INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Evaluation of ZMTP in Constrained Environments

Theresa Müller

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Evaluation of ZMTP in Constrained Environments

Theresa Müller

Supervision:	Prof. Dr. Dieter Kranzlmüller
Advisors:	Dr. Nils gentschen Felde Maximilian Höb
Date:	January, 10th 2019

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 10. Januar 2019

.....

(Unterschrift des Kandidaten)

Abstract

As the prevalence of constrained environments progressed over the past few years, the demand for distributed computing approaches that could potentially remedy the limited processing power of such networks emerged. Since constrained environments, such as the Internet of Things or Wireless Sensor Networks, are typically composed of nodes that exhibit severe restrictions on computational power, memory capacity and energy supply, resource scarcity is a fundamental characteristic of constrained environments. This thesis therefore introduces the ZeroMQ Message Transport Protocol (ZMTP), which enables distributed computing through peer-to-peer networking and is based on TCP. In order to assess the scalability of ZMTP in constrained environments, a protocol port tailored to the RIOT operating system is used in the course of this thesis. An experimental approach comprising a series of experiments with runtime measurements is conducted by means of two distributed applications that constitute representative use cases. The consequent analysis and evaluation of the measurement results reveal an undesired scaling behavior of ZMTP, that is an insufficient performance improvement mainly caused by the vast communication overhead, which limits the speedup through parallelization within both use cases. As a result, the protocol exhibits a rather weak scalability in general, so with respect to these two use cases, ZMTP is considered not suitable for the purpose of efficiently realizing distributed computing in constrained environments.

Zusammenfassung

Während die zunehmende Verbreitung von Constrained Environments in den letzten Jahren stetig vorangeschritten ist, stieg auch der Bedarf an verteilten Rechenansätzen in solchen Netzwerken, um der hierbei auftretenden eingeschränkten Rechenleistung entgegenzuwirken. Da Constrained Environments, wie beispielsweise das Internet der Dinge oder sogenannte Sensornetze, auf Geräten basieren, die meistens nur über eine begrenzte Rechenleistung, Speicherkapazität und Energiezufuhr verfügen, ist Ressourcenknappheit eine fundamentale Eigenschaft solcher Netzwerke. Im Rahmen dieser Arbeit wird das ZeroMQ Message Transport Protocol (ZMTP) vorgestellt, das verteiltes Rechnen durch TCP basierte Peer-to-Peer Kommunikation ermöglicht. Um ZMTP in Constrained Environments anwenden zu können, wird ein Port des Protokolls für das Betriebssystem RIOT verwendet. Mit Hilfe eines experimentellen Ansatzes kann somit die Skalierbarkeit von ZMTP in Constrained Environments untersucht und evaluiert werden. Zu diesem Zwecke wird anhand von zwei repräsentativen Anwendungsfällen, die unterschiedliche, verteilte Applikationen darstellen, eine Reihe von Versuchen inklusive Laufzeitmessungen durchgeführt. Die anschließende Analyse und Auswertung der Messergebnisse zeigt, dass ZMTP nicht das gewünschte Skalierungsverhalten aufweist, da im Allgemeinen keine effiziente Laufzeitverbesserung durch Skalierung erzielt wird. Dies lässt sich auf den hohen Kommunikationsoverhead zurückzuführen, der die Beschleunigung der Berechnung durch Parallelisierung generell einschränkt. Aufgrund der niedrigen Skalierbarkeit des Protokolls im Bezug auf die beiden untersuchten Anwendungsfälle, kann ZMTP für diese Applikationen nicht effizient eingesetzt werden, um verteiltes Rechnen in Constrained Environments zu realisieren.

Contents

1. Introduction **1**

2. Related work **5**

3. Background information on constrained environments **9**
3.1. Constrained devices . . . 9
3.1.1. Definition and classification . . . 9
3.1.2. The IoT protocol stack . . . 11
3.1.3. Fields of application . . . 13
3.1.4. Challenges . . . 13
3.2. The RIOT operating system . . . 14
3.2.1. Overview and features . . . 14
3.2.2. Structure . . . 15
3.2.3. The GNRC network stack . . . 16
3.2.4. GNRC TCP . . . 17
3.3. The ZeroMQ Message Transport Protocol . . . 21
3.3.1. ZeroMQ . . . 21
3.3.2. ZMTP overview . . . 22

4. Experimental approach **29**
4.1. General methodology . . . 29
4.1.1. Conceptual Design . . . 29
4.1.2. Problem space definition . . . 30
4.2. Measurands . . . 31
4.2.1. Execution time . . . 31
4.2.2. Power consumption . . . 32
4.3. Representative use cases . . . 33
4.3.1. Use case 1 - Prime number count . . . 35
4.3.2. Use case 2 - Word and line count . . . 37
4.4. Setups . . . 39
4.4.1. Test environment . . . 39
4.4.2. Test configurations and scaling scenarios . . . 40
4.5. Test procedure . . . 44

5. Evaluation and scaling conclusions **45**
5.1. Measurement accuracy and standard deviation . . . 45
5.2. Measurement results and observations . . . 48
5.2.1. Execution time results . . . 48
5.2.2. Power consumption results . . . 64
5.3. Potential causes . . . 64

5.4. Consequent scaling conclusions 67

6. General conclusion and outlook **69**

List of Figures **71**

List of Tables **73**

List of Listings **75**

Bibliography **77**

Appendix **81**
A. Plots of the remaining scenarios 81
B. Excerpt of raw measurement data 87

1. Introduction

The Internet of Things (IoT) gained a lot of importance and attention over the past few years and there is a clear trend towards a tighter and more widespread interconnection among numerous embedded devices, reaching from smart monitoring devices to pervasive wearables and everyday objects, that are exchanging information over the internet in order to provide certain services. The internet already constitutes an indispensable part of our daily lives, but its role is changing due to the progressive emergence and dissemination of the Internet of Things, which is even referred to as the next technological revolution, as stated in [GBMP13]. This assertion is being supported by the fact that the number of interconnected devices used in the consumer and business sector reached nearly 6.4 billion in 2016, while it is estimated to exceed a threshold of 20.7 billion by 2020, according to Gartner[1].

Although the range of devices that are being deployed into the internet is immense and therefore the network architecture is composed of heterogeneous nodes, many of these embedded devices exhibit more or less severe resource constraints. According to the authors of [HBPT16], these constraints refer to hardware resources like memory capacity, computational power and energy supply. For years, the internet was characterized by stationary high-performance nodes such as servers and personal computers forming fast and reliable networks. But with the rise of low-end embedded devices, fundamentally different network infrastructures are created, which mainly comprise constrained nodes with low processing power, as described in [Bru16]. The interconnection between those constrained devices creates heterogeneous constrained environments, such as Wireless Sensor Networks (WSNs), where nodes generally communicate via low-power wireless transmission technologies. Following [BEK14], this results in unreliable, lossy networks with low throughput, which impose limitations on the applicability of constrained environments and lead to newly emerging demands and challenges. As derived from [LKH+18], novel technologies needed to be developed, for example new wireless communication technologies, IoT-tailored operating systems and protocol standards that enable the extension of the conventional internet with respect to embedded devices. These advancements were made in order to address the new challenges and requirements arising from the comprehensive deployment of constrained environments.

According to [MW16] and [Iwa16], the limited performance of constrained devices is expected to last, since Moore's law is no longer feasible in the future. As a rule of thumb, Moore's law generally implies that processors get smaller, cheaper, as well as more and more powerful, but due to the physical boundaries of the micro-controllers which are deployed on constrained devices, it is anticipated that this law does not apply to these kinds of devices, so their processing power will probably not increase significantly in the next few years. As a result of the continuous resource scarcity, which imposes restrictions on executing cer-

[1] https://www.gartner.com/newsroom/id/3165317

tain computation-intensive tasks, and with the constantly growing number of interconnected devices in mind, the need for distributed computing in constrained environments is increasing. In this way, constrained nodes could work together and thus jointly reach a particular goal.

The ZeroMQ Message Transport Protocol (ZMTP) is a transport layer protocol that defines the messaging between two peers and is based upon TCP and IPC. As such, it presents an appropriate protocol to enable distributed computing in terms of peer-to-peer networking, but so far, it was rarely used in constrained environments. In order to deploy ZMTP on constrained nodes, a protocol port for the RIOT operating system will serve as the basis for analyzing the scalability of this specific protocol in terms of distributed applications. The purpose of this thesis is therefore to ascertain the scaling behavior of the ZeroMQ Message Transport Protocol in constrained environments, which will be accomplished by an experimental approach with two representative use cases as scientific methodology. By varying the numbers of workers and the total input size within several distinct scenarios, strong and weak scaling will be covered in the course of this thesis. During the experiments, runtime measurements will be performed, whereby the execution time will be measured as the primary value and the power consumption will be monitored as a secondary measurand. All measurements will be analyzed and evaluated afterwards in order to be able to draw resilient scaling conclusions from the results. If possible, these conclusions shall also include a generalization predication.

Structure of the thesis

The remainder of this thesis is structured as follows. The next chapter, namely Chapter 2, covers related work on distributed computing in constrained environments by reviewing previously published literature and approaches proposed by different authors, while pointing out the differences of those approaches compared to the work done in the course of this thesis.

Chapter 3 provides useful background information on constrained environments especially focusing on common IoT technologies that were used within the experimental approach of this thesis. It includes a way to classify constrained devices, introduces new protocol standards designed particularly for constrained environments and an IoT-tailored operating system. Additionally, it comprises an overview of the ZeroMQ Message Transport Protocol and the port of this protocol for the RIOT operating system.

In Chapter 4, the experimental approach that is used for this thesis in order to assess the scaling behavior of ZMTP in constrained environments is presented in detail. It covers the general methodology, the representative use cases that are developed for the experiments, as well as the measured values that are obtained though the runtime measurements and investigated afterwards. Important test setups like the test environment and different test configurations that are defined for the series of experiments are described, whereby these test configurations form the scaling scenarios of interest.

The subsequent chapter, namely Chapter 5, presents the results of the conducted measurements including graphical depictions, and also outlines the analysis and evaluation of the

measured values. Furthermore, potential causes of the observations and conclusions that can be drawn from the results are part of this chapter.

As the final chapter of this thesis, Chapter 6 summarizes the findings of this thesis and frames a general conclusion. Additionally, it provides an outlook regarding future work on the topic.

2. Related work

This chapter examines related work on distributed computing in constrained environments in order to give an overview of the current state of the art and classify the approach used within this thesis in the context of existing research efforts. This is done by reviewing previously published literature dealing with the same topic or having a similar objective and subsequently identifying the differences of the introduced approaches compared to the work accomplished in the course of this thesis. As mentioned in Chapter 1, the need for distributed computing mainly arises from the limited processing power, memory capacity and energy supply of constrained devices. In general, there are three main approaches to distributed computing that can be applied to constrained environments. They can be classified into cloud computing, fog computing and peer-to-peer networking resulting in in-network computing as stated in [PMN+18].

Cloud computing Various papers propose the integration of the cloud computing paradigm in the Internet of Things, see [ZLH+13], [AKAH14] or [BdDPP14]. As described by the authors of [PMN+18], cloud computing allows for complex computations to be processed by centralized remote servers, thereby shifting the resource-intensive computational burden from constrained nodes to powerful remote hubs via the internet. Constrained nodes are thus able to dispense tasks, which they cannot efficiently process on-site, to more powerful nodes that are provided by the cloud on demand. The authors of [BdDPP16] also suggest the usage of cloud systems within the IoT, thereby calling this new computing paradigm the *CloudIoT*. Their principal reason for integrating cloud-based models into the Internet of Things is the fact that cloud computing offers "virtually unlimited capabilities in terms of storage and processing power"[BdDPP16], which facilitates the compensation of the limited computational and memory capacities of constrained devices through high-performance cloud servers. Furthermore, this paper identifies even more complementary properties of the IoT and the cloud, while stating that the combination of both technologies can yield several benefits. For instance, following [BdDPP16], the IoT suffers from a lack of interoperability, reliability, efficiency and availability, whereas these properties can be provided by cloud systems and by building cloud-based distributed computing systems within the IoT, such deficiencies can be remedied. According to the previously mentioned paper, the new computing paradigm, that incorporates cloud computing in IoT scenarios, fosters a series of large-scale applications and provides novel smart services, such as wide-area environmental monitoring. On the downside, the *CloudIoT* approach involves several considerable challenges, that are also identified by Botta et al. in [BdDPP16]. They comprise security and privacy concerns, which results from potentially untrusted cloud providers, as well as the challenging heterogeneity of IoT devices and technologies. But the most notable issues of such distributed cloud computing systems are the amount of latency they impose on service delivery, and the increase of network traffic, which can collectively lead to long delays in

service delivery. Since many IoT applications must satisfy strong performance requirements and demand real-time services, this might represent an undesirable trade-off.

Fog computing In order to reduce the latency in cloud-based IoT systems, several papers like [AH14], [SM16] or [DBH15] suggest an extension of such systems through fog computing, which constitutes another distributed computing paradigm. As stated by the authors of [MKB18], the combination of cloud-based systems with fog computing enables the computation facilities to be located closer to the IoT devices, in contrast to approaches exclusively using the cloud computing paradigm. This can be traced back to the fact that computation-intensive tasks are mainly shifted to the edges of a network in fog computing-based architectures, such as gateway routers or switches as opposed to remote cloud servers. In this way, response latency for service requests within real-time applications is improved, since the amount of network hops is reduced and only the most complex computations are executed within the cloud. Even though conventional networking devices are not as powerful as cloud servers, such edge nodes still exhibit significantly more resources and therefore higher processing capabilities than constrained nodes, as required for the computation of complex tasks. For this reason, they can be used to realize offloading techniques within a network in order to disburden constrained nodes. Furthermore, the authors argue that fog computing can create an intermediate layer between constrained network nodes and cloud servers, thereby bridging the the gap between low-level IoT devices and high-level cloud computing nodes. By using the cloud computing paradigm leveraging a fog computing-backed IoT infrastructure, "large geographical distributions of Cloud-based services"[MKB18] can be built. However, fog computing presents a rather centralized approach as well, since "data still needs to reach a centralized collection point"[PMN+18] and thus latency is improved, indeed, but not minimized. Additionally, the authors of [MKB18] identified further issues of fog computing, for instance structural challenges of edge devices resulting from the vital coordination between the processing of computation-intensive tasks and the usual networking activities like routing or packet forwarding. Apart from this, security aspects have to be considered in fog computing infrastructures, since edge devices are susceptible to security attacks and by implementing special security-ensuring mechanisms, the desired Quality of Service (QoS) in real-time applications might not be fully met.

Peer-to-peer networking Although cloud and fog computing systems integrated into the IoT present considerable approaches, they exhibit key technical challenges and suffer from several deficiencies, most notably in terms of service latency, and therefore impose undesirable trade-offs. At the same time, they disregard the collective computation capability constrained devices can potentially provide by working together within a network in order to jointly complete computation-intensive tasks, as proposed in [PMN+18], [WLMQ16] or [APP13]. Furthermore, as the number of constrained devices rapidly grows and thus a denser deployment of such devices is achieved, a cooperation of nodes operating in constrained networks offers a reasonable potential. Following this peer-to-peer networking approach, resource-intensive tasks can be distributed among the constrained nodes within a network and so data is only processed on site, instead of shifting the computations to centralized nodes like it is the case with cloud and fog computing. Therefore, peer-to-peer networking in constrained environments represents a more decentralized approach. The au-

thors of [PMN+18] also state that the distribution of energy consumption is more even and more efficient within such networks in contrast to fog computing infrastructures, thereby saving resources, e.g. in case the constrained nodes are battery-powered. Concerning fog computing, especially the "nodes that are directly connected to the gateway will be necessarily involved in the relaying of all the messages directed to the gateway itself" [PMN+18], which might lead to the so-called energy hole effect. Additionally, the proximity of the cooperating nodes performing in-network computations can reduce the latency in contrast to cloud or fog computing, since the number of hops and the network traffic is minimized. As a result, peer-to-peer networking approaches are potentially capable of realizing applications with real-time demands, but the actual performance also depends on the scalability of the specific implementation of this approach, so that the network is also capable of handling high workloads.

In this sense, the ZeroMQ Message Transport Protocol represents a way to enable distributed in-network computing in constrained environments by providing a transport layer protocol targeted at peer-to-peer communication. By using a port of this protocol, which is tailored to the RIOT operating system, this computing paradigm can be deployed in constrained environments. In order to be able to judge whether ZMTP is suitable and efficiently usable for this specific purpose or not, the scaling behavior of the protocol is investigated and consequently evaluated within this thesis to assess the scalability of the ZMTP port in terms of distributed applications.

3. Background information on constrained environments

This chapter covers background information on constrained environments with special regard to the technologies used within this thesis. Section 3.1 focuses on constrained devices by first classifying them into different categories and then introducing protocol standards for constrained-node networks. In Section 3.2, the RIOT operating system is detailed, since the ZMTP port that is used in the course of this thesis is tailored to this operating system for IoT devices. Finally, Section 3.3 presents the ZeroMQ Message Transport Protocol itself, especially in terms of the design of the protocol in general and the ZMTP port for RIOT.

3.1. Constrained devices

The following sections provide an introduction to constrained environments, whereby Section 3.1.1 proposes a way to categorize constrained devices with respect to their resource characteristics, while the IoT protocol stack as used in constrained environments is depicted in Section 3.1.2. Section 3.1.3 describes a few fields of application for constrained devices, whereas Section 3.1.4 outlines challenges that need to be addressed in constrained environments.

3.1.1. Definition and classification

The Internet of Things comprises a large number of heterogeneous devices, ranging from sensors and actuators to various micro-controllers with different architectures. In order to be able to make a distinction between all these different coexisting devices, they are classified into two categories, depending on their characteristics concerning the level of resource scarcity, as proposed by the authors of [HBPT16], whereby these resource constraints refer to computational power, energy and memory capacity. On the one hand, there are *high-end IoT devices*, such as smartphones or Raspberry Pis, which are constrained to a certain extent, but still able to run common operating systems like Linux, since they possess sufficient hardware resources. On the other hand, more constrained devices called *low-end embedded devices* exist, e.g. Arduino boards or STM32 Nucleo-64 development boards. According to [BGH+18], low-end IoT devices are based on a single-core micro-controller, which comprises a CPU and possesses Random Access Memory (RAM) and Read-Only Memory (ROM) with a size of only a few kilobytes.

The category of low-end devices can be divided up even further to get a finer graduation and therefore an even more precise classification. In RFC 7228 [BEK14], the Internet Engineering

Task Force (IETF) proposes a classification standard for constrained devices, which includes three subcategories that differ in terms of memory capabilities. Each subcategory identifies a specific group of devices that all have more or less the same storage capacity, concerning the ROM/Flash and RAM. In this way, devices with similar memory characteristics can be aggregated into one of three classes, namely class 0, class 1 or class 2, as can be seen in Table 3.1. Within this table, the memory capacities of each class are defined with respect to the data size and the code size they offer. Following [SPKS12], the limited storage capabilities are caused by and depend on the underlying micro-controller architecture used for the embedded devices, which is either an 8-bit, 16-bit or 32-bit architecture.

As described in [HBPT16] and [BEK14], class 0 constrained devices are mainly sensor-like motes and impose severe memory limitations, so usually they offer only a small data set and are not capable of running proper operating systems. Class 1 defines devices that are able to implement light-weight IoT-tailored network stacks and run reprogrammable and hardware-independent applications on top, which facilitates the integration of such devices into an IP network. Therefore, they have to provide a reasonable set of software primitives, that are generally implemented by an operating system. Such operating systems need to be optimized for constrained environments by meeting specific demands, in order to be suitable and applicable in such scenarios. The RIOT operating system represents such an operating system and is described in Section 3.2. Lastly, class 2 devices are not as restricted and thus might possibly support conventional networking protocols, but using protocols that are aligned to IoT requirements can result in advantages concerning the available memory space for applications.

Name	**Data size (e.g. RAM)**	**Code size (e.g. Flash)**
Class 0	$<<$ 10 kB	$<<$ 100 kB
Class 1	$\sim$ 10 kB	$\sim$ 100 kB
Class 2	$\sim$ 50 kB	$\sim$ 250 kB

Table 3.1.: Classification of constrained IoT devices as defined in RFC 7228 [BEK14]

By interconnecting low-end IoT devices, highly dynamic constrained-node networks can be created, such as Low-Power and Lossy Networks (LLNs) or Low-Power Wireless Personal Area Networks (LoWPANs), within which the nodes typically communicate via low-power radio standards like IEEE 802.15.4 instead of Ethernet or Wi-Fi, as stated in [BEK14]. This imposes even more restrictions on the constrained environments in terms of bandwidth and reliability.

The focus of this thesis is primarily on low-end constrained devices, so the wide-ranging field of heterogeneous IoT devices is narrowed down to these very resource constrained devices, because “IoT devices will get smaller, cheaper, and more energy-efficient, instead of providing significantly more memory or CPU power. Therefore, in the foreseeable future, low-end IoT devices with a few kilobytes of memory, such as Class 1 and Class 2 devices, are likely to remain predominant in the IoT”[HBPT16]. Furthermore, high-end devices do not face such a strict resource scarcity and thus do not desperately need new alternative operating systems, while they are even capable of implementing existing networking standards without any modifications, as mentioned in [BEK14].

3.1.2. The IoT protocol stack

Since the constraints and specific requirements of IoT devices imposes restrictions on the applicability of traditional network protocol standards in the context of the Internet of Things, the need for a new protocol stack emerged. Therefore, the IETF developed an IoT-tailored protocol suite that encompasses adapted and standardized communication protocols, as presented in [SYY+13].

Figure 3.1 gives an overview of the cleanly layered IETF IoT protocol stack, consisting of six different layers, namely *Application*, *Transport*, *Network*, *Adaptation*, *Link* and *Physical Layer*, each of which is built on top of one another. Compared to the TCP/IP stack, that comprises four layers in total, the IoT protocol stack includes similar layers, but also implements some novel protocol standards, that are not present in the traditional stack, as explained below. In contrast to the corresponding standards defined by the TCP/IP stack, these new protocols are more suitable for the resource-constrained devices operating in constrained environments. In general, the IETF protocol stack for the IoT "tries to be as compatible as possible to the traditional TCP/IP protocol suite used in the conventional Internet"[Len16], but at the same time, a lot of effort was put into enabling internet connectivity, while providing high energy efficiency and reliability for the resource-constrained devices, as mentioned in [PAV+13]. The individual protocols of each layer of the IoT protocol suite, as shown in Figure 3.1, are described below, starting with the lower layers.

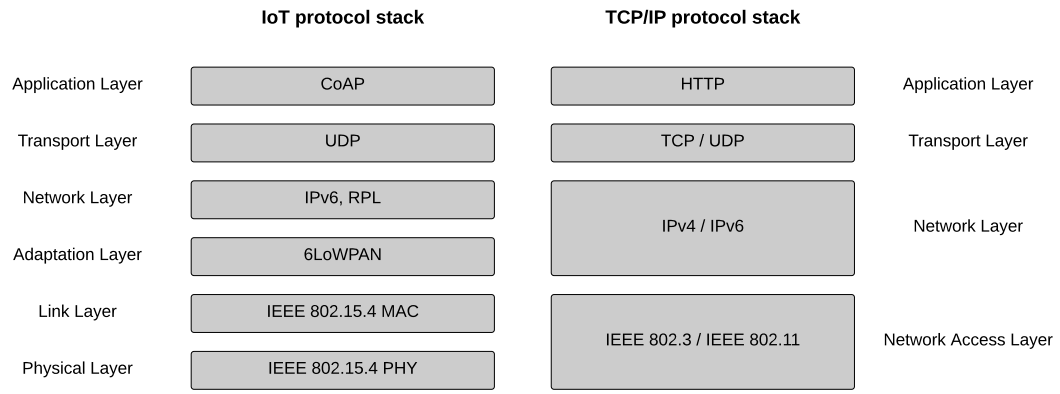


Figure 3.1.: Comparison between the IoT and TCP/IP protocol stack derived from [LB16]

IEEE 802.15.4 According to [SYY+13], IEEE 802.15.4 is a wireless communication standard based on radio transmission and is targeted at low-power and low-data-rate wireless personal area networks (WPANs). It specifies the physical layer, as well as the link layer of the IoT protocol stack. The data rate of IEEE 802.15.4, which operates within the 2.4 GHz frequency band, is limited to a maximum of 250 kbit/s. On the contrary, IEEE 802.3 Ethernet or IEEE 802.11 Wi-Fi are able to transmit several Mbit or Gbit per second [BHW+12], so IEEE 802.15.4 exhibits a clear trade-off between energy efficiency and bandwidth. Furthermore, RFC 4944 [MKHC07] states that IEEE 802.15.4 only has a maximum packet size of 127 bytes, while IPv6 provides a Maximum Transmission Unit (MTU) size of 1,280 bytes

for IPv6 packets transmitted over IEEE 802.15.4, causing the need for a fragmentation and reassembly adaptation layer directly above the link layer, which is realized through 6LoWPAN.

6LoWPAN As specified in RFC 4919 [KMS07] and illustrated in Figure 3.1, IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) is a network protocol that forms an intermediate adaptation layer in between the IP and the data link layer in terms of packet fragmentation and header compression. Despite the frame size constraints of IEEE 802.15.4, that limits the packet size to only 127 bytes, 6LoWPAN defines an encapsulation mechanism, which is described in RFC 4944 [MKHC07] and in [SYY+13]. It is able to fragment data packets for lower layers and to reassemble them for upper layers in order to provide a minimum MTU of 1,280 bytes as required by IPv6. In addition, it optimizes IPv6 transport over IEEE 802.15.4 networks by reducing header overheads by means of header compression.These overheads are usually caused by the MAC header, the link-layer security header, the IPv6 header and the UDP header, which in turn are responsible for a payload size of only 81 bytes for IP packets in the worst case when no header compression is used.

IPv6 Sticking with IPv6 as network layer protocol for the IoT protocol suite provides two main advantages, as argued by [Len16]. On the one hand, due to its 128 bit long addresses, IPv6 provides a large address space of 2^{128} different IP addresses, which represents an important benefit for the usage in the context of IoT, since the number of devices that are being deployed into the Internet of Things is assumed to steadily grow immensely. Therefore, IPv6 is able to withstand the vast amount of nodes by providing enough IP addresses without getting depleted — in contrast to IPv4 — so each of these nodes can be identified and addressed uniquely. On the other hand, the usage of IPv6 ensures that the new protocol stack is compatible with the TCP/IP stack and thus facilitates interconnectivity.

