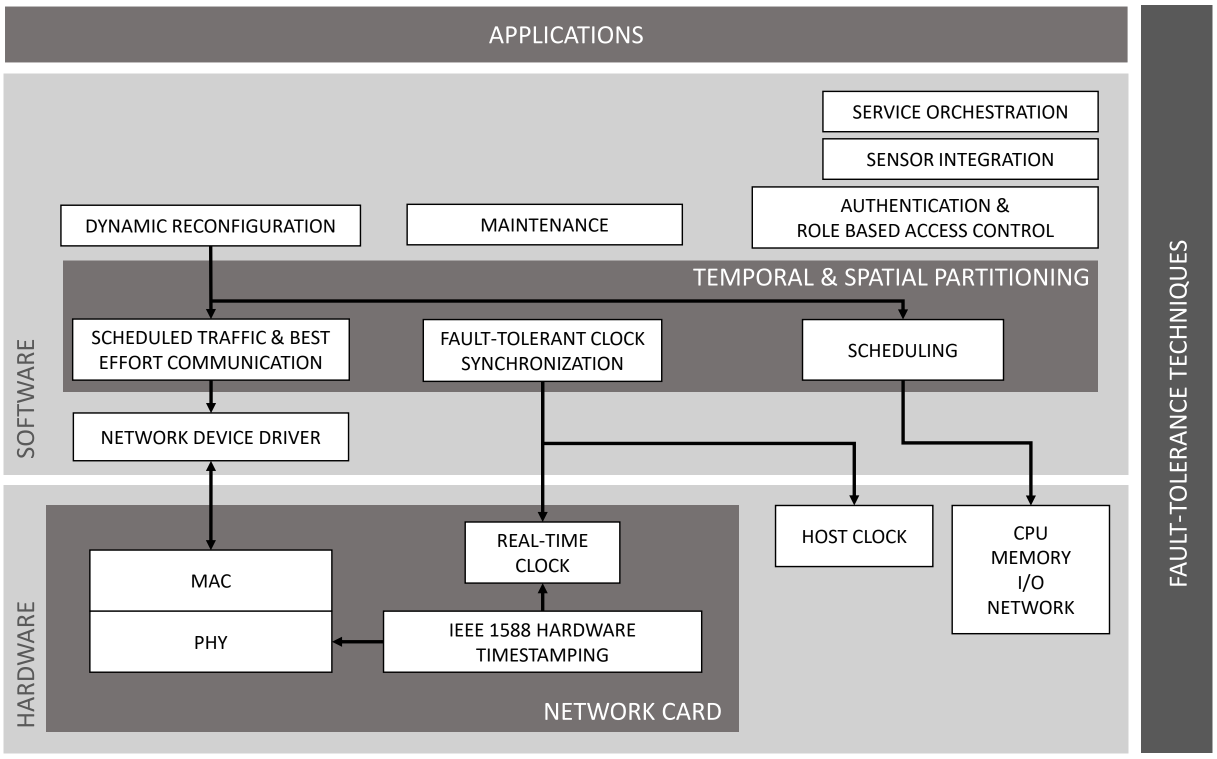

APPLICATIONS
SERVICE ORCHESTRATION
SENSOR INTEGRATION
AUTHENTICATION & ROLE BASED ACCESS CONTROL
DYNAMIC RECONFIGURATION
MAINTENANCE
TEMPORAL & SPATIAL PARTITIONING
SCHEDULED TRAFFIC & BEST EFFORT COMMUNICATION
FAULT-TOLERANT CLOCK SYNCHRONIZATION
SCHEDULING
NETWORK DEVICE DRIVER
SOFTWARE
HOST CLOCK
CPU MEMORY I/O NETWORK
REAL-TIME CLOCK
MAC
PHY
IEEE 1588 HARDWARE TIMESTAMPING
NETWORK CARD
HARDWARE
FAULT-TOLERANCE TECHNIQUES

Figure 5.1: Proposed services - Overview

Figure 5.1 gives an overview about the services of the service gateway regarding their position within the different layers of the presented architecture. A relevant component at hardware level is the network interface with its real-time clock and an IEEE 1588 compliant hardware timestamping implementation. This compliance is of special importance since it is required to implement real-time communication based on scheduled traffic. Further, there is the clock of the host system which is synchronized to the real-time clock of the network card by the clock synchronization service. The synchronization of the host clock is required in order to establish the same notion of time in all service gateways for the establishment of distributed real-time applications that are scheduled by the task scheduling of the temporal and spatial partitioning service. The real-time clock of the network card is in turn synchronized to the clocks of all other communication nodes within the network. This is achieved by using the IEEE 1588 standard. Besides the mentioned service for temporal and spatial partitioning, there are further services for dynamic reconfiguration, service orchestration, sensor integration, maintenance, authentication and role-based access control. Figure 5.1 further shows the introduction of fault-tolerance techniques at the level of hardware and software, as introduced in Section 4.4. Some parts of this chapter were published in [SO18][SO17][SOW18].

5.1 Service Orchestration and Discovery

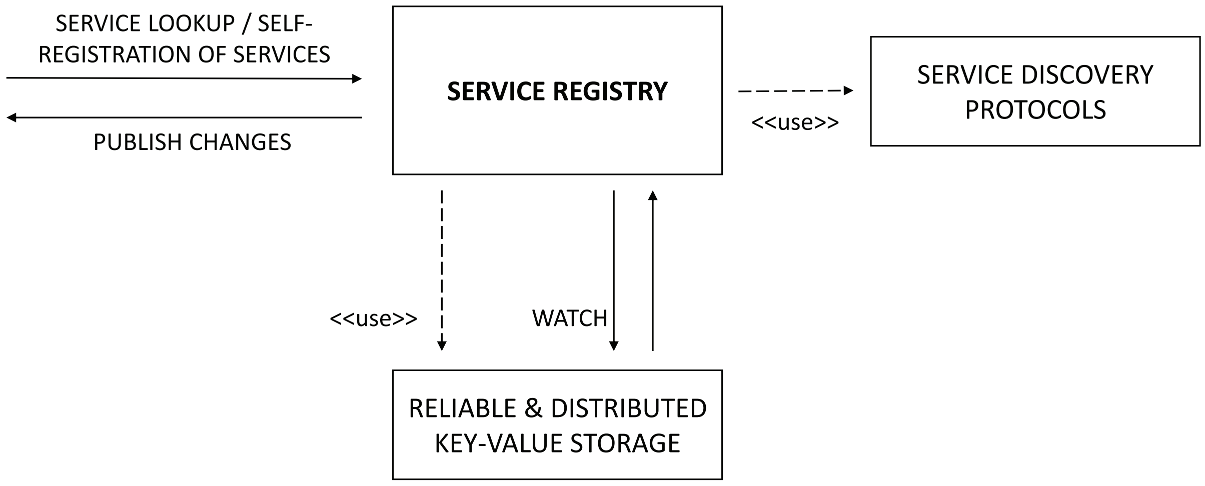

SERVICE LOOKUP / SELF-REGISTRATION OF SERVICES
PUBLISH CHANGES
SERVICE REGISTRY
<<use>>
SERVICE DISCOVERY PROTOCOLS
<<use>>
WATCH
RELIABLE & DISTRIBUTED KEY-VALUE STORAGE

Figure 5.2: Service Orchestration

As introduced in the challenges, a service-oriented architecture requires techniques for registration and discovery of services. Typically, a central or distributed service registry is applied where services can register themselves (service self-registration) or can search (service lookup) for other services. A service registry can also actively scan the local network for new services or devices (service discovery). Likewise, the service registry has to notify services in case of changes to the service registry, which includes the addition, removal or failure of services. In particular, the latter case requires techniques to keep the knowledge of existing services up-to-date. A typical example to address this challenge is the sending of periodic keep-alive messages by the services towards the service registry. Further, the information about the services has to be stored in a reliable way in order to prevent a failure of the service registry in case of faults.

As Figure 5.2 shows, the main component for the service orchestration is the service registry. The service registry provides an interface for services for self-registration and for service lookup. As storage backend, the service registry relies on a reliable and distributed key-value storage. The service registry further uses techniques and protocols for service discovery like the SLP, DNS Service Discovery (DNS-SD) and Zero-configuration networking (Zeroconf) (cf. Section 2.7.4). As already mentioned, services can use the service registry to register themselves or to search for needed services. Here, the interface supports both, message-based requests (e.g. AMQP) and REST based request. This is required since a service may not yet know the contact details for the best-effort communication service implemented by a message broker which is used to provide the message-based interface of the service registry. Since REST introduces dependencies among services (cf. Section 2.7.3), the message-based interface should be the preferred interface used by services.

Services are registered at the service registry associated with their contact information (e.g. IP-Address of the container and port) by using a hierarchical namespace stored in the key-value store of etcd. Figure 5.3 shows the standard hierarchy. Each service gateway has its own namespace identified by a unique Universally Unique IDentifier (UUID), e.g. *78ffb17c-6c65-11ea-bc55-0242ac130003*. The UUID algorithm used is based on a timestamp and the Media Access Control (MAC) address of the service gateway and the UUID is generated at the first start of the service gateway. At the first level of the hierarchy, services are grouped by the main categories, namely platform-services, user-services, sensors and messaging-services. The namespace tree in 5.3 further contains several sensors (for body temperature, blood pressure, blood sugar level, SpO2 and heart-rate) and user-services (user-interface and a database). Some services are optional like the RT-Broker (cf. Section 5.9.2) as well as the real-time and rate-constraint communication services. Multiple services or sensors for the same purpose might be present (e.g. two temperature sensors in the kitchen) at the same time, and each service will get a unique ID at the first registration. This is especially important for sensors to be able to distinguish between their historical values in later stages. In addition, the presented namespace hierarchy is flexible in a way that new types of sensors can be added without creating a namespace for the sensor type first since the namespace for the sensor type is created automatically.

Algorithm 1 Service Lookup at foreign service registries and interconnection server

```
1: procedure ServiceLookup($service$)
2:     if $service.UUID == serviceGateway.UUID$ then
3:         lookup local key-value store for requested service
4:         if $result.length() \geq 1$ then
5:             filter($result$, $service.desiredAttributes$)          ▷ e.g. reliability, resolution
6:             return $result$
7:         else
8:             return $emptyList$
9:     else if $service.UUID = interconnectionServer.UUID$ then
10:        forward request to interconnection server
11:        wait for reply                                                 ▷ Using timeout
12:        return reply from interconnection server
13:    else if $service.UUID != ""$ then
14:        if $foreignRegistries.hasEntry(service.UUID)$ then
15:            forward request to $foreignRegistries.getEntry(service.UUID)$
16:            wait for reply                                             ▷ Using timeout
17:            filter($result$, $service.desiredAttributes$)
18:            return reply
19:        else
20:            return $emptyList$
21:    else
22:        $entries$ = new Map();
23:        forward request to all known service registries
24:        while replies != num(knownServiceGateways) do                  ▷ Using timeouts
25:            wait
26:        combine replies in $entries$
27:        filter($result$, $service.desiredAttributes$)
28:        return $entries$
```

Further, the presented namespace hierarchy in combination with message-oriented communication based on publish-subscribe allows that services can apply wildcards (e.g. * and #) in the subscription. A consumer subscribing with the topic *sensors.kitchen.** would receive the data from all sensors in the kitchen. Likewise, the topic *sensors.#* would address all registered sensors. This also means, that with a careful designed hierarchical topic namespace new applications and sensors can be integrated into the system seamlessly without any action at the subscribers (e.g. performing a subscription for the new component).

In order to keep services up-to-date regarding changes within the service composition in the system (e.g. addition, removal or update of service information), the service registry uses mechanism from the key-value storage. A typical way of how key-value storage solutions notify services about changes are watches. Putting a watch on a key will result in a notification for the watch holder in case of any update of the key. The service registry also uses the mentioned concept of watches to keep track of changes within the key-value storage. Whenever the service registry gets a watch notification, it publishes a message through a dedicated service registry topic in order to notify all subscribed services.

Listing 5.1: etcd - Using the prefix option

```
$ etcdctl put service—gateway.78ffb17c—6c65—11ea—bc55— /
0242ac130003.sensors.kitchen.temperature "{'manufacturer':'GoodTemperature'}"
OK
$ etcdctl put service—gateway.78ffb17c—6c65—11ea—bc55— /
0242ac130003.sensors.kitchen.humidity "{'manufacturer':'GoodHumidity'}"
OK
$ etcdctl get ——prefix service—gateway.78ffb17c—6c65—11ea—bc55—0242ac130003.sensors
service—gateway.78ffb17c—6c65—11ea—bc55—0242ac130003.sensors.kitchen.humidity
{'manufacturer':'GoodHumidity'}
service—gateway.78ffb17c—6c65—11ea—bc55—0242ac130003.sensors.kitchen.temperature
{'manufacturer':'GoodTemperature'}
```

There are different approaches how current key-value storage solutions implement their namespace hierarchy internally. Apache ZooKeeper [APA16] for example handles the namespace hierarchy in hierarchical file-systems like structures whereas the reliable and distributed key-value store *etcd* [Clo20a] uses internally a flat namespace (since etcd version 3). That means that creating the key *service-gateway.78ffb17c-6c65-11ea-bc55-0242ac130003.platform-services.authentication* is handled as a single string by etcd. By this way, creating a hierarchy is very simple. The *prefix* option provided by etcd then allows to match anything that starts with a particular key. Listing 5.1 shows how the prefix option is used in etcd.

In order to support distributed applications, the service registry further is able to contact other service registries running at the same level (another service gateway) or running at a higher level (e.g. at the interconnection server). Likewise, other service registries in the local network may want to query for services located at foreign service gateways. In order to achieve this, the service registry announces itself in the local network. Other service registries now can use this announcement and add the corresponding service gateway to a list with known service gateways. Here, the UUID of the respective service gateway is used for storing the contact information of the foreign service gateway. Depending on the first part of the namespace of the desired service (which is the UUID of the service gateway), the service gateway now can decide, at which service gateway the service is located (cf. Algorithm 1). If the UUID matches the UUID of the local service gateway, the service registry directly can lookup the service details in the local key-value storage. If the UUID does not match with the one of the local service gateway, the service registry will forward the service lookup request to the respective service gateway. The response of this request is then forwarded to the service that made the initial request. In order to comply with data protection requirements, the foreign service registry will only reply with services which are explicitly announced as public services. It is also possible to lookup services without a UUID. In this case, the service registry will forward the service lookup request to all known foreign service registries. Compared to that procedure, the service registry can be configured to use a service registry located on the interconnection server. By this way, also services registered at the interconnection server in the Internet can be used. The presented service registry only handles best-effort services. Real-Time services are handled by the RT-Broker and the RT-Client (cf. Section 5.9.2 and Section 5.9.3).

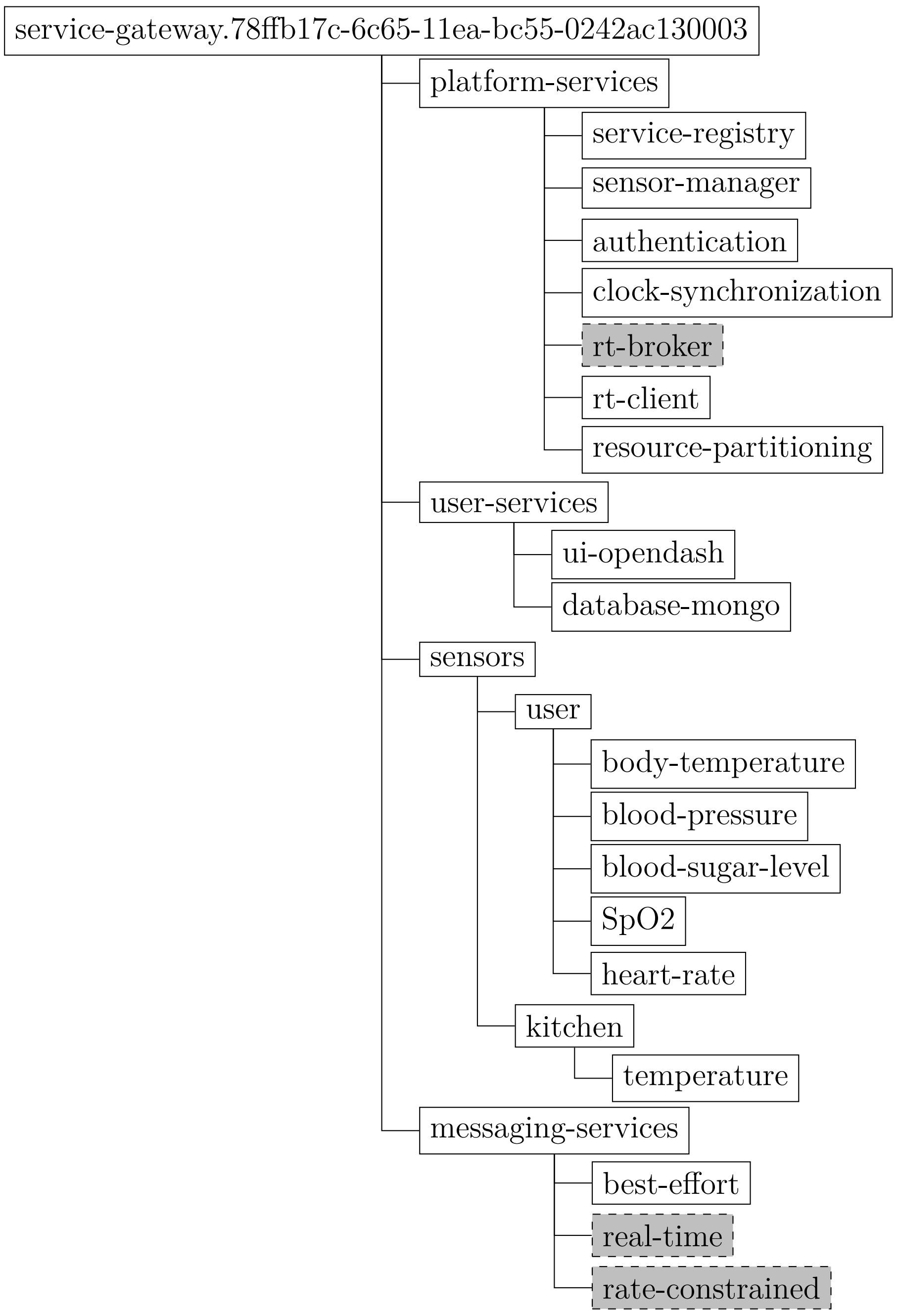


Figure 5.3: Hierarchical namespace tree - Example

5.2 Fault-Tolerance

As part of the fault-hypothesis in Section 4.4.1, which identified different failure modes and failure rates in the presented architecture, different fault-tolerance techniques were introduced. The presented techniques are means to establish fault-tolerance within the two identified FCUs, namely the service gateway and the communication links in the local network. Figure 4.3 shows both FCUs as part of the fault-model for the presented architecture.

Regarding the detection of temporal failures of the service gateways regarding network communication resources, the presented architecture provides two different approaches. The first approach involves the use of rate constraint communication in combination with traffic shaping at the service gateways and traffic policing at the network switches. At the service gateways, the presented architecture implements a traffic shaping layer for rate-constrained communication. The traffic shaping layer prevents nodes from producing more network traffic than allowed, which would cause for example buffer congestion in the switches. Secondly, the traffic shaping layer allows the calculation of a Bandwidth Allocation Gap (BAG) for Ethernet packets. This allows to isolate failures of nodes within the network, like faulty service gateways or sensors connected via the local network. The definition of the BAG is specific for each application. Regarding ISO/IEEE 11073, the BAG values could for example be integrated into the ISO/IEEE 11073 specialization standards as a priori knowledge. This would allow a seamless integration of ISO/IEEE 11073 compatible devices into the rate-constrained communication. The traffic shaping layer will also protect against misbehaving application components that for example monopolize the network by sending continuously data to the network (babbling idiot failures). Additionally, the traffic shaping layer will add priorities to the network traffic in a way that the network switches can apply traffic policing on the traffic based on the associated priorities.

Since Linux is used as the basis of the service gateways, the traffic shaping layer makes use of existing kernel features in order to establish fault-tolerance. One of these features is the Linux Traffic Control (LTC), which is a Linux kernel mechanism that determines, how IP packets are sent to the network. LTC includes features for classification, scheduling and policing of network traffic. Packets can be classified by a large set of filter mechanism like *dsmark* (Differentiated Services Marker, provides filtering for IP header fields), *iptables* or *u32* (filtering of complete packet data). The resulting traffic classes can be assigned to several queueing disciplines, that determine the scheduling of the packets. The default queueing discipline in most Linux distributions is PFIFO_FAST, which is a classless FIFO queueing discipline with three priority bands. A Linux distribution that for example is not using PFIFO_FAST is Arch Linux, which uses FQ_CoDel (Fair Queuing Controlled Delay) as default queueing discipline. In contrast to classless queueing disciplines which have just one level of queuing, classful queueing disciplines can have several levels and can be arranged hierarchically.

The traffic shaping features are realized within the network stack of the Linux kernel and support a full set of queueing disciplines like Hierarchical Token Bucket (HTB) and Hierarchical Fair Service Curves (HFSC). As already mentioned, the traffic shaping layer is required for several reasons. First, it prevents babbling idiot failures of software components, which jeopardize the network by constantly sending high-priority messages. Second, the traffic shaping layer establishes an a priori knowledge about the temporal activity of the service gateway on the network. Therefore, it is possible to detect temporal failures by comparing the activity pattern of the node with the a priori knowledge. Like in AFDX (Arinc 664 Part 7) [ARI09], the traffic shaping layer establishes rate-constrained communication where a minimum bandwidth is guaranteed for each communication channel. Here, the guaranteed bandwidth also ensures an upper bound for both jitter and delay.

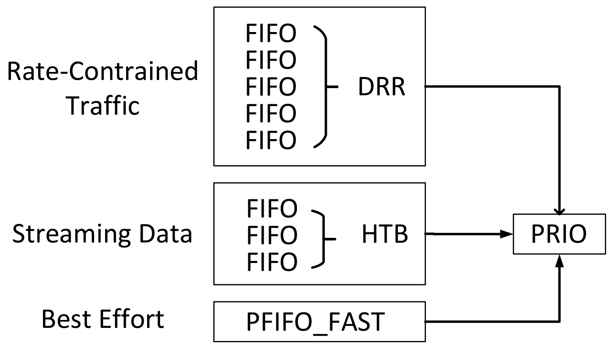


Figure 5.4: Linux Traffic Control (LTC) Configuration - Example

Figure 5.4 shows a configuration of LTC with three traffic classes: *rate-constrained*, *streaming data* and *best-effort*. These classes are scheduled using a priority (PRIO) queueing discipline. The highest priority is provided to the rate-constrained traffic class which is a Deficit Round Robin queueing discipline with multiple classes each containing a FIFO queueing discipline. These FIFO queueing discipline represents the virtual links of the rate-constrained communication. For each virtual link, the traffic shaping layer creates a virtual network interface. Data that is sent to such a virtual network interface is marked with a corresponding VLAN-ID which then is used by the LTC to classify the data for the corresponding FIFO queue (respective virtual link). The streaming traffic class is implemented by using a HTB queueing discipline. The last traffic class is for best-effort communication using the queueing discipline PFIFO_FAST queueing discipline.

The mentioned approach for rate-constrained traffic is reasonable for use cases that do not include distributed real-time applications with strict timing requirements in the low microsecond range. However, if distributed real-time applications with strict timing requirements are executed within the presented architecture, the second approach for the detection of temporal failures of the network nodes is more suitable, which is the introduction of scheduled traffic based on a real-time communication network. As introduced in Section 2.2, scheduled traffic is based on a communication schedule, which provides a-priori knowledge of the communication behavior (e.g. at which instant of time data shall be transmitted) of the involved network nodes. This a-priori knowledge allows for example the introduction of guardians at ingress ports of switches in order to prevent faulty communication nodes from sending packets at non-scheduled time points (e.g. in the event of a "babbling idiot" failure). In order to run the communication schedule at the service gateways, the presented architecture introduces a time aware network traffic scheduler on top of the network driver. Thus, this approach requires an underlying real-time communication network as introduced in Section 4.1 that must further be synchronized to a fault-tolerant global time base. The introduction of scheduled traffic further allows to establish a membership service. Since scheduled traffic is based on a time-triggered paradigm, the periodic communication time slots of the components are indirectly also membership points (life signs) [KGR91]. Therefore it is possible to detect the failure of a component by comparing the interval of two subsequent messages with the scheduled behavior.

The afore mentioned techniques for the establishment of fault-tolerance are all means that are applied at a higher level of the architecture, namely the service gateway and the communication links in the local network. Within the service gateway, techniques for software fault-tolerance can be applied (cf. Section 3.1). However, the application of fault-tolerance techniques like *Timeout*, *Retry* or *Fallback* lies within the scope of the applications running on the service gateway. Nevertheless, some means for fault-tolerance at software level are already provided by the presented architecture. One example is the application of a message broker that supports the queueing of messages in order to tolerate the absence of consumers (e.g. due to a failure) without messages getting lost. Further, the introduced temporal and spatial isolation (cf. Section 5.5) implements FCUs at the level of the services. The partitioning of the resources in the temporal and the spatial domain assures that a faulty service cannot affect further services in other FCUs. A typical example is a faulty service that runs into an infinite loop. Without a temporal isolation of the computational resources, the faulty service would have a negative impact on the whole system. This is not the case when temporal isolation of the computational resources is applied. Here, the computational resources are divided into several partitions and the services running within this partition can only use up the amount of resources that are allocated to the partition. In this way, the availability of resources within a partition is always guaranteed.

5.3 Best-effort communication

In the presented architecture, non real-time services are loosely coupled by using the message-based communication protocol AMQP. AMQP is an application layer protocol based on best-effort communication, which means that the transport latency for a message is not known a priori and thus not guaranteed. The delivery of a message can for example take 1 millisecond or 20 milliseconds depending on the route of the packets during their transport. A message may even be dropped, e.g. due to buffer congestion in the network switches. However, the best-effort communication must meet several requirements in the presented architecture, which are routing, queuing, reliability and security. Most of these four requirements are met by using a message broker for the message exchange. The most commonly used communication model of message brokers is publish-subscribe, where a message is distributed from the originating service (publisher) to one or more receiving services (subscribers). The *routing* of the messages between publishers and subscribers is typically based on so called topics (cf. Section 2.7.3). A topic is like a filter which is used by the message broker to determine to which subscribers the message has to be forwarded. As AMQP is an application layer protocol, the routing of messages also happens at the application layer between the producers and the consumers. The aforementioned routing is independent from the packet routing in the underlying network. Brokers that implement *queuing* are connecting the subscribers to the subscribed topics with an intermediate queue. This queue adds further *reliability* to the communication because it allows to tolerate the absence of subscribers for a certain period of time (where the length of time depends on the size of the queue buffer and the message sizes). The queuing mechanism also allows to update a service with a new version during the run-time of the system without any loss of data. Thus, the used message protocol AMQP operates at application layer, it is also feasible to use communication *security* protocols like Transport Layer Security (TLS) in order to establish a secure message communication. The routing of messages can further be extended to support the transport of messages between brokers, which is called message federating.

In order to use the communication services of the message broker, a service has to know the communication endpoint of the message broker. The communication endpoint depends on the used protocol. AMQP for example uses the Transmission Control Protocol (TCP) at port 5762 in the default configuration. As with all other services, details about the best-effort communication service provided by the message broker is made available for other services via the service registry (cf. Section 5.1). In Figure 5.3 details about the communication endpoints of the best-effort communication service can be found at the namespace *service-gateway.78ffb17c-6c65-11ea-bc55-0242ac130003.messaging-services.best-effort*. The namespace of the best-effort communication service after the service gateway UUID is always fixed and can therefore be integrated into the services in advance. However, the details of the best-effort communication service may vary (e.g. name or used port). This means that a service must contact the service registry immediately after its launch in order to obtain the communication endpoint of the best-effort communication service. Since a change to the best-effort communication service is possible (but unlikely), the service should further subscribe to change notifications of the service registry (cf. Section 5.1).

5.4 Real-Time Communication

Real-time communication is required for distributed services that have timing requirements regarding the communication latency. This precisely means that the message transport latency as well as the jitter in the communication network have to be bounded. These guarantees cannot be provided by a typical Standard Ethernet communication network. As introduced in Section 4.1, a real-time communication network is required in order to run distributed real-time applications. For example, TTEthernet and Time Sensitive Networking (TSN) are appropriate already available solutions that can be applied. Both real-time communication networks are explained in detail in Section 2.2.3 and Section 2.2.4.

As Figure 5.5 shows, appropriate network drivers handle the communication between the network interface and the operating system. A scheduler is responsible for injecting messages at scheduled instants of time to the real-time communication network. Linux for example introduced with kernel version 5.2 the Time-Aware Priority Shaper (TAPRIO), which is a queueing discipline that implements (a simplified version of) the Time-Aware Shaper (TAS) defined in the enhancements for scheduled traffic in IEEE 802.1Qbv (cf. Section 2.2.4). The basic idea of the enhancements for scheduled traffic defined in IEEE 802.1Qbv is to block non time-sensitive traffic in reserved time intervals in order to have an idle port for sending time-sensitive traffic. The TAS allows the transmission of packets in an end station or a bridge by predefined time schedules (called scheduled traffic or protected traffic). By this way, the TAPRIO queueing discipline can partition resources from a network adapter and thus allows to provide temporal partitioning for applications regarding the network resources. Another requirement for real-time communication is to

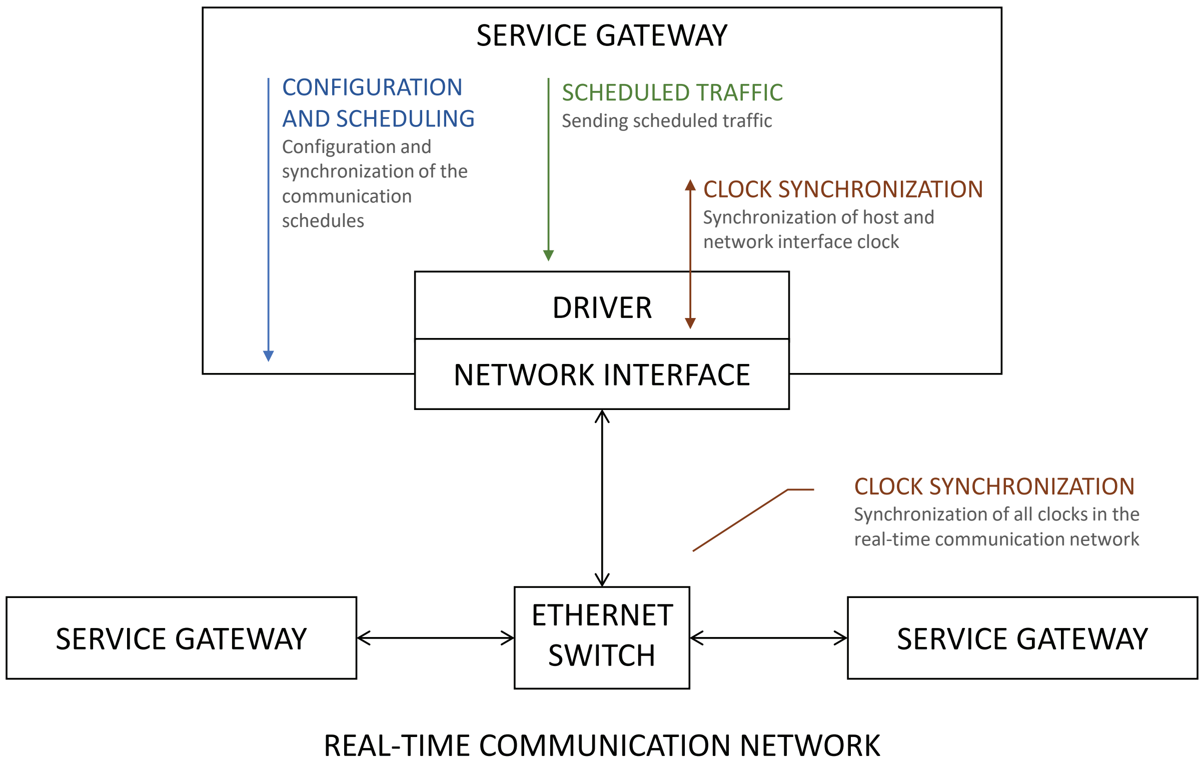

SERVICE GATEWAY
CONFIGURATION AND SCHEDULING
Configuration and synchronization of the communication schedules
SCHEDULED TRAFFIC
Sending scheduled traffic
CLOCK SYNCHRONIZATION
Synchronization of host and network interface clock
DRIVER
NETWORK INTERFACE
CLOCK SYNCHRONIZATION
Synchronization of all clocks in the real-time communication network
SERVICE GATEWAY
ETHERNET SWITCH
SERVICE GATEWAY
REAL-TIME COMMUNICATION NETWORK

Figure 5.5: Real-Time communication - Overview

have all clocks synchronized in the network. As applications and services also have to provide their results and messages at the correct instant of time, the clock of the service gateway has to be further synchronized to the clock of the network card. Therefore, clock synchronization in distributed real-time systems (as it is the case with the presented architecture) is typically a two-step process. At first, all clocks of the network cards are synchronized. Subsequently, the clock of each service gateways is synchronized to the clock of the network card.

The real-time communication service must consider two different scenarios. The first scenario covers the real-time communication between service gateways, where a real-time service located on service gateway A wants to communicate with another real-time service on service gateway B. However, this requires an underlying real-time communication network like TSN. The second scenario considers two real-time services located on the same service gateway. Since the implementation of the temporal and spatial isolation service is based on Linux Containers (LXC) in order to isolate services with different criticalities, the real-time services must communicate across the border of their respective container environment. This requires a real-time container-to-container communication. A communication via shared memory is not possible since the containers are also isolated regarding their memory.

In order to support a dynamic system with real-time service entering and leaving at run-time of the system, techniques for dynamic scheduling and configuration of the real-time communication resources are also required. Section 5.9 addresses this challenge using TTEthernet. For TSN, there is currently no configuration framework available which addresses all aspects of the reconfiguration process. However, there are some initiatives that aim to address this topic by incorporating techniques from the TSN standard like netconf [EBBS11]. One example is AccessTSN [LHI20], a publicly funded research project which was kicked-off in May 2018. AccessTSN pursues several objectives like the establishment of a vendor-independent application interface as an abstraction layer for the different TSN standards and the Linux system configuration and management tools for TSN and AccessTSN.

5.5 Temporal and Spatial Partitioning

The temporal and spatial partitioning service belongs to the non-functional services presented in this thesis. Non-functional means that this service provides no functionality to the applications and services running on the service gateway. The main goal of this service is to establish temporal and spatial isolation among all services and applications running on the service gateway. As introduced in Section 4.3.5, spatial partitioning separates multiple resources into subsets (e.g., one CPU in a multi-core processor) or areas (e.g., memory) while temporal partitioning assigns time-slots in which access to a shared resource is granted. This isolation can for example be established by the introduction of a hypervisor at OS level (cf. Section 2.6).

Figure 5.6 shows an overview about the different components of the temporal and spatial partitioning service. Since the service must be provided at a very low level of the system (e.g. at the level of the operating system), the presented approach introduces a hypervisor at OS-level (type-2 hypervisor) which uses mechanisms from the kernel of the operating system to provide virtualization for the resources of the host (cf. Section 2.6). However, the isolation provided by the hypervisor at OS level does not provide complete spatial and temporal isolation of hardware resources. For a comprehensive isolation of all resources, further techniques have to be applied.

