CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Automatic Configuration Sharing in Peer-to-Peer IoT Networks

Implementation and Evaluation of Automatic Configuration Sharing as a Linux Kernel Module

Master's thesis in Computer science and engineering

William Nilsson
Erik Wetter

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Automatic Configuration Sharing in Peer-to-Peer IoT Networks

Implementation and Evaluation of Automatic Configuration Sharing as a Linux Kernel Module

William Nilsson
Erik Wetter

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Automatic Configuration Sharing in Peer-to-Peer IoT Networks
Implementation and Evaluation of Automatic Configuration Sharing as a Linux Kernel Module
William Nilsson & Erik Wetter

Supervisor: Ahmed Ali-Eldin Hassan, CSE
Advisor: David Brandberg, Combitech AB
Examiner: Ahmed Ali-Eldin Hassan, CSE

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Automatic Configuration Sharing in Peer-to-Peer IoT Networks
Implementation and Evaluation of Automatic Configuration Sharing as a Linux Kernel Module
William Nilsson & Erik Wetter
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Given that Internet of Things (IoT) devices are becoming more popular in the industry, faster ways of configuring these IoT devices need to be explored. This thesis was carried out in collaboration with Combitech AB to explore a new and automatic way of configuring IoT devices using Peer-To-Peer (P2P) networks and Linux Kernel Modules (LKMs). The resulting algorithm uses knowledge-sharing to distribute a configuration efficiently between homogeneous devices in the same P2P network. Tests on both physical and virtual devices show that the algorithm not only reduces the configuration times of new devices but also does so in a lightweight manner.

Keywords: Computer Science, Engineering, Embedded Linux, Kernel Module, Automatic Configuration, Industry, IoT, IIoT, P2P.

Acknowledgements

We want to thank Combitech AB for the opportunity to do our thesis as a collaboration with them, and for their support during the thesis work, providing hardware, office space and insights.

We also want to thank our supervisor and examiner Ahmed Ali-Eldin Hassan for the valuable support and feedback throughout this thesis.

Finally, we want to express our gratitude to our families for their continued support during our years at Chalmers, as well as throughout this thesis.

William Nilsson, Gothenburg, 2024-06-26
Erik Wetter, Gothenburg, 2024-06-26

Contents

List of Figures

List of Tables

List of Abbreviations

TTC	Time To Configured
P2P	Peer-To-Peer
LKM	Linux Kernel Module
DKMS	Dynamic Kernel Module Support
IoT	Internet of Things
IIoT	Industrial Internet of Things
PnP	Plug and Play
GPL	GNU Public License
I/O	Input/Output
DoS	Denial of Service
GPIO	General-Purpose Input/Output
SSH	Secure Shell

1

Introduction

The use of newer technologies is becoming more common in the industry as it moves toward smart factories. This trend is part of what has been named "Industry 4.0" and comes with greater adoption of technologies such as Internet of Things (IoT) devices, AI, machine learning, and cloud computing [1]. Focusing on IoT devices, it is common that these devices need to be configured for a specific purpose in a factory, e.g., a temperature sensor configured to set off an alarm at a certain temperature threshold or multiple devices configured to send their data to a common database.

Combitech AB, which this thesis is in collaboration with, has customers that currently have a manual process to configure new devices. Consequently, the current trend with IoT devices would lead to much more time spent configuring devices. As the configuring is done manually, it has the consequences of being costly in terms of both money and time, as well as potentially introducing human errors. Because of this, Combitech AB has an interest in exploring an automatic way of performing these configurations that can remove the need to perform configurations manually.

By introducing software that can eliminate the need for human intervention in configuring IoT devices, it would be possible to reduce costs and minimize the number of human errors introduced into a system. Therefore, the goal of this thesis is to design and verify a small-scale automatic configuration sharing implementation requiring minimal human intervention.

Multiple ways of how such a solution could be implemented were considered. These were either targeting specific hardware directly (bare metal programming) or by utilizing an operating system as an abstraction layer. The former will be better concerning performance but will be limited to working only on the targeted hardware. Linux, on the other hand, which is an operating system, has support for a vast amount of devices, and by targeting Linux, so will our solution. Thus, this implementation will support a wide range of devices without requiring any modifications, improving the time and complexity that comes with distributing a new system in, for example, a factory.

Furthermore, targeting the Linux operating system opened up the possibility of going two different ways with the implementation since applications in Linux can be implemented in two ways. The first and most common one is in user space, in which applications have restricted access to the system resources. If more access is needed or if it wants to execute privileged operations such as sending an internet

packet or saving information to disk, it has to ask the kernel via system calls. This adds a layer of security, but it also adds overhead that will lessen the performance. The second way of implementing an application is directly in kernel space as a Linux Kernel Module (LKM). This allows for full control over the entire system and removes possible overhead. Furthermore, there is the possibility of creating an extremely minimalized system consisting of only the kernel and a minimal amount of user space applications.

The initial idea was to implement the algorithm presented in this thesis as an implementation in both kernel space and user space and compare the results to find out if the overhead from user space is significant enough to affect the algorithm's speed. Through our initial investigations, however, we determined that the difference would not be big enough to affect the algorithm. Nonetheless, given that a kernel space implementation had other benefits we chose to implement the application in kernel space. This was done to investigate whether it was a viable way of implementing automatic configuration and investigate if the other differences between user and kernel space were beneficial.

There is also the choice between a central server or a Peer-To-Peer (P2P) network to consider. Both have their pros and cons, but a P2P network has the possibility of more easily introducing a fault-tolerant system since there is no single point of failure. In contrast, a central server would mean one point of failure; hence, there is no fault tolerance in the case of the server not working. Given that the target setting for the devices is in a factory, and that IoT will yield significant economical benefits in the near future for Industry 4.0 [2], any downtime of these IoT devices would lead to lost revenue for the company as efficiency goes down and both time and money are spent on fixing the system instead of production. There are also other factors to consider, but when targeting an industrial setting one of the most important aspects is the economic aspect. Hence, the algorithm will be implemented using a P2P architecture. Another reason for using a P2P network architecture was that Combitech was also interested in exploring this area. Given the choice to use a P2P architecture, there is no central distributor of the configurations to the IoT devices as the devices make up the network. As a result, knowledge sharing between the devices will be used. Since an already configured device has access to its configuration, it can share this configuration with another device that has not yet been configured.