RPL A second component of the network layer is the IPv6 Routing Protocol for Low-power and Lossy networks (RPL) as defined in RFC 6550 [WTB+12]. This protocol is based on the distance-vector algorithm and therefore the network is created as a Destination-Oriented Directed Acyclic Graph (DODAG) by gathering and distributing information about link costs and node attributes, among others. Following [Len16], a DODAG is characterized by directed edges without cycles including one or more central control points and facilitates quick multi-hop routing within LLNs. According to [PAV+13] and [SYY+13], this is achieved by its capability of dynamically adapting the network topology and selecting paths based on different link and node metrics by means of the so-called Objective Function.

UDP Referring to [PAV+13] and [BHW+12], the User Datagram Protocol (UPD) constitutes the main transport layer protocol of the IoT stack, since TCP requires more hardware resources and is more complex than UDP. Thus, TCP is considered to be less suitable as transport over low-power and lossy networks. In contrast to TCP, UDP is connectionless, but, in turn, it able to transmit messages between two endpoints with a significantly lower overhead than TCP. The downside of UPD is that there is no guarantee for proper packet delivery, which implies a trade-off between reliability and energy efficiency.

CoAP At the application layer, the IoT protocol suite defines the Constrained Application Protocol (CoAP) as opposed to the Hypertext Transfer Protocol (HTTP), which represents the corresponding protocol of the TCP/IP stack. CoAP is a web transfer protocol optimized for constrained nodes, as specified in RFC 7252 [SHB14]. It enables seamless interoperability between existing internet applications and the IoT stack by providing a subset of REpresentational State Transfer (REST) operations, such as *GET*, *PUT*, *POST* or *DELETE*, as used by HTTP [SYY+13]. Additionally, it was developed with regard to the restrictions imposed by constrained nodes and therefore it is aimed at producing low overhead and guaranteeing less complexity than HTTP. "Unlike HTTP, CoAP is an asynchronous request/response protocol over a datagram oriented transport such as UDP"[PAV+13]. Internally, CoAP is split into two different layers, namely a message layer, which is located directly above the transport layer and responsible for reliability, as well as a request/response layer, located between the message layer and the application layer, supporting the RESTful capabilities, see [PAV+13].

3.1.3. Fields of application

Constrained-node networks open up a wide range of new opportunities in various application domains. They paved the way for building smart environments in terms of home automation systems that can be controlled remotely in order to save energy, intelligent healthcare applications such as ubiquitous health monitoring devices, smart factories in which industrial machines are being monitored, or traffic management systems enabling efficient transportation, as described in [YAH+17] and [BEK14]. Following the authors of [GBMP13], there are also several application possibilities for personal usage, especially regarding consumer electronics that provide certain services to individuals on a small scale, such as smart watches. On the other hand, smart grids represent an example for large-scale applications using constrained environments. According to [PS17], cloud computing systems can also be integrated into constrained networks, whereby fog computing might play a major role in bridging the gap between low-level constrained devices and high-level cloud computing nodes, as mentioned in Chapter 2. So in general, the fields of application are very diverse and application possibilities are hardly limited.

3.1.4. Challenges

Even though the usage of constrained environments provides seemingly endless opportunities, there is a number of challenges they have to face, that still remain unsolved to a certain extent. First of all, sensing devices generate large amounts of data, which need to be stored, managed and processed, so data mining and analysis mechanisms encompassing compressive sensing or data fusion are required, as proposed by [SYY+13] and [PS17]. Another challenge is guaranteeing interoperability among the enormous range of devices with heterogeneous capabilities and constraints, as well as the seamless integration thereof into the existing internet, which consists of less constrained nodes that implement traditional protocol standards, as argued in [YAH+17]. Although the IETF tried to establish a resource-aware, standardized protocol suite for the Internet of Things, which should be compliant with the traditional TCP/IP stack, the coexistence of multiple communication technologies and standards men-

tioned in [SYY+13] makes it hard to interconnect all those distinct devices over the Internet. Besides these issues, the authors of [LL15] state that constrained networks also have to deal with security concerns. Therefore, proven security mechanisms have to be adapted with respect to the particular demands of constrained devices in order to provide confidentiality and authentication, and to protect from different kinds of possible threats [LB16]. In conclusion, some of these challenges are not fully overcome yet and therefore still need to be solved.

3.2. The RIOT operating system

Since RIOT represents the operating system (OS) that is used in the course of this thesis, only this OS is presented in detail. Section 3.2.1 provides an overview and introduces the key features of RIOT, while its internal structure is outlined in Section 3.2.2. RIOT's default communication stack, which is called the GNRC network stack, is covered in Section 3.2.3. FInally, the TCP implementation for this IP stack, referred to as GNRC TCP, is thoroughly described in Section 3.2.4.

3.2.1. Overview and features

In recent years, the need for operating systems which can be deployed on embedded and thus constrained devices has increased as a result of the growing complexity of software used on such hardware, as claimed by [LKH+18]. Therefore, operating systems that are being used in the field of IoT have to meet specific requirements, unlike common operating systems for internet hosts. RIOT OS is an open source operating system that was developed from scratch with regard to meeting these particular demands that involve real-time behavior, high energy efficiency, a low memory footprint and modularity[2]. According to [BGH+18], RIOT focuses on low-end IoT devices and supports a large number of diverse sensors, actuators and different micro-controller architectures, either 8-bit, 16-bit or 32-bit, which in turn ensures interconnectivity and interoperability among such heterogeneous devices. Its micro-kernel architecture enables multi-threading based on a preemptive, tick-less, fixed priority scheduler and provides support for efficient Inter-Process Communication (IPC). The kernel's powerful IPC Application Programming Interface (API) can be used for blocking and non-blocking, as well as synchronous and asynchronous messaging, as described in [LKH+18]. In order to guarantee reliability and real-time behavior, RIOT only permits static memory allocation in the kernel, but dynamic memory allocation is possible for the use in applications, as mentioned in [BHG+13].

The slogan "the friendly OS for the Internet of Things"[3] refers to RIOT as being developer, resource and IoT friendly. Developer friendliness is provided in terms of facilitating standard C or C++ programming, as well as supplying numerous system libraries with hardware-independent accessibility, various data structures and a native port. This native port makes it possible to compile RIOT applications for Linux or Mac OS and then run them as a UNIX process without deploying them on real embedded hardware[2]. Furthermore, [BGH+18]

[2] https://github.com/RIOT-OS/RIOT/wiki/Introduction
[3] https://www.riot-os.org/

argues that RIOT's modular architecture simplifies the integration of new components or stacks and fosters arbitrary configurations for a wide range of use cases, as its modules can be randomly combined. Due to its unified APIs, the cross-platform code is highly portable and code duplication is minimized. Referring to [BHG+13], the low resource consumption points out to the fact that RIOT itself only accounts for a minimum of 1.5 kB of RAM for the data segment and 5 kB of ROM for the code segment, while it allows for maximum energy efficiency at the same time due to built-in energy saving mechanisms. The IoT friendliness is proven by its support for all common IoT standards, such as IPv6, 6LoWPAN, RPL, and so on[3]. Additionally, RIOT offers multiple IP protocol stacks such as lwIP, emb6 or the GNRC network stack (see Section 3.2.3). The source code, which is distributed under the LGPLv2.1 license, is freely available on GitHub[4] and continuously developed and maintained by a worldwide grass-roots community.

Besides RIOT, Contiki or TinyOS represent alternative operating systems for the Internet of Things. Table 3.2 shows the differences between these three open source IoT operating systems and also compares them to Linux. According to this table, RIOT supports most features, in particular, its real-time capabilities are remarkable, while its memory footprint remains rather low. But since Table 3.2 only represents a general feature comparison, several studies have been conducted in order to achieve a more in-depth comparison among such operating systems, e.g. by benchmarking different IoT networking architectures and evaluating their communication performance, including lwIP, emb6 and RIOT's default network stack GNRC, see [LKH+18], or by thoroughly analyzing their technical and non-technical properties, as described in [HBPT16].

OS	Min RAM	Min ROM	C Support	C++ Support	Multithreading	MCU w/o MMU	Modularity	Real-Time
Contiki	< 2kB	< 30kB	partial	no	partial	yes	partial	partial
TinyOS	< 1kB	< 4kB	no	no	partial	yes	no	no
Linux	∼ 1MB	∼ 1MB	yes	yes	yes	no	partial	partial
RIOT OS	∼ 1.5kB	∼ 5kB	yes	yes	yes	yes	yes	yes

Table 3.2.: Comparison between Contiki, TinyOS, Linux and RIOT OS based on [BHG+13]

3.2.2. Structure

According to [BGH+18] and the RIOT documentation[5], which together form the basis of this section, RIOT is composed of eight building blocks. Figure 3.2 gives an overview of RIOT's software structure, that also resembles the code base structure of the RIOT repository available on GitHub.

The key component of this structure is called `core`, since this represents the subdirectory where all the kernel-related code resides. It mainly comprises the scheduler, the IPC and the thread management. The `cpu` and `boards` elements include the platform-specific code such as peripheral configuration and pin-mapping, or CPU-specific power management and interrupt handling implementations. The `periph` component contains all the code for RIOT's peripheral driver interface, which enables unified access to micro-controller peripherals like Serial Peripheral Interface (SPI), General Purpose Input/Output (GPIO) or Universal Asynchronous Receiver Transmitter (UART). Furthermore, the `drivers` subdirectory implements

[4]https://github.com/RIOT-OS/RIOT
[5]https://www.riot-os.org/api/

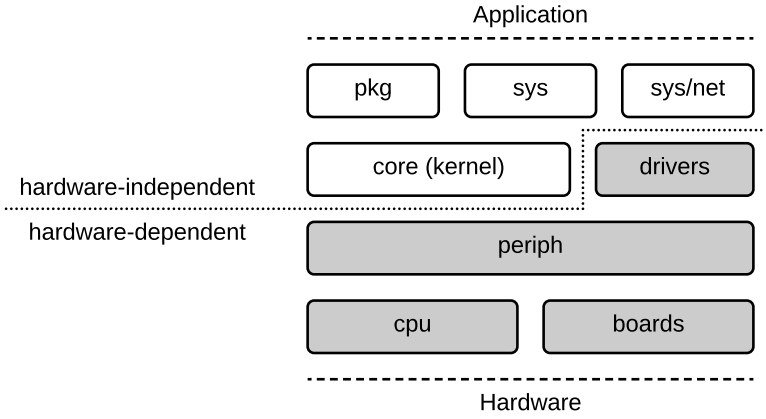


Figure 3.2.: Overview of RIOT's structural elements premised on [BGH+18]

external device drivers for sensors, actuators or network interfaces, independent of the underlying board and CPU. System libraries that do not belong to the kernel or the hardware abstraction can be found in the `sys` building block, e.g. data structures, Posix implementations and the RIOT shell. It also includes the `sys/net` subdirectory, which in turn holds all the code used for networking, such as the IPv6 or the 6LoWPAN library. The last component is called `pkg` and incorporates third-party libraries that can be seamlessly and easily integrated into RIOT applications at build-time by fetching the code from a remote source and applying patches if necessary. In addition to these elements, the RIOT code base also encompasses other subdirectories like `examples`, `tests`, `makefiles`, `doc`, as well as `dist`, which provides different tools such as a serial terminal application and various scripts for flashing, code checking, etc.

3.2.3. The GNRC network stack

While RIOT OS offers various communication stacks, the so-called Generic (GNRC) network stack is the default one, which represents a modular and configurable IP protocol stack[6]. It provides full-featured networking by implementing all the functionalities that are required in the field of IoT networking, as detailed in [Len16] and [LKH+18]. These features comprise high modularity, a network interface API with multi-interface support, well-abstracted APIs with clean protocol separation and the buffering of multiple network packets.

Figure 3.3 displays GNRC's networking architecture with all its components. The cleanly layered structure allows for high flexibility concerning the interchangeability and customizable combination of the different building blocks. In this way, an application only has to include the modules that are actually needed and therefore the degree of freedom during application development is enhanced. Besides its full-fledged IPv6 implementation (`gnrc_ipv6`) with 6LoWPAN extensions (`gnrc_sixlowpan`), GNRC also offers UDP (`gnrc_udp`) and TCP (`gnrc_tcp`) support. The different protocol layers can communicate via GNRC's communication interface `netapi`, which makes use of the kernel's IPC API in order to enable

[6] http://riot-os.org/api/group__net__gnrc.html

interaction between the various threads of the modules, e.g. when a packet is passed up or down the stack. The gray boxes in Figure 3.3 indicate that these GNRC modules are running in their own thread. Southbound, GNRC provides unified access to network interfaces through the generic network device driver API `netdev`, which is achieved by abstracting all available device drivers. Northbound, the GNRC network stack can be accessed from the application layer by the user programming interface `sock`. To sum it up, the GNRC network stack supports all standard protocol specifications of the IoT network stack mentioned in Section 3.1.2, either by directly implementing them, as it is the case with GNRC IPv6, 6LoWPAN, UDP and RPL, or by enabling the integration of any CoAP library to the networking subsystem using third-party packages such as libcoap and microcoap.

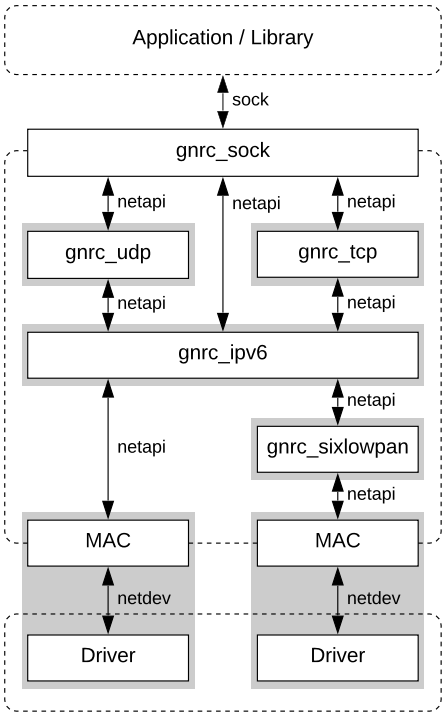


Figure 3.3.: The GNRC networking architecture based on [LKH+18]

3.2.4. GNRC TCP

The following paragraphs about RIOT's implementation of the Transmission Control Protocol (TCP) for the GNRC network stack are based upon [Bru16]. Even though both UDP and TCP are core transport protocols on the Internet, the initial version of the GNRC network stack only provided UDP support. A subsequent additional TCP implementation for RIOT's default IP stack, named GNRC TCP, expanded RIOT's applicability in the context of IoT, since TCP is the most commonly used transport protocol on the internet and thus many protocols rely on TCP. By supplying TCP capabilities within the default network stack, such protocols can be ported to RIOT, thereby easing the integration of devices running RIOT OS into existing networks that are based on TCP communication.

As specified in RFC 793, TCP is a "connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols"[Pos81]. RIOT's TCP implementation for the GNRC network stack is tailored to the particular requirements and constraints of IoT scenarios. Its design goals comprise high memory efficiency, the exclusive use of static memory allocation, optional TCP support, a feature set optimized for the Internet of Things and the compatibility with other TCP implementations. Since a low memory footprint constitutes the most important goal, trade-offs could not be prevented. A trade-off worth mentioning here is the limitation of network throughput to a single TCP packet at a time if the size of the packet equals the predefined Maximum Segment Size (MSS), as outlined below. The use of static memory allocation only ensures deterministic and real-time behavior during runtime, but also imposes some restrictions on the implementation, which results in the impossibility of GNRC TCP being an exact reimplementation of the BSD socket API. Due to the high modularity of the GNRC network stack, it was a comprehensible goal to integrate the TCP capabilities as an optional module of the stack (see Figure 3.3). The general feature set of GNRC TCP does not reflect a full-fledged TCP implementation because all the features were analyzed and evaluated in terms of suitability and applicability in IoT scenarios and therefore TCP extensions such as Selective Acknowledgment options (SACK) or congestion control algorithms were omitted, mainly on account of memory requirement and complexity reduction. Even though GNRC TCP is not fully featured, it is still compatible with other TCP implementation, even with full-fledged ones, in order to guarantee interconnectivity with different TCP implementations. "TCP operations rely on a few basic concepts, namely basic data transfer, reliability, flow control, multiplexing and connection handling"[Bru16]. GNRC TCP realizes these concepts by implementing the following principles.

Transmission Control Block (TCB) The TCB is the most important data structure of GNRC TCP because it holds all the information about a certain connection including the state of the connection, the address family, the local and peer IP addresses and port numbers, fields for the sequence and acknowledge number mechanisms, the MSS, retransmission timer definitions and other information used for connection handling and data exchange between two peers. Each peer has its own TCB for every connection, both of them are being filled with the relevant information during the connection establishment. Since the IP addresses and port numbers are stored inside the TCB, a pair of TCBs uniquely identifies a TCP connection — similar to a pair of BSD sockets — and multiplexing between applications is possible due to the binding of ports to processes.

Connection establishment and termination Before data can be exchanged between two peers, a connection must be established using a 3-Way-Handshake mechanism in order to make sure the connection initialization is carried out correctly. The procedure of a 3-Way-Handshake is demonstrated in Figure 3.4, including the sequence and acknowledgement number mechanism described in the next paragraph. Usually, one node issues a passive open, which involves listening for incoming requests on a certain port. Another node attempts to initiate a connection to the peer by performing an active open, specifying the peer's port number as destination port. This is done by sending a TCP packet with a SYN flag to the peer. If the peer accepts the connection, he sends back a segment containing a SYN and an ACK flag, which in turn has to be acknowledged by the initiating node before the

connection is marked as established. After the exchange of data, the connection must be terminated properly, thereby freeing the previously used resources. Both peers have to indicate that they are willing to close the connection by sending a FIN-flagged packet to their counterpart. Additionally, the reception of these packets has to be confirmed by the dispatch of an ACK-flagged packet by both connection partners.

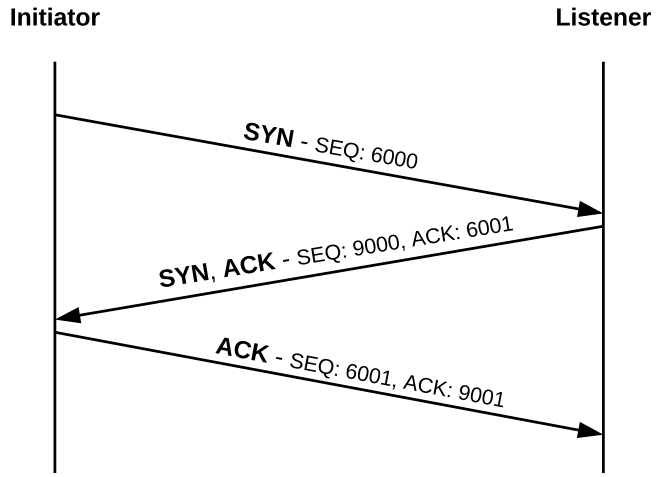


Figure 3.4.: Sequence diagram of a TCP 3-Way-Handshake for connection establishment

Data transfer and sequence numbers TCP packets carry data as byte streams, which can be transmitted in both directions between two connected peers. The size of the segments that can be sent within a single TCP segment is limited by the MSS, which is stored inside the TCB and communicated during the connection establishment. Each byte of data sent over a TCP connection is associated with a sequence number and every TCP packet is associated with the sequence number of its first payload octet within the segment. The sequence number of the last payload byte within the packet is calculated by the packet's sequence number plus the payload size minus one. Therefore, every received segment can be acknowledged by the peer through sending back an ACK-flagged packet that contains the corresponding acknowledgement number, either with or without additional data. Acknowledgements can be carried out in a cumulative fashion, so the acknowledgement number x indicates that all bytes with a sequence number lower than x have been received correctly. For instance, if a peer sends a packet with a sequence number of 6,000 that contains 50 bytes of data, the corresponding acknowledgement number for this segment would be 6,050, since it has to be bigger than the sequence number of the last data byte within the packet, which would be 6,049 in this case. Figure 3.4 also shows how the sequence and acknowledgement number mechanism works for the connection establishment, where no payload data is being sent by either peer. The consecutive sequence and acknowledgement number spaces are defined by multiple variables inside the TCB. For each segment that is being sent, a retransmission timer is started. If the timer expires and the corresponding packet has not been acknowledged yet, it is considered lost and thus resent. In combination, these mechanisms provide the required reliability of a TCP connection, including the detection of missing segments or duplicates.

Header format Figure 3.5 outlines the format of a TCP header. According to this Figure, all the TCP-specific control information is contained in the TCP header of a packet, including source and destination port information, the sequence and acknowledgement number, a data offset field, reserved bits for future use, control bits for flags like *SYN* or *ACK*, a field for the window size, the checksum used for error detection, an urgent pointer, an options field and padding, followed by the actual payload. The minimum header size is 20 bytes, which is the case when no options are set, while the maximum size is 60 bytes, since options can account for up to 40 bytes.

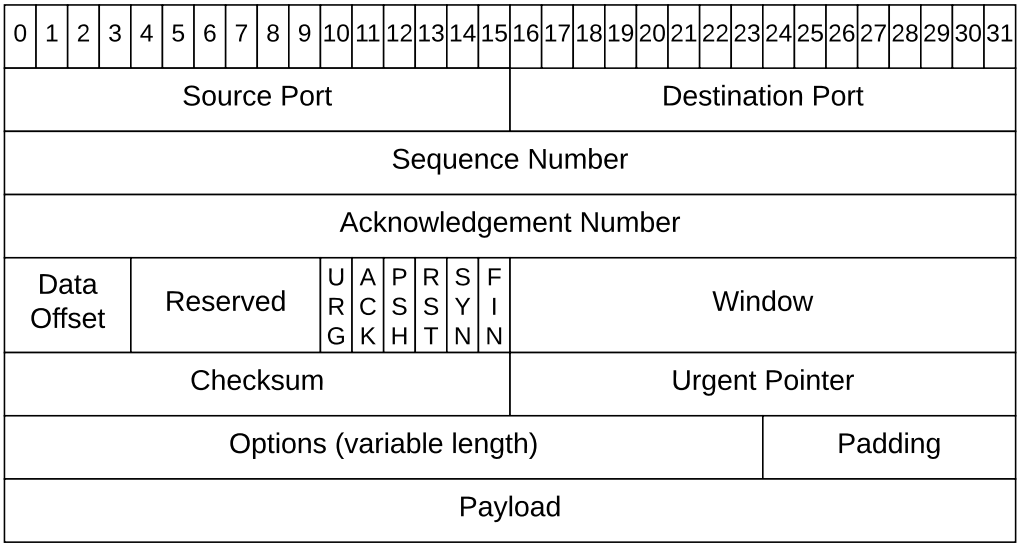


Figure 3.5.: TCP header format based upon [Bru16]

Window management The window management is responsible for the flow control of the connection in such a way that the amount of data, which can be sent to a peer before the sender has to wait for the acknowledgement of these packets, depends on the window size of the receiver. The size of the window is closely intertwined with the currently available receive buffer size and influenced by the MSS. It is communicated with every acknowledgement and indicates the amount of data that can be stored in the receive buffer at this point in time. A typical value for the MSS is 1,220 bytes, because it mirrors the default IPv6 Maximum Transmission Unit (MTU) of 1,280 bytes subtracted by the IPv6 header size of 40 bytes and the minimum TCP header size of 20 bytes. In common TCP implementations, the window size and therefore also the receive buffer size are multiples of the MSS in order to facilitate a higher throughput. GNRC TCP, on the contrary, implements a default receive buffer and window size of 1,220 bytes, which equals the configured MSS. This default configuration limits the TCP throughput to only a single packet at a time. The upside of this implementation is a further reduction of memory consumption, which is directly affected by the size of the receive buffer, and since memory savings represent the most important design goal, this is considered an acceptable trade-off of the GNRC TCP implementation.

TCP Finite State Machine Especially during the connection establishment and termination, several state changes take place. They are caused by either user calls on the application layer, the reception of a packet, or by timeouts. The Finite State Machine (FSM) defines the different states of a connection, namely `LISTEN`, `SYN-SENT`, `SYN-RECEIVED`, `ESTABLISHED`, `FIN-WAIT-1`, `FIN-WAIT-2`, `CLOSE-WAIT`, `CLOSING`, `LAST-ACK`, `TIME-WAIT` and `CLOSED`, as well as the specific events that trigger changes between them. Since the events that lead to a transition of states can occur within multiple threads simultaneously, a mutex is being used as synchronization mechanism. State changes in response to errors are not covered in this simplified FSM version.

In summary, the GNRC TCP implementation for RIOT's default stack provides basic TCP functionality with a reduced set of features that are optimized for the Internet of Things, but still compatible with other, potentially fully featured TCP implementations. Nevertheless, GNRC TCP exhibits some trade-offs concerning throughput, which can be attributed to the most important design goal of low memory consumption.

3.3. The ZeroMQ Message Transport Protocol

In the following sections, ZeroMQ and the associated ZeroMQ Message Transport Protocol are presented, including their key aspects and especially focusing the protocol design of ZMTP. Furthermore, the protocol port for RIOT OS that enables deploying distributed applications in constrained environments is introduced. While Section 3.3.1 focuses on ZeroMQ in general, Section 3.3.2 deals with ZMTP in particular, since it represents the protocol under investigation regarding scalability.

3.3.1. ZeroMQ

The terms ØMQ, 0MQ, or zmq can be used interchangeably and all refer to ZeroMQ, a messaging library that was designed for the use in distributed or parallel applications, as detailed by Pieter Hintjens in his book [Hin13a], which serves as the basis for this section. The main goals of ZeroMQ are ensuring simplicity and scalability, as well as guaranteeing high-performance messaging capabilities. The *zero* in ZeroMQ stands for several other design goals such as zero broker, zero latency, zero administration, zero cost and zero waste.

The ZeroMQ API provides sockets for a quick and efficient way of exchanging messages between nodes, whereby messages are being transported as blobs of data, with a size ranging from zero to several gigabytes. These sockets work together in pairs and can have different socket types, which form ZeroMQ's so-called *messaging patters*. They cannot be combined arbitrarily, because only two matching socket types can form a messaging pattern. Table 3.3 shows the valid socket type combinations.