The resources that have to be isolated depend strongly on the applications. A typical application will use CPU, memory and I/O resources. Applications in distributed systems

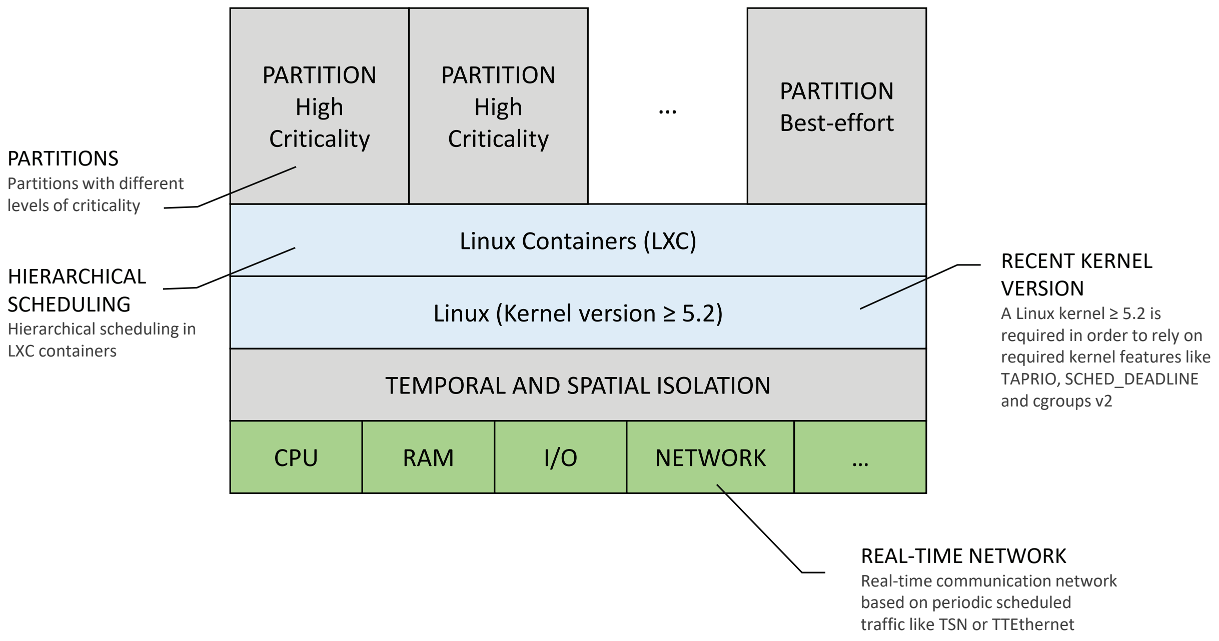

PARTITION High Criticality
PARTITION High Criticality
...
PARTITION Best-effort
PARTITIONS
Partitions with different levels of criticality
Linux Containers (LXC)
HIERARCHICAL SCHEDULING
Hierarchical scheduling in LXC containers
Linux (Kernel version ≥ 5.2)
RECENT KERNEL VERSION
A Linux kernel ≥ 5.2 is required in order to rely on required kernel features like TAPRIO, SCHED_DEADLINE and cgroups v2
TEMPORAL AND SPATIAL ISOLATION
CPU
RAM
I/O
NETWORK
...
REAL-TIME NETWORK
Real-time communication network based on periodic scheduled traffic like TSN or TTEthernet

Figure 5.6: Resource Partitioning - Overview

also may use network resources to fulfill their tasks. Table 5.1 shows an overview of the intended technologies used to achieve spatial and temporal partitioning of shared resources in Linux.

The temporal and spatial isolation by the use of an OS based hypervisor introduces a further requirement to the scheduling of tasks. Since the tasks that run within a container are executed within a separate PID hierarchy that is located under the hierarchy of the underlying operating system, the overall scheduling of all tasks running on the service gateway has to be performed in a hierarchical manner. Here, the presented architecture incorporates the results achieved by Luca Abeni et. al [ABC19], who proposed a real-time deadline-based hierarchical scheduling policy to provide temporal scheduling guarantees to different co-located containers.

Spatial isolation of the network is also provided by Linux Namespaces. However, the temporal partitioning of the network resources is special here, since it puts further requirements to the underlying platform. For example, network hardware that supports scheduled traffic is required in order to achieve the required temporal partitioning of the network resources. Besides proprietary hardware for the TTEthernet protocol, there is already first hardware available for TSN. An example is the Intel®Ethernet-Controller I210, which is a Gigabit-Ethernet controller that supports precise clock synchronization and hardware timestamping (IEEE Standard for a Precision Clock Synchronization (IEEE 1588)) and Forwarding and Queuing Enhancements for Time-Sensitive Streams (IEEE 802.1Qav).

	CPU	RAM	Disc I/O	Network
SCHED_DEADLINE	x$^{(TP)}$			
Memguard		x$^{(TP)}$		
Linux control groups	x$^{(SP)}$	x$^{(SP)}$	x$^{(TP)}$	x$^{(SP)}$
Disk Partitions			x$^{(SP)}$	
TAPRIO				x$^{(TP)}$

TP: Temporal Partitioning
SP: Spatial Partitioning

Table 5.1: Technologies for spatial and temporal partitioning of shared resources in Linux

TSN switches are also already available like the LRTN16R from TrustNode that supports the following protocols: IEEE 802.1AS, IEEE 802.1Qbu (preemption of non-critical data streams in case of a higher priority data stream), IEEE 802.1Qbv, IEEE 802.1CB and IEEE 802.3br (enhancements for the precision of frame preemption).

For the temporal partitioning of the network resources, the presented architecture further relies on TAPRIO which is a queueing discipline (QDISC) that implements (a simplified version of) the TAS defined in the enhancements for scheduled traffic in IEEE 802.1Qbv. TAPRIO was introduced with the mainline Linux Kernel in version 5.2 and thus a recent Linux kernel version with a minimum version 5.2 is required for the temporal and spatial partitioning service.

Since Linux control groups only provide the spatial partitioning of memory, other technologies are required to gain the temporal isolation of the memory. One possible technology is MemGuard, which is a memory bandwidth reservation system that “distinguishes memory bandwidth as two parts: guaranteed and best effort” [YYP+13]. MemGuard supports multi-core platforms and is available as a Linux Kernel module. Cgroups can also be used to achieve temporal isolation of I/O by limiting the I/O throughput of a process. However, the limitation of I/O only performs correctly since the introduction of the new generation of cgroups (cgroup v2). The popular software Docker currently still relies on cgroups v1 and therefore is not able to handle the temporal isolation of I/O correctly. In contrast, Linux Containers (LXC) is already using cgroups v2 and is therefore used in the presented architecture.

However, applying a hypervisor as a basis for the temporal and spatial isolation also puts a further challenge on the real-time communication. In particular, the communication now follows an hierarchical model where the communication between different containers and the communication with other nodes in the network has to be coordinated and scheduled (cf. Section 6.2).

The following sections present the different Linux kernel technologies used within the implementations of the temporal and spatial partitioning service.

5.5.1 Linux Kernel Features and Techniques

Linux Control Groups

Linux control groups (cgroups) is a Linux kernel feature that allows to organize processes into hierarchical groups that can be configured to limit and to monitor resources (e.g. memory and CPU) of the processes associated to the group. While the organization of groups is a feature of the cgroups core, cgroups controllers are responsible for the distribution of resources along the resource hierarchy tree. A process belongs to exactly one cgroup and all threads of this process and later forked processes belong directly after their creation to the same cgroup.

As with Linux kernel 4.5, a new cgroup version (cgroup v2) was marked as non-experimental [Ekl20]. The main difference compared to v1 is that there is only a single hierarchy for all resources instead of one hierarchy per resource. However, not every controller that was available in cgroup v1 is yet available in cgroup v2. At present, cgroup v2 supports the following controllers: CPU, Memory, IO, ProcessID (PID) (e.g prohibit forking of processes), device controller (e.g. manage access to device files), Remote Direct Memory Access (RDMA) as well as some other miscellaneous controllers.

Linux Namespaces

Linux namespaces is a Linux Kernel feature that allows to define virtual instances of the resources of the host and the Linux Kernel. The virtual instances are completely separated from each other and members of one instance cannot see or modify the resources of another instance. That means that processes within the namespace have their own isolated instance of one global resource. Linux namespaces are a key feature when it comes to container based virtualization making resource assignments to a container completely isolated and abstracted from the rest of the system and other containers. Linux currently provides the following namespaces [LIN18]:

- Cgroups
- Inter-Process Communication (IPC)

- Network (provides isolation of network devices, IP protocol stacks (both IPv4 and IPv6), IP routing tables and firewall rules)
- Mount (provides isolation of the list of mount points)
- Process identifier (PID)
- Time (provided isolation for clocks)
- User (provides isolation regarding security-related identifiers and attributes like user IDs and group IDs)
- Unix Time Sharing (UTS) (provides isolation for hostname and Network Information Service (NIS) domain name)

5.5.2 Linux Containers (LXC)

The introduced hypervisor LXC is a userspace interface for the containment features of the Linux kernel like kernel namespaces or control groups (cgroups). LXC is using both cgroups and namespaces to achieve the spatial separation of CPU and memory resources. Linux control groups and namespaces separate processes such that these processes cannot see the resources in other groups. For example, the network namespace allows to establish different and separate network interfaces with individual routing tables that are just visible to a dedicated namespace. These network namespaces then can be connected to the physical network interface by using a network bridge, which can be a standard Linux network bridge or for example an Open vSwitch (OVS) bridge.

5.5.3 Linux Real-Time Scheduling

In today's state-of-the-art there is a growing interest in making the Linux kernel suitable for real-time applications. Reasons are the design of Linux, which guarantees reliability and performance, and its open-source character that enables changes of the source code according to user needs [LSAF16]. Some approaches to achieve real-time guarantees in Linux are Xenomai [XEN19], the RealTime Application Interface (RTAI) [RTA18], the PREEMPT-RT patch [LIN20a] or SCHED_DEADLINE [LSAF16].

Xenomai uses two kernels. A microkernel is responsible to control real-time tasks during their time-critical operations. This kernel runs side-by-side with the regular Linux kernel on which the remaining tasks are running. This way, the microkernel provides very low latencies for real-time applications. Besides running in the microkernel, it is further possible to execute real-time tasks in the regular kernel [Yag08]. As the latencies are much higher in this case, the use of this scheduling technique depends on the requirements of the application. RTAI follows a similar approach, the main difference is that a thread migrating to the Linux space loses its real-time priority.

Fully Preemptive Kernel (PREEMPT_RT)

Besides the advantage of reaching real-time guarantees, both, Xenomai and RTAI have disadvantages in their applicability. Compared to regular Linux, they require a different API for real-time tasks. The PREEMPT-RT patch instead transforms the regular Linux kernel into a fully preemptive one without using a microkernel [LS11]. The developers replaced most kernel spinlocks by preemptive mutexes supporting priority inheritance and moved all interrupts to kernel threads. As a result, almost the whole kernel can be preempted except some small regions of code which significantly improves the maximum and average response time of the real-time tasks.

Using PREEMPT-RT, the kernel schedules tasks based on their priorities. Until kernel version 3.14 there were two scheduling strategies to select the next task to execute, First In First Out (SCHED_FIFO) and Round Robin (SCHED_RR). In SCHED_FIFO, the first available task with the highest priority is chosen which runs until it is preempted by a higher priority task, it terminates or until it relinquishes the CPU. Round robin uses timeslices so that a running task is preempted once it consumed all its execution time [LIN20e]. Both strategies suffer from starvation of lower priority tasks. Another disadvantage of priority-based scheduling is the possibility of priority inversion assuming a high-priority task tries to acquire access to a shared resource which was locked already by a low-priority task. The high-priority task has to wait for the low-priority task to release the resource. If a mid-priority task preempts the low-priority task, it also delays the high-priority task for an arbitrary amount of time. Hence, the task priorities are inverted. Using priority inheritance, the low-priority task receives the priority of the high-priority task during its critical section. Once it released the resource, the high-priority task can continue [SRL90]. Although arbitrary delays due to another task are limited, the execution of the high-priority task is still delayed by the WCET of the low-priority task's critical section. Hence, another mechanism to prevent such a situation is preferable.

Deadline Scheduling (SCHED_DEADLINE)

In addition to the drawbacks of priority-based scheduling, PREEMPT-RT does not give any guarantees on a task's temporal behavior such as deadlines. Due to these reasons, another real-time scheduling strategy called SCHED_DEADLINE was developed. The deadline scheduler was included into the mainline Linux kernel in version 3.14. It enforces temporal isolation between real-time tasks using resource reservation. Each real-time task can run for a maximum runtime Q_i every period T_i as long as the total utilization $\sum_{i=1}^{N} \frac{Q_i}{T_i}$ of all real-time tasks is below a certain threshold [LSAF16]. The combination with Linux control groups in [ABC19] can for example provide temporal scheduling guarantees to different co-located containers as a lightweight virtualization mechanism.

5.5.4 Memguard

At our current knowledge, there is no container based virtualization technology available that provides isolation of memory bandwidth, which makes temporal isolation among applications infeasible. One technology that can be used to achieve this isolation is MemGuard, which is a memory bandwidth reservation system for efficient performance isolation in multi-core platforms [YYP$^+$13]. MemGuard distinguishes two parts of memory bandwidth: guaranteed and best effort. For the guaranteed bandwidth, it provides bandwidth reservation to achieve temporal isolation with efficient reclamation to allow a maximum use of the reserved bandwidth. “It further improves performance by exploiting the best effort bandwidth after satisfying each core’s reserved bandwidth” [YYP$^+$13]. In the current version, MemGuard supports a per-task mode, where the task priority is used a weight for the bandwidth assignment. In relation to containers, this allows to guarantee a reserved memory bandwidth to the init process of the container.

5.6 Authentication and Role Management

An architecture based on microservices has many advantages such as the scalability of the services and flexibility in the choice of the programming language used. Since sensors and applications in the field of elderly care and Ambient Assisted Living (AAL) produce very sensitive and private data, the presented architecture supports authentication and authorization (based on roles), where authentication is the process of identifying the user and authorization refers to what the user is allowed to do.

However, authentication and authorization in a microservices based architecture are more challenging in comparison to a monolithic application design due to its distributed nature. In a monolithic application, authentication and authorization are typically implemented by a central security module. After the security modules has checked the identity of a user (e.g. by username and password), the module generates a session with all information about the user (e.g. permissions, roles, etc.) and returns the session ID (e.g. by using a Cookie) to the user. A common way to check the session in all requests of the client is to use a security interceptor that only allows the processing of the request in case the user has sufficient rights to execute the request. Security interceptors are also a typically used approach in services that are based on Representational State Transfer (REST). However, as mentioned in Section 2.7.3, REST is based on a synchronous communication model that brings dependencies among services. In addition, using a session for authentication and authorization prevents the services from being stateless, which hinders the scaling of the services, e.g. for load balancing reasons. Due to these reasons, there are different solutions available like JSON Web Tokens (JWT) [JBS15].

JWT is a standard for client tokens based on JavaScript Object Notation (JSON). In contrast to sessions where the information is stored on the authenticating part, client tokens are stored on the user side (e.g. in the LocalStorage of a Web-Browser). The client token holds the identity of the user and is a Base64 encoded String that is composed of three parts (concatenated with a dot), namely header, payload and signature. Listing 5.5 shows an example JWT Token that was computed by the header listed in Listing 5.2 and the payload from Listing 5.3. The signature is calculated and can be verified by applying the cryptographic hash function defined in the header of the JWT client token.

Listing 5.2: Example JWT Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Listing 5.3: Example JWT Payload

```
{
  "sub": "michael.schmidt@eti.uni−siegen.de",
  "name": "Michael Schmidt",
  "iat": 1584209607
```

Listing 5.4: Verifying a JWT

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  ft−rt−architecture−%&
)
```

JWTs can also be encrypted in order to provide secrecy between communication parties. For the presented architecture, signed JWTs as mentioned in the example above are sufficient since the signed tokens verify the integrity of the claims contained in the payload. Encrypted tokens hide the claims from other parties.

Listing 5.5: Resulting JWT Token

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJtaWNoYWVsLnNjaG1pZHRAZXRp /
LnVuaS1zaWVnZW4uZGUiLCJuYW1lIjoiTWljaGFlbCBTY2htaWR0IiwiaWF0IjoxNTg0MjA5Nj /
A3fQ.SW4hDpB6N—rIAGacYpKgyAr48pI5NIAdRoLkkymojBs
```

Client tokens like JWTs are typically generated at a central authentication authority like OAuth (cf. Figure 5.7) after the user (or a service) has provided the correct credentials (e.g. username and password). Since client tokens are credentials, these must be handled carefully to avoid security problems.

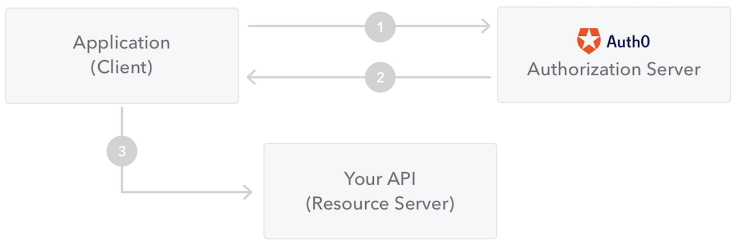


Figure 5.7: Getting a JWT client token from OAuth [AUT20]

5.7 Clock Synchronization

A fundamental basis of any distributed real-time system is clock synchronization among all involved components since the clock synchronization establishes a global view about time. This common view about time is for example required for the temporal ordering of events in the system. Taking data and events from sensor nodes as an example, clock synchronization allows to combine data from different sensor nodes (sensor data fusion) or to guarantee the overall consistency of the sensor data (temporal order). The main challenge with clock synchronization is that physical clocks have a varying drift rate that is “influenced by environmental conditions, e.g. due to a change in the ambient temperature” [Kop11a].

Clock synchronization can be established for example by applying a central master clock synchronization algorithm where a central master node sends periodically the value of its time counter in synchronization messages to all other nodes (slave nodes). By incorporating the delay of the synchronization message in the network, the slave nodes can calculate the difference between the clock of the central master and their own one in order to correct the local clocks. A further approach for clock synchronization in switched Ethernet networks is the IEEE 1588 Precision Time Protocol (cf. Section 2.2.2).

Clock synchronization in the presented architecture is established in a two-step process. The first step is the clock synchronization among all network hardware in the real-time network by using the IEEE1588 standard. In a second step, the clock of the network interface is synchronized with the clock of the service gateway.

5.8 Sensor Integration

When integrating sensors, a major challenge is the heterogeneity of the available sensors. Additionally, it should be possible to disable a certain sensor for example by the command of an elderly person or the nursing staff in order to comply with privacy needs. According to the microservices approach, sensors are integrated into the architecture as services and will use communication endpoints at the message broker to publish their sensor values. The communication endpoints are dedicated message exchanges (cf. Section 2.7.3) within the message broker which are administrated by the service registry. In the presented architecture, sensors are registered in the same way as services. Therefore, each sensor has its own namespace which is part of the namespace hierarchy introduced in Section 5.1.

There are two ways how sensors can be integrated. First, sensors can register themselves at the service registry providing detailed information like type, measurement units and metadata like an ontology membership for example. Second, ISO/IEEE 11073 compatible sensors are registered at the service registry by an ISO/IEEE 11073 service running on top of the open source ISO/IEEE 11073 software stack *Antidote* [Sig14].

Figure 5.8 depicts the procedure for sensors when registering the service registry. During the registration of a sensor, the service registry will return the assigned namespace and the communication endpoint where the sensor can publish data. Additionally, the service registry announces the newly integrated sensor on a dedicated message exchange at the message broker, providing information about the communication endpoint and namespace of the sensor. This information can be used for example by other services to react to the newly added sensors. One possible action could be the subscription to the communication endpoint using the provided namespace to receive the data of the newly added sensor.

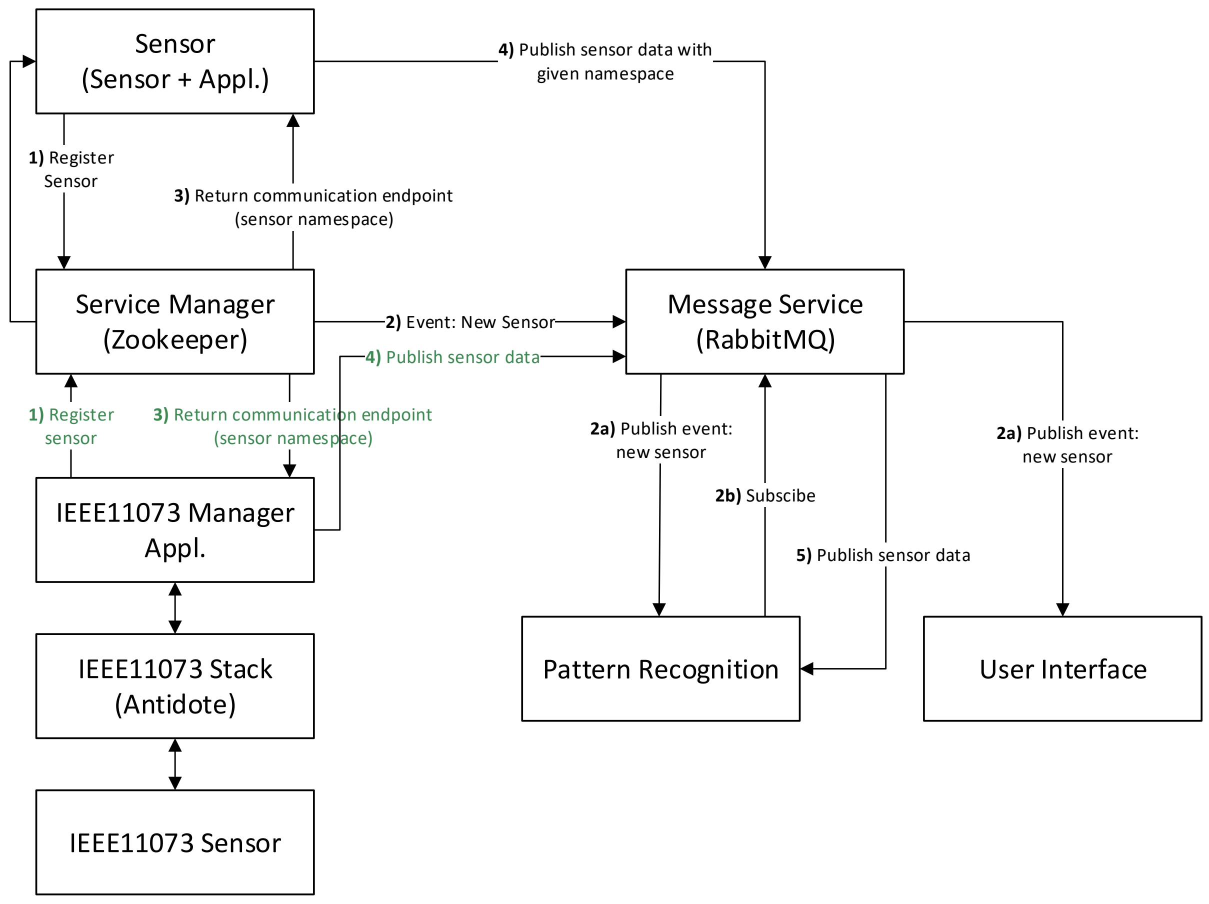

Sensor (Sensor + Appl.)
4) Publish sensor data with given namespace
1) Register Sensor
3) Return communication endpoint (sensor namespace)
Service Manager (Zookeeper)
2) Event: New Sensor
4) Publish sensor data
Message Service (RabbitMQ)
1) Register sensor
3) Return communication endpoint (sensor namespace)
2a) Publish event: new sensor
2b) Subscibe
5) Publish sensor data
2a) Publish event: new sensor
IEEE11073 Manager Appl.
IEEE11073 Stack (Antidote)
IEEE11073 Sensor
Pattern Recognition
User Interface

Figure 5.8: Registration of Sensors as services at the service registry

Discovery and peering of sensors based on ISO/IEEE 11073 is realized by implementing an ISO/IEEE 11073 compliant manager application on top of the *Antidote* ISO/IEEE 11073 software stack. Based on the state machine introduced by ISO/IEEE 11073, the manager will receive events from the stack whenever a device is associated to or disassociated from the ISO/IEEE 11073 stack. These events will be for example used to register the sensor at the service registry. Whenever the manager receives measurements for the associated sensors, the data will be published at the communication endpoint and the namespace provided by the service registry during the sensor registration process. Whenever a sensor is disassociated from the IEEE11073 stack the manager will de-register the sensor at the service registry.

One downside of the ISO/IEEE 11073 stack is the missing support for Bluetooth Low Energy (Bluetooth LE). In order to support Bluetooth LE as well as the ISO/IEEE 11073 stack, a sensor proxy with transcoding capabilities according to the Health Device Profile Implementation Guidance Whitepaper [Blu09] is necessary. Figure 5.10 depicts the data

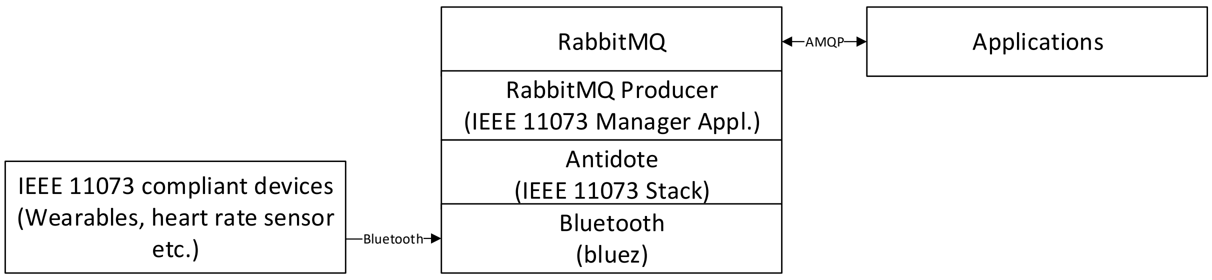

RabbitMQ
AMQP
Applications
RabbitMQ Producer (IEEE 11073 Manager Appl.)
Antidote (IEEE 11073 Stack)
IEEE 11073 compliant devices (Wearables, heart rate sensor etc.)
Bluetooth
Bluetooth (bluez)

Figure 5.9: Integration of ISO/IEEE 11073 compliant devices

flow of the communication model for Bluetooth LE devices. The implementation of the sensor proxy with the transcoding feature provides the ability to map between Bluetooth LE characteristics and ISO/IEEE 11073 device specializations. This will allow to integrate Bluetooth LE devices as well.

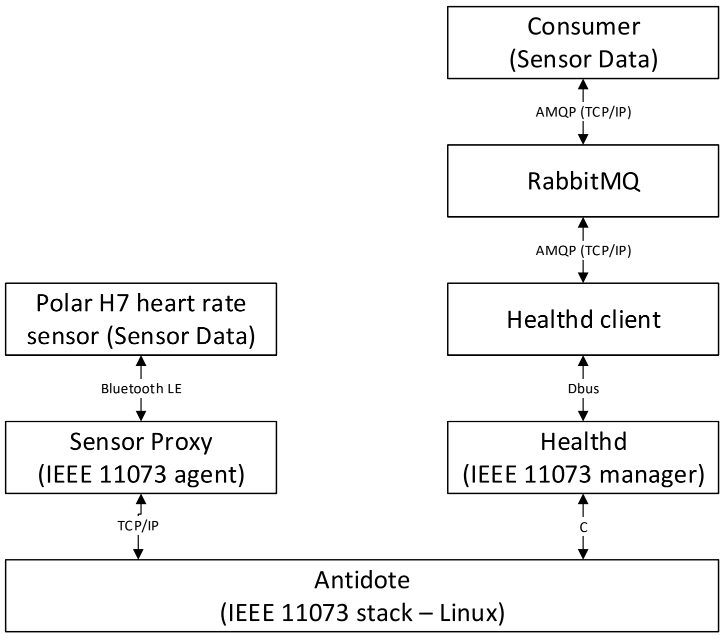

Consumer (Sensor Data)
AMQP (TCP/IP)
RabbitMQ
AMQP (TCP/IP)
Polar H7 heart rate sensor (Sensor Data)
Healthd client
Bluetooth LE
Dbus
Sensor Proxy (IEEE 11073 agent)
Healthd (IEEE 11073 manager)
TCP/IP
C
Antidote (IEEE 11073 stack – Linux)

Figure 5.10: Integration of Bluetooth LE devices into ISO/IEEE 11073 - Example for a Polar H7 heart rate sensor

5.9 Dynamic Reconfiguration of Real-Time Communication Resources

As illustrated in Figure 5.11, the presented approach introduces two services for the dynamic reconfiguration of real-time communication resources: namely the Real-Time Broker

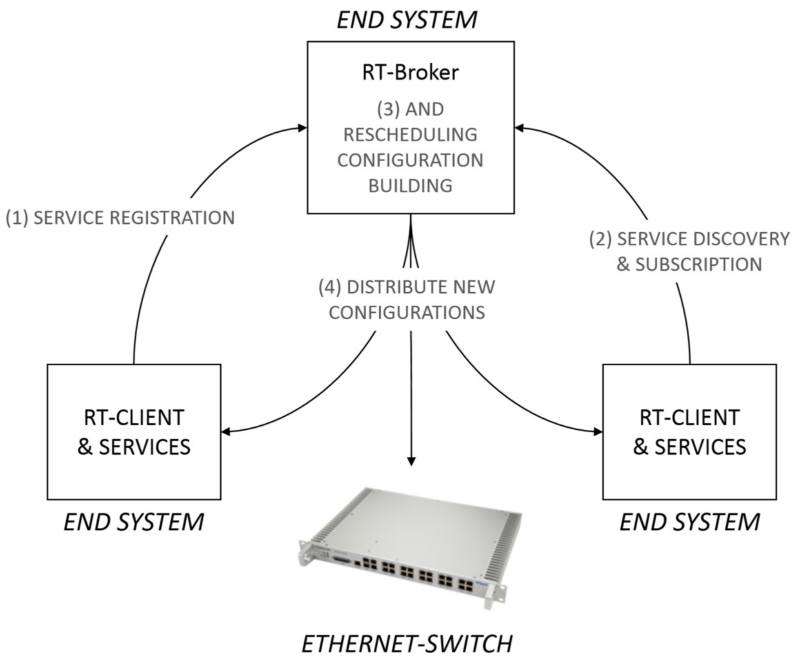


Figure 5.11: System Model

(RT-Broker) and the Real-Time Client (RT-Client). Both services are introduced in detail in Section 5.9.2 and Section 5.9.3. The RT-Broker is responsible for topology and service management, scheduling, configuration building and distribution of communication schedules. The RT-Broker is designed as a central instance and therefore exists only once in the network. The counterpart of the RT-Broker is the RT-Client, which has to be installed on a service gateway that runs services that want to use the real-time communication network. The RT-Client is responsible for the administration of the services running on the corresponding service gateway and informs the RT-Broker which services intend to use the real-time communication services of the network. The RT-Client also provides a data loading interface, which is used by the RT-Broker to push new network schedules for the real-time communication services running on the different service gateways.

Figure 5.11 shows a high-level overview of the intended behavior and responsibilities of the RT-Broker and RT-Client. As depicted, the RT-Broker waits for so called service offers and service usage requests of the RT-Clients. Service offer requests are sent by the RT-Client of the service gateways that offer real-time services. Service offer requests also contain information about the communication requirements of the offered real-time service. In contrast, service usage requests are sent by the service gateways that wants to use a real-time service (on a different service gateway). Whenever a service usage request arrives at the RT-Broker and real-time communication resources are available,

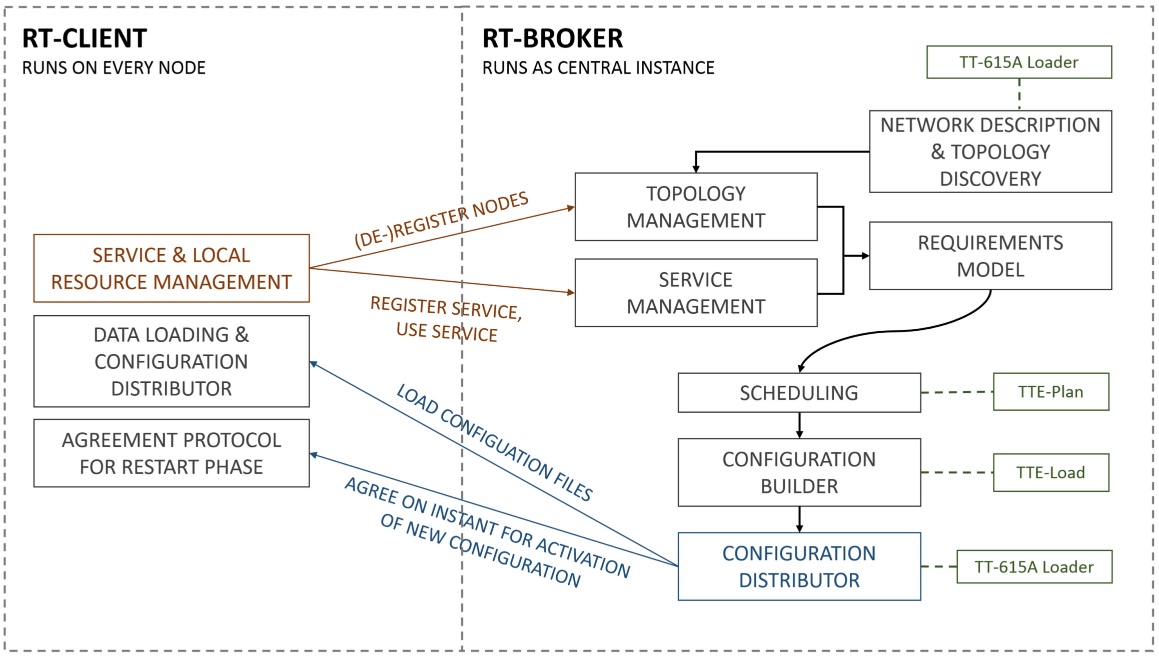

RT-CLIENT
RUNS ON EVERY NODE
RT-BROKER
RUNS AS CENTRAL INSTANCE
TT-615A Loader
NETWORK DESCRIPTION & TOPOLOGY DISCOVERY
TOPOLOGY MANAGEMENT
SERVICE MANAGEMENT
REQUIREMENTS MODEL
SERVICE & LOCAL RESOURCE MANAGEMENT
DATA LOADING & CONFIGURATION DISTRIBUTOR
AGREEMENT PROTOCOL FOR RESTART PHASE
(DE-)REGISTER NODES
REGISTER SERVICE, USE SERVICE
LOAD CONFIGUATION FILES
AGREE ON INSTANT FOR ACTIVATION OF NEW CONFIGURATION
SCHEDULING
TTE-Plan
CONFIGURATION BUILDER
TTE-Load
CONFIGURATION DISTRIBUTOR
TT-615A Loader

Figure 5.12: RT-Broker and RT-Client

a new communication schedule is calculated that includes the new service usage. After distributing the new schedules to the switches and the end systems, the RT-Broker and the RT-Clients perform a consistent restart of all schedules (as shown in the state machine in Figure 5.14). A detailed view on the internal structure and tasks of the RT-Broker and RT-Client is given in Figure 5.12. As it can be seen, the scheduling of the communication resources is not carried out by the proposed architecture. Furthermore, an external tool is used.

5.9.1 Challenges of Dynamic Resource Allocation

Before introducing the main concepts of RT-Broker and RT-Client, the following paragraphs discuss two scenarios that address the challenges of dynamic resource allocation in self-adaptive switched real-time Ethernet networks. The first scenario covers a fixed network topology as typically found in airplanes or cars. In a fixed topology, all network nodes (switches and end systems) are known a-priori and do not change at runtime. The second scenario follows an open-world assumption where network nodes can enter and leave at run-time. In both scenarios, the location and composition of services is not fixed and can change at runtime. It is assumed that the mentioned services are real-time services. After the introduction of the two scenarios, further challenges are discussed that arise whenever a network node like an end system or a switch has to be reconfigured. The description of the scenarios is generalized and hence abstracts from the presented architecture.

Fixed Network Topology The first scenario is based on a fixed network topology where all end systems, switches and interconnections are known a-priori. Within this topology, services are distributed among the interconnected end systems. A service is registered at a global service management where it can be discovered by other services. This enables dynamic service compositions where services can use other services in order to fulfill their own task. Further, the global service management must have a global view on the communication resources of the network and their usage by services. To establish and maintain this global view, services have to send a request to the global service management in order to bind and use other services. This provides the global service management with the capability to decide whether sufficient communication resources are available to fulfill the pending service usage request. If sufficient resources are available, the global service management can calculate a new communication schedule that has to be distributed to all network nodes. This scenario thus shows a set-up where the physical model (network topology, switches and end systems) is fixed, while the logical model can change (e.g. different service compositions, new services, etc.).

Dynamic Network Topology The second scenario is comprised by a dynamic network topology where network nodes can enter and leave dynamically during the run-time of the system. This may require new schedules for the communication resources and can affect already composed services, e.g. if a switch is removed that is part of a redundant path of a service usage. In this case, the composed service has to find an alternative redundant path. Within this scenario, the integration of a network node can either be introduced manually or be automatically detected by a management component. Ideally, this should happen automatically to avoid faults due to human failures. Though, this requires topology discovery mechanisms in order to update automatically the global view of all available communication resources and services. The presented scenario differs from the previous scenario in that the physical model can also change concurrently with the logical model.

Reconfiguration of Network Nodes In order to integrate new nodes into the network, new communication schedules have to be calculated and distributed to all network nodes. The manual distribution of new schedules to nodes is error-prone and time-consuming and does not allow the system to automatically adapt to new services and nodes. Therefore, this task has to be done autonomously. After the new schedules are distributed to all nodes, all affected nodes have to activate the schedule at the same instant of time.

5.9.2 Real-Time Broker

The RT-Broker fulfills several tasks in order to establish an autonomous and self-reconfigurable real-time network. As introduced, the RT-Broker is a server component and therefore has only a single instance on the network. The RT-Broker can be implemented on a dedicated management server or on any service gateway of the network. Figure 5.13 shows a detailed model of the RT-Broker and the RT-Client as Unified Modeling Language (UML) class diagram.

The following sections explain in detail the different parts of the RT-Broker with their underlying concepts. In order to show how to map these concepts to an underlying communication network, examples are given for the time-triggered communication network TTEthernet.

Topology Management Information about the network topology and details about the service gateways are maintained by a topology management sub-service. To keep the view of the network topology and details of the service gateways up to date, the topology management sub-service performs several distinct tasks, as follows.

