TU WIEN Informatics

Virtualization of Internet of Things Devices

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Matthias Leitner, BSc
Matrikelnummer 01327110

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Associate Prof. Matthew Caesar, Ph.D.

Wien, 23. April 2020

Matthias Leitner   Stefan Schulte

Informatics

Virtualization of Internet of Things Devices

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Matthias Leitner, BSc
Registration Number 01327110

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr.-Ing. Stefan Schulte
Assistance: Associate Prof. Matthew Caesar, Ph.D.

Vienna, 23rd April, 2020

Matthias Leitner   Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Matthias Leitner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. April 2020

Matthias Leitner

Acknowledgements

I would first like to thank my thesis advisor Associate Professor Stefan Schulte at the TU Wien for the useful comments, remarks, and engagement through the learning process of this master thesis. Also, his assistance with administrative tasks for my research internship at the University of Illinois at Urbana-Champaign (UIUC) was highly valuable and saved me lots of time. Further, I would like to thank Professor Reinhard Pichler and the Dean's office at TU Wien for their letter of recommendation and for supporting my stay in the US.

I would also like to thank Associate Professor Matthew Caesar at the UIUC, who offered me an exciting topic for my master thesis and provided me the opportunity to visit the UIUC for another semester. The door to his office was always open when I had a question about my research or needed some advice. Also, many thanks to Andrew Li, a member of the research team at the UIUC, for his valuable feedback and the interesting discussions.

Last but not least, I would like to express my gratitude to my family, friends, and colleagues who supported me throughout my studies and always encouraged me in challenging times.

Kurzfassung

Das Internet der Dinge (engl. Internet of Things, IoT) beschreibt die Vernetzung intelligenter Geräte die mit ihrer Umwelt interagieren. Durch die Vielzahl an Anwendungsmöglichkeiten beeinflusst das IoT weite Bereiche der Industrie sowie des alltäglichen Lebens. Die Integration neuer Technologien und die stärker werdende Vernetzung führen zu einer erhöhten Komplexität solcher Systeme.

Die Heterogenität und die Größenordnung von IoT-Netzwerken stellen Forscher und Entwickler vor neue Herausforderungen. Die im traditionellen Software Engineering verwendeten Methoden bieten nicht genügend Möglichkeiten, umfangreiche Tests während des Entwicklungsprozesses von IoT-Systemen durchzuführen. Aus diesem Grund werden neue Lösungen benötigt, welche die Funktionalität der entwickelten Systeme gewährleistet, bevor diese in der realen Welt eingesetzt werden. Eine Möglichkeit, um der Komplexität von IoT-Systemen entgegenzuwirken, bietet der Einsatz von Simulatoren.

Aus diesem Grund befasst sich die vorliegende Arbeit mit der Entwicklung eines neuartigen Simulationsansatzes für IoT-Systeme. Basierend auf dem Entwicklungsprozess von IoT-Systemen werden aktuell verwendete Methodiken analysiert und simulationsbasierte Lösungen mit Alternativen verglichen. Diese Ergebnisse fließen in eine Anforderungsanalyse ein und werden zusätzlich um Erkenntnisse der aktuellen Forschung sowie eines beispielhaften Anwendungsszenarios ergänzt.

Das daraus resultierende Architektur-Framework ermöglicht gemeinsam mit dem entwickelten Netzwerksimulator eine gesamtheitliche Simulation von IoT-Systemen. Das Architektur-Framework repräsentiert eine generische Software-Integrationsschicht, die mittels Virtualisierungstechnologien und Hardware-Emulation virtuelle IoT Geräte in die Simulation einbindet. Der Netzwerksimulator verbindet die virtuellen IoT Geräte und simuliert realitätsnahe Kommunikationskanäle, welche flexibel konfiguriert werden können.

Abschließend wird mittels eines Use-Case-Szenarios die Verwendung der entwickelten Systeme veranschaulicht sowie die Qualität der Simulationsergebnisse analysiert. Dazu wird der Raft Algorithmus implementiert und dessen Performance in einem IoT-Netzwerk evaluiert.

Abstract

The Internet of Things (IoT) influences a wide range of domains and already impacts our day-to-day lives. IoT systems represent networks of interconnected, smart devices that interact with their environment. With emerging technologies entering the market, IoT systems become more sophisticated. Whether it is reducing costs and increasing efficiency in production lines, driving automation and autonomy, or continuously monitoring people's health—the IoT affects consumer-oriented applications as well as safety-critical infrastructure.

The heterogeneity of these networks and their scale impose new challenges on researchers and developers. Tools used in traditional software engineering do not provide sufficient capabilities to evaluate these systems extensively during the development process. Hence, novel approaches and tools are needed that assist researchers in testing and verifying their proposed IoT solutions before deploying them in the real world. Simulating IoT systems upfront in a controlled environment is one possibility to address the increased complexity.

Therefore, this thesis proposes a novel end-to-end simulation approach for the development of IoT systems. We analyze existing tools currently used in IoT research projects and compare simulation-based methods to alternative solutions. Further, we define essential requirements and characteristics for IoT simulation systems according to state-of-the-art research, and an exemplary use-case scenario.

Based on these findings, we propose an architecture framework and a network simulator. The architecture framework is a generic software integration layer that utilizes existing device emulators and simulators and supports hardware emulation. The network simulator imitates real-world networks and connects virtual devices in the simulation.

The work concludes with a comprehensive evaluation. For this, we implement the Raft algorithm and demonstrate the validity of the simulation results as well as the capabilities of the proposed solutions by evaluating the performance of Raft in a simulated IoT network.

Contents

CHAPTER 1

Introduction

The term "Internet of Things" (IoT) describes a network where numerous different objects are connected to the Internet, and provide data storage and data processing capabilities. These things can be sensors, vehicles, cameras, or medical instruments and are already part of our daily lives [1, 2].

Building solutions for the IoT comes with new challenges [3]. The variety of components in IoT systems and the dynamic changes of IoT devices increases the complexity of application development [4]. In addition, a significant amount of IoT devices have very limited resources like CPU, RAM, storage, etc. [1]. Although some of these challenges are not necessarily new in distributed systems [4], the IoT raises new issues for developing and testing applications [3, 5]. To name a few, these are the number of different protocols used in the IoT [6], the support of different variants of a device [3], and the massive number of devices (thousands to millions) which are part of an IoT system [7].

Nonetheless, researchers and developers have to develop, test, and verify their IoT applications. One possible way is to acquire the required hardware and deploy the devices in the real world. This may lead to high costs, and especially small and medium-sized enterprises, researchers as well as the open-source community often do not possess the according funds to set up extensive testbeds [5]. The alternative approach is to lease IoT testbeds. Although this allows testing on physical hardware without the need to maintain the hardware infrastructure, this approach comes with some drawbacks. Most importantly, these testbeds are still expensive, and they are difficult to maintain and complex to use. As new IoT devices are developed continuously, the hardware of testbeds is often outdated, and the heterogeneity of these systems is lower compared to IoT systems in the real world [5].

Therefore, a more cost-effective solution with increased flexibility is needed. One way to achieve this is the utilization of an IoT simulation platform. This allows researchers and developers to set up their IoT systems with little effort and facilitates exploring, as changes can be made quickly. A simulation platform can be used to test the application code of IoT devices, to evaluate the system architecture of IoT setups and to run real-world scenarios in a controlled environment [8]. Although a simulation platform has its drawbacks, e.g., real devices are different compared to simulations and writing an emulator that mimics every existing IoT device is very hard to achieve [5], simulation platforms provide several advantages compared to the previously mentioned solutions. A simulation platform is accessible from nearly anywhere and is much more cost-effective compared to, e.g., real-world testbeds. Besides, simulation platforms allow running tests with different system architectures and devices without the need to buy or change devices in the real world [9].

1.1 Simulation for the IoT

The IoT includes a wide range of applications and covers many different areas [1, 10]. Hence, this diversity has implications on the development and testing process of IoT systems. In the following use-cases, simulations are either beneficial during development or are safety-critical and required before the system can be tested in the real world. In addition, testing these systems in the real world can be a difficult task.

- **Smart Territories** Recently, research started on expanding the concept of smart cities into non-metropolitan areas [11]. This requires IoT services to cover wide areas while being cost-effective. Such large-scale applications highly benefit from simulations, but they impose new challenges on simulation software due to their scale [12].

- **Unmanned Aerial Vehicles** Due to the reduced costs of unmanned aerial vehicles (UAV), UAV swarms gain popularity [13]. Although UAVs are not necessarily autonomous, inter-vehicle communication is essential to control these systems and to allow swarms to coordinate themselves in case of communication limits. Simulations are useful to test algorithms and scenarios before deploying them on physical UAVs in the real world [13].

- **Smart Grids** The utilization of IoT devices transforms traditional power systems into so-called smart grids. This leads to the integration of IoT applications in critical infrastructure. Besides testing the correctness of applications and devices, the simulation of errors is important for smart grids, to assess failure propagation in the system [14].

1.2 Problem Statement

Despite the advantages of IoT simulators [8], they introduce qualitative and quantitative issues [12]. The heterogeneity of IoT systems imposes a challenge on simulators. Similar to testbeds, simulation tools must be able to handle the variety of hardware components and keep up with the velocity of new devices entering the market [5, 10, 12].

In addition, IoT simulation introduces its own challenges. IoT systems in the real world involve a vast number of sensors, actuators, and other computing devices, requiring scalable simulators to support distributed or parallel operation modes [10, 12]. Further, current simulators do not support end-to-end simulations of IoT systems, i.e., to simulate a complete system, multiple simulators are needed. This increases the complexity as the number of available simulators and contradicting recommendations require additional effort [15]. The heterogeneity of IoT creates the need for extendable simulators and to facilitate customization [10]. Thus, providing more flexibility to the users.

Although these requirements are important characteristics for a simulation system, the applicability of such a system highly relies on the quality of the simulation result. The quality of a simulation result depends on the simulation or emulation of IoT devices and the correctness of the simulated network [5, 10].

To the best of our knowledge, this area is actively researched [5, 15, 16, 17, 18], but a solution, which solves all of the issues mentioned above, has not been found yet.

1.3 Aim of the Work

Prof. Matthew Caesar is working on an IoT simulation platform at the University of Illinois at Urbana-Champaign (UIUC). The thesis is part of the research to develop a flexible end-to-end simulation platform for the IoT. The research focus of this thesis is on the virtualization of IoT devices and to provide solutions to the challenges discussed in Section 1.2. The results contribute to future developments of the simulation platform.

The research focus of the thesis consists of three different parts. The following paragraphs define the objectives for each part and provide additional information about their scope.

1.3.1 Part 1: Architecture Framework

The architecture framework addresses the issues with the heterogeneity of IoT devices, their variety, and the need for customization and extensibility.

The architecture framework can be seen as a toolset, which allows the integration of existing IoT device simulators and emulators. The idea is to focus on customization and extensibility, allowing users to integrate their virtual IoT devices and using them on the simulation platform. Hence, users can contribute virtual IoT devices to a community around the platform.

The objective of the framework is to provide a simple integration process without imposing changes on virtual IoT devices. This approach increases the compatibility with existing IoT device simulators and emulators, and by keeping the integration process simple, virtual IoT devices are more likely to get added to the platform. Hence, these characteristics should facilitate community building and further solve the challenges with heterogeneity and variety of IoT devices.

The simulation of real-world IoT scenarios requires a scalable simulation platform [10]. Hence, it must be possible to use the framework in distributed simulations.

Due to the variety of IoT devices, it would go beyond the scope of a Master thesis to consider every possible type of IoT device. Hence, this Master thesis will focus on particular categories of IoT devices. A possible categorization of such devices is based on their functionality in an IoT network. More specifically, the scope of this thesis is limited to IoT devices which can be grouped into the following categories [6]:

1. Sensing (Smart devices): These devices are typically deployed in the environment and measure it or interact with it, for example, a smart sensor. These devices have very limited resources, and besides simple data preprocessing tasks, they forward data to an IoT cloud, a gateway, or other parts of the system [19].

2. Computation (Controllers): In this thesis, devices within this category will be referenced as controllers or edge devices. These are devices with enough resources to run IoT gateways and are the basis for simple edge nodes [6].

1.3.2 Part 2: Network Simulator

One of the objectives for the simulation platform at the UIUC is to enable end-to-end simulations. While Part 1 focuses on IoT devices, the second part of the thesis addresses the simulation of the network layer.

The simulation platform provides a web-based user interface to set up simulations. To support this flexibility, it requires the network simulator to create and simulate network topologies at runtime.

The network simulator developed in this thesis is based on ns-3 to simulate communication channels. The network simulator has to extend ns-3 to provide the required flexibility for network topologies and needs to simplify the setup and deployment process of ns-3.

Both the architecture framework and the network simulator, are independent components and need to communicate through well-defined interfaces. This is important, as the research team at the UIUC considers implementing a new network simulator in the future.

1.3.3 Part 3: Use-Case Simulation

With the architecture framework and the network simulator, it is already possible to run end-to-end IoT simulations. Hence, the third part of the thesis uses the functionality of both components by simulating an IoT scenario. This scenario evaluates the Raft [20] consensus algorithm running on a wireless IoT setup.

Although the importance of consensus in datacenter and cloud environments is well-known [21], it is important in IoT systems as well. Considering the use-cases provided in Section 1.1, consensus algorithms have applications in every scenario. Smart territories are wide area networks connecting vast amounts of IoT devices [12]. Developing applications for these networks may require consensus algorithms, as they are important for building reliable large-scale software systems [20]. Further, UAV swarms use consensus algorithms to reliably exchange information to facilitate swarm behavior coordination [13]. In the third use-case, smart grids become more distributed in the future, and the existing centralized control structure is not sufficient for operating highly distributed systems. Hence, the research examines consensus algorithms in smart grid applications [22].

Following the relevance of consensus algorithms in the IoT, Poirot et al. researched the Paxos algorithm in wireless networks [23]. Raft's relation to Paxos and the relevance of consensus algorithms in the IoT makes Raft a good candidate for the simulation scenario.

The simulation scenario consists of the following steps. First, a simple Raspberry Pi[1] emulator gets developed. The Raspberry Pi is a small yet powerful single-board computer (SBC). Current research shows that SBCs like the Raspberry Pi gain popularity in the IoT and are also used in SBC clusters as edge nodes [24]. Thus, the emulator represents an edge node in the IoT simulation. The architecture framework then integrates the emulator and demonstrates the framework's capabilities. This provides an IoT device for the simulation. In the next step, the network simulator creates a Mesh topology to connect the simulated IoT devices. The scenario concludes with the implementation of Raft, its deployment on every edge node and the evaluation of the algorithm.

1.4 Methodology and Approach

The methodological approach for this thesis follows the objectives in Section 1.3. It follows the following order:

1. **Literature Research**: Due to the interest in simulation for the IoT, this area is actively researched. In the first part of the thesis, state-of-the-art and past approaches are reviewed.

2. **Architecture Framework**: Based on the obtained knowledge during the literature research, the architecture framework is built. The result of this step is a solution

[1]`https://www.raspberrypi.org/`, last access at 2020-03-15

for virtualizing IoT devices. The result definition of the architecture framework is general at this point, as a requirements analysis is part of this step.

3. **Network Simulator**: To build the network simulator, it is necessary to explore the real-time capabilities of ns-3, executing tests, and defining protocols and modules supported by the network simulator. Further, additional research is required to support the creation of network topologies dynamically.

4. **Simulation Evaluation**: Before continuing with the use-case implementation, it is necessary to evaluate the simulation capabilities of the architecture framework and the network simulator. The framework gets evaluated based on the requirements analysis from step 2. Further, an end-to-end simulation setup, including the architecture framework and the network simulator, tests scenarios where both components have to collaborate.

5. **Use-Case Simulation**: The Raspberry Pi emulator requires additional research in cross-platform virtualization. After evaluating possible solutions and developing the emulator, the Raft algorithm gets implemented.

6. **Evaluation**: The final step summarizes the simulation capabilities of the developed system and analyzes its characteristics. It concludes with the evaluation of the Raft algorithm running on a simulated, wireless, mesh network.

1.5 Structure of the Thesis

The architecture framework, the network simulator, as well as the use-case scenario, utilize existing tools and technologies to enhance their functionality and interoperability. Chapter 2 provides background information about the most important concepts, technologies, software tools, and frameworks used in this thesis. The goal of this thesis is to propose viable solutions to current research challenges by utilizing existing work to improve its results. Hence, Chapter 3 covers related work and summarizes current research. Based on this knowledge, Chapter 4 proposes the design of the architecture framework, the network simulator, and the use-case implementation. Following the design of the system, Chapter 5 provides information about implementation details and explains how the system can be used to execute IoT simulations. In the last part of the thesis, Chapter 6 evaluates the architecture framework and the use-case scenario, discusses the benefits and limitations of the proposed system, and provides an outlook and possible future developments in Chapter 7.

CHAPTER 2

Background Information

The following chapter covers essential concepts and relevant information for this thesis. Section 2.1 introduces the IoT and summarizes its characteristics and potential applications. Further, Section 2.2 and Section 2.3 discuss communication in the IoT and virtualization in more detail, respectively, as both areas are relevant to the proposed solution. The chapter concludes with Section 2.4 describing important tools, technologies, and frameworks used by the designed system.

2.1 The Internet of Things

Over the last few years, the IoT became increasingly popular in academia and the industry [25]. Besides universities actively researching this area, major companies like Microsoft, IBM, Google, Samsung, and many more, cooperate to drive innovation in the IoT [26]. Although outlooks from companies like Gartner Inc., RnRMarketResearch, or Cisco vary, all of them predict significant growth of the IoT market and its applications [25, 26].

The term "Internet of Things" has its origins in the field of supply-chain management [26]. Kevin Ashton, the executive director at MIT's Auto-ID Center at that time, used this term during a presentation at Proctor & Gamble in 1999. He linked radio-frequency identification (RFID) in the supply chain to the Internet and believed in the potential of this new concept [25, 26]. The International Telecommunication Union (ITU) introduced the IoT formally in their *Internet Report* in 2005 [25].

Since then, the IoT evolved, and many definitions have been presented [26]. Chernyshev et al. describe the vision of the IoT "to build a smart environment by utilizing smart things/objects/devices that have sensory and communication capability to autonomously generate data and transmit it via the Internet for decision making" [15]. Similarly, Khodadadi et al. emphasizes the context awareness of the "Things" and describes ubiquitous connectivity as a central component [26]. Firouzi et al. examined current definitions and derived the following characteristics [27]:

- **Things or Devices** These are objects with processing capabilities, sensing and actuating features, connectivity, and can be uniquely identified [26, 27].
- **Connectivity** Network connectivity is an important factor to make things “smart“ [5]. Small sensors as well as resourceful backend servers have to support a variety of communication protocols and standards [26].
- **Data** IoT devices create a vast amount of data that needs to be collected and analyzed [25]. It further allows service improvements and is the key to extract knowledge for decision making [25, 27].
- **Intelligence** Artificial intelligence, machine learning, and data analytics utilize IoT data to gain insights into the system or its environment and to provide valuable services [26, 27]
- **Action** Intelligent systems allow automated interactions with the environment or themselves, but also includes humans [25, 27].
- **Ecosystem** The IoT forms an ecosystem consisting of devices, protocols, platforms, data, and communities [26, 27].
- **Heterogeneity** IoT systems typically consist of a variety of different devices. They utilize different protocols to exchange data and integrate with various platforms [5, 12, 27].
- **Dynamic Changes** In IoT networks, devices are constantly joining and leaving, data transmission varies and also the environment may change dynamically [4, 27].
- **Enormous Scale** IoT environments like smart cities consist of large numbers of devices [12], increasing the amount of generated data and connected devices [27].
- **Security and Privacy** As the IoT becomes part of our daily lives, it will affect sensitive areas like healthcare. This requires secure systems to protect individuals. In addition, data sovereignty becomes more important as more personal data will be available online [25, 26, 27, 28].

The IoT developed rapidly since its first mention in 1999 and spread into many different areas [26]. Although there is no general agreement on IoT verticals [25], the *Alliance for Internet of Things Innovation*[1] (AIOTI) provides an overview of the IoT landscape. AIOTI collected information about the main IoT Standards Developing Organisations (SDO), Alliances, and Open Source Softwares (OSS), and categorized them into IoT verticals [29].

[1]`https://aioti.eu`, last access at 2020-03-22

Further, the *European Telecommunications Standards Institute*[2] (ETSI) analyzed the degree of industry and market fragmentation based on these IoT verticals and provides a more detailed description [30]. We use these vertical industry domains to present an overview of IoT applications and how they affect their domain.

- **Home/Building** In this domain, home or building refers to any house, including private and commercial ones [31, 32]. Smart buildings enable remote access to the systems of the house. This allows owners and other stakeholders to monitor and control house functionalities like security or lighting from distance [31]. Despite user-related features, technology for smart building enables energy management and environmental monitoring. Smart buildings optimize energy consumption within the house with regards to electricity, natural gas (if applicable), renewable energy, and other types of energy sources [32].
- **Manufacturing/Industry Automation** Besides cloud computing, service-oriented computing, artificial intelligence, and data science, the IoT is one of the key enablers for smart manufacturing [33]. Also known as Industry 4.0 [33], it describes the concept of utilizing IoT technologies like sensors and communication networks to collect data, automate processes, and improve the production process [33, 34]. Smart manufacturing goes beyond single manufacturing plants, and it is rather a comprehensive approach that integrates logistics, supply chains, and various additional factors in production processes [34].
- **Vehicular/Transportation** Introducing IoT technology into the vehicular and transportation domain leads to smart mobility [30]. Besides vehicles, smart mobility includes infrastructure as well and utilizes vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) technologies to create intelligent traffic systems [35]. These innovations should improve driving safety, decrease traffic congestion, and increase its sustainability, among others [36].
- **Healthcare** Smart healthcare covers a wide range of health-related applications. To name a few, it includes monitoring services with Body Area Networks (BANs) and wearable devices, pervasive access to healthcare information with smartphones and handhelds, and improves human activity recognition for rehabilitation [37]. Solutions like assisted ambient living or remote health monitoring reduce the pressure on healthcare systems without neglecting patients. These services often rely on small, low-powered devices and wireless communication. Hence, smart healthcare expands known IoT concepts like object monitoring and data collection into the traditional healthcare domain [38].
- **Energy** The term "smart energy" is not just a synonym for smart grids. It rather describes technologies for building sustainable energy systems in the future [39]. Besides smart grids and electricity, it covers sectors like heating and cooling and includes energy-related solutions for industry, buildings, and transportation [39].

[2]`https://www.etsi.org`, last access at 2020-03-22

- **Cities** The IoT and advanced information and communication technology (ICT) are key elements in building smart cities. Sensor networks deployed all over the city provide insights into city operations in real-time [40]. In general, technology in smart cities has a passive role. It collects and analyzes data, optimizes and automates infrastructure, and integrates city services [41]. Smart cities heavily rely on ICT [41] to improve quality of living for its citizen [40].

- **Wearables** Smart wearables are small, compact, body-worn devices with limited computing power [42]. They are typically aware of the users' environment and involve many different domains like healthcare and vehicles [43]. According to the ETSI report, smart wearables include technologies like nano-electronics, organic electronics, sensing, communication, and many more [30].

- **Farming/Agrifood** The continuing growth of the world population requires new solutions to increase food production. Using IoT technologies in farming and agriculture improves productivity and efficiency [44]. Similar to other IoT domains, smart farming covers a wide range of technologies and use-cases. It utilizes environmental data collected by sensor networks to monitor the farm, compute forecasts, and to create personalized recommendations [44]. Further, UAVs, especially drones, enable efficient health state inspecting of farms. Hence, farmers make more profound decisions before intervening on the feeding of soil or taking countermeasures against insects/fungi [45]. Embedding IoT systems in farms lead to connected farms, enabling more intelligent agricultural services by facilitating shared expert knowledge [46].

The examples of IoT applications given in the previous paragraphs are only a short excerpt on what is possible. In addition, these domains represent not an exhaustive list, and the IoT may influence other areas in the future as well. Nevertheless, these applications show the potential of the IoT to change existing domains fundamentally.

The IoT is often referred to as an ecosystem due to the variety of applications and the impact it has on a wide range of different domains. Firouzi et al. introduce an IoT ecosystem that consists of four different components, namely, *things, data, people, and processes* [27]. Independently, companies like Cisco identify similar components as key elements for the IoT [25]. Although many definitions and representations of the IoT exist, all of them consider the "thing" as an essential component [47]. Therefore, the following sections discuss important elements of the "thing" and link them to the focus of this thesis, the virtualization of IoT devices with respect to simulation platforms.

2.2 Connectivity and Communication

Independent of the domain and the IoT application, connectivity is one of the key elements of the IoT [25, 26, 27, 47]. Hence, IoT devices require a communication module to exchange messages with other devices and to participate in IoT networks [27]. The IoT

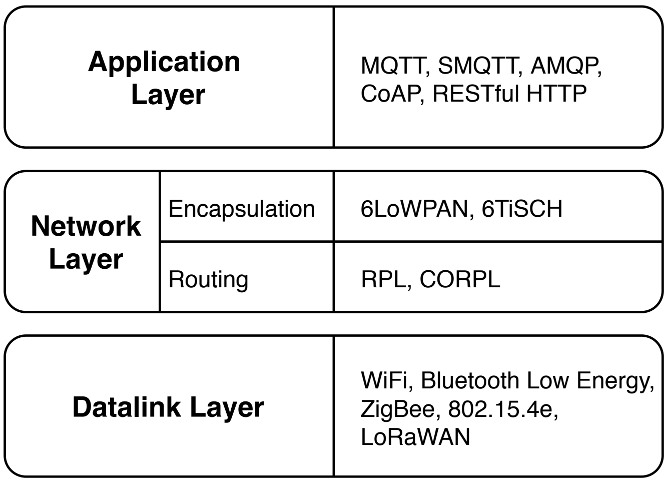

Application Layer
MQTT, SMQTT, AMQP, CoAP, RESTful HTTP
Network Layer
Encapsulation
6LoWPAN, 6TiSCH
Routing
RPL, CORPL
Datalink Layer
WiFi, Bluetooth Low Energy, ZigBee, 802.15.4e, LoRaWAN

Figure 2.1: Overview of important IoT protocols. Selection based on [48].

is an aggregation of different networks and includes a variety of protocols [26]. Devices range from small, low-powered sensors to powerful servers and, therefore, provide different network capabilities [26]. Additionally, the variety of domains and IoT applications have different requirements concerning connectivity, and the heterogeneity of the IoT increases the diversity of protocols [27]. Figure 2.1 shows important network protocols in the IoT and connects them to their corresponding OSI-layer. Due to the vast number of existing protocols, the list in Figure 2.1 is not extensive and protocols were selected based on [26, 48, 49, 50]. In addition, the following paragraphs summarize the protocols shown in Figure 2.1.

- **MQTT** MQTT is a lightweight publish/subscribe messaging transport protocol. It focuses on machine-to-machine (M2M) communication and works on constrained networks with high latency [50]. In addition, there exist extensions like Secure MQTT (SMQTT), that uses a lightweight encryption mechanism for secure message transmission [48].

- **AMQP** The Advanced Message Queuing Protocol (AMQP) transmits data as byte streams and is a wire-oriented protocol [51]. It uses TCP in the Transport Layer and, therefore, supports characteristics like reliable transmission, flow control, etc [51].

- **CoAP** The Constrained Application Protocol (CoAP) is a document transfer protocol similar to HTTP [51]. It was designed for constrained devices, i.e., devices with low resources [26]. It uses UDP in the Transport Layer and supports multicasting [51].

- **RESTful HTTP** Representational State Transfer (REST) is a web API design model for distributed systems and applicable for IoT applications as well [50].

- **6LoWPAN** 6LoWPAN is an acronym for *IPv6 over Low-Power Wireless Personal Area Networks*. It is an adaption layer that allows the transmission of IPv6 packets within small link-layer frames, i.e., using IEEE 802.15.4 [52]. It is compatible with other IP networks and does not require translation gateways or proxies [49]. Due to its efficiency, 6LoWPAN is one of the most commonly used standards in IoT communication [48, 49].

- **6TiSCH** The 6TiSCH protocol enables the transmission of IPv6 packets through the Time-Slotted Channel Hopping (TSCH) mode of the IEEE 802.15.4e standard [48, 53]. It focuses on wireless IP-enabled industrial networks with constrained devices [53].

- **RPL** The Routing Protocol for Low-Power and Lossy Networks (RPL) creates a Destination Oriented Directed Acyclic Graph (DODAG) with exactly one route between leaf nodes and the root node. It is a distance-vector protocol and supports different datalink protocols [48]. An extension of RPL is the cognitive RPL (CORPL) protocol. It uses an opportunistic forwarding approach by selecting a forwarder set, i.e., selecting multiple next-hop neighbors, and ensures that only the best receiver forwards a packet [54].

- **WiFi** The WiFi technology allows wireless data exchange and includes high-speed connections. It is defined by the WiFi Alliance as “Wireless Local Area Network (WLAN) devices that are based on the IEEE 802.11 standards“ [50].

- **Bluetooth Low Energy** The Bluetooth Low Energy (BLE) technology focuses on short-range connectivity. Its design for single-hop connections creates a trade-off between energy consumption, latency, piconet size, and throughput [55]. Its energy consumption can be ten times less than traditional Bluetooth but comes with the cost of higher latency [48].

- **ZigBee** The ZigBee protocol is based on the low-power wireless IEEE 802.15.4 standard and is a competitor to 6LoWPAN [49]. It supports different network topologies like mesh or star, and its design facilitates low data rate, long battery life, and secure networking [49, 51].

- **802.15.4e** This MAC layer protocol transmits frames through IEEE 802.15.4 channels [56]. It supports two operation modes, namely, the beacon-mode and the non-beacon-mode. In the former, CSMA-CA and superframes enable synchronization and deliver a dedicated bandwidth with low-power consumption. The latter is a traditional multiple access system based on CSMA-CA [51].

- **LoRaWAN** The Long-Range Wide Area Network (LoRaWAN) standard facilitates long-range connections with limited bandwidth and features low-power operation [57]. LoRaWAN supports 2-5 km of coverage range in urban areas and 45 km in rural regions [58]. Its network architecture is a “star of stars“ where end-devices communicate with gateways to exchange messages [57, 58].

2.3 Device Virtualization

Besides the importance of connectivity for IoT devices, the variety of applications leads to individual hardware requirements and involves various microcontroller architectures [19, 59]. Furthermore, IoT devices range from resource-constrained hardware with, e.g., little memory and limited power supply, to high-end devices for performing compute-intensive tasks [60]. This influences the choice of an appropriate OS for IoT devices [60] and may impact device virtualization. Further, with respect to the simulation platform and the focus of this thesis, it is important to understand the difference between simulation and emulation. Therefore, Section 2.3.1 clarifies the difference between simulation and emulation. Additionally, the virtualization of IoT devices has to consider the hardware characteristics of real devices. Hence, Section 2.3.2 provides an overview of virtualization technologies. In the last part, Section 2.3.3 summarizes relevant OSs for the IoT and discusses important requirements and characteristics.

2.3.1 Simulation vs. Emulation

As has been mentioned in Section 1.3, the proposed work integrates with existing device simulators and emulators. Before diving into device virtualization and its corresponding technologies, it is necessary to define simulation and emulation in this context and to identify the differences.