There are four built-in core patterns, namely *request-reply*, *pub-sub*, *pipeline* and *exclusive pair*. All of the semantics of these patterns are specified within the ZeroMQ RFC project[7]. The request-reply pattern, defined in ZeroMQ RFC 28 [Hin13d], is based on the traditional

[7] https://rfc.zeromq.org/

client-server model occurring in service-oriented architectures and is used for task distribution or remote procedure calls. The servers provide certain services and the clients function as service requesters. REQ and REP are the socket types that implement this pattern and communicate within a synchronous request-reply dialog, while the counterparts belonging to this pattern, DEALER and ROUTER, enable a nonblocking asynchronous request-response dialog between nodes. As specified in ZeroMQ RFC 29 [Hin14b], the publish-subscribe pattern defines PUB and SUB as socket types and typically acts as a one-way data distribution pattern, where the publishers publish data and the subscribers are filtering all the incoming data, depending on the application's specific requirements. The third pattern, pipeline, is described in ZeroMQ RFC 30 [Hin13c] and requires the PUSH and PULL socket types. It constitutes a fan-out/fan-in pattern, which can be applied to realize parallel task distribution and collection, similar to the divide and conquer paradigm. The exclusive pair pattern is used to connect two threads within a process and the corresponding socket type for this pattern is called PAIR, as presented in ZeroMQ RFC 31 [Hin13b].

	REQ	REP	DEALER	ROUTER	PUB	SUB	PUSH	PULL	PAIR
REQ		X		X					
REP	X		X						
DEALER		X	X	X					
ROUTER	X		X	X					
PUB						X			
SUB					X				
PUSH								X	
PULL							X		
PAIR									X

Table 3.3.: Valid socket type combinations derived from [Hin14a]

Messages are routed and queued according to the specified messaging pattern, because the semantics of the sockets are defined by the individual patterns. With ZeroMQ, messages can be transferred via TCP, inproc, IPC or UDP in unicast and multicast modes. Depending on the transport, ZeroMQ nodes are mapped to threads, processes or peers within a network, whereby 1-to-N, N-to-1 or N-to-N connections can be established.

ZeroMQ's original core engine is called libzmq, which was written in C++, but many other alternative engines are available, in order to support a variety of different programming languages and therefore facilitating language-independent connection of code. All the source codes of the various engines can be found on GitHub[8] as free and open source software. libzmq is from iMatix and licenced under LGPLv3 open source[9].

3.3.2. ZMTP overview

This section is based upon RFC 23 [Hin14a] of the ZeroMQ RFC project, which contains the specification of the ZeroMQ Message Transport Protocol. ZMTP is a peer-to-peer transport layer protocol that is implemented by ZeroMQ's core engine libzmq and many of its

[8] https://github.com/zeromq/libzmq
[9] http://zeromq.org/docs:features

reimplementations, so it constitutes the main protocol used by ZeroMQ. ZMTP defines the messaging between two peers by encapsulating messages and specifying how to read and write frames on a connection. Additionally, it provides extensible security mechanisms with different safety levels.

Using ZMTP, nodes can communicate via TCP or IPC, depending on the application's use case. While libzmq is the core engine of ZeroMQ, ZMTP can also be seen as an alternative engine designed for the specific needs of constrained hardware, which is why it is also called "The protocol of things"[10]. It is implemented as a native C stack and enables two peers to communicate with each other just using ZMTP endpoints. The protocol intends to fix deficiencies that come along with TCP, for example the nonexistence of delimiters in a stream of octets or the impossibility of adding metadata to a frame. These problems are addressed and solved by a message framing mechanism, which defines that each ZMTP frame is composed of a flags field for metadata, a size field and a body.

Design of the protocol

In general, the protocol comprises three main stages: the connection establishment, the transfer of messages and the connection termination, similar to the specification of a TCP connection sequence. As can be seen in lines 4 and 5 of Listing 3.1, which shows the Augmented Backus-Naur Form (ABNF) grammar that defines the connection establishment phase of the protocol, a ZMTP connection consists of a greeting, a handshake, and traffic. While the *greeting* and the *handshake* belong to the connection establishment, *traffic* constitutes the actual communication between peers.

Connection establishment During the greeting, which is defined in lines 7-26 of Listing 3.1, the two peers communicate the protocol details by exchanging their signature (lines 10-11), as well as their version number (lines 13-15), and negotiating a security mechanism (lines 17-20). Additionally, a peer can specify whether it acts as a server (lines 22-23). The subsequent security handshake as described in lines 28-30 is performed by transmitting at least one command, e.g. the *READY* command as part of the NULL security mechanism. The different security mechanisms, namely NULL, PLAIN and CURVE, are defined as extension protocols and provide varying levels of authentication and confidentiality. The mechanism used also determines the commands that are being exchanged during the connection establishment handshake. The NULL security mechanism, which is the default one, neither offers authentication nor confidentiality, while the PLAIN mechanism, defined as ZeroMQ RFC 24 [Hin13f], implements a username-password authentication for each client that wants to connect to a server. In contrast, the CURVE mechanism provides full authentication and confidentiality in order to meet high security requirements, as specified in ZeroMQ RFC 25 [Hin13e].

Transfer of messages The actual communication between two nodes is performed in the data transfer phase, also called *traffic*, which is defined by the formal grammar presented in Listing 3.2. In this phase of the protocol, both peers are able to send and receive discrete messages (see lines 14-18) or more commands (see lines 4-12) intermixed. Apart from the

[10] http://zmtp.org/

Listing 3.1: ABNF grammar of a ZMTP connection establishment as defined in [Hin14a]

```
1  ;   The protocol consists of zero or more connections
2  zmtp = *connection
3
4  ;   A connection is a greeting, a handshake, and traffic
5  connection = greeting handshake traffic
6
7  ;   The greeting announces the protocol details
8  greeting = signature version mechanism as-server filler
9
10 signature = %xFF padding %x7F
11 padding = 8OCTET        ; Not significant
12
13 version = version-major version-minor
14 version-major = %x03
15 version-minor = %x00
16
17 ;   The mechanism is a null padded string
18 mechanism = 20mechanism-char
19 mechanism-char = "A"-"Z" | DIGIT
20      | "-" | "_" | "." | "+" | %x0
21
22 ;   Is the peer acting as server?
23 as-server = %x00 | %x01
24
25 ;   The filler extends the greeting to 64 octets
26 filler = 31%x00            ; 31 zero octets
27
28 ;   The handshake consists of at least one command
29 ;   The actual grammar depends on the security mechanism
30 handshake = 1*command
```

64 octet-sized greeting, all data exchange is carried out by reading and writing frames that contain either a command or a message. As presented in Figure 3.6, each frame is composed of three fields: one field for flags, one for the size and one for the body. Inside the flags field, which has a size of one byte, different flags can be defined by setting the corresponding bit from 0 to 1. The *MORE* flag (bit 0) signals that there are more frames to follow, the *LONG* flag (bit 1) indicates whether it is a short frame with a body ranging between 0 and 255 octets or a long frame with a body of maximum $2^{63} - 1$ octets. As a third option, a *COMMAND* flag (bit 2) can be set, specifying that the frame is carrying a command instead of a message, while the five remaining bits are reserved for future use and have to be set to zero.

The size field only contains the size of the data in bytes and either consists of one octet in case of a short frame or eight octets in case of a long frame. Lastly, the body holds the actual application data of a message or a printable command name and command data, within as many octets as specified by the size field. Regarding the NULL security mechanism, the only commands are *READY* and *ERROR*. The current version of the ZMTP protocol implementation, namely version 3.0, defines that upon dispatch of a message or command, each frame is fragmented into its three parts by the sender and then each field of the frame is sent as a separate TCP packet. On reception of the three packets containing the flags, the size and the body, the receiver reassembles the individual parts into a complete frame. This

Listing 3.2: ABNF grammar of ZMTP traffic as specified in [Hin14a]

```
1  ;   Traffic consists of commands and messages intermixed
2  traffic = *(command | message)
3
4  ;   A command is a single long or short frame
5  command = command-size command-body
6  command-size = %x04 short-size | %x06 long-size
7  short-size = OCTET          ; Body is 0 to 255 octets
8  long-size = 8OCTET          ; Body is 0 to 2^63-1 octets
9  command-body = command-name command-data
10 command-name = OCTET 1*255command-name-char
11 command-name-char = ALPHA
12 command-data = *OCTET
13
14 ;   A message is one or more frames
15 message = *message-more message-last
16 message-more = (%x01 short-size | %x03 long-size) message-body
17 message-last = (%x00 short-size | %x02 long-size) message-body
18 message-body = *OCTET
```

mechanism is used to signalize the receiver whether the frame size is contained in a single octet or encoded in eight octets, which can be determined through the LONG flag, and to indicate how much memory needs to be allocated for the upcoming data by the preceding size field.

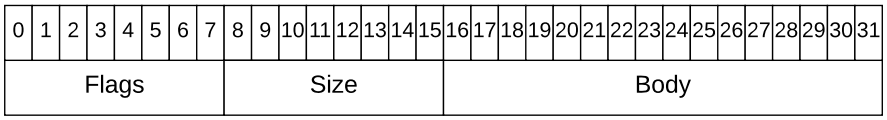


Figure 3.6.: The structure of a ZMTP short frame containing two octets of data

Connection termination Finally, the last stage of the protocol is the connection termination, where both peers close the connection, so no more data can be transferred. The connection can be closed anytime by each peer, but there is no formal grammar specification for this closing stage of the protocol.

Metadata The messaging patterns and corresponding socket types that were mentioned before can be specified by connection metadata entries as part of the security handshake. The ABNF grammar of the NULL security mechanism handshake is also defined by the ZMTP specification of the ZeroMQ RFC 23 [Hin14a] and presented in Listing 3.3. Metadata is communicated as key-value pairs (see lines 3-7), called *property* (line 4) in the listing above. The key is represented by the property name of the metadata (lines 5-6), e.g. *Socket-Type* in case of the messaging patterns, while the value (line 7) includes of one of the socket types such as REQ or REP. In this way, each peer can send its own socket type and also verify the socket type received from the other peer according to the valid combinations as noted above in Table 3.3.

Listing 3.3: ABNF grammar of the NULL security mechanism as specified in [Hin14a]

```
1  null = ready *message | error
2  ready = command-size %d5 "READY" metadata
3  metadata = *property
4  property = name value
5  name = OCTET 1*255name-char
6  name-char = ALPHA | DIGIT | "-" | "_" | "." | "+"
7  value = 4OCTET *OCTET        ; Size in network byte order
8  error = command-size %d5 "ERROR" error-reason
9  error-reason = OCTET 0*255VCHAR
```

ZMTP port for RIOT OS

Since the ZeroMQ Message Transport Protocol was not implemented for RIOT OS yet, the first step was to port this protocol to RIOT as preparatory work in order to use it for further work on distributed computing in the context of IoT. Thus, ZMTP was partially ported to RIOT as part of a practical university course[11] and then served as the basis for this thesis, more particularly, the use case programs for the experimental approach were built upon this protocol port, as described in the subsequent chapter.

The partial ZMTP port was defined as a new package called `libzmtp` and integrated into RIOT's third-party package system within the RIOT/pkg directory. The source and header files of the port can be found on GitHub[12], while the `libzmtp` package folder itself only contains two Makefiles, which specify the URL of the remote repository where the code should be fetched from, as well as the destination folder where the cloned files should be stored locally. The original protocol supports TCP and IPC, but since it was ported with regard to enabling distributed computing on constrained devices, the RIOT port only comes with TCP support, so it does not provide ZMTP's full functionality and is therefore only a partial port. It is built upon RIOT's GNRC TCP implementation for the GNRC network stack and the initial version of the protocol used for the port is version 3.0, which implements the NULL security mechanism by default, so the same applies to the port. Additionally, backwards interoperability is currently not supported, since version 3.0 of ZMTP does not provide this feature either, so the RIOT port will reject older versions of the protocol, just like the original protocol does. Since ZMTP is based on TCP, a prerequisite for porting this protocol to RIOT OS was a functioning TCP implementation on RIOT. As mentioned before, GNRC TCP is not an exact reimplementation of the BSD API, because RIOT does not support the usage of file descriptors, so in order to enable the usage of ZMTP on RIOT, all the function calls from the BSD socket API needed to be abstracted by adding GNRC TCP-specific code.

Regarding the file structure of the ZMTP port, Figure 3.7 depicts the dependencies between the different source and header files that are part of the port. The `zmtp_channel.c` file represents the main class and acts as the starting point of the protocol, since it implements all the user function calls, which in turn trigger functions of other classes such as

[11] Theresa Müller, Practical Course IoT at LMU Munich, summer term 2018, course website: http://www.mnm-team.org/teaching/Praktika/2018ss/iot/

[12] https://github.com/muellerthe/libzmtp

`zmtp_endpoint.c` or `zmtp_msg.c`. It provides functionality for establishing a connection between two nodes, sending and receiving messages, as well as closing a connection, which mirrors the sequence of a TCP connection. The two wrapper classes `zmtp_channel_wrapper.c` and `zmtp_tcp_endpoint_wrapper.c` were additionally added to the initial file structure of ZMTP in order to be able to abstract the BSD socket API calls of the original protocol.

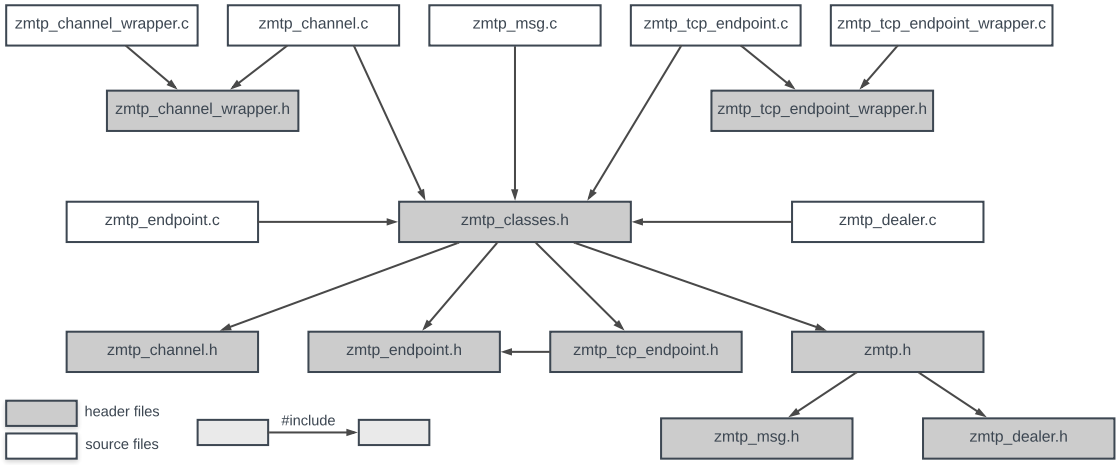


Figure 3.7.: Dependency graph of ZMTP source and header files

4. Experimental approach

The following sections focus on the experimental approach that is used to evaluate the scalability of ZMTP in constrained environments with respect to distributed computing applications. While Section 4.1 details the general methodology of this approach, Section 4.2 introduces the selected measurands for the series of experiments. Furthermore, Section 4.3 presents the two use case applications, which are developed specifically for the purpose of this thesis, whereas the test setups of the experiments are detailed in Section 4.4. The final section, namely Section 4.5, describes the applied test procedure of the experimental approach.

4.1. General methodology

In order to give an overview of the methodology used within this thesis, the conceptual design of the approach that enables the evaluation of ZMTP by conducting experiments and performing measurements at runtime are described in Section 4.1.1. Additionally, the problem space, which is associated with the runtime performance of distributed computing applications, is defined in Section 4.1.2.

4.1.1. Conceptual Design

Since the objective of this thesis is to assess the scalability of the ZeroMQ Message Transfer Protocol in constrained environments, a series of experiments based on two different use cases is conducted in order to analyze the scaling behavior of the protocol and draw conclusions from the runtime measurement results of the underlying applications.

Figure 4.1 shows the different steps of the experimental approach, which is used as methodology because it constitutes a scientific test and verification procedure that is considered best to investigate and evaluate ZMTP in terms of its scaling behavior. As a first step, the measurands for the runtime measurements which are executed during the experiments are determined. Afterwards, two test applications built upon ZMTP are developed, both of which represent disparate use cases focusing on distributed computing and therefore serve as the basis for the experimentation. Subsequently, different test configurations, simulating the scaling of the protocol by covering strong and weak scaling, are defined for both use case applications. These configurations are then tested within a particular test environment by means of a series of experiments, while performing measurements of the previously selected measurands during runtime and under realistic steady state conditions. Afterwards, the results of the measurements are analyzed, evaluated and on the basis of these results, resilient assertions about the scaling behavior of ZMTP are derived.

The first four aspects of the experimental approach as outlined in Figure 4.1 are detailed in the following sections, whereas the result analysis and evaluation, as well as the conclusions that can be drawn from the results, are covered by the next chapter, see Chapter 5.

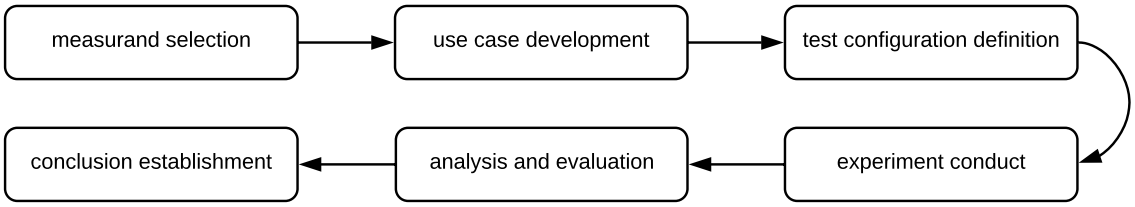


Figure 4.1.: Methodology of the experimental approach

In general, this methodology represents an attempt to determine whether the protocol shows the expected and desired scaling behavior, that is a speedup due to scaling, when increasing the number of computing units, or if a drop in performance can be identified at some point. If possible, more general predications are deduced from the resulting scaling conclusions in order to point out to implications for a broader context, but this is also part of the subsequent chapters.

4.1.2. Problem space definition

The problem space is defined by multiple linearly independent dimensions, each of which influences the runtime performance of a distributed computing system deployed in constrained environments to a certain extent. As can be seen in Table 4.1, there are various influencing factors, e.g. the application program itself or the associated total input size that defines the overall workload. Regarding the network infrastructure, the number of workers has a big effect on the runtime, because this determines among how many computing units the total input size is split and thus influences the workload per node. But also the underlying hardware of the nodes within a network can be responsible for fluctuations, since the hardware is characterized by the available resources in terms of computing power, memory capacity and power supply. Another factor is the spatial arrangement and therefore the actual distance between the nodes of a network, because this parameter can influence the interconnect as well as the radio interference between the nodes. The communication protocol that is being used also needs to be considered, since different standards can have different impacts on the runtime of a system, for instance using a protocol that is based on TCP, like it is the case with ZMTP, versus a protocol that uses UDP as transport layer protocol, thereby imposing less communication overhead as compared to TCP. The transmission technology, e.g. Ethernet, Wi-Fi or IEEE 802.15.4, which is defined by the used hardware, also plays an important role regarding the performance, because it directly interferes with the available data rate and thus causes more or less severe restrictions on bandwidth, transmission speed and throughput. Furthermore, the operating system running on the computation nodes might limit or enhance the performance of the system as well, with regard to constraints possibly imposed on protocol implementations by the underlying OS design. Of course, there are even more factors that influence the runtime of a distributed computing system, so Table 4.1 is not necessarily complete, but provides an excerpt that is useful for the experiment design.

Dimension	Altered during experiments	Value
Application	yes	Use case program
Number of workers	yes	varying
Total input size	yes	varying
Hardware	no	IoT-LAB M3 nodes
Spatial arrangement	no	adjacent nodes
Communication protocol	no	ZMTP
Transmission technology	no	IEEE 802.15.4
Operating system	no	RIOT OS

Table 4.1.: Partial problem space definition by means of dimensions

All of the aforementioned dimensions can be altered and the combination of a specific value for each dimension forms a certain configuration for a distributed computing system. Referring to the series of experiments that are conducted as part of this thesis, some of these dimensions are individually adjusted during the experiments in order to assess the effect of the alternations on the total runtime, while others remain constant. Table 4.1 represents all the dimensions and outlines whether they are adjusted in the course of the experimentation or not. Additionally, it shows the corresponding values that are chosen specifically for the purpose of the thesis. There are only three dimensions that are varied, namely the application, the number of nodes and the total input size, because variations of these parameters reflect strong and weak scaling by means of different use cases and the focus of the experiments is on the scalability of ZMTP. Narrowing the adjusted dimensions to only three limits the number of required experiments, and further measurements of other dimensions would go beyond the scope of this thesis.

While the application can be altered concerning different use case programs, values for the number of nodes and the total input size can range arbitrarily. The specific values and combinations between the three dimensions that are varied during the experiments are described in Section 4.4.2.

4.2. Measurands

Before the use case programs can be developed, the measurands for the experiments and associated runtime measurements, that form the basis for drawing resilient scaling conclusions, need to be determined. The two selected measured quantities, namely execution time (see Section 4.2.1) and power consumption (see Section 4.2.2) are presented in the following sections, which also cover how they are being measured during the experiments.

4.2.1. Execution time

The execution time marks the primary variable that is being measured during the experiments because the resulting values enable the identification of speedups or drops in performance due to scaling. The data acquisition is accomplished by function calls from RIOT's `xtimer` API, to be more specific, the `static uint32_t xtimer_now_usec(void)` function is

used (see Listing 4.1), which yields the current system time since the start in microseconds. In order to obtain the actual execution time of a code fragment, the difference between two measured values for the system time needs to be computed by subtracting the system time when starting the measurement from the system time when ending the measurement. Listing 4.1 demonstrates the implementation of this procedure, which is integrated into the use case programs as demonstrated in Listing 4.2 of Section 4.3. Within each program, several execution time measurements are performed, one for the overall execution time of the program and further ones for measuring the execution time of only some parts of the application.

Listing 4.1: Function calls needed for the execution time measurements

```
1  /* Get current system time for execution time at start of measurements */
2  const uint32_t time_start = xtimer_now_usec();
3
4  /* Code to be measured */
5
6  /* Get current system time for execution time at end of measurements */
7  const uint32_t time_end = xtimer_now_usec();
8
9  /* Compute overall execution time */
10 const uint32_t exec_time = time_end - time_start;
```

4.2.2. Power consumption

Besides the execution time measurements, the power consumption of each node as a secondary measurand is monitored during the experiments in order to determine differences between each node's energy footprint for varying workloads. As opposed to the execution time, measured values for the power consumption of a node are provided by a tool of the IoT-LAB testbed, which represents the test environment used for the experiments as described in Section 4.4.1. This tool allows for monitoring every node participating in an experiment by defining a monitoring profile, which enables measuring the current (in ampere), voltage (in volt) and power consumption (in watt) of each node separately during the experiment. For each monitoring profile, the configurable sampling period needs to be specified by setting values for conversion times (CT) and averaging mode (AV). The sample rate or periodic measure (PM) is computed by the formula $PM = CT \cdot AV \cdot 2$. Depending on the sampling period and especially on the number of averages, the resulting signals might be filtered through noise reduction, so the measurement accuracy can be influenced by these settings[13]. Regarding the monitoring profile used for the experiments, the conversion time value is set to 8,244 µs, while the averaging mode is set to 16, which leads to a slightly filtered signal. As a result, the power consumption is measured every 264 ms. The measured values per node obtained by the tool are stored in an .oml file, that can be plotted by means of a python script, which is also provided by the IoT-LAB testbed, generating a graph representing a node's power consumption.

[13] https://www.iot-lab.info/tutorials/monitoring-consumption-m3/

4.3. Representative use cases

This section describes the two representative use cases that are developed specifically for the purpose of this thesis and used for the series of experiments and corresponding runtime measurements. The use cases comprise two distributed applications, namely a program to compute the amount of prime numbers within a given interval, as presented in Section 4.3.1, and one to determine the number of words and lines of a certain text segment, see Section 4.3.2. Both applications have in common that they are based on a master-worker model as demonstrated in Figure 4.2, so there is always one master node that distributes the workload among a predefined number of worker nodes. Additionally, the master node acts as a data sink by collecting all the partial results from the workers after they are finished with processing the input, followed by combining the received results to get an overall result.

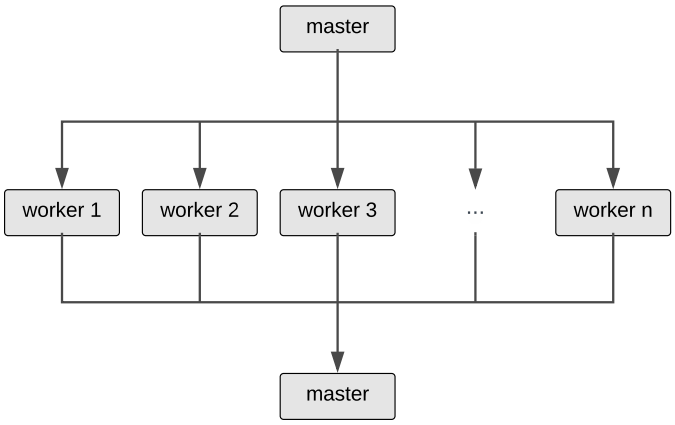


Figure 4.2.: Master-worker model

This communication pattern is also reflected in the topology of the nodes involved in a use case. A star topology is chosen as logical topology for both applications, which is also illustrated in Figure 4.3. In this way, the master node only communicates with the worker nodes, but there is no communication among the worker nodes themselves. Furthermore, the whole communication between the master node and the worker nodes is realized through ZMTP.

For the prime number count use case, the total numeric interval, in which the primes shall be determined, is divided among the workers, whereas a text section is split among the workers in case of the word and line count use case. As stated above, the communication channels between the master node and the worker nodes are established via ZMTP and for each worker, the master needs to create a new thread for handling the ZMTP connection with this particular worker node. For instance, if there are 8 workers in a given experiment, then the master has to create 8 new threads in order to be able to handle all the different ZMTP connections properly. Since a master node always needs to be present within the network, each experiment consists of $n+1$ nodes, whereby n represents the number of workers.