With the topology management sub-service, the RT-Broker can automatically detect network devices that support the ARINC 615A-3 standard, like the TTEthernet switch (TTE Switch A664 Lab [TTT17c]) used in our experimental setup (cf. Chapter 7). For network nodes that do not support the ARINC 615A-3 standard, the RT-Broker and the RT-Client provide a new Data Loading Protocol (DLP). This protocol defines the integration process initiated by the RT-Client and must contain the relevant information about the device. In case of TTEthernet as the underlying communication protocol, the following parameters are mandatory:

- *name*: The name of the device (unique, used for identification)
- *type*: The device type (e.g. service gateway or network switch)
- *sync-role*: The role of the device regarding clock synchronization (e.g. synchronization or compression master)
- *ports*: A list of the physical ports of the device (including information like type or MAC-address)

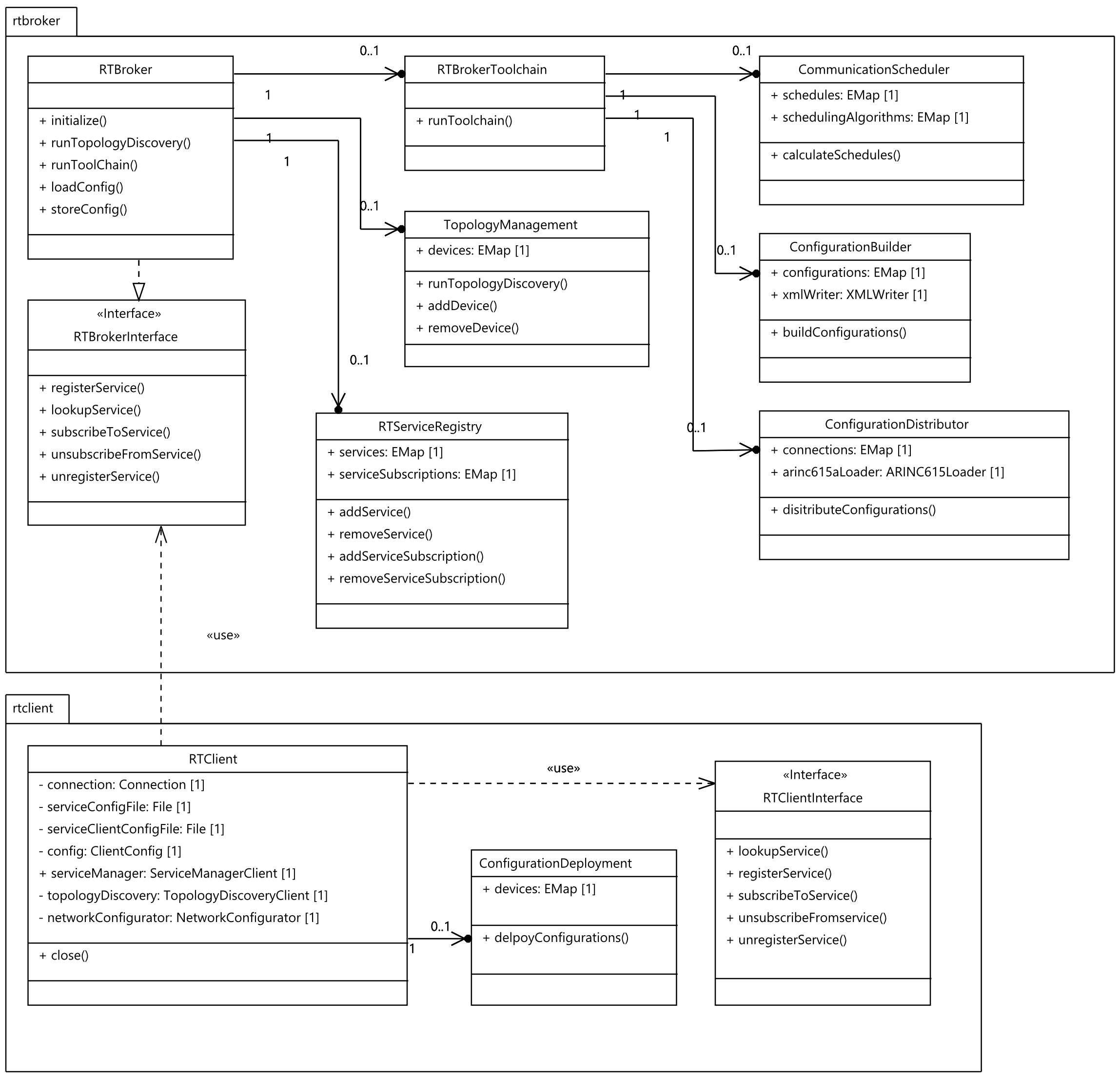


Figure 5.13: UML Class Diagram of RT-Broker and RT-Client

- *links*: A list of all physical links (including information like media type or cable length)

Depending on the device, the topology management sub-service of the RT-Broker decides which action has to be performed. In case of a switch, a new schedule will be calculated and distributed to all switches and service gateways. This reconfiguration is required because new redundancy options might be available in the system that are required by services. In the case of a service gateway, immediate reconfiguration is not necessary, since no services of the newly integrated service gateway are yet registered or used.

Service Management The service management sub-service of the RT-Broker maintains information about all real-time services in the network. By this way, the service management sub-service establishes a knowledge of the communication requirements and the location of all services in the network.

In case of TTEthernet, services can be registered by the RT-Client at the RT-Broker by providing the following information:

- *name*: Name of service (unique, used for identification)
- *device*: Device name (unique name of corresponding end-system)
- *messages*: Messages sent by the service

    name: Name of the message

    type: Message type (Time-Triggered or Rate-Constrained)

    frame-size: Maximum frame size

    redundancy-level: Desired redundancy level

    payload-size: Maximum payload size

Additionally, for time-triggered messages, the communication *period* has to be provided, in which the time-triggered messages are sent. For rate-constrained messages, the following additional information has to be provided: the *maximum allowed jitter* and the *bandwidth allocation gap* which defines the Minimum Inter-Arrival Time (MINT) between two subsequent rate-constrained messages.

Services can request other services by using the RT-Client, which will in turn send a service usage request to the RT-Broker. The RT-Broker first checks if a service is registered with the desired properties. If a corresponding service is found, the RT-Broker registers the service usage and carries out the rescheduling and reconfiguration of the network. If no corresponding service is found, the RT-Client is asked by the RT-Broker whether the service wants to wait for a registration of the desired service in the future. If this is the case, the RT-Broker will store the service usage request. The RT-Client can revoke the pending service usage request at any time.

If a service can no longer be provided, services can be de-registered at the RT-Broker by using the RT-Client. The RT-Broker will inform all services about this event so that they can react to the withdrawn service. In this case, the RT-Broker will not start a rescheduling and reconfiguration process in order to avoid downtime of the communication infrastructure due to the restart phase after activating the new communication schedules in the network. However, the freed resources are made available for future scheduling processes. Although no rescheduling is triggered by the RT-Broker at this moment, the rescheduling is most likely to take place because of changing service usages due to the withdrawn service.

A service usage can be revoked at any time using the RT-Client. The RT-Broker will not conduct a scheduling and reconfiguration due to the same reasons as during the de-registration of services.

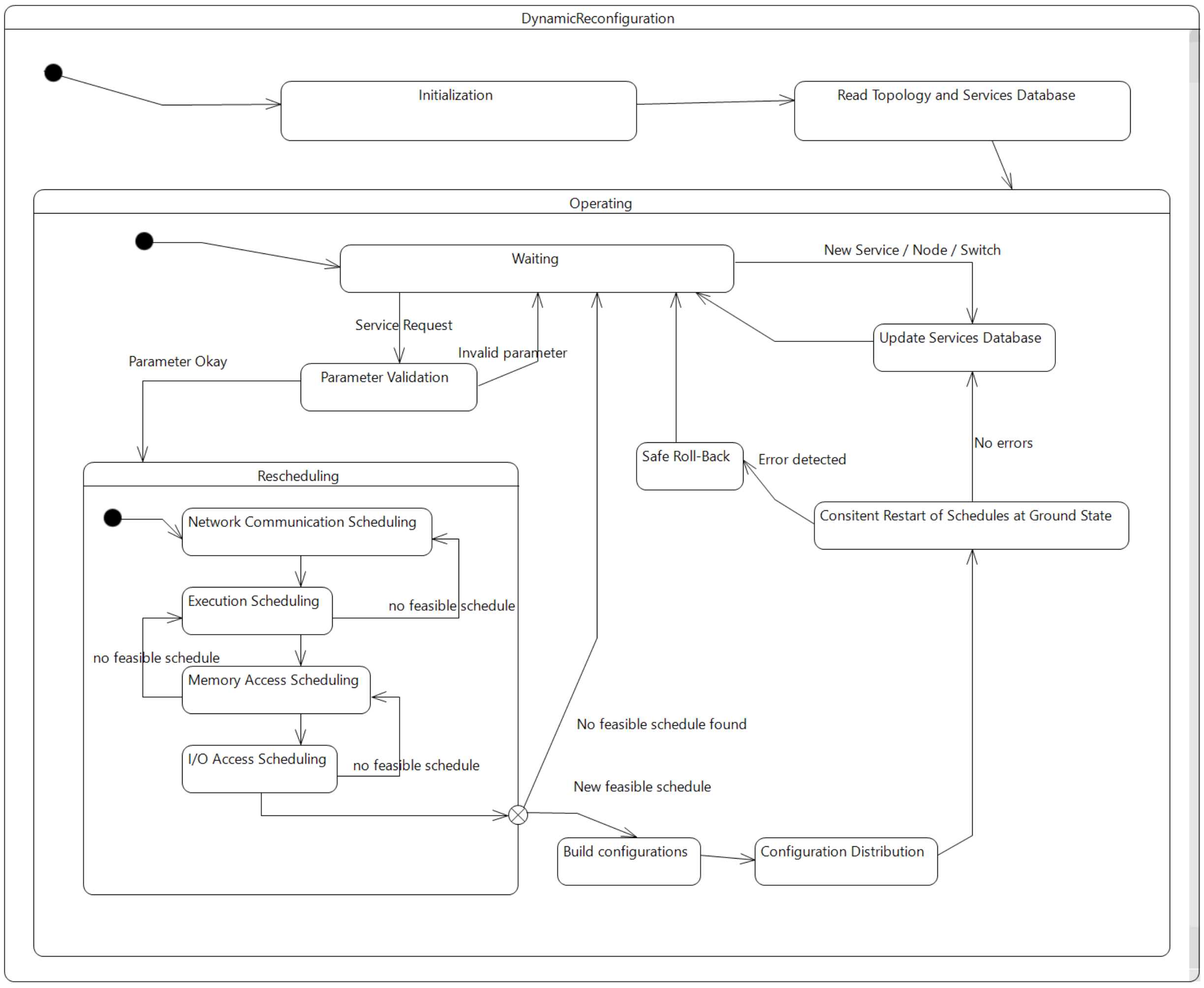


Figure 5.14: Finite State Machine (FSM) of dynamic reconfiguration service

Scheduling of the Communication Resources The global view of the topology and services is further transposed into a requirements model, which forms the basis for the scheduling of the real-time communication resources. Here, the scheduling is performed on the basis of established knowledge of the network topology by the service management sub-service. In most cases, the integration of a scheduler that is specific for the underlying communication protocol is the reasonable choice. For example, in case of our experimental setup we used the scheduler for TTEthernet that was provided by the manufacturer TT-Tech. In case of TSN, different solvers for the scheduling problems have to be used. A good insight and comparison of scheduling in TTEthernet and TSN based real-time networks is provided by [COA17]. In TTEthernet, the knowledge about the network topology is formally stored in an Extensible Markup Language (XML) file. This file contains for example information of all switches (ports, links, etc.), end systems (ports, partitions, etc.), links (e.g. name, transmission speed, media type, ports, cable length, etc.) as well as virtual links, periods and Bandwidth Allocation Gap (BAG) definitions.

Configuration Builder and Distributor After the new communication schedule is calculated, the configuration for the network devices is generated, based on the individual device information in the RT-Broker. This is done by the configuration builder sub-service of the RT-Broker. After the generation, the configuration is distributed to the network nodes by the configuration distributor sub-service. Similar to the topology management sub-service, the configuration distributor sub-service can upload the configuration to the network nodes in two ways, namely by using the RT-Client or by using ARINC 615A. As already mentioned, whenever the node does not support the ARINC 615A standard for loading new configurations, the node can use the RT-Client as data loading provider. For this purpose, the RT-Client implements a data loading sub-service. In the presented thesis, the data loading sub-service uses Secure Copy (SCP) as the underlying technology to upload the new schedules. However, any other data transfer protocol like Trivial File Transfer Protocol (TFTP) can also be used.

5.9.3 Real-Time Client

The counterpart of the RT-Broker is the RT-Client. The RT-Client has to be executed on every service gateway in the network that wants to run services relying on the real-time communication resources. All communication towards the RT-Broker is handled by the RT-Client. Most features of the RT-Client have already been introduced in the context of the RT-Broker. The RT-Client is mainly responsible for the management of the local services running on the service gateways which are using the real-time communication service. As shown in Figure 5.11, the RT-Client provides information about locally registered services and information about the end system self (e.g. link speed) to the topology and service management sub-services of the RT-Broker.

Further, the RT-Client implements the DLP which was introduced in Section 2.2.3. The main purpose of DLP is to distribute new configurations and schedules to the service gateways. After the RT-Broker has completely distributed all configurations and schedules, the RT-Broker negotiates with all RT-Client about the instance of time when the network schedules are activated.

5.10 Maintenance

In contrast to the other services, the maintenance service is a meta-service that is not provided to applications or other services. The maintenance service is provided towards the administrator of the service gateway and is composed of several subservices. These include services for monitoring the system status, installing updates, configuring system properties as well as managing task and network schedules. Since the proposed architecture does not include or prescribe a specific user interface, the maintenance subservices are provided as command line tools. Access to the command line tools is generally provided by a Secure Shell (SSH) connection. To easily determine the IP address of the service gateway, the service gateway publishes the contact details of the SSH service on the local network using the zero-configuration network protocol Avahi. Listing 5.6 shows the Avahi service file to advertise the SSH server on port 22 using Transmission Control Protocol (TCP).

Listing 5.6: Example Avahi service file to advertise the SSH server

```
<?xml version="1.0" standalone='no'?><!--*-nxml-*-->
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
  <name replace-wildcards="yes">%h</name>
  <service>
    <type>_ssh._tcp</type>
    <port>22</port>
  </service>
</service-group>
```

Although the Linux command line places additional demands on the knowledge of the system administrator, access to the command line offers many advantages. This provides an easy way to view log files or to install operating system updates, for example. While some of the maintenance sub-services are provided by the underlying operating system (e.g. installing operating system updates), a further part is provided by the other platform services described in this chapter. As an example, the real-time communication service implements a maintenance command line tool that allows to view the current communication schedule and to replace the current schedule with a new one. The following paragraphs outline the different available maintenance services.

Operating System. As mentioned, access to the operating system via SSH provides access to all of the different tools and services that the operating system provides. This includes for example services for installing operating system updates, viewing logfiles and monitoring of the resource utilization of different hardware resources.

Partitioning. The temporal and spatial partitioning service provides a maintenance tool for getting the status information of all involved partitioning techniques for the different hardware resources.

Real-Time Communication. The real-time communication service provides a tool to show all real-time task that are currently scheduled in the system. Further, the schedule of a particular task can be manually exchanged with a new schedule.

Clock Synchronization. The clock synchronization service provides a tool that allows to check for clock synchronization problems that may affect the precision of the clock synchronization.

In addition to the mentioned command line tools, many applications and services also provide a web interface, which is a convenient way to maintain a service. For example, the message broker RabbitMQ offers a comprehensive web interface for managing and monitoring all publishers, subscribers, topics and queues in the system. Furthermore, the message rates as well as queued messages are visualized which gives a good overview of the current load and performance to the administrator.

6 Implementation

As the previous chapters have shown, the establishment of a fault-tolerant and real-time capable architecture for elderly care is a complex task, which also puts requirements to the underlying platform (cf. Section 4.1). Several hardware building blocks (cf. Section 4.2) and platform services (cf. Section 4.3) are required to establish a platform that allows to run distributed real-time applications. Further, Section 4.4 has presented the different means by which the presented architecture achieves fault-tolerance. This chapter now introduces the details of the proof-of-concept implementations of the platform services presented in Chapter 5. As underlying operating system, a modern Linux distribution (Manjaro) with a recent kernel (5.6.16) was used for the proof-of-concept implementations. Linux is particularly well suited as a basis for an architecture for elderly care, as Linux provides a wide range of techniques required for the services presented. In particular, the virtualization techniques provided by the Linux kernel are important for the establishment of temporal and spatial isolation required to run critical applications. In addition, there are several solutions that enable Linux to run real-time applications. Another valuable aspect of Linux is that it is open source, which brings several advantages such as the use of open standards, transparency, stability and security.

6.1 Service Orchestration and Discovery

As introduced in Section 5.1, the service registry from the service orchestration service uses an underlying key-value store for managing the services in the service gateway. One feasible solution is introduced in Section 2.7.4, namely ZooKeeper from the Apache Foundation. Besides ZooKeeper, there are several other solutions available that can be used for setting up a key-value store. However, the solution for a fault-tolerant real-time architecture has to be reliable and support the intended namespace hierarchy of services and sensors (which are in fact services as well).

Three different technologies were evaluated for their suitability concerning the presented architecture, namely Apache ZooKeeper [APA16], etcd [Clo20a] and Consul [Has20b]. All three are very similar in their architecture, offer a hierarchical key-value store and are designed to be fault-tolerant and scalable. etcd is mainly a key-value store and is implemented in the programming language Go. It offers the definition of so-called *watches*. Watches are like callbacks which are triggered whenever a change to the key defined in the watch is detected. This for example allows a service to react to changes in the stored values. Further, etcd features reliability by supporting the distribution over multiple nodes by using Raft [OO14], which is a fault-tolerant consensus algorithm. Consensus is a fundamental problem in fault tolerant systems where multiple nodes have to agree on data values. The decision that all involved nodes have taken over the values is final, which means that the values cannot be changes at a later instant without finding a new consensus. Just like etcd, Consul is implemented in Go and is an open source project, but it is managed by a company called HashiCorp. In contrast to etcd, Consul provides much more features like dynamic load balancing, health checking and service discovery in large distributed systems. A comprehensive comparison of ZooKeeper, etcd and Consul is provided by [GM19] and [Clo20b].

For the presented architecture etcd is used. The biggest advantage of etcd are the provided reliability features. For example, etcd can operate over distributed etcd nodes for fault-tolerance and unlike ZooKeeper, the provided watch mechanism is reliable and drops no events even in case of the disconnection of a client. Furthermore, ZooKeeper watches are one-time triggers where clients have to set a new watch after a watch was triggered. Here, a major drawback is that events can get lost because of the latency between the received watch event and the registration of the new watch. etcd further implements role-based user permissions which perfectly fits into the presented architecture since it also uses a role based authorization model. Further, etcd offers libraries for almost every popular programming language and even provides a JSON-API via Representational State Transfer (REST) that allows to query etcd from the command line, e.g. with the command line tool curl. Based on etcd, the service registry allows applications and services to find each other. For storing services and their data, the service registry uses the hierarchical key-value store of etcd. In high-availability scenarios, distributed etcd nodes can be used to avoid the loss of the service storage of the service registry. Compared to ZooKeeper, etcd uses a flat namespace (keyspace in the terminology of etcd) hierarchy since version 3. This means that keys are not stored internally in hierarchical file-system like structures. Instead, keys like *service-gateway.78ffb17c-6c65-11ea-bc55-0242ac130003.platform-services.authentication* are handled as a single string. However, this allows the same flexibility as a hierarchical file-system while offering more consistency and efficiency in clustered systems. The prefix option of etcd allows to match anything that starts with a particular key value. For example, performing a *get* with the prefix *service-gateway.78ffb17c-6c65-*

11ea-bc55-0242ac130003.platform-services would result in etcd returning all entries of the registered platform services. Likewise, watches allow to use the prefix option to keep track of changes of multiple keys stored in etcd.

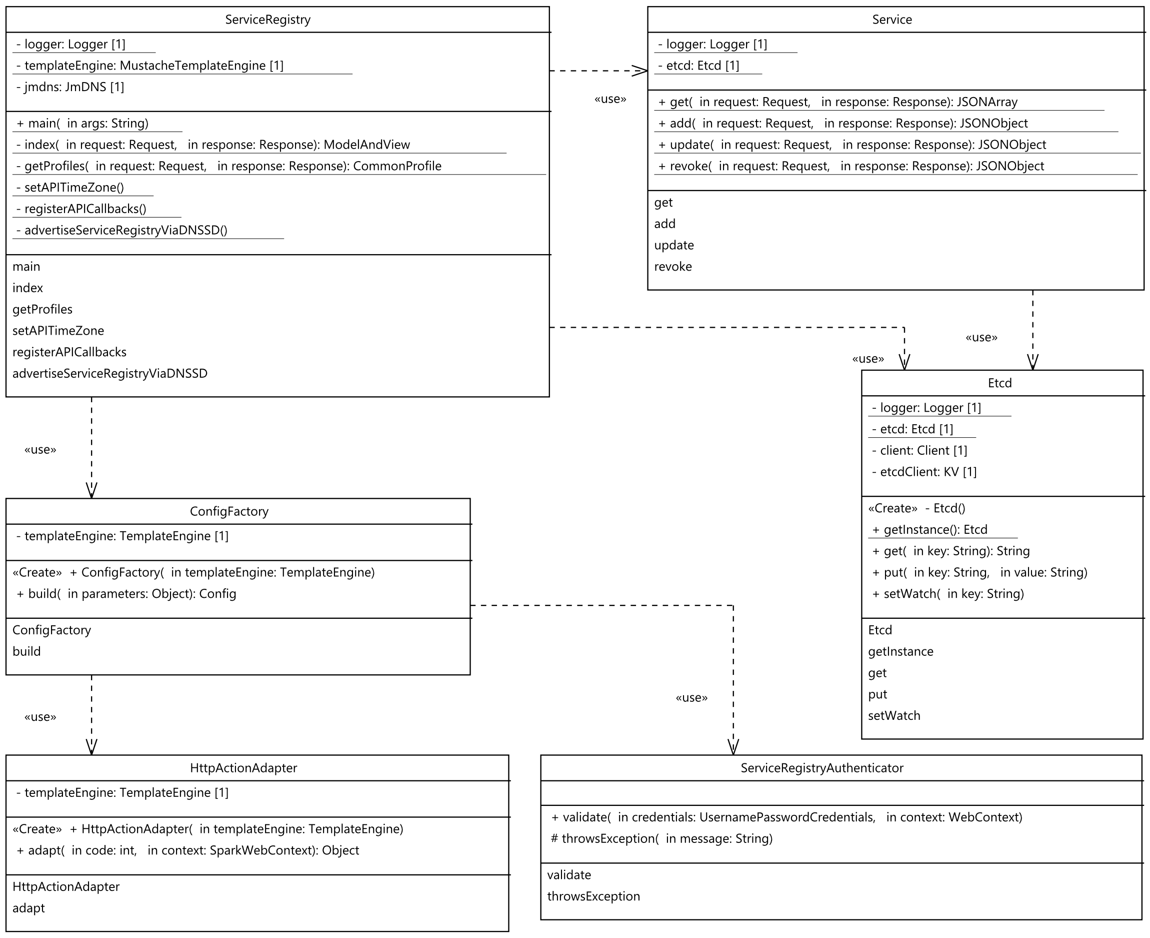

ServiceRegistry
- logger: Logger [1]
- templateEngine: MustacheTemplateEngine [1]
- jmdns: JmDNS [1]
+ main(in args: String)
- index(in request: Request, in response: Response): ModelAndView
- getProfiles(in request: Request, in response: Response): CommonProfile
- setAPITimeZone()
- registerAPICallbacks()
- advertiseServiceRegistryViaDNSSD()
main
index
getProfiles
setAPITimeZone
registerAPICallbacks
advertiseServiceRegistryViaDNSSD
«use»
Service
- logger: Logger [1]
- etcd: Etcd [1]
+ get(in request: Request, in response: Response): JSONArray
+ add(in request: Request, in response: Response): JSONObject
+ update(in request: Request, in response: Response): JSONObject
+ revoke(in request: Request, in response: Response): JSONObject
get
add
update
revoke
«use»
«use»
Etcd
- logger: Logger [1]
- etcd: Etcd [1]
- client: Client [1]
- etcdClient: KV [1]
«Create» - Etcd()
+ getInstance(): Etcd
+ get(in key: String): String
+ put(in key: String, in value: String)
+ setWatch(in key: String)
Etcd
getInstance
get
put
setWatch
«use»
ConfigFactory
- templateEngine: TemplateEngine [1]
«Create» + ConfigFactory(in templateEngine: TemplateEngine)
+ build(in parameters: Object): Config
ConfigFactory
build
«use»
«use»
HttpActionAdapter
- templateEngine: TemplateEngine [1]
«Create» + HttpActionAdapter(in templateEngine: TemplateEngine)
+ adapt(in code: int, in context: SparkWebContext): Object
HttpActionAdapter
adapt
ServiceRegistryAuthenticator
+ validate(in credentials: UsernamePasswordCredentials, in context: WebContext)
# throwsException(in message: String)
validate
throwsException

Figure 6.1: Service registry - UML Class Diagram

The service registry is implemented in Java and uses *jetcd*, which is the official Java library for etcd [Clo20c]. The key-value store etcd itself is running inside a LXC container on the service gateway in order to provide temporal and spatial partitioning for the etcd key-value store. The used version of etcd is 3.4.7 based on GO in version go1.14.2. Figure 6.1 shows a class diagram of the service registry and all involved components. As it can be seen, the service registry uses PAC4J [CAS20] for authentication and authorization. Further, Java Spark [Per20] in combination with Mustache [Chr20] as a template engine is used to provide the REST interface for the communication with the service registry, e.g in order to query the service database. Java Spark is a micro framework for creating small web applications and has an integrated small embedded webserver based on *Jetty*. For service discovery, the service registry uses DNS-SD provided by the Java library jmDNS.

In this thesis, the service registry was implemented with only a REST interface. However, the service registry uses the best-effort communication provided by the message-broker in order to publish changes in the service registry, like e.g. the registration of a new service.

6.2 Real-Time Communication

This section presents the implementation of the real-time communication presented in Section 5.4. The presented solution for real-time communication provides not only the fundamental basis for closed-control loops in distributed medical scenarios with strict timing requirements, but it is also an essential component of the services for temporal and spatial partitioning presented in Section 5.5. For this reason, the real-time communication must also take the communication between different containers into account. Furthermore, the presented real-time communication model must consider a hierarchical communication model (cf. Figure 6.2).

Since Linux is used as the basis of the presented architecture, the implementation of the real-time communication service relies on features and techniques provided by the Linux Kernel. One of the used Kernel features is Time-Aware Priority Shaper (TAPRIO), which is a queueing discipline (QDISC) introduced with Linux Kernel version 5.2 that implements (a simplified version of) the Time-Aware Shaper (TAS) (cf. Section 2.2.4) defined in the enhancements for scheduled traffic in IEEE 802.1Qbv. By this way, the TAPRIO queueing discipline can provide temporal isolation of the network resources and thus is able to isolate applications from each other regarding the network resources.

The TAPRIO queueing discipline can directly be applied to network interfaces. However, the network interface has to provide multiple sending and receiving queues in order to map the different traffic classes of TAPRIO. This can easily be checked by having a look at the queues directory of the network interface in the Linux */sys/class/net* folder. Listing 6.1 shows the directory structure for an Intel i210 network card, which has four dedicated queues per direction.

Listing 6.1: Sending/Receiving Queues of a network interface

```
[michael@lxc-dev ~]$ ls /sys/class/net/enp5s0/queues/
rx-0 rx-1 rx-2 rx-3 tx-0 tx-1 tx-2 tx-3
```

Provided that the network interface supports multiple queues, the TAPRIO queueing discipline can be assigned to the network interface using the command line tool *tc*. A programmatic assignment of the TAPRIO queueing discipline is also possible (e.g. in a

C++ program). Listing 6.2 shows an example *tc* command which assigns the TAPRIO queueing discipline with three different traffic classes (*num_tc 3*) to the network interface with the name *enp5s0*.

The current implementation of the TAPRIO queueing discipline uses the priority field of the Linux socket buffer structure (*sk_buf*) to determine the traffic class of the packet. The traffic class implementation of the TAPRIO queueing discipline allows to define up to 16 different traffic classes. It should be mentioned that this is different to the TAS defined in IEEE 802.1Qbv which defines a maximum limit of eight different traffic classes due to the usage of the three-bit Priority Code Point (PCP) in the VLAN tag for identifying the traffic class of the packet. Like other qdiscs, TAPRIO can be assigned to network devices by using the *tc* (traffic control) command which is part of the *iproute2* user space utilities for managing and monitoring network related issues in the Linux kernel. The *map* option of the queueing discipline command defines, how packets are mapped to these 16 different traffic classes. Therefore, the *map* option has to be defined as a list with 16 single mappings. The example in Listing 6.2 can be read as follows: map priority 2 to traffic class 1 (e.g. packets with deadline), map priority 3 to traffic class 0 (e.g. packets with fixed transmission time) and map all other priorities to traffic class 2 (e.g. best-effort).

Listing 6.2: Using tc to configure a network card with the TAPRIO queueing discipline

```
[michael@lxc−dev ~]$ tc qdisc replace dev enp5s0 parent root handle 100 taprio \
        num_tc 3 \
        map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2 \
        queues 1@0 1@1 2@2 \
        base−time 1590406880000000000 \
        sched−entry S 01 300000 \
        sched−entry S 02 300000 \
        sched−entry S 04 400000 \
        clockid CLOCK_TAI
```

The next parameter *queues* of the qdisc command defines, how TAPRIO maps the traffic classes to the sending queues of the network interface (cf. Listing 6.1). The *queues* parameter takes a list where each entry represents a traffic class. In the presented example, the list must contain three entries since we have three distinct traffic classes. The first entry represents traffic class 0, the second entry represents traffic class 1 and the third entry represents traffic class 2. The entries have to be defined with the *schema queue_count@queue_offset*. In the example, *1@0 1@1 2@2* would now map traffic class 0 to queue 0 (queue_offset 0), traffic class 1 to queue 1 (queue_offset 1) and traffic class 2 to the two queues 2 and 3 (beginning at queue_offset 2). This means that the traffic classes for deadline and fixed-time scheduled packets will get one transmission queue each while the best-effort traffic class will get two transmission queues of the network interface. The *base-time* parameter then defines the start time of the schedule that is defined by the parameter *sched-entry*. The *sched-entry* parameter in the example has three entries which define each a transmission timeslot for the respective traffic class. The entries of the *sched-entry* parameter have to be defined by the following format: sched-entry <command> <gatemask> <interval>. Currently, the TAPRIO queueing discipline only supports *"S"* as <command>, which represents the Time Sensitive Networking (TSN) command *SetGateStates* (cf. [IEE16a]) and opens the gate defined by <gatemask> for transmission. The parameter *<gatemask>* has to be provided as a bitmask, where each bit represents one traffic class. The parameter *<interval>* then defines how long the gate remains open and is a duration in nanoseconds. The period is implicitly defined by the sum of the intervals of all schedule entries. In the presented example, the period results in an total of 1ms.

Listing 6.3: Using tc to check assigned queueing discipline of a network interface

```
[lxc-dev ~]# tc qdisc show dev enp5s0
qdisc taprio 100: root refcnt 9 tc 3 map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2
queues offset 0 count 1 offset 1 count 1 offset 2 count 2
clockid TAI base-time 1593443961000000000 cycle-time 1000000 cycle-time-extension 0
        index 0 cmd S gatemask 0x1 interval 300000
        index 1 cmd S gatemask 0x2 interval 300000
        index 2 cmd S gatemask 0x4 interval 400000

qdisc pfifo 0: parent 100:4 limit 1000p
qdisc pfifo 0: parent 100:3 limit 1000p
qdisc pfifo 0: parent 100:2 limit 1000p
qdisc pfifo 0: parent 100:1 limit 1000p
```

The TAPRIO queueing discipline was developed to work with physical network interfaces like the Intel i210. However, having a look at the source code of the TAPRIO queueing discipline, it can be seen that TAPRIO only puts the requirement of the support for multiple sending and receiving queues on the underlying network interface. This means, that the TAPRIO queueing discipline can be used for any network interface as long as it supports multiple queues. This characteristic of the TAPRIO implementation is of interest for the presented architecture, since it allows to use TAPRIO with virtual network interfaces like *veth* or *tun* devices in Linux. Both virtual devices support multiple queues for sending and receiving packets. Precisely, the network device has to support at least as many queues as traffic classes are defined in the schedule of TAPRIO.

The first experiments with TAPRIO in combination with virtual network devices triggered a use-after-free bug resulting in a completely frozen operating system. Besides system crashes like the one mentioned, use-after-free bugs can also lead to serious security issues [BLCH19]. Here, my acknowledgments go to Vinicius Costa Gomes from Intel for fixing this bug. He is the current maintainer of the TAPRIO queueing discipline. As with the writing of this thesis, the bugfix was already integrated into the mainline Linux kernel with version 5.4 [LIN20g].

veth devices are of special interest in combination with Linux Containers (LXC) (cf. Section 5.9), which is used as the basis for temporal and spatial isolation. LXC can be configured to use *veth* devices to provide network connectivity to the containers. Although *veth* devices support multiple queues as already mentioned, LXC creates *veth* devices with only a single queue per direction. As part of the presented implementation, a standard LXC (version 4.0.2) was extended in such a way that during the start of a container, its virtual ethernet devices (veth) are created with multiple tx and rx queues (instead of single queue devices which is the original behavior). This is important since the TAPRIO queueing discipline requires an underlying multiqueue capable network device in order to map the traffic classes. *veth* devices represent a virtual connection, therefore *veth* devices are always created in pairs, as it can be seen in Figure 6.2. The first *veth* device is connected to the LXC container and the second *veth* is then connected to a virtual Linux network bridge (cf. [Ben06, Chapter 16]).

Since the created *veth* devices now provide multiple queues by using the already mentioned multiqueue patch, TAPRIO can be applied to the *veth* device connected to the virtual Linux network bridge. With the implementation in this thesis, a standard virtual Linux network bridge was used. However, it is also possible to use a bridge provided by *Open vSwitch* [LIN20f]. By the application of the TAPRIO queueing discipline, the presented approach establishes scheduled traffic among LXC containers. The virtual Linux network bridge is further connected to the physical network interface, which is in this case the Intel i210 of the service gateway. Having the TAPRIO queueing discipline also applied to the physical network interface then establishes scheduled traffic among service gateways. Combining both scenarios, the container-to-container and the host-to-host real-time communication results in a hierarchical real-time communication as Figure 6.2 shows.

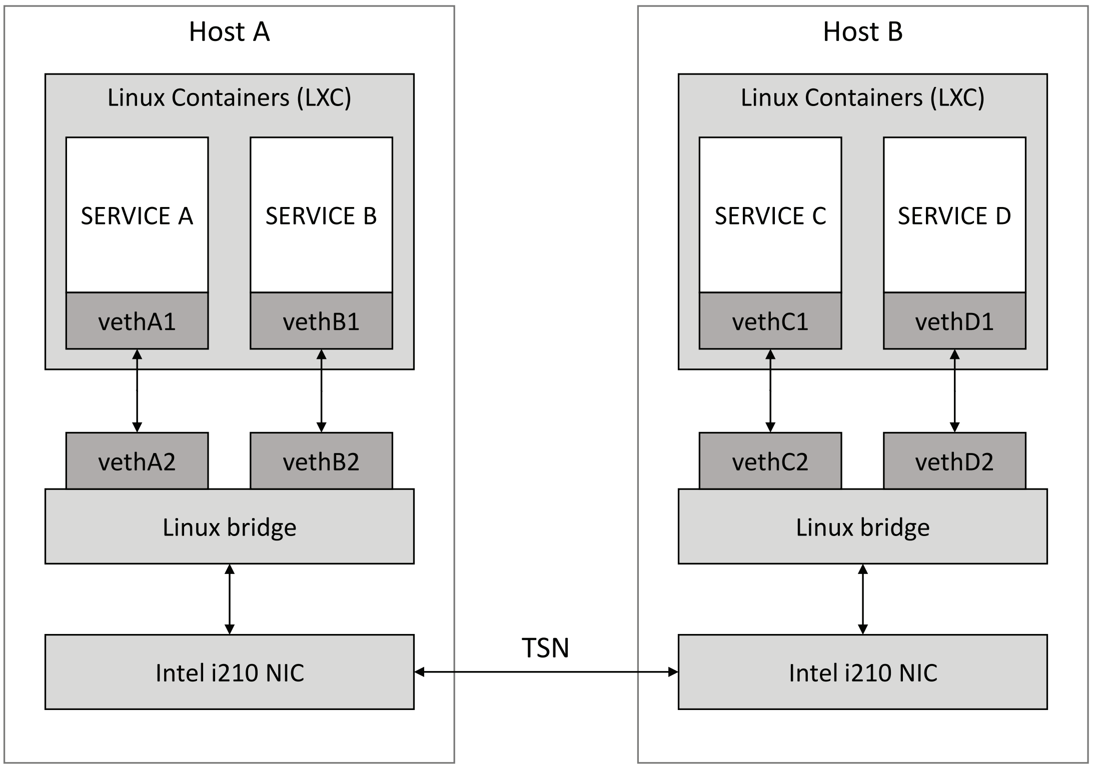


Figure 6.2: Hierarchical real-time container communication model

6.3 Fault-Tolerant Clock Synchronization

The fault-tolerant clock-synchronization service fully relies on *linuxptp* [Ric18], which is an implementation of Precision Time Protocol (PTP) for Linux maintained by Richard Cochran. *linuxptp* supports hardware and software time stamping and further support IEEE 802.1AS-2011 (in the role of end station), which is especially important to support TSN networks.