1.1 Problem Statement

In this thesis, we will develop, implement, and test an algorithm that performs automatic configuration of IoT devices. It should be implemented as an LKM using a P2P network structure. Our target is to significantly reduce the time it takes to configure devices. The goal is also for the algorithm to work without existing devices having prior knowledge of new devices.

1.2 Limitations

The scope is limited to accomplish the project within the given time frame. This section presents the decided limitations to the project's scope.

Firstly, the project will be carried out assuming that the onboarding of new devices to a network can be done securely. Instead, the focus will be solely on the configuration and knowledge-sharing part of the proposed automatic configuration solution. This is done since the subject of secure onboarding is a significant topic on its own, meaning it would be infeasible to investigate both automatic configuration and secure onboarding in this project.

Secondly, to further limit the scope, the resulting implementation of this project will only accommodate homogeneous networks where only a single type of configuration is present. An implementation accommodating heterogeneous networks would require multiple configurations in a single network. This would cause problems such as having to store all configurations in the network despite a corresponding device not being present and matching devices with their corresponding configuration when joining a network. As this project focuses on the fundamentals of automatic configuration in peer-to-peer networks, heterogeneous networks are deemed out of the scope.

Lastly, it is assumed that one node with the correct configuration is always present in the network to share the configuration with other devices. If this was not the case, an external service that is able to provide the configuration to the P2P network must exist, which is not the intended focus of this project.

2

Technical Background

This chapter presents relevant technical background information to provide the reader with a better understanding of the project. Topics presented in this chapter include IoT, P2P networks, Linux, and the hardware used to test the algorithm presented in this thesis.

2.1 Internet of Things

The IoT as a concept has been around for some time, at least in theory. The term ubiquitous computing, coined in the late 1980s by Mark Weiser, can be seen as an early predecessor to the modern-day IoT. As Weiser envisioned it, ubiquitous computing essentially means that there are small "invisible" computers around us that can perform simple computing tasks to aid humans in everyday life [3]. Although this description might have its similarities to IoT, they differ in that ubiquitous computing is more centered around computers, whereas the term IoT incorporates many other devices as well. Even though similar ideas existed earlier, the term *Internet of Things* did not emerge until Kevin Ashton coined it in the year 1999 [4, pp. 301–337].

The general interpretation of what IoT is has been changing through the years. In a more recent definition of IoT, IBM describes IoT as the collective name of networks consisting of physical internet-enabled devices that are connected through the internet. They also explain that the devices in these networks usually have sensors and actuators, enabling the devices to collect data and perform specific actions, and that given the data collection, actuation, and network connectivity, these devices can share data and work in unison to accomplish tasks autonomously. According to IBM, the term *IoT device* can include anything, ranging from a simple thermostat in a building to larger and more complex machinery used in the industry, as long as they have network connectivity [5].

Building on this definition, when IoT is used in the industry, it is commonly referred to as Industrial Internet of Things (IIoT), which essentially is a subcategory of IoT with the added specification that devices are designed for factories. Common use cases for this category of devices include aiding in the digitalization of factories, monitoring manufacturing, and managing inventory. Implementing IIoT in factories today has several benefits, such as lowered cost, reduced downtime by identifying faults earlier, and data gathering to aid in decision making [2].

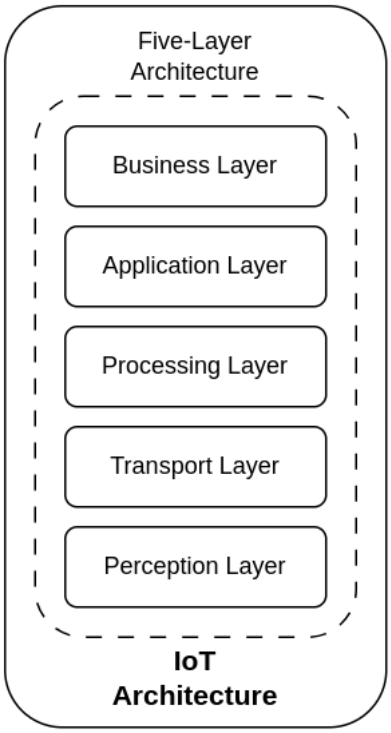


Figure 2.1: A representation of IoT through a five-layer architecture

2.1.1 Architectures

There have been many proposed architectures for IoT in the literature, but none have been universally agreed upon [6]. One of the proposed architectures is the five-layer architecture. The five-layer architecture divides IoT into the following five layers as seen in figure 2.1:

1. Business Layer
2. Application Layer
3. Processing Layer
4. Transport Layer
5. Perception Layer

Firstly, the perception layer consists of physical devices that can sense their surroundings or sense other physical devices nearby. A typical example of a device that can sense its surroundings would be a temperature or humidity sensor. Secondly, the transport layer handles data transportation between the perception and processing layers. Thirdly, the processing layer is responsible for processing and storing the data produced by the perception layer. Fourthly, the application layer consists of the different applications of IoT, as well as including application services provided to users. The final layer is the business layer, which consists of the business side of IoT. This includes policies for privacy, profit models for IoT, etc [6].

We consider our proposed algorithm to be somewhere between the application and processing layers. As the proposed algorithm works by having the IoT devices collectively process the configuration of a new device, it is not an application targeted at the user. At the same time, the algorithm's goal is to achieve the same results as that of a configuration application aimed at a user.

2.1.2 Typical Hardware Characteristics

Because the use cases of IoT often revolve around connecting devices and performing simple tasks, the performance is often limited as an abundance of computing resources is not needed in such cases. As mentioned before, a benefit of IIoT is reducing costs. Hence, it is natural that these devices are as cost-effective as possible. This cost-effectiveness also contributes to IoT devices generally being resource-constrained, as cost and device performance correlate.