Both applications are developed for RIOT OS and for each use case, two programs are needed, one for the master node and one for the worker nodes, so in total four programs are designed. As mentioned in Section 4.2.1, the use case programs include function calls from

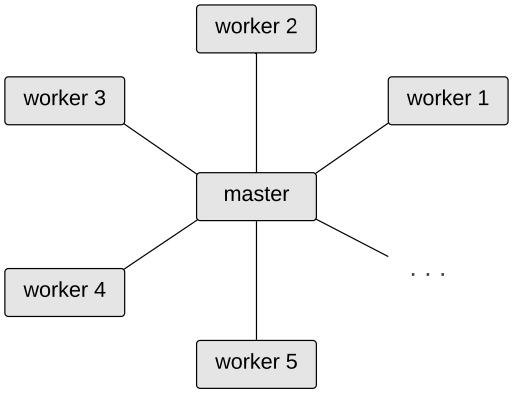


Figure 4.3.: Logical topology of the nodes

the RIOT API to realize the execution time measurements, whereby each of them outputs multiple execution time measurements, namely a measured value for the overall execution time of the master or worker program, respectively, and one or more measured values for the time needed to execute only specific parts of the program. Listing 4.2 constitutes a generalized template that shows how two values for the execution time are measured within the program code.

In case of the master program, the further measurements comprise the separate execution time of each connection handling thread, only excluding the ZMTP connection establishment and termination as well as the IPC that takes place between the connection handling threads and the main thread in order to put the partial results of the individual workers together and generate an overall result. Regarding the word and line count use case, it additionally excludes the time used to compute the array of text passages, which are later sent to the workers, within the main thread. This kind of computation is not necessary in the prime number count use case. As a result, if n workers are involved in an experiment, the master program generates $n + 1$ measured values for the execution time, whereby n values mirror the execution times of the separate connection handling threads and one value represents the overall execution time of the master program.

As for the worker program, that completely runs within the main thread since each worker only needs to maintain one connection, the ZMTP connection establishment and termination are not part of the second measurement as well, so it encompasses the input the worker receives from the master node, the algorithm computing the results, and the dispatch of these results to the master. Thus, two measured values per worker are generated, which also makes it possible to calculate the time required for the connection establishment and termination for each worker by taking the difference between those two values.

The following sections detail the individual design and also the specific workflow of each test case separately, starting with the prime number count application as the first use case, followed by the word and line count program as the second use case.

Listing 4.2: Integration of execution time measurements in the use case programs

```
1  /* Declaration and initialization of variables */
2
3  /* Get current system time for overall execution time at start of
       measurements */
4  const uint32_t time_start_1 = xtimer_now_usec();
5
6  /* Code for connection establishment etc. */
7
8  /* Get current system time for partial execution time at start of
       measurements */
9  const uint32_t time_start_2 = xtimer_now_usec();
10
11 /* Code to be measured within partial execution time */
12
13 /* Get current system time for partial execution time at end of
       measurements */
14 const uint32_t time_end_2 = xtimer_now_usec();
15
16 /* Code for connection termination etc. */
17
18 /* Get current system time for overall execution time at end of
       measurements */
19 const uint32_t time_end_1 = xtimer_now_usec();
20
21 /* Compute overall execution time */
22 const uint32_t exec_time_1 = time_end_1 - time_start_1;
23
24 /* Compute partial execution time */
25 const uint32_t exec_time_2 = time_end_2 - time_start_2;
```

4.3.1. Use case 1 - Prime number count

When starting the master program, the measurement for the overall execution time is triggered right away and the threads that are required to handle the different ZMTP connections with the workers are created within the main thread. This number depends on the predefined number of workers used for a specific test configuration, as described in Section 4.4.2. The main thread then just waits for IPC messages from the connection handling threads containing the partial results as computed by the workers. This is done by calling a blocking receive method of RIOT's IPC API.

Each of the connection handling threads executes the same code, whereby the workflow that includes the communication between one connection handling thread of the master and its connected worker during the computation of prime numbers in a certain interval is shown in Figure 4.4. This workflow also constitutes the part of the program whose execution time is measured within a separate measurement, on both the master and the worker side, excluding the connection establishment and termination in both cases, as well as the dispatch of results to the main thread in case of the master thread measurements. In the following, the term master thread refers to one of the connection handling threads the master maintains in order to communicate with several workers.

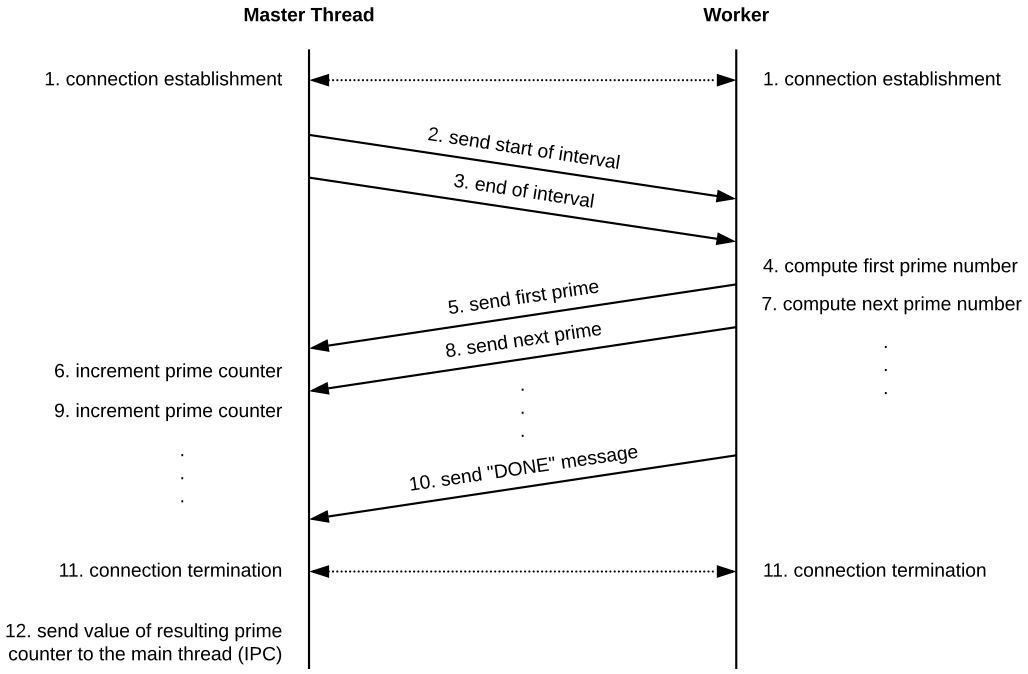


Figure 4.4.: Partial workflow of the prime number count use case

Assuming that the ZMTP connection between a master thread and a worker is established properly (see Figure 4.4: step 1), the master thread determines the numeric interval that is associated with its connected worker and then sends the start and the end of the interval to the worker by transmitting two messages, each containing one interval boundary (see Figure 4.4: step 2-3). So for instance, if the worker has to compute the primes within an interval from 1,001 to 2,000, then the first packets contains “1,001” as data and the second one holds “2,000” as payload. Upon reception of the two numbers, the worker starts with computing the first prime number within the given interval and as soon as it is found, he sends the prime back to the master thread before continuing with further calculations (see Figure 4.4: step 4-5). For each prime number the master thread receives from the worker, a prime counter related to the worker is incremented (see Figure 4.4: step 6). The procedure of calculating the next prime number, sending it to the master thread and incrementing the prime counter is repeated (see Figure 4.4: step 7-9) until the whole interval is processed and all primes are determined by the worker. In order to indicate that the worker is finished with processing the whole interval, he sends a message containing “DONE” to the master thread (see Figure 4.4: step 10). After receiving this message and subsequently closing the ZMTP connection (see Figure 4.4: step 11), the master thread dispatches an IPC message to the main thread, including the resulting prime counter that corresponds to the interval processed by the connected worker (see Figure 4.4: step 12).

As a last step for the computation, the master’s main thread collects and adds up all the partial results received from the different connection handling threads and then triggers the end of the overall execution time measurement. The application is finished when both

the master and the worker programs printed the measured values of the execution time measurements and the corresponding metadata to the console. Due to the transmission of every prime number in a separate packet, which constitutes an intentional design decision, this use case produces a lot of network traffic, however, the payload per packet that is transferred from one node to another only includes one number at a time, thus minimizing the payload and also the packet size.

4.3.2. Use case 2 - Word and line count

Similar to the prime number count use case, the measurement for the overall execution time of the master program is triggered upon start, but unlike the first use case, an array containing the individual text passages needs to be generated by splitting the total text segment into multiple parts that can be divided among the workers. The division of the overall text segment into array fields depends on the number of workers and the number of text passages per worker, as defined by the test configurations in Section 4.4.2. This computation is carried out by the main thread, followed by the creation of the connection handling threads within the main thread in order to ensure that the array holds the proper text passages before the connection handling threads try to access specific fields of the array and send them to the workers. The execution time of this computation is only part of the master program's overall execution time measurement, but it is not included in the measurements carried out for each of the threads' individual execution times.

Like before, the terms master thread and connection handling thread are used interchangeably in the following. Figure 4.5 outlines the steps that are required to process the text passages associated with a certain worker. As in the first use case, a ZMTP connection needs to be established between one of the master threads and a worker (see Figure 4.4: step 1). Afterwards, the master thread is able to initiate the word and line count computation by sending the first text passage to the connected worker (see Figure 4.4: step 2). The worker then processes the passage by counting the words and lines (see Figure 4.4: step 3) and sends the results back to the master thread in two separate packets (see Figure 4.4: step 4-5). Upon reception, the master thread adds the received results to the word and line counters that he maintains specifically for the connected worker (see Figure 4.4: step 6-7). Subsequently, the master thread dispatches the next text passage assigned to the worker (see Figure 4.4: step 8), which again processes the passage and sends the results back to the master thread (see Figure 4.4: step 9-11). The master thread then updates the word and line counter accordingly (see Figure 4.4: step 12-13), before the ZMTP connection is terminated (see Figure 4.4: step 14). Similar to the first use case, it sends the resulting word and line counter to the main thread via IPC for further calculations (see Figure 4.4: step 15), as soon as all the associated text passages are processed. In this example, the worker needs to process two text passages, but depending on the test configuration, step 8-13 of Figure 4.4 might be repeated an arbitrary amount of times or omitted completely, as specified by the particular test configuration.

Just like in the implementation of the first use case, the master's main thread finally acts a data sink by totaling the partial results received from the connection handling threads and also stops the overall execution time measurement of the master program before printing the measured values together with the corresponding metadata to the console.

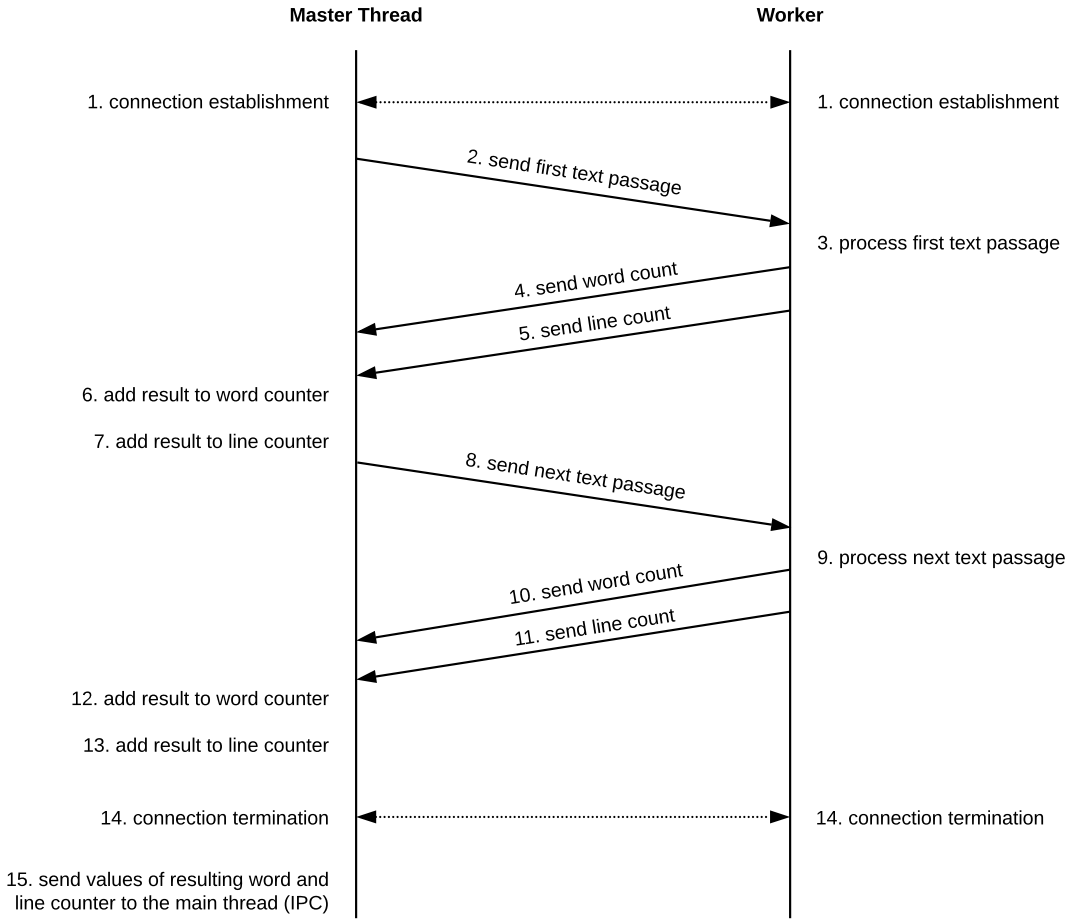


Figure 4.5.: Partial workflow of the word and line count use case

The word and line count use case differs from the prime number count use case in such a way that the input, which is sent to the workers, consists of a whole text passage rather than only two separate numbers. Additionally, on the worker side, there are only two results per input that need to be sent back to the master node, namely the number of words and the number of lines counted within the text, as compared to the first use case, where the workers send each prime number back to the master individually instead of aggregating the result. In this way, the word and line count use case produces less network traffic in terms of the number of packets transmitted from the worker nodes to the master node, but in order to provide the text passages to the workers, the master threads need to dispatch them through the network resulting in generally bigger payload and thus packet sizes as opposed to the prime number count use case.

4.4. Setups

All experiments and associated measurements are conducted by means of a consistent setting in order to ensure steady state conditions. Therefore, the test environment that was used in the course of this thesis is presented in Section 4.4.1. Furthermore, the predefined test configurations as needed for the measurements and the associated scaling scenarios are introduced in Section 4.4.2.

4.4.1. Test environment

All the experiments, that have to be conducted as part of this thesis, are deployed on the FIT IoT-LAB, which represents a large-scale experimental testbed for IoT applications. As introduced by the authors of [ABF+15] and described on the official IoT-LAB website[14], this testbed provides open access to a remote testing environment that encompasses a large number of heterogeneous low-power network nodes. In total, it comprises 1,791 wireless nodes which are located at multiple sites in France, namely Grenoble, Lille, Saclay, Strasbourg, Paris and Lyon, and form low-power wireless sensor networks. This enables users to develop and test their applications on real physical hardware and under realistic conditions by running experiments on previously selected nodes. The IoT-LAB offers static and mobile nodes, that are fully-programmable and can be arbitrarily allocated, since all nodes of the IoT-LAB, which are spread across the six testbed sites, are interconnected through a global networking backbone.

Each IoT-LAB node is composed of three different components, an Open Node (ON), a Gateway (GW) and a Control Node (CN). During an experiment, the user has full access to the memory of the open node, which represents a programmable low-power device, so any firmware can be uploaded and flashed onto the ON. The real-time access to the nodes is granted by a web portal, as well as by Command-Line Interface (CLI) tools that can be accessed via the Secure Shell (SSH). The serial port of the ON and also the one of the CN are connected to the gateway, which in turn provides a wired connection to the backbone and thus to the back-end servers of the IoT-LAB. Together with the gateway, the control node is capable of monitoring the open node in terms of consumption and other measured values obtained from sensors. In the course of this thesis, these monitoring capabilities are used for measuring the power consumption of the individual nodes during the experiments, as described in Section 4.2.2. Furthermore, the CN selects the power supply, namely battery or Power-over-Ethernet (PoE).

In order to provide a variety of IoT hardware, the IoT-LAB offers three different types of nodes, which are characterized by the open node they operate. The available open nodes, namely WSN430, M3 and A8 open nodes, differ in their processor architectures and wireless chips. WSN430 open nodes are based on ultra-low-power micro-controller units (MCUs) with only 48 kB of Flash memory and 10kB of RAM, while A8 open nodes rather constitute high performance IoT-LAB nodes that are capable of running high-level operating systems such as Linux, so they can execute applications typically deployed on smartphones or tablets. On the contrary, M3 open nodes can be categorized somewhere in between the WSN430 and the

[14] https://www.iot-lab.info/

A8 in terms of memory capacity and applicability, since they operate 32-bit ARM Cortex-M3 MCUs with up to 64 kB of RAM and up to 512 kB of Flash memory. Compared to WSN430 and A8 nodes, M3 open nodes exhibit the most appropriate properties and are considered to be most suitable for the purpose of this thesis, that is assessing the scalability of ZMTP in constrained environments. Figure 4.6 represents an M3 ON operating an STM32F103REY micro-controller as well as an AT86RF231 radio interface and various sensors, e.g. pressure or temperature sensors. Like most of the WSN430 and A8 open nodes, M3 open nodes communicate within the 2.4 GHz frequency band, since they use the IEEE 802.15.4 wireless transmission standard, which facilitates the intercommunication between them.

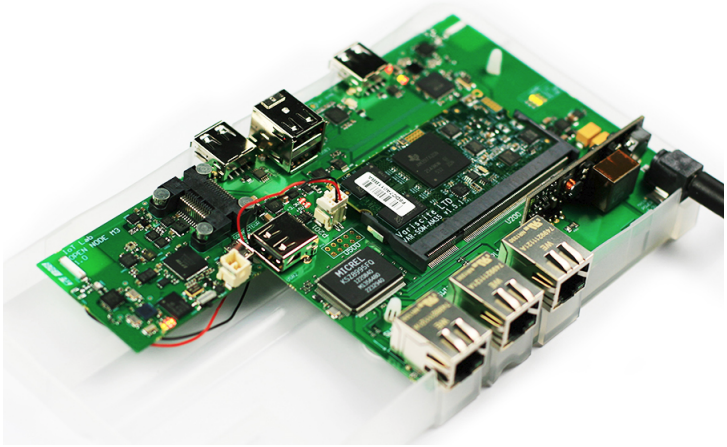


Figure 4.6.: An IoT-LAB node operating an M3 open node[15]

To sum it up, the IoT-LAB offers an open-access scientific testbed with a large number of heterogeneous nodes that can be arbitrarily selected, managed, reprogrammed and monitored by the user during an experiment and therefore it improves the analysis, evaluation and reproducibility of experiments and their results, while mirroring real-world conditions. For those reasons, it is considered a suitable physical infrastructure and test environment for the purpose of conducting experiments including measurements performed at runtime as part of this thesis.

4.4.2. Test configurations and scaling scenarios

Compared to the definition of the term "configuration" in Section 4.1.2, which refers to a configuration that defines a whole distributed computing systems in terms of specific values for each problem space dimension described before, a *test configuration* specifies the selection of certain values regarding only a subset of these problem space dimensions. Since solely the application itself, the total input size and the number of workers are varied during the experiments, while the other dimensional values remain constant, a test configuration consists of values for exactly those three dimensions. Combinations of different test configurations, in turn, form several scaling scenarios for the experimental approach. A scenario is defined by a set of test configurations with distinct properties in order to reflect various types of

[15] https://www.iot-lab.info/hardware/

scaling. Each block within the tables described below represents one or more scaling scenarios consisting of multiple test configurations. With respect to the experiments performed within this thesis, the only possible values for the application are the two use case programs introduced in Section 4.3. The selected values for the other dimensions are detailed in the following.

Prime number count use case As shown in Table 4.2, which outlines all the test configurations involved in the scaling scenarios of the prime number use case (see Section 4.3.1), the input size and the number of workers are varied as follows.

In terms of the scenarios that focus on varying the total input size, the total interval of numbers, in which the primes shall be computed within a specific scenario, is continuously expanded. Therefore, it contains either 1,000, 2,000, 3,000, 4,000 or 10,000 numbers, while the interval itself always starts from 1,001. The complete interval is then divided among a constant number of workers, either 1, 2, 4, 8 or 10, as illustrated in the first five blocks of Table 4.2. The reason for choosing 1,001 as the lower interval boundary is that the primes within this interval are more evenly spread compared to the interval from 1 to 1,000, thus fostering a fairer distribution of work among the workers, since the complexity of the individual intervals does not differ as much.

Furthermore, the prime number count use case encompasses several strong scaling scenarios. Within each strong scaling scenario, the total input size is fixed, while the number of workers is increased. With respect to the prime number count use case, the number of workers is scaled between 1 and 10, to be more specific, the individual values are 1, 2, 4, 8 and 10, while the overall interval remains the same within each scenario. The scenarios in which only the number of workers are varied can be composed of individual test configurations of the other scenarios, resulting in five different test scenarios, whereby the specific input sizes represent intervals of 1,000, 2,000, 3,000, 4,000 or 10,000 numbers, respectively. For reasons of space, the penultimate block of Table 4.2 aggregates exactly these five scenarios reflecting strong scaling.

The last block of Table 4.2 summarizes the two weak scaling scenarios of the prime number count use case. These scenarios are characterized by an increasing total input size, while the number of workers is also scaled by the same factor, so both of these dimensions are varied. The number of workers represents either 1, 2, 4 or 8 within both scenarios, thereby representing a growth rate of two, whereas the values for the input size differ among the two weak scaling scenarios. The first one starts with a total interval size of 1,000, so with a scaling factor of two, the values for the input size of the remaining test configurations are 2,000, 4,000 and 8,000, respectively. On the contrary, the second scenario defines either 2,000, 4,000, 8,000 or 16,000 as total input size, depending on the specific test configuration. Some of test configurations of these scenarios are equal to the ones of other scenarios, but they also include new ones that are not used in any other scenario apart from these ones.

The complete scenarios with their associated test configurations are illustrated in the tables of the subsequent chapter, where they are compared to each other on the basis of the resulting measured values derived from the experiments.

Application	Workers	Total input size
Prime number count	1	1,000 numbers
Prime number count	1	2,000 numbers
Prime number count	1	3,000 numbers
Prime number count	1	4,000 numbers
Prime number count	1	10,000 numbers
Prime number count	2	1,000 numbers
Prime number count	2	2,000 numbers
Prime number count	2	3,000 numbers
Prime number count	2	4,000 numbers
Prime number count	2	10,000 numbers
Prime number count	4	1,000 numbers
Prime number count	4	2,000 numbers
Prime number count	4	3,000 numbers
Prime number count	4	4,000 numbers
Prime number count	4	10,000 numbers
Prime number count	8	1,000 numbers
Prime number count	8	2,000 numbers
Prime number count	8	3,000 numbers
Prime number count	8	4,000 numbers
Prime number count	8	10,000 numbers
Prime number count	10	1,000 numbers
Prime number count	10	2,000 numbers
Prime number count	10	3,000 numbers
Prime number count	10	4,000 numbers
Prime number count	10	10,000 numbers
Prime number count	1	1,000 — 2,000 — 3,000 — 4,000 — 10,000 numbers
Prime number count	2	1,000 — 2,000 — 3,000 — 4,000 — 10,000 numbers
Prime number count	4	1,000 — 2,000 — 3,000 — 4,000 — 10,000 numbers
Prime number count	8	1,000 — 2,000 — 3,000 — 4,000 — 10,000 numbers
Prime number count	10	1,000 — 2,000 — 3,000 — 4,000 — 10,000 numbers
Prime number count	1	1,000 — 2,000 numbers
Prime number count	2	2,000 — 4,000 numbers
Prime number count	4	4,000 — 8,000 numbers
Prime number count	8	8,000 — 16,000 numbers

Table 4.2.: Scenarios and associated test configurations for the prime number count use case

Word and line count use case Regarding the second use case (see Section 4.3.2), the input refers to the text section that needs to be processed in total by dividing it into multiple text passages which are individually assigned to the workers. Within this use case, not the total size of the text segment itself is altered, but rather the number of text passages every worker receives, so either the passage associated to a certain worker is sent as a whole or it is further split into smaller passages that are sent to the worker separately, one at a time and only when he is finished with the previous text passage. This procedure is used because the current design of the word and line count use case does not allow for an arbitrarily big

text size, since this would exceed the memory capacity of the constrained nodes. As a result, weak scaling scenarios cannot be realized within this use case. Therefore, the total input size for all test configurations is constant and simulated by a text passage with 1,345 words divided into 117 lines.

As for the varying text passage allocation scenarios, the number of text passages per worker ranges from 1 to 4, increased in linear steps with a growth rate of one. Depending on the specific scenario, the number of workers is constantly set to 1, 2, 4, 8 or 10, respectively. The resulting scenarios are represented by the first five blocks of Table 4.3.

Additionally, the word and line count use case includes strong scaling scenarios similar to the prime number count use case. The values for the number of workers used in the scenarios of the first use case also apply to the word and line count application, so the number of workers ranges between 1, 2, 4, 8 and 10. Within each of these scenarios, the text section representing the total input size also covers 1,345 words and 117 lines and is split into either 1, 2, 3 or 4 individual passages per worker, which leads to four scenarios that are again aggregated into one block, namely the last block of Table 4.3. Similar to the first use case, these four scenarios are combinations of test configurations from other scenarios.

Application	**Workers**	**Total input size**
Word and line count	1	1,345 words : 1 passage/worker
Word and line count	1	1,345 words : 2 passages/worker
Word and line count	1	1,345 words : 3 passages/worker
Word and line count	1	1,345 words : 4 passages/worker
Word and line count	2	1,345 words : 1 passage/worker
Word and line count	2	1,345 words : 2 passages/worker
Word and line count	2	1,345 words : 3 passages/worker
Word and line count	2	1,345 words : 4 passages/worker
Word and line count	4	1,345 words : 1 passage/worker
Word and line count	4	1,345 words : 2 passages/worker
Word and line count	4	1,345 words : 3 passages/worker
Word and line count	4	1,345 words : 4 passages/worker
Word and line count	8	1,345 words : 1 passage/worker
Word and line count	8	1,345 words : 2 passages/worker
Word and line count	8	1,345 words : 3 passages/worker
Word and line count	8	1,345 words : 4 passages/worker
Word and line count	10	1,345 words : 1 passage/worker
Word and line count	10	1,345 words : 2 passages/worker
Word and line count	10	1,345 words : 3 passages/worker
Word and line count	10	1,345 words : 4 passages/worker
Word and line count	1	1,345 words : 1 — 2 — 3 — 4 passages/worker
Word and line count	2	1,345 words : 1 — 2 — 3 — 4 passages/worker
Word and line count	4	1,345 words : 1 — 2 — 3 — 4 passages/worker
Word and line count	8	1,345 words : 1 — 2 — 3 — 4 passages/worker
Word and line count	10	1,345 words : 1 — 2 — 3 — 4 passages/worker

Table 4.3.: Scenarios and associated test configurations for the word and line count use case

In summary, three dimensions are considered in the scaling scenarios of the two use cases, namely the application, the number of workers and the input. Table 4.2 and Table 4.3 therefore summarize the resulting scenarios, that encompass multiple test configurations characterized by their corresponding dimensional values as used for the experiments. For each application, the number of workers is increased by multiple steps and the input is varied as defined by the separate scenarios, which leads to 21 different test scenarios in total, 12 for the prime number count use case and 9 for the word and line count use case. In this way, the impact and scaling behavior of strong and weak scaling, among others, can be determined and evaluated as part of the subsequent chapter.