PTP requires compatible network interfaces. A prominent PTP compatible network card is the Intel i210, which supports a wide range of PTP protocol features such as hardware timestamping. The Intel i210 is explicitly supported by *linuxptp* and thus used within this thesis for the experiments. In Linux, the compatibility of network hardware can be checked with *ethtool*, which is a configuration and diagnostic tool for wired network cards. Listing 6.4 shows the output of *ethtool* for the Intel i210 network card.

Listing 6.4: Output of *ethtool* for the Intel i210 network card supporting PTP

```
[michael@lxc-dev ~]$ ethtool -T enp5s0
Time stamping parameters for enp5s0:
Capabilities:
        hardware−transmit (SOF_TIMESTAMPING_TX_HARDWARE)
        software−transmit (SOF_TIMESTAMPING_TX_SOFTWARE)
        hardware−receive (SOF_TIMESTAMPING_RX_HARDWARE)
        software−receive (SOF_TIMESTAMPING_RX_SOFTWARE)
        software−system−clock (SOF_TIMESTAMPING_SOFTWARE)
        hardware−raw−clock (SOF_TIMESTAMPING_RAW_HARDWARE)
PTP Hardware Clock: 0
Hardware Transmit Timestamp Modes:
        off (HWTSTAMP_TX_OFF)
        on (HWTSTAMP_TX_ON)
Hardware Receive Filter Modes:
        none (HWTSTAMP_FILTER_NONE)
        all (HWTSTAMP_FILTER_ALL)
```

Here, it can be seen that the Intel i210 has a PTP compatible hardware clock and the Media Access Control (MAC) supports hardware timestamping. The mentioned "PTP Hardware Clock: 0" in Listing 6.4 is misleading, since it does not mean that there is no hardware clock available. It means that the hardware clock with the ID 0 is selected as an associated clock for hardware timestamping.

Listing 6.5 shows the output of *ethtool* where the MAC of the network card only supports software time stamping.

Listing 6.5: Output of *ethtool* for network card that does not support PTP

```
[michael@lxc−dev ~]$ ethtool −T enp6s0
  Time stamping parameters for enp6s0:
  Capabilities:
        software−receive (SOF_TIMESTAMPING_RX_SOFTWARE)
        software−system−clock (SOF_TIMESTAMPING_SOFTWARE)
  PTP Hardware Clock: none
  Hardware Transmit Timestamp Modes: none
  Hardware Receive Filter Modes: none
```

The PTP implementation *linuxptp* consists mainly of two tools, which are *ptp4l* and *phc2sys*. While *ptp4l* implements PTP and is responsible for the synchronization of the real-time clocks of all network devices, *phc2sys* synchronizes the network clocks with the clock of the host. The clock-synchronization service is started at boot time of the service gateway using two systemd service files (ptp4l.service and phc2sys.service). The advantage of using systemd is its capability to automatically recover and restart failed services, including the clock-synchronization service. This ensures a fault-tolerant behavior of the clock-synchronization service.

6.4 Sensor Integration

This section presents the implementations that were created in the context of this thesis concerning the integration of sensors. This includes several techniques like sensor integration using ISO/IEEE 11703, Bluetooth Low Energy (Bluetooth LE), sensor data provided by cloud services and the possibility to store data in a database in order to be able to monitor sensor data over a certain period of time or for getting a single historical value.

6.4.1 ISO/IEEE 11703

As introduced in Section 5.8, the presented architecture supports the integration of devices that are compatible to ISO/IEEE 11703. Here, the sensor integration service relies on the open source implementation *Antidote* of the ISO/IEEE 11703 communication stack from Signove Tecnologia [Sig14]. For Bluetooth, *Antidote* uses the Linux Bluetooth stack implementation *bluez*. However, in the current version of the *Antidote* stack, *bluez* is only supported up to version 4.x. With the release of *bluez* in version 5 major changes in the Application Programming Interface (API) took place. In order to work with newer Linux Distributions, a patch was created as part of the implementation which modifies the bluetooth plugin of the *Antidote* stack to work with current version of the *bluez* Bluetooth stack in version 5 [Sch16].

Manager Application

As mentioned in Section 5.8, the main task of the implemented ISO/IEEE 11703 Manager is to publish data from ISO/IEEE 11703 compatible devices to message broker and to keep the service registry up-to-date regarding attached ISO/IEEE 11703 agents (e.g. notify in case of a new agent). This allows the access of data from ISO/IEEE 11703 compatible devices for all services running on the service gateway. Since *Antidote* is implemented in C, the ISO/IEEE 11703 Manager Application (cf. Figure 5.9) is also implemented in C. For the communication with the Message Broker of the best-effort communication service, the Manager Application uses the official library rabbitmq-c provided by RabbitMQ.

Transcoding Agent

ISO/IEEE 11703 compliant sensors can be directly connected to the service gateway by directly using the *Antidote* stack. *Antidote* currently allows the connection of agents by standard Bluetooth (via Health Device Protocol (HDP)) or TCP/IP. In addition, an extension of the *Antidote* stack for any other transport technology like ZigBee is feasible. Extending the stack for Bluetooth LE would be possible as well, but this would require the Bluetooth LE devices to be directly connected to the service gateway and the *Antidote* stack would be extended by that single transport technology.

In order to open ISO/IEEE 11703 for further communication technologies, this thesis follows a different approach by introducing an ISO/IEEE 11703 compliant transcoding agent (cf. [WG15]). The transcoding agent takes data from an incompatible sensor device and converts it into ISO/IEEE 11703 compliant data. Precisely, the agent transcodes received sensor data (JavaScript Object Notation (JSON)) to Application Protocol Data Units (APDU) packets and forwards them to *Antidote*. The mapping of the data structures is carried out as proposed in the Personal Health devices transcoding white paper [WG15] from the Bluetooth SIG. The white paper covers all mandatory attributes from the ISO/IEEE 11703 device specializations and explains how to map them to the Bluetooth LE device profiles. Figure 6.3 shows the data flow as proposed by the white paper.

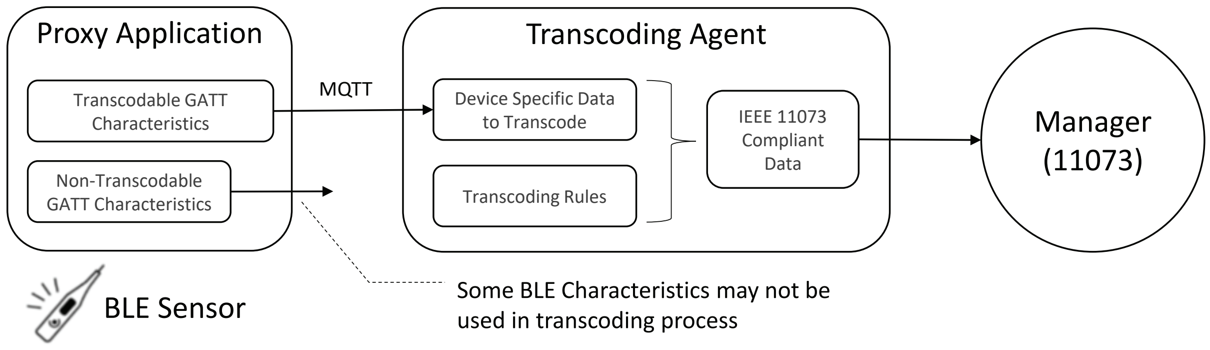
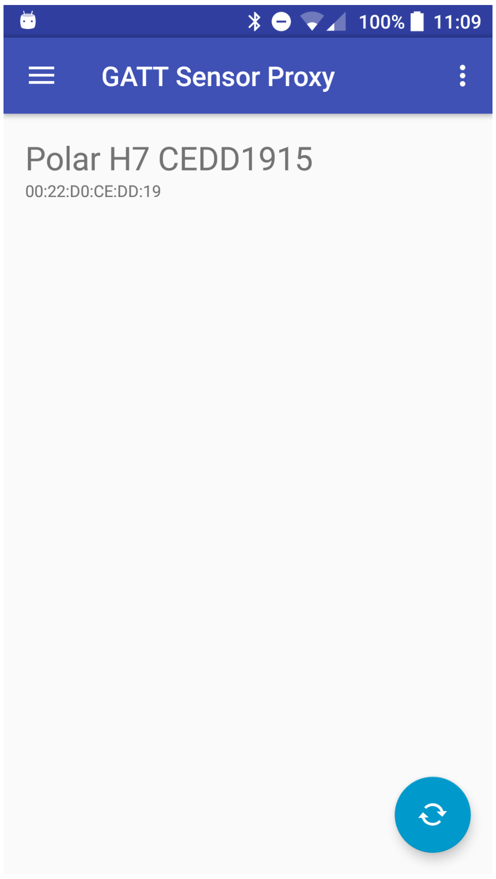

Proxy Application
Transcodable GATT Characteristics
Non-Transcodable GATT Characteristics
MQTT
Transcoding Agent
Device Specific Data to Transcode
Transcoding Rules
IEEE 11073 Compliant Data
Manager (11073)
BLE Sensor
Some BLE Characteristics may not be used in transcoding process

Figure 6.3: IEEE11073 Transcoding Agent [WG15]

The reason for that decision is the trend of connecting medical and health care devices using energy efficient communication protocols like Bluetooth LE, which allows more energy efficient devices. However, these devices are typically connected to a smartphone in order to configure them or to display and monitor the data of the devices. Therefore, the presented implementation uses a smartphone for the integration of these devices in the ISO/IEEE 11703 ecosystem. In the presented implementation a dedicated application on the smartphone was implemented, but the functionality for sending the data to the transcoding agent could also be easily integrated into any proprietary application. The data format used for sending data to the transcoding agent is formatted in JSON with a data structure as proposed by the Bluetooth SIG in the GATT REST API white paper [BLU14]. In short, the white paper defines, how GATT resources can be made accessible using standard HTTP methods. Similar to the presented approach, the white paper introduces a gateway device to which the Bluetooth LE devices are connected. The gateway device contains a RESTful web-service application that serves as an interface between the clients (Hypertext Transfer Protocol (HTTP) requests) and the Bluetooth LE devices (Generic Attribute Profile (GATT) profiles). The interface towards the clients is realized as a stateless Representational State Transfer (REST) service. For standardization, the concepts of the whitepaper were introduced as Internet Engineering Task Force (IETF) draft in 2016 [BK16].

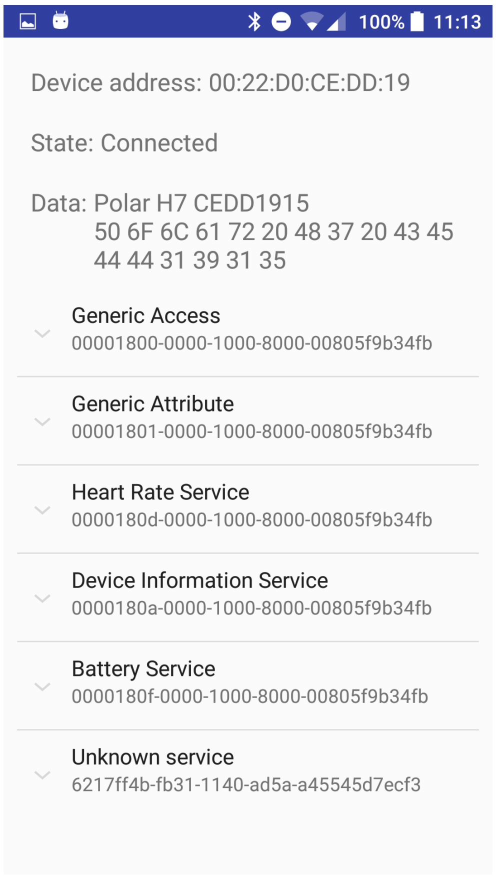
In comparison to the white paper, the presented implementation uses a small smartphone application which uses Message Queuing Telemetry Transport (MQTT) rather than providing a RESTful API over standard HTTP. MQTT was chosen since it has a lower footprint and implementation overhead in Android compared to Advanced Message Queuing Protocol (AMQP). In order to be able to receive MQTT message, the MQTT plugin of the message broker RabbitMQ was used. According to the state machine of ISO/IEEE 11703, the smartphone application announces every newly associated Bluetooth LE device via MQTT to the ISO/IEEE 11703 compliant transcoding agent on the service gateway. The transcoding agent then associates the sensor to the ISO/IEEE 11703 software stack *Antidote*. In a similar manner, the smartphone application announces the disconnection of a Bluetooth LE device. In this case, the agent would disassociate the sensor from the ISO/IEEE 11703 software stack. For the identification of the Bluetooth LE devices, their Media Access Control (MAC) addresses are used. Once a device is connected to the smartphone application and associated to the ISO/IEEE 11703 stack by the transcoding agent, the smartphone application allows to transmit the data of one or more GATT attributes to the transcoding agent. This can be triggered either manually or configured to be synchronized with every GATT measurement notification (e.g. when a particular characteristic changes on the Bluetooth LE device). The GATT attributes which shall be submitted can be configured. The Bluetooth LE devices remain associated to the ISO/IEEE 11703 software stack *Antidote* until the transcoding agent receives an disconnect message from the smartphone application. The application on the smartphone was implemented for Android. Figure 6.4 shows the two main activities of the smartphone application.

Both, the transcoding agent and the smartphone application were implemented and tested by integrating a Polar H7 heart rate sensor (cf. Figure 6.5). The Polar H7 heart



Polar H7 CEDD1915
00:22:D0:CE:DD:19

(a) Start Acivity



Device address: 00:22:D0:CE:DD:19

State: Connected

Data: Polar H7 CEDD1915
50 6F 6C 61 72 20 48 37 20 43 45
44 44 31 39 31 35

Generic Access
00001800-0000-1000-8000-00805f9b34fb

Generic Attribute
00001801-0000-1000-8000-00805f9b34fb

Heart Rate Service
0000180d-0000-1000-8000-00805f9b34fb

Device Information Service
0000180a-0000-1000-8000-00805f9b34fb

Battery Service
0000180f-0000-1000-8000-00805f9b34fb

Unknown service
6217ff4b-fb31-1140-ad5a-a45545d7ecf3

(b) GATT attributes of Bluetooth LE device

Figure 6.4: Implemented Android application

rate sensor was paired with an Android Smartphone running the presented smartphone application. The Polar H7 heart rate sensor provides the following GATT attributes:

- Device Information Service (org.bluetooth.service.device_information, 0x180A)
- Heart Rate Service (org.bluetooth.service.heart_rate, 0x180D)
- Battery Information Service (org.bluetooth.service.battery_service, 0x180F)

The smartphone application was configured to forward data to the service gateway. The smartphone application forwarded the heart rate measurement characteristic (UUID 0x2A37) of the Polar H7 to the dedicated topic exchange of the transcoding agent. Listing 6.6 shows an example MQTT data payload for the heart rate measurement characteristic of the Polar H7.

Listing 6.6: GATT characteristic in JSON representation

```
{
 "resource": "mqtt://smartphone:b4:9d:0b:5f:34:2c
```

```
    /gatt/nodes/00:22:d0:ce:dd:19/0x2a37/value",
 "timestamp": 1484251974,
 "handle": "0x2a37",
 "value": 68,
}
```

Figure 6.5: Polar H7 - Heart rate sensor

Based on the subscription of the transcoding agent, the agent receives the heart rate measurements from the Polar H7 heart rate sensor and transcodes the measurement data to an APDU according to the device specialization for a Basic ECG (heart rate) ISO/IEEE 11703-10406 device. After the transcoding process, the agent forwards the generated APDU to *Antidote*. The APDU then passes the *Antidote* stack and is forwarded to the already introduced manager application. The manager application then forwards the received data to a general sensor topic where the sensor data is forwarded to every subscriber that has a filter on that topic.

6.4.2 Cloud Sensor Integration

As mentioned above, modern healthcare devices are increasingly connected via low-power communication protocols such as Bluetooth LE. However, there is also a trend towards transmitting sensor data to cloud-based health platforms. This has several advantages for the users of these devices. An example would be summaries of activities performed on a daily or weekly basis. The biggest advantage, however, is the ability to use the health devices even when not being at home. Therefore, supporting these cloud platforms is also very important for an architecture for elderly care. For this reason, the presented implementation also includes an exemplary implementation for such a cloud platform, wich is the cloud platform of Withings.

Withings provides a public API in order to access the data of sensors in their cloud platform [Wit20]. The access to the API is secured by using OAuth2, which is an industry standard protocol [Har12] [JH12] that provides authorization flows for different kinds of applications like cloud platform APIs.

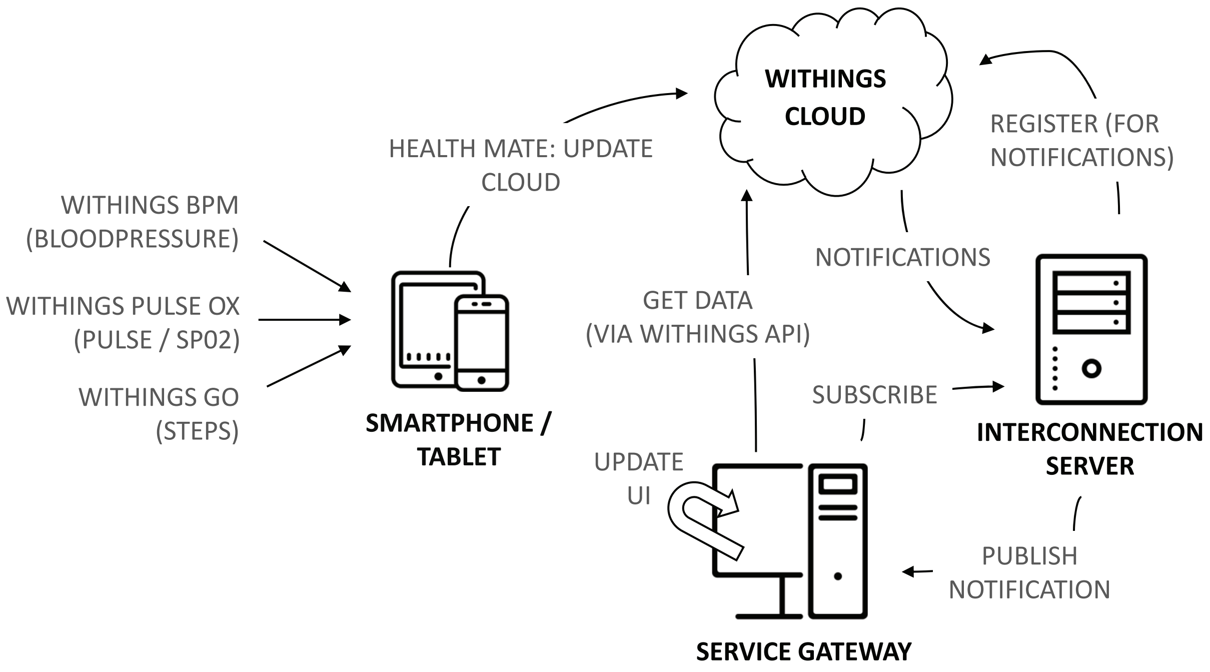

WITHINGS CLOUD
HEALTH MATE: UPDATE CLOUD
REGISTER (FOR NOTIFICATIONS)
WITHINGS BPM (BLOODPRESSURE)
WITHINGS PULSE OX (PULSE / SP02)
WITHINGS GO (STEPS)
SMARTPHONE / TABLET
NOTIFICATIONS
GET DATA (VIA WITHINGS API)
SUBSCRIBE
INTERCONNECTION SERVER
UPDATE UI
PUBLISH NOTIFICATION
SERVICE GATEWAY

Figure 6.6: Withings Cloud Sensor Integration

As Figure 6.6 shows, the Withings Sensors first trasmit their measurements to a smartphone or tablet that runs the Withings App *Health Mate*. In the context of the presented implementation, two sensors from Withings were used, namely the Withings Pulse Ox and the Withings BPM. The Withings Pulse Ox is capable of pulse oximetry, which is a noninvasive method for measuring the blood oxygen saturation (SpO2) of a person. Additionally it can measure the heart beat (pulse) of the person. The Withings BPM is a blood pressure monitor. *Health Mate* transmits the received data afterwards in predefined intervals to the Withings cloud platform. In order to get notified about new data available, the Withings cloud platform supports the registration of a Uniform Resource Locator (URL) as callback.

For the presented implementation, the callback URL is provided by a small server component which just takes the data submitted with the callback URL and publishes it to subscribed Consumers, which in the presented case is a small sensor service for Withings sensor devices running on the service gateway. The Withings sensor service then starts a request to the Withings Cloud in order to obtain the new data. The Withings sensor service as well as the small server component were implemented in Java. Figure 6.8 shows a class diagram of the Withings sensor service.

6.4.3 Historical Sensor data

Besides the integration of sensors, it is also valuable to have access to historical sensor data. This is of especial interest for generating daily or weekly reports about the activities of elderly people. This helps elderly people to reflect their activities or to keep track of their health status. Due to these reasons, the presented implementation contains a service for the storage and access of historical data. In addition, the historical sensor data service provides an easy and comfortable method to get the latest data of a sensor stored in the database. This is for example helpful if a sensor only provides new data every hour or every day. Thus, no data is being transmitted between the intervals and e.g. a visualization framework would not have data for visualization.

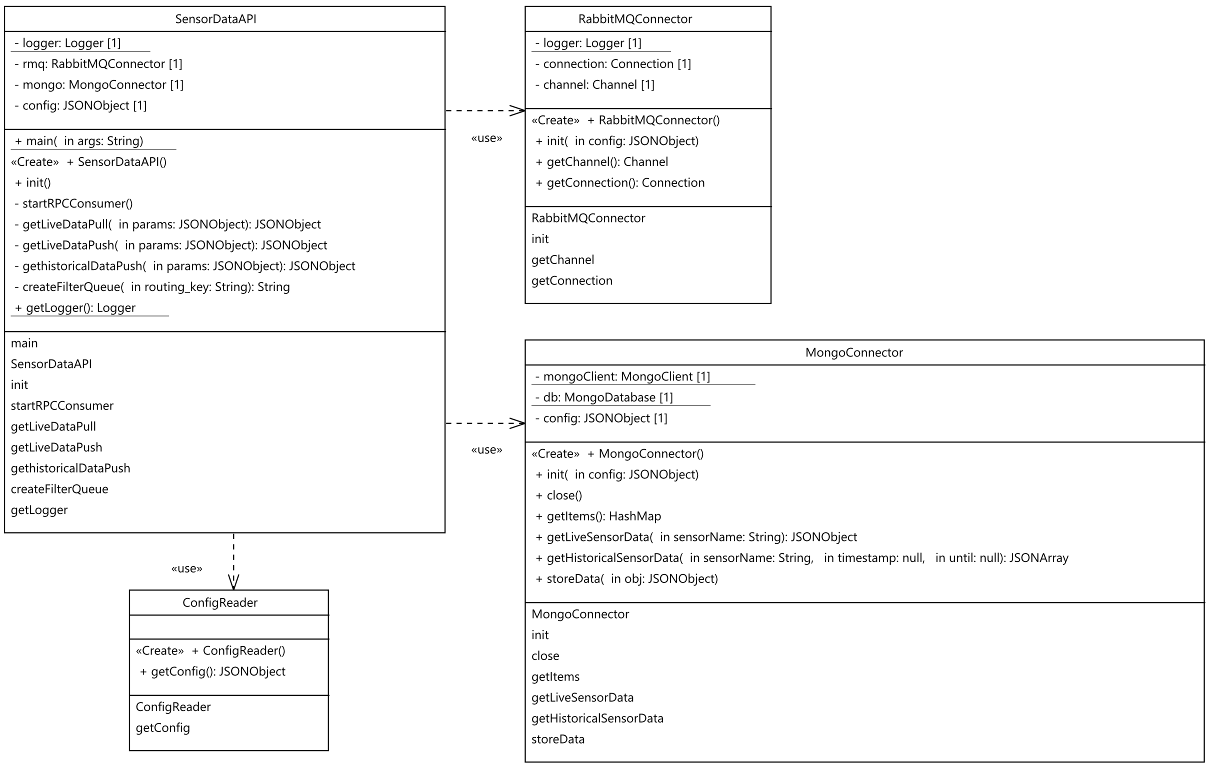

SensorDataAPI
- logger: Logger [1]
- rmq: RabbitMQConnector [1]
- mongo: MongoConnector [1]
- config: JSONObject [1]
+ main(in args: String)
«Create» + SensorDataAPI()
+ init()
- startRPCConsumer()
- getLiveDataPull(in params: JSONObject): JSONObject
- getLiveDataPush(in params: JSONObject): JSONObject
- gethistoricalDataPush(in params: JSONObject): JSONObject
- createFilterQueue(in routing_key: String): String
+ getLogger(): Logger
main
SensorDataAPI
init
startRPCConsumer
getLiveDataPull
getLiveDataPush
gethistoricalDataPush
createFilterQueue
getLogger
«use»
RabbitMQConnector
- logger: Logger [1]
- connection: Connection [1]
- channel: Channel [1]
«Create» + RabbitMQConnector()
+ init(in config: JSONObject)
+ getChannel(): Channel
+ getConnection(): Connection
RabbitMQConnector
init
getChannel
getConnection
«use»
MongoConnector
- mongoClient: MongoClient [1]
- db: MongoDatabase [1]
- config: JSONObject [1]
«Create» + MongoConnector()
+ init(in config: JSONObject)
+ close()
+ getItems(): HashMap
+ getLiveSensorData(in sensorName: String): JSONObject
+ getHistoricalSensorData(in sensorName: String, in timestamp: null, in until: null): JSONArray
+ storeData(in obj: JSONObject)
MongoConnector
init
close
getItems
getLiveSensorData
getHistoricalSensorData
storeData
«use»
ConfigReader
«Create» + ConfigReader()
+ getConfig(): JSONObject
ConfigReader
getConfig

Figure 6.7: Class Diagram of Historical Sensor Data Service

Within the presented implementation, the data of the sensors is stored in the document-oriented database MongoDB. MongoDB uses a JSON based syntax for storing and access of data. Listing 6.7 shows an example of the implemented JSON data structure.

Listing 6.7: JSON data structure example for a temperature sensor

```
{
  "info": {
    "description": "This sensor measures the ambient temperature and the humidity in the kitchen",
    "sensor_name": "Kitchen Temperature and Humidity",
    "value_info": {
      "1": {
        "description": "Timestamp of data sample",
        "name": "timestamp",
        "unit": "unix-timestamp in seconds"
      },
      "2": {
        "description": "Temperature in the kitchen",
        "name": "temperature",
        "unit": "celsius"
      },
      "3": {
        "description": "Humidity in the kitchen",
        "name": "relative-humidity",
        "unit": "percentage"
      }
    }
  },
  "sensor_id": "service-gateway.78ffb17c-6c65-11ea-bc55-0242ac130003.sensors. /
                kitchen.temperature.550e8400",
  "timestamp_hour": "2018-03-14T14:00:00.000Z",
  "user": "user",
  "values": [
    {
      "timestamp": 1521033729280,
      "value": [
        1521033720592,
        24,
        56
      ]
    }
  ]
}
```

The historical sensor data service provides two methods for the data access, namely PUSH and PULL. Both methods are implemented by using the Remote Procedure Call (RPC) technique of AMQP. The PULL method can be used by a service that just wants to get the latest entry from the database. The PULL method does not provide any functionality to get more than this last entry. This is because the service is blocked until the RPC returns the data. As mentioned in Section 2.7.3, this should be avoided since it brings dependencies among services. Thus, the PULL method only queries the last entry in the database, and the blocking time is kept as short as possible. In contrast, the PUSH method allows to access all of the historical data in the databases. In order to keep the blocking time in calling services as short as possible, the historical sensor data service first creates a queue in RabbitMQ and then creates a thread to query the data in the database. The created thread afterwards sends the result of the query via the previously created queue. In addition to that, the PUSH method optionally creates another queue that has a topic filter on the provided sensor-id. The calling service can then use this queue to get all future data pushed by the sensor. Figure 6.7 show a class diagram of the implemented historical sensor data service.

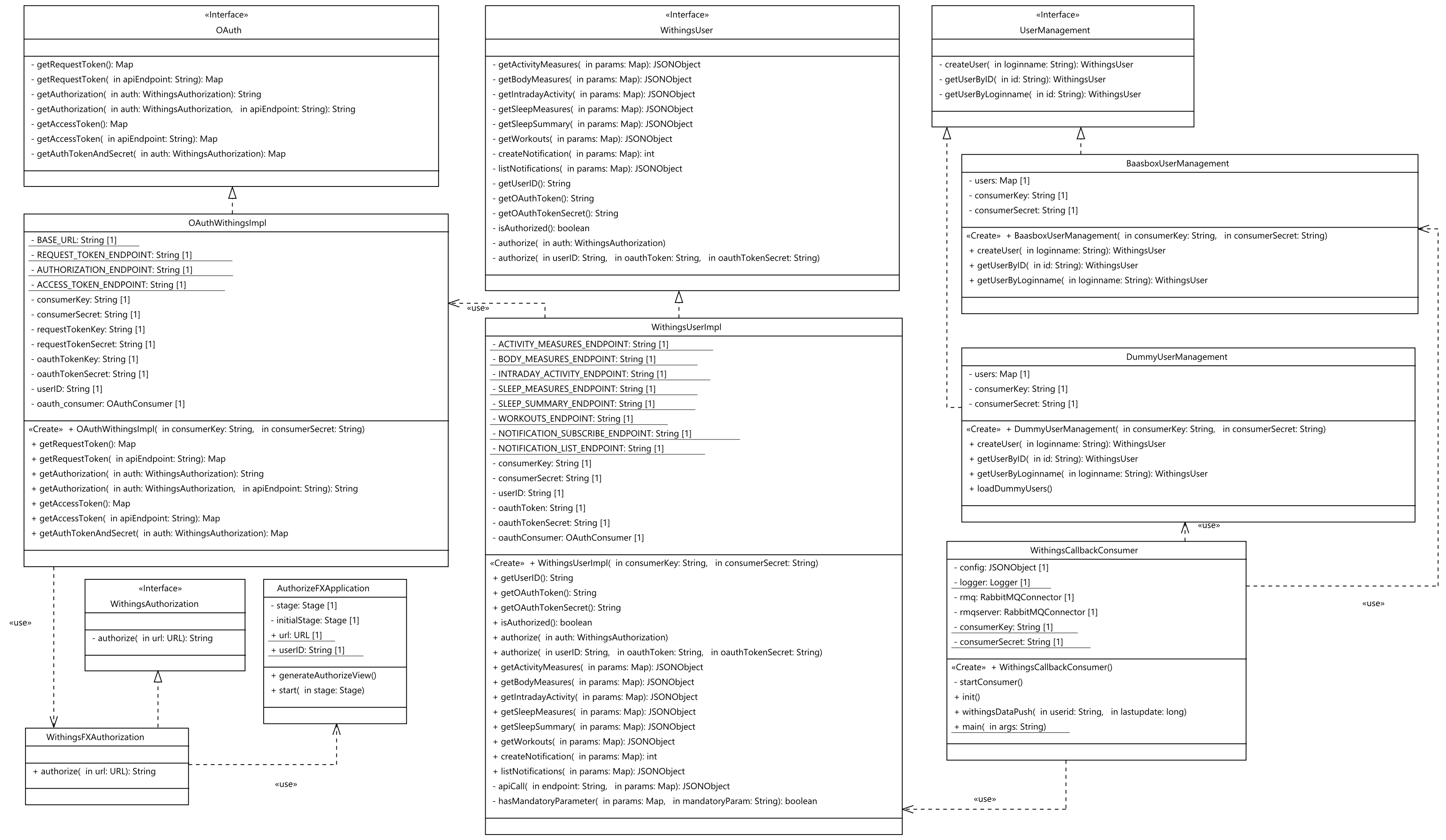

«Interface»
OAuth
- getRequestToken(): Map
- getRequestToken(in apiEndpoint: String): Map
- getAuthorization(in auth: WithingsAuthorization): String
- getAuthorization(in auth: WithingsAuthorization, in apiEndpoint: String): String
- getAccessToken(): Map
- getAccessToken(in apiEndpoint: String): Map
- getAuthTokenAndSecret(in auth: WithingsAuthorization): Map
«Interface»
WithingsUser
- getActivityMeasures(in params: Map): JSONObject
- getBodyMeasures(in params: Map): JSONObject
- getIntradayActivity(in params: Map): JSONObject
- getSleepMeasures(in params: Map): JSONObject
- getSleepSummary(in params: Map): JSONObject
- getWorkouts(in params: Map): JSONObject
- createNotification(in params: Map): int
- listNotifications(in params: Map): JSONObject
- getUserID(): String
- getOAuthToken(): String
- getOAuthTokenSecret(): String
- isAuthorized(): boolean
- authorize(in auth: WithingsAuthorization)
- authorize(in userID: String, in oauthToken: String, in oauthTokenSecret: String)
«Interface»
UserManagement
- createUser(in loginname: String): WithingsUser
- getUserByID(in id: String): WithingsUser
- getUserByLoginname(in id: String): WithingsUser
BaasboxUserManagement
- users: Map [1]
- consumerKey: String [1]
- consumerSecret: String [1]
«Create» + BaasboxUserManagement(in consumerKey: String, in consumerSecret: String)
+ createUser(in loginname: String): WithingsUser
+ getUserByID(in id: String): WithingsUser
+ getUserByLoginname(in loginname: String): WithingsUser
OAuthWithingsImpl
- BASE_URL: String [1]
- REQUEST_TOKEN_ENDPOINT: String [1]
- AUTHORIZATION_ENDPOINT: String [1]
- ACCESS_TOKEN_ENDPOINT: String [1]
- consumerKey: String [1]
- consumerSecret: String [1]
- requestTokenKey: String [1]
- requestTokenSecret: String [1]
- oauthTokenKey: String [1]
- oauthTokenSecret: String [1]
- userID: String [1]
- oauth_consumer: OAuthConsumer [1]
«Create» + OAuthWithingsImpl(in consumerKey: String, in consumerSecret: String)
+ getRequestToken(): Map
+ getRequestToken(in apiEndpoint: String): Map
+ getAuthorization(in auth: WithingsAuthorization): String
+ getAuthorization(in auth: WithingsAuthorization, in apiEndpoint: String): String
+ getAccessToken(): Map
+ getAccessToken(in apiEndpoint: String): Map
+ getAuthTokenAndSecret(in auth: WithingsAuthorization): Map
«use»
WithingsUserImpl
- ACTIVITY_MEASURES_ENDPOINT: String [1]
- BODY_MEASURES_ENDPOINT: String [1]
- INTRADAY_ACTIVITY_ENDPOINT: String [1]
- SLEEP_MEASURES_ENDPOINT: String [1]
- SLEEP_SUMMARY_ENDPOINT: String [1]
- WORKOUTS_ENDPOINT: String [1]
- NOTIFICATION_SUBSCRIBE_ENDPOINT: String [1]
- NOTIFICATION_LIST_ENDPOINT: String [1]
- consumerKey: String [1]
- consumerSecret: String [1]
- userID: String [1]
- oauthToken: String [1]
- oauthTokenSecret: String [1]
- oauthConsumer: OAuthConsumer [1]
«Create» + WithingsUserImpl(in consumerKey: String, in consumerSecret: String)
+ getUserID(): String
+ getOAuthToken(): String
+ getOAuthTokenSecret(): String
+ isAuthorized(): boolean
+ authorize(in auth: WithingsAuthorization)
+ authorize(in userID: String, in oauthToken: String, in oauthTokenSecret: String)
+ getActivityMeasures(in params: Map): JSONObject
+ getBodyMeasures(in params: Map): JSONObject
+ getIntradayActivity(in params: Map): JSONObject
+ getSleepMeasures(in params: Map): JSONObject
+ getSleepSummary(in params: Map): JSONObject
+ getWorkouts(in params: Map): JSONObject
+ createNotification(in params: Map): int
+ listNotifications(in params: Map): JSONObject
- apiCall(in endpoint: String, in params: Map): JSONObject
- hasMandatoryParameter(in params: Map, in mandatoryParam: String): boolean
DummyUserManagement
- users: Map [1]
- consumerKey: String [1]
- consumerSecret: String [1]
«Create» + DummyUserManagement(in consumerKey: String, in consumerSecret: String)
+ createUser(in loginname: String): WithingsUser
+ getUserByID(in id: String): WithingsUser
+ getUserByLoginname(in loginname: String): WithingsUser
+ loadDummyUsers()
«use»
WithingsCallbackConsumer
- config: JSONObject [1]
- logger: Logger [1]
- rmq: RabbitMQConnector [1]
- rmqserver: RabbitMQConnector [1]
- consumerKey: String [1]
- consumerSecret: String [1]
«Create» + WithingsCallbackConsumer()
- startConsumer()
+ init()
+ withingsDataPush(in userid: String, in lastupdate: long)
+ main(in args: String)
«use»
«use»
«use»
«Interface»
WithingsAuthorization
- authorize(in url: URL): String
AuthorizeFXApplication
- stage: Stage [1]
- initialStage: Stage [1]
+ url: URL [1]
+ userID: String [1]
+ generateAuthorizeView()
+ start(in stage: Stage)
WithingsFXAuthorization
+ authorize(in url: URL): String
«use»

Figure 6.8: Class Diagram of Withings Sensor Service

6.5 Dynamic Reconfiguration of Real-Time Communication Resources

Within the context of this thesis, the service for the dynamic reconfiguration of real-time communication resources was implemented exemplarily based on TTEThernet as underlying real-time communication protocol. However, the concepts and the models introduced within this thesis can be adopted to other technologies, like for example Time Sensitive Networking (TSN) [IEE16b].