The term *emulation* can be defined as a computer system that imitates another computer system. Hence, the imitating system produces the same results as the imitated system when running the same computer program with the same inputs on both systems [61]. Further, an *emulator* can be a software that emulates different hardware, software, or firmware systems and mimics the characteristics of the imitated systems [61].

In comparison, the term *simulation* refers to a computer program that represents certain features, characteristics, parameters, or particular behavior of another system [61]. Therefore, a *simulator* describes a functional unit that imitates the behavior of a target system [61].

Although both, emulation and simulation, aim at imitating a certain target system, they have significant differences. Emulation focuses on reproducing original environments. It maintains a close connection to the real object and, therefore, can be time-consuming and difficult to achieve [62]. On the contrary, simulations provide a higher degree of freedom that often leads to biased simulation setups and inaccuracies [63]. Nonetheless, both concepts follow different objectives [64]. Simulations offer cost-effective and flexible environments that facilitate experimentation and are especially useful for demonstration purposes [64]. The development of an emulator may require more effort than building a simulator, but provides a precise imitation of the target system [62, 64].

2.3.2 Virtualization Technologies

The integration of IoT devices in the form of emulators and simulators requires proper virtualization technologies. Viable virtualization technologies have to support different target architectures to deal with the variety of microcontroller and processor architectures in the IoT [19, 59]. Therefore, this section explains current virtualization approaches, summarizes state-of-the-art virtualization mechanisms, and identifies relevant technologies with respect to the thesis.

The term "virtualization" can be defined as a software layer that abstracts computing resources [65]. This software layer represents the *Virtual Machine Monitor (VMM)* or *hypervisor*. The hypervisor has full control over the hardware resources and hides them from the OS. Hence, it is possible to run multiple OSs in parallel; each referred to as *Virtual Machine (VM)* [65].

Virtualization abstracts and isolates low-level functionalities and hardware and enables portability [65]. This flexibility leads to the adoption of virtualization technologies in areas like Cloud Computing, the IoT, and Network Function Virtualization (NFV) [66]. Thus, there exist many different virtualization types like Mobile, Data, Memory, Desktop, Storage, Server, and Network virtualization [67]. Due to the wide range of different virtualization types, describing each of them would go beyond the scope of this thesis. Hence, the following paragraphs focus on virtualization types relevant to the virtualization of IoT devices with respect to the simulation platform.

As has been previously mentioned, virtualization enables the emulation and simulation of IoT devices and is a key technology for dealing with different microcontroller and processor architectures. Based on [67], *server virtualization* provides the required functionalities. It is also known as *machine virtualization* and *cpu virtualization* and has the capabilities to run entire VMs with their own OSs on top of the host OS [67]. Server virtualization is a wider term that covers the following virtualization approaches [67]:

Emulation

Using emulation as a virtualization technique imitates a complete hardware architecture in software. Hence, it replicates a hardware processor and enables running OSs for different target architectures. This provides a high degree of flexibility and operates with unmodified guest OSs. Nonetheless, these advantages impact the performance as each instruction on the guest OS gets translated to the host [67].

Binary Translation

With binary translation, the hypervisor masks the hardware from the guest OS and mimics certain hardware capabilities. It is compatible with unmodified guest OSs and emulates instruction sets through code translation [68]. Hence, the guest OS is unaware of the virtualization. The hypervisor ensures safety and security and traps each instruction to perform translation tasks [67]. Although these mechanisms enable multi-platform portability and provide flexibility, it comes at the cost of low performance and significant design complexity [67, 68].

Hosted OS

In this approach, the hypervisor runs on top of an underlying OS, just like any other application [67]. The guest system inherits hardware support and device compatibility from the host. While the host OS desktop can be used continuously, the hypervisor layer creates additional overhead that decreases the performance [67].

Hardware-assisted Virtualization (Full Virtualization)

Similar to hosted OS virtualization, the hypervisor runs as an application in the user space [65]. Hardware-assisted virtualization utilizes processor virtualization extensions like AMD-V or Intel VT-x to improve its overall performance. This virtualization approach requires processors with virtualization support that allows virtualization without the need for binary translation or para-virtualization [68].

Para-virtualization

In contrast to other server virtualization techniques, the guest OS is aware that it is running in a virtualized environment [65]. Para-virtualization requires modifications to the running guest OS that reduces instruction translations and improves the performance [65, 67]. Besides, it does not simulate any hardware resources and relies on physical device drivers of the underlying host [67].

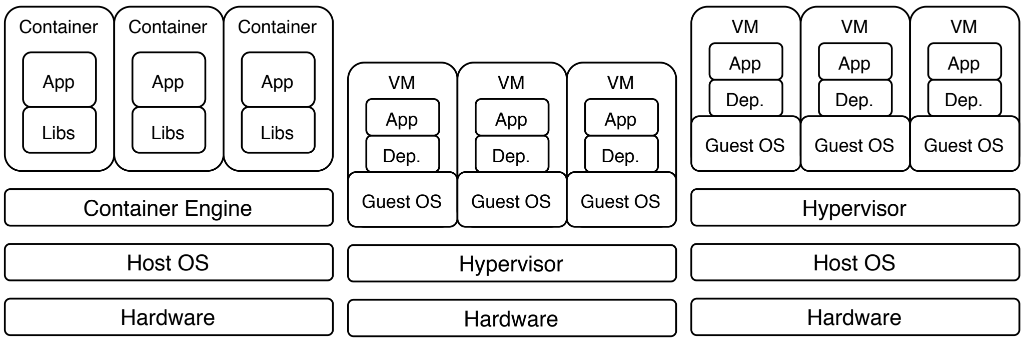
Native Virtualization (Hybrid Virtualization)

This hybrid approach combines full virtualization and para-virtualization. It uses input/output (I/O) acceleration techniques that increase its performance, except in setups where the translated instructions rely on emulated actions [67].

Container Virtualization

Container-based virtualization follows a very different approach compared to the previous virtualization technologies and does not use a hypervisor. It is rather a lightweight alternative that does not depend on hardware virtualization [66]. Containers run on top of the host OS and share the same system kernel. They use process isolation at the OS level and avoid hardware virtualization overhead [66]. Due to the shared kernel, the guest OS needs to be compatible with the host OS [65]. This improves efficiency and allows a higher density of virtualized instances [66, 67] but provides less isolation than hypervisors [66].

Due to the differences between containers and VMs, research distinguishes between container-based virtualization and hypervisor-based virtualization [26, 66, 69], and further classifies hypervisors into two different types, i.e., type-1 and type-2 [67]. For modern



(a) Container virtualization. (b) Hypervisor Type-1. (c) Hypervisor Type-2.

Figure 2.2: Hypervisor and container virtualization.

hypervisors, the distinction between them is not always clear as many hybrid versions exist [66]. Figure 2.2 shows the difference between both hypervisor types and container virtualization. Figure 2.2a illustrates the previous described characteristics for container virtualization. A container instance encloses an application with its dependencies and runs an isolated process in the user space of the host OS [66, 69]. Unlike hypervisors, the container engine does not virtualize hardware and is rather an abstraction layer that facilitates container management [69]. Type-1 hypervisors are native or bare-metal hypervisors and operate directly on the host's hardware [66] as shown in Figure 2.2b. This approach produces less overhead compared to type-2 hypervisors and leads to higher capacity and increased performance [70]. The type-2 hypervisors, or hosted hypervisors, run on top of the host OS, as shown in Figure 2.2c. The host OS provides virtualization services like I/O support and memory management [67]

2.3.3 OSs for the IoT

Another important component of IoT devices is the OS that runs on these devices. IoT devices include low-end as well as high-end hardware that affect viable OS solutions. Especially low-end hardware with constrained resources requires new systems as traditional OSs like Linux or BSD exceed the hardware capabilities [60]. While high-end devices utilize traditional systems like Linux [19], many new OSs emerged that require little resources and cope with IoT-specific characteristics [60]. Hence, the following paragraphs focus on OSs specifically designed for the IoT. They can be categorized based on their architectural concept into the following categories [60]:

- **Event-Driven OS** Programs on this type of OS are typically expressed as finite state machines [71]. The kernel resembles an infinite loop and acts on external events, i.e., interrupts. It is a resource-efficient approach with low complexity but restricts flexibility for programmers [60]. Examples of event-driven OSs are Contiki and TinyOS [60].

- **Multithreading OS** This is the common approach for traditional OSs like Linux. Each thread has its context and stack memory, and a scheduler performs context switching between threads. Therefore, multithreading OSs typically create some memory overhead [60]. Nevertheless, OSs like RIOT focus on IoT characteristics to address these issues [72].

- **Pure RTOS** The objective of RTOS is to fulfill real-time requirements and is mainly used in industrial and commercial contexts. Due to the importance of formal verification and model checking, these systems offer less flexibility to programmers [60]. One widely-used RTOS is FreeRTOS [60].

The variety of OSs for the IoT introduces significant challenges for interoperability and creates silos with incompatible solutions. Hence, reducing the number of different OSs for the IoT and providing consistent APIs across different platforms would be beneficial for the IoT ecosystem [60].

2.4 Technologies & Frameworks

The previous sections introduced the IoT, emphasized important characteristics, and summarized related concepts. This section provides background information about the tools, technologies, and frameworks used in this thesis.

2.4.1 libvirt

The libvirt project[3] is a toolkit for managing virtualization platforms. The following paragraphs are based on the libvirt documentation, the wiki, and the application development guide [73, 74, 75]. Essential components of libvirt include an API library, a daemon called *libvirtd*, and the command-line utility *virsh*. It is a collection of software that enables VM management and additional virtualization-related functionality. The following section defines the terminology used by libvirt and its objectives, provides a high-level view on the architecture, summarizes the lifecycle management, and highlights important features. Further, it explains how libvirt enables virtual network management and outlines methods to utilize libvirt in projects.

Terminology and Objectives

The term “node“ describes a single physical machine. Further, the “hypervisor“ keeps its meaning and describes a software layer allowing to virtualize a node. The last important concept is the “domain“ and refers to an instance of an operating system.

libvirt has the objective of providing a common and stable layer to securely manage domains on a node, including remote ones. It provides these characteristics by:

[3]`https://libvirt.org`, last access at 2020-03-22

- Secure API access to remote nodes.
- Targeted APIs to specific virtualization environments.
- Exposing a comprehensive, flexible, and stable API to manage domains and allow access to third-party software.

The main features of libvirt include VM management, remote machine support, storage management, network interface management, and virtual NAT and route based networking. Therefore, libvirt provides a foundation for higher level management tools and applications. It supports a wide range of hypervisors including Qemu, VirtualBox, VMware ESX, Xen, Microsoft Hyper-V, etc.

Basic Architecture

libvirt exposes an API for management applications, as shown in Figure 2.3. It uses hypervisor-specific mechanisms for controlling hypervisors and executing API requests. The architecture follows a modular approach to support extensibility. Further, libvirt utilizes driver modules to communicate with hypervisors. These drivers implement hypervisor-specific functionalities and provide a defined interface to the libvirt API. Nevertheless, drivers do not necessarily support all API functions, as hypervisors may not provide the required functionality.

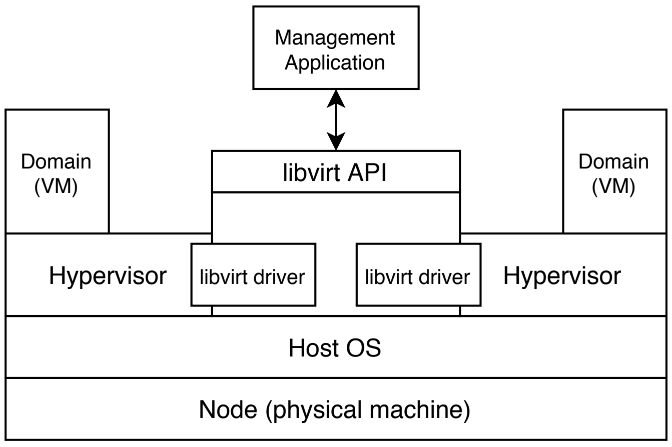


Figure 2.3: High-level architecture of libvirt.

VM Lifecycle

libvirt domains, i.e., VMs, can either be transient or persistent. When a transient domain is shutdown, or the host restarts, libvirt deletes it automatically. In contrast, persistent domains exist until they get deleted manually. Independent of the domain type, libvirt enables saving and restoring domain states.

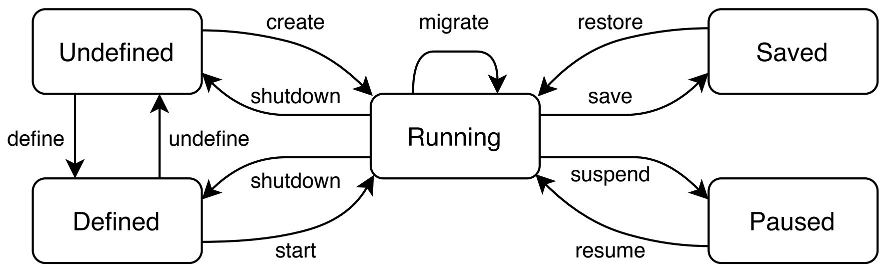


Figure 2.4: libvirt domain lifecycle.

Figure 2.4 shows the domain states and possible state transitions, indicated by rectangles and arrows, respectively. The base state is *Undefined*, and libvirt does not know anything about a domain in this state, as it has not been defined or created yet. *Defined*, or *Stopped* refers to domains that have been defined but are not running. Hence, only persistent domains can be in this state, as transient domains no longer exist after they have been stopped. The *Running* state applies to started domains that are being executed on the corresponding hypervisor. Further, *Paused* indicates that a domain has been suspended and libvirt stores the domain state temporarily until it is resumed. Similarly, *Save* suspends the domain but stores the state to a persistent storage where it can be restored in the future.

Networking

libvirt provides the functionality to create and manage virtual networks and to connect network interfaces of guest domains to them. The virtual networks can either remain isolated or use the active network interface on the host, e.g., to provide Internet access to guest domains. Similar to domains, virtual networks can be either transient or persistent.

The main components of libvirt networks are virtual network switches. In Linux systems, a virtual network switch corresponds to a network bridge. The major network configuration modes are *NAT forwarding*, *bridged networking*, and *PCI passthrough*. NAT forwarding allows outgoing connections for domains, e.g., access to the Internet. It allows incoming connections from the host, from other guests if they are in the same libvirt network, and blocks everything else. While NAT forwarding is useful for fast and easy deployments, bridged networking enables more advanced setups. In such a setup, the guest domain is connected directly to the local area network (LAN), and configuration requires advanced knowledge about networking and the underlying OS distribution of the host. The third configuration mode is PCI passthrough. This configuration option allows PCI network devices of the host to be directly assigned to guest domains.

Applications

The libvirt project offers a variety of applications to manage and maintain virtualization environments based on the libvirt API. As has been previously mentioned, virsh is such a tool. It is an interactive shell and batch scriptable application that provides all the functionality offered by libvirt. It is the only tool that is part of the libvirt core distribution.

Further, there exist many third-party tools, applications, and plugins for other software. For example, continuous integration tools like Jenkins and server monitoring software like Nagios offer libvirt plugins. Also, widely-used Infrastructure as a Service (IaaS) software like Nimubs and OpenStack rely on libvirt. In addition, libvirt offers libraries for a variety of programming languages like C, C++, Java, Python, and many more.

2.4.2 Qemu

The heterogeneity of IoT devices, including hardware and software, increases the complexity of device emulation [5, 19, 59, 60]. Dealing with different processor architectures requires a hypervisor that supports hardware virtualization and emulation of different target architectures. Qemu[4] is a generic machine emulator and virtualizer that supports guests with different target architectures. It is a type-2 hypervisor that runs in the user space of the host OS [76]. Besides, it is compatible with the Kernel-based Virtual Machine (KVM) module from the Linux Kernel that enables virtualization with near-native performance [77]. Qemu has two operating modes:

- **User mode emulation** This operating mode enables cross-compilation and cross-debugging for programs with different processor architectures [77]. In addition, it allows hosts to run programs compiled for different instruction sets [77, 78].
- **Full system emulation** Qemu also provides full system emulation, including emulation of processors and peripherals, and allows launching different OSs [77] like Linux, Solaris, Microsoft Windows, and so on. Further, it supports the virtualization of RTOS systems in simulation environments [79].

Therefore, Qemu is a powerful emulation software that supports the emulation of a variety of processor architectures, including 32 bit ARMv7, ARMv8, MIPS, x86, OpenRISC, and many more [77, 78].

2.4.3 Docker

The term “Docker“ relates to three different things, namely (i) *Docker Inc.*, the company, (ii) the container runtime and orchestration technology, and (iii) the open-source project that is now called *Moby* [80]. The following paragraph only focuses on the container runtime and orchestration technology.

Docker is an open platform to develop and manage containers [81]. One of the main components is the *Docker Engine*, a client-server application that consists of [81]:

[4]`https://www.qemu.org`, last access at 2020-03-22

- The Docker daemon process *dockerd* that manages Docker objects like images, containers, networks, etc.
- A REST API that enables the communication between external tools and the Docker daemon.
- A command-line interface (CLI) to control and interact with the Docker daemon.

The Docker platform is a popular ecosystem for container virtualization and provides many features to developers [80]. Hence, a comprehensive description would go beyond the scope of this thesis, and many books already exist [80, 81]. Therefore, the following focuses on the fundamental concepts of Docker containers.

The underlying technology utilizes several features of the Linux kernel to provide its capabilities [80, 81]:

- **Namespaces** Namespaces provide a layer of isolation to containers. It is a concept from the Linux kernel that wraps global resources in an abstraction that isolates a process from the rest of the system [82]. The Docker engine uses namespaces to isolate processes, network interfaces, inter-process communication, filesystem mount points, and kernel and version identifiers [81].
- **Control groups** Control groups (cgroups) are responsible for limiting resources to specific applications. This enables the Docker engine to share available hardware resources while maintaining the possibility to enforce limits and constraints to a specific container [81].
- **Union file system** The union file system (UnionFS) is a lightweight and fast file system that is organized in layers [81].
- **Container format** The container format represents a wrapper that combines namespaces, cgroups, and the UnionFS. Docker uses the libcontainer format, but future support for other container formats is possible [81].

2.4.4 Linux Networking Capabilities

The following paragraphs summarize Linux networking capabilities that are important for this thesis. Hence, it is not an exhaustive list.

Bridges

The Linux bridge provides functionality similar to network switches. It links different network segments and forwards packets between interfaces that are connected to it. A common use-case of bridges is to enable communication between VMs, containers, and the host [83].

Tun/Tap Devices

Tun and tap devices are virtual network kernel interfaces. They can be seen as simple Ethernet devices that enable user space programs to retrieve and transmit packets. Hence, the virtual Ethernet device receives packets from a user space program, instead of physical media. Further, sending packets to the tun or tap device writes them to the user space program instead of transmitting it via physical hardware [84]. Tun and tap devices operate on different network layers. Therefore, tun devices read and write IP frames, while tap devices operate on Ethernet frames [84].

Veth Pairs

The veth pair is a local Ethernet tunnel and consists of two virtual Ethernet devices that are connected with each other. Veth pairs can be used as standalone network devices but are especially useful for connecting different network namespaces, i.e., by placing one Ethernet device of the veth pair into each namespace [85].

Network Namespaces

As has been previously mentioned, namespaces isolate system resources for processes. Therefore, network namespaces provide isolation for network-related system resources. This includes network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, port numbers, and certain subdirectories in the */proc* directory [86].

Iptables

Iptables and ip6tables set up, maintain, and inspect the rules for packet filtering in the Linux kernel. These tables consist of lists of rules, i.e., chains, that define how the kernel handles network packets if a rule matches [87].

2.4.5 ns-3

ns-3[5] is a discrete-event network simulator. It is an open-source software that focuses on research and education. The ns-3 project provides a number of different documents that cover the software architecture and the core of ns-3 [88], the protocol, and device model libraries [89], and a general overview with practical examples [90]. Hence, the following section is based on these official documents [88, 89, 90].

The ns-3 software follows a modular approach and distinguishes between modules and models. ns-3 is organized into separate modules. These are individual software libraries that are built separately. Thus, ns-3 programs only link libraries required for the simulation. Models can be used in simulations to abstract real-world objects, protocols, devices, etc. Hence, a single module consists of one or several models.

[5]`https://www.nsnam.org`, last access at 2020-03-22

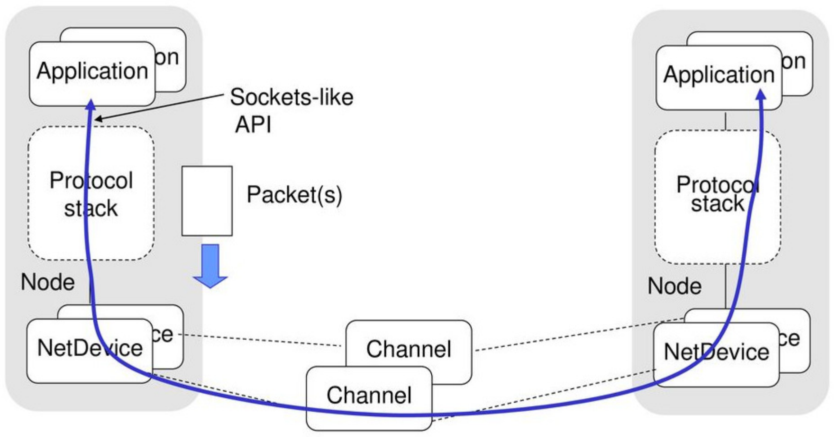


Figure 2.5: The basic ns-3 architecture [91]

Furthermore, ns-3 uses different levels of abstractions that facilitate flexibility to users. Figure 2.5 shows the basic architecture of ns-3. The key concepts of ns-3 are:

- **Nodes** In ns-3, node refers to the basic computing device abstraction. It can be compared to a computer in the real world.
- **Applications** In ns-3, the concept of OSs does not exist. Nevertheless, an application in ns-3 is like a user program running on a computer. Hence, applications run on nodes and generate activities during simulations.
- **Channels** The channel abstracts the communication link between nodes. It ranges from simple models, e.g., a single wired connection, to complex abstractions of Ethernet or wireless networks.
- **Net Devices** The net device in ns-3 refers to a Network Interface Card (NIC) in the real-world. It represents a virtual NIC and its corresponding drivers and can be installed on nodes. Net devices connect nodes to channels and enable nodes to communicate with each other.
- **Topology Helpers** Topology helpers simplify the configuration process of nodes, applications, channels, and net devices and reduce the effort to set up simulations.

Furthermore, ns-3 includes a real-time scheduler and provides net devices to interface with external devices. Hence, it allows real-time simulations and enables external components, e.g., VMs and containers, to participate.

2.4.6 Remote Procedure Calls

In programming languages, ordinary procedure calls are special cases of communication, with the invocation and the procedure residing within the same environment, e.g., on the same physical machine [92]. In distributed systems, invocation statements and procedure bodies are not necessarily on the same machine. Therefore, remote procedure calls (RPCs) offer a syntax that is similar to ordinary procedure calls, while the user is unaware of the location of the procedure body [92]. Hence, RPCs enable calls to remote procedures in different address spaces, independent of the machines both programs are running on [93].

The RPC concept is a client-server model and supports synchronous and asynchronous communication [92, 93]. In the last few years, RPC approaches got traction in high-performance computing (HPC). Further, Google released gRPC in 2015, a flexible and lightweight RPC framework that is widely used [93].

Apache Thrift

The Apache Thrift[6] framework is a high-performance RPC framework [94]. It supports a variety of programming languages and integrates a code generation engine. Thrift follows a modular approach and supports different serializers. Hence, users can choose serialization formats with respect to speed, size, and readability [94]. Further, it supports different transport protocols, including raw TCP and HTTP [94].

2.4.7 Micronaut

Micronaut[7] is a JVM-based framework for building modular microservices and serverless applications. It provides a rich set of features, including Dependency Injection (DI) and Inversion of Control (IoC), auto-configuration, service discovery, HTTP routing, and HTTP clients with client-side load-balancing [95]. It is a competitor to well-known frameworks like Spring[8], SpringBoot[9], and Grails[10].

Compared to alternative frameworks like SpringBoot, Micronaut offers advantages like fast startup time and a reduced memory footprint. It achieves this goal through the use of Java's annotation processors that precompile necessary metadata to perform tasks like DI. Further, Micronaut has special support for GraalVM[11] that allows Micronaut applications to utilize the *nativeimage tool* from GraalVM [95].

[6]`https://thrift.apache.org`, last access at 2020-03-22
[7]`https://micronaut.io`, last access at 2020-03-22
[8]`https://spring.io`, last access at 2020-03-22
[9]`https://spring.io/projects/spring-boot`, last access at 2020-03-22
[10]`https://grails.org`, last access at 2020-03-22
[11]`https://www.graalvm.org`, last access at 2020-03-22

CHAPTER 3

Related Work

The following chapter compares state-of-the-art developments in IoT simulations. It shows the relevance of simulations in IoT development processes and discusses alternative solutions with respect to their applicability. Further, it reviews current developments in consensus algorithms and systems for the IoT.

3.1 IoT Simulation & Research

Simulation in the IoT introduces many obstacles, and the development of sophisticated tools is a challenging task [5, 10, 12, 15]. Hence, this section follows a broader approach and tries to answer the question, why simulation is even necessary in the first place. It examines the research process of IoT systems and applications, discusses state-of-the-art solutions for each development stage, and shows the relevance of IoT simulations.

3.1.1 IoT Research

Papadopoulos et al. analyzed the research process cycle for the IoT [96]. Figure 3.1 shows the ideal process that starts with analyzing a new idea and ends with a real-world deployment after developing a successful prototype. In the first stage, researchers perform theoretical analysis and use models to gain insights into the proposed idea [96]. While modeling of IoT systems is actively researched [97, 98, 99], they focus on specific subjects like communication protocols [97], deployment of IoT systems [98], or particular domains [99]. Although more general modeling approaches exist [100], some parameters and environments cannot be modeled accurately [101]. Hence, modeling and simulations intertwine in some use-cases [10, 100].

In the second stage, researchers use simulators to verify and refine their ideas [96]. Simulations are especially useful for prototyping and developing a proof of concept, as

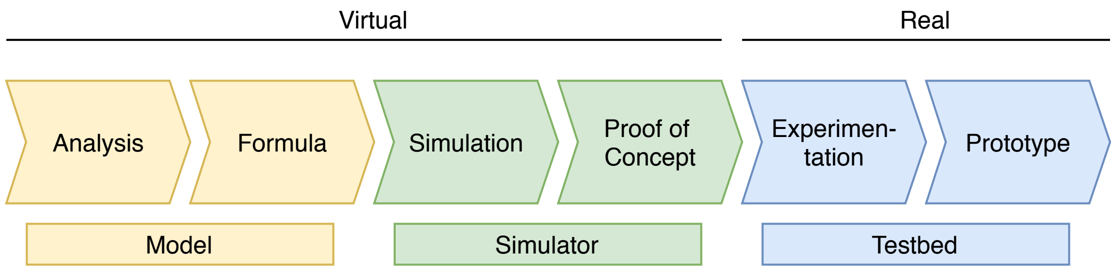


Figure 3.1: IoT research process cycle based on [96].

real-world experimentations can be avoided [15, 96]. Hence, researchers can test their ideas in virtual environments without setting up physical devices [96].

The third stage evaluates the proof of concept on real hardware [15]. As most simulators abstract low-level functionalities of IoT devices, evaluation results lose realism and complexity to some extent [96]. Therefore, using hardware devices allows researchers to test their system in real-world environments that current simulators cannot fully imitate [96].

Simulators are important in the IoT research process and close the gap between theoretical analysis, models, and tests on physical hardware in real-world environments [15, 96]. Further, researchers use models in combination with simulations for verifying and developing their idea. Hence, the first development stage already uses simulators [10, 100]. Both, evaluating models and running simulations, use virtual environments, and avoid expensive real-world setups [96]. Compared to virtual environments, running hardware-based tests is more expensive and work-intensive [5, 15, 96].

Therefore, the following paragraphs review hardware-based testing with its challenges and benefits. Followed by current solutions and the latest developments in IoT simulations, the section concludes with a discussion of both concepts and provides an outlook on future research directions.

3.1.2 Testbeds

Hardware-based testing in real environments, i.e., testbeds, allows researchers and developers to evaluate experiments in real-world like contexts [15, 102]. Testing IoT systems in real-world deployments uncovers potential problems in pre-deployment tests and enable performance analysis [103]. Formerly, researchers had to set up their own testbeds [15]. Building testbeds is an expensive task that requires proper resources, skills, and creates additional overhead [5, 15]. The emergence of open testbeds that researchers have access to solved these issues and increased the relevance of testbeds in IoT research [5, 15]. Initiatives like the CPS/IoT Ecosystem project, a cooperation of Technische Universität Wien (TU Wien), Austrian Institute of Technology (AIT), and Institute for Science and Technology (IST), advance research in the IoT by building IoT infrastructures in the real world that serve as research platforms and educational tools [104]. The following paragraphs summarize the characteristics and capabilities of popular open testbeds [15, 102, 103]. Nevertheless, the list is by far not extensive, as the number of available testbeds is substantial [102].

FIT IoT-LAB

The FIT IoT-LAB consists of 2729 low-power wireless nodes and 117 mobile robots [105]. It allows large-scale wireless IoT experiments, including tests with low-level protocols and advanced internet services [105]. The testbed uses nodes based on TI MSP430 and ARM hardware architectures and does not enforce specific operating systems [102]. It is a multi-site testbed and operates in six different but interconnected locations in France [102, 105]. In contrast to other testbeds, FIT IoT-LAB includes robots and offers controlled mobility [102].

Users manage experiments via a web-based frontend [102]. Also, FIT IoT-LAB provides a RESTful API and a CLI tool that enables developers to interact with the system [105] and allows them to build additional tools on top of it based on their individual needs [102]. Furthermore, it allows ssh access to the infrastructure to manage nodes [105].

SmartSantander

The SmartSantander testbed focuses on IoT applications in urban areas [106]. It is an experimental test facility that enables research in large-scale IoT systems in a real-world environment for smart cities. It consists of IoT devices deployed across Santander City, e.g., sensors attached to lampposts and nodes buried under the asphalt [106]. Further, mobile devices attached to public transports like buses and taxis generate valuable mobility patterns for experimentation [106]. An important characteristic of the testbed is its device heterogeneity [102]. SmartSantander includes indoor as well as outdoor deployments and involves citizens [102, 106]. It offers a variety of IEEE 802.15.4 devices, GPRS components, RFID and NFC tags, and QR codes [102, 103, 106].

The testbed provides a web interface and REST-based web services to manage experiments, i.e., to schedule jobs, specify resources, set up nodes, and so forth [102, 106]. Further, SmartSantander provides command-line scripts to automate and repeat experiments [102].

WISEBED

WISEBED is a large-scale system that federates existing testbeds [102, 107]. It provides an infrastructure of interconnected testbeds and uniform access to users [103]. Therefore, it consists of many heterogeneous devices and a variety of sensor types [102]. WISEBED defines a set of standardized APIs that operators from different testbeds can implement to participate in the federated testbed [107]. Besides, WISEBED supports virtual links between nodes and enables dynamic experiment specifications, involving physical and virtual nodes and links [102].