4.5. Test procedure

As mentioned in Section 4.4.1, all the experiments conducted in the course of this thesis are deployed on nodes of the IoT-LAB testbed. All the nodes required for the experiments are reserved at the IoT-LAB site in Grenoble, whereby the *m3-1* node serves as the master node in every experiment that is submitted to the testbed. The adjacent nodes, namely *m3-2*, *m3-3*, *m3-4*, and so on, are then used as worker nodes. In this way, the spatial arrangement of the nodes is the same for all experiments, only the number of reserved nodes differs depending on the test configurations. Additionally, this procedure only requires the recompilation of the master program when starting with another test configuration, whereas the worker program does not need to be changed at all, since the number of workers and the total input size are defined in the master program of each application and the IP address of the master node, which is specified in the makefile of the worker program, remains the same during the whole series of experiments. The compiled Executable and Linkable Format (ELF) binaries can then be uploaded on the web portal of the testbed, where they are automatically flashed onto the assigned nodes.

Each individual test configuration was repeated five times in a row by using the "reset" feature of the IoT-LAB web portal, so all nodes involved in an experiment can be reset at the same time. Therefore, each experiment submitted to the testbed through the web portal represents a series of five repetitions of the same configuration, so five measured values can be obtained for each measurement of a test configuration. Both use cases together comprise 48 disparate test configurations, whereby the prime number count use case makes up 28 thereof, while the word and line count use case accounts for 20, since some of the test configurations can be used for more than one scenario. As a result, 240 runs split among 48 experiments have to be carried out in total.

In order to aggregate the various measured values for the execution time measurements that are printed to the nodes' consoles together with the corresponding metadata at the end of each run, the *serial aggregator* tool of the IoT-LAB is used. This tool enables the aggregation of the output of all nodes participating in an experiment via SSH by accessing the serial links of the nodes all at once.

During the course of the experiments, the resulting measured values for the execution time as obtained by repeated measurements of test configurations are collected and aggregated into multiple files for further processing. The concrete measurement results are presented, analyzed and evaluated in the following chapter.

5. Evaluation and scaling conclusions

This chapter covers the evaluation and consequent scaling conclusions that can be derived from the measurement results of the experiments. As already mentioned, the primary goal of this thesis is to assess the scalability of ZMTP in constrained environments, whereby an experimental approach based on two different use cases is chosen as methodology, enabling to determine whether the protocol shows the expected and desired scaling behavior or not. The desired scaling behavior in terms of strong scaling would be a speedup in performance when increasing the number of workers within different scenarios, while keeping the total input size fixed. Regarding weak scaling, the desired behavior would be the processing of a total input size that is n times bigger in the same amount of time since the number of workers is also increased by n, so the same growth rate is used to scale both dimensions.

In order to evaluate the scaling efficiency of the ZeroMQ Message Transport Protocol, a profound and qualitative analysis of the measurement data produced by the series of experiments is conducted, which consequently results in conclusions about the scalability of the protocol drawn from the elaborated findings. Section 5.1 generally deals with the measurement accuracy as well as the calculation of the standard deviation of measured values, while Section 5.2 outlines the measurement results and includes observations made on the basis of these results. Additionally, Section 5.3 identifies potential causes for the observed behavior of ZMTP and Section 5.4 introduces the consequent scaling conclusions.

5.1. Measurement accuracy and standard deviation

Before the results of the runtime measurements are presented and evaluated, the measurement accuracy needs to be considered individually and also in conjunction with the standard deviation of measurements.

In general, measurements should not influence one another, but due to the design and the chosen measurands of the use cases, this is inevitable, since the overall runtime of each node and also the measured execution time values of only parts of the application are variables of interest and thus all of them are needed in order to allow for an in-depth analysis of the measured values. As a result of the execution time measurements being realized through two function calls, each measured value for the total execution time of a node is slightly influenced by other function calls required for further measurements because they reside within the code that is observed through this overall measurement. Depending on the number of workers, which is equal to the number of connection handling threads created by the master, the influence on the total execution time of the master node increases with a higher number of workers because each connection handling thread performs his own measurement, so the amount of measurements might differ between separate test configurations. The impact

on the overall execution time of a worker node is constant, since the worker program only contains two execution time measurements in total, no matter which test configuration is considered. In this way, the time required for the two function calls delivering the current system time plus the calculation of the resulting measured value for each execution time measurement is included in the overall execution time measurements, but since these function calls and computations are rather simple ones and therefore do not take a significant amount of time, they can be omitted.

Concerning the authenticity of the experiments and associated measurements, the choice of test environment is decisive. For this reason, the IoT-LAB testbed is used for all experiments because it comprises a large number of nodes, thereby enabling the deployment of applications on real physical hardware and mirroring real-life steady state conditions, as mentioned in Chapter 4. When multiple nodes are communicating with each other in a wireless environment, radio interference can occur, possibly resulting in different values for the radio signal power measured by the Received Signal Strength Indication (RSSI). Since the IoT-LAB provides numerous nodes that can be split among various users conducting their experiments at the same time, realistic radio interference is being caused. Therefore, this remote testbed is considered a suitable test infrastructure implying an ideal trade-off between a managed or controlled setting and realistic conditions. Additionally, the reproducibility of the experiments is improved by using the IoT-LAB, since all the experiments are performed on the same nodes and only stationary nodes are used, so the spatial arrangement of these nodes is not altered and the experiments can be accurately replicated.

On the downside, environmental influences caused by the testbed might have impacts on the runtime measurements. For instance, radio interference can have negative effects on the runtime. The IoT-LAB site in Grenoble operates 640 nodes in total, whereby 384 thereof are M3 open nodes as used for the experiments and 256 are A8 open nodes[16], so all of them realize wireless radio communication through the IEEE 802.15.4 radio transmission standard and within the 2.4 GHz frequency band by including an AT86RF231 radio interface. Depending on how many nodes are actively used in other experiments while measuring the runtime of one's own experimental setup, the radio interference can fluctuate. As a result, the end-to-end latency might be affected and can thus differ between multiple nodes and also between individual experimental runs. Furthermore, adjacent nodes involved in the same experiment could potentially even influence each other.

Standard deviation

With respect to the reliability of the measurement results, the standard deviation is calculated for the five different measured values of each test configuration in order to determine the amount of discrepancies between repeated measurements. According to [BA96], measured values resulting from several repeated measurements on the same subject are not necessarily constant. Instead, a series of measurements has varying results due to a measurement error. Regarding the experiments conducted within this thesis, the term "subject" refers to a certain test configuration. By assuming that all measured values of the same test configuration are normally distributed, the size of the measurement error for repeated measurements can

[16] https://www.iot-lab.info/deployment/grenoble/

be assessed through the sample standard deviation. The sample standard deviation is then defined as the square root of the variance of the measured values.

Before the standard deviation is computed, potential outliers of single measurements within a series of experiments can be detected and consequently eliminated, since such outliers might significantly influence the average runtime and also the sample standard deviation calculated from multiple repetitions of the same test configuration. Within this thesis, each measured value is considered an important part of the result and therefore the procedure of detecting and removing individual outliers is not part of this work. As a result, outliers that are not eliminated might cause high values for the sample standard deviation of a test configuration.

Since the measurements constitute only samples of a larger population of measurements, the sample standard deviation is applied instead of the population standard deviation. In order to determine the sample standard deviation of the measured values derived from repeated measurements, R is chosen for the statistical computations[17]. As stated in [BHL+12], the following formula defines how the sample standard deviation for temporarily consecutive measurements of the same test configuration is calculated, which resembles the *sd(x)* function in R, whereby n represents the number of measurements, x_i the measured value for measurement i and $\bar{x}$ the mean measured value.

$$sd(x) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

If the resulting value of the sample standard deviation, which represents an estimate of the standard deviation of the whole population, is low, then this points out to the fact that the individual measured values are located rather close to their mean as a general rule. On the contrary, a high standard deviation refers to the measured values being more distributed in a wider range of measured values.

The value of the sample standard deviation calculated by the formula above has the same unit as the measured values used for the computation thereof, so for instance, when determining the standard deviation of repeated execution time measurements, then the resulting value is also specified in minutes. Since the associated test configurations usually differ in their means, a relative standard deviation is required to accurately compare the results among test configurations with various means. The relative standard deviation, which is also called the coefficient of variation (CV), is defined as the quotient of the sample standard deviation to the mean, as stated in [Abd10]. This formula yields a decimal number that can then be multiplied by 100 in order to assess the relative standard deviation in percent, indicating the scattering of the measured values relative to the mean.

In the following sections, the sample standard deviation as well as the coefficient of variation are determined for all test configurations of the prime number count and the word and line count use case.

[17] https://www.r-project.org/

5.2. Measurement results and observations

This section presents the results of the runtime measurements conducted during the series of experiments by thoroughly assessing the measured values and deriving observations thereof. The focus of this section is on the measured values for the execution time, which are investigated in Section 5.2.1, while the power consumption measurements are covered in Section 5.2.2. The following sections mainly introduce the observations that can be derived from the measurement results, while possible explanations and potential causes for these findings are elaborated in Section 5.3, since many of them apply to both use cases and can thus be aggregated into a single section.

5.2.1. Execution time results

The results of the execution time measurements for both use cases are presented in tables and also plotted using R for the statistical computations and for generating multiple graphs to visualize the results. In the following, these results are analyzed and discussed, with respect to the scaling behavior of ZMTP in distributed applications. While the tables together include all scenarios of each use case, not every scenario is additionally represented graphically for reasons of space, since most of the plots look similar. Therefore, only some of them are selected and included in the following. The remaining plots that are not contained in the following can be found in the appendix, along with some excerpts of the raw measured values (see Appendix A). In the remainder of this thesis, the total runtime of the application refers to the overall execution time measured by the master node.

Prime number count use case

First, the measurement results of the prime number count use case are presented and analyzed. Within this use case, the total input size as specified in the tables refers to the amount of numbers that are contained in the overall interval that needs to be processed by all the workers together, while this interval is equally split among the specified number of workers. For instance, a total input size of 1,000 means that the workers together have to process an interval of 1,000 numbers and determine the primes within this interval.

The various scenarios of this use case can be divided into categories, depending on the dimensions that are being scaled within each of these scenarios, resulting in three categories, namely varying input size, strong scaling and weak scaling. Before the measured values are plotted, the sample standard deviation and the coefficient of variation of the repeated measurements for each individual test configuration are calculated. This approach also allows for including error bars derived from the sample standard deviation inside the plots.

Varying input size scenarios As for the repeated measurements of each test configuration included in the varying input size scenarios, Table 5.1 shows the average overall execution time of the master program in minutes, which is associated with the mean total runtime of the application. Furthermore, it comprises the corresponding sample standard deviation in minutes, as well as the coefficient of variation in percent. Within each of these scenarios,

the total input size is increased for a fixed number of workers, so the overall number interval grows and therefore the workload per worker is raised accordingly.

As mentioned in Chapter 4, five temporarily consecutive measurements of each test configuration are performed, whose resulting measured values are taken into account for the calculations. All values within this table — as well as the values of the subsequent tables — are rounded to two decimal places.

Workers	Total input size	Mean runtime	Stand. deviation	CV
1	1,000 numbers	4.79 min	0.14 min	2.85 %
1	2,000 numbers	7.74 min	0.11 min	1.43 %
1	3,000 numbers	10.82 min	0.37 min	3.39 %
1	4,000 numbers	13.83 min	0.24 min	1.70 %
1	10,000 numbers	28.99 min	0.45 min	1.57 %
2	1,000 numbers	3.08 min	0.11 min	3.62 %
2	2,000 numbers	4.83 min	0.14 min	2.84 %
2	3,000 numbers	6.40 min	0.21 min	3.29 %
2	4,000 numbers	8.04 min	0.23 min	2.87 %
2	10,000 numbers	16.93 min	0.57 min	3.37 %
4	1,000 numbers	2.39 min	0.11 min	4.63 %
4	2,000 numbers	3.25 min	0.07 min	2.17 %
4	3,000 numbers	4.14 min	0.09 min	2.21 %
4	4,000 numbers	5.24 min	0.29 min	5.54 %
4	10,000 numbers	9.99 min	0.36 min	3.59 %
8	1,000 numbers	2.34 min	0.21 min	9.13 %
8	2,000 numbers	3.03 min	0.25 min	8.40 %
8	3,000 numbers	3.68 min	0.16 min	4.38 %
8	4,000 numbers	4.07 min	0.19 min	4.63 %
8	10,000 numbers	7.27 min	0.44 min	6.03 %
10	1,000 numbers	2.30 min	0.08 min	3.39 %
10	2,000 numbers	2.92 min	0.17 min	5.87 %
10	3,000 numbers	3.49 min	0.12 min	3.35 %
10	4,000 numbers	3.97 min	0.18 min	4.61 %
10	10,000 numbers	7.40 min	0.36 min	4.88 %

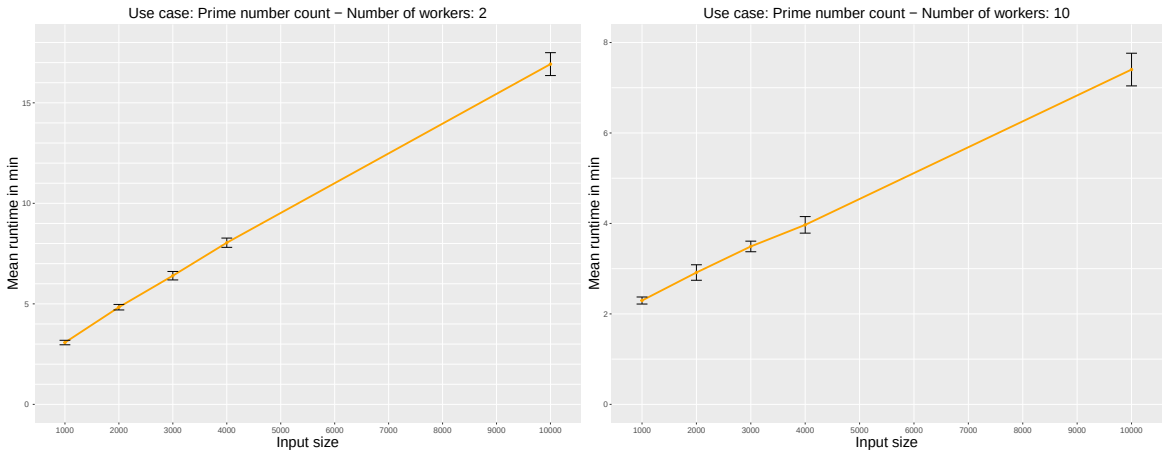
Table 5.1.: Varying input size scenarios based on the prime number count use case including mean runtime, sample standard deviation and coefficient of variation (CV)

Regarding the sample standard deviations of the test configurations within each scenario, the value for the test configuration involving 10,000 as total input size is always the highest one in each of the scenarios, but the concrete values of the sample standard deviations can only be interpreted in relation to the corresponding mean. Since the means differs among the test configurations, the sample standard deviations cannot be directly compared to each other. Therefore, the relative standard deviation is primarily considered in the remainder of this thesis. One of the observations that can be deduced from Table 5.1 is that all the values for the coefficient of variation, denoted as CV inside the table, are relatively low for all test configurations, whereby 1.43 % and 9.13 % represent the lowest respectively highest value

within this table. This indicates that the individual measured values for the total execution time within these test configurations are not widely spread out over a large range of values, but rather located close to the mean, so none of them includes significant outliers that are not in line with the remaining measured values. So generally, the measurements of these test configurations are relatively reliable.

Within each scenario, the coefficients of variation, which refer to the relative standard deviation, do not differ significantly among the associated test configurations. The second scenario encompasses the smallest range with values between 2.84 % and 3.62 % for the coefficients of variation, whereas the fourth scenario covers the largest range of CVs with values between 4.38 % and 9.13 %. But in general, no solid patterns in terms of a continuous increase or decrease of the coefficients of variation can be identified in Table 5.1, neither within nor among the scenarios. Potential causes of fluctuations between measurements and among standard deviations are further explained in Section 5.3. Concerning the mean runtime, a continuous increase can be observed within each of the scenarios in Table 5.1, which indicates that the average overall execution time rises with a bigger input size. This reflects the expected behavior, since the workload per worker is raised from one test configuration to another.

In order to facilitate a better traceability of the observations made in terms of the mean runtime and the corresponding standard deviations, the results of two scenarios are additionally illustrated by means of two plots, whereby the second scenario, which involves two workers, is outlined in Figure 5.1a and the last scenario, which contains ten workers, is presented in Figure 5.1b. Both figures are based on Table 5.1 and demonstrate how the average overall execution time changes — visualized by the orange/gray line inside each plot — when increasing the workload per worker realized through varying the total input size, while the number of workers stays constant. The black error bars within the graphs reflect the sample standard deviations of Table 5.1, but the size of the different error bars cannot be contrasted among the plots because the ranges of the y axes differ, which is also indicated by the size of the grid tiles.



(a) Scenario: 2 workers

(b) Scenario: 10 workers

Figure 5.1.: Plots of two different varying input size scenarios based on the prime number count use case including error bars

As expected, the execution time continuously increases on average with a larger total input size, but raising the input size by a certain rate does not result in a multiplication of the runtime by the same rate, which can be deduced from the two plots, but also from the table above. The reason for this behavior is that the total execution time is mainly composed of the computation itself, as well as the time needed for the connection establishment, termination and the actual communication between the master node and the worker nodes. When doubling the total input size, the interval in which the primes are calculated is doubled, so the computational burden is duplicated and the amount of communication between the master and the worker nodes increases because more primes have to be dispatched to the master node. As a result, these two parts of the total execution take longer than before, but the time required for starting and ending the connections should remain constant, which is why the total runtime results in a value less than twice as much when the input size is doubled. This effect is intensified when using a larger growth rate, e.g. regarding the scenario with two workers as presented in Figure 5.1a, the mean runtime for 1,000 numbers as total input size is around 3 minutes, while it takes nearly 17 minutes to complete an input of 10,000 numbers, instead of requiring the tenfold amount of time. When looking at the scenario displayed in Figure 5.1b, which comprises ten workers, it takes 2.3 minutes on average to process an input of 1,000 numbers, whereas 7.4 minutes are needed for a tenfold higher input size. As mentioned before, only the plots of two scenarios are presented here, since the diagrams of the remaining scenarios look similar, so they are omitted here for reasons of space, but can be found in the appendix.

In summary, the lines of both graphs are nearly straight and rising steadily, thereby indicating a continuous increase of the average overall execution time when raising the total input size, while the number of workers remains constant.

Strong scaling scenarios Similar to the varying input size scenarios, the scenarios simulating strong scaling are presented in a table with the associated test configurations and their mean runtime, sample standard deviation as well as the coefficient of variation. Table 5.2 presents exactly these values measured and calculated for the scenarios where the total input size is fixed and only the number of workers is increased. The goal of strong scaling is to process the same overall input in a smaller amount of time due to a larger number of workers, whereby the optimum would be a linear speedup. In case of a linear speedup, the execution is performed n times faster when n workers are involved, as compared to the total runtime it takes if only one worker is responsible for the computation, which represents the reference value. The speedup S of a fixed problem size N and a specific number of workers W can then be calculated by means of the average total execution time T. The resulting formula[18] is then defined as follows.

$$S(N, W) = \frac{T(N, 1)}{T(N, W)}$$

Typically, the actual value of $S(N, W)$ is smaller than W, since an equivalence to W indicates a linear speedup and because of Amdahl's law this cannot be achieved[19]. According to

[18] https://www.archer.ac.uk/training/course-material/2016/12/mpi_scaling_manc/Slides/Scaling.pdf
[19] https://www.cac.cornell.edu/education/training/StampedeJan2015/Scalability.pdf

Amdahl's law, a program can be divided into two kinds of categories, namely sequential sections of the program and potentially parallel sections[18]. The serial fractions cannot be parallelized, so no speedup can be accomplished for those parts of the program, which is why a linear speedup of the total runtime is hardly possible. Nevertheless, the goal of strong scaling is to get as close to a linear speedup as possible. Therefore, Table 5.2 additionally includes the calculated values for the speedup of the different test configurations.

Workers	Total input size	Mean runtime	Speedup	Stand. dev.	CV
1	1,000 numbers	4.79 min	—	0.14 min	2.85 %
2	1,000 numbers	3.08 min	1.56	0.11 min	3.62 %
4	1,000 numbers	2.39 min	2.00	0.11 min	4.63 %
8	1,000 numbers	2.34 min	2.05	0.21 min	9.13 %
10	1,000 numbers	2.30 min	2.09	0.08 min	3.39 %
1	2,000 numbers	7.74 min	—	0.11 min	1.43 %
2	2,000 numbers	4.84 min	1.60	0.14 min	2.84 %
4	2,000 numbers	3.25 min	2.38	0.07 min	2.17 %
8	2,000 numbers	3.03 min	2.56	0.26 min	8.41 %
10	2,000 numbers	2.92 min	2.65	0.17 min	5.87 %
1	3,000 numbers	10.82 min	—	0.37 min	3.39 %
2	3,000 numbers	6.40 min	1.69	0.21 min	3.29 %
4	3,000 numbers	4.14 min	2.62	0.09 min	2.21 %
8	3,000 numbers	3.68 min	2.94	0.16 min	4.38 %
10	3,000 numbers	3.49 min	3.10	0.12 min	3.35 %
1	4,000 numbers	13.83 min	—	0.24 min	1.70 %
2	4,000 numbers	8.04 min	1.72	0.23 min	2.87 %
4	4,000 numbers	5.24 min	2.64	0.29 min	5.54 %
8	4,000 numbers	4.07 min	3.40	0.19 min	4.63 %
10	4,000 numbers	3.97 min	3.48	0.18 min	4.61 %
1	10,000 numbers	28.99 min	—	0.45 min	1.57 %
2	10,000 numbers	16.93 min	1.71	0.57 min	3.37 %
4	10,000 numbers	9.99 min	2.90	0.36 min	3.59 %
8	10,000 numbers	7.27 min	3.99	0.44 min	6.03 %
10	10,000 numbers	7.40 min	3.92	0.36 min	4.88 %

Table 5.2.: Strong scaling scenarios based on the prime number count use case including mean runtime, speedup, sample standard deviation and coefficient of variation (CV)

Except for the last scenario, the calculated values for the speedup generally increase within each scenario as the number of workers is raised, which represents a desired behavior as explained above. This observation can be traced back to the fact that the mean runtime continuously decreases within these scenarios. Additionally, it can be observed that the speedup values grow in relation to a higher total input size, so they are usually closer to their optimum as the total input size increases. In general, this finding indicates that ZMTP probably scales better in case of a higher total input size as opposed to a smaller one. For example, the maximum speedup of the first scenario, which defines a total input size of 1,000

numbers, only amounts to 2.09 in case of ten workers, whereas the scenario specifying 10,000 as input size exhibits a maximum speedup of 3.99, which is the case when eight workers are involved. The speedup of 3.99 constitutes the maximum speedup value of the whole table, so this value marks the saturation point of the prime number count use case within the strong scaling scenarios. From there on, a slight drop in performance can be identified when looking at the speedup, which only amounts to 3.92, and the mean runtime of the test configuration defining the same total input size, but split among ten workers instead of eight. The reason for this behavior is probably the outweighing overhead, as further detailed in conjunction with the plots belonging to these scenarios.

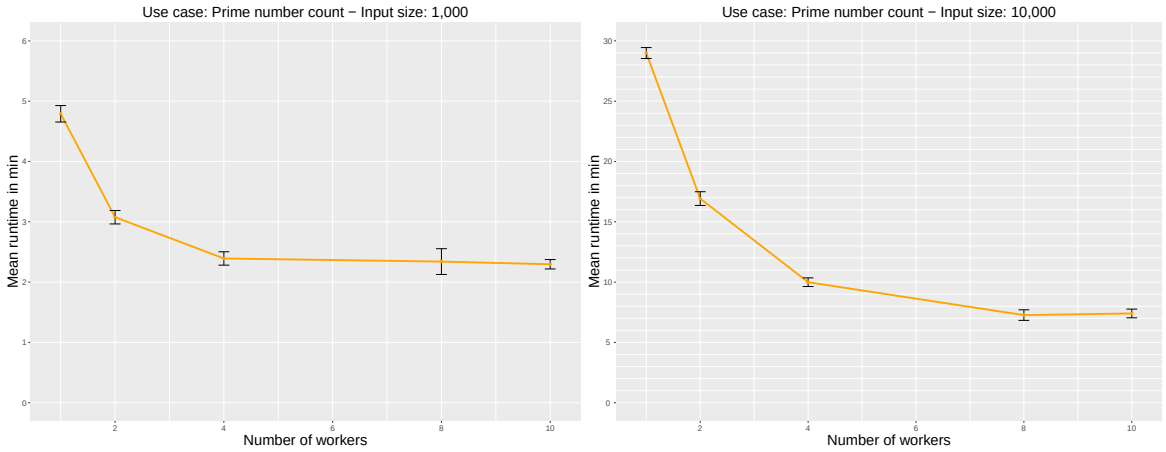
However, many of the speedup values are far below their optimum in general, especially regarding the test configurations with eight or ten workers. For instance, when looking at the first scenario of Table 5.2, the speedup in case of eight workers is 2.05, whereby the optimum would be 8. In terms of ten workers, the speedup is 2.09, which is significantly lower than the optimum of 10. Furthermore, these two values do not differ much, which is an observation that can be applied to most scenarios, so the values for the speedup with a scenario do not reveal significant differences and are therefore rather far from a linear speedup, which would be the ideal case. This indicates that the overall interval is not processed significantly faster if more workers are included. Therefore, the actual scaling behavior with respect to the speedup in strong scaling scenarios does not meet the expectations, since an continuously efficient performance improvement due to an increased number of workers cannot be observed. Explanations for this behavior are subsequently discussed on the basis of the plots that are provided within this section.