The implementation of the Real-Time Broker (RT-Broker) and the Real-Time Client (RT-Client) were both implemented in Java according to the class diagrams presented in Figure 5.13. An overview about the classes for the specific TTEthernet implementation is provided by Figure 6.9. For the communication between the RT-Clients and the RT-Broker, the best-effort communication service based on RabbitMQ was used.

6.5.1 TTEthernet-Toolchain Integration

Due to TTEthernet as the underlying communication protocol, we integrated the TTEthernet Toolchain [TTT17a] shown in Figure 6.10 into the RT-Broker implementation. The assignment of the RT-Broker components to the specific TTEthernet tools is depicted in Figure 5.12. The TTEthernet-Toolchain also uses a network description (see also Section 2.2.3) to define the topology of the network. The TTEthernet network description is not generated automatically and therefore has to be provided by the maintainer of the system. Further, the network description contains information about the logical connections, which are called Virtual Links (VLs). According to the traffic classes introduced in Section 2.2.3, there are VLs for Time-Triggered (TT), Rate-Constrained (RC) and Best-Effort (BE) messages available. The endpoints of the VLs are called Data Ports. As VLs are unidirectional, a VL always has one sending Data Port and one or multiple receiving Data Ports. The definition of a VL in the network description also includes the properties of the messages, like maximum message size and period in which the data frames are transmitted. Further, the network definition defines the network periods, that are required for the establishment of the cluster cycles and the VLs. For TT messages, the TTEthernet network description additionally defines a synchronization domain that further specifies the behavior of the clock synchronization within the network. By implementing a parser for the TTEthernet network description, the RT-Broker can use an existing network description from the TTEthernet-Toolchain as a starting point for the network topology management.

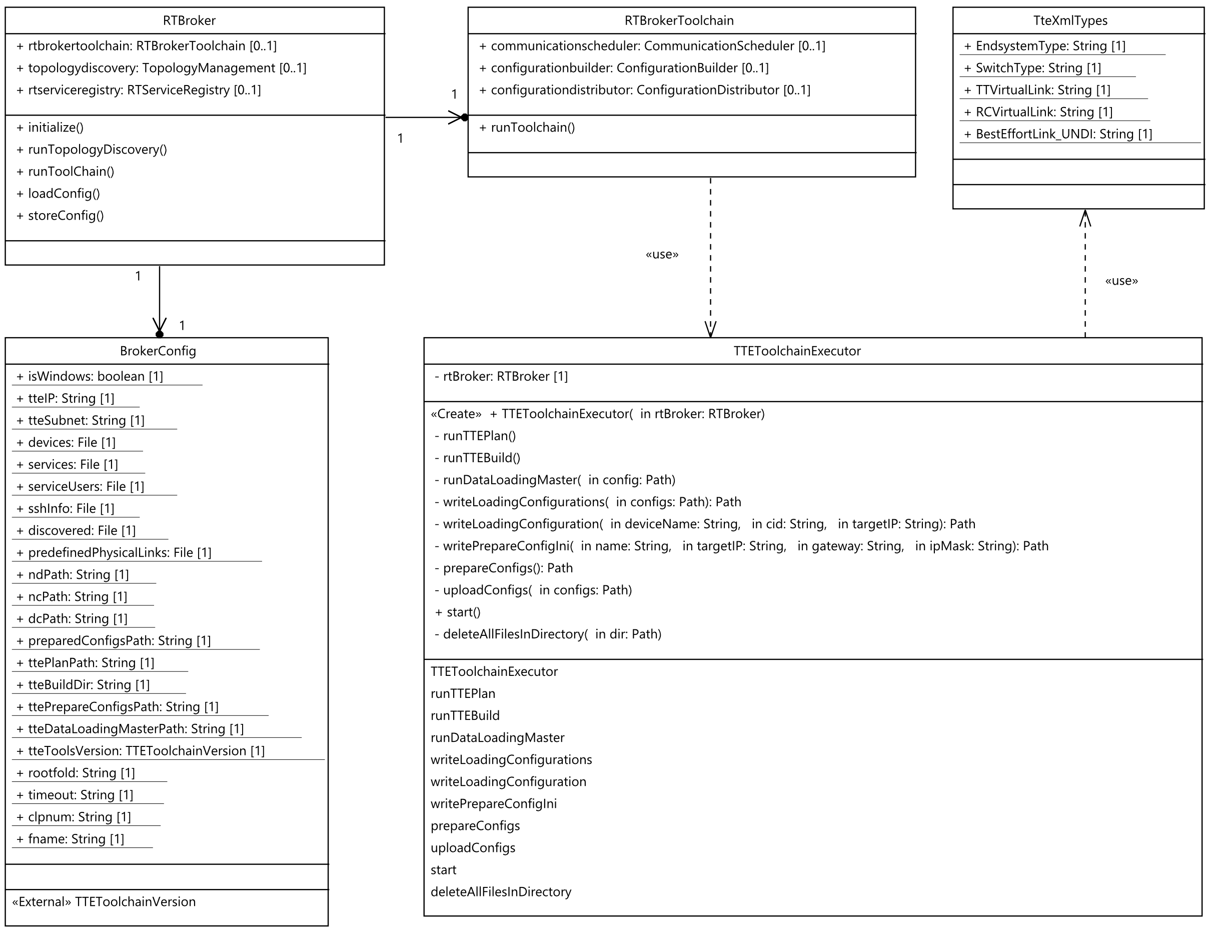

RTBroker
+ rtbrokertoolchain: RTBrokerToolchain [0..1]
+ topologydiscovery: TopologyManagement [0..1]
+ rtserviceregistry: RTServiceRegistry [0..1]
+ initialize()
+ runTopologyDiscovery()
+ runToolChain()
+ loadConfig()
+ storeConfig()
1
1
RTBrokerToolchain
+ communicationscheduler: CommunicationScheduler [0..1]
+ configurationbuilder: ConfigurationBuilder [0..1]
+ configurationdistributor: ConfigurationDistributor [0..1]
+ runToolchain()
TteXmlTypes
+ EndsystemType: String [1]
+ SwitchType: String [1]
+ TTVirtualLink: String [1]
+ RCVirtualLink: String [1]
+ BestEffortLink_UNDI: String [1]
«use»
«use»
1
1
BrokerConfig
+ isWindows: boolean [1]
+ tteIP: String [1]
+ tteSubnet: String [1]
+ devices: File [1]
+ services: File [1]
+ serviceUsers: File [1]
+ sshInfo: File [1]
+ discovered: File [1]
+ predefinedPhysicalLinks: File [1]
+ ndPath: String [1]
+ ncPath: String [1]
+ dcPath: String [1]
+ preparedConfigsPath: String [1]
+ ttePlanPath: String [1]
+ tteBuildDir: String [1]
+ ttePrepareConfigsPath: String [1]
+ tteDataLoadingMasterPath: String [1]
+ tteToolsVersion: TTEToolchainVersion [1]
+ rootfold: String [1]
+ timeout: String [1]
+ clpnum: String [1]
+ fname: String [1]
«External» TTEToolchainVersion
TTEToolchainExecutor
- rtBroker: RTBroker [1]
«Create» + TTEToolchainExecutor(in rtBroker: RTBroker)
- runTTEPlan()
- runTTEBuild()
- runDataLoadingMaster(in config: Path)
- writeLoadingConfigurations(in configs: Path): Path
- writeLoadingConfiguration(in deviceName: String, in cid: String, in targetIP: String): Path
- writePrepareConfigIni(in name: String, in targetIP: String, in gateway: String, in ipMask: String): Path
- prepareConfigs(): Path
- uploadConfigs(in configs: Path)
+ start()
- deleteAllFilesInDirectory(in dir: Path)
TTEToolchainExecutor
runTTEPlan
runTTEBuild
runDataLoadingMaster
writeLoadingConfigurations
writeLoadingConfiguration
writePrepareConfigIni
prepareConfigs
uploadConfigs
start
deleteAllFilesInDirectory

Figure 6.9: RTBroker for TTEthernet with TTEthernet-Toolchain integration - class diagram

Listing 6.8 shows an example output of the tool with one TTEthernet switch in the network.

Listing 6.8: Switch Discovery with the TTEthernet Toolchain

```
tt_615a3_console −r −l 10.10.10.20 −m 255.255.255.0
        Ver: 1.0.1.TT_615A3_VERSION_BUILD_NUMBER
        Ver: 1.0.1.TT_615A3_VERSION_BUILD_NUMBER
        Ver: 1.0.1.TT_615A3_VERSION_BUILD_NUMBER
E000: <INIT> operation finished successfully.
        Local IP Address found: 10.10.10.20
Broadcast: host −> 10.10.10.10!!!
E000: <FIND> operation finished successfully.
Find operation list of clients:

        Number of hosts found: 1
        Client ID: TTT—SWITCH—LAB_GROUND
IP address: 10.10.10.10
Target Hardware Identifier: TTT—SWITCH—LAB
Target Hardware Type Name: TTT24LAB
Target Hardware Position: GROUND
Literal Name: TTT—SWITCH—LAB—24P
Manufacturer Code: TTT
Opening listener ...
Success !!!
Closing listening thread ...
Exiting tt_615a3_tftp_open_listener
E000: <DISCOVER INFORMATION> operation finished successfully.
Discover information operation result:
——————————————— BEGIN —————————————————
Client ID: TTT—SWITCH—LAB_GROUND
Literal Name: TTT—SWITCH—LAB
Serial Number: 94702—00162
Part Number: TTT5401208803
Amendment: TTT—SWITCH—LAB 2.0.0 — ES: 1.6.38 (0x80011b74) /
 SWE: Unknown Switch IP (0x00015432)
Part Designation Text: LAB USE ONLY
———————————————— END ——————————————————
```

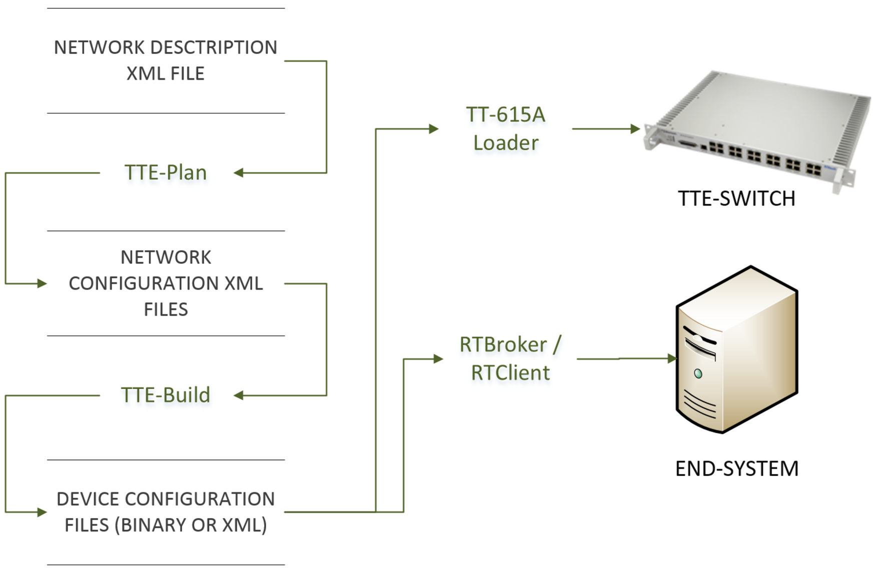

NETWORK DESCTRIPTION XML FILE
TTE-Plan
NETWORK CONFIGURATION XML FILES
TTE-Build
DEVICE CONFIGURATION FILES (BINARY OR XML)
TT-615A Loader
TTE-SWITCH
RTBroker / RTClient
END-SYSTEM

Figure 6.10: TTEthernet-Toolchain

If a new communication schedule has to be calculated (e.g. due to new network nodes or service usages), the broker uses the scheduler of the TTEthernet-Toolchain, which is included in the tool TTE-Plan for TTEthernets (TTE-Plans). The outputs of this tool are configuration files for each TTEthernet device in the network. Since our experimental setup is based on TTEthernet, a further step is necessary in order to load the configurations on the devices. The configuration files have to be transformed with the tool TTE-Build for TTEthernets (TTE-Builds) into binaries that can be uploaded to the devices. The TTEthernet switches (TTE Switch A664 Lab) used for the experimental setup support the ARINC 615A-3s (ARINC615As) standard. Therefore the RT-Broker uses the TT-615A-Loaders (TT-615A-Loaders) of the TTEthernet-Toolchain in order to load the configuration on the switch. The situation for the end systems (TTE End System A664) is different, because these devices do not the support ARINC615As for data loading. As introduced in the model, the RT-Broker uses in this case the data loading feature of the RT-Client. During the execution of the data loading protocol, the RT-Broker transfers the device configuration files using Secure Copy (SCP). Second, the RT-Client distributes the configuration files to the corresponding services. Since the IDs of the VLs are assigned dynamically during the scheduling with the tool TTE-Plans, the services cannot use directly the IDs of the VLs as virtual communication links. For this purpose, the RT-Client implements a link manager that must be used by the services in order to find their corresponding VL. The required information in the link manager is also updated by the RT-Broker during the execution of the data loading protocol. Finally, after the configuration files are distributed, the RT-Broker and the RT-Client run an agreement protocol to negotiate the restarting phase of the network. Table 6.1 shows a reconfiguration matrix that contains information about the cases when a reconfiguration of the real-time communication services is necessary.

Action	**Reconfiguration necessary**
New end system	-
New switch	X
New service	X (If a pending request for this service is found)
Removal of an end system	X
Removal of a switch	X
Service revocation	X

Table 6.1: Dynamic reconfiguration of real-time resources - Reconfiguration matrix

7 Evaluation and Results

While the last two chapters presented the models and implementations of the services, the main objective of this chapter is to evaluate the results of this thesis, which includes an evaluation of the implementations. The remainder of this chapter is structured as follows. First the evaluation objectives are presented. In a second step, various experiments and their results are presented to provide a benchmark for the performance of the implementations established in this work. Finally, the results are discussed with regard to the key challenges presented in Chapter 3. Some parts of this chapter were published in [SO18][SOW18].

7.1 Evaluation Objectives

As introduced in Chapter 3, a fault-tolerant real-time architecture for elderly care has to cope with several key challenges. A major challenge is to provide support for applications with different levels of criticality. In this respect, an architecture must be able to provide *techniques for fault tolerance and the isolation of resources.* In addition, medical applications with tight control loops require a deterministic real-time behavior of the system, which poses in particular challenges for the *provisioning of computational and communication resources.* Further challenges are the heterogeneity of the underlying technologies and the different integration levels at which the architecture is applied. Regarding sensors, this requires the support of *standards for medical and health device communication.* Likewise, it should be possible to add or remove sensors at run-time of the system. If sensors are used in applications with real-time requirements, this requires techniques for *dynamic reconfiguration of the real-time resources.*

The main objective is now to assess the results presented in relation to the key challenges mentioned above. The evaluation is based on several experiments, ranging from a real scenario for the integration of sensors to experiments in a laboratory. However, the experiments carried out concentrate on the open research questions of this thesis. From this perspective, only technologies and concepts that extend the state of the art are evaluated. These experiments and their results are presented in the following.

7.2 Experiments

7.2.1 Dynamic Reconfiguration

As introduced in Section 3.3, many use cases that apply for architectures in the field of elderly care require the integration of heterogeneous sensors like wearables or stationary health sensors. Further, this integration has often to be conducted at run-time demanding a dynamic reconfiguration of the system. This applies particularly to the composition of sensors and services. The open-world assumption requires techniques for dynamic-reconfiguration. This is especially true for real-time resources like communication resources in the real-time network. Since the focus of this work is on the support for real-time applications, this thesis includes a dynamic-reconfiguration service for the underlying real-time communication network (cf. Section 5.9 and Section 6.5).

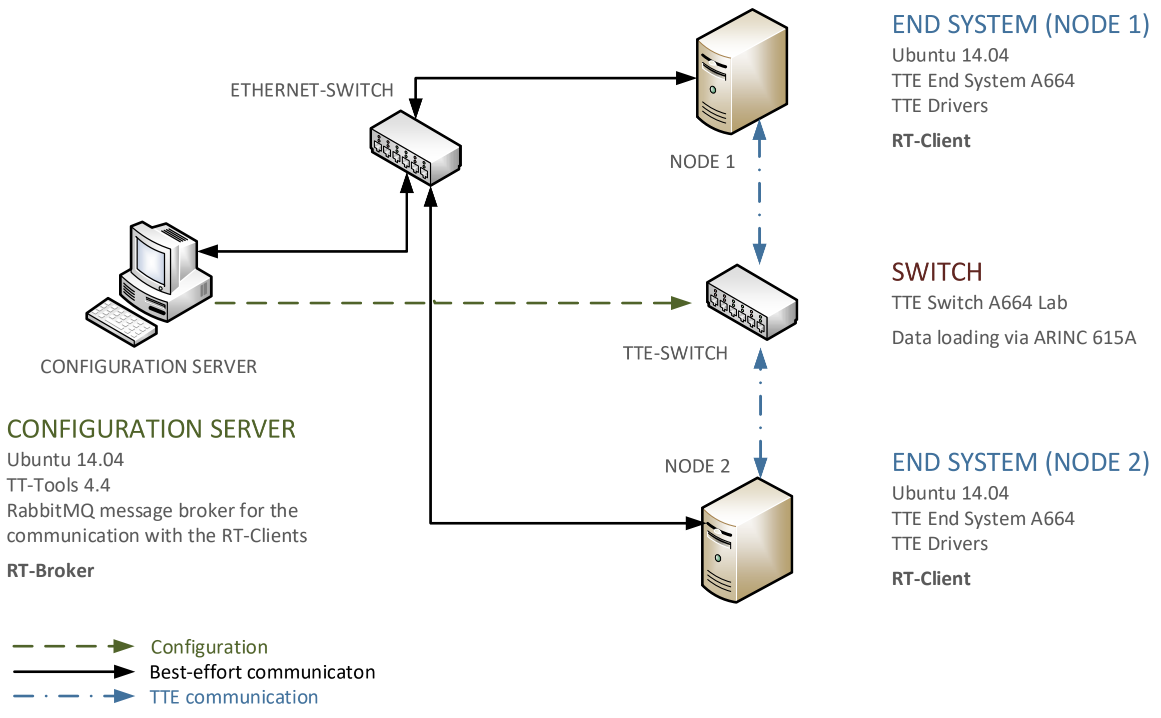


Figure 7.1: Dynamic reconfiguration - Experimental setup

Figure 7.1 shows the experimental setup used for the evaluation of the dynamic reconfiguration service. As depicted, the real-time communication network consists of two end systems equipped with TTEthernet network cards (TTE End System A664 [TTT17b]) and one TTEthernet switch (TTE Switch A664 Lab [TTT17c]). These end systems are running each a RT-Client and are supposed to offer services or to use services. Additionally, our setup comprises a management network based on Ethernet which is required for the monitoring of the end system during the reconfiguration phases of the real time network. This management network is also used in the presented experimental setup for the communication between the RT-Broker and the RT-Clients since this management network traffic does not require any temporal guarantees. The RT-Broker is running on a third end system, the configuration server. As it can be seen in Figure 7.1, the configuration server is not part of the real-time communication network and thus not connected to the TTEthernet switch.

In the presented experimental setup, the rescheduling and the distribution of the device configuration files takes about 80 to 90 seconds in total (end system with Intel Core i5-5200U and 4GB memory). During this time, the real time network is still fully operational. A downtime will first occur when the end systems and the switches adopt the new configurations. Most time is required by the discovery (about five seconds) and data loading (about 50-55 seconds) process of the switches using the ARINC 615A-3 standard. Further, the generation of the network description with TTE-Plans typically needs five to six seconds and the generation of the device configuration files with TTE-Builds takes about 20 seconds. Finally, the configuration files are distributed in about five seconds to the end systems.

As it can be seen, the measured durations for scheduling and data loading of the switches are quite long. Since the current configuration of the network is still working while the new configuration is calculated and loaded into the switches and end systems, the measured durations are acceptable. Furthermore, the new configuration is validated by the switches and the end systems prior to its activation. In case of a failed validation and in case of a fault occurring during the switch of the configuration, a safe roll back to the old configuration is conducted (cf. Figure 5.14).

7.2.2 Sensor Integration

Most use-cases for applications in elderly care require the data of sensors. Therefore, a seamless and effortless integration of sensors in an architecture for elderly care is of utmost importance. This includes complex medicals sensors like a pulse-oximeter as well as simple sensors like a clinical thermometer. However, the biggest challenge is the heterogeneity of the sensors and the underlying technologies used. This includes communication protocols and data formats. Section 6.4 already presented the large variety of methods for sensor integration supported by the presented architecture. A huge contribution for the sensor integration is the integration of the ISO/IEE 11073 software stack *Antidote* with several additionally implemented modules in order to extend the functionality of ISO/IEE 11073. For example, the transcoding agent in combination with the smartphone application provides a comfortable way for the integration of Bluetooth LE sensors. Further, the integration of sensor data provided by a cloud data storage was implemented. This allows to collect data during the mobility of the elderly people. In addition, a historical sensor data service was implemented to provide a convenient way of storing sensor data. This allows to create daily or weekly reports. These reports can help, for example, to determine whether a person has drunk enough during the day or during the week.

The integration of the sensors was evaluated in the context of the BMBF (Federal Ministry of Education and Research) funded research project Cognitive Village. Here, the historical sensor data service was used by the dashboard system open.DASH [Uni17a], which was implemented at the University of Siegen in the context of the BMWi (German Federal Ministry for Economic Affairs and Energy) funded project SmartLive [Uni17b]. Further, the Withings cloud sensor service was used to access the data of the deployed Withings sensors during the project. Figure 7.2 shows a screenshot of open.DASH showing the sensor values of a Withings Pulse Ox and Withings BPM.

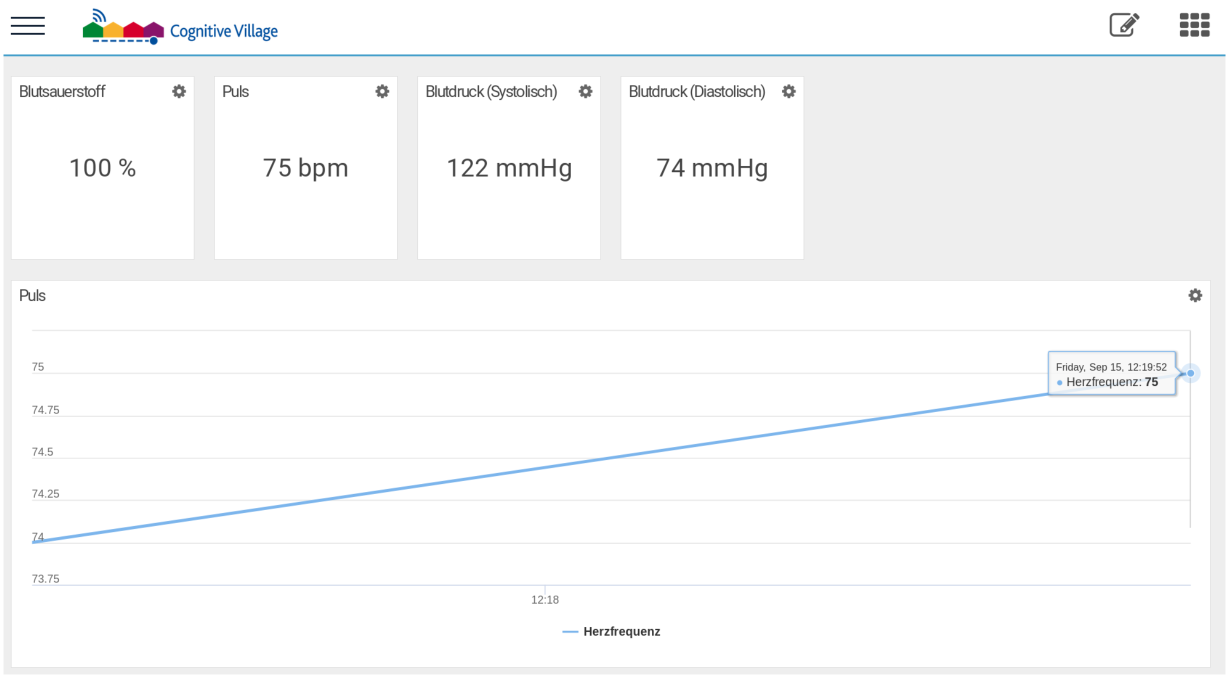


Figure 7.2: Visualization of Withings Data on open.DASH

The implementations were tested and evaluated by participating in a living lab experiment of the Cognitive Village project. The main goal of the living lab was the collection of sensor data during daily activities of the person living in the flat in order to train the pattern recognition algorithms developed within the project. The research focus was on the collection of physiological and behavioral data of the users in order to extract knowledge about the physical, cognitive, and emotional status of the users [SG16].

In the living lab experiment, an intelligent floor underlay called *SensFloor®* [STS+13] from the company FutureShape was installed in the whole flat (excluding the bathroom for privacy issues). SensFloor® is a textile-based underlay with integrated microelectronics and proximity sensors which can be used for activity monitoring, indoor localization and especially for fall detection. Further, a set of Estimote Bluetooth Beacons were installed in the flat for indoor localization as well. Besides these sensors, the person living in the flat was asked to wear several mobile devices: a smartphone (LG G5) and a smartwatch (Huawei Watch). Both devices were delivering sensor information about the movement of body and the hand of the person. The data was collected by a smartphone application which was implemented in a collaborative effort with other project members of the Cognitive Village project. The main purpose of the smartphone app was the management of the connectivity to the smartwatch and smartphone sensors. The smartphone was connected to a service gateway (Asus VivoMini UN45H) which used different parts of the presented architecture, e.g. for sensor data integration, data storage and service discovery. The main application running on the service gateway was the already mentioned pattern recognition software. Figure 7.3 shows a picture of the set of devices that were used during the living lab experiment.

All involved sensors in the Living Lab used the sensor integration services and the best-effort communication services from the presented architecture for sending their data. For example, the pattern recognition application had a subscription to the corresponding message exchanges at the message broker. The measurement data was stored in a database by using the historical sensor data service.

Summarized, it can be stated that the presented architecture and implementations regarding the integration of sensors helped to overcome the heterogeneity of the underlying technologies. Further, the flexibility of the proposed architecture regarding different programming languages was a benefit in the project Cognitive Village and allowed a heterogeneity of used programming languages. For example, the pattern recognition application was implemented both in C and in Python. The smartphone application was implemented in Java, as well as the Withings cloud sensor service and the historical data services. Further, the dashboard open.DASH used a combination of JavaScript and Java.

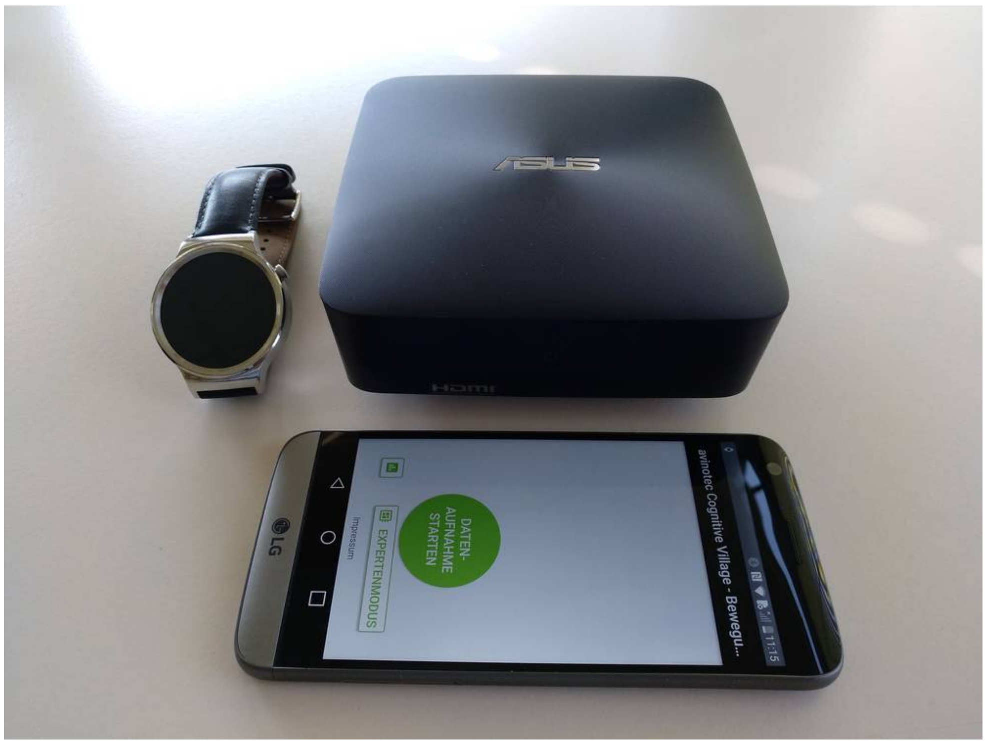


Figure 7.3: Living Lab - Hardware setup

7.2.3 Rate-Constrained Communication

The effectiveness of the implemented fault-tolerance techniques was evaluated in an experimental setup. The Traffic Shaping Layer as part of the presented fault-tolerance techniques was evaluated. As depicted in figure 7.4, the experimental setup consists of three computing nodes, each based on an Intel Atom processor (N270, 1.6 GHz) which runs a Linux kernel at version 4.8.0-36 with FQ_CoDel (Fair Queuing Controlled Delay) as default queueing discipline. Further, each node is connected to two separate networks. The first network is a management network based on Ethernet. The main purpose of the management network is to provide access to the node via Secure Shell (SSH). This network is only used in the presented experiments and is not required in the final system. The second network is a switched Ethernet network, which is exclusively used for the conducted measurements. The network switch used in the measurement network is a 24-port GbE Smart Managed Switch GS1900-24E from ZyXEL, which was limited to Fast Ethernet at each port for the experiments.

The effectiveness of the Traffic layer was evaluated in three experiments as follows. The results are listed in table 7.1. The measured latencies are Round-Trip Times (RTTs) of the packets.

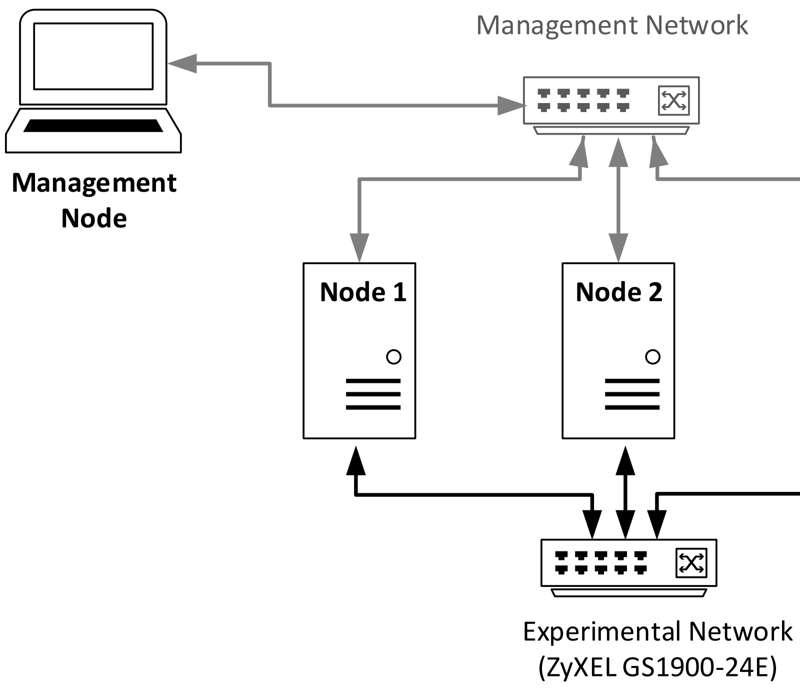


Figure 7.4: Rate-Constrained Communication - Experimental setup

RC traffic only without Traffic Shaping In a first step, the jitter and the packet-loss ratio for the rate-constrained traffic (User Datagram Protocol (UDP) packets with 560 bytes payload and period of 1ms) in an idle scenario were measured. In this scenario, where low jitter and zero packet loss is expected, the rate-constrained messages are sent from node 1 to node 2 without any other background network traffic. This results in an average bandwidth utilization at the network link of 4.80% (4.80 Mbits/sec) with a maximum jitter of 0.052 ms and zero packet loss (0 of 321330 sent packets).

RC and BE without Traffic Shaping In the next step, background workload (best effort) was added by sending Transmission Control Protocol (TCP) messages with 560 Bytes payload at maximum speed from node 1 to node 3. By sending the RC and the BE traffic to two different nodes, the messages are enqueued at two different egress queues in the switch and the RC traffic contends at the egress queue of node 1 with the BE traffic. During this experiment, a packet loss ratio of 37% (117626 of 321330 sent packets) was measured with an average bandwidth utilization of 4.75% (4.75 Mbits/sec) and maximum jitter of 2.479 ms regarding the rate-constrained traffic. This experiment shows clearly that without any traffic shaping mechanisms, the rate-constrained traffic is jeopardized by the best effort background traffic.

Listing 7.1: Qdisc Configuration for Traffic Shaping

```
tc qdisc del dev enp1s0 root
tc qdisc add dev enp1s0 root handle 1: prio bands 3
tc qdisc add dev enp1s0 parent 1:1 handle 10 prio bands 3
tc qdisc add dev enp1s0 parent 1:2 handle 20 tbf /
rate 80mbit latency 50ms burst 600k
tc qdisc add dev enp1s0 parent 1:3 handle 30 pfifo_fast
tc filter add dev enp1s0 parent 1:0 prio 1 protocol ip /
u32 match ip tos 0x28 0xff flowid 1:1
tc filter add dev enp1s0 parent 1:0 prio 2 protocol ip /
u32 match ip tos 0x48 0xff flowid 1:2
tc filter add dev enp1s0 parent 1:0 prio 3 protocol ip /
u32 match ip tos 0x58 0xff flowid 1:3
```

RC and BE with Traffic Shaping For the last scenario, Traffic Control in the Linux kernel was used as shown in Listing 7.1. Here, the default root qdisc was changed to a PRIO qdisc with 3 bands [LIN20c]. The first band with the highest priority is designated for the rate-constrained traffic. For the experimental setup, rate-constrained traffic was assigned the highest Type of Service (ToS) level and mapped to the first band by using an *ip tos filter*. In addition, more complex filters can be applied to assign traffic to the different bands. Further, a Token Bucket Filter (TBF) qdisc is assigned to the second band in order to limit the traffic for streaming data. Finally, the third band is configured as PFIFO_FAST qdisc for best effort messages.