The devices also require some more advanced capabilities that cheaper microcontrollers might not have. For example, for this thesis, we assume that a device will have a network interface and support for running a Linux-based operating system. We are still assuming that the devices are resource-constrained which motivates the choice of implementing the system as a LKM. And by targeting Linux, there are no additional requirements for hardware other than Linux having support for the architecture.

2.2 Peer-to-Peer Networks

P2P networks are defined as distributed networks in which no central controller is present [7]. Hence, all participants of a P2P network contribute to sustaining the network itself and the services offered by that network. Participants of a P2P network are often called peers or nodes [4, pp. 43–44]. The fact that there is no central controller means that each peer must communicate directly with other peers. This, together with the fact that participants are referred to as peers, is why the networks are referred to as *peer-to-peer*.

2.2.1 Peer-to-Peer and Internet of Things

P2P networks are of high relevance when it comes to IoT. The need for scalable network solutions must be considered as the popularity of IoT devices keeps increasing due to the large-scale adoption of IIoT in Industry 4.0. In essence, a P2P network would be suited for this type of task. As mentioned above, in terms of scalability, a P2P network can scale arbitrarily and cost-efficiently. With no central controller, the fault tolerance increases, and the time spent maintaining and fixing broken components goes down, which increases productivity and thus also the revenue for the company [2], as mentioned in the introduction. However, despite these positives, to our knowledge, the use of a P2P architecture is uncommon in a production setting when looking at IIoT. This is especially prevalent regarding automatic configuration solutions for IIoT devices where a central controller is common.

2.2.2 Architectures

P2P networks are usually categorized as one of two architectures. These architectures are structured and unstructured networks, as shown in figure 2.2. The unstructured P2P networks do not have peers in a specific structure in the network. Instead, the network is created through peers forming connections with other peers in an unstructured manner. In other words, any peer can form a connection to any other peer in the network, the network itself does not maintain any specific order. On the contrary, a structured P2P network has a certain structure enforced. These structured networks can make searching for, e.g., a file in a P2P network more efficient [4, pp. 43–44]. As seen in figure 2.2, the structured network sacrifices fault tolerance since a node leaving would disrupt the ring. However, protocols for structured P2P networks often have measures in place to keep the network fault tolerant, such as in the case of the structured P2P protocol Chord. In the Chord protocol, a node keeps track of not only its immediate successor but instead the r succeeding nodes. For the network to be disrupted, all of these succeeding nodes must fail, which can be made quite improbable even with r being a relatively small number [7].

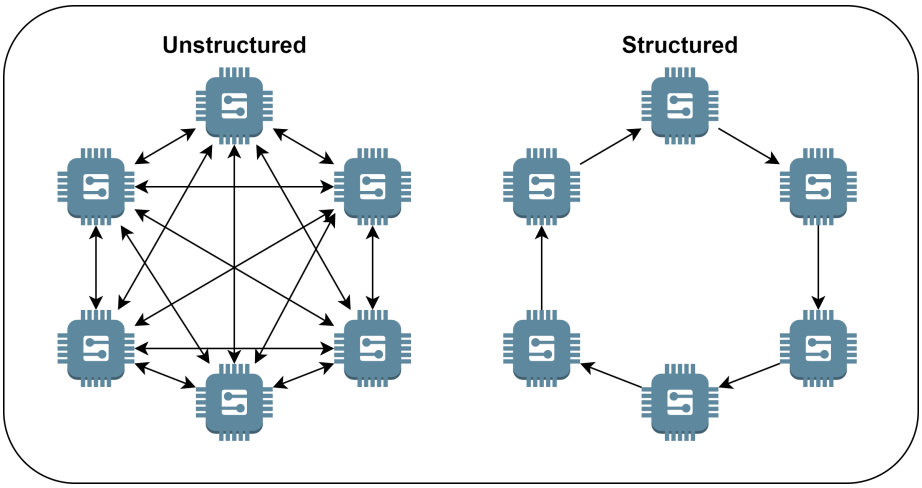


Figure 2.2: A simplified depiction of an unstructured and a structured P2P network

A P2P network can also be classified according to Schollmeier's definitions [8]. Sholmeier separates P2P into smaller subcategories, namely pure P2P and hybrid P2P. Pure P2P entails that the departure of a single, arbitrarily chosen node from a network should not affect the services the network provides. Hybrid P2P is defined as a P2P network that requires a central controller to provide a subset of the network's services that would not be available without the central controller. Examples of pure P2P protocols are Chord [7] and Kademlia [9], and an example of a hybrid P2P network would be the original mp3-sharing P2P network Napster with its central server for indexing [7].

2.2.3 Peer-to-Peer vs Client-Server

P2P offers both advantages and disadvantages compared to a Client-Server architecture. The following categories include some of the ways the two approaches fundamentally differ from each other.

Fault Tolerance

One of the significant advantages compared to Client-Server architectures is that a P2P network generally can be made fault tolerant more easily. As the Client-Server architecture often relies on a central server to serve all clients, the risk of the service being unavailable would be higher as there is a single point of failure. However, when the client-server architecture is used, steps are usually taken to remedy these problems. For example, it is possible to have backup servers running in the background that are ready to take over if the active server goes down or cannot handle requests. However, this solution has its own problem, being that it could be costly to have extra servers and difficult to keep the backup server synchronized with the active server. In contrast, a P2P network can continue to function unaffected even when an arbitrary node leaves, given that it is a pure P2P network [8]. As mentioned before, perfect fault tolerance is not achieved when using a structured P2P protocol, but a high fault tolerance can still be achieved with no added cost, as seen in the Chord protocol [7].

Computing Resources

P2P also has the advantage of sharing computing resources. This means that as the network grows, the total compute power of the network increases [4, pp. 43–44]. Therefore, the network can scale arbitrarily if an efficient P2P protocol is used. On the other hand, the central server in the typical Client-Server architecture will eventually become saturated as the network grows and more devices require its services [4, pp. 43–44]. Of course, increasing the server's performance would be possible, but then again, this would also increase the costs.