Therefore, users can access different testbeds with the same APIs, either directly or via the federated testbed. This enables researchers to deploy the same experiment to a number of different testbeds automatically [107]. Besides, WISEBED offers the functionality to repeat experiments automatically [102].

Conclusion

Multiple large-scale testbeds have been deployed in recent years [5] and got increasingly popular [15]. Open testbeds and research efforts to ease management and experimentation tasks enable researchers to run experiments on already deployed hardware [15, 102]. Testbeds enable testing in a real-world context and require the system under test to deal with unpredictable events and physical characteristics [102]. Further, open testbeds facilitate reproducibility, i.e., allowing different researchers to verify experimentation results [15].

Besides the advances in IoT testbeds, they have some considerable drawbacks. Deploying and maintaining testbeds are expensive tasks [5]. While these systems support an increasing number of different devices [102], they are still homogeneous compared to real IoT systems and quickly run on outdated devices as new hardware enters the market frequently [5]. Furthermore, researchers tune parameters and hardcode scenarios into the firmware of existing testbeds. This makes reproducible results difficult to obtain [103]. Besides, in-depth profiling is missing for certain experiments [103] and only a few testbeds support mobility [96]. Although testbeds consist of real-world deployments, there are still differences to targeted environments and IoT systems in the real-world [5, 103].

3.1.3 IoT Simulaton

Besides testbeds, simulators are widely used and actively studied for IoT research [96, 108]. Chernyshev et al. classify simulators based on the scope and the coverage of architectural layers, resulting in the following categories [15]:

- **Full stack** Simulators in this category enable researchers to represent entire IoT applications. The simulations include spaces, devices, operations, events, and may support sensors and actuators [15].

- **Big data processing** Tools within this category focus on the data processing capabilities of IoT systems. Therefore, these systems focus on cloud computing, big data processing, and reproducing data center mechanisms [15].

- **Network** Network-based simulators provide the majority of currently used systems [15]. This is mostly due to the similarity between IoT and wireless sensor networks (WSN), allowing researchers to re-use network simulators developed for WSNs [5, 8].

3.1.4 IoT Simulaton Tools and Frameworks

Due to the variety of IoT simulators, the following paragraphs summarize popular simulation tools currently used in research [12, 15, 109].

OMNeT++

OMNeT++ is a discrete event simulation environment [110]. It supports the simulation of communication networks, multiprocessors, and distributed systems [110]. Its general design [110] facilitates extensibility and allows the integration of use-case specific modules, e.g. urban mobility provisioning [15]. Therefore, OMNeT++ is rather a collection of frameworks and tools for building simulation scenarios than a simulator [109].

In addition, OMNeT++ focuses on WSN and network simulations [109] and does not support relevant IoT models out-of-the-box [15]. Nevertheless, researchers can integrate them manually [15], and several models are available to support researchers [110].

ns-3

ns-3 is a popular open-source network simulator that allows the simulation of large-scale networks [109]. It is the successor of ns-2 but is not an extension of it [109]. ns-3 is a new discrete-event simulator with a modern software core, and a design focused on scalability, modularity, and code quality [109, 111]. Its attention to realism provides components similar to their counterparts in the real world, e.g., network devices, nodes with multiple interfaces, and so forth [111]. Further, ns-3 offers a sophisticated framework for tracing and gathering statistics [111].

The simulator follows a modular approach and provides many built-in models [89], including relevant IoT protocols like 6LoWPAN and IEEE 802.15.4 [109]. Furthermore, it supports real-time simulations [88] and enables the integration of real hardware, e.g., testbeds, and virtualized systems [111].

Cooja

While simulators often miss important characteristics of real hardware due to abstractions, Cooja's design tries to close the gap between simulation and experimentation [96]. It simulates low-level functionality, i.e., sensor node hardware, and high-level behavior [112]. Cooja is a simulator designed for the Contiki OS [112], a popular OS for IoT sensors [15]. Hence, Cooja nodes can access standards and protocols of the Contiki OS during the simulation, providing more realistic scenarios [15]. Due to these characteristics, Cooja enables the development and simulation of low-level software [112], i.e., the firmware on virtual nodes can be deployed on real hardware with only minor modifications necessary [15].

Besides, Cooja supports a variety of network protocols relevant to the IoT, including MQTT, CoAP, 6LoWPAN, and 802.15.4 [15].

IOTSim

Based on the simulator categories discussed at the beginning of the section, IOTSim is part of the big data processing category [15]. IoT systems create massive amounts of data and, therefore, utilize cloud infrastructure for storage and computing tasks [113]. Developing and researching big data applications for the IoT is challenging, as setting up big data processing systems in the cloud is a complex task and can hinder progress due to time or budget constraints [113]. The IOTSim simulator addresses these issues by enabling simulations of big data processing models and cloud computing environments [15, 113].

It focuses on reproducing data center components, e.g., VM configurations, rather than sensor simulation [15]. Also, IOTSim includes a storage layer and models different storage types like Amazon S3, Azure Blob Storage, and HDFS [113]. For data processing, IOTSim supports batch processing with MapReduce simulations [15] and will implement stream computing models in the future [113].

iFogSim

The full-stack simulator iFogSim provides an environment that supports simulations of fog nodes, sensors, actuators, and application processing components [114, 115]. It allows large-scale simulations with customized fog environments and IoT devices [115]. iFogSim focuses on fog computing [114] and simulates fog devices as physical components [115]. Further, it provides sensors and actuators in the form of data sources and sinks, respectively. Both, data sources and sinks, can be customized to represent any data-emitting or information receiving IoT device [114].

3.1.5 Current Research in IoT Simulation

While the previous section looked into popular tools and frameworks for simulating IoT systems, the following paragraphs summarize current research and discuss novel approaches for IoT simulators. Nevertheless, the following list provides a small excerpt of currently researched IoT simulators and is by far not complete.

Dockemu

Dockemu focuses on simplifying the setup process of a simulation environment [116]. It utilizes Docker containers and emulates the network layer through Linux bridges and ns-3 [116]. The containers provide the user space of current OSs for applications to run. Further, ns-3 enables network emulation of wired and wireless layer two connections [116]. Since the first version of Dockemu in 2015 [116], more recent publications followed [117, 118].

Dockemu intends to enable the simulation of complex IoT scenarios, including client-server applications and network protocols relevant to the IoT [117]. Therefore, Portabales and Nores [117] propose a network setup that enables the emulation of LTE and 6LoWPAN networks in combination with applications running in Docker containers [117]. Unfortunately, the current version of ns-3 does not fully support IPv6 for external devices [119], and therefore, reduces the functionality of Dockemu.

VIoLET

VIoLET is a simulator for deploying large-scale IoT setups in virtual environments, i.e., within cloud VMs [120]. Users submit deployment documents that describe the simulation in JSON format. VIoLET represents virtual devices as Docker containers and automatically provisions VMs in the cloud. Further, it provides synthetic sensors for data generation [120]. The simulator uses an overlay-network to connect containers and allows distinct public and private networks in virtual environments [120].

The framework enables users to restrict computing resources of containers via the built-in functionality of Docker [81, 120]. Furthermore, the network topology configuration allows defining public and private networks, setting the visibility of devices in the network, and limiting bandwidth and latency between pairs of devices [120]. The framework enforces bandwidth and latency requirements with the Linux iproute2 package, and Traffic Control (TC) rules [120].

ELIoT

ELIoT is an emulation platform for the IoT and allows the simulation of large-scale IoT scenarios [17]. It uses Docker containers as a virtualization technology and aims at portability, i.e., facilitating a convenient and flexible setup process of test environments [17]. ELIoT does not emulate low-level network protocols like 802.15.4 or 6LoWPAN and focuses rather on the application layer. Its nodes implement a full CoAP network stack and provide LWM2M/IPSO objects to simulate IoT entities and their corresponding interactions [17].

Although ELIoT refers to its virtual devices as “emulated“, it does not emulate any hardware characteristics and abstracts networking within Docker containers [17]. It instead emulates the behavior and communication pattern of real devices [17]. Virtual devices, i.e., containers, use a minimal Node.js base image built on top of Alpine Linux. The image already includes a CoAP/LWM2M module and provides scripts for simulating the behavior of different sensors [17].

Hybrid Virtualization Platform

Recently, Lai et al. published their work on a hybrid virtualization approach for simulating IoT systems in virtual environments [121]. While network emulators and other simulation platforms utilize x86-based VMs and Docker containers to virtualize nodes within simulations, the proposed platform focuses on the emulation of devices with different target architectures [121]. They utilize multiple virtualization technologies, i.e., KVM, Qemu, and Docker, to emulate heterogeneous IoT systems [121].

Further, Lai et al. utilize the cloud to improve the scalability of their proposed platform. Similar to VIoLET, network emulation is based on modifications of the link between a pair of nodes [121].

Multi-level Simulation

Building general IoT environments to simulate heterogeneous IoT systems is a difficult task. Therefore, D'Angelo et al. proposed a multi-level simulation system to cope with these challenges [122]. These systems consist of a simulation coordinator that orchestrates multiple, task-specific simulation models [122]. In addition, models can operate on different levels of detail and enable researchers to target their systems precisely [122]. Further, it provides the flexibility to decide whether granular simulations are necessary for certain parts of the system. Utilizing higher-level simulations improves performance and requires fewer resources. Therefore, running low-level simulations only on critical parts of the system increases the overall performance of the simulation [12, 123].

Due to the integration of multiple simulators, the multi-level simulation has to deal with interoperability among simulators and facilitating inter-model interactions, i.e., models may have to synchronize or need to exchange state information [12].

Conclusion

Established simulators as shown in Section 3.1.4 miss important requirements for IoT simulations, i.e., a single simulator is not sufficient for end-to-end simulations [15]. The issues with these tools include the lack of support for IoT-specific protocols, missing application layer capabilities, dealing with device heterogeneity, imitating the dynamic behavior of IoT systems, and missing support for real-time requirements [15, 124].

Current research tries to address these issues and defines essential characteristics and requirements for IoT simulations, including the deployment of heterogeneous technologies, extensibility, scalability, reproducibility, cost-effectiveness, support for IoT standards and protocols, and overall practicality, i.e., quality of experience (QoE) [15, 118, 120, 121, 124].

Dockemu, VIoLET, ELIoT, as well as the hybrid virtualization platform proposed by Lai et al. address scalability as a core requirement [17, 117, 120, 121]. Further, Dockemu uses ns-3 for network simulation and, therefore, covers layer two and three network protocols relevant for the IoT [109, 117]. While Dockemu focuses on a simple setup process, it does not support heterogeneous devices, e.g., different target architectures, and changing network topologies and modifying applications on devices introduces overhead [117].

Besides low-level devices like sensors and actuators, IoT deployments include fog, edge, and cloud computing as well [10, 120]. Therefore, VIoLET provides a virtual environment that specifically targets these systems [120]. It enables a cost-effective, scalable simulation tool with convenient features for users, for example, pre-configuration and data generation. Nevertheless, VIoLET does not provide network and device emulation and omits mobility [120].

Similarly, ELIoT neglects network emulation and focuses on simulating the behavior of devices rather than hardware emulation [17]. It emphasizes the simulation of the application layer, provides reasonable scalability, and improves QoE with ready-to-use devices [17].

The platform proposed by Lai et al. focuses on the heterogeneity of the IoT and enables emulated devices with a different target architecture compared to the host [121]. In addition, it allows the network topology to change dynamically but does not emulate low-level protocols [121]. Furthermore, the proposed work does not provide enough information to reason about usability-related tasks, e.g., setting up nodes and running developed IoT applications, repeating experiments and reproducibility, or obtaining results via profiling, tracing, and monitoring [121].

3.1.6 Open Challenges

Although progress has been made in testbeds and IoT simulation, current solutions still lack important features [10, 15, 102, 103]. These include sophisticated monitoring mechanisms, detailed profiling, and support for cybersecurity research [15, 103].

Furthermore, the heterogeneity of IoT systems imposes a challenge on both, testbeds and simulators [10, 102]. This includes extensibility as well, as new IoT devices enter the market frequently [5, 10].

Compared to IoT simulators, testbeds offer more reliable results and do not miss important device characteristics due to abstractions or virtualizations [5, 15]. Nevertheless, simulators provide more flexibility, enable advanced mobility scenarios, and are more cost-effective, as setting up and maintaining testbeds are expensive tasks [15, 102].

3.2 Consensus in the IoT

Finding consensus in a distributed system is a well-known problem in computer science. It is often associated with the work of Fischer, Lynch, and Paterson [125] several decades ago on distributed consensus [126]. Consensus describes the process of reaching an agreement about a certain state or value between entities [126, 127]. Furthermore, it enables consistent behavior within a group of entities, even in the case of failure [20].

In contrast to solutions for consensus in synchronous systems, Fischer, Lynch, and Paterson [125] have shown that fully asynchronous settings may never reach consensus due to the possibility of nontermination [23]. Therefore, many approaches have been proposed since then, including popular algorithms like Paxos [23].

Although the consensus problem has been studied extensively [21], it is still actively researched due to its importance in cloud computing, task coordination, and data storage systems like databases [20, 21, 23]. In recent years, further use-cases for consensus algorithms have been developed due to the emergence of the IoT, i.e., large-scale, highly distributed systems [20, 128, 129].

Many established consensus algorithms and specific implementations operate in data centers and cloud computing environments [21, 130]. Hence, their design does not necessarily facilitate IoT systems and introduces further challenges to consensus algorithms [128]:

- Devices have limited power supply and constrained resources [26]. Therefore, efficiency is an essential characteristic [128].
- Networks involve unreliable, wireless connections and introduce noise, latency, and changing topologies [128].
- Devices are easily compromised and, therefore, pose a security threat [128].

Due to these requirements, novel consensus algorithms have been proposed for the IoT. Li et al. [129] introduced a decision making procedure for service-oriented IoT deployments based on distributed consensus. At, for example, edge nodes, services may have insufficient information for making coherent decisions. The proposed algorithm divides the system into clusters, calculates local decisions, and uses this information to reach global consensus [129].

Further, the work of Zhao et al. [131] focuses on cyber-physical systems (CPSs) in the IoT. The proposed algorithm emphasizes event detection for industrial process monitoring, automatic alert systems, prediction of potentially dangerous events, and attack detection [131]. Its design uses a jointly connected graph model that supports dynamic topologies and improves reliability [131].

Colistra et al. [132] developed a consensus approach for task allocation in IoT systems. It enables IoT devices to cooperate and share the effort on certain tasks. Thus, IoT devices make their resources available and execute applications assigned to the whole network [132]. The proposed optimization protocol allocates available resources within the network to reach the common goal most effectively [132].

Li et al. [133] proposed a distributed, robust control mechanism for uncertain multi-agent systems. Applications relying on these methods include satellite formations, vehicle platoons like unmanned ground vehicles, networked sensings, and so forth [133]. Multi-agent networks consider node dynamics, models of interacted topologies, formation geometry, and the distributed robust controller [133]. It uses a virtual leader approach to control formations, and information is shared among surrounding neighbors or the virtual leader to achieve global consensus [133].

Furthermore, the IoT imposes significant challenges on privacy and security solutions [134]. Systems with a centralized authority imply a single point-of-failure and provide insufficient scalability [134]. These issues can be solved with decentralized consensus systems that utilize trustless and immutable public ledgers. Thus, introducing blockchain technology to the IoT [134]. Yeow et al. [134] review state-of-the-art consensus systems and provide an overview of blockchain-based consensus algorithms that can be utilized in the IoT [134].

Although these approaches differ in their application areas and utilize various mechanisms, algorithms, and technologies, the evaluations to demonstrate the performance and the effectiveness of the proposed solutions rely on simulations [129, 131, 132, 133]. This underlines the importance of sophisticated simulation tools during the IoT research cycle, as shown in [96]. Furthermore, Colistra et al. [132] use real hardware to validate the simulation results of their proposed algorithm successfully.

Besides novel approaches for consensus algorithms in the IoT, Méndez et al. [135] evaluate the well-known bully algorithm for leader election on low-performance IoT devices. The paper demonstrates that concepts of traditional consensus algorithms might be applicable in IoT scenarios without device limitations affecting their suitability [135]. Furthermore, Poirot, Nahas, and Landsiedel [23] propose *Wireless Paxos*, a consensus algorithm based on Paxos for low-power wireless networks. The evaluation of Wireless Paxos on two different testbeds demonstrates reasonable performance and shows correctness [23]. Hence, supporting the conclusion from Méndez et al. [135]. Therefore, assessing the Raft algorithm on a wireless IoT system provides an adequate use-case for the evaluation of the proposed simulation platform.

CHAPTER 4

Software Engineering Process

The scope of this thesis can be structured into three parts. Following the categories previously defined in Chapter 1, these are the architecture framework, the network simulator, and Raft. For every part, this chapter provides a detailed description followed by the requirements and explains essential design decisions. First, Section 4.1 defines the architecture framework and introduces its design. Then, Section 4.2 addresses the need for simulated communication channels and defines the capabilities of the network simulator. Further, Section 4.3 demonstrates the interactions between a simulated IoT device and the network simulator, which already enables the simulation of IoT systems. Last, Section 4.4 provides information about the implementation of the Raft algorithm.

The thesis uses a novel terminology to distinguish between components. This nomenclature is not universally recognized, but rather introduced in this work to allow a more precise naming. As previously stated, the architecture framework can be seen as a toolset to integrate IoT simulators and emulators. Further, the *integration layer* specifies an instance of the architecture framework, i.e., a process integrating a single and unique instance of a simulator or emulator. For simplicity, IoT simulators and emulators are further referred to as *virtualized devices*. Last, the term virtualized IoT device (VIoTD) describes the combination of a virtualized device and its integration layer, i.e., an IoT device in the simulation.

Figure 4.1 provides an overview of how the architecture framework and the network simulator integrate into the simulation platform at UIUC. While Section 4.1 and Section 4.2 focus on the capabilities of a simulated device and the network simulation, respectively, the backend of the simulation platform is responsible for managing devices, allowing users to interact with and execute different scenarios. For example, users can upload files and access them during a simulation. Figure 4.1 provides a simplified view on the simulation platform. The platform itself consists of different components with much higher complexity. These details are omitted at this point, as they are not relevant for this thesis.

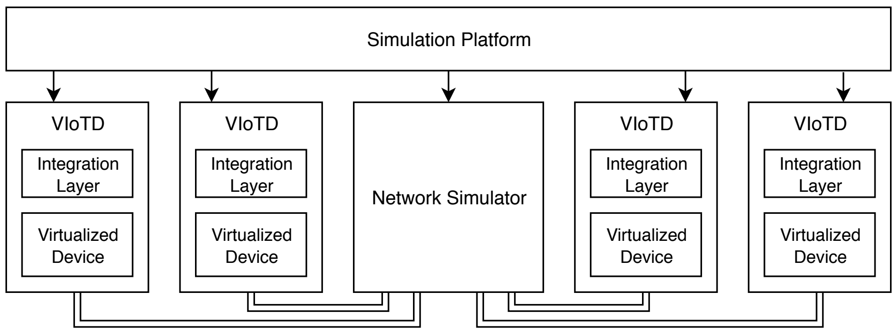


Figure 4.1: Integration of the architecture framework and the network simulator into the simulation platform.

4.1 Architecture Framework

The main objectives of the architecture framework are abstracting virtualized devices and providing a uniform interface to the simulation platform while adding as little computational overhead as possible. Based on [10], the architecture framework supports VMs and containers as virtualized devices. Further, the framework supports processes as virtualized devices as well due to specific requirements of the simulation platform at the UIUC and to improve flexibility.

To decide whether the framework is responsible for a certain functionality, the following definition can be used:

Definition 1 *The framework includes only functionality needed by an individual instance of a virtualized device.*

In certain cases, features, which seem useful in the first place, are not included by intent, as they do not fulfill Definition 1. An example is the hypervisor setup for a VM. If a VIoTD requires a specific hypervisor, the architecture framework is not responsible for its installation. Any instance of the virtualized device requires this hypervisor, not only an individual one. Once the installation process is done, this functionality is dispensable.

4.1.1 Requirements Analysis

The concept of a VIoTD is abstract and requires a specific definition of its functionalities and capabilities. Hence, a requirements analysis is necessary. As the design and the functionality of the architecture framework affect different stakeholders, providing a clear definition for each stakeholder improves the process for creating user stories. Hence, the architecture framework has the following stakeholders:

- **User**: Someone who uses the simulation platform to simulate IoT scenarios, i.e., the end-user.

- **Developer**: Someone who developed an emulator or simulator and wants to integrate and use it in the simulation platform.
- **Platform Administrator (Admin)**: Platform administrator, developer, or maintainer. Owner of the platform.

To create a design for the architecture framework that considers all stakeholders equally, each role has its own user stories. Therefore, Table 4.1 shows all stories for the *User* role, Table 4.2 defines the requirements for *Developer* and Table 4.3 specifies important functionalities for the *Admin*.

Furthermore, the user stories comprehend essential characteristics of IoT simulation platforms discussed in Section 3.1. Convenient integration of virtualized devices improves extensibility and addresses the heterogeneity of the IoT [10]. In addition, they cover relevant network protocols and include monitoring and logging, so users are able to analyze the simulation results [15, 117]. Further, they consider QoE for users as well as maintainers of the simulation platform [117, 121].

Table 4.1: User stories for *User*.

No.	**User Story**
1	As a user, I want to simulate systems with more than one virtual IoT device.
2	As a user, I want to build systems where virtual IoT devices communicate with each other.
3	As a user, I want that a virtual IoT device has the same hardware resources as the real device.
4	As a user, I want to connect a sensor to a virtual IoT device with USB/GPIO/other interfaces.
5	As a user, I want to create a virtual smart sensor that measures the environment (e.g., by calling an API in the environment).
6	As a user, I want to create a virtual smart sensor that gets input data (e.g., by providing an API) from the environment.
7	As a user, I want to create a virtual smart sensor that sends pre-defined values as sensor data.
8	As a user, I want to create a smart actuator which interacts with the environment.
9	As a user, I want to create a smart actuator that gets feedback from the environment, e.g., whether the actuator executed its job successfully.
10	As a user, if I choose an emulated SBC as VIoTD, I want to specify which libraries need to be installed on the emulated SBC.
11	As a user, I want to know which IP address a VIoTD has during the simulation.

Table 4.1: User stories for *User*.

No.	User Story
12	As a user, I want to specify how my device is connected to the IoT infrastructure (wired vs. wireless).
13	As a user, I want to specify the protocol of the communication channel (Ethernet, WiFi, etc.).
14	As a user, I want to specify coordinates for a wireless VIoTD.
15	As a user, I want to run multiple simulations with the same set of devices.
16	As a user, I want to specify to which devices a VIoTD is connected to in a wired setup.
17	As a user, I want to know when a VIoTD is ready, so I can upload code to it or start the simulation.
18	As a user, I want to submit my code to the VIoTD.
19	As a user, I want to remove all uploaded code from a VIoTD.
20	As a user, I want to access the uploaded code on a VIoTD.
21	As a user, I want to specify arguments that are passed to my application when it gets started.
22	As a user, I want that my program can access any port of the VIoTD during runtime.
23	As a user, I want to specify tasks that should get executed between the start of the VIoTD and the execution of my code.
24	As a user, I want to run a VIoTD in a simulation for a pre-defined time (e.g., 10 minutes)
25	As a user, I want to start a simulation.
26	As a user, I want to stop a simulation.
27	As a user, I want to terminate a simulation.
28	As a user, I want to see when a device in my simulation crashed.
29	As a user, I want to know when the simulation has finished.
30	As a user, I want to see the log output of my executed code.
31	As a user, I want to see which device did not work properly if my simulation fails.

Table 4.2: User stories for *Developer*.

No.	**User Story**
32	As a developer, I want to configure how my virtualized device gets started.
33	As a developer, I want to specify what hardware resources (CPU, RAM, storage, Networking) my virtualized device has in the real world.
34	As a developer, I want to enable a health check for my virtualized device.
35	As a developer, I want to integrate my virtualized device without making major changes to it.
36	As a developer, I want to provide a short description for my virtualized device.
37	As a developer, I want that users can run custom code on my virtualized device.
38	As a developer, I want to specify how the simulated environment can interact with my virtualized device (if applicable).

Table 4.3: User stories for *Admin*.

No.	**User Story**
39	As an admin, I want to specify where the platform can obtain the logs of any virtualized device.
40	As an admin, I want to specify where the platform can obtain the application logs (if applicable) of any virtualized device.
41	As an admin, I want that simulations can run in parallel, i.e., different users run simulations simultaneously.
42	As an admin, I want to see when a VIoTD crashed (only if the device crashed but not if the code from the user failed).
43	As an admin, I want to get the log output of crashed devices.
44	As an admin, I want the VIoTD to be resource efficient. (architecture framework must not add considerable overhead).
45	As an admin, I want to specify the logging infrastructure which should be used by the VIoTD for its application logs.

The user stories in Table 4.1, 4.2, and 4.3 do not present an exhaustive list of value-adding functionalities. It is rather a foundation providing important features to run IoT simulations. It can be seen as a proof of concept where features will be specified and added based on future feedback.

Besides the features for *User*, *Developer*, and *Admin*, there are additional requirements for the architecture framework. IoT devices have constrained resources, and therefore, they use operating systems (OS) specifically designed for the IoT or particular use-cases. Lightweight OS and Real-Time Operating Systems (RTOS) are commonly used in IoT applications [6]. Despite the variety of OS running on IoT devices, these devices differ on the hardware level from traditional microcontrollers as well [19]. For example, microcontrollers with ARM architecture are frequently used in IoT devices [59]. Hence, the architecture framework must be compatible with OS used in IoT and has to enable different hardware architectures for virtualized devices.

The architecture framework is based on these requirements, and the following sections describe its design and implementation.

4.1.2 Design

The design of the architecture framework is modular to facilitate extendability and to reduce the cost of changes. As it is part of an early-stage research project, future changes are likely to happen, and more sophisticated use-cases require additional features. The design of the architecture framework can be seen in Figure 4.2. The following sections provide a detailed description of the framework design and the capabilities of each module.

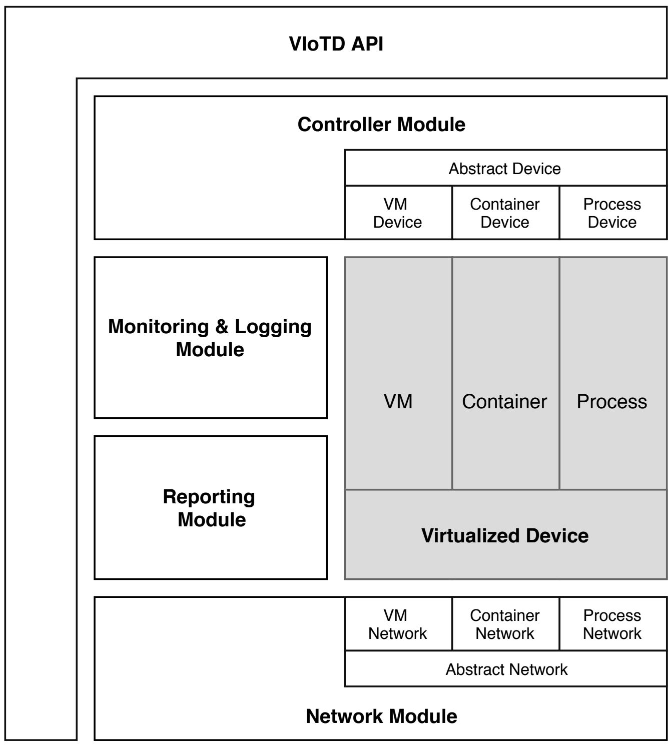


Figure 4.2: Module design of the architecture framework.

Controller Module

The controller module interacts directly with the virtualized device and is responsible for managing its lifecycle. Further, it abstracts any interactions with the virtualized device for other modules, i.e., it hides virtualized device specific characteristics behind well-defined interfaces. Therefore, other modules do not have to be aware of the virtualized device and its virtualization technology.

Based on the requirements analysis, the VIoTD can be in the states and run actions shown in Figure 4.3. The controller module is triggering state transitions, i.e., executing actions, and brings the VIoTD and the virtualized device into the correct state. Further, it performs only actions if they are allowed in the current state and reports an error otherwise. If the execution of an action fails, it puts the integration layer into the *VIoTD error* state and waits for the user to intervene.

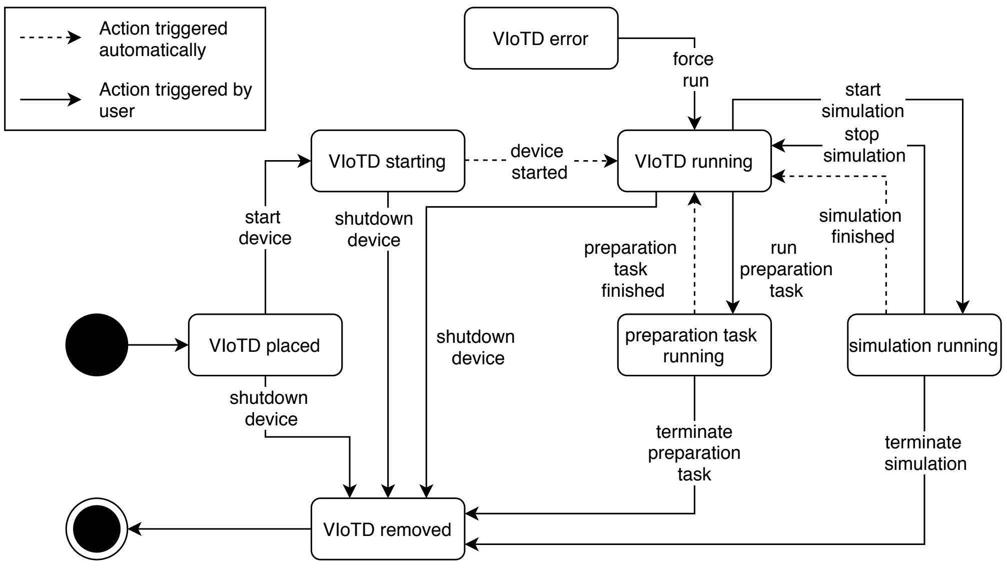


Figure 4.3: Possible states and actions of the VIoTD.

In the *VIoTD placed* state, only the integration layer is running, but not the virtualized device. The simulation platform is responsible for starting a new integration layer process. The integration layer reads the VIoTD and platform configuration files and exposes an API for further interaction.

The device is in the *VIoTD running* state when it has started successfully, and neither a simulation nor a preparation task is running, meaning the device is idle. Only in this state, the controller module accepts actions other than shutdown or terminate.

The *preparation task running* and the *simulation running* states are very similar. When the VIoTD transitions into each state, both read their corresponding input file. As previously mentioned, users can upload files, i.e., custom commands and source code, which can then be executed in the respective state. The controller oversees the execution of the preparation task and the simulation and changes into the *VIoTD running* state as soon as the tasks have finished. The simulation can be stopped by the user as well. This puts the VIoTD into the running state, and further preparation tasks and simulations can be executed. In case the user terminates the running commands, the controller moves the VIoTD into the *VIoTD removed* state.