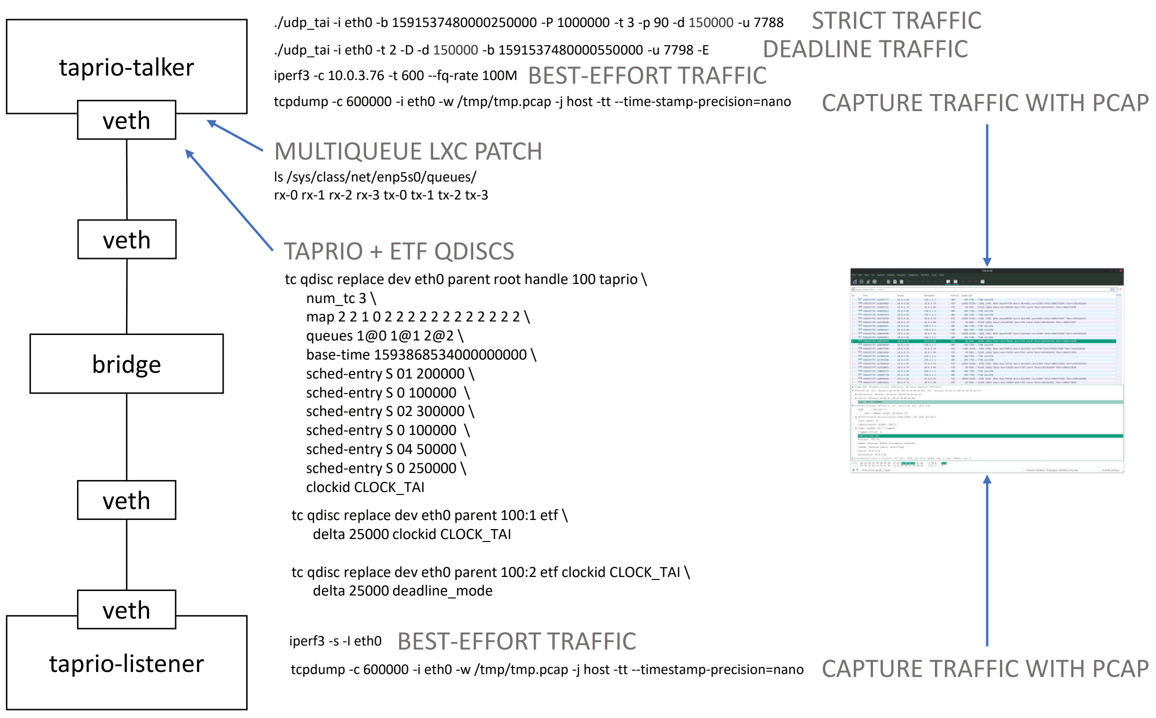
The result shows the effectiveness of the Traffic Shaping Layer: the background traffic does not affect any more the rate constrained traffic regarding the jitter (which is similar to the jitter measured in the idle scenario) and leads to no packet loss of the rate constrained traffic. The average bandwidth utilization was 4.80% (4.80 Mbits/sec) with a maximum jitter of 0.061 ms and zero packet loss (0 of 321330 packets).

Traffic	Payload	max. Latency (RTT)	max. Jitter	Packet Loss
No traffic shaping				
RC only	560 bytes (RC)	0.076 ms	0.052 ms	0 %
RC & BE	560 bytes (each)	7.988 ms	2.479 ms	37 %
Traffic Shaping Layer				
RC only	560 bytes (RC)	0.076 ms	0.053 ms	0 %
RC & BE	560 bytes (each)	1.227 ms	0.061 ms	0 %

Table 7.1: Rate-Constrained communication - Traffic characteristics

7.2.4 Real-Time Communication

The approach for real-time communication presented in this thesis establishes a hierarchical communication model with a physical real-time communication network on the lowest level and a virtual real-time communication network as a second level. The virtual real-time communication network is used for the communication within one node between different containers (cf. Figure 5.5). A bridge handles the routing between both levels. This section shows the evaluation results of the virtual real-time communication network at second level regarding the packet transport latency and jitter.



```
./udp_tai -i eth0 -b 1591537480000250000 -P 1000000 -t 3 -p 90 -d 150000 -u 7788
./udp_tai -i eth0 -t 2 -D -d 150000 -b 1591537480000550000 -u 7798 -E
iperf3 -c 10.0.3.76 -t 600 --fq-rate 100M
tcpdump -c 600000 -i eth0 -w /tmp/tmp.pcap -j host -tt --time-stamp-precision=nano
```

```
ls /sys/class/net/enp5s0/queues/
rx-0 rx-1 rx-2 rx-3 tx-0 tx-1 tx-2 tx-3
```

```
tc qdisc replace dev eth0 parent root handle 100 taprio \
    num_tc 3 \
    map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2 \
    queues 1@0 1@1 2@2 \
    base-time 1593868534000000000 \
    sched-entry S 01 200000 \
    sched-entry S 0 100000  \
    sched-entry S 02 300000 \
    sched-entry S 0 100000  \
    sched-entry S 04 50000 \
    sched-entry S 0 250000 \
    clockid CLOCK_TAI

tc qdisc replace dev eth0 parent 100:1 etf \
    delta 25000 clockid CLOCK_TAI

tc qdisc replace dev eth0 parent 100:2 etf clockid CLOCK_TAI \
    delta 25000 deadline_mode
```

```
iperf3 -s -I eth0
tcpdump -c 600000 -i eth0 -w /tmp/tmp.pcap -j host -tt --timestamp-precision=nano
```

Figure 7.5: Hierarchical real-time communication - Experimental setup

The experimental setup shown in Figure 7.5 follows the hierarchical structure presented in Figure 6.2 and introduces two Linux Containers (LXC) containers called *taprio-talker* and *taprio-listener*. Both containers are based on the Linux distribution Arch Linux which was also used as the basis for the system running the hypervisor LXC in version 4.0.2 patched with the multiqueue patch developed in the context of this thesis. Both containers are connected via a virtual Linux network bridge. Since LXC is partly responsible for the isolation of resources by using namespaces (cf. Section 5.5.1) for the spatial isolation of network resources (cf. Table 5.1), the egress veth devices of the container are created in the network namespace of the container. Since this network namespace is different from the host and other containers, the network namespace of a container cannot be accessed without special permissions (which e.g. is available via the root user) from other containers or the host. Further, LXC creates unnamed namespaces, which means that the created network namespaces cannot be accessed without mounting the network namespace to a network namespace mount under */run/netns*. Since the presented experiments require the application of the Time-Aware Priority Shaper (TAPRIO) queueing discipline at the egress *veth* device of both containers, the following procedure is used to mount the network namespace of a container (in the example the *taprio-talker*) in order get access to the network namespace of the container. At first, the Process identifier (PID) of the container has to be retrieved. This can be done by using the LXC command line tool *lxc-info* as shown in Listing 7.2.

Listing 7.2: Output of the command line tool *lxc-info* for the taprio-talker container

```
Name: taprio—talker
State: RUNNING
PID: 1743
IP: 10.0.3.41
IP: 10.0.3.42
CPU use: 186.63 seconds
BlkIO use: 40.92 MiB
Memory use: 83.45 MiB
KMem use: 24.70 MiB
Link: taprio—talker
 TX bytes: 106.29 GiB
 RX bytes: 483.82 MiB
 Total bytes: 106.76 GiB
```

The PID can then be used to mount the network namespace as shown in Listing 7.3 to get a bash shell to the network namespace of the container. The bash shell is then used in the experiments for assigning the TAPRIO queueing discipline to the *veth* device of the corresponding container.

Listing 7.3: Mounting the network namespace of the taprio-talker container

```
mkdir /run/netns
touch /run/netns/taprio—talker
mount —o bind /proc/1743/ns/net /run/netns/taprio—talker
ip netns exec taprio—talker bash
```

Having a look at the Linux virtual network bridge, which connects the corresponding pairs of the *veth* devices of the container, it can be seen that the bridge as well as the *veth* are created within the namespace of the host (cf. Listing 7.4).

Listing 7.4: Bridge and *veth* devices in host network namespace

```
[lxc−taprio ~]# ip addr show | egrep "lxcbr0|taprio−talker|taprio−listener"
4: lxcbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
   qdisc noqueue state UP group default qlen 1000
   inet 10.0.3.1/24 scope global lxcbr0
5: taprio−talker@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
   qdisc noqueue master lxcbr0 state UP group default qlen 1000
6: taprio−listener@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
   qdisc noqueue master lxcbr0 state UP group default qlen 1000
```

Throughout the experiments, different periodic schedules were created and evaluated. A common attribute of all schedules was a cycle time of 1000000 nanoseconds (1 millisecond). In TAPRIO the cycle time does not have to be defined explicitly, but the cycle time is calculated from the sum of all periods defined by the sched-entry attributes: $\sum_{i=1}^{N} \frac{Q_i}{T_i}$. Further, the Earliest TxTime First (ETF) queueing discipline was used. Main purpose of the ETF queueing discipline is to ensure a correct chronological order of packets. TAPRIO in combination with the ETF queueing discipline allows to send scheduled traffic in two different modes: *strict* and *deadline*. The first mode *strict* allows to define a transmission time for each packet. This allows a very precise control of the instant when a packet is sent via the network interface. However, this requires a physical network interface that supports hardware offloading (also named LaunchTime Control or Time-Based Scheduling) like the Intel® i210 network card. The second mode *deadline* implements the behavior of the Time-Aware Shaper (TAS) (cf. Section 2.2.4) defined in the enhancements for scheduled traffic in IEEE 802.1Qbv. Packets that are sent with this mode are enqueued as soon as possible within the assigned period. However, the packets are ordered according to their assigned timestamps. If a packet cannot be sent within that period, it is dropped.

Listing 7.5: TAPRIO schedule and ETF settings

```
tc qdisc replace dev eth0 parent root handle 100 taprio \
      num_tc 3 \
      map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2 \
      queues 1@0 1@1 2@2 \
      base−time 1593271066000000000 \
      sched−entry S 01 200000 \
      sched−entry S 0 100000 \
      sched−entry S 02 300000 \
      sched−entry S 0 100000 \
      sched−entry S 04 50000 \
      sched−entry S 0 250000 \
        clockid CLOCK_TAI

tc qdisc replace dev eth0 parent 100:1 etf \
        delta 25000 clockid CLOCK_TAI

tc qdisc replace dev eth0 parent 100:2 etf clockid CLOCK_TAI \
        delta 25000 deadline_mode
```

Figure 7.6 shows the concrete schedule used for the presented results. This schedule consists of three scheduled time-slots for the traffic classes *strict* (periodic traffic with Linux Socket Buffer (SKB) priority 3), *deadline* (periodic deadline traffic with SKB priority 2) and *best-effort* (traffic with all other SKB priorities). Each time-slot is followed by a guard band (GB) which is configured for the traffic classes *strict* and *deadline* to 100 microseconds and for the traffic class *best-effort* to 250 microseconds.

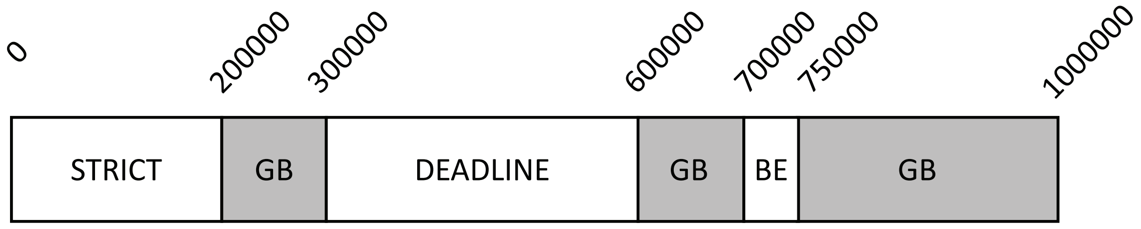


Figure 7.6: Hierarchical real-time communication - Communication schedule

Guard bands can be configured in TAPRIO by assigning the gatemask 0 to the corresponding *sched-entry* parameter. Here it is important to know that the *sched-entry* parameter uses a bit mask for the transmission selection of the traffic classes. A bit mask with the value of 0x01 means that the first traffic class is meant, which is in fact the traffic class 0. A gatemask with a bit mask of 0x00 means that all gates are closed (cf. [IEE16a]). The guard bands are required because packets can get queued in the transmission queues of the network devices, which is also true for the used *veth* devices in this experiment. Here, the guard bands allow the network device to empty the queues before the gate for the next traffic class is opened by the TAPRIO queueing discipline. In this way, the guard bands prevent the interfering of queued packets to the transmission windows of other traffic classes.

The *queues* parameter of the TAPRIO schedule in Listing 7.5 then maps the three traffic classes to the queues of the *veth* device. It is also possible to put a further queueing discipline between the TAPRIO queueing discipline and the queues of the *veth* devices, which is also done within the presented configuration. Here, two additional ETF queueing disciplines are introduced where the first queueing discipline (100:1) is responsible for the *strict* traffic class and the second queueing discipline (100:2) is responsible for the *deadline* traffic class (the option *deadline_mode* also indicates this). Figure 7.7 shows the packet flow for each traffic class. The ETF queueing disciplines are used to have precise control of the instant when a packet gets enqueued from the traffic control layer of the Linux network stack into the network device, which is the *veth* device in the presented experimental setup. In order to fulfill this task, the ETF queueing discipline uses the transmission time (txtime) assigned to the packet in the SKB structure and sorts the packets in the queue according to the transmission time (earliest transmission time first). Further, the ETF queueing discipline has an optional parameter *deadline_mode* which changes the behavior of the queueing discipline in such a way that the transmission time of the packet in the SKB structure is set to *now* [LIN20b]. This has the effect, that enqueued packets are enqueued as soon as possible by the ETF queueing discipline. However, if packets cannot get enqueued during the specified interval, the ETF queueing discipline drops the packet.

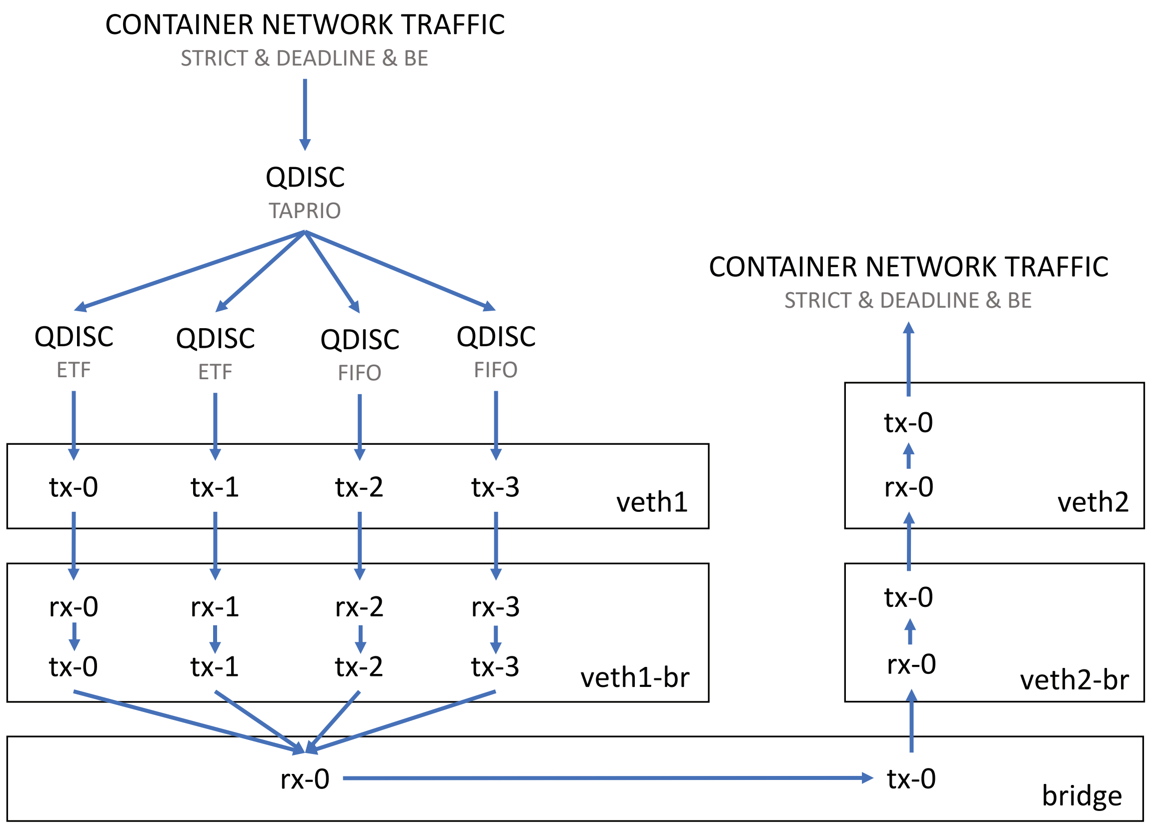


Figure 7.7: Container-to-container communication - Traffic flow

As it can be seen in Figure 7.7, all three traffic classes are first processed by the TAPRIO queueing discipline which allows packets passing according to the schedule provided by the *sched-entry* configuration parameters. The packets from both traffic classes *strict* and *deadline* are each mapped to the respective ETF qdisc. All other traffic classes (best-effort traffic) are mapped to two First In First Out (FIFO) qdiscs. Further, all four queueing disciplines are each mapped to a single transmission queue (tx-0 up to tx-3) of the *veth* device. Since *veth* devices are technically interconnected by connecting the transmission queues of the first *veth* device to the reception queues of the second *veth* device (and vice versa), the packets are instantly received at the second *veth* device, which is itself further connected to the Linux bridge. The bridge itself then forwards the packets to the target *veth* device. The following system setup was used for the presented results:

- Intel® Core(TM) i7-8700 CPU @ 3.20GHz
- Linux kernel: 5.6.16-1-MANJARO (already contains the taprio use-after-free fix)
- Hyper-Threading and Turbo Boost disabled in BIOS
- Two additional Linux kernel boot parameters: *processor.max_cstate=1* and *idle=poll*
- LXC in version 4.0.2 (with multiqueue patch)

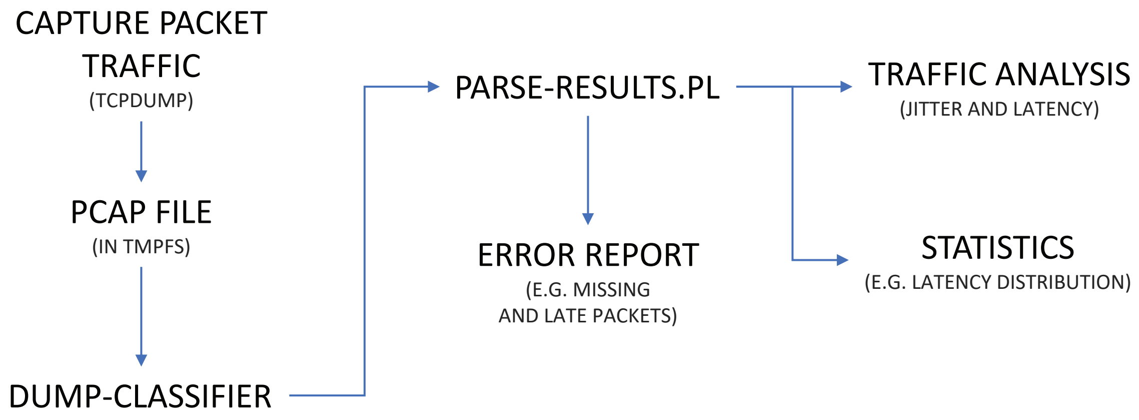


Figure 7.8: Hierarchical real-time communication - Toolchain and methodology

For all experiments, the packet timestamps were measured with the command line tool *tcpdump*, which itself uses *libpcap* for the capturing of packets. Figure 7.8 shows tcpdump embedded in the toolchain used for the presented experiments. Regarding the packet capturing, it is important to know that the Linux Network protocol stack has dedicated measurements points for network taps like libpcap. When packets are received, the packets are forwarded to the network tap directly after the SKB has been created for the packet (in particular before a potential ingress queueing discipline). At the egress side, packets are forwarded to the network tap directly when they leave the (last) queueing discipline assigned to the network interface. Further, the packet dump files were written to a tmpfs file system [LIN20d] in order to minimize the latency of the tcpdump thread. The packet dump files were then pre-processed by the command line tool dump-classifier [SP20b] and the further processed by the Perl script *parse-results.pl*. *parse-results.pl* was implemented within the scope of this thesis in order to check for missing or late packets and to generate traffic statistics.

For the transmission of periodic (traffic class *strict*) and deadline (traffic class *deadline*) traffic, the program udp_tai was used [SP20b]. Listing 7.6 shows the command syntax for the traffic class *strict* and Listing 7.7 the command syntax for the traffic class *deadline*. Further, the program udp_tai was modified as part of the experiments to use the SCHED_DEADLINE scheduler of the Linux kernel instead of the provided real-time priorities for deadline traffic. The payload of each packet is always 256 bytes and contains the timestamp at which the packet should be sent.

Listing 7.6: udp_tai command example for periodic traffic (traffic class *strict*)

```
[lxc−taprio ~]# ./udp_tai −i $IFACE −b $BASE\_TIME −P 1000000 −t 3 −d 150000 −u 7788
```

Listing 7.7: udp_tai command example for deadline (traffic class *deadline*) traffic

```
[lxc−taprio ~]# ./udp_tai −i $IFACE −t 2 −d 150000 −D −b $BASE\_TIME −u 7798 −E
```

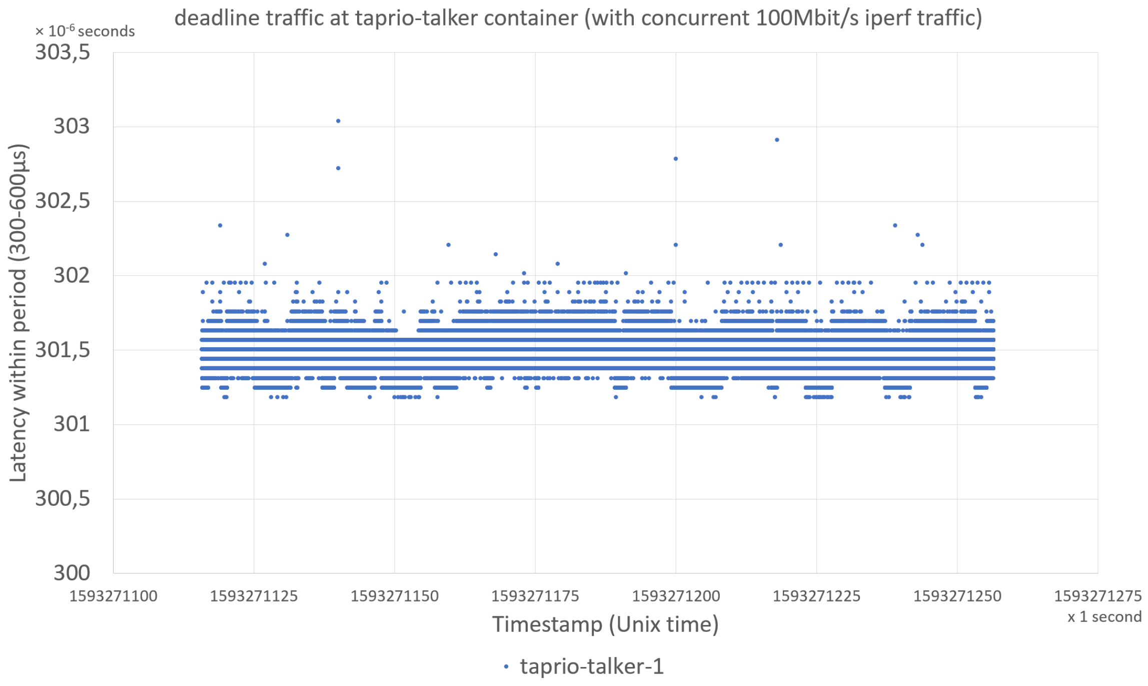


Figure 7.9: Packet latency at the taprio-talker container

In addition to the *strict* and *deadline* traffic classes, the Linux network stack was loaded with best-effort background traffic (here iperf3 was used) at a bit rate of 100Mbits per

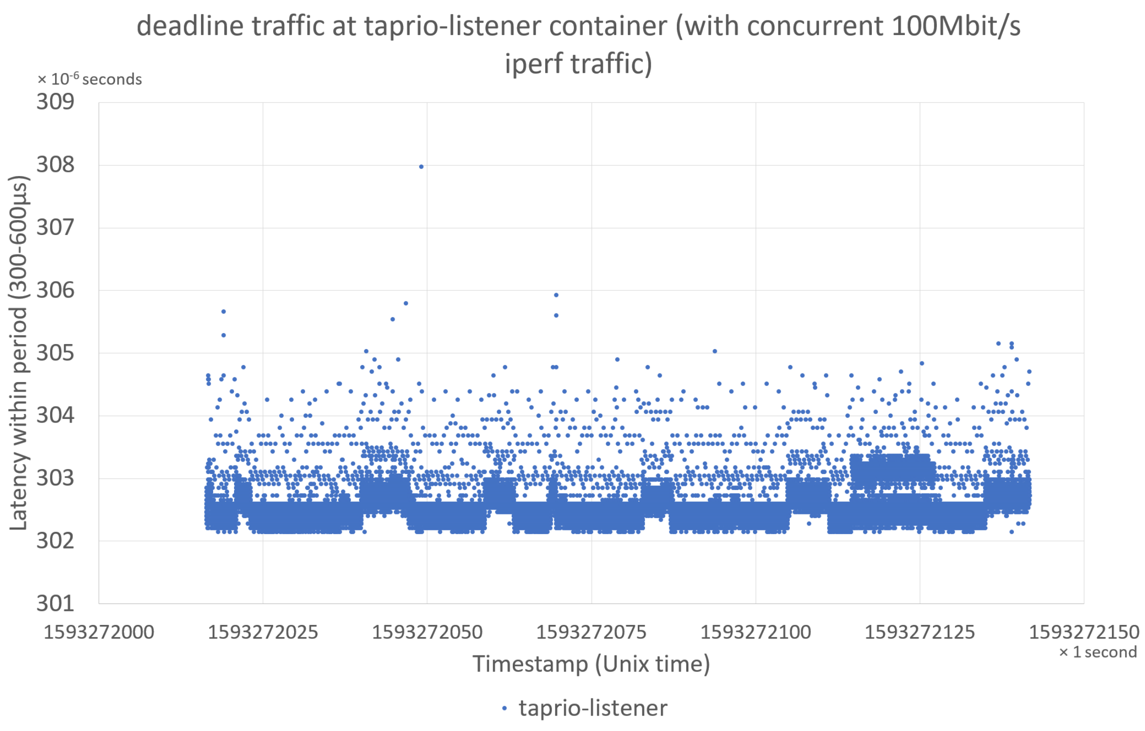


Figure 7.10: Packet latency at the taprio-listener container

second sent from the taprio-talker to the taprio-listener. Figure 7.9 and Figure 7.10 show the respective measured latencies at the taprio-talker and at the taprio-listener. The latencies are plotted on the y-axis, while the x-axis shows the progression of time as Unix timestamp. The Unix timestamp represents a point in time as the numbers of seconds passed since 00:00:00 UTC on January 1st 1970.

Both figures show that the latencies introduced by the Linux network stack and in particular by the TAPRIO and the ETF queueing disciplines are at a low microsecond range. By having a look at the latencies at the taprio-listener, one can see that the latencies introduced by the Linux virtual switch are also in the low microsecond range. The measured jitter is about 2 microseconds at the talker and about 6 microseconds at the talker. Another remarkable aspect is the narrow band of the measured latencies in Figure 7.11. During all experiments, no packets were dropped due to a missed deadline.

Listing 7.8: TAPRIO limiting the throughput of best-effort traffic

```
[root@taprio—talker ~]# iperf3 —c 10.0.3.223 —t 86400 ——fq—rate 1000M
[ 5] 159.00—160.00 sec 46.5 MBytes 390 Mbits/sec 0 35.4 KBytes
[ 5] 160.00—161.00 sec 46.6 MBytes 391 Mbits/sec 0 35.4 KBytes
[ 5] 161.00—162.00 sec 46.5 MBytes 390 Mbits/sec 0 35.4 KBytes
```

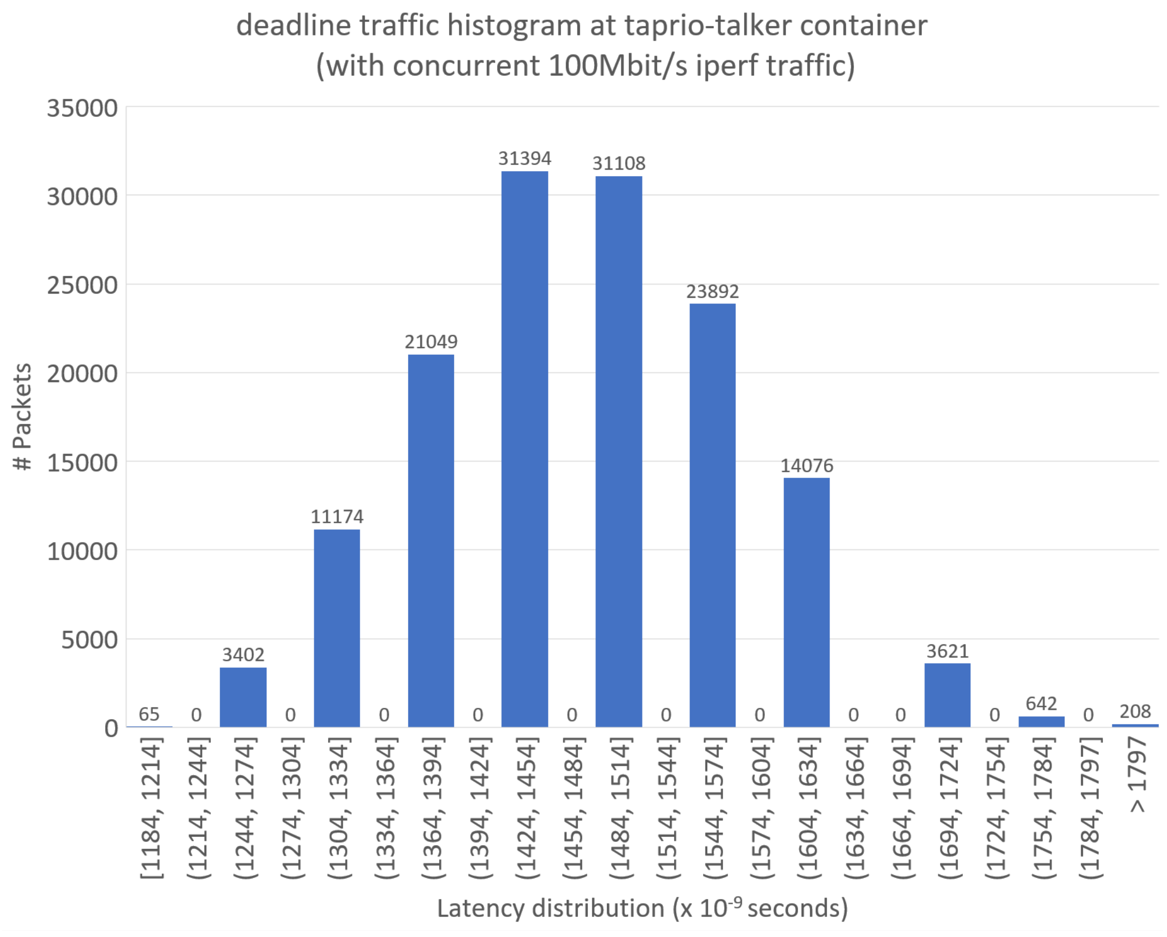


Figure 7.11: Packet latency at the taprio-talker container as histogram

Another interesting aspect discovered during the experiments is that the TAPRIO queueing discipline is capable of limiting the traffic throughput of a traffic class. As Listing 7.8 shows, the TAPRIO queueing discipline only allows best-effort traffic to be transmitted with a bit rate of approx. 390 Mbits per second (instead of 1000Mbits/sec as requested in the iperf3 command). This is due to the fact, that in the used TAPRIO schedule (cf. Listing 7.5) only a transmission period of 50 microseconds is assigned for best-effort traffic.

Preemptive Linux Kernel A further experiment was executed with a Linux Kernel that was patched with the PREEMPT_RT patch (5.6.14_rt7-2-MANJARO). As with the previous experiments, udp_tai was used to send deadline traffic in the corresponding period. Here, a lot of packets were dropped by the ETF queueing discipline because of a missed deadline. Dropped packets can be identified by using the SOF_TXTIME_REPORT_ERRORS flag of the SO_TXTIME Application Programming Interface (API) [SP20a]. A further method that was used for checking for missing and late packets is the Perl script *parse_results.pl*.

Figure 7.12 shows the packet latencies measured with the preemptive Linux kernel. Here, it can clearly be seen that the TAPRIO queueing discipline is periodically preempted by a task. Packets that were dropped due to a missed deadline are drawn in Figure 7.12 in red color. In the presented results, 28 packets out of a total number of 90041 deadline packets were dropped due a missed deadline. A possible reason for the packet drops when using the PREEMPT_RT patch is that the Linux kernel is made preemptive with the PREEMPT_RT patch, and this also applies for the network stack as part of the Linux kernel. Here, the processing and transmission of network packets in the queueing disciplines, the *veth* devices and the Linux network bridge can be preempted by user space applications (with high priorities).

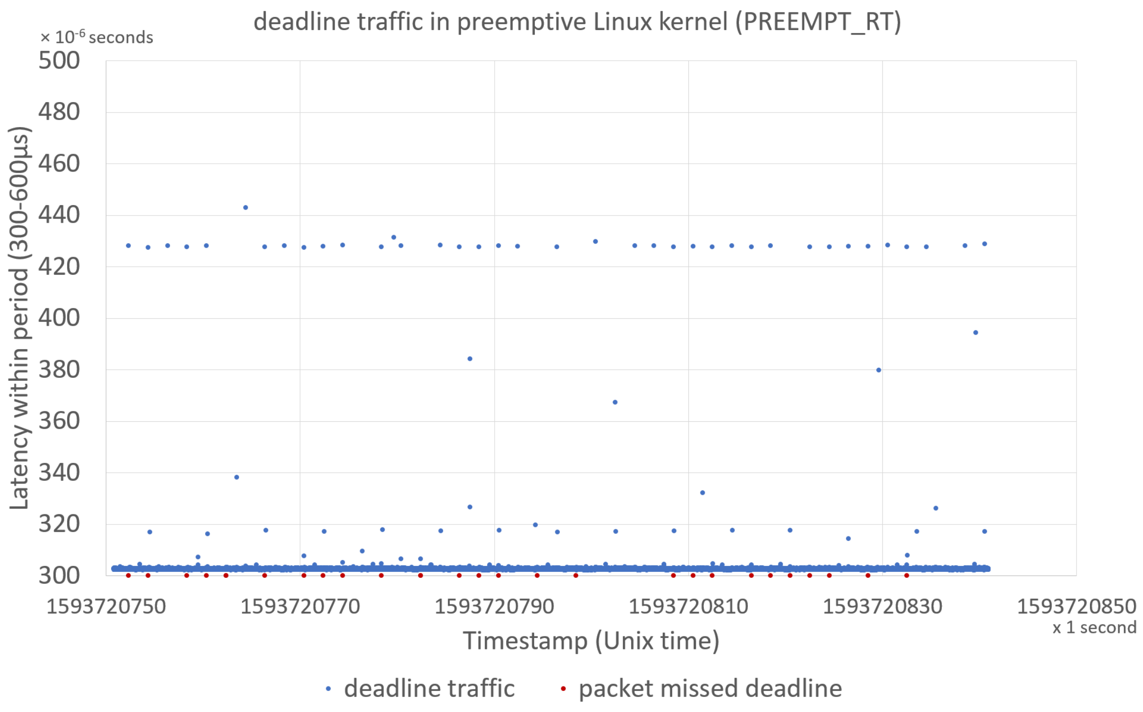


Figure 7.12: Packet latency at the taprio-talker container (preemptive Linux Kernel)

Scalability In order to evaluate the scalability of the presented approach, further experiments were executed with different amounts of containers. A real-time communication setup with multiple containers can be achieved by several ways. The first approach is to have a separate traffic class for each container. Here, the performance is equivalent to the already presented approach with one taprio-talker and one taprio-listener. A downside of the first approach is that it is limited by the maximum number of traffic classes, which is 16 for container-to-container communication and 8 for host-to-host communication using Time Sensitive Networking (TSN). However, this results in a predictable temporal behavior, as already presented.

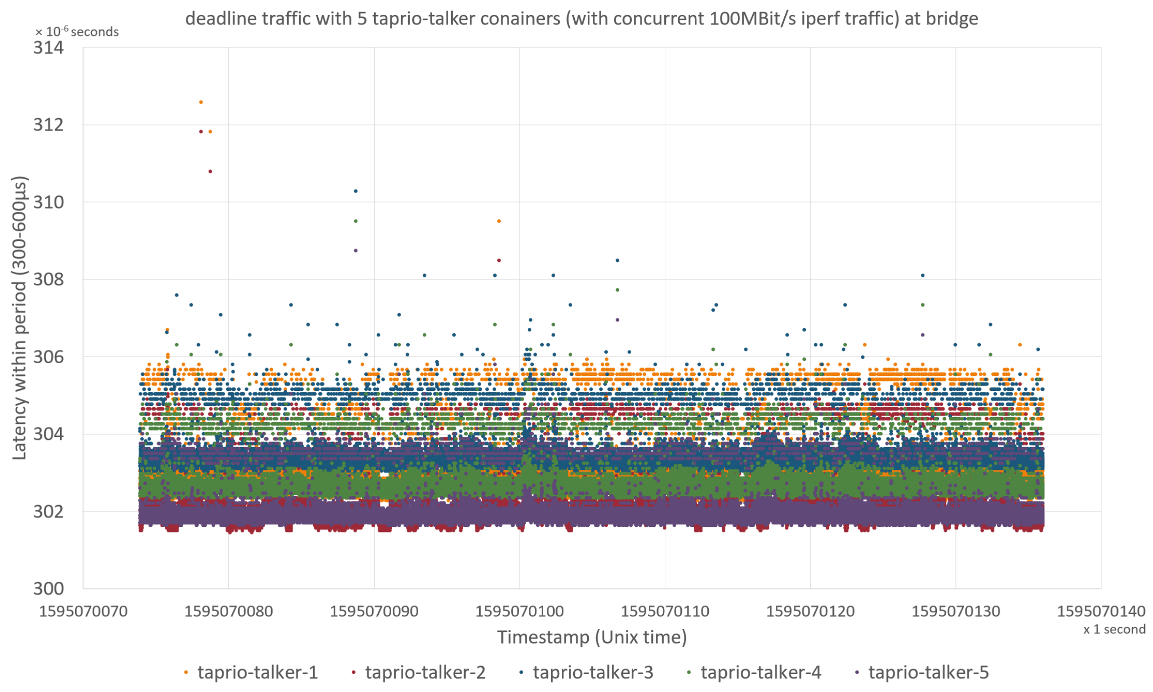


Figure 7.13: Packet latency for 5 taprio-talker containers (at network bridge)

Container	Min. Latency (nanoseconds)	Max. Latency (nanoseconds)	Avg. Latency (nanoseconds)	Jitter (nanoseconds)	Missed Deadline
taprio-talker-1	2272	12576	2951	10304	0
taprio-talker-2	1440	11808	2063	10368	0
taprio-talker-3	2976	10272	3379	7296	0
taprio-talker-4	2272	9504	2655	7232	0
taprio-talker-5	1632	8736	1920	7104	0

Table 7.2: Traffic statistics for 5 taprio-talker container (at network bridge)

A second approach is to let all containers share the same traffic class for time critical traffic. This approach does not have a limitation regarding possible traffic classes. However, it is expected that an increasing number of containers sharing the same traffic class will introduce more latency as well as jitter to the real-time communication. A further way is a combination of the first and the second approach. The goal of following experiments is to evaluate the magnitude of the additional introduced latency and jitter when using the second approach.