The possibility of exhausting the resources of a central server also makes the Client-Server architecture vulnerable to Denial of Service (DoS) attacks. In such attacks, an attacker would purposefully consume a server's resources to stop legitimate clients from using the server's services. P2P networks do not suffer from this weakness as much. This is because of the fundamental difference which is that all devices in a P2P network can act as a server.

Security

Regarding security, one of the disadvantages with P2P compared to a Client-Server architecture is that every node is equal. Since every node in a P2P network can act as both a server and client, a malicious node could theoretically join the network and start spreading, e.g., malware. In a client-server architecture, this is inherently solved by there being a central server, that is known to be trustworthy, providing all services. A way of mitigating the problem for P2P networks could be to implement trust in the network. With trust implemented, a device can choose which device's services it uses based on a trust score, making it harder for a malicious device to

cause harm. It would also be possible to eject nodes if their trust score is low, removing potentially malicious devices altogether.

2.3 Linux for Internet of Things

IoT devices' architectures are diverse and include various capabilities. They can utilize 64-bit multicore or simple unicore processors, running either a fully-fledged operating system or bare-metal code. Bare-metal code runs directly on the hardware and is often found in microcontrollers with a specific use case. The benefit of running a system without an operating system is that the performance of simple applications will be good, and the energy consumption will be lower. The many abstractions and added features of an operating system might be unnecessary for simpler use cases. However, as applications become more advanced, utilizing parallelization, networking, and more, an operating system could give a lot of benefits.

One popular operating system used for both servers, personal computers, and IoT is Linux. According to the Eclipse Foundation's 2023 IoT & Edge Developer Survey Report, it is the most used Embedded OS for constrained devices [10].

Linux is an open-source operating system platform [11]. Linus Torvalds originally developed Linux based on the existing conventions of the UNIX family of operating systems, and it quickly grew into a production ready system [12]. Linux is unique, partly because of its license, GNU Public License (GPL), which says that users are allowed to freely copy, modify, and redistribute its source code and any resulting binaries as long as the modified source code is publically available. This is different than, for example, Windows, where the source code is proprietary and only the binaries are being distributed and available to the end user.

Thanks to this open nature, anyone can write drivers for new hardware and add support for new CPUs. This makes it easier to take an already working operating system and deploy it to the ever-expanding list of newly developed processors. Thanks to this flexibility, targeting Linux will increase the number of devices the applications will support compared to writing bare-metal code.

Furthermore, the Linux system platform can be seen as a stack of layers as seen in Figure 2.3. The bottom layer is an interface to the hardware on which the software is being executed. The layer above is where the actual operating system lies, which handles memory and process management, the file system, Input/Output (I/O), and more. The next layer above that is the library interface, which gives programmers easier access to kernel features such as file processing (open, close, read) and process management (fork). At the very top lies all the utility programs which the user normally interacts with (shell, browser, text editor, etc).

2.3.1 Modes of Execution

In Linux, programs can be run in two different modes: *kernel mode* and *user mode* [13]. The kernel mode is also called privileged mode, and when code is executed in this mode, it has access to all the computer's resources including the

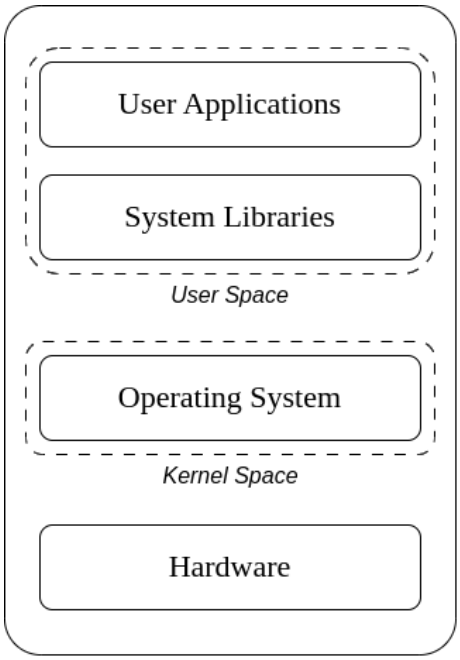


Figure 2.3: An illustration of the various layers of a Linux system.

address space of all running processes and full control of what happens on the computer.

To avoid giving all programs full access to the device it runs on, they can be run in *user mode*. In this mode, the program can only access its own process's memory space. Additionally, it can only use non-privileged instructions, and if it wants to use restricted instructions, it does so via system calls to the kernel through system library functions. The kernel then decides what to do with the request, and can maintain control over what process accesses which device, what file, what memory address, or other resources. Thus, the kernel can maintain order and security of the system.

However, a context switch is required whenever a system call is executed before the kernel routine can be run [13]. To ensure the previously run process can be resumed later, the context, such as the CPU register values, process state, and memory information, must be saved. The time it takes to save and restore this information can be considered performance overhead since the processor can not do anything useful simultaneously.

2.3.2 Kernel Modules

The Linux kernel has support for dynamically loading and unloading new features and drivers through *kernel modules*, normally called an LKM [13]. These modules are additions to the kernel code and share both memory and code space. LKMs enable developers to work within kernel space without needing to recompile the kernel and reboot after each change. Instead, the modules can be compiled and inserted into a running kernel on demand. With Dynamic Kernel Module Support (DKMS),

modules can also be automatically rebuilt when a new kernel version is installed, ensuring it works immediately after a kernel upgrade instead of waiting for the developer to update it for the new version manually. It will also ensure the module works for a wider range of kernel versions already running without requiring pre-compiling and distributing each version separately.

Kernel modules are convenient partly because they allow for easier development of device drivers. Instead of modifying the kernel source code, recompiling, linking, and reloading the kernel, one could compile and load a module into the running kernel, making the development cycle more efficient. Once the driver is complete, distributing it to other devices is also more convenient when deployed as a module instead of as a patch for the kernel source code since kernel recompilation is unnecessary. Another reason kernel modules are convenient is that they are not hindered by the GPL license like the kernel. Instead, both companies and individuals can develop and distribute modules on their own terms, choosing whether or not to open-source them and who should have access to them.