Concerning the coefficient of variation, abbreviated with CV in Table 5.2, which enables comparisons among the different test configurations, all values are rather low. Even though the coefficients of variation within each of the scenarios differ slightly more than the ones of the varying input size scenarios as presented Table 5.1 in general, these differences are still not very substantial. This observation can be traced back to the fact that the third scenario, which represents the one with the smallest range, includes relative standard deviations between 2.21 % and 4.38 %, whereas the second scenario constitutes the one with the biggest range by encompassing values between 1.43 % and 8.41 %. Similar to the varying input size scenarios, the strong scaling scenarios do not reveal any patterns with respect to a continuous increase or decrease of the coefficients of variation within or between scenarios.

The measurement results of two strong scenarios are additionally visualized by plots based on Table 5.2, in a similar manner to the varying input size scenarios discussed earlier. Figure 5.2a shows the graph resulting from the scenario which encompasses a total input size of 1,000, whereas the scenario representing a total input size of 10,000 is illustrated in Figure 5.2b. By inspecting these two figures, it becomes clear that the protocol does not always scale as desired in case of the prime number count use case, since the mean total execution time from one test configuration to another does not decrease as expected, especially regarding the test configurations comprising eight or ten workers.

For instance, when increasing the number from four to eight workers, the mean runtime nearly remains the same in the scenario depicted in Figure 5.2a, even though the number of workers is doubled. As for the scenario shown in Figure 5.2b, the average runtime decrease when eight workers are involved as opposed to four, but not as much as desired, which is also reflected by a speedup of 3.99, while the optimum would be 8.

These observations are also strengthened by the above statements about the speedup. In the scenario displayed in Figure 5.2a, the overall execution time on average nearly stagnates if four or more workers are involved. The exact values for the mean runtime can also be seen in Table 5.2, which states that it takes 2.39 minutes on average in case of four workers, 2.34 minutes in case of eight workers and 2.30 minutes in case of ten workers, so there is no significant performance improvement between these three test configurations. Likewise the speedups of these three test configurations do not differ as much as expected and are far below their optimum values as stated above. Regarding the scenario depicted in Figure 5.2b, the average execution time is still decreasing from four to eight workers, but not from eight to ten, rather it actually increases slightly. An increase in execution time as the number of workers is raised is exactly the opposite of the desired scaling behavior.



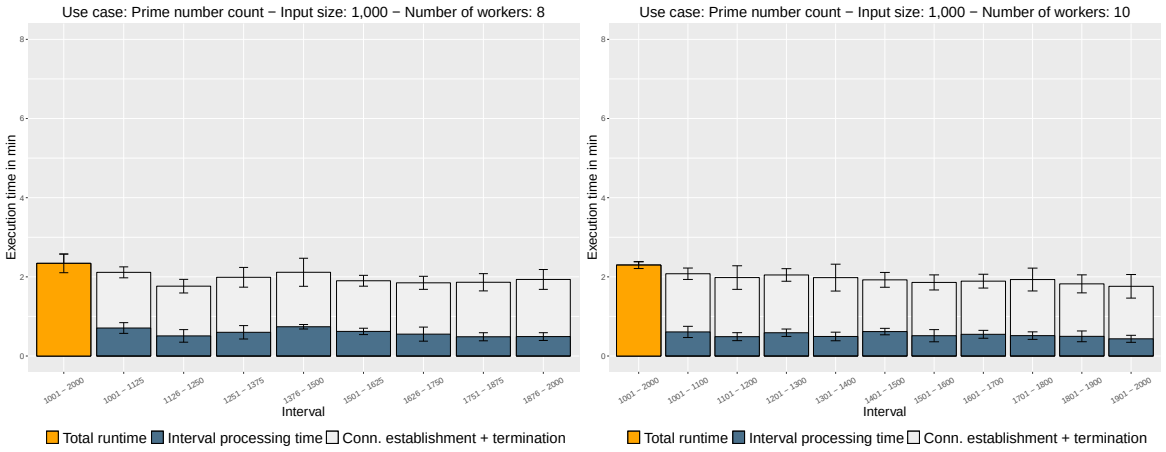
(a) Scenario: 1,000 as total input size

(b) Scenario: 10,000 as total input size

Figure 5.2.: Plots of two different strong scaling scenarios based on the prime number count use case including error bars

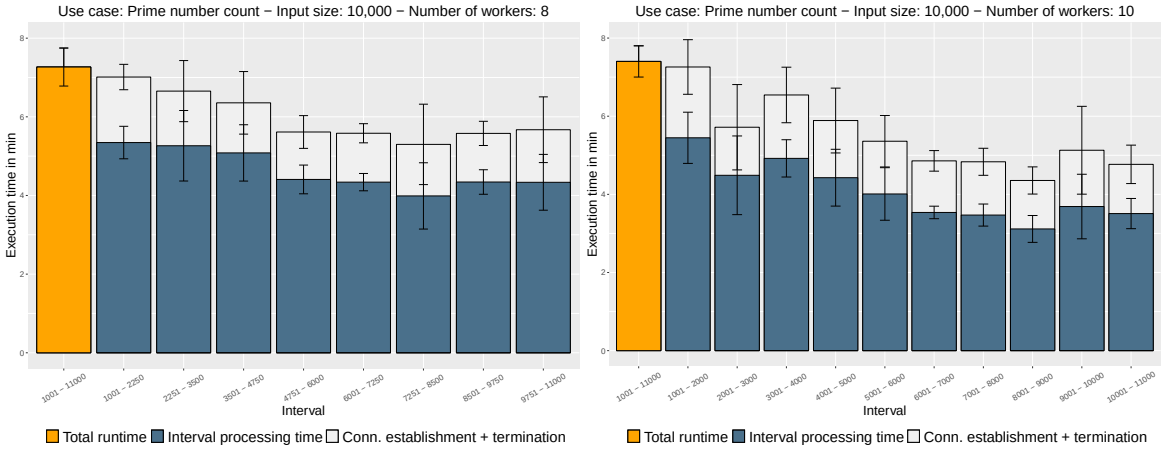
In order to exemplify this behavior, Figures 5.3a, 5.3b, 5.4a and 5.4b demonstrate the differences between selected test configurations in terms of multiple execution time measurements. The mean runtime, as measured by the master node, is indicated by the leftmost bar within each plot and colored in orange/medium gray, while the average execution time required for each worker to process the assigned interval is visualized by the blue/dark gray fraction of the remaining bars, and the time it takes to establish and terminate a connection measured at each worker node is represented by the light gray fraction of the same bars. These two fractions of the remaining bars within each plot together form the total execution time of the worker nodes associated with the individual intervals. Therefore, the plots encompass the mean total runtime of the application, as well as the measured values of each worker node on average. A legend defining the color scheme is also included in the plots.

The first pair of plots encompasses two test configurations with a total input size of 1,000 and eight respectively ten workers, thereby mirroring the last two test configurations of the first scenario in Table 5.2. On the contrary, the second pair of plots outlines the test configurations defining 10,000 as total input size and also eight respectively ten workers, which represent the last two test configurations of the fifth scenario in Table 5.2.



(a) Number of workers: 8 (b) Number of workers: 10

Figure 5.3.: Plots of two different test configurations based on the prime number count use case defining 1,000 as input size including error bars



(a) Number of workers: 8 (b) Number of workers: 10

Figure 5.4.: Plots of two different test configurations based on the prime number count use case defining 10,000 as input size including error bars

When investigating Figure 5.3a in more detail, it becomes clear that the connection establishment and termination part vastly exceeds the computational part, so for each worker it takes significantly more time to start and end the connection with the master than to actually process the associated interval. This behavior is even more intensified in Figure 5.3b, which also points out that the decrease of processing time in case of ten workers as compared to eight workers cannot balance the increased time required for the connection establishment and termination resulting from a larger number of workers. Referring back to Amdahl's law, the whole communication between the peers including connection establishment, termination and data exchange constitutes the serial fraction of the program and since there is no speedup for the serial parts, the total speedup is fundamentally limited by this fraction.

Additionally, the sequential component makes up for a bigger proportion of the total execution time the more workers are involved, since the communication overhead increases with a higher number of workers. Possible reasons for this behavior can be found in Section 5.3. These observations can also be applied to Figure 5.4a and 5.4b, whereby the computational part is not exceeded by the connection handling part in case of 10,000 numbers as total input size in comparison with 1,000 numbers, rather it is significantly higher. But nevertheless, the mean total execution time is slightly increasing instead of decreasing between those two test configurations due to the rising time required for the connection start and end, even though the processing of the intervals takes less time per node on average.

To sum it up, these four plots provide an explanation for the low speedup, especially regarding the test configurations comprising eight and ten workers. With respect to Amdahl's law, the serial part of the program is too predominant and even increases as the number of workers grows. As a result, strong scaling in terms of the prime number count use case can only be realized efficiently on a very small scale, e.g. if the workload is divided among two or four workers, but the more workers are involved, the more the values for the speedup differ from their optimum, mainly caused by the vast protocol overhead.

Weak scaling scenarios As for the mean runtime, corresponding standard deviation and coefficient of variation of each individual test configuration, Table 5.3 outlines all the resulting values in case of the weak scaling scenarios based on the prime number count use case. Weak scaling implies that the number of workers and also the total input size is varied by means of the same factor, so both of these dimensions are scaled in the following scenarios. In this way, the amount of work that needs to be processed by each worker remains the same within the test configurations of each scenario. Ideally, the goal of weak scaling is to keep the time required for the execution of the application on a steady level within a scenario, thus resulting in a constant overall execution time. With regard to these weak scenarios, the growth rate equals two, so the number of workers and the input size is always doubled from one test configuration to another.

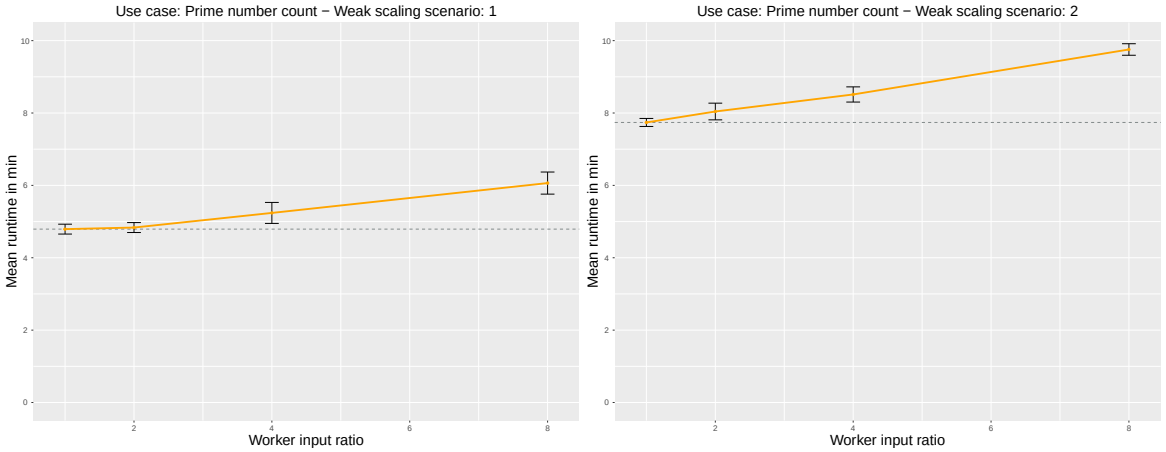
Workers	Total input size	Mean runtime	Stand. deviation	CV
1	1,000 numbers	4.79 min	0.14 min	2.85 %
2	2,000 numbers	4.84 min	0.14 min	2.84 %
4	4,000 numbers	5.24 min	0.29 min	5.54 %
8	8,000 numbers	6.06 min	0.31 min	5.04 %
1	2,000 numbers	7.74 min	0.11 min	1.43 %
2	4,000 numbers	8.04 min	0.23 min	2.87 %
4	8,000 numbers	8.51 min	0.21 min	2.47 %
8	16,000 numbers	9.76 min	0.16 min	1.63 %

Table 5.3.: Weak scaling scenarios based on the prime number count use case including mean runtime, sample standard deviation and coefficient of variation (CV)

When investigating the calculated values presented in Table 5.3, a slight difference in the coefficients of variation can be observed between the two scenarios. While all values for the relative standard deviation are generally low, the second scenario includes three of the four smallest coefficients of variation and therefore comprises values that only range between

1.43 % and 2.87 %. On the contrary, the first scenario encompasses coefficients of variation ranging from 2.84 % to 5.54 %, so the values of this scenario are comparatively higher. Additionally, the CV values differ more within the first scenario as compared to the ones belonging to the second scenario. However, since these differences are not very significant in general, they may simply result from varying environmental influences during the different measurements, as further described in Section 5.3.

Since there are only two weak scaling scenarios, both of them are graphically visualized. The two scenarios are plotted in Figure 5.5a and 5.5b, whereby none of them shows the desired behavior, that is a constant total execution time on average as displayed by the dashed gray line within each graph. Furthermore, as can be seen in Figure 5.5b, the graph of the second scenario drifts even further away from the reference line representing the ideal behavior, in contrast to the first scenario depicted in Figure 5.5a. A possible reason for the increasing total execution might be that the time required for the connection establishment and termination as well as for the data exchange increases with a larger number of workers and a bigger total input size, so the serial fraction of the application does not remain constant. This effect is even intensified in the second scenario as compared to the first one.



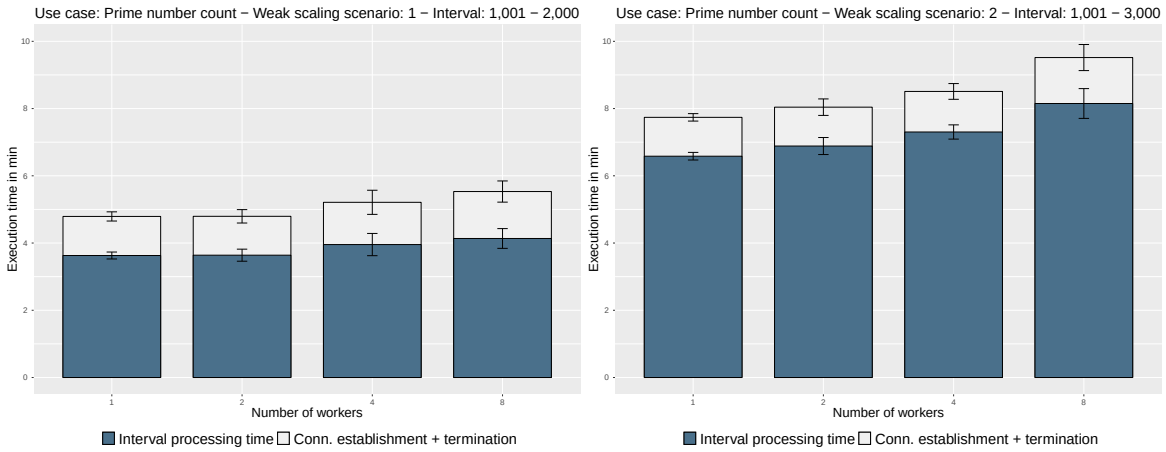
(a) Weak scaling scenario: 1

(b) Weak scaling scenario: 2

Figure 5.5.: Plots of both weak scaling scenarios based on the prime number count use case including error bars

In order to illustrate this behavior in more detail, Figure 5.6a and Figure 5.6b present further diagrams. For this demonstration, the first interval that needs to be processed by a worker within each test configuration of a scenario is chosen. This interval never changes within a given scenario since the workload that needs to be processed by each worker stays constant among the different test configurations as defined by weak scaling, while the number of workers and the total input size are increased by the same factor. Regarding the first weak scaling scenario, the first interval of each test configuration is the one from 1,001 to 2,000, thereby encompassing 1,000 numbers. In case of the second scenario, the interval from 1,001 to 3,000 constitutes the selected one, which contains 2,000 numbers in total. Both of these intervals are computed by one worker node in every single test configuration of each scenario, so the measured values of the time required to process this interval in every test configuration can be compared. This comparison is illustrated in Figure 5.6a for the

first scenario and in Figure 5.6b for the second scenario. Ideally, the processing time of the interval remains constant within a scenario, but when contrasting the mean measured values of the different test configurations as indicated by the blue/dark gray fractions of the bars in Figure 5.6a and 5.6b, a continuous increase in processing time can be observed in both plots. So even though the exact same interval needs to be processed in every test configuration, it takes longer the more workers are involved in total, which goes hand in hand with a bigger overall input size. Since the processing time includes the separate transmission of the calculated primes to the master and since the overall amount of primes grows with a bigger input size, more messages need to be dispatched to the master in general when the input size is increased. At the same time, as the number of workers grows as well, it also takes slightly more time for each worker to establish and terminate the connection to the master, as indicated in the plots by the light gray fraction of each bar. Therefore, the serial fraction of the application increases from one test configuration to another, resulting in a higher overall execution time of the worker program and thus representing an undesired scaling behavior in terms of weak scaling. Section 5.3 provides further explanations for this rise of the serial component.



(a) Scenario: 1 — Interval: 1,001 - 2,000

(b) Scenario: 2 — Interval: 1,001 - 3,000

Figure 5.6.: Plots focusing on two specific intervals of different test configurations from the weak scaling scenarios based on the prime number count use case including error bars

According to Gustafson's law, larger problems are needed for a larger number of workers since they usually scale better[20]. This can be traced back to the fact that the serial component generally becomes less important in case of larger problems and therefore does not dominate anymore in proportion to the parallel fraction of the application. Regarding the prime number use case, this would mean that the amount of work processed by each worker must be increased by heightening the total input size in order to maintain the scaling and to achieve a desired scaling behavior, but this would require additional measurements.

In summary, the actual measured values do not match the ideal case due to the predominance of the serial fraction, which represents an undesired behavior of the protocol in terms of the weak scaling scenarios based on the prime number count use case.

[20] https://www.archer.ac.uk/training/course-material/2016/12/mpi_scaling_manc/Slides/Scaling.pdf

Word and line count use case

In the following, the measurement results of the word and line count use case are outlined and examined with respect to the scalability of ZMTP. The input that corresponds to this use case is represented by a text section which comprises 1,345 words split among 117 lines. Due to the scarce memory capacity of the constrained nodes involved in the measurements, it is not possible to significantly increase the input size of 1,345 words with the current design of the use case. Therefore, the total input size is fixed for all scenarios of the word and line count use case, so meaningful weak scaling scenarios cannot be realized.

The scenarios associated with this use case are divided into two categories. The first category comprises the scenarios where the number of text passages per worker is altered, so the workers always have to process roughly the same text, but depending on the specific test configuration the individual input per worker might be further split into two or more text passages which are processed separately. On the contrary, the second category defines strong scaling scenarios, so the number of workers is increased from one test configuration to another, while the total input size always remains constant. Similar to the first use case, the results are presented in tables composed of the mean runtime per test configuration and the associated sample standard deviation, as well as the coefficient of variation. Additionally, some of the scenarios are visualized through plots.

Varying text allocation scenarios Each of the five blocks in Table 5.4 represents a scenario, where the number of text passages per worker differs between the individual test configurations, while the number of workers is fixed within a scenario, but still varies among different ones. When examining each scenario separately, the values for the coefficients of variation usually do not differ much, except for the fourth use case, which specifies eight workers and whose values range between 9.64 % and 19.31 %. The relative standard deviation of 19.31 % also represents the highest value of all test configurations in the table and is probably caused by one or more outliers.

On the contrary, when the scenarios of Table 5.4 are not only inspected separate from each other, the most apparent finding that can be derived from this table is the notable discrepancy between the coefficients of variation compared among scenarios. In general, the values of the scenarios comprising fewer workers are lower than the ones of the scenarios involving a larger number of workers. For instance, the coefficients of variation range between 1.25 % and 3.10 % within the scenario specifying only one worker, namely the first one, while the last scenario, which is the one with the largest number of workers, includes values between 10.47 % and 14.47 %. So the coefficients of variation generally increase as the number of workers grows. Possible explanations for this interrelation are covered in connection to the plots that belong to these scenarios, whereas general potential causes of the fluctuations in coefficients of variation are identified in Section 5.3.

As derived from Table 5.4, it usually takes slightly more time on average to process the complete text section when the input, that is assigned to a certain worker, is further split into four parts as compared to when it is processed as a whole, whereby the first scenario is the only one not following this pattern. The reason for this behavior is probably the fact that more messages have to be sent between the master and the worker in case of four

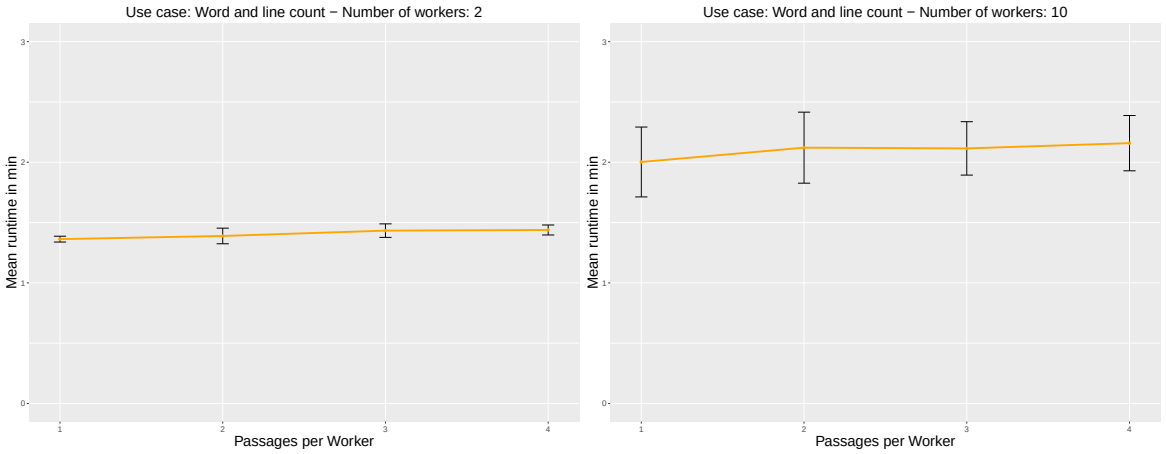
Workers	Text allocation	Mean runtime	Stand. deviation	CV
1	1 pass. per worker	1.48 min	0.02 min	1.25 %
1	2 pass. per worker	1.47 min	0.05 min	3.10 %
1	3 pass. per worker	1.51 min	0.03 min	2.15 %
1	4 pass. per worker	1.46 min	0.03 min	1.96 %
2	1 pass. per worker	1.36 min	0.02 min	1.77 %
2	2 pass. per worker	1.39 min	0.07 min	4.66 %
2	3 pass. per worker	1.43 min	0.06 min	3.91 %
2	4 pass. per worker	1.44 min	0.04 min	2.88 %
4	1 pass. per worker	1.42 min	0.04 min	2.84 %
4	2 pass. per worker	1.55 min	0.07 min	4.47 %
4	3 pass. per worker	1.57 min	0.08 min	4.84 %
4	4 pass. per worker	1.67 min	0.07 min	4.33 %
8	1 pass. per worker	2.07 min	0.28 min	13.72 %
8	2 pass. per worker	2.09 min	0.40 min	19.31 %
8	3 pass. per worker	2.04 min	0.23 min	11.30 %
8	4 pass. per worker	2.15 min	0.21 min	9.64 %
10	1 pass. per worker	2.00 min	0.29 min	14.47 %
10	2 pass. per worker	2.12 min	0.29 min	13.88 %
10	3 pass. per worker	2.12 min	0.22 min	10.47 %
10	4 pass. per worker	2.16 min	0.23 min	10.62 %

Table 5.4.: Varying text allocation scenarios based on the word and line count use case including mean runtime, sample standard deviation and coefficient of variation (CV)

text passages per worker, because the individual text passages are being sent to workers separately and the partial results of each text passage also need to be dispatched to the master individually before the next text passage can be processed by a worker. However, in general, the calculated values of the average runtime do not differ as much within each single scenario. This observation is also reflected in the plots presented in Figure 5.7a and 5.7b.

Figure 5.7a illustrates the second scenario of Table 5.4 including two workers, while Figure 5.7a displays the last scenario of this table, thus involving ten workers. Since both diagrams present the same range of values on their y axis, the mean runtime can be directly compared to each other among the figures. When contrasting these two plots, it becomes clear that the average overall execution time is generally higher in case of ten workers as opposed to two worker, which is examined more precisely in the strong scaling scenarios of this use case.

To sum it up, the mean total execution time does not increase significantly when the input per worker is divided into more than one text passage, which represents a desired behavior of the investigated protocol, but the coefficients of variation notably differ among the scenarios, depending on how many workers are involved, which is investigated in more detail in the following section focusing on strong scaling scenarios.



(a) Scenario: 2 workers

(b) Scenario: 10 workers

Figure 5.7.: Plots of two different varying text passage allocation scenarios based on the word and line count use case including error bars

Strong scaling scenarios In terms of the strong scaling scenarios, the resulting values of the mean runtime, the sample standard deviation and the coefficient of variation are displayed in Table 5.5. Just like the strong scaling scenarios of the prime number count use case, the ones of the word and line count use case additionally include the calculated speedups associated with the different test configurations which are computed by the same formula.

Considering the calculated values in Table 5.5, the assumption that the number of workers influences the coefficients of variation as observed in the varying text allocation scenarios of the word and line count use case is strengthened by these values. As stated above, the coefficient of variation is generally higher in case of a larger number of workers. This observation can also be made in terms of the strong scaling scenarios presented in Table 5.5. When comparing the relative standard deviations within each of the scenarios, the resulting values are usually higher if more workers, namely eight or ten, are involved in a test configuration and lower if only one, two or four workers are participating in the experiment. This indicates that the measured values of the repeated measurements are spread in a wider range of values in case of eight or more workers.