The remaining states, namely *VIoTD starting* and *VIoTD removed*, highly depend on the virtualization technology used by the virtualized device. The controller module supports virtualized devices, which are either VMs, containers, or processes. The following paragraphs explain each supported virtualization technology in more detail and discuss state-specific characteristics.

VM The controller module uses libvirt[1] to manage and interact with VMs. Therefore, the architecture framework supports any hypervisor, which is supported by libvirt. The integration of a VM with the architecture framework does not require any significant changes to the VM. The only requirement is that the VM allows serial connections automatically, e.g., for a Debian Linux[2] system, a proper solution is to start getty[3] automatically during boot.

In the *VIoTD starting* state, the controller provisions a new VM, starts it, and establishes a connection. If login data is provided in the VIoTD configurations, the controller executes the login automatically after the VM has finished booting and transitions into the running state, where it expects further actions from the user.

In the *VIoTD removed* state, the controller tries to gracefully shutdown the VM. If this is not possible within a certain time, the shutdown is enforced. The VIoTD configuration specifies how long the controller should wait before shutting down a VM forcefully. In the following cleanup task, the previously provisioned VM image gets removed so that any image changes during the simulation do not affect future runs. Further, the controller triggers the network module to remove previously created interfaces and stops the integration layer.

Container The controller module supports Docker[4] containers, which is a widely used platform for running containers. The architecture framework is compatible with every Docker image and does not require any changes.

In the *VIoTD starting* state, the controller module uses the image specified in the VIoTD configuration to create a container. The current implementation uses a limited number of configuration options provided by the Docker SDK[5] to create the container. These options are memory limit, volumes, sysctls for adding IPv6 support and options needed

[1] `https://libvirt.org`, last access at 2020-01-16
[2] `https://www.debian.org`, last access at 2020-01-16
[3] `https://wiki.debian.org/getty`, last access at 2020-01-16
[4] `https://www.docker.com`, last access at 2020-01-16
[5] `https://docker-py.readthedocs.io/en/stable`, last access at 2020-01-16

by the controller module like tty or remove[6]. These options are sufficient for the current platform but can be easily extended in the future if needed.

The controller module mounts a directory into the container, which is only available to this specific VIoTD. The user can upload source code or custom files to this directory via the simulation platform. During the execution of the simulation or preparation task, the VIoTD has access to these directories and can read or execute the previously uploaded files.

When the controller receives a shutdown or terminate command, it simply stops the container. Due to the *remove* configuration option, the container is deleted automatically.

Process The process is the most lightweight option to run a virtualized device. The controller module supports any runtime environment and interpreted language, as long as it is installed on the host machine. The architecture framework does not enforce any implementation details upon the process, which keeps the integration procedure simple. As previously mentioned, the architecture framework is not responsible for setting up the host machine or installing necessary dependencies. Therefore, the *VIoTD starting* and the *preparation task running* states cannot be used to set up the host machine. Further, the *preparation task running* state does not affect the VIoTD.

When the controller module receives the command to run the simulation, it executes the process according to the options specified in the VIoTD configuration. Program arguments can be passed to the process via the simulation file, which users can upload via the simulation platform. If any program argument needs to reference any previously uploaded files, any path in the argument should be replaced with the placeholder *<working-dir>*. The process executor replaces this placeholder before the process gets started and allows access to any previously uploaded data.

When the simulation has finished, is stopped by the user, or a termination signal is received by the controller module, the process gets terminated.

Monitoring & Logging Module

The module is responsible for providing state and health information about the virtualized device and the integration process itself. Its functionalities can be grouped into

- VIoTD logging
- State monitoring
- Device monitoring

VIoTD logging includes the configuration of the logging framework used by the integration layer. This configuration can be set in the platform configuration file. Currently, logs of

[6]`https://docker-py.readthedocs.io/en/stable`, last access at 2020-01-16

the integration process can be either printed to stdout or stored in a file. Due to this point, there are no additional requirements for providing VIoTD logs, and therefore, console output and file are the only two options currently available. However, the modular implementation allows developers to add additional methods in the future without making significant changes to the code base.

In addition to logging, the monitoring module exposes an API method with which the platform can obtain the state and health of the virtualized device. During the integration of the virtualized device into the platform, the developer of the virtualized device has to specify how health information can be obtained. This configuration option is specified in the VIoTD configuration file.

Reporting Module

The reporting module is responsible for obtaining log output and forwarding it to the platform to provide feedback to the user. The difference to the monitoring module is that the reporting module is responsible for delivering logs created by the applications or commands running on the virtualized device. It defines an interface that accepts a file descriptor where log data is expected. It obtains the log output from the file descriptor and forwards it to the configured reporting appender. With this modular approach, the reporting module follows the design of logging frameworks like log4j[7].

The current version of this module provides two appenders

- **File Appender** Log output on the virtualized device is stored in a file. The directory where the application logs are stored can be configured via the platform configurations.

- **Kafka Appender** The module creates a Kafka producer that forwards log output to a Kafka topic. The topic name, as well as the host and port of the Kafka service, can be configured via the platform configurations.

These two appender types were chosen due to two reasons. First, the simulation platform uses Kafka[8] as a logging infrastructure, and therefore, the architecture framework has to support it. Second, storing logs in a file provides additional flexibility and can be used by the platform to store logs for a longer period. However, this module is easily extendable. Every appender has to inherit from the abstract appender type class. After implementing the required methods and defining the configuration option for the new appender, it can be used by developers without any additional changes.

In order for the module to work correctly, it expects any custom code which runs on the virtualized device to print logs to stdout.

[7]`https://logging.apache.org/log4j/2.x`, last access at 2020-01-16
[8]`https://kafka.apache.org`, last access at 2020-01-16

Network Module

When VIoTDs communicate with each other on the simulation platform, the network simulator described in Section 4.2 simulates the communication channels. The quality of a simulation highly depends on the capability of the simulator to mimic real-world characteristics [5, 10]. Hence, as communication and networking is a major factor in the IoT [5], forcing users to make changes to their network and communication code or requiring modifications to the network stack of virtualized devices would decrease the quality of the simulation.

Therefore, the main goal of this module is compatibility. The network module abstracts the connection to the network simulator. By doing so, any code running on the virtualized device is unaware of the underlying network simulator, and virtualized devices can be used without adjustments. This means that any data sent or received by a virtualized device runs through the network simulator without further intervention. The advantage of this abstraction layer is that any library, framework or tool with communication capabilities running on the virtualized device can be used in the same way as it would be on a real device.

The network module achieves this by using tap interfaces, bridges, and veth pairs. The specific setup depends on the virtualization technology used by the virtualized device. The network simulator assigns the IP addresses for containers and processes, and by a libvirt network for VMs. The reason that VMs get their IP addresses from a libvirt network and not from the network simulator improves the compatibility with the current network simulator. The detailed description can be found in Chapter 5.3.

VIoTD API

The aforementioned modules and functionalities focus on the virtualized device. To enable interaction between the simulation platform and a VIoTD, the VIoTD exposes a REST API to control and manage the device. Appendix A provides the API description of the VIoTD API and gives an overview on available endpoints.

4.1.3 Virtualized Device Configuration

The integration of a virtualized device is done via configuration files. Depending on the virtualization technology of the virtualized device, some additional steps may be necessary.

Independent of the virtualization technology, a VIoTD requires (i) a platform configuration, (ii) a runtime configuration, and (iii) a VIoTD configuration. The following sections describe each of these configurations in detail and with proper examples.

Platform Configuration

The platform configuration provides information about the simulation platform to the VIoTD and is the only configuration that is independent of the virtualized device and the virtualization technology. It depends on the infrastructure setup, and an example is shown in Listing 4.1. This configuration is the same for every VIoTD and maintained by the platform administrator. Hence, no changes are required during the integration process.

Listing 4.1: Example platform configuration yaml file.

```yaml
vm_storage_pool_path: /opt/simpool/devel
device_setup_file_name: preparation.txt
simulation_commands_file_name: simulation.txt
stop_simulation_commands_file_name: stop_simulation.txt
net_namespace_directory: /var/run/netns
program_executable_directory: /opt/
reporting:
    type: file
    log_directory: .
monitoring:
    type: console
```

The configuration options from Listing 4.1 are explained in Table 4.4.

The reporting, as well as the monitoring & logging module, supports several configuration options. The reporting module can be configured to redirect its output to a file, as shown in Listing 4.1, or forward it to a Kafka topic. For the latter, Listing 4.2 shows the proper configuration snippet.

Listing 4.2: Reporting module configuration to forward output to Kafka topic.

```yaml
reporting:
    type: kafka
    host: 127.0.0.1
    port: 9092
```

The monitoring & logging module supports an additional configuration option as well. The configuration in Listing 4.1 uses *console* as type, which forwards any logging output to stdout. Storing the output in a file instead of printing it to stdout is possible with the configuration snippet in Listing 4.3.

Runtime Configuration

The runtime configuration is passed to the VIoTD by the simulation platform via program arguments. The platform is responsible for passing valid values to the VIoTD, and therefore, it is not necessary to specify any parameters during the integration of a

Table 4.4: Explanation of the platform configuration options.

vm__storage__pool__path	The path to the storage pool where VM images are stored. Images are stored temporarily in this pool, i.e., they get created when a VIoTD is started and get removed during the shutdown process. The storage pool is created during the infrastructure setup.
device__setup__file__name	The name of the file which contains the device preparation commands. The content of the file is uploaded by the user and executed on the VIoTD during the *preparation task running* state.
simulation__commands __file__name	Similar to the *device__setup__file__name*, but instead of the preparation commands, it contains the commands for the simulation. The user uploads the file content, and it gets executed when the simulation is started.
stop__simulation__commands __file__name	As the integration layer has no knowledge about the simulation itself and how to stop it, it relies on the user to provide the proper command. If the simulation tasks run in the foreground and interrupting it with *CTRL+C* is possible, adding *CTRL+C* to this file provides the desired effect.
net__namespace__directory	The directory where the host stores the network namespaces. This directory must exist; the VIoTD does not create it.
program__executable__directory	The directory where virtualized devices, which run as a process, are stored on the host.
reporting	Configuration of the reporting module.
monitoring	Configuration of the monitoring & logging module.

Listing 4.3: Monitoring & logging module configuration to store output in a file.

```yaml
reporting:
    type: file
    log_directory: /var/log/viotd
```

Table 4.6: Explanation of the runtime configuration options.

port	**required**	The port the REST API of the integration layer is exposed on.
device-id	**required**	The unique device identifier for this instance.
directory	**required**	The working directory for the VIoTD.
viotd-config-path	**required**	The path to the VIoTD configuration file.
platform-config-path	**required**	The path to the platform configuration file.
ns-registration	**required**	The endpoint of the network simulator to register the VIoTD.
verbose	**optional**	Flag parameter. Turns on verbose logging.
libvirt-network	**optional**	Only required if the virtualized device is a VM. It specifies the libvirt network a VM uses to connect during boot. The VM obtains an IP from this network and gets access to the Internet through it.
kafka-log-topic	**optional**	If the platform is configured to use Kafka to send application logs to the user, the name of the topic, which the VIoTD should send logs to, needs to be specified here.

virtualized device. The runtime configuration properties for the VIoTD are unique for an individual instance, i.e., a running VIoTD in a specific simulation, and can be different in every simulation setup. Most of the properties are independent of the virtualized device and mainly affect the integration layer.

The list of runtime configuration properties and their explanations can be found in Table 4.6.

VIoTD Configuration

The VIoTD configuration consists of virtualized device specific properties. The general properties in Listing 4.4 are independent of the virtualized device type, i.e., if it is a process, container, or VM. The explanation of these general properties is given in Table 4.8.

Listing 4.4: Example general VIoTD configuration.

```yaml
framework_version: v0.1
device:
    device_type: vm
    device_info: >-
      Several lines of text,
      with some "quotes" of various 'types',
      and also a blank line:

      plus another line at the end.
```

Table 4.8: Explanation of the general VIoTD configuration options.

framework_version	The version of the architecture framework the configuration was created for.
device.device_type	The type of the device. Can be either process, container or vm.
device.device_info	General information about the device. Provided by the developer of the virtualized device.

Besides that, most of the configuration depends on the device type of the virtualized device. The following paragraphs provide an example for each type and further details about device-specific configuration properties.

VM Configuration The VIoTD uses libvirt to manage VMs and, therefore, requires a valid libvirt XML configuration. This XML configuration is part of the VIoTD configuration. The example in Listing 4.5 shows a shortened libvirt XML configuration, as it is VM specific, and recommended to generate it for each VM individually.

The description of the configuration properties in Listing 4.5 can be found in Table 4.10.

Listing 4.5: Example VM VIoTD configuration.

```
framework_version: v0.1
device:
    device_type: vm
    device_info: >-
      Several lines of text,
      with some "quotes" of various 'types',
      and also a blank line:

      plus another line at the end.
    # VM specific config
    template_vm_image_path: /opt/imgs/debian-7.qcow2
    vm_force_stop_time: 60
    hypervisor_uri: qemu:///system
    boot_finished_msg: "debian login: "
    vm_idle_msg: "root@debian:~# "
    login:
        username: root
        password: root
    xml_template_str: >
      <domain type='qemu'>
        ....
      </domain>
```

Table 4.10: Explanation of the VM VIoTD configuration options.

template_vm_image_path	The path to the template (original) VM image. The path includes the image name.
vm_force_stop_time	The time the integration layer should wait during the shutdown process before the VM is shut down forcefully.
hypervisor_uri	The URI of the hypervisor libvirt should connect to and run the VM on.
vm_idle_msg	The message shown by the VM on the serial console when the VM is idle.
boot_finished_msg	The message shown by the VM on the serial console when the boot process has finished. If login is disabled on the VM, the same message used for *vm_idle_msg* should be used.
login.username	**Optional.** If the VM is login protected, the integration layer needs a user to login.
login.password	**Optional.** If the VM is login protected, the integration layer needs the password for the specified user.
xml_template_str	The libvirt XML configuration the integration layer uses to manage the VM.

Container Configuration The VIoTD configuration for the container is simple and does not need extensive explanations. The example configuration in Listing 4.6 and the description in Table 4.12 provide sufficient information.

Listing 4.6: Example container VIoTD configuration.

```
framework_version: v0.1 # current version of the framework
device:
    device_type: container
    device_info: >-
      Several lines of text,
      with some "quotes" of various 'types',
      and also a blank line:

      plus another line at the end.
    # Container specific config
    container_name: "ubuntu:18.04"
    docker_uri: "unix://var/run/docker.sock"
    max_memory: 256m
```

Table 4.12: Explanation of the container VIoTD configuration options.

container_name	The name of the container image.
docker_uri	The URI to the local Docker daemon.
max_memory	The maximum memory a container is allowed to consume.

Process Configuration Similar to the container, the process requires little configuration. The example configuration in Listing 4.7 is described in Table 4.14.

Listing 4.7: Example process VIoTD configuration.

```yaml
framework_version: v0.1
device:
    device_type: process
    device_info: >-
      Several lines of text,
      with some "quotes" of various 'types',
      and also a blank line:

      plus another line at the end.
    # Process specific config
    program_execution_file: "testprocess.py"
    start_command: "python3"
```

Table 4.14: Explanation of the process VIoTD configuration options.

start_command	The command used to start the program. This can be an interpreter or a language-specific runtime. If the program is, for example, a binary and does not require any command, this option can be left empty.
program_execution_file	The name of the file to execute. The host machine stores the program in a known directory. Therefore, this parameter requires the filename, not a path.

The VIoTD configuration file allows developers to integrate new virtualized devices into the simulation platform without making changes to the virtualized device itself. Besides the configuration file, additional steps may be necessary before using the VIoTD in a simulation. Therefore, Section 5.2 provides additional information about the integration process of virtualized devices

4.2 Network Simulator

The network simulator creates the network topology for a simulation. While the architecture framework's responsibility is, among others, to provide a proper abstraction of the network capabilities of a virtualized device, the network simulator uses this well-defined interface to connect VIoTDs and to simulate different communication models and protocols.

The network simulator consists of the network service and ns-3[9]. The network service exposes a REST API to control the network simulator, and ns-3 simulates communication channels between VIoTDs to mimic real-world characteristics for data transmission. To simplify the setup process of the network simulator, a Docker image was created and published to Dockerhub[10], which includes the network service and ns-3. Providing a Docker image has several advantages, like running multiple simulations in parallel on a single host and improving the scalability of the simulation platform in the future.

4.2.1 Network Simulation

When a VIoTD gets placed into a new simulation, it registers itself at the network simulator. The registration request includes the name of the network interface the VIoTD uses. This informs the network simulator that the VIoTD wants to participate in any future simulation and allows the network simulator to include the network interface of the VIoTD in the simulation. Besides the registration process, VIoTDs have to unregister as well.

The network simulator creates network topologies at runtime. It uses a JSON format to represent network nodes, links, and additional network-specific configuration options. To provide this flexibility, a new ns-3 module was developed that interprets the submitted JSON and builds the corresponding network topology dynamically. The description of the JSON schema is provided in Section 4.2.2. The developed ns-3 module supports CSMA links, point-to-point connections, WiFi access points (AP), WiFi Adhoc networks, and mesh topologies according to the IEEE 802.11s standard[11].

The ns-3 implementation of CSMA is similar to Ethernet in the real-world. Although it operates rather with a collision avoidance than collision detection approach, it is the recommended module to simulate Ethernet links[11]. The point-to-point link simulates a full-duplex RS-232 or RS-422 link, which connects precisely two devices[11]. The simulation of WiFi networks allows infrastructure and ad-hoc modes and uses a model based on the IEEE 802.11 standard[12]. The official ns-3 documentation provides a detailed

[9] `https://www.nsnam.org`, last access at 2020-01-16

[10] `https://hub.docker.com/repository/docker/mle110/ns`, last access at 2020-01-17

[11] `https://www.nsnam.org/docs/release/3.29/models/html`, last access at 2020-01-17

[12] `https://www.nsnam.org/docs/release/3.29/models/html/wifi-references.html`, last access at 2020-01-17

description of the characteristics and restrictions of the WiFi implementation[13]. The mesh implementation of ns-3 extends the WiFi module and adds mesh networking capabilities based on the IEEE 802.11s standard[11]. The ns-3 Mesh module implements only a subset of 802.11s, which restricts its usability. The current model misses a mesh AP and a mesh portal and has compatibility issues with other ns-3 modules. To simulate network topologies which connect mesh networks with CSMA, peer-to-peer, or WiFi nodes, modeling the mesh network in ad-hoc WiFi mode is an alternative solution and supported by the network simulator.

4.2.2 Network Topology JSON

As previously mentioned, the network topologies for simulations are defined in a JSON format. The following paragraphs specify the JSON format and describe possible configuration options. Listing 4.8 shows the JSON schema and possible values for each parameter.

Listing 4.8: JSON format for defining network topologies.

```
{
  "devices": [{
    "device_id": "string",
    "type": "[vm|container|process] | switch | router | ap",
    "tap_if_name": "string",
    "xpos": float,
    "ypos": float,
    "zpos": float
  }, { ...
  }],
  "network": [{
    "network_type": "ADHOC | AP_STA | CSMA | P2P | MESH",
    "general_config": {
      "key": "value",
      ...
    },
    "address": {
      "ip": "string",
      "netmask": "string"
    },
    "devices": [ "<device_id>", ... ]
  }]
}
```

[13]https://www.nsnam.org/docs/release/3.29/models/html/wifi-design.html, last access at 2020-01-17

The devices list contains every device participating in the simulation, and each JSON object in the list corresponds to a single device. Table 4.16 explains this JSON object in more detail.

Table 4.16: Explanation of the device JSON object.

device_id	The device ID is mandatory and uniquely identifies a device in the simulation.
type	If the device is a VIoTD, the user does not have to submit this field. The network service is aware of each VIoTD in the simulation and enriches the JSON with the device type. On the contrary, the values bridge, router, and ap refer to nodes created by ns-3 during the simulation. If the topology requires one of these nodes in the simulation, the type field is mandatory.
tap_if_name	This field specifies the name of the network interface used by the VIoTD. The user does not have to submit this field, as the network service is aware of the network interface of the VIoTD and adds this field automatically before submitting the JSON to the ns-3 module.
xpos, ypos, zpos	These values refer to a position in a three dimensional coordinate system. They are only required for nodes participating in a wireless network.

The second field in the network topology JSON, called *network*, specifies the links and connections between nodes. Each JSON object in the list corresponds to a single link or network with the values specified in Table 4.18.

This modular approach allows the specification of complex network topologies. Appendix C provides example JSON representations for basic network structures.

4.2.3 Network Management

Besides simulated networks, the network service creates and manages networks used by VM VIoTDs. In contrast to processes and containers, the integration layer cannot manually set IP addresses or add routing rules to a VIoTD if the virtualized device is a VM. Hence, it is necessary to connect VMs to libvirt networks with a properly configured DHCP server. The networks created by the network service are in NAT mode to provide Internet access to VMs. This requires additional management, as libvirt creates IP table rules in NAT mode, which impacts a VIoTD during a simulation. The network simulator manages the configuration of libvirt networks, their lifecycle, and updates IP table rules to avoid any unexpected behavior during simulations.

Table 4.18: Explanation of the network JSON object.

network_type	The network type refers to the ns-3 module used to simulate this link or network. While CSMA, P2P (point-to-point), and MESH refer to the previously explained ns-3 modules, the values ADHOC and AP_STA refer to WiFi networks in ad-hoc and infrastructure mode, respectively.
general_config	The general configuration contains key-value pairs specific to the network type. For CSMA and P2P, supported options are the data rate (“data_rate”) and the link delay (“delay”) of the connection. If the network type is ADHOC or AP_STA, the WiFi SSID (“ssid”) needs to be specified. Networks with type MESH do not expect any configuration. The examples in Appendix C show possible values for these configuration options.
address	The address object specifies the IP subnet of the link or network. This field is optional, as links between switches do not need an IP address.
devices	This list specifies the devices which participate in the link or network. Each list entry references a device from the previously defined devices object.

4.2.4 API Description

The API of the network service provides REST endpoints to interact with the network simulator. Appendix B offers a list of all available endpoints and their functionality.

4.3 Simulation Workflow

The previous chapters describe the concept of VIoTDs and characterize the network simulator. Combining both components allows the simulation of IoT systems. The following paragraph provides a system overview and demonstrates the interaction between VIoTDs and the network simulator.

Figure 4.4 demonstrates a simple simulation setup, consisting of three VIoTDs and the network simulator, and includes the communication flow between the components.

The communication flows and the API endpoints in Figure 4.4 are numbered, indicating the order in which they appear. Besides steps 0 and 1, the order of steps 2 to 7 is not strictly enforced and may change depending on the simulation setup. The following steps description provides a detailed explanation.

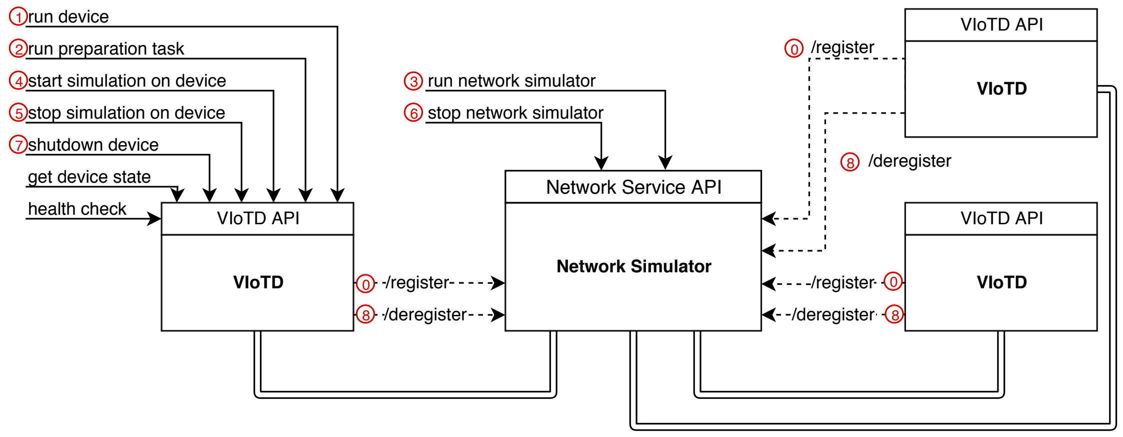


Figure 4.4: Example of an IoT simulation with VIoTDs and the network simulator.

Step 0 The VIoTD registers itself automatically when it is placed in the simulation, and no user interaction is required.

Step 1 Placing a VIoTD in a simulation implies that the integration layer is running, but not the virtualized device. To enable further interaction with the VIoTD, it is necessary to run it first.

Step 2 The preparation task is optional. It allows the user to put the virtualized device in a well-defined state before running the simulation code. It is possible to run another preparation task after a simulation has finished.

Step 3 This is the first step to run a simulation. The network simulator starts ns-3 and creates the network channels for the VIoTDs. It is necessary to start the simulation on the network simulator before calling the simulation endpoint on any VIoTD.

Step 4 After running the network simulator, it is possible to start the simulation on the VIoTDs. This step executes previously uploaded code or commands, and reports log output back to the user.

Step 5 The running simulation task on the VIoTD can be stopped by calling the stop endpoint of the VIoTD API. Besides terminating the simulation program, it executes additional cleanup tasks and prepares the VIoTD for further interactions, i.e., preparation or simulation tasks.

Step 6 After stopping the simulation on VIoTD devices, it is necessary to stop the simulation on the network simulator. Besides stopping ns-3, it restores the libvirt networks and creates a clean setup for further simulations.

Step 7 After the user does not want to execute further simulation tasks, the next step is to shutdown the VIoTD. This action turns off the VIoTD and performs necessary cleanup tasks.

Step 8 Similar to the registration step, the VIoTD deregisters itself automatically. After shutting down the virtualized device, the VIoTD sends the deregistration request to the network simulator and exits.

4.4 Raft

The simulation workflow shown in the previous chapter describes how the architecture framework and the network simulator enable end-to-end IoT simulations. This simulation infrastructure allows the evaluation and experimentation of real-world IoT scenarios. This thesis evaluates the Raft consensus algorithm in a wireless IoT setup and utilizes the architecture framework and the network simulator. Hence, the following chapter starts with an introduction to the Raft consensus algorithm and summarizes its characteristics. It then elaborates implementation-specific decisions and illustrates efforts to mimic real-world characteristics for the IoT setup.

4.4.1 The Raft consensus algorithm

Raft is a consensus algorithm developed by Diego Ongaro and John Ousterhout [20] at the University of Stanford. Their goal was to develop a consensus algorithm with better properties than Paxos [136]. Paxos is a very popular protocol for finding consensus in distributed systems, taught in many courses at universities, and is a starting point for many consensus implementations. Besides its popularity, Paxos has considerable drawbacks, as pointed out in [20]. Raft's design facilitates understandability and tries to address the shortcomings of Paxos by reducing the state space of nodes as well as by decomposition, i.e., separating problems into mostly independent parts. In Raft, these subproblems are leader election, log replication, safety, and membership changes [20].

This part of the thesis focuses on consensus in IoT setups. Raft addresses the consensus problem by leader election, log replication, and safety. The following section summarizes these subproblems and describes how Raft implements consensus. An extensive description of Raft is provided by [20] and [130], and the official github page[14] provides an interactive visualization as well as additional sources for understanding Raft.

[14]`https://raft.github.io`, last access at 2020-01-17

General Concepts

At any given time, each server participating in a Raft cluster is either in the leader, candidate, or follower state. Depending on this state, it has certain responsibilities and executes state-specific tasks:

- **Leader**: In a Raft cluster, there is at most one leader. It handles all requests from clients, replicates logs to other servers, and commits logs after replicating them to a majority of servers (under certain restrictions, see Section 4.4.1).
- **Candidate**: Servers may change to the candidate state when a new leader election starts. After a successful leader election, servers change to follower or leader.
- **Follower**: In the initial state of the Raft cluster, each server is in the follower state. Followers are passive servers in the cluster. They wait for logs from the leader and reply to leader requests only. Hence, followers do not initiate any communication.

Raft works in so-called terms, i.e., a logical clock that increases monotonically over time, to be aware of more recent events, and to detect stale servers. A term has arbitrary length and starts with a leader election. In a successful election, the term remains the same until a new election gets initiated. If the election was not successful, a new term with another election starts.

The servers in the Raft cluster communicate via RPCs. The consensus algorithm requires only two different types of RPCs, namely RequestVote and AppendEntries. The leader issues any RPC calls in parallel for performance reasons. Besides candidates during leader election, the communication flow between Raft servers happens only in one direction, i.e., from the leader to the followers.

As previously mentioned, each term starts with a leader election. Hence, when the first term starts, the Raft cluster elects a new leader. The following section describes this process in more detail.

Leader election

The Raft algorithm enables consensus via a strong leader. The properties of the algorithm ensure that leader election is possible as long as a majority of servers are functioning.

As previously mentioned, each server is initially in the follower state and waits for a randomized period of time, the so-called election timeout. After the election timeout, a server starts an election by incrementing its current term and transitioning into the candidate state. It then votes for itself, sends RequestVote RPCs to the other servers in the cluster, and waits for responses. If a server receives a RequestVote RPC, it issues exactly one vote in one term on a first-come-first-serve basis.

Candidates keep their state until one of the following events occur:

- **The candidate wins the election.** This happens when the candidate receives a positive response from a majority of servers in the cluster. After winning the election, the candidate transitions into the leader state and starts sending heartbeat messages periodically to other servers. The heartbeat messages inform followers about the active cluster leader and prevent another election.
- **The candidate receives a heartbeat message.** If the term in the heartbeat message is equal or bigger than the candidates' term, the candidate has lost the election and changes its state to follower. If the heartbeat message includes an outdated term, the candidate rejects the message and keeps its state.
- **The election has no winner.** If many servers become candidates simultaneously, the election ends with a split vote, i.e., no candidate is able to obtain a majority of votes. In this case, the election timeout elapses again, and another election starts.

In the worst case, an election results in a split vote. The randomized election timeout helps in keeping split votes rare and resolving them quickly. Further, the servers' term is part of every message, including responses. This allows the cluster to detect stale leaders and to prevent outdated candidates from being elected. If, for example, a leader or a candidate receives a message with a higher term, they change their state to follower. Hence, a candidate stops participating in the current election and when a leader steps down, a new election starts automatically after the first server times out.

Log replication

Every server in the Raft cluster stores a sequence of log entries, i.e. commands. These logs can then be fed into, for example, replicated state machines to maintain a consistent state over all nodes participating in the cluster. The consensus algorithm enables consistent distribution of such logs across the cluster and allows state machines to keep identical copies.

The Raft algorithm uses a strong leader to facilitate consensus. Hence, only the leader processes client requests. For every client request, the leader appends a new log entry to its logs, where each log entry consists of the request data and the current term. It then propagates the log entry in parallel to each server in the cluster via AppendEntries RPCs. If the data transmission was unsuccessful, e.g. due to lost packets, the leader retries indefinitely. In addition to the new log entry, an AppendEntries RPC contains information about the previous log stored by the leader. When a follower receives a log entry from the leader, it checks if the term of the leader is up-to-date and whether its last log entry is consistent with the previous log of the leader. If so, the follower adds the new log entry to its logs and issues a successful response to the leader. Otherwise, the follower informs the leader about its outdated logs. If the leader becomes aware of a follower with outdated logs, it starts sending older logs to the follower to find the first mutual log entry, so that the leader can replicate its logs to the follower in the correct order. If a log entry at the follower conflicts with the new log entry, the follower deletes the conflicting log entry and all that follow it, and adds the new entry to its logs. Hence, the leader forces followers to duplicate its logs.