One limitation of writing kernel modules that has to be considered is that C is one of few programming languages supported as the kernel itself is currently written in C [14]. Rust is another language being integrated into the kernel, but is, at the time of writing, still experimental and under development. This differs from user space, where there are many more languages to choose from. C is a very powerful language, but it also comes with a few risks if the programmer is not careful. Combined with writing code in kernel space, this could potentially introduce some very dangerous vulnerabilities.

2.3.3 Kernel Networking

The developer can utilize various system libraries to make the process easier when developing a network application in user space. However, when developing a kernel module, using those libraries is impossible. Instead, it is more complex because it interacts directly with the network stack and thus requires more internal networking knowledge. Furthermore, the debug process is more complex since common tools that many developers might be more used to are unavailable.

But, since there is no restriction on what a kernel module can do, there are plenty of benefits to writing network-enabled applications in the kernel. It is possible to implement new network protocols, do packet filtering for firewalls, or improve the performance of sending raw messages [13]. Additionally, all code is already being executed in the kernel context, avoiding all context switches that are normally required by such operations.

2.4 Hardware

Raspberry Pis are used to test and evaluate the solution proposed in this thesis. More specifically, the Raspberry Pi Zero 2 W model is used, which offers wireless connectivity and decent performance in a small-scale package. Figure 2.4 shows a

picture of five of these Raspberry Pis. The Raspberry Pi Zero 2 W is produced by the Raspberry Pi Foundation and features a quad-core 64-bit ARM Cortex-A53 CPU clocked at 1 GHz with 512MB of RAM [15]. These specifications, together with the fact that it supports wireless connectivity up to 600 Mbit/s using the WiFi 4 protocol, make the Pi Zero 2 W (Pi) a suitable selection to represent a generic IoT device capable of running a fully-fledged operating system.

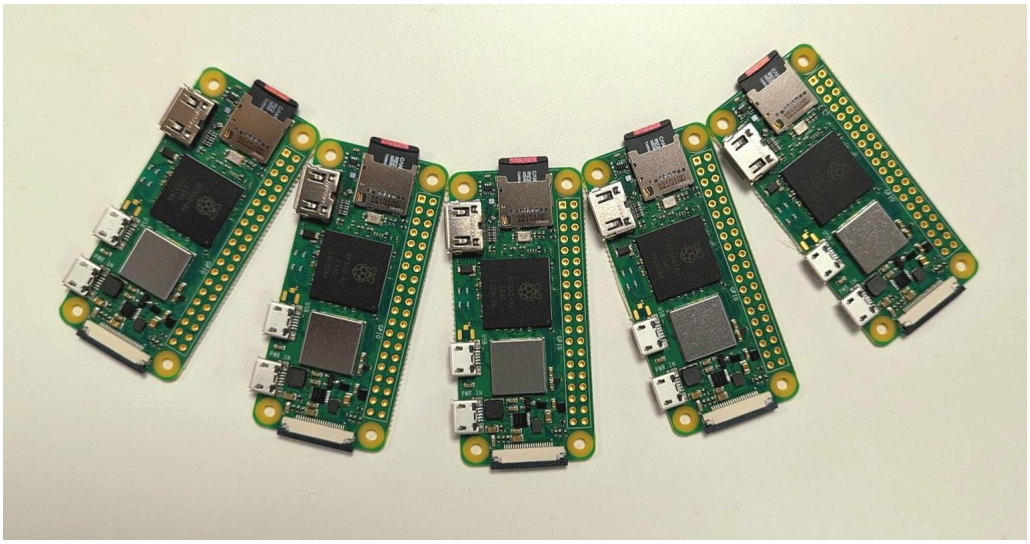


Figure 2.4: Five Raspberry Pi Zero 2 W

Another reason these devices were chosen was because, through the General-Purpose Input/Output (GPIO) pins, it is possible to access a serial terminal on the Pi. This means that it can be entirely controlled from another device, removing the need for peripherals, such as a keyboard, mouse, or display, to be connected. Additionally, using a physical cable instead of, for example, Secure Shell (SSH), will ensure that the connection will work from the moment the device boots and that the Pi will continue to print errors after a kernel crash.

3

Related Work

In this chapter, we review existing strategies to perform automatic configuration of network-connected devices. By examining these strategies, we aim to find gaps and situate our proposed algorithm in the context of automatic configuration. The following review serves as the foundation for our proposed algorithm to ensure its originality.

Automatic Configuration

A proposed solution for automatic configuration of IoT sensors from Madsen et al. [16] is the use of QR codes to avoid having to input metadata for new devices manually. In their proposed solution, QR codes are located on IoT sensors and walls in rooms where IoT sensors are known to be installed. A QR code on a sensor gives information about the device itself and could, by using their solution, contain any pre-defined information such as brand, model, MAC address, etc. A QR code on a wall works similarly but contains location data instead, such as which floor, room ID, building ID, etc. The system Madsen et al. have created works by scanning a QR code containing location data and a QR code with device information, which is then sent to a backend. The backend then associates the device information with the given location information and creates a digital twin of the device in a database. After this, the device is configured and can be used, given that the metadata associated with the QR codes is correct. The obvious limitation of this strategy is that it still requires a human to scan the QR codes to set up a new device. Not having to input configuration data manually would probably reduce configuration times, but it does not provide the level of automation of the approach presented in this thesis. On the other hand, the approach presented by Madsen et al. can handle heterogeneous networks, which the approach presented in this thesis cannot do in its current state.

Chatzigiannakis et al. [17] propose yet another solution for automatic configuration. Their solution is based on the fact that devices with similar metadata, i.e., similar types of IoT devices, have the same trends in their raw data output. By comparing the raw data output from a yet-to-be-configured device with all devices in the same network with known metadata, their proposed solution can suggest what type of device the yet-to-be-configured device is and, thus, guides a person in configuring it correctly based on this information. Similar to the approach presented by Madsen et al. this approach also requires human input to work, not reaching the level of automation aimed at in this thesis.