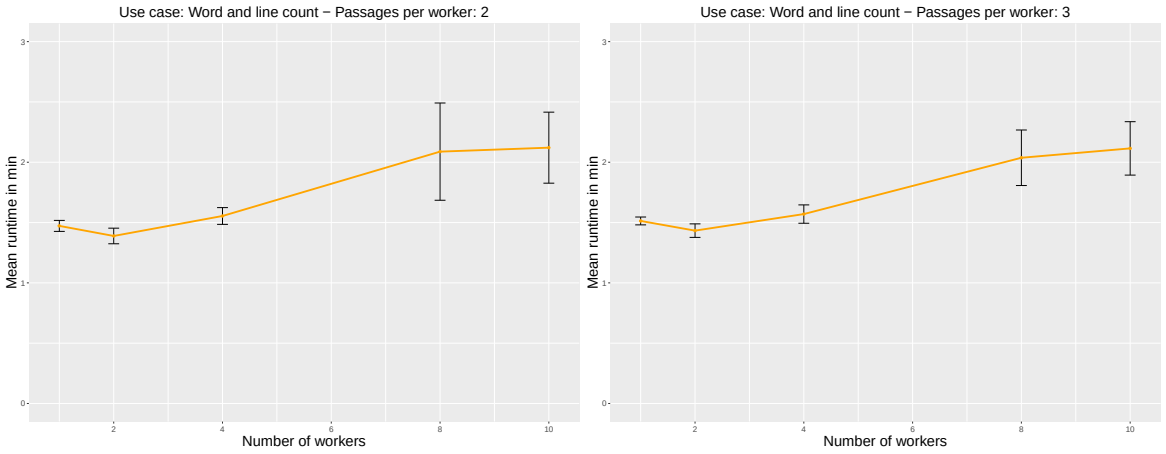
Similar observations can also be made about the average total execution time, which is comparatively higher for test configurations that encompass more workers, so running the application takes longer for a larger number of worker as compared to only a few workers. This means, that the average total runtime increases in general when the total text section is split among more workers, which directly contradicts the goal of strong scaling and thus represents an undesired behavior. But these observations only apply to the test configurations including two or more workers, since the mean runtime decreases when scaling from one to two workers, but from then one, it continuously rises. When looking at the speedup, the calculated values are rather usually decreasing instead of increasing within the scenarios, while they are far below their optimum and many of them reflect a “negative” speedup, that is a value lower than one. Therefore, the protocol apparently does not scale as expected in terms of the strong scaling scenarios of this use case.

Workers	Text allocation	Mean runt.	Speedup	Stand. dev.	CV
1	1 pass. per worker	1.48 min	—	0.02 min	1.25 %
2	1 pass. per worker	1.36 min	1.09	0.02 min	1.77 %
4	1 pass. per worker	1.42 min	1.05	0.04 min	2.84 %
8	1 pass. per worker	2.07 min	0.72	0.28 min	13.72 %
10	1 pass. per worker	2.00 min	0.74	0.29 min	14.47 %
1	2 pass. per worker	1.47 min	—	0.05 min	3.10 %
2	2 pass. per worker	1.39 min	1.06	0.07 min	4.66 %
4	2 pass. per worker	1.55 min	0.95	0.07 min	4.47 %
8	2 pass. per worker	2.09 min	0.71	0.40 min	19.31 %
10	2 pass. per worker	2.12 min	0.69	0.29 min	13.88 %
1	3 pass. per worker	1.51 min	—	0.03 min	2.15 %
2	3 pass. per worker	1.43 min	1.06	0.06 min	3.91 %
4	3 pass. per worker	1.57 min	0.96	0.08 min	4.84 %
8	3 pass. per worker	2.04 min	0.74	0.23 min	11.30 %
10	3 pass. per worker	2.12 min	0.72	0.22 min	10.47 %
1	4 pass. per worker	1.46 min	—	0.03 min	1.96 %
2	4 pass. per worker	1.44 min	1.02	0.04 min	2.88 %
4	4 pass. per worker	1.67 min	0.87	0.07 min	4.33 %
8	4 pass. per worker	2.15 min	0.68	0.21 min	9.64 %
10	4 pass. per worker	2.16 min	0.68	0.23 min	10.62 %

Table 5.5.: Strong scaling scenarios based on the word and line count use case including mean runtime, speedup, sample standard deviation and coefficient of variation (CV)

While the plot of the second scenario is illustrated in Figure 5.8a, Figure 5.8b represents the graph resulting from the third scenario of Table 5.5. These two diagrams exemplify how the average overall execution time slightly decreases from one to two workers, but subsequently rises from then on and even surpasses the mean runtime measured when only one worker is involved in processing the complete text segment. Therefore, the assertions about the mean runtime made above are strengthened by these plots. Furthermore, the observed behavior represents the exact opposite of what is considered the desired behavior, that is processing a fixed workload in less time due to more workers, so in general, the protocol does not efficiently scale for these scenarios. A possible reason for this behavior might be that the overhead outweighs in contrast to the performance improvement achieved by more parallelization, which is further explained in the following.

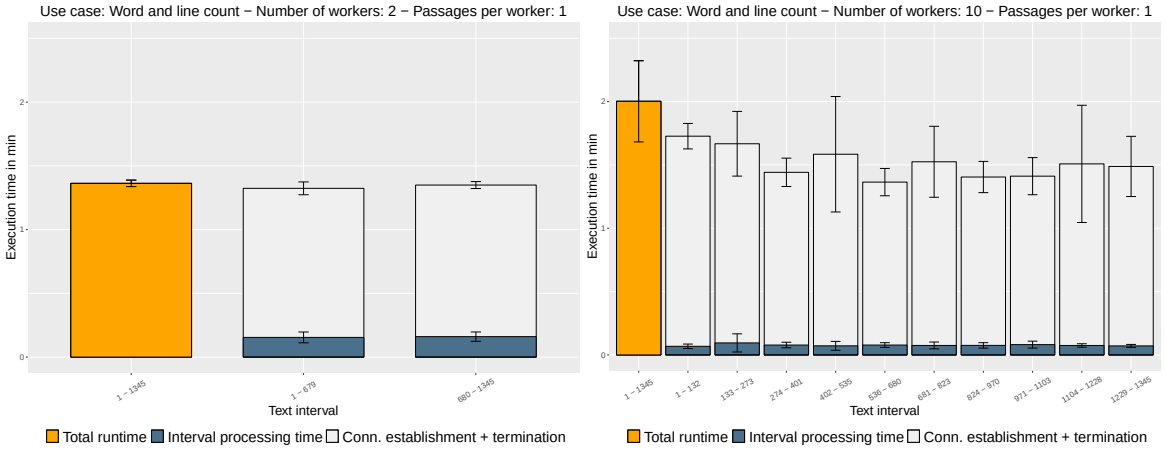
In order to ascertain possible reasons for the observations established above, further plots are required. Figure 5.9a and Figure 5.9b therefore show the plots of two different test configurations including the mean runtime measured by the master node (orange/medium gray bar), as well as the average execution time required to process the interval associated with each single worker (blue/dark gray fraction of the bars) and the time needed for the connection establishment and termination (light gray fraction of the bars), both of which are measured by each worker node separately. Therefore, the color scheme of the bar charts contained in the the prime number count use case can also be applied to these plots, which represent the second respectively the fifth test configuration of the first scenario of Table 5.5.



(a) Scenario: 2 passages per worker

(b) Scenario: 3 passages per worker

Figure 5.8.: Plots of two different strong scaling scenarios based on the word and line count use case including error bars



(a) Number of workers: 2

(b) Number of workers: 10

Figure 5.9.: Plots of two different test configurations defining one passage per worker based on the word and count use case including error bars

What can be clearly seen in both plots illustrated in Figure 5.9a and 5.9b, is that the time required for the connection establishment and termination on average for each worker node surpasses the actual processing time of the individual text passage many times over. This effect is even more dominant in Figure 5.9b where ten workers are involved in the computation as compared to only two workers. These observations indicate that the total input size of this use case is way too small, so the measurements mainly determine the overhead caused by the ZeroMQ Message Transport Protocol.

As already analyzed in the prime number count use case, the communication overhead per worker grows with an increasing number of workers, so the serial fraction of the application significantly rises and causes a higher total execution time. Referring to Amdahl's law,

the sequential part of the application is too significant, whereas the computational part that can be parallelized is not complex enough as a result of the small overall input size. Hence, the performance improvement gained by the division of the total input among more workers is defeated by the rising communication overhead. The domination of the serial component in contrast to the parallel one might be remedied by adding extra complexity to the computational part or by considerably increasing the input size within the word and line count use case as suggested by Gustafson's law[21]. Additionally, this measure is indispensable since mainly the protocol overhead is assessed by the conducted measurements due to the small input size. Referring back to the resource scarcity of the IoT-LAB M3 open nodes, as used for the series of experiments, which restricts a considerable increase of the input size, a bigger overall text section cannot be realized with the current design of this use case. However, streaming data might represent an alternative approach that could possibly remedy the limitation of the input size due to the low memory capacity of the nodes, but implementing this approach and conducting measurements thereof is out of the scope of this thesis.

In summary, the findings clearly contradict the goal of processing a fixed workload in less time due to a larger number of workers, so ZMTP does not meet the expectations regarding the strong scaling scenarios of the word and line count use case, since it exhibits an undesired scaling behavior.

5.2.2. Power consumption results

Besides the execution time measurements, the power consumption of each IoT-LAB node was measured as a second value during the experiments, but the measurement results did not reveal any significant differences between the various test configurations, or even among disparate scenarios, so no solid conclusions about a varying energy footprint of the constrained nodes can be drawn from these measured values. This applies to the values obtained from the master node, as well as the ones acquired for each of the worker nodes. Therefore, the power consumption results are not discussed in more detail in the course of this thesis and are not taken into account in the remaining part of the evaluation.

5.3. Potential causes

While the previous section presents the measurement results and covers findings resulting from investigating the measured values (see Section 5.2), potential causes that might explain the observations and findings described above are identified and analyzed in the following.

In general, the values calculated for the coefficients of variation are rather low, but some discrepancies within or among different scenarios can be observed, so reasons for these fluctuations have to be established. Most of the comparatively high values of the coefficients of variation, which is also being referred to as the relative standard deviation, are mainly

[21] https://www.archer.ac.uk/training/course-material/2016/12/mpi_scaling_manc/Slides/Scaling.pdf

caused by outliers that are not in line with the remaining values of the repeated measurements. As mentioned in Section 5.1, no outliers are eliminated in the course of this thesis, which can result in a wide range of measured values within repetitions of the same test configuration. Concerning the experiments conducted by means of the IoT-LAB, variations in measured values can be caused by environmental influences, such as network latency or radio interference within the testbed, which can fluctuate between measurements, as stated in Section 5.1, since they usually vary over time. Depending on how many other nodes are actively communicating apart from the nodes involved in one's own experiment, the network load can differ and thus the communication between the master node and the worker nodes as part of the measurements might take longer than usual. However, this behavior reflects realistic conditions, instead of representing an artificially constructed, fully controlled setting, so it is considered an acceptable trade-off caused by the test environment.

With respect to the observed scaling behavior derived from the execution time measurements in case of strong scaling, which is examined for both use cases in Section 5.2, Amdahl's law prevails. The overall speedup of an application is limited by the serial component and in terms of the two use cases assessed within this thesis, the serial fraction is too predominant and thus generally impedes efficient scaling with a high speedup in strong scaling scenarios. According to Gustafson's law, larger problems are required in order to maintain scaling, otherwise the predominance of the serial fraction increases as the parallel fraction is split among a larger number of workers, but the results of the weak scaling scenarios of the first use case did not meet the expectations either. This can also be traced back to the fact that the communication overhead of ZMTP during the connection establishment, connection termination and also the actual data exchange phase, rises as the number of workers grows and thus outweighs the performance improvement achieved by more parallelization.

The cause of this undesired scaling behavior can probably be attributed to three different factors, whereby the first one resides in RIOT's threading model. Since the ZMTP port that is used for evaluating the protocol in terms of constrained environments is a protocol port for RIOT OS, and ZMTP requires one thread per connection on the master side, the strategy of the underlying threading model is essential. As previously described in Section 3.2.1, RIOT enables multi-threading by implementing a preemptive, tick-less, fixed priority scheduler inside the kernel, that differs from many other operating systems. Due to the tick-less property of the scheduler, threads are not switched on a continuous basis with periodic timers in order to simulate parallel execution, but instead, context switches are initiated (1) preemptively, through interrupts realized by the Interrupt Service Routine (ISR), (2) voluntarily, by calling the `thread_yield()` or the `thread_sleep()` function, or (3) implicitly, through function calls unblocking a higher prioritized thread[22]. This scheduling strategy was chosen "to minimize occurrences of thread switching"[BHW+12] within the RIOT operating system.

Regarding the fixed priorities of threads, "a higher priority means that the scheduler will run this thread whenever it becomes runnable instead of a thread with a lower priority. In case of equal priorities, the threads are scheduled in a semi-cooperative fashion. That means that unless an interrupt happens, threads with the same priority will only switch due to voluntary or implicit context switches."[22] According to [BGH+18], this scheduling policy is called *run-to-completion*, so when applying this approach to the use case programs of

[22] https://riot-os.org/api/group__core__sched.html

the experiments, all the other connection handling threads of the master program have to wait until the actively running one is either interrupted or switches context voluntarily or implicitly. Therefore, the connection establishment and termination, as well as the actual communication between each worker and the corresponding connection handling thread of the master, is carried out sequentially without imitating concurrent execution. As a result, the more workers are involved in a certain test configuration, the longer it takes to establish and terminate all the connections and to realize data exchange between the peers, which is why the serial fraction of the application grows as the number of workers increases.

Additionally, due to the protocol specifications of ZMTP and the underlying GNRC TCP implementation it is based on, the general communication overhead is significantly high and the delay of messages is rather long. Referring back to Section 3.2.4, the default configuration of GNRC TCP, which defines a receive buffer and window size equal to the MSS of 1,220 bytes, leads to a limited amount of TCP packets that can be transmitted at the same time. Hence, RIOT's TCP implementation is characterized by a low throughput, so only a single MSS sized packet can be transferred at a time and needs to be acknowledged before the next packet can be dispatched. According to [Bru16], this represents an acceptable trade-off to the detriment of data throughput, since memory saving constitutes the most important design goal of the GNRC TCP implementation.

Furthermore, the ZeroMQ Message Transport Protocol generally adds additional delay as a result of the vast protocol overhead it produces. As mentioned in Section 3.3.2, each field of a ZMTP message frame, namely the flags, the size and the body field, is sent as a separate TCP packet, so for every ZMTP message that needs to be dispatched after the connection is established, three TCP packets are required, resulting in high network traffic, while only one packet can be dispatched at a time. All of these segments need to be acknowledged as well, which delays the transmission of packets during the actual communication even more. Regarding the connection establishment, it takes comparatively long to set up a connection between two ZMTP peers because of the numerous TCP segments that need to be exchanged during the greeting and handshake before data can be sent, including the signature, version number and security mechanism used, and also as a consequence of the low throughput imposed by the GNRC TCP implementation.

In summary, all of the described factors are collectively responsible for the undesired scaling behavior of ZMTP in terms of the two use cases examined within this thesis. Since the delay, caused by the low throughput of GNRC TCP, is exacerbated by the communication overhead of ZMTP, the serial fractions of the applications are too predominant. These fractions do not remain constant, additionally by reason of RIOT's scheduling strategy, which restricts the scalability of the protocol even further. A larger number of workers generally causes a bigger overhead thus raising the serial fraction because establishing and terminating all the connections with the workers takes more time as compared to test configurations involving less workers. This behavior applies to the strong as well as the weak scaling scenarios, since both types of scaling include a varying number of workers. Within the prime number use case, each prime number calculated by a worker is sent back to the master, so a larger input size in case of weak scaling consequently leads to more communication overhead in the data exchange phase, because more messages have to be delivered and each additional message is split into three sequentially sent packets, whereby the serial fraction is increased even further. Therefore, a performance improvement cannot be efficiently realized.

5.4. Consequent scaling conclusions

The last section of this chapter sums up the observations established in Section 5.2 and presents conclusions that can be drawn from these results in terms of the scalability of ZMTP in constrained environments, thereby completing the evaluation of the measurements.

As a first step, observations and implications thereof that fit more than one kind of scaling scenarios or even more than one use case have to be identified. In general, the ZMTP port for RIOT tends to reveal an undesired overall scaling behavior, especially regarding the strong scaling scenarios of both use cases and also the weak scaling scenarios of the prime number count use case. However, the results of the measurements conducted for the word and line count use case exhibit that this use case almost solely measures the communication overhead and therefore does not scale with respect to strong scaling. But still, the measured values show that the overhead during the connection establishment, the actual communication and the connection termination increases on average when the number of workers is raised.

The prime number count use case, on the contrary, does not efficiently scale as well, neither regarding weak scaling nor strong scaling, due to the vast communication overhead and the consequent predominance of the serial fraction of the application. Hence, this overhead constitutes a performance bottleneck and thus significantly limits an overall improvement in performance as explained in Section 5.3. Although a speedup can be observed within the strong scaling scenarios, it is not high enough for many of the test configurations and nearly stagnates at some point in most cases, which represents an undesired scaling behavior and indicates that the decrease of processing time due to more parallelization cannot balance or even surpass the growing communication overhead. Usually, it is easier to achieve satisfactory results for weak scaling[23], but the mean measured values of these scenarios do not meet the expectations likewise. Referring back to Gustafson's law, this behavior can possibly be remedied when a very large input size is used, as mentioned during the analysis of the strong scaling scenarios of each use cases, but focusing on the current measurement results, it can be concluded that the investigated protocol port generally reveals a rather weak scalability with respect to the prime number count use case.

Nevertheless, the scenarios of both use cases represent scaling on a rather small scale, conducting measurements with ten workers at most, but including more test configurations or scenarios would go beyond the scope of this thesis. Therefore, the conclusions above can only be definitely applied to the examined scaling scenarios, but the measurement results indicate that the serial overhead produced by the protocol is generally too high and increases too rapidly, so with regard to these use cases, scaling can probably not be maintained efficiently on a larger scale either.

In summary, even though ZMTP represents a promising approach to realize distributed computing in constrained networks, the measurement results and the consequent evaluation thereof indicate that the RIOT port of this protocol is generally not suitable for this purpose because of an insufficient performance improvement in terms of both use cases.

[23] https://www.archer.ac.uk/training/course-material/2016/12/mpi_scaling_manc/Slides/Scaling.pdf

6. General conclusion and outlook

The final chapter sums up the results of this thesis, includes a general conclusion derived from the elaborated findings and gives an outlook concerning recommendations for future work on the topic.

Within this thesis, the ZeroMQ Message Transport Protocol was introduced as a way of realizing distributed computing on constrained nodes by using a port of the protocol tailored to the RIOT operating system. In general, the primary goal of the thesis was to ascertain the scalability of ZMTP in constrained environments on the basis of two different use cases, which constitute representative distributed computing applications. In order to achieve the intended aim, an experimental approach was chosen as scientific methodology. Therefore, a series of experiments was conducted by means of the use cases developed particularly for the purpose of this thesis. During the experiments, runtime measurements were performed, whereby the execution time represented the primary measurand, while the power consumption was monitored as a secondary measurand. The series of experiments covered strong and weak scaling scenarios by executing numerous test configurations that differed in terms of their total input size or the number of workers they specified. All experiments were carried out on the IoT-LAB testbed and for each test configuration, five temporarily consecutive runs were executed to provide multiple values obtained from repeated measurements.

Since the measurement results of the power consumption monitoring did not reveal any significant differences, neither among various nodes involved in an experiment nor between the measured values for distinct test configurations, these results were omitted in the subsequent steps of the analysis and evaluation. Hence, only the measured values for the execution time were processed in order to better visualize them through tables, including the calculated values of the standard deviations, and to plot them in graphs. This procedure allowed for an easier analysis of the measurement results and additionally eased the traceability of the observations derived from the measured values. The elaborated findings of the different scenarios were then aggregated and evaluated, so solid conclusions about the scalability of the ZMTP port could consequently be drawn with respect to the two use cases that were tested during the experiments.

The evaluation revealed that a vast overhead caused by the protocol limits the performance improvements as intended by the parallelization of the computation through distributing the workload among a number of workers. This overhead, which is present during the connection establishment and termination, as well as during the actual communication, was therefore identified as a performance bottleneck of the protocol port. As a result, the investigation of the measured values exhibited a generally rather weak scalability of ZMTP.

However, it cannot be automatically concluded that the two considered use cases do not scale at all as a consequence, even in the event of minor modifications regarding their design as stated in Section 5.2. In terms of the prime number count use case, such changes can

comprise the definition of a bigger total input size, whereby the workload per node would be increased since the overall interval, that needs to be processed by the workers, is expanded. Furthermore, a small restructuration of the workflow could be realized, which should reduce the amount of communication between the master and the worker nodes during the actual computation by determining the number of primes of the assigned number interval on the worker side and then sending only this single result back to the master by the time the interval is completely processed, instead of dispatching each detected prime number separately.

Concerning the word and line count use case, the total input size cannot be adjusted as easily due to the memory scarcity of the IoT-LAB M3 open nodes, so an alternative approach to providing the input for this application, such as streaming data, would have to be implemented, as mentioned previously. Compared to the first use case, these changes would require more effort. All these modifications might lead to an improved scaling behavior that approaches the desired one despite the vast communication overhead. Especially the prime number count use case seems to be promising, since this application already showed better results than the word and line count use case within the current measurements. But up until now, these are just assumptions which have to be checked by conducting further experiments and corresponding runtime measurements as part of future work on this topic.

Regarding the general conformance of the observed scaling behavior with other use cases, that are distinct from the ones examined within this thesis, a profound and resilient generalization predication is hard to deduce. One the one hand, the poor scalability of ZMTP in terms of strong and weak scaling that could be derived from the measurement results indicates that this specific protocol is generally not suitable for the purpose of realizing distributed computing in constrained environments on account of the overhead it causes, so the usage of ZMTP does not constitute a universal solution which can be efficiently applied to any use case with respect to its scaling behavior. On the other hand, the findings and conclusions about the scalability of the protocol are only confirmed for the corresponding use cases, since the measurements could only be performed for those two applications due to the limited time. Therefore, disparate applications might exhibit scaling behaviors that differ from the ones observed on the basis of the current use cases, but the actual behavior cannot be accurately predicted.

In order to provide more extensive insights into the scaling behavior of ZMTP regarding other use cases, further experiments including runtime measurements are recommended as future work on this topic, besides the additional experiments suggested above. The results of such measurements might then either support or dismiss the assertions made about the scalability of ZMTP, depending on the specific characteristics of the use case that is examined. Nevertheless, it is assumed that only some particular applications can substantially benefit from the usage of ZMTP with regard to its scalability, since the fundamental aspects of the protocol port that were identified as potential causes of the weak scaling behavior are assumed to have negative impacts on other use cases as well.

In summary, the goal of this thesis, that is assessing and evaluating the scalability of ZMTP in constrained environments, was accomplished on the basis of two representative use cases, but even though the results pointed out to a relatively weak scalability of the protocol port, distinct applications based on ZMTP should be considered separately in order to determine their concrete scaling behavior.

List of Figures

3.1. Comparison between the IoT and TCP/IP protocol stack derived from [LB16] 11
3.2. Overview of RIOT's structural elements premised on [BGH+18] 16
3.3. The GNRC networking architecture based on [LKH+18] 17
3.4. Sequence diagram of a TCP 3-Way-Handshake for connection establishment . 19
3.5. TCP header format based upon [Bru16] 20
3.6. The structure of a ZMTP short frame containing two octets of data 25
3.7. Dependency graph of ZMTP source and header files 27

4.1. Methodology of the experimental approach 30
4.2. Master-worker model 33
4.3. Logical topology of the nodes 34
4.4. Partial workflow of the prime number count use case 36
4.5. Partial workflow of the word and line count use case 38
4.6. An IoT-LAB node operating an M3 open node 40

5.1. Plots of two different varying input size scenarios based on the prime number count use case including error bars 50
5.2. Plots of two different strong scaling scenarios based on the prime number count use case including error bars 54
5.3. Plots of two different test configurations based on the prime number count use case defining 1,000 as input size including error bars 55
5.4. Plots of two different test configurations based on the prime number count use case defining 10,000 as input size including error bars 55
5.5. Plots of both weak scaling scenarios based on the prime number count use case including error bars 57
5.6. Plots focusing on two specific intervals of different test configurations from the weak scaling scenarios based on the prime number count use case including error bars . . 58
5.7. Plots of two different varying text passage allocation scenarios based on the word and line count use case including error bars 61
5.8. Plots of two different strong scaling scenarios based on the word and line count use case including error bars 63
5.9. Plots of two different test configurations defining one passage per worker based on the word and count use case including error bars 63

List of Tables

3.1. Classification of constrained IoT devices as defined in RFC 7228 [BEK14] . . 10
3.2. Comparison between Contiki, TinyOS, Linux and RIOT OS based on [BHG+13] 15
3.3. Valid socket type combinations derived from [Hin14a] 22

4.1. Partial problem space definition by means of dimensions 31
4.2. Scenarios and associated test configurations for the prime number count use case 42
4.3. Scenarios and associated test configurations for the word and line count use case 43

5.1. Varying input size scenarios based on the prime number count use case including mean runtime, sample standard deviation and coefficient of variation (CV) 49
5.2. Strong scaling scenarios based on the prime number count use case including mean runtime, speedup, sample standard deviation and coefficient of variation (CV) 52
5.3. Weak scaling scenarios based on the prime number count use case including mean runtime, sample standard deviation and coefficient of variation (CV) . 56
5.4. Varying text allocation scenarios based on the word and line count use case including mean runtime, sample standard deviation and coefficient of variation (CV) 60
5.5. Strong scaling scenarios based on the word and line count use case including mean runtime, speedup, sample standard deviation and coefficient of variation (CV) 62

List of Listings

3.1. ABNF grammar of a ZMTP connection establishment as defined in [Hin14a] . 24
3.2. ABNF grammar of ZMTP traffic as specified in [Hin14a] 25
3.3. ABNF grammar of the NULL security mechanism as specified in [Hin14a] . . 26

4.1. Function calls needed for the execution time measurements 32
4.2. Integration of execution time measurements in the use case programs 35

Bibliography

[Abd10] Hervé Abdi. Coefficient of variation. In *Encyclopedia of Research Design*, pages 170–171. SAGE Publications, Inc., 2010.

[ABF+15] 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 *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 459–464, Dec 2015.

[AH14] M. Aazam and E. Huh. Fog computing and smart gateway based communication for cloud of things. In *2014 International Conference on Future Internet of Things and Cloud*, pages 464–470, Aug 2014.