Once a new log entry got replicated on a majority of servers, the leader considers it as committed, i.e. it is save to apply the log to the state machine. The leader keeps track of committed logs with a commit index. The commit index is part of the AppendEntries RPC to maintain a consistent replicated state machine across the cluster.

Safety

The described characteristics for leader election and log replication are not sufficient to ensure consistency in a Raft cluster under all circumstances. Hence, additional restrictions are necessary. For leader election, Raft prevents a candidate from being elected if its log does not contain all committed entries. While other consensus algorithms allow leaders to update themselves after being elected, Raft enforces data flowing from leaders to followers only to improve the understandability of the algorithm.

Another restriction deals with log replication. If a leader crashes before it is able to replicate a log on a majority of servers, i.e., before it is able to commit a log, a future leader may continue replicating the log. Nonetheless, leaders do not commit log entries from previous terms. There exist several edge cases where this could lead to an issue, as shown in [20, 130]. If the leader replicates new log entries, old logs get committed automatically once the new entries got replicated on a majority of servers.

Formal proof of Raft is provided in [130].

4.4.2 IoT Simulation: Raft Implementation

For the use-case simulation, the previously described Raft algorithm gets implemented and evaluated in a simulated IoT system. The IoT setup for the use-case simulation consists of virtualized Raspberry Pis connected via a wireless mesh network. Therefore, a simple Raspberry Pi emulator got developed and integrated with the architecture framework. The following section starts with describing the developed Raspberry Pi emulator and concludes with details and design decisions of the Raft implementation.

Raspberry Pi Emulator

The Raspberry Pi emulator runs on a 64-bit Raspbian[15] ARM image. For the simulation to be resource-efficient, the emulator facilitates lightweight virtualization by running inside a Docker container. The virtual Raspberry Pi does not emulate any peripherals or sensors, the goal of the emulator was rather to explore processor emulation, e.g., running an ARM image on an x86 host machine, and how the architecture framework and the end-to-end simulation setup integrates with it.

[15] `https://www.raspberrypi.org/downloads/raspbian`, last access at 2020-01-20

To run the ARM image on a host with x86 architecture, the emulator utilizes Qemu[16]. More specifically, it uses the user mode emulation binaries of Qemu. In this mode, Qemu is able to launch Linux processes that were originally compiled for a CPU different from the CPU of the host machine.

When the Docker container of the Raspberry Pi emulator starts, the image tries to run its binaries on the host. In this step, the Linux kernel of the host OS must be aware of the Qemu user-mode emulation. Otherwise, it cannot run ARM binaries. The binfmt[17] feature of the Linux kernel allows the registration of user space programs in the OS kernel. This tells the kernel which interpreter to invoke for which binaries. During the binfmt registration process, it is necessary to provide a magic byte sequence for the interpreter or user space program. When a binary gets executed on the host OS, binfmt recognizes the binary-type by matching some bytes at the beginning of the binary with the registered byte magic. The registration script for the Raspberry Pi emulator is in Appendix D. This script registers the Qemu user mode emulator for 32 and 64-bit binaries and builds upon existing solutions[18].

Utilizing Qemu and binfmt allows running ARM Docker containers on an x86 host. The last step for the Raspberry Pi emulator is to build the actual Docker image. Since Raspbian is based on Debian Linux[19], utilizing the debootstrap tool[20] to bootstrap the base image simplifies the process. To reduce the image size, unnecessary components like manuals got removed before creating the Raspberry Pi emulator image with the Docker import tool[21]. The script for creating the Raspberry Pi emulator is provided in Appendix D as well.

Raft Implementation

The Raft algorithm for the use-case scenario is implemented in Java 11 and follows exactly the description provided in [20]. As previously mentioned, the implementation includes leader election and log replication while guaranteeing the safety argument.

As defined in [20], Raft servers communicate via RPCs. Many RPC frameworks support Java, and while frameworks like gRPC[22] gained popularity in recent years, the Raft implementation uses Apache Thrift[23]. The advantage of Thrift over, for example, gRPC, is that Thrift operates from the transport layer downwards [137], while gRPC uses HTTP/2 for communication[24]. Hence, Thrift packets do not include HTTP headers,

[16]`https://www.qemu.org`, last access at 2020-01-20
[17]`https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html`, last access at 2020-01-20
[18]`https://hub.docker.com/r/multiarch/qemu-user-static`, last access at 2020-01-20
[19]`https://www.debian.org`, last access at 2020-01-20
[20]`https://linux.die.net/man/8/debootstrap`, last access at 2020-01-20
[21]`https://docs.docker.com/engine/reference/commandline`, last access at 2020-01-20
[22]`https://grpc.io`, last access at 2020-01-21
[23]`https://thrift.apache.org`, last access at 2020-01-21
[24]`https://grpc.io/docs`, last access at 2020-01-21

which reduces the overhead during data transmission. Considering IoT setups with limited power supply, restricted bandwidth, and lossy wireless communication channels, reducing data transmission is beneficial for the system.

In many use-cases, consensus algorithms assist in keeping a consistent state across servers as, for example, in the context of replicated state machines [20]. Hence, a server receives input from a client or the environment, and the consensus algorithm replicates the input across the cluster. The use-case simulation in this thesis focuses on Raft, and therefore, neither clients nor any environment gets simulated. To mock clients and applications interacting with the Raft cluster, each server exposes a REST API to start and stop the Raft process, to send inputs (logs) to the cluster, and to get a snapshot of the current server state. The Raft implementation uses Micronaut[25] to run an HTTP server for exposing the REST API. Compared to the popular SpringBoot[26] framework, Micronaut provides similar functionality while requiring less resources[27]. This keeps the Raft implementation more resource-efficient without having any negative impact on the application itself.

Use-Case Simulation

After creating the Raspberry Pi emulator and implementing the Raft algorithm, it is possible to simulate the use-case. The Raspberry Pi emulator gets integrated into the simulation setup with the architecture framework. Listing 4.9 shows the corresponding VIoTD configuration file that integrates the Raspberry Pi emulator into the simulation. After registering the Qemu user-mode binaries as previously described, the emulator can participate in the simulation.

Listing 4.9: VIoTD Configuration for the Raspberry Pi emulator.

```yaml
framework_version: v0.1
device:
    device_type: container
    device_info: >-
      Raspberry Pi emulator based on
      Raspbian Buster.
    container_name: "mle110/raspbian:buster"
    docker_uri: "unix://var/run/docker.sock"
    max_memory: 1024m
```

It is necessary to copy the Raft jar into the working directory created automatically by the integration layer of the Raspberry Pi VIoTD to run the Raft algorithm on the

[25]`https://micronaut.io`, last access at 2020-01-21
[26]`https://spring.io/projects/spring-boot`, last access at 2020-01-21
[27]`https://jaxlondon.com/blog/cloud-native-java-with-micronaut-an-alternative-to-spring`, last access at 2020-01-21

emulated Raspberry Pi. Also, it is necessary to specify how the integration layer can start and stop the simulation. Therefore, Listing 4.10 shows the content of the start simulation file and Listing 4.11 provides the content of the stop simulation file.

Listing 4.10: Content of the start simulation file to start the Raft algorithm on the Raspberry Pi emulator.

```
java -Draft.serverid=server1 -Draft.port=10000
-Dmicronaut.server.port=11000
-Draft.cluster-properties-path=/workdir/cluster.properties -jar
/workdir/viotd-raft-1.0-SNAPSHOT.jar
```

Listing 4.11: Content of the stop simulation file to stop the Raft algorithm on the Raspberry Pi emulator.

```
CTRL+C
```

CHAPTER 5

Implementation & Infrastructure Setup

The previous chapter focuses on the characteristics of the system and provides information about design decisions and system behavior. This chapter explains the required infrastructure setup, the integration of new virtualized devices, implementation details, and technologies the system uses.

5.1 Infrastructure Setup

Infrastructure in this context refers to the VM or host machine where the IoT simulation is running. Due to OS-specific components used by the architecture framework and the network simulator, the simulation environment only runs on Linux systems.

Both the architecture framework and the network simulator have dependencies that require infrastructure provisioning before running any simulation. Besides, VIoTDs may require additional infrastructure setup that depends on the virtualized device, e.g., if the device is implemented as a process that requires a specific runtime. Table 5.1 shows the minimum set of dependencies needed by the architecture framework and the network simulator.

Besides the dependencies in Table 5.1, the hypervisors used by VM VIoTDs need to be installed as well. To enable running VM VIoTDs, it is necessary to create hypervisor-specific storage pools where VM images can be stored.

The following example uses the qemu hypervisor:

```
virsh -c qemu:///system pool-define-as devel dir
                            --target /opt/testpool/devel
virsh -c qemu:///system pool-autostart devel
virsh -c qemu:///system pool-start devel
```

These commands create a new pool *devel* in the */opt/testpool* directory for the Qemu hypervisor. After these steps, a new host is prepared to run VIoTDs with VMs.

Table 5.1: Required dependencies.

Dependency	**Min. Version**	**Description**
Docker	19.03	Container runtime required by the network simulator and container VIoTDs.
libvirt-dev	5.10.0	Virtualization API for running VM VIoTDs.
iproute2	4.15.0	Tool which unifies network interface configuration.
python	3.6	Python runtime.
pip	19.3.1	Python package manager.
requirements.txt	-	Python dependencies of the architecture framework. The requirements file is part of the repository and can be installed via pip.

Besides the libvirt setup, it is necessary to create a directory where programs and source code for processes get stored. This is a plain directory and does not require further configuration.

After the installation of the components in Table 5.1 and considering setup specific modifications, the infrastructure can be used to run simulations. The following section describes the integration process of virtualized devices and provides additional insights into possible implications for the infrastructure.

5.2 Virtualized Device Integration

Independent of whether the virtualized device is a process, a container, or a VM, creating a VIoTD configuration file is mandatory. Section 4.1.3 provides the required information to create a valid VIoTD configuration, and the following paragraphs explain additional steps and important remarks.

Listing 5.1: Provision a VM image using virsh.

```
sudo virt-install --connect qemu:///system --name test0
--memory 512 --vcpus=1 --import --os-variant debian7 --disk
vol=devel/debian-7.qcow2,device=disk,format=qcow2
virsh dumpxml test0
```

5.2.1 VM Integration

During the integration process of a VM, it is necessary to interact with the libvirt daemon on the host machine. The libvirt installation includes a command-line tool called *virsh* for performing management tasks for libvirt and will be used throughout the VM integration process.

The architecture framework requires the host machine to store the VM image in one of the libvirt storage pools. To avoid undesirable changes, the VIoTD creates a copy of the VM image before it uses it in any simulation. Therefore, the VM image integrated in this step can be seen as a template. The following commands

```
sudo cp debian-7.qcow2 /opt/testpool/devel/
sudo virsh -c qemu:///system pool-refresh devel
```

copy the image "debian-7.qcow2" into the "testpool" storage pool. It is important, that the storage pool already exists. The official virsh documentation[1] and Section 5.1 provide additional information on managing storage pools.

Next, the VIoTD configuration file requires a libvirt XML configuration of the VM image. There are different approaches to create the XML configuration, and the official documentation[2] provides an in-depth explanation about possible options. Listing 5.1 shows one possible approach using virt-manager[3]. This command provisions the VM image and creates a libvirt XML configuration with the name "test0", using the Qemu hypervisor, with 512MB memory, based on the debian7 image which is stored in the "qcow2" format in the pool from the previous step.

Before copying the XML configuration into the VIoTD configuration file, the lines shown in Listing 5.2 need to be removed.

After removing these lines, the XML can be copied into the VIoTD configuration file, and the VM integration process is done.

[1] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-managing_guest_virtual_machines_with_virsh-storage_pool_commands, last access at 2020-01-23

[2] https://libvirt.org/formatdomain.html, last access at 2020-01-23

[3] https://virt-manager.org, last access at 2020-01-23

Listing 5.2: Lines to be removed from libvirt XML configuration.

```
<name>test0</name>
<uuid>...</uuid>
```

5.2.2 Container Integration

The container integration requires the Docker daemon of the host to have access to the Docker image. The recommended solution is to push the image to a container registry where the Docker daemon has access to. The container name in the VIoTD configuration must match the name and the version of the Docker image. Besides that, no further actions are necessary.

5.2.3 Process Integration

During the Infrastructure Setup described in Section 5.1, a directory to store integrated programs gets created. The VIoTD can only work with programs stored in this directory, and therefore, the new program needs to reside in this directory as well. As the VIoTD does not install any program dependencies, it is necessary to install them manually. After creating the VIoTD configuration, the integration is done.

5.3 Network Layer

In a broader sense, the network module of the VIoTD and the network simulator represent the network layer of the simulation. Individual design decisions in the network module and the characteristics of ns-3 influence the implementation of the network layer. This section summarizes the most important implications and their effect on the implementation.

Due to the importance of communication characteristics in IoT simulations [5], the network module has to abstract the network stack of the virtualized device without affecting it in any way. Hence, the virtualized device should be unaware of the underlying network layer.

Furthermore, the design of ns-3 imposes additional restrictions on the network layer. ns-3 has the concept of a *Tap NetDevice*[4], which allows ns-3 simulations to interface with external systems like VMs or the host machine using tap interfaces. The *Tap NetDevice* reads incoming traffic from the tap interface and forwards it to the simulated network. In addition, it sends traffic arriving from the network simulation to the tap interface.

The operating mode of the *Tap NetDevice* affects the tap interface and its compatibility with other ns-3 devices, which are required to simulate wired or wireless networks. The

[4] `https://www.nsnam.org/docs/release/3.30/models/html/tap.html`, last access at 2020-01-23

following is a summary of the three operating modes supported by ns-3. The official ns-3 documentation for a *Tap NetDevice*[4] provides the detailed description.

- **ConfigureLocal Mode** The ns-3 simulation creates the tap interface on the Linux host and assigns IP and MAC addresses automatically. ns-3 permits the configuration of the IP and MAC addresses of a tap interface in the source code of the simulation. This mode is compatible with any other net devices, as ns-3 has full control over the tap interface.
- **UseLocal Mode** This mode requires an existing tap interface. The user or a third-party application is responsible for creating and configuring the tap interface. ns-3 spoofs the MAC address of arriving packets. Therefore, only a single device is allowed to use the tap interface to enable compatibility with other net devices.
- **UseBridge Mode** Similar to the UseLocal mode, ns-3 assumes an existing and configured tap interface. In this mode, the tap interface is attached to a bridge on the host and logically extends the bridge into ns-3. This mode is useful to integrate virtual hosts configured by another system. The flexibility of this mode reduces its compatibility. Hence, net devices must support the *SendFrom()* method of the ns-3 device interface to be compatible with the *UseBridge* mode.

Based on this information, the preferred operation mode of a ns-3 Tap NetDevice for the network layer is the *ConfigureLocal* mode. It is compatible with any ns-3 net device and reduces the complexity of the network layer, as the tap interface is configured automatically without additional logic in the network layer.

Considering the desired behavior of the network module and the characteristics of the ns-3 Tap NetDevice, the main objective of the network layer is to maximize its simulation capability, i.e., to support as many protocols and ns-3 devices as possible. Due to these reasons, each device type requires an individual network setup. The following paragraphs summarize the implementation details for each device type.

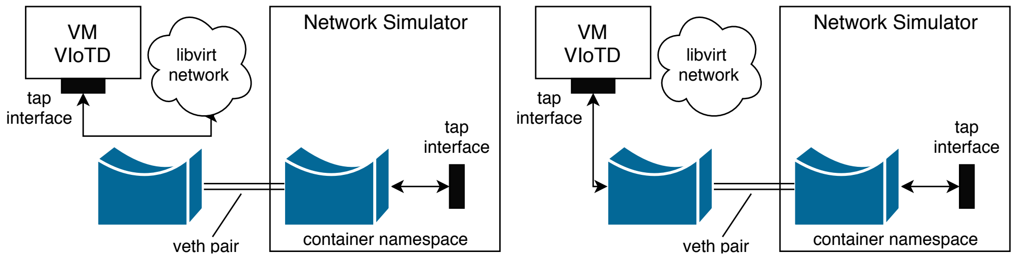
5.3.1 VM Network

The VM network has several challenges. Although libvirt supports tap interfaces, once the VM allocates the interface, ns-3 cannot use it and vice-versa. Hence, the *ConfigureLocal* mode is not applicable. Based on the ns-3 documentation, the VM network uses the *UseBridge* mode for the ns-3 simulation.

Using the *UseBridge* mode requires to (i) manually set up the network devices on the host and (ii) to provide an IP address to the VM. To solve (i), in addition to the network setup shown in the ns-3 documentation[5], the VM network requires further network devices

[5]`https://www.nsnam.org/docs/release/3.30/models/html/tap.html#tapbridge-usebridge-mode`, last access at 2020-01-23

as ns-3 runs in a Docker container and therefore, uses a different network namespace. To connect the VM to a ns-3 simulation, the network layer uses a combination of tap interfaces, bridges, and veth pairs. The solution for (ii) requires a libvirt network. The reason is that the network module should not impose changes to the VM. Hence, it is not a viable solution to configure a static IP within the VM. In addition, libvirt does not support static IPs for VMs, as some hypervisors do not provide this functionality. Therefore, the VM receives its IP from a DHCP server running in the libvirt network. The network module is responsible for the VM network, and Figure 5.1 shows the different setups.



(a) VM network after running the VIoTD. (b) VM network during the simulation.

Figure 5.1: The VM network setup.

The network setup in Figure 5.1a shows the VM network in its initial state. The VIoTD API responsible for running the virtualized device creates both bridges, the veth pair, and the tap interfaces.

Before executing the simulation code on the VM, the network module of the VIoTD moves the tap interface from the libvirt network to the host bridge to connect the VM to the simulated network. Figure 5.1b shows the network setup when the simulation is running. During the simulation, ns-3 uses the tap interface in the container namespace.

5.3.2 Container Network

Container virtualization with Docker uses Linux namespaces[6] to isolate programs running inside of containers. This characteristic allows the network layer to use ns-3 Net TapDevices in *ConfigureLocal* mode.

When the simulation starts, the network module moves the tap interface from the network simulator namespace to the network namespace of the container. As ns-3 sets the IP address of the tap interface, the container is unaware of any IP subnets in the simulation besides the one assigned to the tap interface. The integration layer configures static routes in the network namespace of the container to solve this issue.

[6]`https://docs.docker.com/engine/docker-overview`, last access at 2020-01-23

Any application running inside the container is able to use the tap interface as a network interface to communicate with the simulated network. The tap interface can be treated like any other network interface and does not make any difference to the application.

5.3.3 Process Network

The network setup for Process VIoTDs is similar to containers. The network module creates a network namespace to isolate the process from the host network. This allows the network layer to use ns-3 Tap NetDevices in *ConfigureLocal* mode as well. When the simulation starts, the network module moves the tap interface from the network simulator namespace to the network namespace of the process. In addition, the configuration of static routes allows the process running within the network namespace to reach different IP subnets in the simulation. This allows the process to communicate on the simulated network.

CHAPTER 6

Evaluation

The following chapter describes the evaluation process of the architecture framework, the network simulator, and the use-case scenario, i.e., the Raft algorithm. The architecture framework and the network simulator enable end-to-end IoT simulations. Therefore, Section 6.1 assesses the functional capabilities of both components and discusses implementation decisions with their benefits and drawbacks. Further, Section 6.2 evaluates the Raft algorithm and focuses on the non-functional characteristics of the system.

6.1 Architecture Framework & Network Simulator

The functional evaluation assesses the capabilities of the system with a simple use-case scenario. Section 6.1.1 describes the use-case, defines the VIoTDs participating in the simulation, and specifies the simulated network topology. Based on the user stories from Section 4.1.1, Section 6.1.2 describes the simulation process, matches each step with its corresponding user story and Section 6.1.3 discusses benefits and shortcomings of the proposed solution.

6.1.1 Use-Case Setup

The use-case of the functional evaluation scenario could be part of a smart city application. It consists of a smart temperature sensor that provides the current temperature at the sensors' position to a controller node, i.e., an edge node. The controller collects temperature data, and if the temperature exceeds a certain threshold, it notifies an edge server. The edge server can then react to the notification and execute further steps. Environment monitoring is a common IoT application in smart cities and necessary for weather monitoring and automation services [138].

The use-case scenario consists of the following components:

1. **Sensor** The virtual temperature sensor is a python script and virtualized as a process in the simulation. Upon request, the sensor reads one line of an input data file and replies with the corresponding value. The simulation uses openly available IoT data from the Beach Weather Stations[1] provided by the City of Chicago. This collection consists of sensor data from many different stations. The sensor in the evaluation scenario only uses data from the Oak Street Weather Station.

2. **Controller** The controller polls temperature data from the sensor and notifies the server if the temperature exceeds a certain threshold. The threshold value is provided via the program arguments. In the simulation, the code for the controller runs in the Raspberry Pi emulator described in Section 4.4.2.

3. **Server** The server represents an edge server in the simulation and runs in an Ubuntu VM. It exposes a REST endpoint to receive notifications and logs incoming requests to stdout.

4. **Network** The network topology consists of a WiFi network that connects the sensor, the controller, and a router, and an Ethernet network that connects the router and the server.

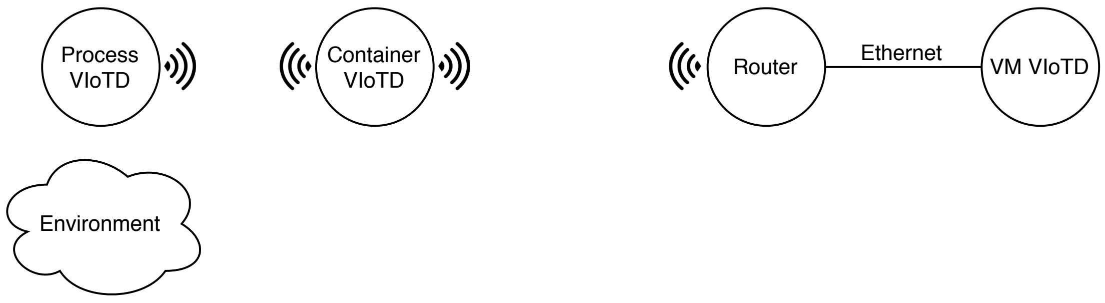


Figure 6.1: Simple IoT setup.

In summary, Figure 6.1 shows the simulation setup. The use-case scenario uses the architecture framework to integrate the sensor as a process, the controller as a container, and the server as VM, and simulates wired and wireless networks with the network simulator.

[1]https://data.world/cityofchicago/beach-weather-stations-automated-sensors, last access at 2020-04-14

6.1.2 Use-Case Simulation

The simulation of the previously defined use-case requires the integration of the sensor, the Raspberry Pi emulator, and the Ubuntu VM with the architecture framework and the simulation of the network shown in Figure 6.1. The experimental setup is summarized in Table 6.1. The following paragraphs describe the necessary steps for the simulation in more detail and match the corresponding user stories defined in Section 4.1.1, indicated by *{USXX}*.

Table 6.1: Experimental setup for the functional evaluation.

OS	Arch Linux x86_64
Kernel	5.5.10
CPU	Intel i7-8565U (1.8GHz, 4 cores)
Memory	16GB
Disk	512GB SSD with ext4 file system

Sensor

The simulated sensor is a python script that implements the previously defined functionality. Appendix E.1 shows the source code. The integration process follows the steps described in Section 5.2.3. After installing python3 and the dependencies of the sensor script on the host, it is necessary to copy the python script into the program directory of the architecture framework. The VIoTD configuration file shown in Listing 6.1 finishes the integration process for the sensor.

Listing 6.1: VIoTD configuration for the simulated sensor.

```
framework_version: v0.1
device:
    device_type: process
    device_info: >-
      Process simulating a sensor.
      Upon request, the sensor reads one line
      of the provided sensor data file and
      provides this value in the reply.
    program_execution_file: "sensor.py"
    start_command: "python3"
```

The VIoTD configuration in Listing 6.1 provides a short description about the virtualized device {US36}. In addition, it defines the source file for the process and specifies the required information to start the sensor {US32}. Besides that, the script requires program arguments, i.e., the path to the dataset and the port the sensor should listen on. The *start simulation* file allows users to specify program arguments that are then passed to the process when the simulation starts {US21}. Listing 6.2 shows the content of this file for the sensor. In this simulation, the sensor listens on port 10000 and, as shown in Appendix E.1, binds its socket to the obtained IP address {US11}. As the process runs in a separate network namespace, it can access any port without interfering with the host or other VIoTDs {US22}. Further, it uses the shared folder where users can upload arbitrary resources to {US19}. In this use-case, the folder contains data that mimics the environment {US05, US06}. Besides reading data from the shared directory, virtualized devices are allowed to write content to it as well, e.g., to simulate a sensor or actuator interacting with the environment {US08}. Besides that, the *start simulation* files allow users to forward a pre-defined set of values to the virtualized device {US07}.

Listing 6.2: Start simulation file for the sensor.

```
--data-path <working-dir>/temperature_data.txt --port 10000
```

The architecture framework requires the user to specify a command in the *stop simulation* file to stop the sensor properly. Listing 6.3 shows the content of the corresponding file for the sensor.

Listing 6.3: Stop simulation file for the sensor.

```
CTRL+C
```

Controller

In the simulation, the controller is an emulated Raspberry Pi running a python script that implements the business logic {US37}. As the Raspberry Pi emulator is already part of the platform, the controller re-uses an already existing VIoTD. Section 4.4.2 describes the integration process and provides the corresponding VIoTD configuration. The configuration file for the Raspberry Pi emulator specifies the hardware resources {US33} of the emulator that are similar to a real Raspberry Pi {US03}. Also, it includes a short description of the emulator {US36}.

The business logic for the controller in Appendix E.2 gets uploaded to the shared directory and is available during the simulation {US18, US20}. The controller script requires python and some additional libraries. Therefore, the preparation task installs required dependencies on the virtualized device {US10, US23}. Listing 6.4 shows the content of the preparation task file. Similar to the process, the user has full control over the shared directory {US19}.

Listing 6.4: Preparation task file for the controller.

```
apt update && apt install -y python3 python3-pip;
pip3 install -r /workdir/requirements.txt
```

The VIoTD requires additional information to run the controller script during the simulation. Therefore, Listing 6.5 and Listing 6.6 provide the commands for starting and stopping the simulation on the VIoTD. The start script specifies additional program arguments required by the business logic {US21}.

Listing 6.5: Start simulation command for the controller.

```
python3 /workdir/controller.py --sensor-host "10.1.1.2"
--sensor-port 10000 --server-host "10.1.2.41" --server-port
11000 --threshold 23
```

Listing 6.6: Stop simulation command for the controller.

```
CTRL+C
```

Server

The server represents a small edge server and runs in an Ubuntu VM. After following the steps in Section 5.1 and Section 5.2.1, it is possible to integrate the VM into the simulation. Similar to processes and containers, the architecture framework requires a VIoTD configuration file. Listing 6.7 shows the configuration file for the Ubuntu VM. The VIoTD configuration provides a short description of the virtualized device {US36}, but does not include the complete libvirt configuration due to its length.

Similar to containers, VMs allow users to run arbitrary code {US37}. The business logic for the server is provided in Appendix E.3. Again, the server script requires python3 and some additional libraries. In addition, the source code for the server is in a Git[2] repository hosted on Gitlab[3]. Therefore, it is necessary to set up the server by installing python3, the required libraries, git, and to check out the repository from Gitlab {US18, US23}. The preparation task provides the needed functionality and allows users to run custom

[2] `https://git-scm.com/`, last access at 2020-04-14
[3] `https://gitlab.com/mLe110/viotdserver.git`, last access at 2020-04-19

Listing 6.7: VIoTD configuration for the Ubuntu VM.

```
framework_version: v0.1
device:
    device_type: vm
    device_info: >-
      Plain Ubuntu image. Can be used to
      simulate servers.
    template_vm_image_path:
    /opt/testpool/devel/xenial-server-cloudimg-amd64.img
    vm_force_stop_time: 60
    hypervisor_uri: qemu:///system
    boot_finished_msg: "ubuntu login: "
    vm_idle_msg: "ubuntu@ubuntu:~ $ "
    login:
      username: ubuntu
      password: asdfqwer
    xml_template_str: >
      <domain type='qemu'>
        ...
      </domain>
```

commands on the virtualized device {US19, US20}. Listing 6.8 shows the *preparation task* file for the server.

Listing 6.8: Preparation task file for the server.

```
sudo apt update;
sudo apt install -y python3 python3-pip git;
git clone https://gitlab.com/mLe110/viotdserver.git;
pip3 install -r viotdserver/requirements.txt;
```

To start the server in the simulation, it is necessary to specify the proper command in the *start simulation* file {US21, US22}. Listing 6.9 shows the content of the start simulation file and Listing 6.10 shows the stop command.

Listing 6.9: Start simulation command for the server.

```
python3 viotdserver/server.py --port 11000
```

Listing 6.10: Stop simulation command for the server.

```
CTRL+C
```

The architecture framework enables the integration of the sensor, the Raspberry Pi emulator as well as the Ubuntu VM without making any changes to the virtualized devices {US35}. Besides, its implementation focuses on resource efficiency by keeping the number of frameworks low and running only functionalities required by the virtualized device {US44}. After executing these steps, the VIoTDs can participate in the simulation.

Network

Besides the device integration, the network setup is the second part of the simulation. As the use-case involves a VM as a virtualized device, it is necessary to create a libvirt network for the VM. The network simulator provides a REST endpoint to create such a network. The API description of the *Create libvirt network* endpoint is in Appendix B. Listing 6.11 shows the payload used in the simulation to set up the network.

Listing 6.11: JSON payload to create the libvirt network for the VM.

```
{
  "network_name": "firstnet",
  "gateway_ip": "10.1.2.1",
  "netmask": "255.255.255.0",
  "start_ip": "10.1.2.40",
  "end_ip": "10.1.2.50"
}
```

Further, the network simulator requires a JSON as input that defines the network topology. The network JSON for this use-case scenario is in Appendix E.4 and represents the topology in Figure 6.1. The simulated network allows communication between VIoTDs {US02}. It creates a wireless network to connect the sensor and the controller {US12, US13}, simulates a wired connection for the server {US16}, and connects both network segments via a virtual router. In addition, it specifies the position of the nodes that participate in the wireless network {US14}.

Simulation

Besides the integration of the VIoTDs and the network setup, the simulation requires a platform configuration to provide details about the host configuration to the VIoTDs and to define the logging infrastructure for the simulation. Listing 6.12 shows the platform configuration for this use-case scenario.