Another proposed solution comes from Reinhart et al. [18] and introduces automatic reconfiguration of robot cells based on Industrial Ethernet Networks with real-time capabilities. Their solution aims to implement Plug and Play (PnP) functionality for new devices into an already existing network. A five-step process was defined to fully integrate a new device:

1. Physical Connection
2. Discovery
3. Basic Communication
4. Capability Assessment
5. Configuration

The paper presented two different concepts and validated them in three different setups. The first concept uses the conventional Ethernet protocol during all the steps mentioned above and could, therefore, use standard methods for discovery (pings, broadcasting) and for communication (UDP, TCP). A primary node will then respond and initiate the configuration process. This comes with the significant benefits of having access to many tools focused on Ethernet and having more people already knowing those systems. The second concept uses periodic polling for new devices on the network. When a new node enters, the primary node will interrupt its boot process, modify its configuration, and restart it. Something in common between these concepts is a centralized node managing the configuration of the controlled nodes. Both concepts proved successful after validation in three different setups, and the reconfiguration time was reduced, at best, down to seconds. The paper shows a result that is similar to what this thesis aims to achieve. The main difference is that it requires a special node to monitor and configure the new devices, while we will implement this system as a P2P network where the running nodes themselves are responsible for configuring a new device. However, one feature that the paper mentions, that we lack, is the capability assessment step that will allow for the devices to be configured differently depending on their capabilities.

There has also been some research on automating the first step of configuring new devices for wireless network connectivity, namely wireless network setup, which the previously mentioned papers did not focus on. Boskov et al. [19] propose two solutions for this, both of which use a mediator device. The first one uses the concept of a software access point (Soft-AP), which essentially means that a device, which is normally a client, becomes a virtual access point using its wireless interface. In their solution, a mediator device connects to this access point, enabling the transfer of a given network's credentials to the new device. The new device then applies this network configuration and restarts as a client. Their other solution works similarly, but Bluetooth is used instead of a soft AP. Additionally, their Bluetooth approach does not require manually inputting network credentials, unlike their soft AP approach. In their testing, they found that the Bluetooth approach was the fastest, and it could improve network configuration times by up to roughly 900% compared to a non-expert manually configuring the network settings and up to roughly 400% compared to an expert manually configuring the network settings. After the device

has received the network credentials, it fetches a configuration from a remote server to complete the configuration. However, as with many other approaches mentioned, both of these methods require that a human controls the mediator device. It also lies outside of the scope of this thesis as this would be categorized as part of the onboarding. Although it could provide a way of securely performing onboarding it does fit with the goal of full automation.

Furthermore, Amazon Technologies has a patent for a process of authenticating and integrating new devices in cloud networks with the help of generic provisioning certificates [20]. These generic provisioning certificates are installed on the devices during the manufacturing process. Devices of the same type have identical provisioning certificates. Even though the certificates are identical, each device is given unique credentials when integrated according to their process. The patent describes a process in which a new IoT device sends a request to be provisioned to a provisioning service, which is then forwarded to an authentication service. The authentication service's task is to check the validity of the provisioning certificate that the device originally came with. Once the validity of the provisioning certificate is confirmed, the provisioning service loads a user-defined authentication workflow. After the device has been fully authenticated the provisioning service sends a request to fetch a unique certificate for the requesting device from a certificate authority. The provisioning service asynchronously loads a user-defined provisioning workflow that performs necessary tasks such as associating the new certificate with the IoT device on the server side. Lastly, the certificate authority sends the newly generated certificate to the requesting IoT device, making it uniquely identifiable. This approach could probably be adapted to perform the onboarding part of the automatic configuration approach presented in this thesis. It does reach the level of automation achieved by our approach, and their method could provide a way of uniquely identifying devices.

4

Methods

This chapter describes the protocol used to communicate within the P2P network, the messages used, and how the nodes should behave in certain situations. Section 4.1 introduces the protocol requirement set up early on in the project, while Section 4.2 presents the onboarding process for new devices connected to the physical network and about to join the P2P network, as well as the functionality required to maintain a P2P network and how to handle a device leaving. Section 4.3 clarifies how and when the nodes will share their configuration with each other, both when a new device joins and when there's a newer configuration available. Finally, Section 4.4 describes the various states a node could be in.

4.1 Design Requirements

Since this thesis explores a new way of achieving automatic configuration of IoT devices, certain things must be considered regarding the design of the proposed solution. Firstly, the solution must minimally depend on human input to automate as much as possible. Secondly, the proposed solution must be relatively fast to show an improvement in TTC over manual provisioning, motivating an algorithm adaptation. Lastly, the proposed solution must be lightweight to enable IoT devices with limited computing power to run the algorithm.

Table 4.1: A list of the messages from the protocol introduced in this thesis.

Command	**Content**	**Description**
HELLO	IP Address	Broadcast presence and request to join a P2P network.
JOINME	IP Address Successor IP	Response to HELLO, telling new node where to join the ring.
FETCH	Configuration File	A request to download the current configuration file from one of its successors.
STATUS	State Config Version Successor List	Periodic status update sent to the closest successor to make sure it is alive and updated.

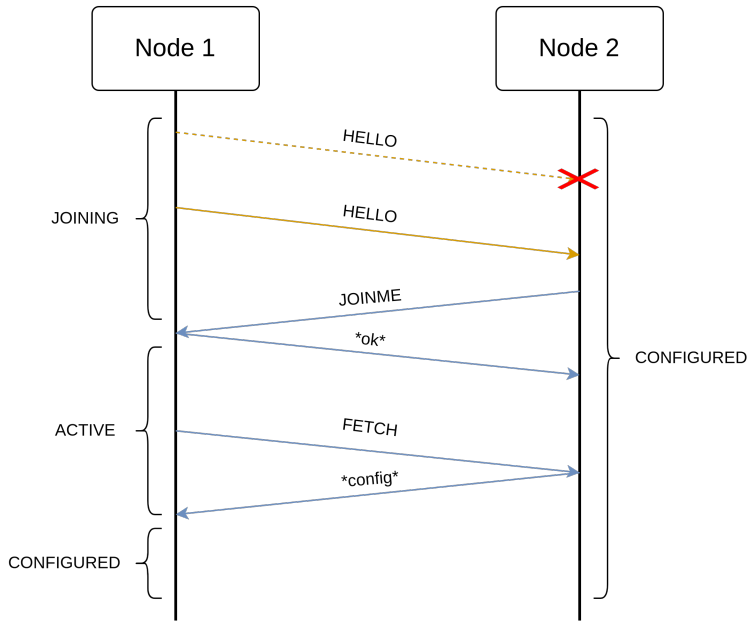


Figure 4.1: State transfers and messages sent for a new device joining a P2P network.