[AKAH14] M. Aazam, I. Khan, A. A. Alsaffar, and E. Huh. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences Technology (IBCAST) Islamabad, Pakistan, 14th - 18th January, 2014*, pages 414–419, Jan 2014.

[APP13] D. Alessandrelli, M. Petraccay, and P. Pagano. T-res: Enabling reconfigurable in-network processing in iot-based wsns. In *2013 IEEE International Conference on Distributed Computing in Sensor Systems*, pages 337–344, May 2013.

[BA96] J Martin Bland and Douglas G Altman. Statistics notes: Measurement error. *BMJ*, 312(7047):1654, 1996.

[BdDPP14] A. Botta, W. de Donato, V. Persico, and A. Pescapé. On the integration of cloud computing and internet of things. In *2014 International Conference on Future Internet of Things and Cloud*, pages 23–30, Aug 2014.

[BdDPP16] A. Botta, W. de Donato, V. Persico, and A. Pescapé. Integration of cloud computing and internet of things: A survey. *Future Generation Computer Systems*, 56:684 – 700, 2016.

[BEK14] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, RFC Editor, May 2014.

[BGH+18] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. Riot: an open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, pages 1–1, 2018.

[BHG+13] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. Riot os: Towards an os for the internet of things. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 79–80, April 2013.

[BHL+12] Ilja N. Bronstein, Juraj Hromkovic, Bernd Luderer, Hans-Rudolf Schwarz, Jochen Blath, Alexander Schied, Stephan Dempe, Gert Wanka, and Siegfried Gottwald. *Taschenbuch der Mathematik*. Springer, 1st edition edition, 2012.

[BHW+12] Emmanuel Baccelli, Oliver Hahm, Matthias Wählisch, Mesut Günes, and Thomas Schmidt. RIOT: One OS to Rule Them All in the IoT. Research Report RR-8176, INRIA, December 2012.

[Bru16] Simon Brummer. Concept and implementation of tcp for the riot operating system and evaluation of common tcp-extensions for the internet of things, April 2016.

[DBH15] S. K. Datta, C. Bonnet, and J. Haerri. Fog computing architecture to enable consumer centric internet of things services. In *2015 International Symposium on Consumer Electronics (ISCE)*, pages 1–2, June 2015.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications - Big Data, Scalable Analytics, and Beyond.

[HBPT16] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating systems for low-end devices in the internet of things: A survey. *IEEE Internet of Things Journal*, 3(5):720–734, Oct 2016.

[Hin13a] Pieter Hintjens. *ZeroMQ*. O'Reilly Media, 1st edition edition, March 2013.

[Hin13b] Pieter Hintjens. Zeromq exclusive pair. Zeromq rfc, iMatix Corporation, 2013.

[Hin13c] Pieter Hintjens. Zeromq pipeline. Zeromq rfc, iMatix Corporation, 2013.

[Hin13d] Pieter Hintjens. Zeromq request-reply. Zeromq rfc, iMatix Corporation, 2013.

[Hin13e] Pieter Hintjens. Zmtp curve. Zeromq rfc, iMatix Corporation, 2013.

[Hin13f] Pieter Hintjens. Zmtp plain. Zeromq rfc, iMatix Corporation, 2013.

[Hin14a] Pieter Hintjens. Zeromq message transport protocol. Zeromq rfc, iMatix Corporation & Contributors, 2014.

[Hin14b] Pieter Hintjens. Zeromq publish-subscribe. Zeromq rfc, iMatix Corporation, 2014.

[Iwa16] H. Iwai. End of the scaling theory and moore's law. In *2016 16th International Workshop on Junction Technology (IWJT)*, pages 1–4, May 2016.

[KMS07] N. Kushalnagar, G. Montenegro, and C. Schumacher. Ipv6 over low-power wireless personal area networks (6lowpans): Overview, assumptions, problem statement, and goals. RFC 4919, RFC Editor, August 2007.

[LB16] Huichen Lin and Neil W. Bergmann. Iot privacy and security challenges for smart home environments. *Information*, 7(3), 2016.

[Len16] Martine Lenders. Analysis and comparison of embedded network stacks - design and evaluation of the gnrc network stack. Master thesis, Freie Universität Berlin, April 2016.

[LKH+18] Martine Lenders, Peter Kietzmann, Oliver Hahm, Hauke Petersen, Cenk Gündogan, Emmanuel Baccelli, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. Connecting the world of embedded mobiles: The RIOT approach to ubiquitous networking for the internet of things. *CoRR*, abs/1801.02833, 2018.

[LL15] In Lee and Kyoochun Lee. The internet of things (iot): Applications, investments, and challenges for enterprises". *Business Horizons*, 58(4):431 – 440, 2015.

[MKB18] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In Beniamino Di Martino, Kuan-Ching Li, Laurence T. Yang, and Antonio Esposito, editors, *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*, pages 103–130. Springer Singapore, 2018.

[MKHC07] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of ipv6 packets over ieee 802.15.4 networks. RFC 4944, RFC Editor, September 2007.

[MW16] M. Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530:144–147, 2016.

[PAV+13] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler. Standardized protocol stack for the internet of (important) things. *IEEE Communications Surveys Tutorials*, 15(3):1389–1406, Third 2013.

[PMN+18] E. Di Pascale, I. Macaluso, A. Nag, M. Kelly, and L. Doyle. The network as a computer: A framework for distributed computing over iot mesh networks. *IEEE Internet of Things Journal*, 5(3):2107–2119, June 2018.

[Pos81] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981.

[PS17] P. V. Paul and R. Saraswathi. The internet of things - a comprehensive survey. In *2017 International Conference on Computation of Power, Energy Information and Communication (ICCPEIC)*, pages 421–426, March 2017.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014.

[SM16] Subhadeep Sarkar and Sudip Misra. Theoretical modelling of fog computing: a green computing paradigm to support iot applications. *IET Networks*, 5:23–29(6), March 2016.

[SPKS12] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine*, 50(12):144–149, December 2012.

[SYY+13] Z. Sheng, S. Yang, Y. Yu, A. V. Vasilakos, J. A. Mccann, and K. K. Leung. A survey on the ietf protocol suite for the internet of things: standards, challenges, and opportunities. *IEEE Wireless Communications*, 20(6):91–98, December 2013.

[WLMQ16] Qian Wang, B. Lee, N. Murray, and Y. Qiao. Cs-man: Computation service management for iot in-network processing. In *2016 27th Irish Signals and Systems Conference (ISSC)*, pages 1–6, June 2016.

[WTB+12] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. Rpl: Ipv6 routing protocol for low-power and lossy networks. RFC 6550, RFC Editor, March 2012.

[YAH+17] I. Yaqoob, E. Ahmed, I. A. T. Hashem, A. I. A. Ahmed, A. Gani, M. Imran, and M. Guizani. Internet of things architecture: Recent advances, taxonomy, requirements, and open challenges. *IEEE Wireless Communications*, 24(3):10–16, June 2017.

[ZLH+13] Jiehan Zhou, T. Leppanen, E. Harjula, M. Ylianttila, T. Ojala, Chen Yu, Hai Jin, and L. T. Yang. Cloudthings: A common architecture for integrating the internet of things with cloud computing. In *Proceedings of the 2013 IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 651–657, June 2013.

Appendix

A. Plots of the remaining scenarios

The following plots represent the remaining scenarios of the two use cases, that are not graphically visualized within Section 5.2 of Chapter 5. As for the prime number use case, three of the varying input size scenarios and three of the strong scaling scenarios are illustrated below. Since there are only two weak scaling scenarios for this use case, both of them are already included in Chapter 5. In terms of the word and line count use case, three of the varying text allocation scenarios and two of the strong scaling scenarios are additionally presented in the following.

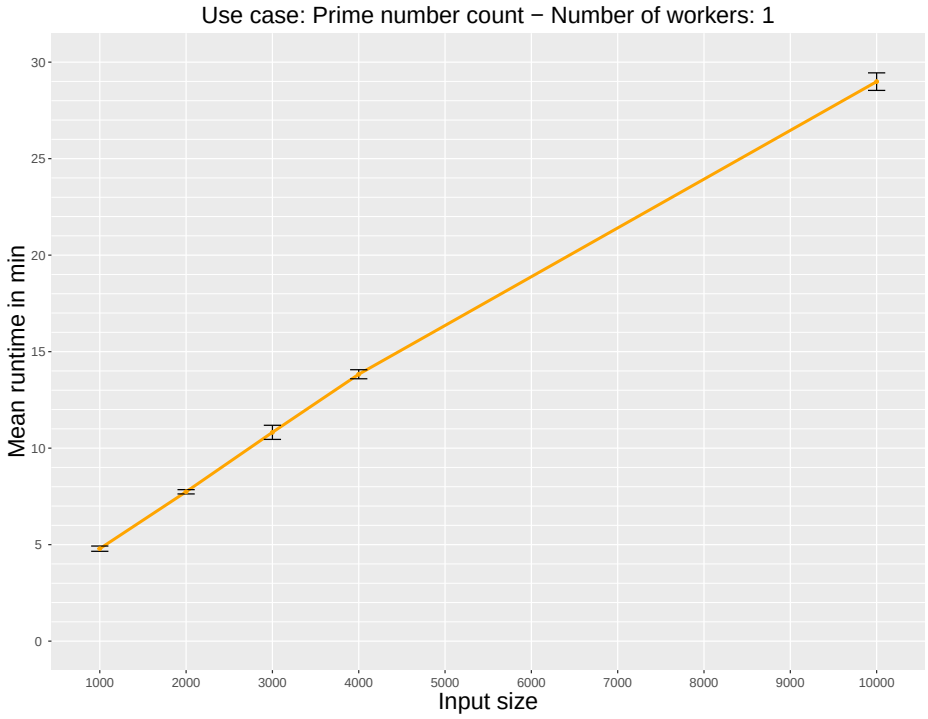


Figure 1.: Varying input size scenario no. one involving 1 worker

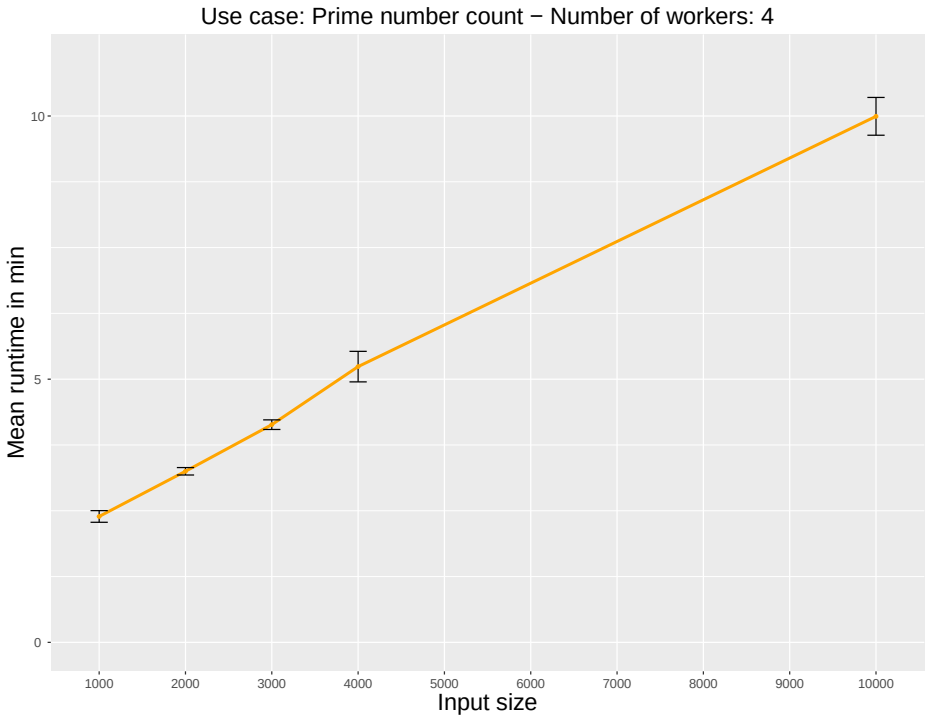


Figure 2.: Varying input size scenario no. three involving 4 workers

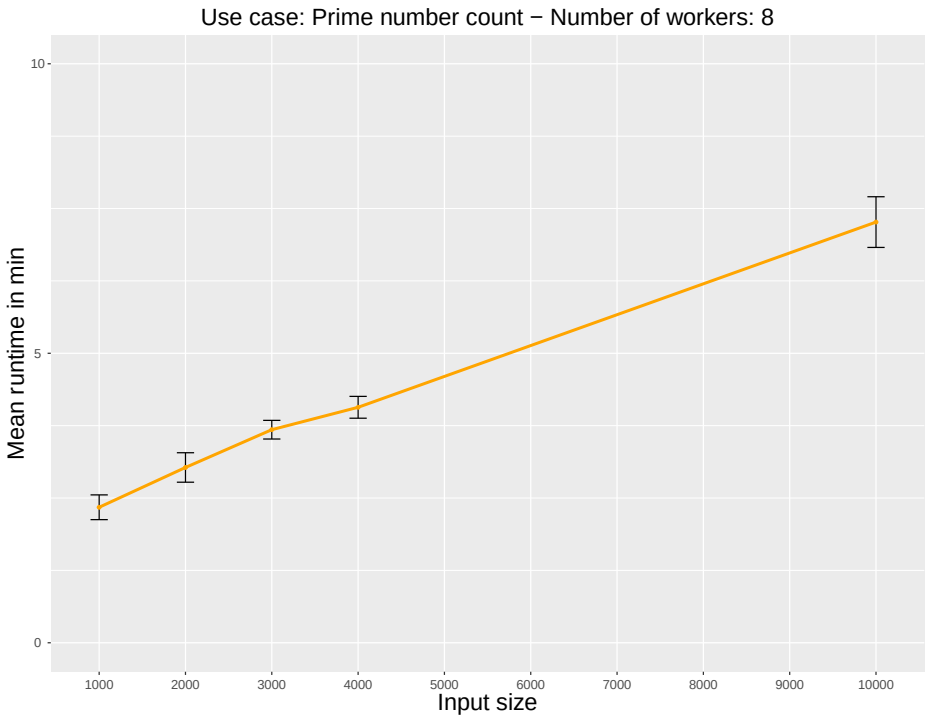


Figure 3.: Varying input size scenario no. four involving 8 workers

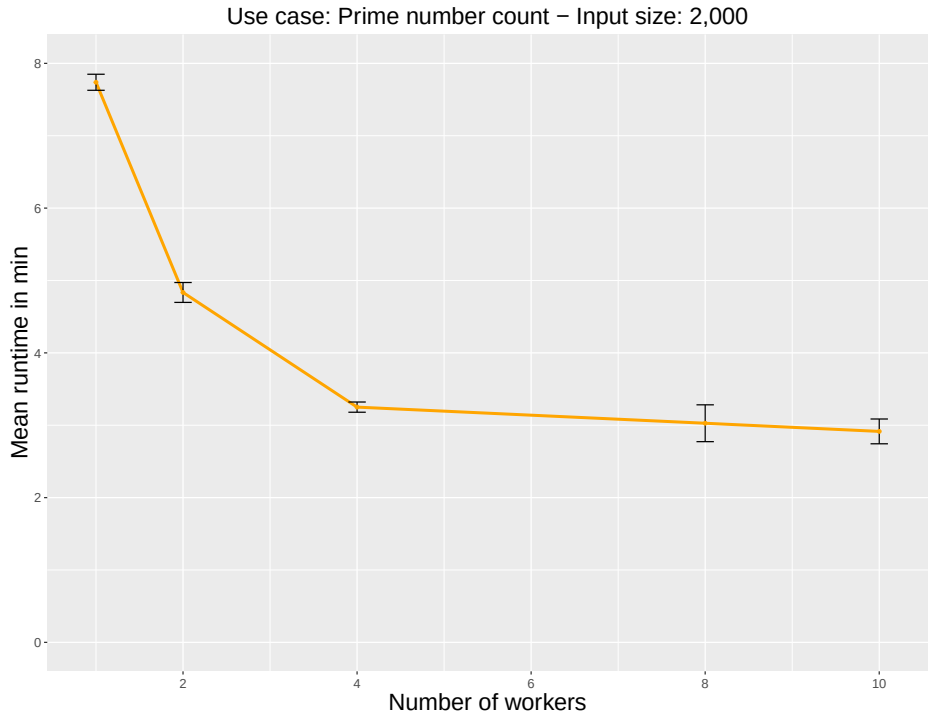


Figure 4.: Strong scaling scenario no. two defining 2,000 as total input size

Figure 5.: Strong scaling scenario no. three defining 3,000 as total input size

Figure 6.: Strong scaling scenario no. four defining 4,000 as total input size

Figure 7.: Varying text allocation scenario no. one involving 1 worker

Figure 8.: Varying text allocation scenario no. three involving 4 workers

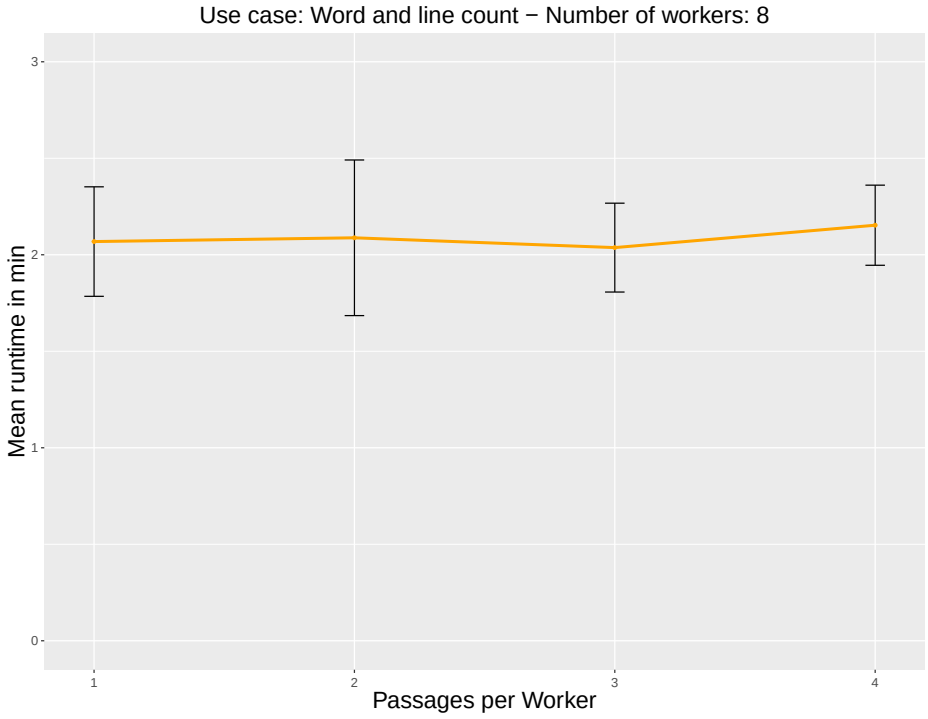


Figure 9.: Varying text allocation scenario no. four involving 8 workers

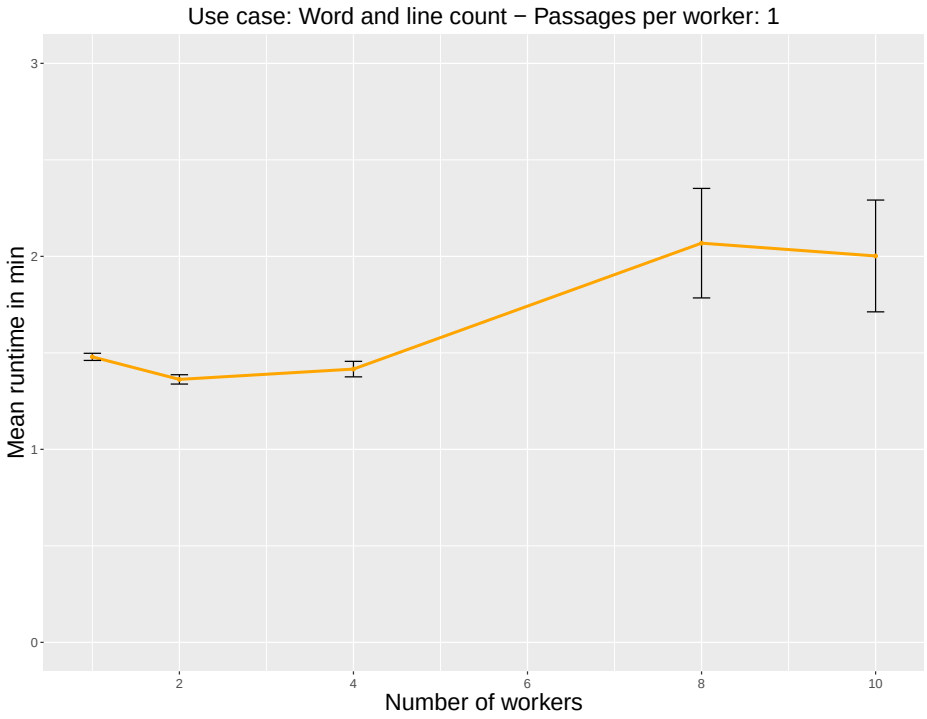


Figure 10.: Strong scaling scenario no. one defining 1 text passage per worker

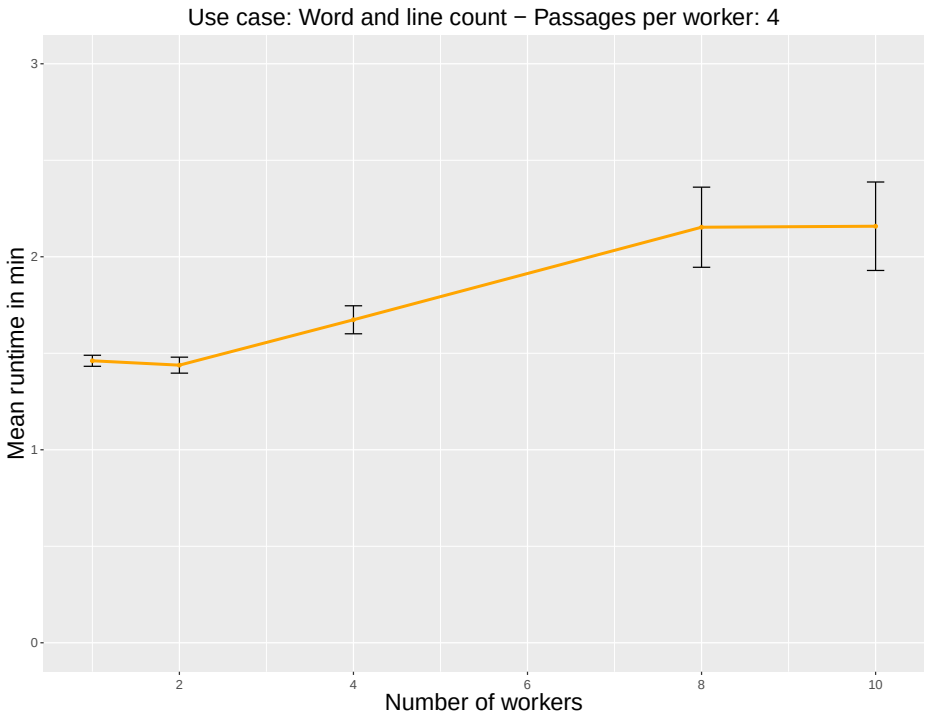


Figure 11.: Strong scaling scenario no. four defining 4 text passages per worker

B. Excerpt of raw measurement data

In order to give a short overview of the raw data that was measured for the execution time of the nodes during the experiments, an excerpt of one of the resulting files is shown in the table below. Compared to the original file, the measured values have already been converted from microseconds to minutes some other measured values were omitted in this table.

Series	Workers	Worker interval	Worker runtime	Master runtime
1.1	1	1,001 - 11,000	29.543578 min	29.543214 min
1.2	1	1,001 - 11,000	28.634883 min	28.634486 min
1.3	1	1,001 - 11,000	28.431420 min	28.431320 min
1.4	1	1,001 - 11,000	29.083334 min	29.083436 min
1.5	1	1,001 - 11,000	29.258953 min	29.259012 min
2.1	2	1,001 - 6,000	16.693261 min	16.695216 min
2.1	2	6,001 - 11,000	14.287963 min	16.695216 min
2.2	2	1,001 - 6,000	17.999446 min	17.998990 min
2.2	2	6,001 - 11,000	13.928416 min	17.998990 min
2.3	2	1,001 - 6,000	16.563307 min	16.564612 min
2.3	2	6,001 - 11,000	13.742645 min	16.564612 min
2.4	2	1,001 - 6,000	16.623095 min	16.623026 min
2.4	2	6,001 - 11,000	14.418386 min	16.623026 min
2.5	2	1,001 - 6,000	16.745418 min	16.744842 min
2.5	2	6,001 - 11,000	13.297820 min	16.744842 min
3.1	4	1,001 - 3,500	9.834560 min	9.834552 min
3.1	4	3,501 - 6,000	9.384086 min	9.834552 min
3.1	4	6,001 - 8,500	8.954884 min	9.834552 min
3.1	4	8,501 - 11,000	8.184703 min	9.834552 min
3.2	4	1,001 - 3,500	10.013794 min	10.013444 min
3.2	4	3,501 - 6,000	8.942943 min	10.013444 min
3.2	4	6,001 - 8,500	9.330387 min	10.013444 min
3.2	4	8,501 - 11,000	8.119858 min	10.013444 min
3.3	4	1,001 - 3,500	9.437159 min	9.450286 min
3.3	4	3,501 - 6,000	9.449376 min	9.450286 min
3.3	4	6,001 - 8,500	8.771658 min	9.450286 min
3.3	4	8,501 - 11,000	8.325073 min	9.450286 min
3.4	4	1,001 - 3,500	10.505340 min	10.505493 min
3.4	4	3,501 - 6,000	9.671213 min	10.505493 min
3.4	4	6,001 - 8,500	8.831652 min	10.505493 min
3.4	4	8,501 - 11,000	8.169167 min	10.505493 min
3.5	4	1,001 - 3,500	10.165001 min	10.164781 min
3.5	4	3,501 - 6,000	9.800943 min	10.164781 min
3.5	4	6,001 - 8,500	7.922632 min	10.164781 min
3.5	4	8,501 - 11,000	8.579721 min	10.164781 min

Table 1.: Excerpt of the file containing the measured values for the last strong scaling scenario of the prime number count use case defining 10,000 numbers as total input size