Listing 6.12: Platform configuration for the use-case scenario.

```
vm_storage_pool_path: /opt/simpool/devel
device_setup_file_name: preparation.txt
simulation_commands_file_name: start_simulation.txt
stop_simulation_commands_file_name: stop_simulation.txt
net_namespace_directory: /var/run/netns
program_executable_directory: /opt/simulation
reporting:
    type: file
    log_directory: /var/viotd-evaluation/simulation/logs
monitoring:
    type: console
```

The platform configuration defines the log appenders for the VIoTDs and the application logs. Users have access to both log information. The VIoTD log provides information about the integration layer and the virtualized device {US28}. In addition, the reporting module forwards log output from the application that runs during the simulation {US30}. VIoTDs provide their log output separately, which allows the user to identify the behavior of each device in the simulation individually {US31}. The platform administrator is responsible for the platform configuration and, therefore, is able to define and configure the log appender for the reporting and the monitoring module {US39, US40, US45}. The administrator has access to all logs and uses this information to identify crashed devices {US42, US43}.

With all the configuration files in place, the simulation can be executed {US01} following the workflow from Section 4.3. The APIs provided by the VIoTDs and the network simulator can be used to manage the lifecycle of the simulation setup. They provide endpoints to run the VIoTD, execute the preparation tasks, start and stop the simulation {US25, US26}, as well as terminating a VIoTD {US27}, i.e., stopping the simulation and shutting down the VIoTD. Additional endpoints provide information about the current device state {US17, US29}, and enable health checks automatically {US34}. After stopping the simulation, users can run another simulation with the same VIoTDs {US15}. Between simulations, it is possible to perform additional preparation tasks or to make changes to uploaded code.

The network simulator runs in a Docker container and isolates the simulated network. Therefore, it is possible to run multiple simulations in parallel without interferences {US41}.

As previously mentioned, the system provides VIoTD and application logs. Therefore, these logs can be used to analyze the simulation result. Listing 6.13, Listing 6.14, and Listing 6.15 show a short example of the log files from the sensor, the controller, and the server, respectively. In this use-case scenario, the setup runs for 60 seconds before the server stops automatically. This can be achieved with the *duration* parameter of the *start simulation* API from the VIoTD. By specifying a duration, the VIoTD stops the simulation on the virtualized device automatically after the specified time {US24}. We use this functionality to show the behavior of the system when one node fails.

Listing 6.13: Log output of the sensor during the simulation.

```
2020-02-25 00:19:01,970 root INFO Measure temperature: 21.70
2020-02-25 00:19:02,973 root INFO Measure temperature: 24.00
2020-02-25 00:19:03,985 root INFO Measure temperature: 24.20
2020-02-25 00:19:04,987 root INFO Measure temperature: 21.50
```

Listing 6.14: Log output of the controller during the simulation.

```
2020-02-25 00:19:01,971 root INFO Poll sensor data: 21.7
2020-02-25 00:19:02,973 root INFO Poll sensor data: 24.0
2020-02-25 00:19:02,974 root INFO Threshold exceeded: 24.0.
Send notification to server.
2020-02-25 00:19:03,985 root INFO Poll sensor data: 24.2
2020-02-25 00:19:03,985 root INFO Threshold exceeded: 24.2.
Send notification to server.
2020-02-25 00:19:03,986 root ERROR Cannot send notification to
server
2020-02-25 00:19:04,987 root INFO Poll sensor data: 21.5
```

Listing 6.15: Log output of the server during the simulation.

```
2020-02-25 00:18:46,971 werkzeug INFO * Running on
http://0.0.0.0:11000/ (Press CTRL+C to quit)
2020-02-25 00:19:02,982 root INFO Notification received. Data:
{'threshold': 23.0, 'value': 24.0}
```

During the simulation, VIoTDs have access to their shared folder. They can use previously uploaded resources or write device-specific output files into this directory. Hence, it provides a basic environment functionality, especially to simulated sensors and actuators {US05, US08, US09}. Additionally, the architecture framework is compatible with, for example, sensor or actuator units connected via USB or GPIO to an SBC {US04}. The current system treats both components as a single VIoTD, as only network connections described in Section 4.2 get simulated.

6.1.3 Benefits & Shortcomings

The previous sections demonstrate the capabilities of the architecture framework and the network simulator by executing a simple use-case scenario. This functional evaluation shows how, and to what extent, the system meets the requirements defined in Section 4.1.1. The following paragraphs discuss both components on a higher level and elaborate the impact of certain design decisions on the system.

In the real world, IoT networks consist of a variety of different devices [4]. Additionally, microcontroller architectures of IoT devices often differ from the x86 processor architecture used in many cloud data-centers and on personal computers [19]. This is challenging for the implementation of the architecture framework, as one of its goals is to integrate accurate device emulators. Hence, the architecture framework has to support hardware virtualization as well. Fortunately, Docker containers are compatible with Qemu, and binfmt allows mappings between binary types and interpreters. Therefore, hardware virtualization for processes and containers can be done with reasonable effort, as shown in Section 4.4.2. The integration of guest VMs with a different target architecture requires additional work. The architecture framework addresses this issue by integrating libvirt to manage VMs. This framework supports a variety of hypervisors including Qemu, Xen, Bhyve and many more[4]. Especially Qemu is of high interest due to its hardware virtualization capabilities[5], including target architectures like ARM that is widely used in the IoT [139].

Another important characteristic of the architecture framework is its compatibility with existing emulators and simulators. The functional evaluation shows the frameworks' capability to integrate virtualized devices without enforcing any changes onto the devices. In addition, it provides an abstraction layer to the network. Therefore, virtualized devices use their original network stack during the simulation. This is beneficial to the quality of the simulation, as the network layer is essential in IoT systems [140] and any changes imposed by the framework could create a skew in the simulation behavior. Although these are valuable characteristics, it is unlikely that the architecture framework is compatible with any existing IoT emulator and simulator. There exist an unknown number of virtualized devices, and therefore, it is important to acknowledge that incompatibilities may happen. Besides that, the proposed framework is a first step into building a generalized integration solution and is compatible with a reasonable number of different virtualization technologies.

The architecture framework and the network simulator provide flexibility to users. Using the *preparation task* functionality, users can customize VIoTDs before running their IoT applications during the simulation. Also, the newly developed ns-3 module enables users to create simulation networks dynamically. To the best of our knowledge, the combination of virtualized IoT devices and simulated networks is a novel approach for currently available IoT simulation platforms.

[4]`https://libvirt.org/drivers.html`, last access at 2020-04-14
[5]`https://wiki.qemu.org/Documentation/Platforms`, last access at 2020-04-14

Besides these benefits, the proposed solution has some notable shortcomings as well. Table 6.3 provides a summary of missing or partially implemented requirements. Most importantly, the architecture framework does not cover all of the user stories defined in Section 4.1.1, and implements some of them scarcely. The missing user stories are {US06} and {US38}. In addition, the framework provides a simplistic implementation of any environment-related requirement. The reason for this deficiency is the currently immature specification. Due to time constraints, the environment is still an abstract idea, and a specific plan, how it could look like, does not exist yet. Similar to the architecture framework, the environment has to provide an abstraction layer between virtualized devices and their virtual surroundings. Hence, including this complexity in addition to the proposed system would go beyond the scope of this thesis and is left out for future research. The current solution provides a shared folder to include, e.g., input files for sensors and allows virtualized devices to create output files.

Table 6.3: Missing or partially implemented requirements.

Affected Requirement	**Functionality**	**Description**
USB/GPIO abstraction	Missing	Further specifications required.
Environment	Limited	Due to time constraints and the design complexity of the environment, the architecture framework provides only limited functionalities.
6LoWPAN and 802.15.4 support	Missing	The current version of ns-3 has a bug that prevents TapDevices from using 6LoWPAN and 802.15.4.

More importantly, the network simulator misses some important protocols and standards. Due to resource-constrained devices, communication and network protocols in IoT systems are more restricted than in traditional networks [141]. Therefore, many IoT systems use protocols designed explicitly for the IoT. Especially the 6LoWPAN protocol and the IEEE 802.15.4 standard are some of the most important protocols in this domain [142]. Although ns-3 provides 6LoWPAN and 802.15.4 modules, it seems that the ns-3 core has issues with IPv6 addresses and TapBridges. More specifically, the TapBridge lacks IPv6 support, as one of the reported ns-3 bugs indicates[6]. In addition, ns-3 terminates when it routes IPv6 packets from external hosts due to serialization issues. We reported this issue in the ns-3 google group[7] and after a short discussion with a member of the ns-3 community, concluded that this is a bug in ns-3. Unfortunately, only TapBridges

[6]`https://www.nsnam.org/bugzilla/show_bug.cgi?id=1433`, last access at 2020-04-14

[7]`https://groups.google.com/forum/#!topic/ns-3-users/5F7cZtL2ibo`, last access at 2020-04-14

allow external devices to participate in the network simulation. Therefore, the network simulator cannot support 6LoWPAN and 802.15.4 until the bug gets fixed. Nonetheless, the developed ns-3 module and the network JSON schema can be easily extended once the bug is fixed. The structure of the network JSON supports new network types out-of-the-box, and the ns-3 module has an extendable design, i.e., it is sufficient to add a new method to the *NetworkHelper*[8] class that handles the new link type.

Another improvement to the current solution relates to {US04}. The architecture framework treats, for example, a virtual sensor connected to an SBC via USB as a single virtualized device. Hence, it is not possible for users to reuse single components or to control them individually. Separating these components requires further research to determine if, and to what extent, it is possible to define a generic interface that allows independently developed components to interact. Any design decisions to address this task need to be well-thought to avoid restrictions on virtualized devices.

6.2 Raft

The evaluation continues with the simulation of the use-case scenario, i.e., the Raft algorithm (see Section 4.4), and shows the applicability of the proposed solutions with a real-world example. The objective of this evaluation is to determine the performance of the Raft algorithm in an IoT setup. Therefore, the scenario measures the leader election time of the algorithm running on emulated Raspberry Pis in an 802.11s mesh network. The following sections describe the test setup, define the metrics used for the evaluation, and demonstrate the validity of the obtained results.

6.2.1 Test Environment

The Raft algorithm sends multiple messages in parallel for heartbeat information and elections. Depending on the heartbeat frequency and the election interval, this can lead to substantial network traffic. The experimental setup described in Table 6.1 does not provide sufficient computing power for the Raft evaluation, as ns-3 requires additional processing capacity. Therefore, the evaluation uses a C2 instance on the Google Cloud Platform (GCP) optimized for compute-intensive tasks[9]. Table 6.5 summarizes the characteristics of the experimental setup for the use-case scenario.

The use-case simulation evaluates the performance of the Raft algorithm in a wireless setup. Hence, in order to provide comparable results, the simulation scenarios follow the metrics defined in [20]. Ousterhout and Ongaro [20] evaluate the performance of Raft by measuring the leader election time, i.e., the time it takes the cluster to detect a crashed leader and to elect a new one. The leader crashes randomly within its heartbeat interval. Further, it broadcasts a heartbeat message before it crashes. This creates a worst-case scenario, as each server in the cluster resets its election timeout at approximately the

[8]`https://github.com/mLe110/network-simulator`, last access at 2020-04-14
[9]`https://cloud.google.com/compute/docs/machine-types`, last access at 2020-04-15

Table 6.5: Experimental cloud setup for the Raft evaluation.

Platform	Google Cloud Platform
Instance Type	c2-standard-8
OS	Ubuntu 19.10 x86_64
Kernel	5.3.0
CPU	Intel Xeon Scalable (3.1GHz, 8 cores)
Memory	32GB
Disk	128GB SSD with ext4 file system

same time when the leader crashes and, therefore, increases the election time and makes split votes more likely. Furthermore, the heartbeat interval is half of the minimum election timeout, and thus, the smallest possible downtime of the cluster is equal to the heartbeat interval.

Ousterhout and Ongaro [20] suggest that the broadcast time should be an order of magnitude less than the minimum election timeout. The broadcast time includes the network round-trip-time, and the disk writes, i.e., the time it requires a server to send a message to each participant in the cluster and receive a response.

Besides Raft-specific configurations and definitions, the test environment has additional important characteristics. It automatically runs the emulated Raspberry Pi's, executes the network simulator, and starts the simulation. The VIoTDs use the Kafka appender and stream the application logs back to the test environment. Further, the test environment stops the simulation after 1000 trials, where each trial corresponds to a server crash followed by a leader election. Equivalent to [20], each scenario consists of five server nodes.

The scenarios running on the emulated mesh network use the ns-3 mesh module. Appendix C.3 provides an example network topology JSON. The network simulator uses the default configuration of the YansWifiChannel module in the WiFi mesh to simulate the wireless communication channels between servers in the network. The channel has a propagation delay based on a constant speed model, i.e., the speed of light, and calculates propagation loss with a log distance model[10].

[10]`https://www.nsnam.org/docs/models/html/wifi-user.html#yanswifichannelhelper`, last access at 2020-04-15

6.2.2 Raft Implementation Baseline

While Section 6.1 reviews the functionalities of the proposed components, it is still left to show that the simulation results are valid. Hence, it is necessary to ensure that the Raft implementation behaves similarly to the proposed implementation in [20].

The Raft implementation from Ousterhout and Ongaro is written in C++ and part of RAMCloud, a novel storage system for datacenters developed at the University of Stanford [20, 143]. Therefore, the performance results obtained in [20] are based on the RAMCloud implementation.

The first scenario evaluates our Raft implementation and compares it to [20]. Based on the previously described test setup, the Raft algorithm runs as a process and communicates over the localhost network interface to avoid possible interferences from the network simulator. The Linux Traffic Control[11] utility was used to mimic the broadcast time of about 15ms by introducing a delay on the localhost network interface.

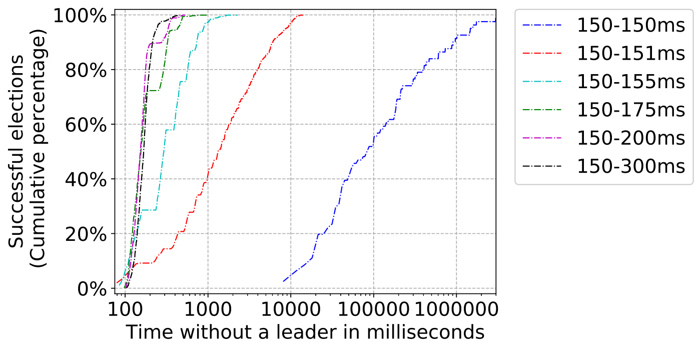


Figure 6.2: Performance of the Raft implementation running on localhost and using different election intervals.

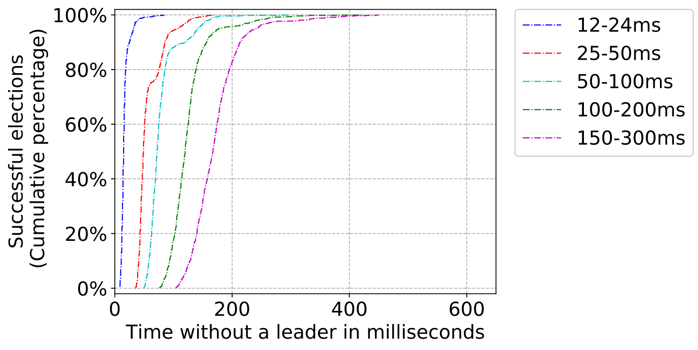


Figure 6.3: Performance of the Raft implementation running on localhost and using different minimum election timeouts.

[11]`https://linux.die.net/man/8/tc`, last access at 2020-04-15

The evaluation results are shown in Figure 6.2 and Figure 6.3. The y-axis shows the percentage of successful elections, and the x-axis refers to the duration the cluster requires to elect a new leader after a server crash. Each line represents a specific election timeout and illustrates the number of elections that finish within a certain time. For example, the election timeout of 150-300ms in Figure 6.3 shows that about 80% of the elections require at most 200ms. Further, we use this type of diagram throughout the section to demonstrate the leader election time of the Raft algorithm.

Figure 6.2 shows the time the cluster is without a leader for different election intervals and uses a logarithmic scale on the x-axis, as some intervals require significantly more time to finish. Similar to [20], increasing the randomness in election intervals has a positive impact on the leader election time. Furthermore, the election timeout of 150-150ms diverges from [20]. Without randomness in the election interval, servers always start elections simultaneously. Hence, candidates can only win elections due to varying runtime delays on the servers. We conclude that the deviation for 150-150ms in Figure 6.2 is due to differences in the experimental setup and the applied programming language for the Raft implementation and therefore, does not falsify the correctness of our implementation.

Further, Figure 6.3 determines the performance of the algorithm for different minimum election timeouts. The timing characteristics of Raft restrict timeouts below 12ms in this scenario, as the broadcast time is around 15ms. For further comparisons, Figure 6.4 shows the number of won elections for each server, grouped by the same election intervals as in Figure 6.3. The number of won elections for varying election intervals is comparable to Figure 6.4, and therefore, not explicitly shown. In conclusion, the behavior of our Raft implementation is similar to [20].

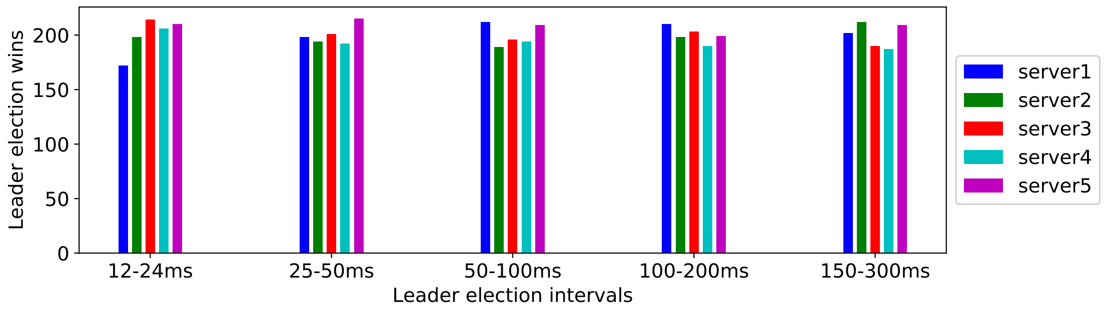


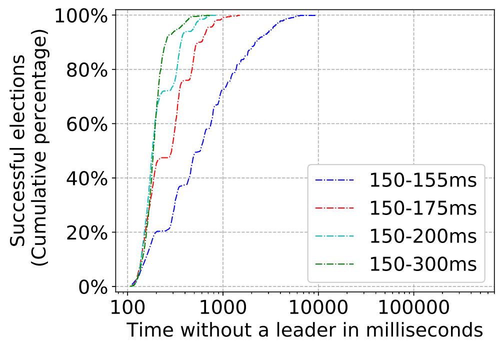
Figure 6.4: Number of won elections for each server per election interval in the baseline evaluation.

6.2.3 Simulation Baseline

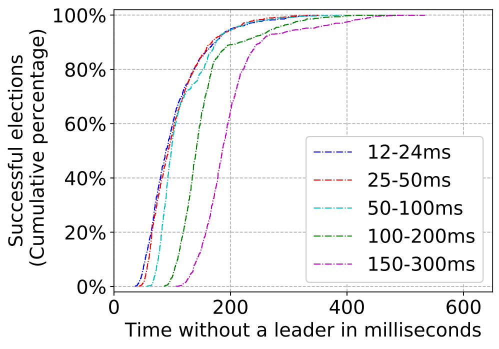
After showing that our Raft implementation mirrors the behavior of the original implementation by Ousterhout [20], the second scenario integrates the network simulator into the evaluation. The setup runs the Raft algorithm on emulated Raspberry Pi VIoTDs. Further, the network simulator connects the VIoTDs via a simulated CSMA network with similar characteristics as the network described in [20].

The objective of this scenario is to evaluate if the architecture framework and the network simulator impact the simulation results, e.g., by introducing any skew into the performance of the Raft algorithm. Hence, the simulated network restrains the broadcast time to about 15ms, and the evaluation follows the methodology described in Section 6.2.2.

Figure 6.5 shows the performance of the Raft algorithm in the simulated CSMA network. Comparing Figure 6.5b to the baseline in Section 6.2.2 leads to the following observations. While the graphs for 100-200ms and 150-300ms show similar characteristics, their behavior diverges at election timeouts of 50-100ms and below. With an election timeout of 50-100ms, Figure 6.3 shows that 90% of the elections require at most 100ms. The equivalent line in Figure 6.5b shows that only 60% of the elections finish within 100ms. For election timeouts of 12-24ms and 25-50ms, the deviation from the baseline in Figure 6.5b is more significant. Furthermore, the leader election time seems to consolidate below a certain threshold.



(a) Different election intervals.



(b) Different minimum election timeouts.

Figure 6.5: Performance of the Raft implementation in the CSMA simulation. Each line represents 1000 trials.

Besides reducing the minimum election timeout as demonstrated in Figure 6.5b, the scenario evaluates the Raft algorithm on the simulated CSMA network with different election timeouts as well. Figure 6.5a shows the evaluation results. Comparing the election time of Raft on the simulated CSMA network with its baseline in Figure 6.2 shows consistent results for the 150-300ms election timeout. Similar to Figure 6.2, Figure 6.5a uses a logarithmic scale on the x-axis. Further, the lines for 150-200ms match for around 70% of the leader elections, but slightly diverge afterward. For election timeouts below, the CSMA network provides different results than the baseline evaluation.

For example, with an election timeout of 150-175ms, the longest trial in the baseline simulation is about 900ms, but 1500ms in the CSMA network. With 150-155ms, less than 20% of the elections match those in the baseline. For 150-151ms and 150-150ms, the election time increases significantly, and the results are substantially different. Hence, we stopped the evaluation at this point.

We presume that ns-3 introduces this skew as it slows down network traffic due to its resource-intensive computations. This assumption is based on two observations. First, Raft creates substantial amounts of messages simultaneously, especially for low minimum election timeouts that entail small heartbeat intervals, and when split votes occur. Further, low randomness, i.e., small election intervals, increase split votes [20]. Therefore, ns-3 has to process most of the messages within short time frames. Second, the behavior can be verified by simulations with further election timeouts, and executing the scenario on the experimental setup described in Table 6.1, i.e., on hardware with less computing capacity, shows a similar skew and already impacts the results for larger election timeouts like 150-300ms. In conclusion, the simulation on the experimental setup described in Table 6.5 provides viable results for election timeouts above 50-100ms.

Nevertheless, Figure 6.6 shows that the number of won elections is independent of the minimum election timeout and is uniformly distributed among the servers. Hence, it provides similar results as in Section 6.2.2.

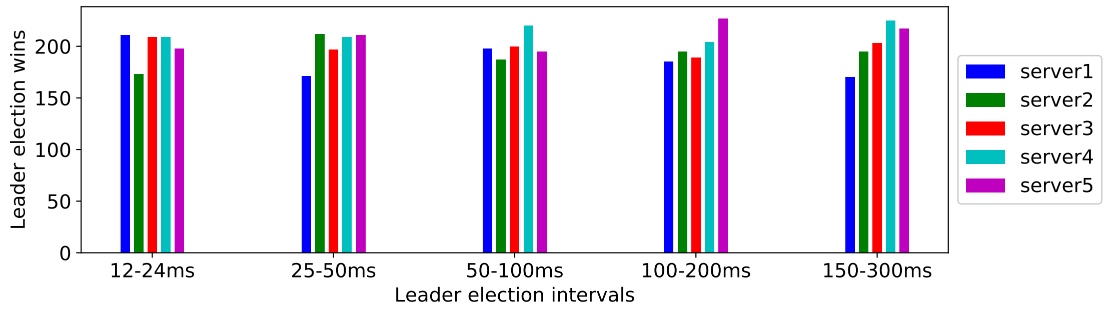


Figure 6.6: Number of won elections in the CSMA simulation for each server per election interval.

6.2.4 Mesh Base Scenario

After validating the performance of the Raft implementation and assessing the characteristics of the simulation, the following scenario introduces the first IoT setup. Therefore, the network simulator executes a wireless mesh network with approximately the same distance between servers. Figure 6.7 illustrates the setup, where L refers to the distance between servers. In this topology, the round-trip-time of messages is independent of the source and the target. Hence, the result shows how wireless communication in the mesh network influences the performance and the characteristics of the Raft algorithm compared to its implementation in the datacenter. Therefore, this scenario simulates the network topology shown in Figure 6.7 with two distinct distances.

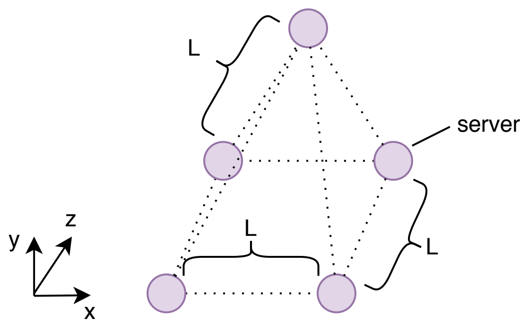


Figure 6.7: Network topology for the mesh base scenario.

The first simulation evaluates a mesh network with a broadcast time of about 15ms. As the wireless network is fluctuating, i.e., messages with the same source and target have significantly different round-trip-times, the broadcast time in the mesh network represents an average. Therefore, a server sends a message to every member in the cluster simultaneously and measures the time until it receives the last response. The server executes 1000 repetitions of the previous step and calculates the average that represents the broadcast time. After assessing the broadcast time for multiple distances L, a distance of 10m showed the desired time of 15ms.

The evaluation of the mesh network with a distance of 10m is shown in Figure 6.8. Due to significant differences in the leader election time, Figure 6.8 uses a logarithmic scale on the x-axis. In comparison to the performance of Raft in CSMA networks, i.e., Figure 6.3, the average leader election time is higher in the wireless mesh network. In addition, some elections result in many split votes, indicated by the election times above 1s. Another noticeable characteristic is the slant between 40% and 60% that occurs in every simulated election timeout. It emerges at around 100ms, 200ms, 300ms, and 600ms for the election timeouts 50-100ms, 100-200ms, 150-300ms, and 300-600ms, respectively. Hence, about 40% to 60% of the elections are not successful in the first attempt and cause split votes.

In the second simulation, the distance L between servers is 100m. The broadcast time in the simulated mesh network is about 250ms and represents the average over 1000

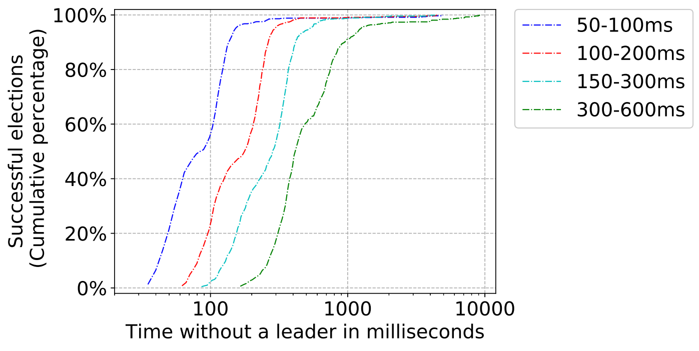


Figure 6.8: Performance of Raft in a mesh network with varying minimum election timeouts and 10m distance between servers.

measurements. Figure 6.9 shows the outcome of the evaluation and uses a logarithmic scale on the x-axis. As proposed by [130], the election timeout of 2500-5000ms is one magnitude larger than the broadcast time and represents the benchmark for this evaluation. Further simulations reduced the election timeout subsequently without violating the timing constraint of Raft [20].

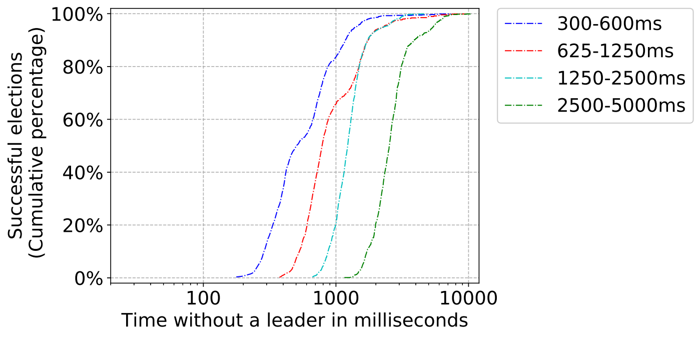


Figure 6.9: Performance of Raft in a mesh network with varying minimum election timeouts and 100m distance between servers.

While the election timeouts of 1250-2500ms and 2500-5000ms show similar characteristics, 300-600ms and 625-1250ms lead to increased numbers of split votes and thus, finish approximately at the same time as the simulation with an election timeout of 1250-2500ms. In wireless networks, the distance between nodes influences the data transmission speed, i.e., larger distances reduce the bandwidth and increase packet losses and delays [144, 145]. Based on the test setup description in Section 6.2.1, the Raft servers in the cluster reset their election timeout when the leader crashes. Therefore, the probability of a split vote depends on the randomness of the election timeout, i.e., the size of the election timeout interval [130]. When the first server times out, it starts a leader election and sends a vote request to the remaining cluster. In this scenario, the duration between sending the vote request and other servers receiving it is longer compared to the CSMA network due to the distance between servers and the characteristics of wireless networks. Hence, servers are more likely to timeout before they receive the vote request.

The election timeouts of 1250-2500ms and 2500-5000ms offer more randomness compared to 300-600ms and 625-1250ms due to their interval size. In addition, their minimum election time tolerates higher delays and packet losses in the network, resulting in fewer split votes, as shown in Figure 6.9.

6.2.5 Mesh Line Scenario

Ousterhout and Ongaro [20] recommend an election timeout that is an order of magnitude greater than the broadcast time. In a wireless network, the distance between nodes impacts the data transmission between them [144, 145] and therefore, affects the broadcast time. Further, as Raft sends vote requests and heartbeat messages in parallel, its broadcast time depends on the most distant servers. For this reason, the following scenario simulates the network topology shown in Figure 6.10 that arranges servers on a single line. Hence, while servers in the center have similar round-trip-times to their neighbors, the servers on both ends define the longest distance in the network that may involves multiple hops. Furthermore, L in Figure 6.10 refers to the distance between two consecutive nodes. Similar to the previous scenario, the following section evaluates the line topology with two distinct distances.

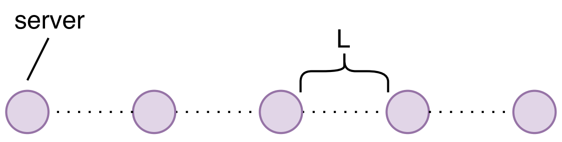


Figure 6.10: Network topology for the mesh line scenario.

The first simulation defines an L of 10m, and thus, the total distance between the outermost servers is 40m. The broadcast time in this network, i.e., the average over 1000 measurements, is about 20ms, and therefore, the election timeout of 200-400ms establishes the reference for the remaining tests. Figure 6.11 shows the simulation results for the line topology with four different election timeouts and a logarithmic scale on the x-axis. As has been previously shown, ns-3 affects the performance of Raft for minimum election timeouts below 50ms. Therefore, the lowest election timeout is 50-100ms, as shown in Figure 6.11, although the broadcast time would allow timeouts below.

In the simulation with an election timeout of 200-400ms, about 60% of the leader elections complete without split votes. For timeouts below, this is only the case for about 50%. Hence, while smaller election timeouts require less time to detect a crashed leader and elect a new one, split votes are more likely to happen. In contrast, the election timeout of 400-800ms does not create split votes in about 75% of the elections, but the cluster needs more time to recover, on average, after a leader crashes.