4.2 Device Onboarding

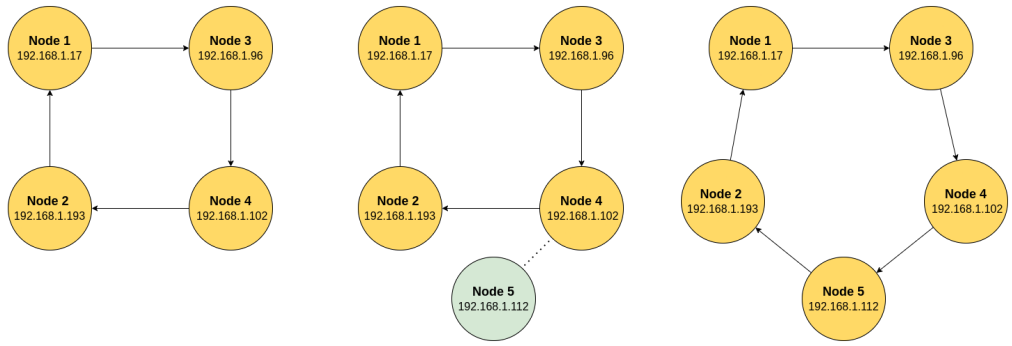
This section covers the design of the onboarding process of a new non-configured device. The onboarding process stretches from starting a new device to when it is fully integrated into the targeted P2P network. When new devices join a network, they are assumed to have no previous knowledge of the network or devices within it, i.e. a device should not require any prior configurations. The messages used to communicate between the devices can be seen in Table 4.1 and will be described in more detail in the following subsections.

4.2.1 Device Presence

When a yet-to-be-configured device has connected to a network where an underlying P2P network exists, it must make its presence known to at least one device within the P2P network to join that network. The solution in this thesis is to broadcast a HELLO packet from a new device to all other devices in the network. This packet contains the sending device's IP and is a request to join the P2P network. This solution will ensure that the packet will reach at least one device that is part of the P2P network since all devices in the network will receive it, given no packet loss.

To protect against packet loss, this message will be broadcasted at an even interval to make sure it will reach the correct target eventually. This is visible in Figure 4.1 where the first HELLO message does not reach Node 2. This solution means that a new device and the existing devices in a target P2P network do not need any previous knowledge of each other. The repeated HELLO message further guarantees eventual successful integration, even though the UDP messages might be prone to get dropped.

4.2.2 Device Organization



(a) Before (b) During (c) After

Figure 4.2: Process showing what onboarding a new device could look like from a topology perspective.

The P2P architecture used in this algorithm takes inspiration from the P2P protocol Chord [7]. As such, the devices are structured in a ring and sorted based on their IP address. An example of how a network containing four devices is structured can be seen in Figure 4.2a. Note that even though the devices might join differently, all nodes point to a successor with a higher IP (except the last one, which instead points to the one with the lowest IP). Thus, there is a natural order to how the devices relate to each other, and it will always be clear where a new device belongs.

After the device has made its presence known, one of the devices in the network will be responsible for helping to integrate the new device. The device responsible is the predecessor and will be the only device to respond. This can be seen in Figure 4.2b where Node 4 responds to Node 5's request with a JOINME message. The message contains Node 4's IP address and the IP address of Node 2 which is the new successor to Node 5. If this step is successful, and Node 5 responds with an OK, the node is successfully integrated into the P2P network and changes state to ACTIVE.

The next step is for the newly connected node to contact its successor and request their configuration. This is done though the STATUS message that is sent periodically and serves multiple purposes.

4.2.3 Maintaining the Network

After a node has joined the network, it must also keep its information updated to ensure it is aware of any changes made to the layout. It could be that another device has joined that should be stored in the successor list, a device has failed or been disconnected, or a new configuration entered the network and should be distributed.

To detect whether a device has dropped out of the network after breaking or being disconnected, each node periodically pings its successor and expects a response using the STATUS message. If a node no longer receives a response from its closest successor, it will remove that node from its successor list and change its successor to be the next device in the ring.

To accomplish this, each node must also keep track of a few extra nodes as redundancy. They do so by keeping a list of a constant size with the following few devices' IP addresses. To keep this list updated, each node will periodically ask its closest successor for their successor list and use it to update its local list.

The response to the STATUS message contains useful information for the sender, in addition to just indicating availability. As can be seen in Table 4.1, the response contains the current state, the current configuration version number, and a list of all successors. Thus, when receiving this response, the requesting node will take the new list of successors, remove the last entry, and change its current list to be the immediate successor node followed by the newly received list. This will ensure that the nodes will also recognize changes in the P2P network further away.

4.3 Knowledge Sharing

This section covers the design of the knowledge-sharing process in the P2P network. The knowledge-sharing process stretches from when a device is integrated into the P2P network to when it has been configured. It will also occur whenever a newer configuration is detected in the network.

4.3.1 Configuration Transfer

After a device has joined the P2P ring the next step is to fetch the configuration. Since the network is homogeneous (every node shares the same configuration), the device can ask its successor about their configuration. It does so via a *FETCH* command sent over TCP. The successor then responds with the most recent version of the configuration it has in memory. In case the successor does not have a config file ready for transfer, which could happen if that device also joined the network recently, the successor list comes into use and the node fetching can iterate through the list until a node with a config is found. In case no such node exists, the fetching node will time out and try again later.

4.3.2 Handling Updated Configuration

Similarly to detecting new devices, detecting when a new configuration appears requires continuous polling of the other devices. This is done in the same message as for the successor list. Whenever a newer configuration version is detected, the calling node will fetch the newer configuration from its successor. It does, therefore, not matter which device gets the updated version first. Eventually, it will propagate through the network and all nodes will run the latest version.