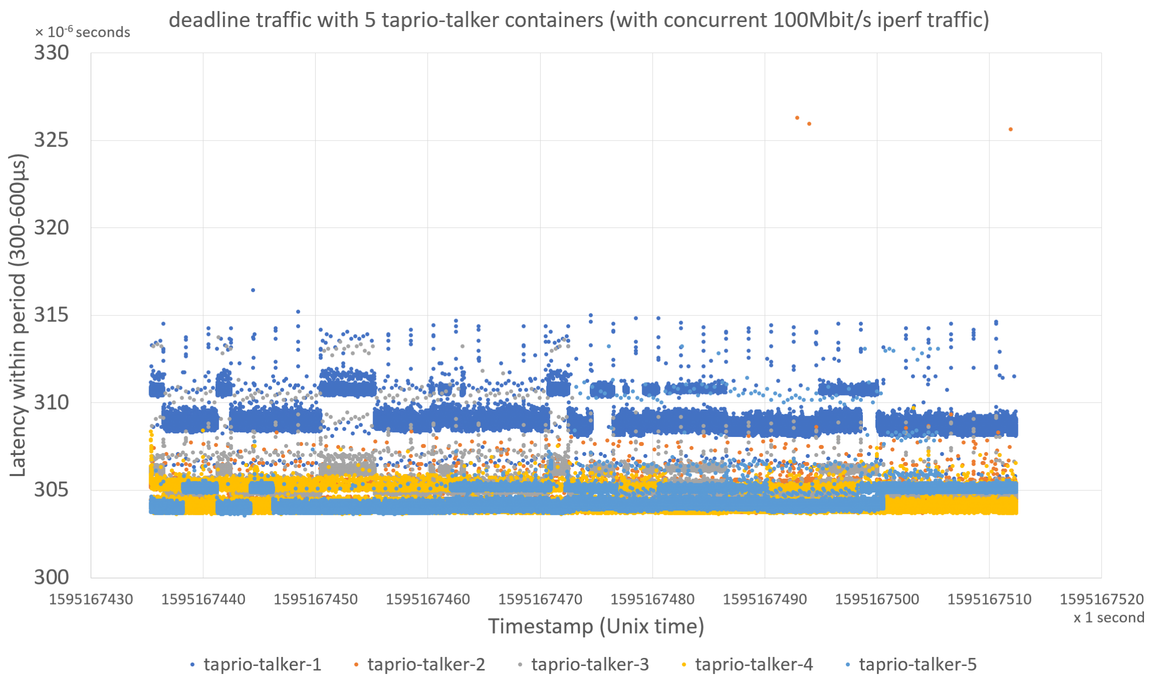


Figure 7.14: Packet latency for 5 taprio-talker containers (at taprio-listener)

Container	Min. Latency (nanoseconds)	Max. Latency (nanoseconds)	Avg. Latency (nanoseconds)	Jitter (nanoseconds)	Missed Deadline
talker-deadline-1	5024	16416	8496	11392	0
talker-deadline-2	3680	26272	4822	22592	0
talker-deadline-3	4000	13728	4755	9728	0
talker-deadline-4	3616	31712	4629	28096	0
talker-deadline-5	3552	13216	5168	9664	0

Table 7.3: Traffic statistics for 5 taprio-talker container (at taprio-listener)

The following experiment comprises a setup of 5 taprio-talker containers sending traffic to 5 taprio-listener containers. Figure 7.13 shows the measured latencies for the taprio-talker containers at the ingress side of network bridge. It can be seen that the delay and jitter at the bridge for the majority of the packets ranges from 2μs to 6μs. However, there are some few packets having a maximum latency with up to 13μs. These exceptions are probably due to spinlocks on the ingress side of the Linux kernel's network stack or due to the high workload of the memory controller when writing the captured packets to the tmpfs memory filesystem. Table 7.2 shows a summary of the packet latencies for the presented experiment. Figure 7.14 shows the latencies and jitter measured in the ingress of the taprio-listener containers. The results show that the network bridge has added some further delay and jitter to the communication. The majority of the packets have a measured latency from 4μs to 15μs. However, as with the measurements at the bridge (cf. Figure 7.13) there are some few packets that exceed the band of 4μs to 15μs with a measured latency up to 31μs. Table 7.3 shows a summary of the packet latencies measured at the taprio-listener containers.

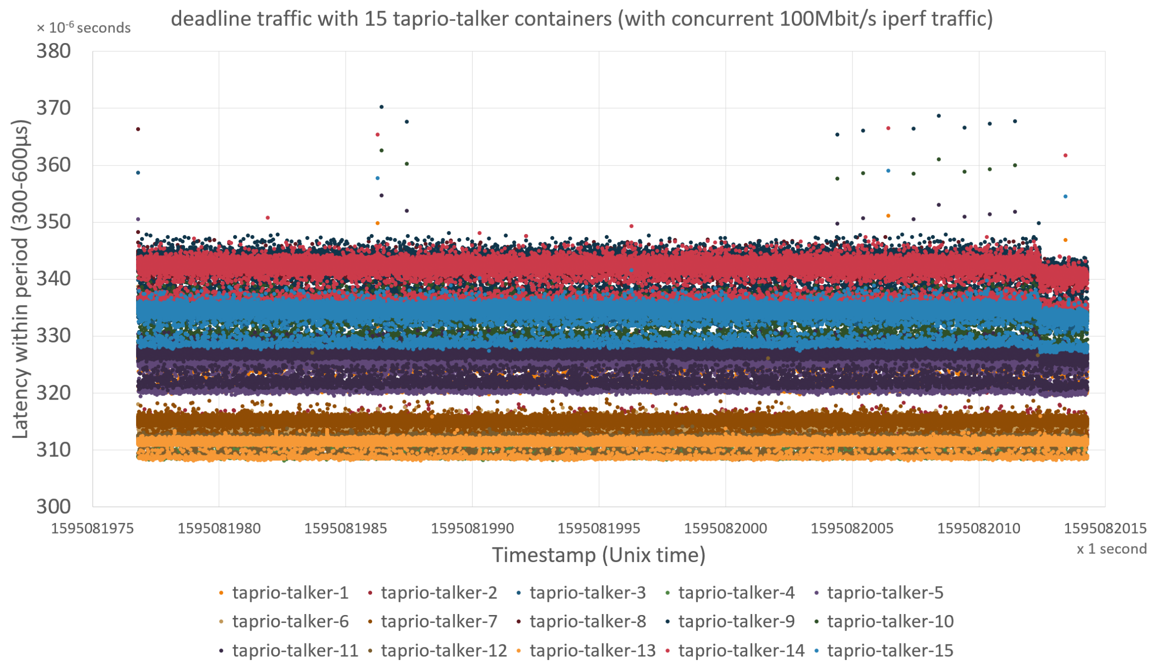


Figure 7.15: Packet latency for 15 taprio-talker containers (at taprio-listener)

The next experiment was conducted with a setup of 15 taprio-talker containers sending traffic to 5 taprio-listener containers. Figure 7.15 shows the measured latencies in the taprio-listener containers combined in one figure. The results show that the majority of the packets reach the taprio-listener containers with a band of 9μs to 35μs. As with the setup of five taprio-talker containers, there are some packets exceeding this band with a measured latency up to 70μs. As with the previous experiments, these exceptions are probably due to spinlocks on the ingress side of the Linux kernel's network stack or due to the high workload of the memory controller when writing the captured packets to the tmpfs memory filesystem. Table 7.4 shows a summary of the packet latencies and jitter for this experiment. One interesting fact is the increasing offset when the first packets are sent. This offset has increased from about 2 microseconds (from the experiment with 5 taprio-talker containers) to about 8 microseconds (in the experiment with 15 taprio-talker containers). A possible reason for this increasing delay could be management overhead in the used queueing disciplines TAPRIO and ETF.

As expected, delay and jitter of the packet transmission increase with the number of simultaneous containers using the same traffic class. However, a number of 20 concurrent containers within one traffic class leads to the loss of packets (cf. Table 7.5). The

Container	Min. Latency (nanoseconds)	Max. Latency (nanoseconds)	Avg. Latency (nanoseconds)	Jitter (nanoseconds)	Missed Deadline
taprio-talker-1	34208	66336	40543	32128	0
taprio-talker-2	35040	70240	41537	35200	0
taprio-talker-3	27744	62560	34100	34816	0
taprio-talker-4	20320	54624	26446	34304	0
taprio-talker-5	8288	37536	11710	29248	0
taprio-talker-6	8096	15904	10960	7808	0
taprio-talker-7	34272	66464	40588	32192	0
taprio-talker-8	27104	59040	33371	31936	0
taprio-talker-9	19680	51104	25349	31424	0
taprio-talker-10	10976	39840	14575	28864	0
taprio-talker-11	26656	58656	32900	32000	0
taprio-talker-12	8096	40352	10360	32256	0
taprio-talker-13	19360	50464	24301	31104	0
taprio-talker-14	10592	38688	13831	28096	0
taprio-talker-15	11104	42592	14734	31488	0

Table 7.4: Traffic statistics for 15 taprio-talker containers (measured at taprio-listener)

packet loss is because of packets being dropped in the ETF queueing discipline due to a missed deadline. The reason for the dropped packets are not expected to be due to insufficient bandwidth because even in the case of 20 concurrent containers, there is a lot of unused bandwidth remaining (cf. Figure 7.16). Moreover, it seems that the administrative overhead in the ETF queueing discipline seems to be the reason for the missed deadlines.

Regarding the scalability of the presented approach, it was shown that multiple container can use the same traffic class without negative impacts like packet loss with an amount of 15 containers in a period of 100 μs (cf Table 7.4). For example, this allows a setup with up

Number of Containers	Latency			Jitter (nanoseconds)	Total Packets	Missed Deadlines	Missed Deadlines (%)
	Min.	Max.	Avg.				
1	2144	7968	2493	5824	125291	0	0
5	2552	31712	5574	29160	385305	0	0
15	8096	70240	27077	62144	562410	0	0
20	5152	81312	25620	76160	600040	44	0,0074
25	5408	157920	42497	152512	600075	88	0,0147
30	6112	187040	47894	180928	601140	1168	0,1943

Table 7.5: Overall traffic statistics (measured at taprio-listener)

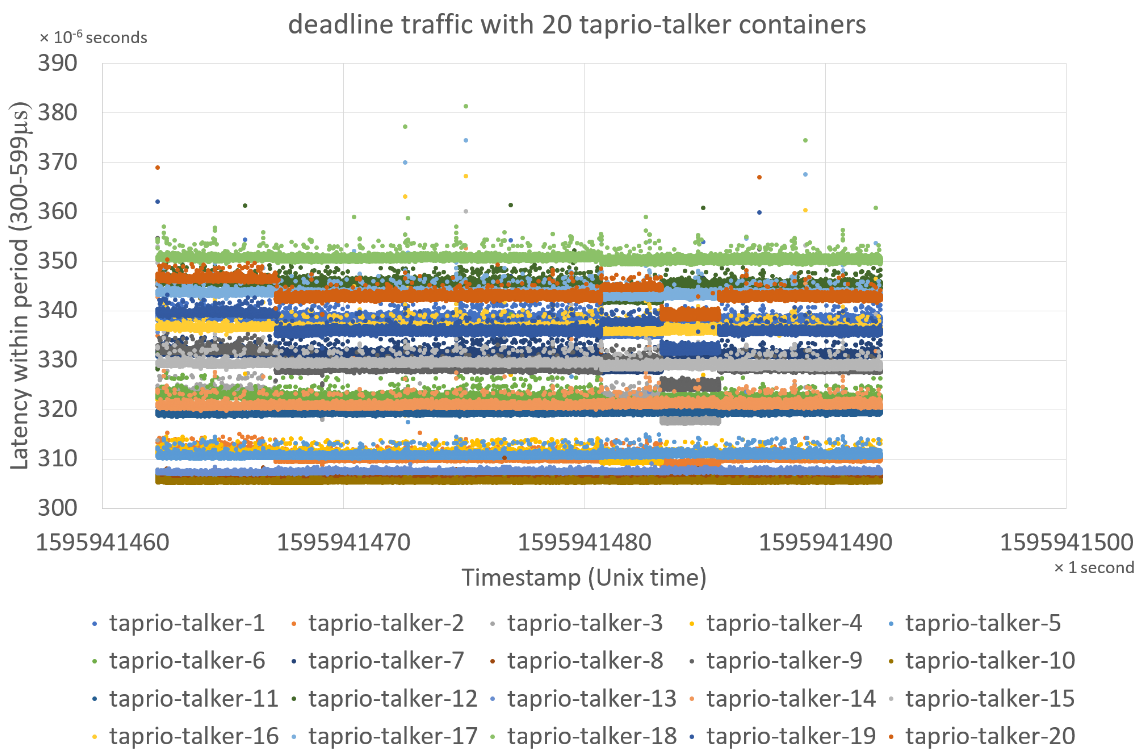


Figure 7.16: Packet latency for 20 taprio-talker containers (at taprio-listener)

to 135 containers using hierarchical real-time communication in a cycle of 1ms with nine periods of 100 μs each for time critical traffic and a period of 100 μs for best effort traffic.

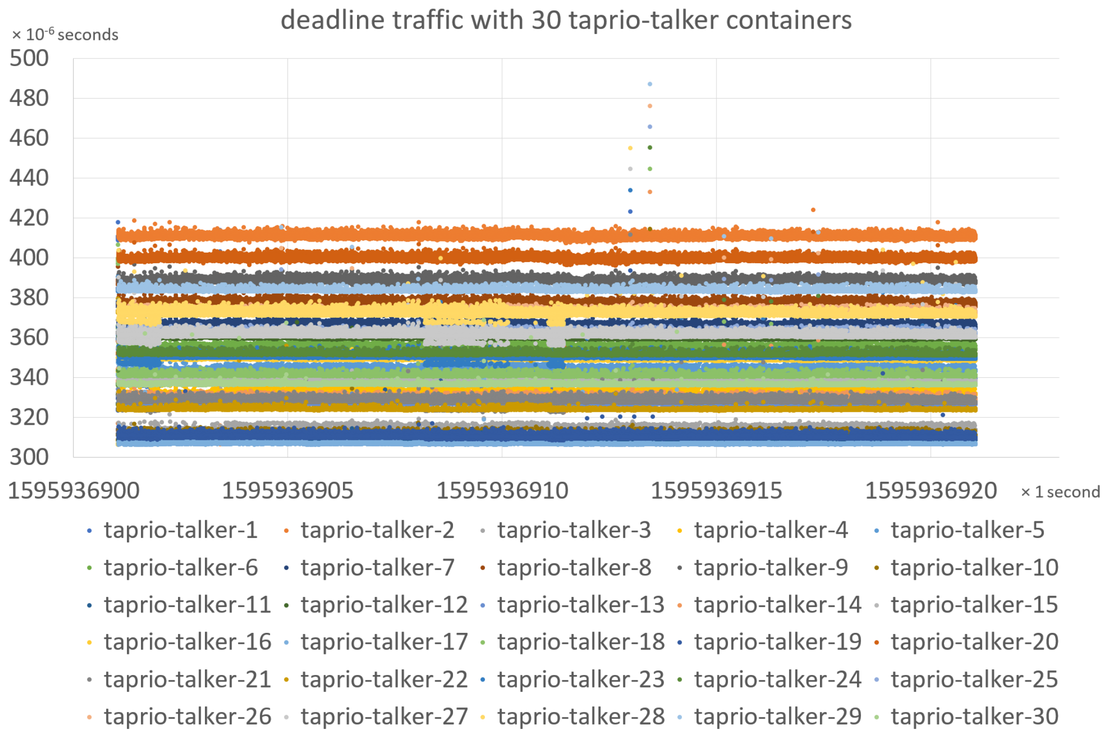
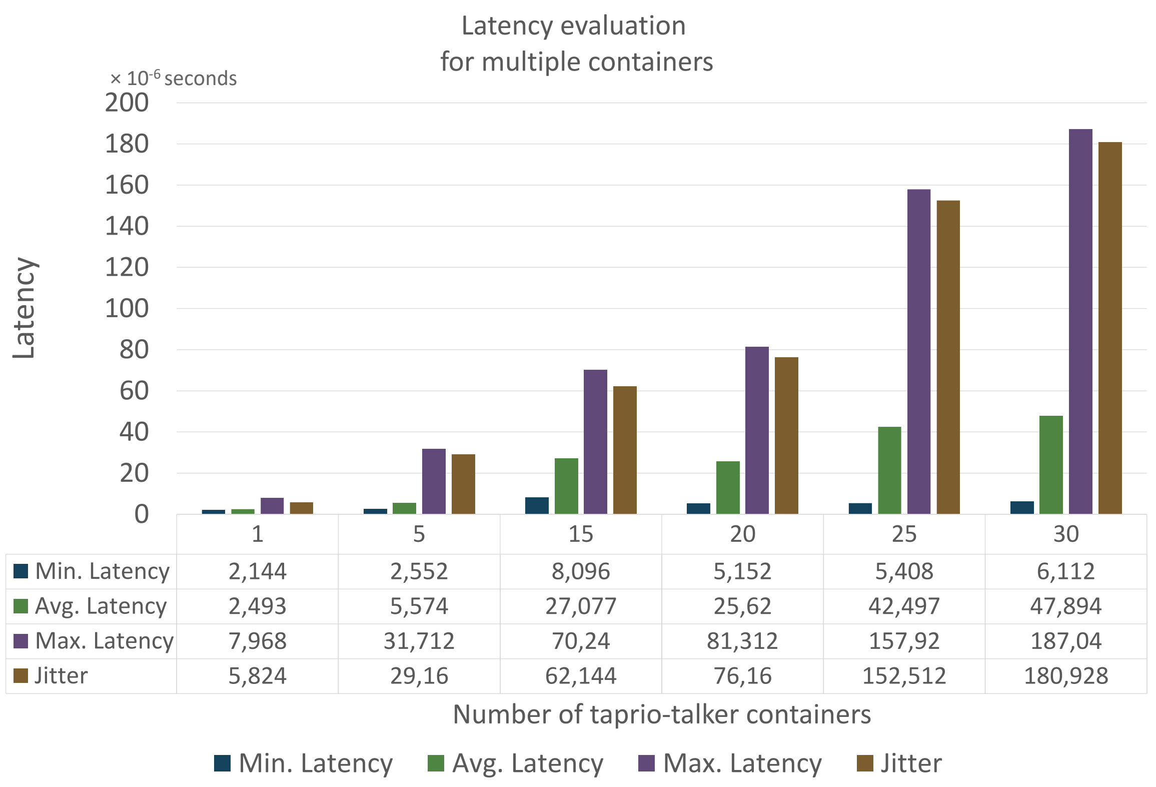


Figure 7.17: Packet latency for 30 taprio-talker containers (at taprio-listener)



	1	5	15	20	25	30
■ Min. Latency	2,144	2,552	8,096	5,152	5,408	6,112
■ Avg. Latency	2,493	5,574	27,077	25,62	42,497	47,894
■ Max. Latency	7,968	31,712	70,24	81,312	157,92	187,04
■ Jitter	5,824	29,16	62,144	76,16	152,512	180,928

Figure 7.18: Packet latency evaluation for multiple taprio-talker containers

Hierarchical Communication In order to evaluate the performance of the hierarchical real-time communication, a setup of two hosts was used as shown in Figure 6.2. The clocks of the network interfaces as well as the clocks of both hosts were synchronized by using linuxptp as described in Section 6.3 (linuxptp was used in version 2.0-00155-g61c6a70 for this experiment). Further, Energy-Efficient Ethernet (EEE) of both Intel® i210 network cards was disabled for the following experiment. EEE has to be disabled for time-critical traffic since it introduces additional latency to the network communication. For example, EEE can be deactivated by using the command line tool *ethtool* (cf. Listing 7.9).

Listing 7.9: Disabling EEE with the command line tool *ethtool*

```
# ethtool --set-eee enp5s0 eee off
```

In order to establish a hierarchical communication between the virtual real-time communication network that is connecting the containers and the physical network which interconnects the two hosts, the network interface of the Intel® i210 network card has to be connected to the Linux bridge of the virtual real-time communication layer on both hosts. This can be achieved by using the *iproute2* tools, as Listing 7.10 shows.

Listing 7.10: Attaching the physical network interface to the bridge

```
# ip link set enp5s0 master lxcbr0
```

In a further step, the TAPRIO queueing discipline and the ETF queueing discipline have to be assigned to the network interface of the Intel® i210 and the network interfaces of the containers. For this experiment, a different schedule than in the previous experiments was used. One reason is that the Precision Time Protocol (PTP) packets have to be sent separately from other network traffic in order to achieve the highest possible precision for the clock synchronization of the network clocks.

In particular, this is achieved by using a dedicated Virtual Local Area Network (VLAN) interface whose traffic is mapped to an independent traffic class. Listing 7.11 and Listing 7.12 show the required commands for setting up the clock synchronization with an independent traffic class via two VLAN interfaces. In particular, Listing 7.11 shows the creation of a VLAN interface that assigns a SKB priority of 7 to all outgoing packets. This priority can afterwards be used in the configuration of the TAPRIO queueing discipline to map the traffic to an independent traffic class. For the presented PTP setup, a Root Mean Square (RMS) value of 5-12ns was achieved in *linuxptp*.

Listing 7.11: Setting up PTP on the taprio-talker host

```
ip link add link enp5s0 name enp5s0.3 type vlan id 3 egress-qos-map 7:7
ip addr add 10.50.3.2/24 brd 10.50.3.255 dev enp5s0
ip addr add 10.50.1.2/24 brd 10.50.1.255 dev enp5s0
ip link set dev enp5s0.3 up
sudo ./setup_clock_sync.sh —i enp5s0.3 —s —v
```

Listing 7.12: Setting up PTP on the taprio-listener host

```
ip link add link enp5s0 name enp5s0.3 type vlan id 3
ip addr add 10.50.3.1/24 brd 10.50.3.255 dev enp5s0
ip link set dev enp5s0.3 up
./setup_clock_sync.sh —i enp5s0.3 —M —v
```

Figure 7.19 shows the hierarchical schedule used for the evaluation of the hierarchical container-to-container communication. As it can be seen, the schedule at the level of the host contains dedicated intervals for the PTP packets.

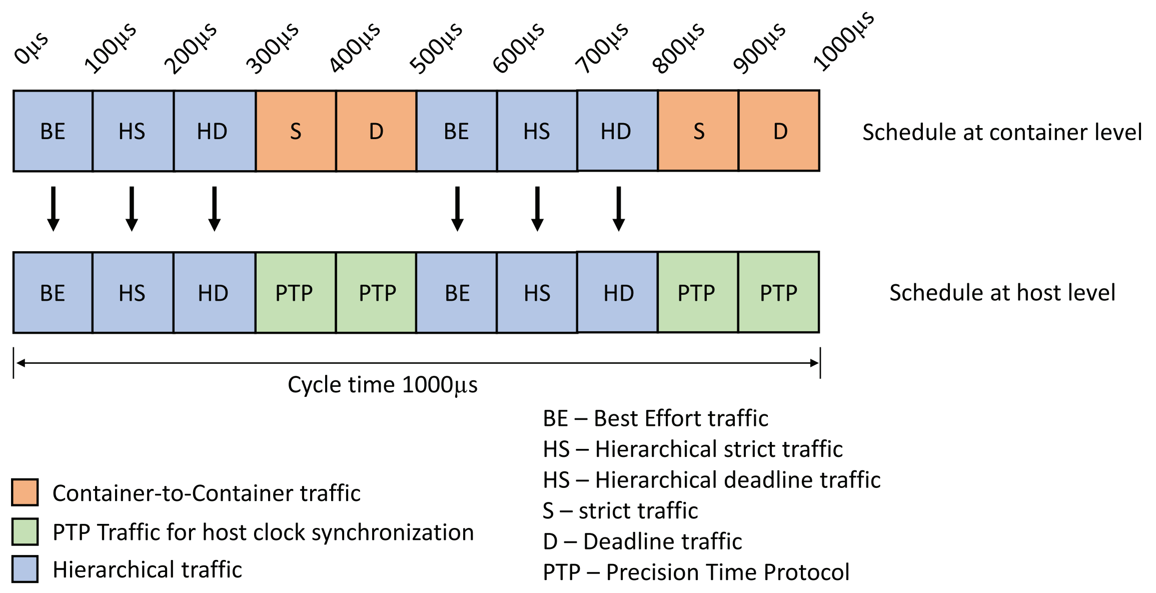


Figure 7.19: Communication schedule for hierarchical traffic

Since the intervals for PTP traffic cannot be used for hierarchical traffic, the schedule at the container level provides here intervals for real-time container-to-container communication for containers that are on the same host. It is important that both schedules are in alignment with each other. Otherwise the TAPRIO queueing discipline would drop the packets due to a closed gate. However, the schedule could be optimized by incorporating the delays in the virtual Ethernet (veth) devices and in the bridge as an offset for the start of the periods. Listing 7.13 shows the corresponding TAPRIO queueing discipline schedule used at the host level.

Listing 7.13: TAPRIO schedule of the taprio-talker host

```
qdisc replace dev eth0 parent root handle 100 taprio \
        num_tc 4 \
        map 3 3 0 1 3 3 3 2 3 3 3 3 3 3 3 3 \
        queues 1@0 1@1 1@2 1@3 \
        base-time 1596379522000000000 \
        sched-entry S 08 100000 \
        sched-entry S 01 100000 \
        sched-entry S 02 100000 \
        sched-entry S 04 200000 \
        sched-entry S 08 100000 \
        sched-entry S 01 100000 \
        sched-entry S 02 100000 \
        sched-entry S 04 200000 \
        clockid CLOCK_TAI

qdisc replace dev eth0 parent 100:1 etf \
        delta 200000 clockid CLOCK_TAI skip_sock_check

qdisc replace dev eth0 parent 100:2 etf \
        delta 200000 clockid CLOCK_TAI skip_sock_check
```

A third step is required for the establishment of the hierarchical communication, which is the (re-)classification of the network traffic. The reason is that the TAPRIO queueing discipline uses the priority field of the SKB structure in the Linux Kernel. In the case of the virtual real-time communication layer, the lifetime of the SKB structure ends as soon as the packet passes the Linux bridge. Therefore, it is necessary to reclassify the packets by setting the priority field of the SKB structure with the according traffic class of the packet. One approach is to use the command line tool *iptables*, which is commonly used to define packet filtering rules in the Linux kernel. Since iptables typically operates on a higher level than the Linux bridge (which is at layer 2), the network traffic on the bridge has to be exposed to iptables by activating the kernel module *br_netfiler*. Listing 7.14 shows the commands used for loading the kernel module and for the classification of both traffic classes *strict* and *deadline*.

Listing 7.14: Classification of network traffic using *iptables*

```
# modprobe br_netfilter
# iptables -t mangle -A POSTROUTING -p udp --dport 7798 -j CLASSIFY --set-class 0:2
# iptables -t mangle -A POSTROUTING -p udp --dport 7788 -j CLASSIFY --set-class 0:3
```

However, setting the priority field of the SKB structure is not sufficient, since the SOCK_TXTIME option in the SKB structure is not set for all incoming packets. This would lead to a packet drop in the ETF queueing discipline. Therefore, the *skip_sock_check* option has to be used when setting up the ETF queueing discipline. This option disables the SOCK_TXTIME check in the ETF queueing discipline. Further, the ETF queueing discipline would still drop all packets, since no transmission timestamp is assigned to the SKB structure. In this context, the default implementation of the ETF queueing discipline has been extended within this experiments to assign the appropriate timestamp to all packets and to dequeued packets directly to the network driver in order to make use of the hardware offloading feature of the Intel® i210, which is called *LaunchTime*. The hardware offloading feature allows the driver of the network card to pre-fetch Ethernet frames from system memory to the transmission buffer inside the Ethernet Media Access Control (MAC) controller ahead of its specified transmission time [Int20]. This allows a very precise injection of packets with low jitter, as Figure 7.20 shows.

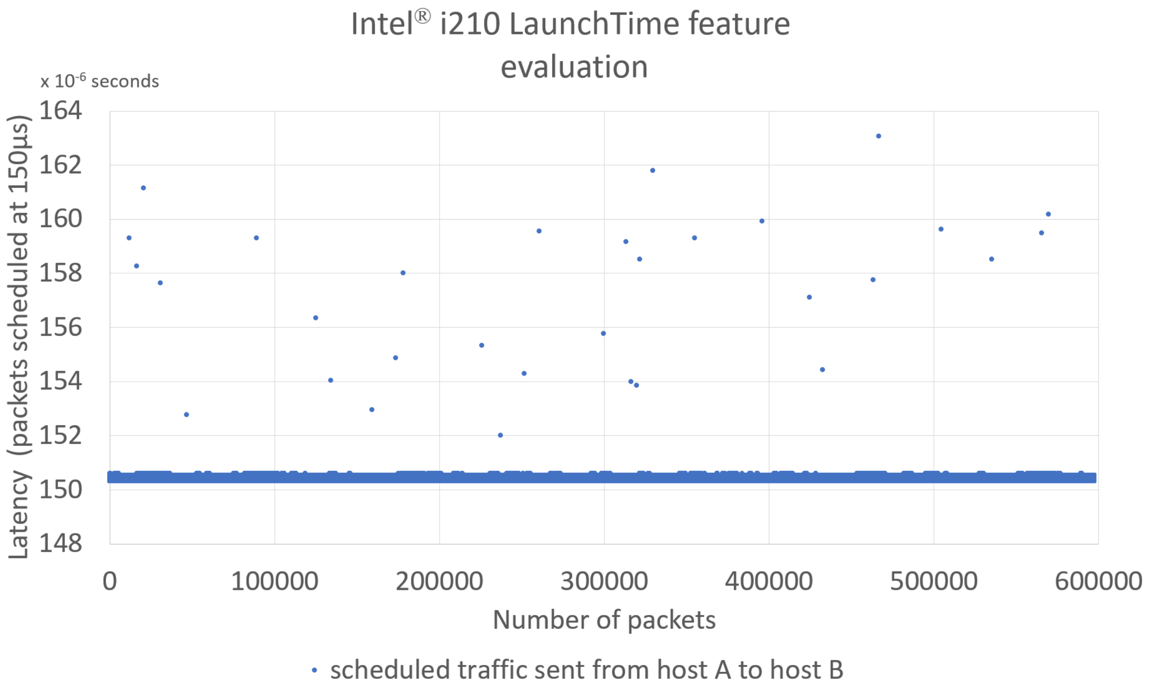


Figure 7.20: Intel® i210 LaunchTime feature evaluation

However, the hardware offloading feature of the Intel® i210 is only supported by the first hardware transmission queue, which is tx-0 (cf. Section 6.2). This has to respected when assigning the traffic classes to the hardware queues in the configuration of the TAPRIO queueing discipline.

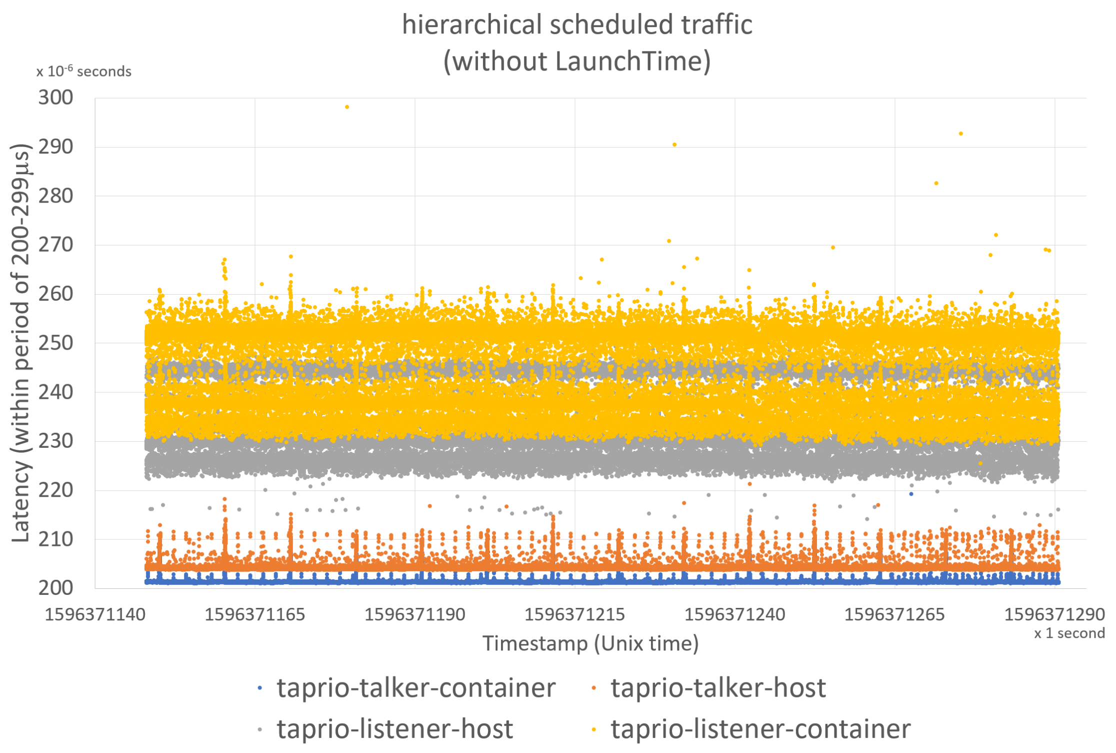


Figure 7.21: Packet latency for hierarchical communication without LaunchTime

Figure 7.21 shows the performance of the hierarchical real-time communication without using the hardware offloading feature of the Intel® i210. For this experiment, the ETF queueing discipline was configured with hardware offloading disabled. It can be seen that the majority of all packets are sent within a band of about 40 μs. However, there is also a small number of packets transmitted outside this band, reaching up to a delay of 100 μs. Another interesting detail that can be seen in Figure 7.21 are the delay peaks that occur at the taprio-talker container and the taprio-talker host about every 10 seconds. The reason for these delay peaks could be narrowed down to the activated *br_netfilter* kernel module. However, it can also be seen that the delay peaks do not have a negative impact to the communication at the taprio-listener host and the taprio-listener container. Due to this reason, no further investigations were made to mitigate the impacts of the *br_netfilter* kernel module and this was left for future work.

Figure 7.22 and Figure 7.23 show the performance of the hierarchical real-time communication using the hardware offloading feature of the Intel® i210. Figure 7.22 shows the results for the hardware offloading with enabled EEE. As it can be seen, the measured jitter is within a band of 20 μs, which is lower than the measured jitter without hardware offloading, but still not optimal. First, with EEE being disabled, the hardware offloading feature of the Intel® i210 reaches its full performance, as Figure 7.23 shows. Now, the jitter at the taprio-listener container remains within a small band of about 10 μs. However, there is also a small number of packets outreaching this band with an delay up to 80 μs.

During the experiments, several configurations for the transmission time used for the hardware offloading feature of the Intel® i210 were evaluated. In Figure 7.22 and Figure 7.23 it can be seen that a transmission time for the hardware offloading was configured to be at 250 μs. The results showed that transmission times for hardware offloading which have a smaller offset than 50 μs from the start of the interval lead to an indeterministic behavior of the hardware offloading feature. In future work, a detailed analysis of this behavior is necessary in order to make smaller offsets possible.