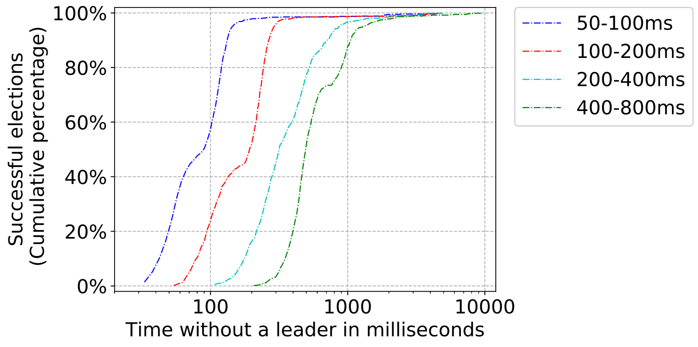


Figure 6.11: Performance of Raft in a one dimensional mesh network with a distance of 10m between two consecutive nodes.

The second simulation uses a distance of 50m between two consecutive servers. The total distance of 200m between both ends causes a broadcast time of about 160ms. Following the same procedure as in the previous simulations, the broadcast time corresponds to the average duration based on 1000 broadcasts. Thus, the election timeout of 1600-3200ms determines the baseline for the remaining tests.

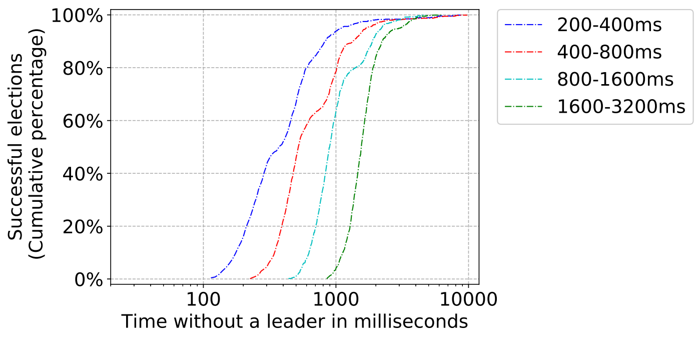


Figure 6.12: Performance of Raft in a one dimensional mesh network with a distance of 50m between two consecutive nodes.

Figure 6.12 shows the simulation results for the line topology with an L of 50m. Similar to before, the x-axis uses a logarithmic scale. The tendency of increased split votes in wireless mesh networks can be observed in Figure 6.12 as well. Further, only 20% of the elections for 200-400ms and 400-800ms finish successfully in the first attempt. For 800-1600ms and 1600-3200ms, it is about 35% and 55%, respectively. Hence, this behavior aligns with previous remarks in Section 6.2.4 on leader election in wireless networks. The line topology with 50m between servers amplifies this effect, as routes between servers involve multiple hops, which impacts the message transmission.

As the wireless mesh network influences the performance of the Raft algorithm, the scenario further examines if the network impacts the electability of servers in the cluster.

Therefore, Figure 6.13 shows the won elections for each server in the line topology with an L of 50m, grouped by the election timeout. The result shows similar characteristics as the baseline evaluation in Section 6.2.2, and therefore, the wireless network seems not to influence the probability that a server wins an election.

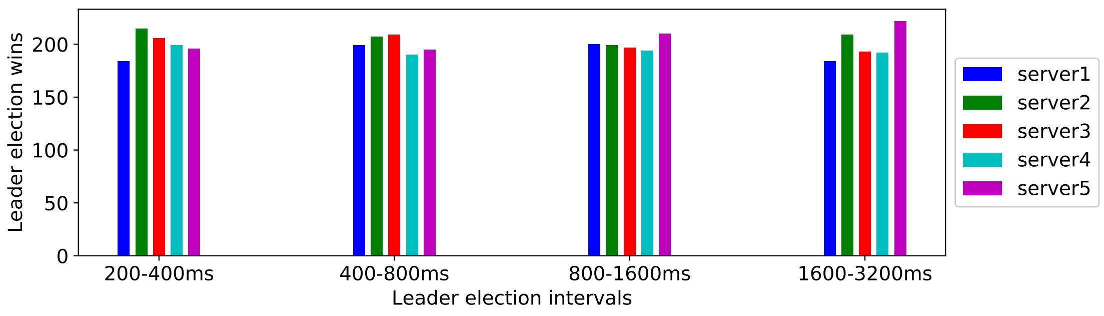


Figure 6.13: Number of won elections for each server per election interval in the line topology simulation with 50m distance between servers.

6.2.6 Discussion

The stability of the Raft algorithm depends on timing constraints [130]. In a Raft cluster, leaders must be able to broadcast heartbeat messages to followers before they start an election. Hence, there is a trade-off between detecting a crashed leader early and avoiding unnecessary leader changes, both impacting the cluster availability [20].

While the election timeout relates to the stability of the cluster, it also affects the probability of split votes [130]. As shown in Section 6.2.4 and 6.2.5, lower election timeouts and reduced randomness significantly increases split votes. Although this characteristic applies to LAN networks in datacenters as well, it has a higher impact in wireless mesh networks.

Furthermore, due to the larger broadcast time in wireless mesh networks, using an election timeout that is a magnitude higher than the broadcast time leads to long leader elections, and it takes the cluster more time to recover if the leader fails. For example, the broadcast time in the line topology in Section 6.2.5 is about 160ms when the distance between servers is 50m. Doubling the distance to 100m results in a broadcast time of about 500ms. Hence, the corresponding minimum election timeout is 5000ms.

Nonetheless, Raft demonstrates reasonable performance in wireless mesh networks. The Raft implementation in the previous scenarios strictly follows the description in [20, 130]. Hence, it does not make use of possible improvements that could benefit the performance in wireless networks. Replacing the RPCs with a UDP-based communication method would reduce the message overhead. Further, as wireless networks are broadcast oriented, UDP would improve the efficiency of broadcasts for heartbeat messages and vote requests [23].

Besides performance characteristics, Raft elects leaders uniformly, which is beneficial for IoT networks. For example, assuming specific nodes in the network are less stable than others, a biased leader election algorithm that accidentally favors an unstable node would reduce the stability of the cluster.

In conclusion, Raft performs well in the evaluation scenarios on wireless mesh networks. Although split votes and large election timeouts are not necessarily beneficial for its applicability, its simplistic design, i.e., compared to other consensus algorithms like Paxos, facilitates changes. Therefore, Raft shows potential for future developments [23, 130].

CHAPTER 7

Conclusion & Future Work

The IoT paradigm describes context-aware systems that are embedded in the environment and consist of smart devices with ubiquitous connectivity. Since its origins in 1999, the IoT developed rapidly and, nowadays, impacts a wide range of domains. With the emergence of new technologies and applications in safety-critical environments, IoT systems became more sophisticated and impose new challenges on researchers and developers. Due to the complexity of building IoT systems in the real world, advanced tools to test and verify proposed solutions are needed.

Therefore, this thesis proposes a novel end-to-end simulation approach based on an architecture framework and a network simulator to test and evaluate IoT systems and applications. Further, to demonstrate their applicability, the Raft consensus algorithm gets evaluated in a simulated IoT system.

The literature research in the thesis provides essential background information and covers related work. It presents a comprehensive definition of the IoT and illustrates its potential by summarizing state-of-the-art applications in various domains. As connectivity and communication are fundamental characteristics of IoT systems [26], Chapter 2 reviews relevant network protocols. Further, a comprehensive analysis of virtualization technologies and OSs used in the IoT provides important background information for the proposed architecture framework. Then, Chapter 3 analyzes the IoT research process, identifies the development stages, and discusses relevant tools and approaches. In addition, it shows the importance of simulations and summarizes alternatives. The chapter concludes with an extensive analysis of currently used simulation tools, state-of-the-art research on simulators, and consensus algorithms in the IoT.

The proposed solution consists of an architecture framework and a network simulator. The design process of both components described in Chapter 4 includes a requirements analysis based on current research that identifies essential characteristics and functionalities for end-to-end simulations. The heterogeneity of IoT devices and the high release frequency of new devices pose a significant challenge on simulators [5, 10, 12]. The architecture framework follows a different approach compared to state-of-the-art solutions. It can be seen as a toolset that integrates with existing IoT device simulators and emulators and provides a uniform control interface. Hence, it facilitates extensibility and focuses on a simple integration process. As IoT devices use different processor architectures, the architecture framework enables hardware emulation to increase the accuracy of the simulation.

The network simulator utilizes ns-3 to simulate communication channels and provides a management interface based on a REST API. Further, a new ns-3 module was developed, which is able to create network topologies dynamically at runtime.

The thesis concludes with the evaluation of the architecture framework and the network simulator. The first part demonstrates the functional capabilities of the proposed systems, emphasizes essential design decisions, and discusses their benefits and limitations. The second part focuses on the simulation of the use-case, i.e., the performance evaluation of the Raft algorithm in wireless mesh networks. For this, the Raft algorithm has been implemented according to [20]. The evaluation follows multiple scenarios that ensure the correctness of the simulation. The first scenario assesses the implementation of the algorithm to detect potential discrepancies. In the second scenario, the architecture framework integrates Raspberry Pi emulators that represent edge nodes in the simulation and uses the network simulator to mimic the network characteristics of the experimental setup described in [20]. Hence, this scenario determines how the simulation affects the performance of Raft. It shows that ns-3 cannot cope with short minimum election timeouts and low randomness, as it is unable to process the number of concurrent messages. Nevertheless, the results for election timeouts of 50-100ms or higher are not affected. Further, the third and fourth scenarios evaluate the performance of the Raft algorithm in two distinct wireless mesh networks and discuss the characteristics of Raft in IoT systems.

7.1 Key Contributions

The architecture framework addresses the device heterogeneity of the IoT by utilizing existing IoT device simulators and emulators and by providing a simple integration process. Further, it supports hardware emulation that improves the simulation quality and allows application code running on emulated devices to be transferred to real devices with little modifications. Besides, its log appenders provide important monitoring and logging output that helps users to understand the behavior of the system.

The network simulator imitates real-world networks and simulates low-level protocols. With the developed network topology JSON schema, users are able to simulate networks with little effort and can change topologies between simulations at runtime. Further, the network simulator is a single Docker container and, therefore, facilitates a simple setup process of the whole simulation system.

The proposed system supports reproducibility. Simulations rely on configuration files, e.g., the VIoTD configuration files and the network topology JSON, that can be shared easily. Also, VIoTDs can access third-party infrastructure or applications during the simulation. For example, they can exchange data with IoT cloud systems or communicate with external middleware.

To the best of our knowledge, current solutions do not support an equivalent set of different functionalities. ns-3 and OMNeT++ focus on network simulation, but omit device emulation and simulation and provide less flexibility than our network simulator. While Dockemu tries to address this issue by introducing Docker containers, it still misses the flexible process to set up the network topology. IOTSim and iFogSim support a limited set of simulated devices and neglect network simulation. Similarly, VIoLET and ELIoT mainly focus on the application layer, but neither support network simulation nor device emulation. Although Cooja addresses some of these issues by simulating the network layer and running an IoT OS, it only supports Contiki OS.

Only the Hybrid Virtualization Platform proposed by Lai et al. supports device emulation, but does not address the issue with managing devices in the simulation and provides very limited network simulation capabilities.

In conclusion, the proposed systems enable end-to-end IoT simulations and demonstrate reasonable results in the evaluation. Furthermore, the architecture framework introduces a generic approach to utilize existing emulators and simulators without modifications. While this addresses the heterogeneity of the IoT, the processor emulation and network simulation capabilities enable a simulation environment that is closer to the real-world compared to currently available solutions.

7.2 Future Work

This thesis contributes to a research project at the UIUC that develops an IoT simulation platform (see Section 1.3). The proposed architecture framework and the network simulator can be seen as a proof-of-concept for future developments. Although the current solutions already enable end-to-end simulations, further improvements and additional features are possible.

At this point, the architecture framework and the network simulator provide only limited capabilities to mock an environment. Besides simple interactions between the VIoTD and a simulated environment, the idea is to model real-world scenarios in the future, i.e., VIoTDs can be placed into virtual worlds that influence the device connectivity, provide feedback to VIoTDs, and allow controlled mobility models. The environment has already been part of the requirements analysis in Chapter 4, but an advanced implementation was omitted due to time constraints. Building a sophisticated environment requires thorough planning and additional research. The communication between VIoTDs and the environment should not impose changes on virtualized devices. In addition, interactions with the network simulator need to be defined and implemented, as environment models interfere with the simulated network. Also, the solution requires a proper configuration interface for users that allows the definition of environment modules and mobility models.

Furthermore, additional tests and simulation scenarios are necessary to validate the simulation results for different configurations, especially load and performance tests to assess the limits of ns-3. As have been previously shown, ns-3 simulations are resource-intensive and demonstrate limited real-time capabilities. This impacts the scalability of the proposed system and requires additional research. While developing a scaling approach for ns-3 may solve the problem, building a network simulator that focuses on real-time simulations could be considered as well.

Besides, the integration of emulated network devices would add valuable features to the proposed solutions. The current system uses synthetic routers and switches. These could be replaced with device images from real network devices. As NFV becomes increasingly popular, supporting real instead of synthetic images enables network-related research and introduces a variety of new use-cases.

APPENDIX A

VIoTD API Description

Run VIoTD

This endpoint starts the virtualized device and puts the VIoTD into an idle state.

Resource URL

Method	URL
GET	/api/v1/run

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Run preparation task

This endpoint runs the preparation task on the virtualized device. After this task has finished, the VIoTD returns into an idle state automatically.

Resource URL

Method	URL
GET	/api/v1/preparation

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Start simulation

This endpoint runs the simulation commands on the virtualized device. After this task has finished, the VIoTD returns into an idle state automatically. In addition, it is possible to run the simulation on the VIoTD for a pre-defined time. Using the optional query parameter *duration*, the VIoTD stops the simulation automatically after the specified duration.

Resource URL

Method	URL
GET	/api/v1/simulation/start?duration=value

Query Parameter

Name	Description	Data type
duration	Optional. Run the simulation on the VIoTD for a pre-defined time, specified in seconds.	Integer

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Stop simulation

This endpoint stops the simulation on the virtualized device. If the simulation is not running, an error is returned. After this task has finished, the VIoTD returns into an idle state automatically.

Resource URL

Method	URL
GET	/api/v1/simulation/stop

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Shutdown VIoTD

This endpoint stops the VIoTD. It turns off the virtualized device, executes required cleanup tasks and exits the VIoTD.

Resource URL

Method	URL
GET	/api/v1/shutdown

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Health Check

This endpoint runs a health check on the virtualized device and returns its result.

Resource URL

Method	URL
GET	/api/v1/health

Response

The endpoint responds with the health information of the virtualized device and HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Example Response

```
{
  "device": "testDeviceId",
  "health": "BUSY"
}
```

Get VIoTD state

This endpoint runs the state of the VIoTD.

Resource URL

Method	URL
GET	/api/v1/state

Response

The endpoint responds with the current state of the VIoTD and HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Example Response

```
{
  "state": "SIMULATION_RUNNING"
}
```

APPENDIX B

Network Simulator API Description

Register VIoTD

This endpoint is for VIoTDs to register themselves at the network simulator.

Resource URL

Method	URL
POST	/api/v1/register

Payload

Name	Description	Data type
device_id	The unique device identifier of the VIoTD.	String
device_type	The type of the device. Possible values: vm, container, process	Enum
tap_if_name	The network interface of the VIoTD which should be used in future simulations.	String

Example Request

```
{
  "device_id": "testDeviceId",
  "device_type": "vm",
  "tap_if_name": "tap-testDeviceId"
}
```

Response

The name of the network namespace of the network simulator.

Unregister VIoTD

This endpoint is for VIoTDs to unregister themselves at the network simulator.

Resource URL

Method	URL
DELETE	/api/v1/deregister/<device-id>

URL Parameter

Name	Description
device-id	The unique device identifier of the VIoTD.

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Start network simulation

This endpoint starts ns-3. It creates the submitted network topology and establishes the channels between VIoTDs.

Resource URL

Method	URL
POST	/api/v1/simulation/start

Payload

The payload represents the network topology for the simulator in a JSON format. It contains a list of devices with device-specific configuration options and a list of networks, providing information about the connection between devices. Due to the variety of parameters and values, Section 4.2.2 provides a detailed explanation about the JSON structure, possible parameters, and allowed values. The following description is just a brief summary.

Name	Description	Data type
devices	The list of devices participating in the simulation.	String
network	The list of networks connecting the devices during the simulation.	

Example Request

```
{
  "devices": [{
    "device_id": "container1"
  }, {
    "device_id": "container2"
  }],
  "network": [{
    "network_type": "CSMA",
    "general_config": {
      "data_rate": "5000000",
      "delay": "2"
    },
    "address": {
      "ip": "10.1.1.0",
      "netmask": "255.255.255.0"
    },
    "devices": [ "container1", "container2"]
  }]
}
```

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Stop network simulation

This endpoint stops the network simulation.

Resource URL

Method	URL
GET	/api/v1/simulation/stop

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Create libvirt network

This endpoint creates a new libvirt network for VM VIoTDs according to the configuration provided in the payload.

Resource URL

Method	URL
POST	/api/v1/network/create

Payload

Name	Description	Data type
network_name	The name of the libvirt network.	String
gateway_ip	The IP address of the gateway.	String
netmask	The netmask specifying the IP subnet of the network.	String
start_ip	The lowest IP address the DHCP server should assign to any device joining the network.	String
end_ip	The highest IP address the DHCP server should assign to any device joining the network.	String

Example Request

```
{
  "network_name": "testnet",
  "gateway_ip": "10.1.1.1",
  "netmask": "255.255.255.0",
  "start_ip": "10.1.1.40",
  "end_ip": "10.1.1.50"
}
```

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

Remove libvirt network

This endpoint removes a previously created libvirt network. This endpoint does not consider VIoTDs connected to the network. If the network is removed while VIoTDs are still connected, these devices will lose their connectivity.

Resource URL

Method	URL
DELETE	/api/v1/network/remove/<network-name>

URL Parameter

Name	Description
network-name	The name of the libvirt network which should be removed.

Response

The endpoint responds with the HTTP status code 200 if the request was successful. Otherwise, a proper error message is returned.

APPENDIX C

Network Topology JSON Examples

C.1 Switch Example

The network topology in Figure C.1 shows two nodes connected via a switch, and Listing C.1 shows the corresponding JSON representation. The data rate of each CSMA link is set to 5Mbps with a delay of 2ms.

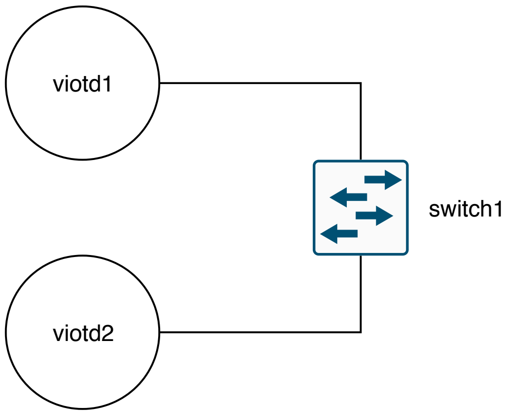


Figure C.1: Example of a network connecting two VIoTDs via a switch.

Listing C.1: JSON representation of a network connecting two VIoTDs via a switch.

```
{
  "devices": [{
    "device_id": "viotd1",
    "type": "container",
    "tap_if_name": "tap-viotd1"
  }, {
    "device_id": "viotd2",
     "type": "vm",
    "tap_if_name": "tap-viotd2"
  }, {
    "device_id": "switch1",
    "type": "switch"
  }],
  "network": [{
    "network_type": "CSMA",
    "general_config": {
      "data_rate": 5000000,
      "delay": 2
    },
    "address": {
      "ip": "10.1.1.0",
      "netmask": "255.255.255.0"
    },
    "devices": [ "viotd1", "switch1"]
  }, {
    "network_type": "CSMA",
    "general_config": {
      "data_rate": 5000000,
      "delay": 2
    },
    "address": {
      "ip": "10.1.1.0",
      "netmask": "255.255.255.0"
    },
    "devices": [ "viotd2", "switch1"]
  }]
}
```

C.2 Router and WiFi AP Example

The network topology in Figure C.2 connects two VIoTDs using a WiFi AP and a router. The corresponding JSON representation in Listing C.2 provides an example of a point-to-point connection and a WiFi network in infrastructure mode.

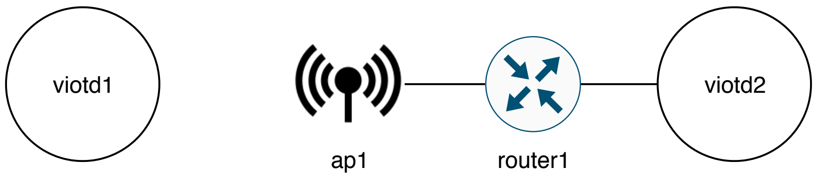


Figure C.2: Example of a network connecting VIoTDs, a router and a WiFi AP.

Listing C.2: JSON representation of a network connecting VIoTDs, a router and a WiFi AP.

```
{
  "devices": [{
    "device_id": "viotd1",
    "type": "container",
    "tap_if_name": "tap-viotd1",
    "xpos": 0.0,
    "ypos": 0.0,
    "zpos": 0.0
  }, {
    "device_id": "viotd2"
    "type": "process",
    "tap_if_name": "tap-viotd2",
  }, {
    "device_id": "ap1",
    "type": "ap",
    "xpos": 10.0,
    "ypos": 0.0,
    "zpos": 0.0
  }, {
    "device_id": "router1",
    "type": "router"
  }],
  "network": [{
    "network_type": "AP_STA",
    "general_config": {
      "ssid": "test-net"
    },
    "address": {
      "ip": "10.1.1.0",
      "netmask": "255.255.255.0"
    },
    "devices": [ "viotd1", "ap1"]
  }, {
    "network_type": "P2P",
    "general_config": {
      "data_rate": "512kbps",
      "delay": "10ms"
    },
    "address": {
      "ip": "10.1.2.0",
      "netmask": "255.255.255.0"
    },
    "devices": [ "ap1", "router1"]
  }, {
    "network_type": "CSMA",
    "general_config": {
      "data_rate": 5000000,
      "delay": 2
    },
    "address": {
      "ip": "10.1.3.0",
      "netmask": "255.255.255.0"
    },
    "devices": [ "router1", "viotd2"]
  }]
}
```

C.3 Mesh Example

The network topology in Figure C.3 shows a mesh network consisting of three VIoTDs. The JSON representation in Listing C.3 specifies the ns-3 mesh module via the network type *MESH*.

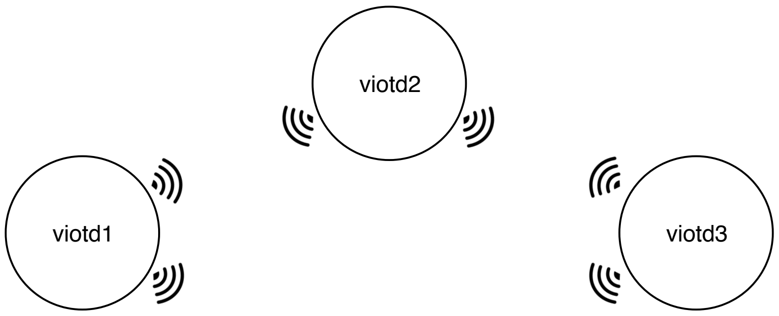


Figure C.3: Example of a mesh network.

Listing C.3: JSON representation of a mesh network using the ns-3 mesh module.

```
{
  "devices": [{
    "device_id": "viotd1",
    "type": "container",
    "tap_if_name": "tap-viotd1",
    "xpos": 0.0,
    "ypos": 10.0,
    "zpos": 0.0
  }, {
    "device_id": "viotd2",
    "type": "process",
    "tap_if_name": "tap-viotd2",
    "xpos": 10.0,
    "ypos": 0.0,
    "zpos": 0.0
  }, {
    "device_id": "viotd3",
    "type": "vm",
    "tap_if_name": "tap-viotd3",
    "xpos": 10.0,
    "ypos": 20.0,
    "zpos": 0.0
  }],
  "network": [{
    "network_type": "MESH",
    "address": {
      "ip": "10.1.1.0",
      "netmask": "255.255.255.0"
    },
    "devices": [ "viotd1", "viotd2", "viotd3"]
  }]
}
```

APPENDIX D

Raspberry Pi Emulator

D.1 Binfmt Registration Script

The script in Listing D.1 registers the byte magic for 32 and 64-bit ARM images. After executing the script, it is possible to run ARM Docker images on x86 hosts. Listing D.1 requires the static Qemu binaries[1] on the host machine to work properly.

Listing D.1: Binfmt registration script for the Raspberry Pi emulator.

```
#!/bin/bash

# constants
QEMU_PATH=/usr/bin

qemu_target_list="arm arm_be aarch64 aarch64_be"
# define magic byte sequence for arm processors (for more
    details, see: https://www.kernel.org/doc/html/latest/admin-
    guide/binfmt-misc.html)
# little endian
# 32-bit
arm_magic='\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00 \x00
    \x00\x02\x00\x28\x00'
arm_mask='\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff\xff \xff
    \xff\xff\xff\xfe\xff\xff\xff'
arm_family=arm
arm_flags='F'
```

[1] https://github.com/multiarch/qemu-user-static/blob/master/docs/developers_guide.md, last access at 2020-01-20

```
# 64-bit
aarch64_magic='\x7fELF\x02\x01\x01\x00\x00\x00\x00\x00\x00 \x00
    \x00\x00\x02\x00\xb7\x00'
aarch64_mask='\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff \xff
    \xff\xff\xff\xff\xfe\xff\xff\xff'
aarch64_family=arm
aarch64_flags='F'

# big endian
# 32-bit
arm_be_magic='\x7fELF\x01\x02\x01\x00\x00\x00\x00\x00\x00 \x00\
    x00\x00\x00\x02\x00\x28'
arm_be_mask='\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff \xff\
    xff\xff\xff\xff\xff\xfe\xff\xff'
arm_be_family=armeb
arm_be_flags=''
#64-bit
aarch64_be_magic='\x7fELF\x02\x02\x01\x00\x00\x00\x00\x00 \x00\
    x00\x00\x00\x00\x02\x00\xb7'
aarch64_be_mask='\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff \xff\
    xff\xff\xff\xff\xff\xff\xfe\xff\xff'
aarch64_be_family=armeb
aarch64_be_flags='F'

get_file_occurrences() {
  return $(find /proc/sys/fs/binfmt_misc -type f -name "qemu-
    $cpu" | wc -l);
}

binfmt_already_registered() {
  for cpu in ${qemu_target_list} ; do
    get_file_occurrences
    if [ ! $? -gt 0 ]; then
      return -1
    fi
  done
  return 0
}

check_binfmt_misc() {
  # load the binfmt_misc module
  if [ ! -d /proc/sys/fs/binfmt_misc ]; then
    if ! /sbin/modprobe binfmt_misc ; then
```

```
      exit 1
    fi
  fi
  if [ ! -f /proc/sys/fs/binfmt_misc/register ]; then
    if ! mount binfmt_misc -t binfmt_misc /proc/sys/fs/
 binfmt_misc ; then
      exit 1
    fi
  fi

  if [ ! -w /proc/sys/fs/binfmt_misc/register ] ; then
    echo "ERROR: cannot write to /proc/sys/fs/binfmt_misc/
 register"
    exit 1
  fi
}

register_binfmt_type() {
  echo ":qemu-$cpu:M::$magic:$mask:$qemu:$flags" > /proc/sys/fs
 /binfmt_misc/register
}

set_binfmts() {
  # probe cpu type
  host_family=$(uname -m)

  # register the interpreter for each cpu
  for cpu in ${qemu_target_list} ; do
    magic=$(eval echo \$${cpu}_magic)
    mask=$(eval echo \$${cpu}_mask)
    family=$(eval echo \$${cpu}_family)
    flags=$(eval echo \$${cpu}_flags)

    qemu="$QEMU_PATH/qemu-$cpu-static"

    if [ "$host_family" != "$family" ] ; then
      echo "Setting $qemu as binfmt interpreter for $cpu"
      register_binfmt_type
    fi
  done
}


```

```
# register binary types
check_binfmt_misc

if [ "${1}" = "--reset" ]; then
  shift
  find /proc/sys/fs/binfmt_misc -type f -name 'qemu-*' -exec sh
    -c 'echo -1 > {}' \;
fi

binfmt_already_registered
if [ $? -ne 0 ]; then
  set_binfmts
fi
```

D.2 Raspberry Pi Docker Image

The script in Listing D.2 creates the Docker image for the Raspberry Pi emulator. It uses the Raspbian Buster[2] OS as the base image.

Listing D.2: Script to create the Docker image for the Raspberry Pi emulator.

```
#!/bin/bash
# script based on https://hub.docker.com/r/raspbianos/stretch

debootstrap --variant=minbase --arch=armhf buster raspbian-
   buster http://archive.raspbian.org/raspbian

chroot raspbian-buster apt install -y man-db
chroot raspbian-buster locale-gen en_US.UTF-8 && dpkg-
   reconfigure locales && ls /usr/share/locale | grep -v en |
   xargs rm -rf
chroot raspbian-buster rm -rf /usr/share/doc/* /usr/share/
   common-licences/*
chroot raspbian-buster ls /usr/share/man | grep -v man | xargs
   rm -rf
chroot raspbian-buster rm -f /var/cache/apt/archives/.deb /var/
   cache/apt/archives/partial/.deb /var/cache/apt/*.bin

tar -C ./raspbian-buster -cf raspbian-buster.tar .
docker import --change "ENTRYPOINT [\"/bin/bash\"]" raspbian-
   buster.tar mle110/raspbian:buster
```

[2]https://www.raspberrypi.org/downloads/raspbian, last access at 2020-01-20

APPENDIX E

Evaluation Scripts

E.1 Simulated Sensor

Listing E.1: Python script of a simulated sensor.

```python
import argparse, socket, logging, sys
from time import sleep
from netifaces import interfaces, ifaddresses, AF_INET


logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
  format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s')


def open_sensor_data_file(path):
    return open(path, "r+")


def get_ip_address():
    ifaces = interfaces()
    ifaces.remove("lo")
    # We know that a process only has one additional interface
    # besides the loopback interface
    ipv4_data = ifaddresses(ifaces[0])[AF_INET]
    ipv4 = ipv4_data[0]['addr']
    return ipv4


```

```python
def run_sensor(data_file, port):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((get_ip_address(), port))
        s.listen()
        logging.info("Sensor started successfully. Listening on port {}".format(port))
        conn, addr = s.accept()

        while conn.fileno() > -1:
            try:
                conn.recv(1024)
                sensor_data = data_file.readline()
                logging.info("Measure temperature: %s", sensor_data)
                conn.sendall(str.encode(sensor_data))
            except BrokenPipeError:
                logging.info("Shutting down sensor.")
                conn.close()


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Sensor process for the VIoTD evaluation.')
    parser.add_argument('--data-path', dest='path', required=True, help='The path to the sensor data file.')
    parser.add_argument('--port', dest='port', required=True, type=int, help='The port the sensor should listen on.')
    args = parser.parse_args()

    sensor_data_file = open_sensor_data_file(args.path)
    run_sensor(sensor_data_file, args.port)
```

E.2 Controller Program

Listing E.2: Controller program for the evaluation use-case.

```python
import argparse, socket, requests, logging, sys
from time import sleep


logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
  format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s')


class ControllerData:
    def __init__(self, sensor_host, sensor_port, server_host, server_port, threshold):
        self.sensor_host = sensor_host
        self.sensor_port = sensor_port
        self.server_host = server_host
        self.server_port = server_port
        self.threshold = threshold

    def get_notification_url(self):
        return "http://{}:{}/notification".format(self.server_host, self.server_port)

    @classmethod
    def from_args(cls, args):
        return cls(args.sensor_host, args.sensor_port, args.server_host, args.server_port,
                   args.threshold)


def process_sensor_data(data, controller_data):
    if (data > controller_data.threshold):
        # send notification to server
        logging.info("Threshold exceeded: {}. Send notification to server.".format(data))
        payload = {
            "threshold": controller_data.threshold,
            "value": data
        }


        try:
            requests.post(url = controller_data.get_notification_url(), json = payload)
        except:
            logging.error("Cannot send notification to server")


def request_sensor_data(sock):
    sock.sendall(b'value')
    return float(sock.recv(1024).decode())


def run_controller(controller_data):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((controller_data.sensor_host, controller_data.sensor_port))
        logging.info("Start polling for sensor values.")

        while True:
            sleep(1)
            # poll sensor
            sensor_data = request_sensor_data(s)
            logging.info("Poll sensor data: {}".format(sensor_data))
            # check threshold and send notification
            # if applicable
            process_sensor_data(sensor_data, controller_data)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Controller for the VIoTD evaluation.')
    parser.add_argument('--sensor-host', dest='sensor_host', required=True, help='The host where the sensor is running.')
    parser.add_argument('--sensor-port', dest='sensor_port', required=True, type=int, help='The port where the sensor is listening on.')
    parser.add_argument('--server-host', dest='server_host', required=True, help='The host where the server is running.')
    parser.add_argument('--server-port', dest='server_port', required=True, type=int, help='The port where the server is listening on.')
    parser.add_argument('--threshold', dest='threshold',
    required=True, type=float, help='The sensor data threshold.
    Values higher than the threshold will be reported to the
    server.')
    args = parser.parse_args()

    controller_data = ControllerData.from_args(args)
    run_controller(controller_data)
```

E.3 Server Program

Listing E.3: Server program for the evaluation use-case.

```
import argparse, logging, sys
from flask import Flask, request


logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
  format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
app = Flask(__name__)


@app.route("/notification", methods=["POST"])
def notification_received():
    logging.info("Notification received. Data: {}".format(
    request.get_json()))
    return "OK"

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Server for
    the VIoTD evaluation.")
    parser.add_argument("--port", dest="port", required=True,
    type=int, help="The port the sensor should listen on.")
    args = parser.parse_args()

    app.run(host = "0.0.0.0", port = args.port)
```

E.4 Network Topology JSON

Listing E.4: JSON representation of the network functional use-case evaluation.

```
{
        "devices": [{
                "device_id": "sensor",
                "xpos": 0.0,
                "ypos": 0.0,
                "zpos": 0.0
        }, {
                "device_id": "controller",
                "xpos": 10.0,
                "ypos": 0.0,
                "zpos": 0.0
        }, {
                "device_id": "server"
        }, {
                "device_id": "router1",
                "type": "router",
                "xpos": 25.0,
                "ypos": 0.0,
                "zpos": 0.0
        }],
        "network": [{
                "network_type": "ADHOC",
                "general_config": {
                        "ssid": "sensornet"
                },
                "address": {
                        "ip": "10.1.1.0",
                        "netmask": "255.255.255.0"
                },
                "devices": [ "sensor", "controller", "router1"]
        }, {
                "network_type": "CSMA",
                "general_config": {
                        "data_rate": "5000000",
                        "delay": "2"
                },
                "address": {
                        "ip": "10.1.2.0",
                        "netmask": "255.255.255.0"
```

```
                    },
                    "devices": [ "router1", "server"]
            }]
}
```

List of Figures

List of Tables

List of Listings

Bibliography

[1] A. S. Abdul-Qawy, P. Pramod, E. Magesh, and T. Srinivasulu, “The Internet of Things (IoT): An Overview,” *International Journal of engineering Research and Applications*, vol. 1, no. 5, pp. 71–82, 2015.

[2] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” *Computer Networks*, vol. 54, pp. 2787–2805, 2010.

[3] M. Bures, T. Cerny, and B. S. Ahmed, “Internet of Things: Current Challenges in the Quality Assurance and Testing Methods,” in *Information Science and Applications*, pp. 625–634, Springer, 2019.

[4] A. Taivalsaari and T. Mikkonen, “A Roadmap to the Programmable World: Software Challenges in the IoT Era,” *IEEE Software*, vol. 34, no. 1, pp. 72–80, 2017.

[5] P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann, “A Distributed Test System Architecture for Open-source IoT Software,” in *Workshop on IoT Challenges in Mobile and Industrial Systems*, pp. 43–48, ACM, 2015.

[6] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,” *Communications Surveys & Tutorials*, vol. 17, pp. 2347–2376, 2015.

[7] J. A. Stankovic, “Research Directions for the Internet of Things,” *Internet of Things Journal*, vol. 1, pp. 3–9, 2014.

[8] E. Ojie and E. Pereira, “Simulation Tools in Internet of Things: A Review,” in *International Conference on Internet of Things and Machine Learning*, pp. 1–7, ACM, 2017.

[9] S. Sotiriadis, N. Bessis, E. Asimakopoulou, and N. Mustafee, “Towards Simulating the Internet of Things,” in *International Conference on Advanced Information Networking and Applications Workshops*, pp. 444–448, IEEE, 2014.

[10] G. Kecskemeti, G. Casale, D. N. Jha, J. Lyon, and R. Ranjan, “Modelling and Simulation Challenges in Internet of Things,” *IEEE Cloud Computing*, vol. 4, no. 1, pp. 62–69, 2017.

[11] G. D'Angelo, S. Ferretti, and V. Ghini, "Smart multihoming in smart shires: Mobility and communication management for smart services in countrysides," in *Symposium on Computers and Communication*, pp. 970–975, IEEE, 2016.

[12] G. D'Angelo, S. Ferretti, and V. Ghini, "Simulation of the Internet of Things," in *International Conference on High Performance Computing Simulation*, pp. 1–8, IEEE, 2016.

[13] D. T. Davis, T. H. Chung, M. R. Clement, and M. A. Day, "Consensus-based data sharing for large-scale aerial swarm coordination in lossy communications environments," in *International Conference on Intelligent Robots and Systems*, pp. 3801–3808, IEEE, 2016.

[14] X. Li, Q. Huang, and D. Wu, "Distributed Large-Scale Co-Simulation for IoT-Aided Smart Grid Control," *IEEE Access*, vol. 5, pp. 19951–19960, 2017.

[15] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, "Internet of Things (IoT): Research, Simulators, and Testbeds," *Internet of Things Journal*, vol. 5, no. 3, pp. 1637–1647, 2018.

[16] Y. Kuwabara, T. Yokotani, and H. Mukai, "Hardware emulation of IoT devices and verification of application behavior," in *Asia-Pacific Conference on Communications*, pp. 1–6, IEEE, 2017.

[17] A. Mäkinen, J. Jiménez, and R. Morabito, "ELIoT: Design of an emulated IoT platform," in *International Symposium on Personal, Indoor, and Mobile Radio Communications*, pp. 1–7, IEEE, 2017.

[18] A. Detti, G. Tropea, G. Rossi, J. A. Martinez, A. F. Skarmeta, and H. Nakazato, "Virtual IoT Systems: Boosting IoT Innovation by Decoupling Things Providers and Applications Developers," in *Global IoT Summit*, pp. 1–6, IEEE, 2019.

[19] E. J. Marinissen, Y. Zorian, M. Konijnenburg, C.-T. Huang, P.-H. Hsieh, P. Cockburn, J. Delvaux, V. Rožić, B. Yang, D. Singelée, I. Verbauwhede, C. Mayor, R. van Rijsinge, and C. Reyes, "IoT: Source of test challenges," in *IEEE European Test Symposium*, pp. 1–10, IEEE, 2016.

[20] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX Annual Technical Conference*, pp. 305–319, USENIX Association, 2014.

[21] A. Ailijiang, A. Charapko, and M. Demirbas, "Consensus in the Cloud: Paxos Systems Demystified," in *International Conference on Computer Communication and Networks*, pp. 1–10, IEEE, 2016.

[22] S. Kar, G. Hug, J. Mohammadi, and J. M. Moura, "Distributed State Estimation and Energy Management in Smart Grids: A Consensus+ Innovations Approach," *Journal on Selected Topics in Signal Processing*, vol. 8, no. 6, pp. 1022–1038, 2014.

[23] V. Poirot, B. A. Nahas†, and O. Landsiedel, "Paxos Made Wireless: Consensus in the Air," in *International Conference on Embedded Wireless Systems and Networks*, pp. 1–12, Junction Publishing, 2019.

[24] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox, and J. Singer, "Commodity single board computer clusters and their applications," *Future Generation Computer Systems*, vol. 89, pp. 201–212, 2018.

[25] A. Rayes and S. Salam, *Internet of Things From Hype to Reality*. Springer, 2 ed., 2019.

[26] F. Khodadadi, A. Dastjerdi, and R. Buyya, *Internet of Things: Principles and Paradigms*. Morgan Kaufmann, 2016.

[27] F. Firouzi, B. Farahani, M. Weinberger, G. DePace, and F. S. Aliee, *Intelligent Internet of Things: From Device to Fog and Cloud*. Springer, 2020.

[28] F. Assaderaghi, G. Chindalore, B. Ibrahim, H. de Jong, M. Joye, S. Nassar, W. Steinbauer, M. Wagner, and T. Wille, "Privacy and security: Key requirements for sustainable IoT growth," in *Symposium on VLSI Technology*, pp. 8–13, IEEE, 2017.

[29] Alliance for Internet of Things Innovation, "IoT LSP Standard Framework Concepts (Release 2.9)," 2019. [Online]. Available: https://aioti.eu. [Accessed: Mar. 13, 2020].

[30] European Telecommunications Standards Institute, "IoT Standards landscape and future evolutions," 2016. [Online]. Available: http://www.etsi.org. [Accessed: Mar. 13, 2020].

[31] I. U. Sari and D. Karlikaya, "Evaluation of User Preference Criteria on Smart Technologies for Smart Buildings," in *Intelligent and Fuzzy Techniques in Big Data Analytics and Decision Making*, pp. 713–721, Springer, 2020.

[32] D. Minoli, K. Sohraby, and B. Occhiogrosso, "IoT Considerations, Requirements, and Architectures for Smart Buildings—Energy Optimization and Next-Generation Building Management Systems," *Internet of Things Journal*, vol. 4, no. 1, pp. 269–283, 2017.

[33] A. Kusiak, "Smart manufacturing," *International Journal of Production Research*, vol. 56, no. 1-2, pp. 508–517, 2018.

[34] H. S. Kang, J. Y. Lee, S. Choi, H. Kim, J. H. Park, J. Y. Son, B. H. Kim, and S. D. Noh, "Smart Manufacturing: Past Research, Present Findings, and Future Directions," *International Journal of Precision Engineering and Manufacturing-Green Technology*, vol. 3, pp. 111–128, 2016.

[35] P. Pype, G. Daalderop, E. Schulz-Kamm, E. Walters, G. Blom, and S. Westermann, *Automated Driving: Safer and More Efficient Future Driving*. Springer, 2017.

[36] R. Faria, L. Brito, K. Baras, and J. Silva, "Smart mobility: A survey," in *International Conference on Internet of Things for the Global Community*, pp. 1–8, IEEE, 2017.

[37] A. Abbas, S. U. Khan, and A. Y. Zomaya, *Handbook of Large-Scale Distributed Computing in Smart Healthcare*. Springer, 2017.

[38] S. B. Baker, W. Xiang, and I. Atkinson, "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities," *IEEE Access*, vol. 5, pp. 26521–26544, 2017.

[39] H. Lund, P. A. Østergaard, D. Connolly, and B. V. Mathiesen, "Smart energy and smart energy systems," *Energy*, vol. 137, pp. 556 – 565, 2017.

[40] Q. Le-Dang and T. Le-Ngoc, *Handbook of Smart Cities: Software Services and Cyber Infrastructure*. Springer, 2018.

[41] S. Dustdar, S. Nastić, and O. Šćekić, *Smart Cities: The Internet of Things, People and Systems*. Springer, 2017.

[42] H. Yoon, S. H. Park, and K. T. Lee, "Lightful user interaction on smart wearables," *Personal and Ubiquitous Computing*, vol. 20, no. 6, pp. 973–984, 2016.

[43] W. Sun, J. Liu, and H. Zhang, "When Smart Wearables Meet Intelligent Vehicles: Challenges and Future Directions," *Wireless Communications*, vol. 24, no. 3, pp. 58–65, 2017.

[44] P. Jayaraman, A. Yavari, D. Georgakopoulos, A. Morshed, and A. Zaslavsky, "Internet of Things Platform for Smart Farming: Experiences and Lessons Learnt," *Sensors*, vol. 16, no. 11, p. 1884, 2016.

[45] P. Tripicchio, M. Satler, G. Dabisias, E. Ruffaldi, and C. A. Avizzano, "Towards Smart Farming and Sustainable Agriculture with Drones," in *International Conference on Intelligent Environments*, pp. 140–143, IEEE, 2015.

[46] M. Ryu, J. Yun, T. Miao, I. Ahn, S. Choi, and J. Kim, "Design and implementation of a connected farm for smart farming system," in *Sensors*, pp. 1–4, IEEE, 2015.

[47] M. Papert and A. Pflaum, "Development of an Ecosystem Model for the Realization of Internet of Things (IoT) Services in Supply Chain Management," *Electronic Markets*, vol. 27, no. 2, pp. 175–189, 2017.

[48] T. Salman and R. Jain, "Networking Protocols and Standards for Internet of Things," *Internet of Things and Data Analytics Handbook*, vol. 2015, pp. 215–238, 2015.

[49] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi, "Internet of Things (IoT) communication protocols: Review," in *International Conference on Information Technology*, pp. 685–690, IEEE, 2017.

[50] "IoT Technology Guidebook," 2020. [Online]. Available: https://www.postscapes.com/internet-of-things-technologies/. [Accessed: Mar. 15, 2020].

[51] C. Sobin, "A Survey on Architecture, Protocols and Challenges in IoT," *Wireless Personal Communications*, pp. 1–47, 2020.

[52] J. Olsson, "6lowpan demystified," 2014. [Online]. Available: https://www.ti.com/lit/wp/swry013/swry013.pdf. [Accessed: Mar. 15, 2020].

[53] D. Dujovne, T. Watteyne, X. Vilajosana, and P. Thubert, "6TiSCH: deterministic IP-enabled industrial internet (of things)," *Communications Magazine*, vol. 52, no. 12, pp. 36–41, 2014.

[54] A. Aijaz and A. H. Aghvami, "Cognitive Machine-to-Machine Communications for Internet-of-Things: A Protocol Stack Perspective," *Internet of Things Journal*, vol. 2, no. 2, pp. 103–112, 2015.

[55] C. Gomez, J. Oller, and J. Paradells, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," *Sensors*, vol. 12, no. 9, pp. 11734–11753, 2012.

[56] D. De Guglielmo, S. Brienza, and G. Anastasi, "IEEE 802.15.4e: A survey," *Computer Communications*, vol. 88, pp. 1–24, 2016.

[57] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, "Understanding the Limits of LoRaWAN," *Communications Magazine*, vol. 55, no. 9, pp. 34–40, 2017.

[58] J. de Carvalho Silva, J. J. P. C. Rodrigues, A. M. Alberti, P. Solic, and A. L. L. Aquino, "LoRaWAN — A low power WAN protocol for Internet of Things: A review and opportunities," in *International Multidisciplinary Conference on Computer and Energy Science*, pp. 1–6, IEEE, 2017.

[59] T. Adegbija, A. Rogacs, C. Patel, and A. Gordon-Ross, "Microprocessor Optimizations for the Internet of Things: A Survey," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 7–20, 2018.

[60] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2016.

[61] M. H. Weik, *Computer Science and Communications Dictionary*. Springer, 2001.

[62] K. N. Patel and R. h. Jhaveri, "A Survey on Emulation Testbeds for Mobile Ad-hoc Networks," *Procedia Computer Science*, vol. 45, pp. 581–591, 2015.

[63] M. Kropff, T. Krop, M. Hollick, P. S. Mogre, and R. Steinmetz, "A survey on real world and emulation testbeds for mobile ad hoc networks," in *International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities.*, pp. 448–453, IEEE, 2006.

[64] I. McGregor, "The relationship between simulation and emulation," in *Proceedings of the Winter Simulation Conference*, pp. 1683–1688, IEEE, 2002.

[65] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues," in *International Conference on Computer and Network Technology*, pp. 222–226, IEEE, 2010.

[66] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *International Conference on Cloud Engineering*, pp. 386–393, IEEE, 2015.

[67] R. Y. Ameen and A. Y. Hamo, "Survey of Server Virtualization," *International Journal of Computer Science and Information Security*, vol. 11, no. 3, pp. 65–74, 2013.

[68] F. Rodríguez-Haro, F. Freitag, L. Navarro, E. Hernánchez-sánchez, N. Farías-Mendoza, J. A. Guerrero-Ibáñez, and A. González-Potes, "A summary of virtualization techniques," *Procedia Technology*, vol. 3, pp. 267–272, 2012.

[69] R. Morabito, "Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[70] D. Rountree, *Security for Microsoft Windows System Administrators*. Syngress, 2011.

[71] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *International Conference on Local Computer Networks*, pp. 455–462, IEEE, 2004.

[72] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Conference on Computer Communications Workshops*, pp. 79–80, IEEE, 2013.

[73] "libvirt Docs." [Online]. Available: https://libvirt.org/docs.html. [Accessed: Mar. 18, 2020].

[74] "libvirt Wiki." [Online]. Available: https://wiki.libvirt.org. [Accessed: Mar. 18, 2020].

[75] W. D. Ashley, D. Berrange, C. Lalancette, L. Stump, D. Veillard, D. Coulson, D. Jorm, and S. Radvan, "libvirt Application Development Guide Using Python." [Online]. Available: https://libvirt.org/docs/libvirt-appdev-guide-python/en-US/html/. [Accessed: Mar. 18, 2020].

[76] S. Cheruvu, A. Kumar, N. Smith, and D. M. Wheeler, *Demystifying Internet of Things Security: Successful IoT Device/Edge and Platform Security Deployment*. Apress, 2020.

[77] "QEMU User Documentation." [Online]. Available: https://qemu.weilnetz.de/doc/qemu-doc.html. [Accessed: Mar. 18, 2020].

[78] J. Lai, J. Tian, D. Jiang, J. Sun, and K. Zhang, "A Hybrid Virtualization Approach to Emulate Heterogeneous Network Nodes," in *Simulation Tools and Techniques*, pp. 228–237, Springer, 2019.

[79] M. Becker, H. Zabel, and W. Mueller, "A Mixed Level Simulation Environment for Stepwise RTOS Software Refinement," in *Distributed, Parallel and Biologically Inspired Systems*, pp. 145–156, Springer, 2010.

[80] N. Poulton, *Docker Deep Dive*. Leanpub, 2018.

[81] "Docker Documentation." [Online]. Available: https://docs.docker.com. [Accessed: Mar. 18, 2020].

[82] "Linux Namespaces." [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html. [Accessed: Mar. 18, 2020].

[83] "Linux Bridge." [Online]. Available: https://wiki.linuxfoundation.org/networking/bridge. [Accessed: Mar. 19, 2020].

[84] "Linux TUN/TAP device driver." [Online]. Available: https://www.kernel.org/doc/Documentation/networking/tuntap.txt. [Accessed: Mar. 19, 2020].

[85] "Linux Veth." [Online]. Available:http://man7.org/linux/man-pages/man4/veth.4.html. [Accessed: Mar. 19, 2020].

[86] "Linux Network Namespaces." [Online]. Available: http://man7.org/linux/man-pages/man7/network_namespaces.7.html. [Accessed: Mar. 19, 2020].

[87] "Linux Iptables." [Online]. Available: http://man7.org/linux/man-pages/man8/iptables.8.html. [Accessed: Mar. 19, 2020].

[88] "ns-3 Manual." [Online]. Available: https://www.nsnam.org/docs/release/3.30/manual/ns-3-manual.pdf. [Accessed: Mar. 19, 2020].

[89] "ns-3 Models." [Online]. Available: https://www.nsnam.org/docs/release/3.30/models/ns-3-model-library.pdf. [Accessed: Mar. 19, 2020].

[90] “ns-3 Tutorial.” [Online]. Available: https://www.nsnam.org/docs/release/3.30/tutorial/ns-3-tutorial.pdf. [Accessed: Mar. 19, 2020].

[91] “ns-3 Training.” [Online]. Available: https://slideplayer.com/slide/12553294. [Accessed: Mar. 19, 2020].

[92] L. Czaja, *Introduction to Distributed Computer Systems : Principles and Features*. Springer, 2018.

[93] H. Bagci and A. Kara, “A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication.,” in *International Conference on Software Technologies*, pp. 117–124, SciTePress, 2016.

[94] R. Abernethy, *Programmer’s Guide to Apache Thrift*. Manning Publications, 2019.

[95] “Micronaut.” [Online]. Available: https://docs.micronaut.io/latest/guide/index.html. [Accessed: Mar. 19, 2020].

[96] G. Z. Papadopoulos, A. Gallais, G. Schreiner, E. Jou, and T. Noel, “Thorough IoT testbed characterization: From proof-of-concept to repeatable experimentations,” *Computer Networks*, vol. 119, pp. 86–101, 2017.

[97] V. Gupta, S. K. Devar, N. H. Kumar, and K. P. Bagadi, “Modelling of IoT Traffic and Its Impact on LoRaWAN,” in *Global Communications Conference*, pp. 1–6, IEEE, 2017.

[98] F. Al-Turjman, E. Ever, and H. Zahmatkesh, “Small Cells in the Forthcoming 5G/IoT: Traffic Modelling and Deployment Overview,” *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 28–65, 2019.

[99] A. Strielkina, D. Uzun, and V. Kharchenko, “Modelling of healthcare IoT using the queueing theory,” in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, vol. 2, pp. 849–852, IEEE, 2017.

[100] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, and M. Viroli, “Modelling and simulation of Opportunistic IoT Services with Aggregate Computing,” *Future Generation Computer Systems*, vol. 91, pp. 252–262, 2019.

[101] G. Z. Papadopoulos, J. Beaudaux, A. Gallais, T. Noël, and G. Schreiner, “Adding value to WSN simulation using the IoT-LAB experimental platform,” in *Wireless and Mobile Computing, Networking and Communications*, pp. 485–490, IEEE, 2013.

[102] A. Tonneau, N. Mitton, and J. Vandaele, “A Survey on (mobile) Wireless Sensor Network Experimentation Testbeds,” in *International Conference on Distributed Computing in Sensor Systems*, pp. 263–268, IEEE, 2014.

[103] J. Ma, J. Wang, and T. Zhang, “A Survey of Recent Achievements for Wireless Sensor Networks Testbeds,” in *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pp. 378–381, IEEE, 2017.

[104] H. Isakovic, D. Ratasich, C. Hirsch, M. Platzer, B. Wally, T. Rausch, D. Nickovic, W. Krenn, G. Kappel, S. Dustdar, *et al.*, "CPS/IoT Ecosystem: A Platform for Research and Education," in *Cyber Physical Systems. Model-Based Design*, pp. 206–213, Springer, 2018.

[105] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT IoT-LAB: A large scale open experimental IoT testbed," in *World Forum on Internet of Things*, pp. 459–464, IEEE, 2015.

[106] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, and D. Pfisterer, "SmartSantander: IoT experimentation over a smart city testbed," *Computer Networks*, vol. 61, pp. 217–238, 2014.

[107] H. Hellbrück, M. Pagel, A. Köller, D. Bimschas, D. Pfisterer, and S. Fischer, "Using and operating wireless sensor network testbeds with WISEBED," in *Annual Mediterranean Ad Hoc Networking Workshop*, pp. 171–178, IEEE, 2011.

[108] M. Sharif and A. Sadeghi-Niaraki, "Ubiquitous sensor network simulation and emulation environments: A survey," *Journal of Network and Computer Applications*, vol. 93, pp. 150–181, 2017.

[109] A. Nayyar and R. Singh, "A Comprehensive Review of Simulation Tools for Wireless Sensor Networks (WSNs)," *Journal of Wireless Networking and Communications*, vol. 5, no. 1, pp. 19–47, 2015.

[110] A. Varga and R. Hornig, "An Overview of the OMNeT++ Simulation Environment," in *International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pp. 1–10, Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 2008.

[111] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network Simulations with the ns-3 Simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[112] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA," in *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pp. 641–648, IEEE, 2006.

[113] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *Journal of Systems Architecture*, vol. 72, pp. 93–107, 2017.

[114] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[115] R. Buyya and S. N. Srirama, *Fog and Edge Computing: Principles and Paradigms*. Wiley, 2019.

[116] M. A. To, M. Cano, and P. Biba, "DOCKEMU – A Network Emulation Tool," in *International Conference on Advanced Information Networking and Applications Workshops*, pp. 593–598, IEEE, 2015.

[117] A. R. Portabales and M. L. e. Nores, "Dockemu: An IoT Simulation Framework Based on Linux Containers and the ns-3 Network Simulator — Application to CoAP IoT Scenarios," in *Simulation and Modeling Methodologies, Technologies and Applications*, pp. 54–82, Springer, 2018.

[118] E. Petersen, G. Cotto, and M. Antonio To, "Dockemu 2.0: Evolution of a Network Emulation Tool," in *Central America and Panama Convention*, pp. 1–6, IEEE, 2019.

[119] "ns-3 IPv6 Bug." [Online]. Available: https://groups.google.com/forum/#!topic/ns-3-users/5F7cZtL2ibo. [Accessed: Mar. 25, 2020].

[120] S. Badiger, S. Baheti, and Y. Simmhan, "VIoLET: A Large-Scale Virtual Environment for Internet of Things," in *Parallel Processing*, pp. 309–324, Springer, 2018.

[121] J. Lai, J. Tian, R. Liu, Z. Yang, and D. Jiang, "A Hybrid Virtualization Approach to Emulate Network Nodes of Heterogeneous Architectures," *Mobile Networks and Applications*, pp. 1–13, 2020.

[122] G. D'Angelo, S. Ferretti, and V. Ghini, "Modeling the internet of things: a simulation perspective," in *International Conference on High Performance Computing Simulation*, pp. 18–27, IEEE, 2017.

[123] G. D'Angelo, S. Ferretti, and V. Ghini, "Multi-level simulation of internet of things on smart territories," *Simulation Modelling Practice and Theory*, vol. 73, pp. 3–21, 2017.

[124] A. Čolaković and M. Hadžialić, "Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues," *Computer Networks*, vol. 144, pp. 17–39, 2018.

[125] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[126] D. Carvin, P. Owezarski, and P. Berthou, "A generalized distributed consensus algorithm for monitoring and decision making in the IoT," in *International Conference on Smart Communications in Network Technologies*, pp. 1–6, IEEE, 2014.

[127] J. Qin, Q. Ma, Y. Shi, and L. Wang, "Recent Advances in Consensus of Multi-Agent Systems: A Brief Survey," *Transactions on Industrial Electronics*, vol. 64, no. 6, pp. 4972–4983, 2017.

[128] C. Chen, S. Zhu, X. Guan, and X. S. Shen, *Wireless Sensor Networks: Distributed Consensus Estimation*. Springer, 2014.

[129] S. Li, G. Oikonomou, T. Tryfonas, T. M. Chen, and L. D. Xu, "A Distributed Consensus Algorithm for Decision Making in Service-Oriented Internet of Things," *Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1461–1468, 2014.

[130] D. Ongaro, *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[131] S. Li, S. Zhao, P. Yang, P. Andriotis, L. Xu, and Q. Sun, "Distributed Consensus Algorithm for Events Detection in Cyber-Physical Systems," *Internet of Things Journal*, vol. 6, no. 2, pp. 2299–2308, 2019.

[132] G. Colistra, V. Pilloni, and L. Atzori, "Task allocation in group of nodes in the IoT: A consensus approach," in *International Conference on Communications*, pp. 3848–3853, IEEE, 2014.

[133] K. Li, S. E. Li, F. Gao, Z. Lin, J. Li, and Q. Sun, "Robust Distributed Consensus Control of Uncertain Multi-Agents Interacted by Eigenvalue-Bounded Topologies," *Internet of Things Journal*, pp. 1–1, 2020.

[134] K. Yeow, A. Gani, R. W. Ahmad, J. J. P. C. Rodrigues, and K. Ko, "Decentralized Consensus for Edge-Centric Internet of Things: A Review, Taxonomy, and Research Issues," *IEEE Access*, vol. 6, pp. 1513–1524, 2018.

[135] M. Méndez, F. G. Tinetti, A. M. Duran, D. A. Obon, and N. G. Bartolome, "Distributed Algorithms on IoT Devices: Bully Leader Election," in *International Conference on Computational Science and Computational Intelligence*, pp. 1351–1355, IEEE, 2017.

[136] L. Lamport, "The Part-Time Parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, 1998.

[137] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable Cross-Language Services Implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[138] J. Shah and B. Mishra, "IoT enabled environmental monitoring system for smart cities," in *International Conference on Internet of Things and Applications*, pp. 383–388, IEEE, 2016.

[139] T. Adegbija, A. Rogacs, C. Patel, and A. Gordon-Ross, "Enabling Right-Provisioned Microprocessor Architectures for the Internet of Things," in *International Mechanical Engineering Congress and Exposition*, vol. 14, American Society of Mechanical Engineers, 2015.

[140] S. Li, L. Da Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[141] A. Haroon, M. A. Shah, Y. Asim, W. Naeem, M. Kamran, and Q. Javaid, "Constraints in the IoT: The World in 2020 and Beyond," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 11, pp. 252–271, 2016.

[142] A. Triantafyllou, P. Sarigiannidis, and T. D. Lagkas, "Network Protocols, Schemes, and Mechanisms for Internet of Things (IoT): Features, Open Challenges, and Trends," *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1–24, 2018.

[143] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMCloud," *Communications of the ACM*, vol. 54, no. 7, pp. 121–130, 2011.

[144] R. Lara-Cueva, D. Benítez, C. Fernández, and C. Morales, "Performance Analysis of Wireless Network Modes in Conformance with IEEE 802.11b and WDS," in *Asia-Pacific Conference on Computer Aided System Engineering*, pp. 370–373, IEEE, 2015.

[145] J. H. Lam, S.-G. Lee, and W. K. Tan, "Performance Evaluation of Multi-Channel Wireless Mesh Networks with Embedded Systems," *Sensors*, vol. 12, no. 1, pp. 500–517, 2012.

[146] G. Shah, R. Valiente, N. Gupta, S. O. Gani, B. Toghi, Y. P. Fallah, and S. D. Gupta, "Real-Time Hardware-In-the-Loop Emulation Framework for DSRC-based Connected Vehicle Applications," in *Connected and Automated Vehicles Symposium*, pp. 1–6, IEEE, 2019.