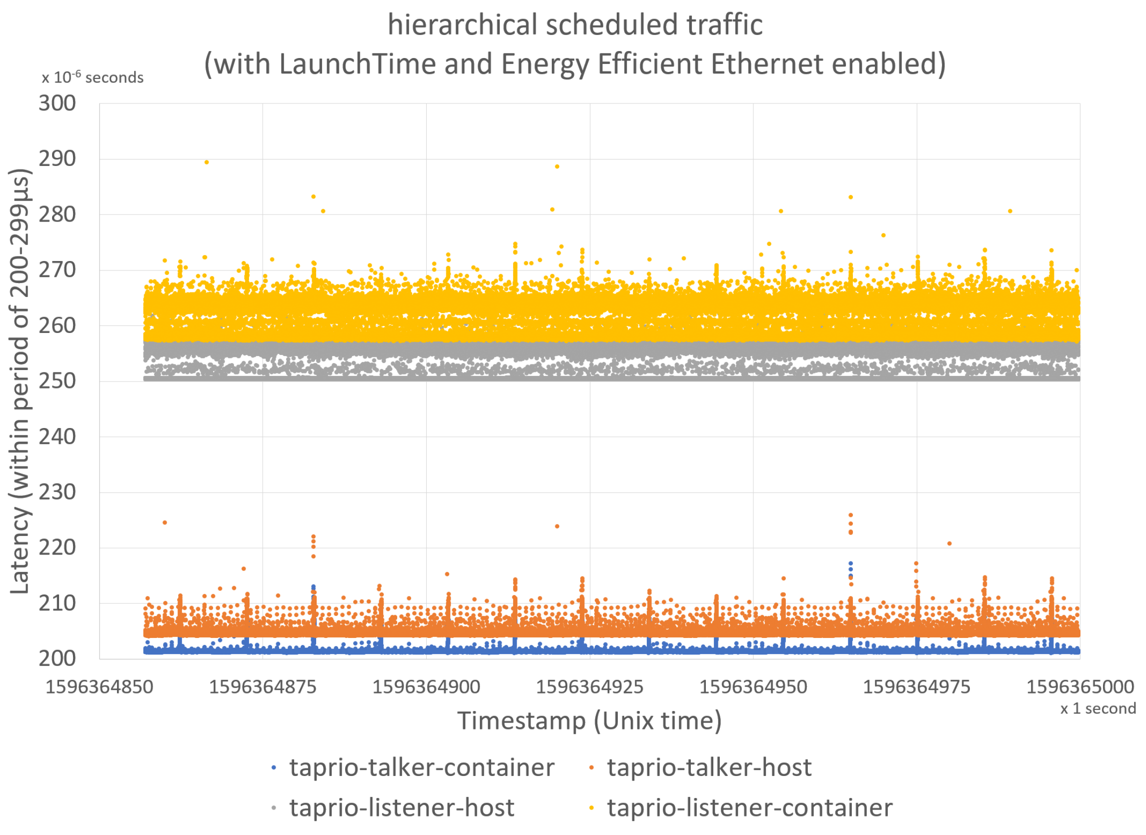


Figure 7.22: Packet latency for hierarchical communication (LaunchTime enabled, EEE enabled)

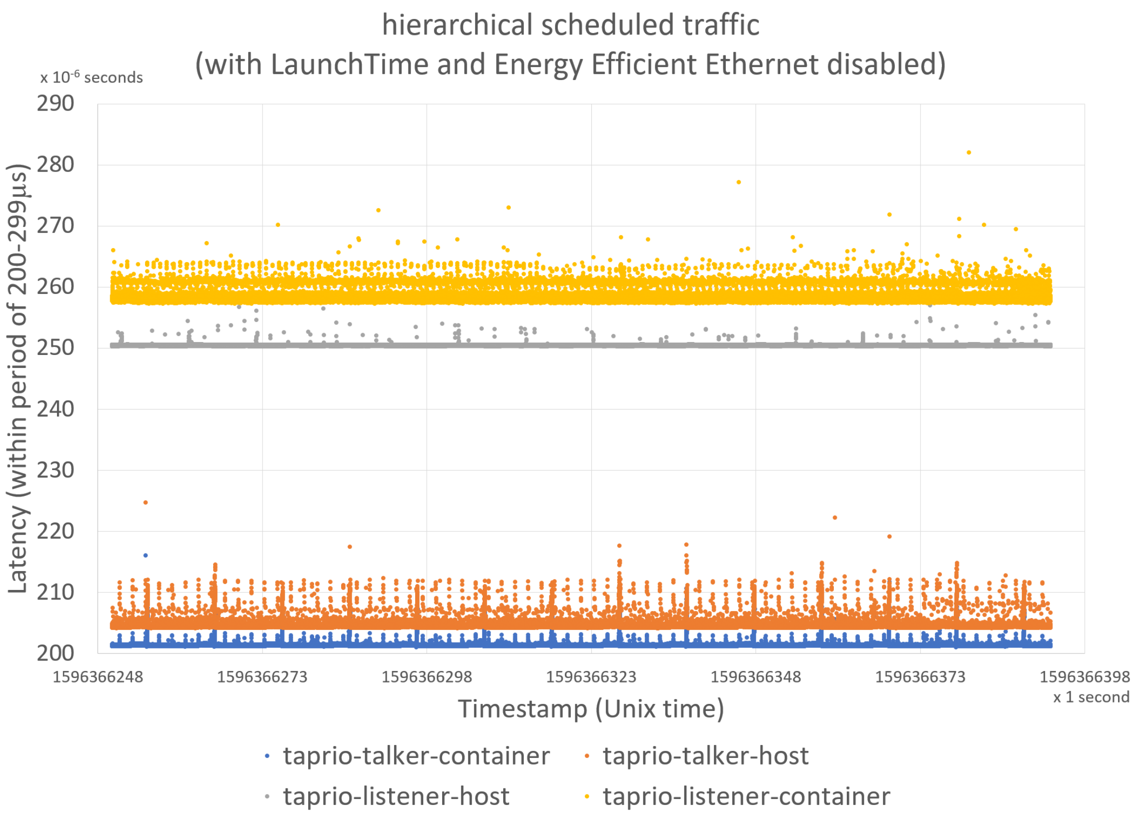


Figure 7.23: Packet latency for hierarchical communication (LaunchTime enabled, EEE disabled)

7.3 Results

The presented experiments have shown that the implementations performed in the context of this thesis are capable of meeting the key challenges presented in Chapter 3. For example, the rate-constrained communication as part of the fault-tolerance techniques allows the establishment of a Bandwidth Allocation Gap (BAG) which in turn allows to isolate failures of nodes within the network (cf. Section 7.2.3). The introduction of the hypervisor LXC in combination with the Linux kernel technologies and further technologies like MemGuard allow the isolation of resources within the service gateway in order to provide a solid basis for mixed-criticality applications. The executed experiments concerning the hierarchical real-time container-to-container communication (cf. Section 7.2.4) also showed clearly that the presented solution is fully suitable for the isolation of network resources as part of the temporal and spatial isolation strategy presented in this thesis. In addition, the achieved performance regarding communication latencies and jitter in the low microsecond range allows the establishment of distributed real-time applications with strict timing requirements. Further, the introduction of SCHED_DEADLINE allows the temporal partitioning of computational resources. In particular, SCHED_DEADLINE was used in the experiments to schedule the real-time task that was responsible for sending the network traffic for the deadline traffic class.

In addition, the experiments for the dynamic reconfiguration of real-time resources showed that the presented architecture is capable to address the open-world assumption as one central part of the presented key challenges. In order to address the heterogeneity of the underlying technologies as part of the key challenges, the presented approach fosters the integration of the medical health and device communication standard ISO/IEEE 11073 as well as the integration of cloud services as part of the ongoing trend for using cloud services for modern health devices. Further, a solution was presented for storing sensor data in order to have access to historical data. This is of interest in elderly care scenarios, since this allows elderly people to reflect their activities or to keep track of their current health status.

Further, the presented results show that scheduled traffic in Linux exhibits deterministic performance in terms of latency and jitter and is suitable for real-time applications that have tight timing constraints in the low microsecond range even when being executed in the hypervisor LXC. Further, it was shown that the scheduling of real-time task with SCHED_DEADLINE task scheduler of Linux is also fitting to the needed real-time constraints. By using the SCHED_DEADLINE task scheduler it is also possible to use a regular Linux kernel for real-time tasks without making the Linux kernel preemptive. It was also shown that a preemptive kernel may even have negative impacts on the real-time performance of the Linux network stack. Further, the experiments showed that the performance of the Linux kernel regarding real-time behavior is further influenced by the powersaving and performance mechanisms of today's CPUs like hyperthreading and c-states of the processor of powersaving features of the network hardware like EEE. Further, it is worth mentioning that the TAPRIO queueing discipline is capable of changing the schedule during runtime. This especially allows a dynamic reconfiguration of the communication resources without any downtime of the network.

However, further experiments seem to be necessary in order to investigate the worse performance of the preemptive Linux kernel. Several opportunities for optimization are provided by the Linux kernel. For example, it is possible to pin the threads of the transmission queues of the *veth* device to dedicated CPU cores (e.g. the core that is running the real-time task). Further, the threads of the transmission queues could be scheduled with SCHED_DEADLINE in order to meet the timing requirements, since the SCHED_DEADLINE scheduler has a higher priority in the Linux kernel than SCHED_RR and SCHED_FIFO which are both used to schedule tasks in the context of a preemptive

Linux kernel. Another option is to implement the TAPRIO queueing discipline in a way that it cannot be preempted by real-time tasks in the user space. Another possible option to evaluate is to use the Power Management (PM) Quality Of Service Interface (PM QoS) of the Linux kernel [Fou20] in the taprio qdisc in order to minimize the wake up latencies of the CPU while processing the schedule. This would for example mitigate the requirement of both kernel flags (*processor.max_cstate=1* and *idle=poll*) used for latency optimization during the experiments.

8 Summary and Conclusions

The demographic change in society is characterized by a shift from traditional care for the elderly to the use of modern technologies from the Cyber Physical Systems (CPSs) sector. This transition also opens up new fields of applications in the domain of elderly care, such as robot assistance or the development of an artificial pancreas [Dan20]. Taking up this trend, this work has identified the key challenges that arise from this transition. These are (1) real-time support, (2) reliability, and (3) support of an open-world assumption, taking into account multiple integration levels and the heterogeneity of the underlying technologies (cf. Chapter 3).

A review of state-of-the-art architectures has shown that there is currently no solution available which addresses all of these challenges. The proposed architecture bridges this gap by introducing a fault-tolerant real-time architecture for elderly care and Ambient Assisted Living (AAL) based on microservices. In particular, the proposed architecture provides support for distributed real-time applications, taking into account an open-world assumption where dynamic changes in the composition of the system are possible at any time. To meet the above mentioned challenges, the presented architecture (cf. Chapter 4) introduces several services (cf. Chapter 5), including solutions for sensor integration based on ISO/IEEE 11073, support for applications with mixed criticality and hierarchical real-time communication. In addition, several proof-of-concept implementations (cf. Chapter 6) and experiments (cf. Chapter 7) were presented, demonstrating that the proposed architecture is capable of meeting the challenges posed by the applications in the field of elderly care and AAL. A major contribution of this thesis is the introduction of a distributed and hierarchical real-time container-to-container communication. Further, new concepts were introduced, such as fault containment among containers for high-critical applications as well as real-time container-to-container communication with latencies and jitter in the low microsecond range using techniques based on standard Linux. In summary, it can be stated, that the presented architecture achieves all key challenges for a fault-tolerant real-time architecture for elderly care.

8.1 Future Work

The conducted experiments have shown that there are still open points left for future work. For example, an important aspect to be examined is the negative impact of real-time applications on the Linux network stack in case of a preemptive Linux Kernel is used. The experiments indicate that it is even possible interference in the network stack is possible when processing packets, which can lead to the drop of packets due to missed deadlines in the Linux packet scheduler. Another field for future work concerns the integration of standards for medical health records like openEHR [EHR20] and HL7 (Health Level 7) [HL20]. This is especially important for an architecture for elderly care since the ongoing trend in the health care sector for telemedicine will also affect the sector of elderly care in the near future. This requires the integration of an architecture for the elderly care in more complex environments including a telematic infrastructure connecting various stakeholders in the health sector.

Scientific Contributions

[DEDAO+18] I. Dhiah El Diehn, S. Alouneh, R. Obermaisser, M. Schmidt *et al.*, "Admission control and resource allocation for distributed services in system-of-systems: Challenges and potential solutions," in *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES).* IEEE, 2018, pp. 1–4.

[SO17] M. Schmidt and R. Obermaisser, "Middleware for the integration of bluetooth le devices based on mqtt and iso/ieee 11073," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE).* IEEE, 2017, pp. 1–4.

[SO18] ——, "Adaptive and technology-independent architecture for fault-tolerant distributed aal solutions," *Computers in biology and medicine*, vol. 95, pp. 236–247, 2018.

[SOW18] M. Schmidt, R. Obermaisser, and C. Wurmbach, "Dynamic resource allocation of switched ethernet networks in embedded real-time systems," in *International Conference on Information Technologies in Biomedicine.* Springer, 2018, pp. 353–364.

[WSR+19] V. Wiese, M. Schmidt, T. Reitz, R. Obermaisser, F. Mahdi, and S. Danush, "System-on-chip platform for safety-relevant structural health monitoring applications," in *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1. IEEE, 2019, pp. 3106–3111.

Bibliography

[ABC19] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[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, 2004.

[AO17] H. Ahmadian and R. Obermaisser, "Temporal partitioning in mixed-criticality nocs using timely blocking," in *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Sep. 2017, pp. 98–105.

[AOS06] H. Ahn, H. Oh, and C. O. Sung, "Towards reliable osgi framework and applications," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 1456–1461. [Online]. Available: https://doi.org/10.1145/1141277.1141617

[APA16] *Apache ZooKeeper™*, The Apache Software Foundation, 2016, https://zookeeper.apache.org/ (accessed February 17, 2020).

[APA20] *Apache Mesos - A distributed systems kernel*, The Apache Software Foundation, 2020, https://mesos.apache.org/ (accessed February 17, 2020).

[ARI09] *664P7-1 Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network, International Standard.* Aeronautical Radio Incorporated, 2009.

[AS611] S. AS6802, "Time-triggered ethernet," *SAE International*, 2011.

[AUT20] *Introduction to JSON Web Tokens*, Auth0, 2020, https://jwt.io/introduction/ (accessed March 10, 2020).

[BD19] A. Burns and R. Davis, "Mixed criticality systems - a review," *Department of Computer Science, University of York, Tech. Rep*, Mar. 2019.

[Ben06] C. Benvenuti, *Understanding Linux network internals.* " O'Reilly Media, Inc.", 2006.

[BK16] C. Bormann and A. Keränen, "The BLE (Bluetooth Low Energy) URI Scheme and Media Types," Internet Engineering Task Force, Internet-Draft draft-bormann-t2trg-ble-uri-00, Jul. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-bormann-t2trg-ble-uri-00

[BLCH19] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19).* Renton, WA: USENIX Association, Jul. 2019, pp. 255–268. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/bai

[Blu09] *HEALTH DEVICE PROFILE Implementation Guidance Whitepaper*, Bluetooth® SIG, 2009.

[BLU14] *GATT REST API Whitepaper*, Bluetooth® SIG, 2014.

[BSP05] U. Brinkschulte, E. Schneider, and F. Picioroaga, "Dynamic real-time reconfiguration in distributed systems: timing issues and solutions," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, May 2005, pp. 174–181.

[CAS20] *PAC4J - The Java security engine to protect all your web applications and web services*, The CAS and pac4j consulting company, 2020, https://www.pac4j.org/ (accessed May 19, 2020).

[Chr20] *Mustache - Logic-less templates*, Chris Wanstrath, 2020, https://mustache.github.io/ (accessed May 19, 2020).

[Clo20a] *etcd - A distributed, reliable key-value store for the most critical data of a distributed system*, Cloud Native Computing Foundation as part of the Linux Foundation, 2020, https://etcd.io/ (accessed March 21, 2020).

[Clo20b] *etcd versus other key-value stores*, Cloud Native Computing Foundation as part of the Linux Foundation, 2020, https://etcd.io/docs/v3.3.12/learning/why/ (accessed March 21, 2020).

[Clo20c] *jetcd - A Java Client for etcd*, Cloud Native Computing Foundation as part of the Linux Foundation, 2020, https://github.com/etcd-io/jetcd (accessed March 21, 2020).

[COA17] S. S. Craciunas, R. S. Oliver, and T. AG, "An overview of scheduling mechanisms for time-sensitive networks," *Proceedings of the Real-time summer school LÉcole dÉté Temps Réel (ETR)*, pp. 1551–3203, 2017.

[COR20a] *GENESYS - Generic Embedded System Platform (Grant agreement ID: 213322)*, CORDIS - The European Commission, 2020, https://cordis.europa.eu/project/id/213322 (accessed March 31, 2020).

[COR20b] *OASIS - Open Architecture for Accessible Services Integration and Standardisation (Grant agreement ID: 215754)*, CORDIS - The European Commission, 2020, https://cordis.europa.eu/project/id/215754 (accessed March 31, 2020).

[COR20c] *PERSONA - Perceptive spaces promoting independent aging (Grant agreement ID: 045459)*, CORDIS - The European Commission, 2020, https://cordis.europa.eu/project/id/045459 (accessed March 31, 2020).

[Dan20] *OpenAPS*, Dana Lewis & the OpenAPS community, 2020, https://openaps.org/ (accessed January 02, 2019).

[DIN06] *DIN EN 60812: Analysetechniken für die Funktionsfähigkeit von Systemen–Verfahren für die Fehlzustandsart-und-auswirkungsanalyse (FMEA)*, 2006.

[DLNW13] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless communications and mobile computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[EBBS11] R. Enns, M. Björklund, A. Bierman, and J. Schönwälder, "Network Configuration Protocol (NETCONF)," RFC 6241, Jun. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6241.txt

[Ecl19] *Eclipse MicroProfile® White Paper*, Eclipse Foundation, Inc., 2019, https://microprofile.io/download/6339/ (accessed January 02, 2020).

[Ecl20] *Eclipse MicroProfile Fault Tolerance*, Eclipse Foundation, Inc., 2020, https://microprofile.io/project/eclipse/microprofile-fault-tolerance (accessed January 02, 2020).

[EHR20] *openEHR - Open industry specifications, models and software for e-health*, openEHR Foundation, 2020, https://www.openehr.org/ (accessed March 10, 2020).

[Eid06] J. C. Eidson, *Measurement, control, and communication using IEEE 1588*. Springer Science & Business Media, 2006.

[Ekl20] *LWN.net - Understanding the new control groups API*, Eklektix, Inc, 2020, https://lwn.net/Articles/679786/ (accessed March 21, 2020).

[Emp15] *E4 wristband*, Empatica Inc., 2015, https://www.empatica.com/e4-wristband (accessed January 14, 2017).

[Eur15] *The 2015 Ageing Report*, European Commission, 2015, http://www.aal-europe.eu/wp-content/uploads/2015/08/Ageing-Report-2015.pdf (accessed January 14, 2017).

[Fou20] T. L. Foundation, *PM Quality Of Service Interface*, 2020, https://www.kernel.org/doc/html/latest/power/pm_qos_interface.html (accessed June 02, 2020).

[FTVF19] N. Finn, P. Thubert, B. Varga, and J. Farkas, "Deterministic Networking Architecture," RFC 8655, Oct. 2019. [Online]. Available: https://rfc-editor.org/rfc/rfc8655.txt

[GB12] E. Geisberger and M. Broy, *agendaCPS: Integrierte Forschungsagenda Cyber-Physical Systems.* Springer-Verlag, 2012, vol. 1.

[GD08] K. Gama and D. Donsez, "Service coroner: A diagnostic tool for locating osgi stale references," in *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, Sep. 2008, pp. 108–115.

[GIJ$^+$03] M. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky *et al.*, "Final version of dsos conceptual model (csda1)," *School of Computing Science Technical Report Series*, 2003.

[GM19] P. Grzesik and D. Mrozek, "Evaluation of key-value stores for distributed locking purposes," in *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis*, S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, Eds. Cham: Springer International Publishing, 2019, pp. 70–81.

[Gro19] E. Grossman, "Deterministic Networking Use Cases," RFC 8578, May 2019. [Online]. Available: https://rfc-editor.org/rfc/rfc8578.txt

[Har12] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6749.txt

[HAR14] S. A. Haque, S. M. Aziz, and M. Rahman, "Review of cyber-physical system in healthcare," *international journal of distributed sensor networks*, vol. 10, no. 4, p. 217415, 2014.

[Has20a] *Consul - A multi-cloud service networking platform to connect and secure services across any runtime platform and public or private cloud.*, HashiCorp, 2020, https://www.hashicorp.com/products/consul/ (accessed January 02, 2020).

[Has20b] *Consul - Secure Service Networking*, HashiCorp, Inc., 2020, https://www.consul.io/ (accessed March 21, 2020).

[HKJR10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851

[HKZ+11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[HL20] *HL7 - Health Level 7 Standards*, Health Level Seven International, 2020, https://www.hl7.org/ (accessed March 10, 2020).

[HMH+11] S. Hanke, C. Mayer, O. Hoeftberger, H. Boos, R. Wichert, M.-R. Tazari, P. Wolf, and F. Furfari, "universaal–an open and consolidated aal platform," in *Ambient assisted living.* Springer, 2011, pp. 127–140.

[HO13] O. Höftberger and R. Obermaisser, "Ontology-based runtime reconfiguration of distributed embedded real-time systems," in *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, June 2013, pp. 1–9.

[HPMS11] R. S. Hall, K. Pauls, S. McCulloch, and D. Savage, "Osgi in action," *Creating Modular Applications in Java*, 2011.

[IEE08a] "Ieee standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pp. 1–300, July 2008.

[IEE08b] "Ieee standard for a precision clock synchronization protocol for networked measurement and control systems - redline," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002) - Redline*, pp. 1–300, July 2008.

[IEE10] "Health informatics–personal health device communication part 10421: Device specialization–peak expiratory flow monitor (peak flow)," *IEEE Std 11073-10421-2010*, pp. 1–63, Nov 2010.

[IEE12] "Health informatics - personal health device communication part 00103: Overview," *IEEE Std 11073-00103-2012*, pp. 1–80, Aug 2012.

[IEE14] "Ieee health informatics–personal health device communication - part 20601: Application profile- optimized exchange protocol," *IEEE Std 11073-20601-2014 (Revision of ISO/IEEE 11073-20601:2010)*, pp. 1–253, Oct 2014.

[IEE16a] "Ieee standard for local and metropolitan area networks – bridges and bridged networks - amendment 25: Enhancements for scheduled traffic," *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)*, pp. 1–57, March 2016.

[IEE16b] *Time Sensitive Networking (TSN)*, IEEE Time-Sensitive Networking Task Group, 2016, http://www.ieee802.org/1/pages/tsn.html (accessed January 14, 2016).

[IEE17] "Ieee standard for local and metropolitan area networks–frame replication and elimination for reliability," *IEEE Std 802.1CB-2017*, pp. 1–102, Oct 2017.

[iet15] "Deterministic Networking," Internet Engineering Task Force, Internet-Draft charter-ietf-detnet-01, Oct. 2015, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/charter-ietf-detnet-01

[Int20] *Intel® - TSN Reference Software for Linux*, Intel® Corporation, 2020, https://github.com/intel/iotg_tsn_ref_sw (accessed March 31, 2020).

[ISO16] "Information technology – message queuing telemetry transport (mqtt) v3.1.1," *ISO/IEC 20922:2016*, 2016.

[JBS15] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7519.txt

[JH12] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," RFC 6750, Oct. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6750.txt

[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 2005.*, May 2005, pp. 22–33.

[KGR91] H. Kopetz, G. Grünsteidl, and J. Reisinger, “Fault-tolerant membership service in a synchronous distributed real-time system,” in *Dependable Computing for Critical Applications.* Springer, 1991, pp. 411–429.

[KLK13] Y. Kopelman, R. J. Lanzafame, and D. Kopelman, “Trends in evolving technologies in the operating room of the future,” *JSLS: Journal of the Society of Laparoendoscopic Surgeons*, 2013.

[KLR97] S. Katz, P. Lincoln, and J. Rushby, “Low-overhead time-triggered group membership,” *Distributed Algorithms*, pp. 155–169, 1997.

[Kop11a] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer Science & Business Media, 2011.

[Kop11b] ——, *Real-Time Systems: Design Principles for Distributed Embedded Applications (Real-Time Systems Series).* Springer, 2011.

[Lev11] N. Leveson, *Engineering a safer world: Systems thinking applied to safety.* MIT press, 2011.

[LHI20] *AccessTSN - Modular open source driver for realtime applications within a TSN based network*, Collaborative effort of Linutronix GmbH, Hirschmann Automation and Control GmbH and the Institute of Reliable Embedded Systems and Communication Electronics at at Offenburg University, 2020, https://www.accesstsn.com/ (accessed March 10, 2020).

[LIN18] *Namespaces*, The Linux Foundation, 2018, http://man7.org/linux/man-pages/man7/namespaces.7.html (accessed January 22, 2019).

[LIN20a] *The Linux Foundation Wiki*, The Linux Foundation, 2020, https://wiki.linuxfoundation.org/realtime/start (accessed January 02, 2019).

[LIN20b] *Linux Manpages - ETF - Earliest TxTime First (ETF) Qdisc*, The Linux Foundation, 2020, http://man7.org/linux/man-pages/man8/tc-etf.8.html (accessed January 02, 2019).

[LIN20c] *Linux Manpages - tc - show / manipulate traffic control settings*, The Linux Foundation, 2020, https://man7.org/linux/man-pages/man8/tc.8.html (accessed January 02, 2019).

[LIN20d] *Linux Manpages - tmpfs - a virtual memory filesystem*, The Linux Foundation, 2020, https://man7.org/linux/man-pages/man5/tmpfs.5.html (accessed January 06, 2020).

[LIN20e] *Linux Programmer's Manual - sched(7)*, The Linux Foundation, 2020, http://man7.org/linux/man-pages/man7/sched.7.html (accessed February 08, 2019).

[LIN20f] *Open vSwitch*, The Linux Foundation, 2020, https://www.openvswitch.org/ (accessed March 21, 2020).

[LIN20g] *taprio: Fix sending packets without dequeueing them*, The Linux Foundation, 2020, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=b09fe70ef520e011ba4a64f4b93f948a8f14717b (accessed March 21, 2020).

[LMA+19] M. Lee, K. Murata, K. Ameyama, H. Yamazoe, and J.-H. Lee, "Development and quantitative assessment of an elbow joint robot for elderly care training," *Intelligent Service Robotics*, vol. 12, no. 4, pp. 277–287, 2019.

[LO17] A. Larrucea Ortube, "Development and certification of dependable mixed-criticality embedded systems," 2017.

[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, Mar. 2011.

[LSAF16] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Softw. Pract. Exper.*, vol. 46, no. 6, pp. 821–839, Jun. 2016. [Online]. Available: https://doi.org/10.1002/spe.2335

[LSG18] P. Lagodzinski, K. Shirahama, and M. Grzegorzek, "Codebook-based electrooculography data analysis towards cognitive activity recognition," *Computers in biology and medicine*, vol. 95, pp. 277–287, 2018.

[MPO08a] "Middleware platform for empowering cognitive disabled and elderly," in *MPOWER Deliverable 1.1 - Overall Architecture*, 2008.

[MPO08b] "Middleware platform for empowering cognitive disabled and elderly," in *MPOWER Deliverable 1.2 - MPOWER Developers Handbook*, 2008.

[MWP+14] M. Memon, S. R. Wagner, C. F. Pedersen, F. H. A. Beevi, and F. O. Hansen, "Ambient assisted living healthcare frameworks, platforms, standards, and quality attributes," *Sensors*, vol. 14, no. 3, pp. 4312–4341,

2014. [Online]. Available: http://www.mdpi.com/1424-8220/14/3/4312

[Net20] *Eureka - AWS Service registry for resilient mid-tier load balancing and failover.*, Netflix, 2020, https://github.com/Netflix/eureka (accessed January 02, 2020).

[NTA+19] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. Elbakoury, “Performance comparison of ieee 802.1 tsn time aware shaper (tas) and asynchronous traffic shaper (ats),” *IEEE Access*, vol. 7, pp. 44 165–44 181, 2019.

[OAOD14] Z. Owda, M. Abuteir, R. Obermaisser, and H. Dakheel, “Predictable and reliable time triggered platform for ambient assisted living,” in *2014 8th International Symposium on Medical Information and Communication Technology (ISMICT).* IEEE, 2014, pp. 1–5.

[OAS14] “Oasis message queuing telemetry transport (mqtt),” *OASIS MQTT v3.1.1*, 2014.

[Obe11] R. Obermaisser, *Time-triggered communication.* CRC Press, 2011.

[OK09] R. Obermaisser and H. Kopetz, *GENESYS - An ARTEMIS Cross-Domain Reference Architecture for Embedded Systems.* Saarbrücken: Südwestdeutscher Verlag für Hochschulschriften, 2009.

[OO14] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[OP06] R. Obermaisser and P. Peti, “A fault hypothesis for integrated architectures,” in *2006 International Workshop on Intelligent Solutions in Embedded Systems*, June 2006, pp. 1–18.

[OSG20] *OSGi Compendium Release 7*, OSGi Alliance, 2020, https://osgi.org/specification/osgi.cmpn/7.0.0/ (accessed March 21, 2020).

[OW14] R. Obermaisser and D. Weber, “Architectures for mixed-criticality systems based on networked multi-core chips,” *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp. 1–10, 2014.

[Per20] *Spark - a tiny web framework*, Per Wendel, 2020, http://sparkjava.com/ (accessed May 19, 2020).

[PZ12] C. Prehofer and M. Zeller, “A hierarchical transaction concept for runtime adaptation in real-time, networked embedded systems,” in *Proceedings of*

2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012), Sept 2012, pp. 1–8.

[RAR07] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-osgi: Distributed applications through software modularization," in *Middleware 2007*, R. Cerqueira and R. H. Campbell, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–20.

[REA08] "make it reaal," in *ReAAL Deliverable 5.3 - Evaluation, validation and evidence report*, 2008.

[REA20] *ReAAL - make it ReAAL (Grant agreement ID: 325189)*, CORDIS - The European Commission, 2020, https://cordis.europa.eu/project/id/325189 (accessed March 31, 2020).

[Ric15] C. Richardson, *Service Discovery in a Microservices Architecture*, Eventuate, Inc., 2015, https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (accessed January 16, 2020).

[Ric18] *The Linux PTP Project*, Richard Cochran, 2018, http://linuxptp.sourceforge.net/ (accessed January 02, 2020).

[RLSS10] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Design automation conference.* IEEE, 2010, pp. 731–736.

[RP05] A. Rasche and A. Poize, "Dynamic reconfiguration of component-based real-time software," in *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Feb 2005, pp. 347–354.

[RTA18] *RTAI - Real Time Application Interface Official Website*, 2018, https://www.rtai.org/ (accessed February 08, 2019).

[RWDD09] T. Richardson, A. J. Wellings, J. A. Dianes, and M. Díaz, "Providing temporal isolation in the osgi framework," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–10. [Online]. Available: https://doi.org/10.1145/1620405.1620407

[SBFH19] S. M. Sulaman, A. Beer, M. Felderer, and M. Höst, "Comparison of the fmea and stpa safety analysis methods–a case study," *Software quality journal*, vol. 27, no. 1, pp. 349–387, 2019.

[Sch16] M. Schmidt, *Bluez 5 Compatibility Patch for the ISO/IEEE 11073 Open-Source stack Antidote*, 2016, https://groups.google.com/d/msg/antidote-lib/scMJ5Bt3rLM/JOpNBQriGQAJ.

[Ser17] A. W. Services, *FreeRTOS reference manual: API functions and configuration options.* Amazon.com, Inc., 2017.

[SG16] K. Shirahama and M. Grzegorzek, *Emotion Recognition Based on Physiological Sensor Data Using Codebook Approach.* Cham: Springer International Publishing, 2016, pp. 27–39. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-39904-1_3

[Sig14] *Antidote, an open-source IEEE 11073-20601 stack*, Signove Tecnologia, 2014.

[SOP08] "Soprano initial architecture," in *SOPRANO Deliverable D2.1.1*, 2008.

[SP20a] J. Sanchez-Palencia, *Make etf report drops on error_queue (patch for the net-next branch)*, 2020, https://patchwork.ozlabs.org/project/netdev/patch/20180703224300.25300-15-jesus.sanchez-palencia@intel.com/ (accessed June 02, 2020).

[SP20b] ——, *Scheduled Tx Tools*, 2020, https://gist.github.com/jeez/bd3afeff081ba64a695008dd8215866f (accessed June 02, 2020).

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[STS$^+$13] M. Sousa, A. Techmer, A. Steinhage, C. Lauterbach, and P. Lukowicz, "Human tracking and identification using a sensitive floor and wearable accelerometers," in *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2013, pp. 166–171.

[TCR09] C. Torrão, N. A. Carvalho, and L. Rodrigues, "Ft-osgi: Fault tolerant extensions to the osgi service platform," in *On the Move to Meaningful Internet Systems: OTM 2009*, R. Meersman, T. Dillon, and P. Herrero, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 653–670.

[TFRF10] M.-R. Tazari, F. Furfari, J.-P. L. Ramos, and E. Ferro, "The persona service platform for aal spaces," in *Handbook of Ambient Intelligence and Smart Environments.* Springer, 2010, pp. 1171–1199.

[TTT17a] *TTE Development Tools*, TTTech, 2017, https://www.tttech.com/products/aerospace/development-test-vv/development-tools/ (accessed December 11, 2017).

[TTT17b] *TTE End System A664 Lab*, TTTech, 2017, https://www.tttech.com/products/aerospace/development-test-vv/development-end-systems/tte-end-system-a664-lab/ (accessed December 11, 2017).

[TTT17c] *TTE Switch A664 Lab*, TTTech, 2017, https://www.tttech.com/products/aerospace/development-test-vv/development-switches/tte-switch-a664-lab/ (accessed December 11, 2017).

[Uni17a] *open.DASH Dashboard System*, Universität Siegen, 2017, https://opendash.de (accessed January 14, 2017).

[Uni17b] *SmartLive: Sustainable and innovative Smart Home/Smart Energy solutions in living labs*, Universität Siegen, 2017, https://smart-live.info (accessed January 14, 2017).

[UNI20a] *UNIVERsal open platform and reference Specification for Ambient Assisted Living (Grant agreement ID: 247950)*, CORDIS - The European Commission, 2020, https://cordis.europa.eu/project/id/247950 (accessed March 31, 2020).

[UNI20b] *universAAL IoT - Semantic interoperability for rapid integration & deployment*, universAAL IoT, 2020, https://github.com/universAAL/ (accessed March 31, 2020).

[VCTMHO10] J. Vila-Carbo, J. Tur-Massanet, and E. Hernandez-Orallo, "Analysis of switched ethernet for real-time transmission," in *Factory Automation*, J. Silvestre-Blanes, Ed. Rijeka: InTech, 2010, ch. 11. [Online]. Available: http://dx.doi.org/10.5772/9505

[Wan17] K. Wang, *Embedded and Real-Time Operating Systems.* Springer International Publishing, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-51517-5

[WG08] M. D. WG, *Multi-Channel Adaptation Protocol (MCAP).* Bluetooth SIG, 2008.

[WG09] ——, *HEALTH DEVICE PROFILE - Implementation Guidance Whitepaper.* Bluetooth SIG, 2009.

[WG12] ——, *HEALTH DEVICE PROFILE - Specification (Revision 11).* Bluetooth SIG, 2012.

[WG15] ——, *Personal Health Devices Transcoding.* Bluetooth SIG, 2015.

[Wit20] *Withings API developer documentation*, Withings France SA, 2020, https://developer.withings.com/oauth2 (accessed March 31, 2020).

[WSK08] P. Wolf, A. Schmidt, and M. Klein, "Soprano-an extensible, open aal platform for elderly people based on semantical contracts," in *3rd Workshop on Artificial Intelligence Techniques for Ambient Intelligence (AITAmI'08), 18th European Conference on Artificial Intelligence (ECAI 08), Patras, Greece.* Citeseer, 2008.

[WSM09] S. Walderhaug, E. Stav, and M. Mikalsen, "Experiences from model-driven development of homecare services: Uml profiles and domain models," in *Models in Software Engineering*, M. R. V. Chaudron, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 199–212.

[XEN19] *The Xenomai Project*, 2019, https://xenomai.org/ (accessed January 02, 2019).

[Yag08] K. Yaghmour, *Building Embedded Linux Systems*, 2nd ed. O'Reilly Media, Inc., 2008.

[YYP$^+$13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 55–64.

[ZBRY19] Z. Zhou, M. S. Berger, S. R. Ruepp, and Y. Yan, "Insight into the ieee 802.1 qcr asynchronous traffic shaping in time sensitive network," *Advances in Science, Technology and Engineering Systems Journal*, vol. 4, no. 1, pp. 292–301, 2019.

[ZHLL18] L. Zhao, F. He, E. Li, and J. Lu, "Comparison of time sensitive networking (tsn) and ttethernet," in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, Sep. 2018, pp. 1–7.

[ZMN05] F. Zhu, M. W. Mutka, and L. M. Ni, "Service discovery in pervasive computing environments," *IEEE Pervasive Computing*, vol. 4, no. 4, pp. 81–90, Oct 2005.