Table 4.2: A list of the states from the protocol introduced in this thesis.

State	Description
JOINING	The node is trying to join an existing P2P network
ACTIVE	The node is an active part of a P2P network, but has not been configured
CONFIGURED	The node is an active part of a P2P network and has been configured

4.4 States

Each node implements a state machine to maintain order and simplify the network behavior, these states can be seen in Table 4.2. This makes it easier for each node to know when to respond to various messages, e.g., if it is possible to respond to a FETCH request from another node, see Table 4.1. In other words, a requestee should only respond to a requester's FETCH request if the requestee is already configured. To enforce this, a node keeps track of its state by changing it whenever an event necessitating a state change has happened. For example, if a JOINME message is received when it is in the JOINING state, the node will change its state to ACTIVE. The state machine is also important to know when to run certain parts of the algorithm, such as stabilizing the network, and when to attempt fetching a config. As these are run at fixed intervals, the state machine ensures that these parts are only run when necessary. For example, a node would not attempt to request a config of the same version if it is already configured, as this would only cause unnecessary network traffic and serve no purpose for the node.

5

Design and Implementation

This chapter will highlight the various methods used when implementing, debugging, and validating the protocol.

5.1 Linux Kernel Module

To start developing a kernel module, there are some requirements. Firstly, it is necessary to install kernel headers for the Linux version that is targeted [21]. The simplest way would be to install it from a package manager. For this project, since the module is supposed to be run on a device running a different architecture, we opted to compile the Linux kernel from the source and use the headers created at that point. Secondly, in our case, as the Pi is an ARM-based system, a compiler that supports compiling for a different architecture is required unless you are compiling the kernel on the target machine. We chose to not compile on the target machine as compiling is a computationally heavy task. Instead, we used a cross compiler on another system with more performance.

5.2 Emulation

When developing kernel features, using a virtual machine for testing is recommended instead of running the module directly on the host machine [21] to reduce the risk of damaging the host system. It also reduces the complications when the host machine crashes after a faulty module has been inserted into the kernel. The authors of *The Linux Kernel Module Programming Guide* describes it best: "[...] if you start writing over data because of an off-by-one error, then you're trampling on kernel data (or code). This is even worse than it sounds, so try your best to be careful." [21, p. 24].

5.2.1 Preparations

Since it was known early on in the project that the implementation would target Raspberry Pi devices, it was decided that the emulator should run the Raspberry Pi OS on an ARM Cortex A53 CPU with 512 MB of memory. This is the same setup as a Raspberry Pi Zero 2 W [15] that was used for the testing done on physical devices. The goal of emulating the same architecture was to reduce possible complications

when deploying to the physical devices. QEMU [22] was the chosen full-system emulation tool due to its ease of use for emulating specific platforms and processors.

The kernel used for emulating needs to be replaced with the official kernel instead of the customized version provided by the Raspberry Pi Foundation to emulate the Raspberry Pi OS in QEMU. As far as we know, this was the case because the official Raspberry Pi kernel is incompatible with QEMU. The official kernel was fetched and compiled from the source.

Building the kernel from the source comes with the added benefit of being able to configure it with additional features that help while debugging. One such feature is the option to forcefully unload modules, which is useful when you want to unload a module that is in use, is marked unsafe, or does not want to be removed [23].

The Linux version compiled was 6.1.63 and was chosen to match the version that, at the time of writing, the most recent version of Raspberry Pi OS was based on. The compiled kernel must match the version used for the Raspberry Pi OS or the module will not be insertable because of a version mismatch.

5.3 Data collection

In order to obtain relevant results regarding the algorithm, tests had to be performed. This section presents the complete setup used for the tests and the type of tests performed.

5.3.1 Test Setup

We performed multiple tests with various setups to measure both the TTC and the network overhead created by this protocol. The test setup can be seen in Figure 5.1 and features one host running the emulators, another host running an HTTP server and Wireshark, and five physical devices, all connected to the same router.

The server listened to incoming status updates from the nodes, indicating that they were fully configured. It saved the timestamp of each device's connection and the total time it took from the first configured device to connect until the last. Using this method, we could measure the time it took from having multiple devices up and running (but not configured) until they were all configured after a manual configuration of one device. The configuration used for all tests was simple, including a fictive sensing interval and a fictive IP address to send data to. These were represented by two integer values.

With one host running Wireshark, which is a network packet analyzer tool [24], we could also track the messages sent back and forth between the devices. This allowed us to evaluate the overall overhead on both the devices and the network that the protocol introduced.

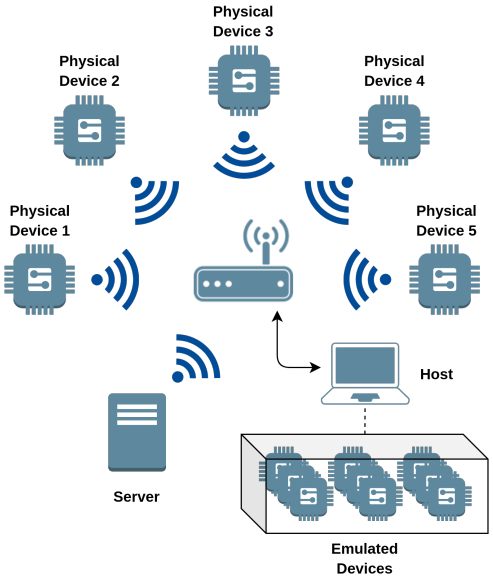


Figure 5.1: The setup used when acquiring data

5.3.2 Tests performed

TTC Measurements

We initially conducted a test to compare the time difference between physical devices and virtual devices. This was done to see if virtual devices would give similar results to that of physical devices. Next, we performed tests with a single device pre-configured, representing a group of devices where the initial manual configuration of one device is performed, and this configured device then shares its configuration with the other devices that join its network. We measured the time until all devices were configured for groups of five, ten, fifteen, and twenty devices, respectively. We also performed a test to measure the time it takes for a single device to be integrated into an existing network of configured devices.

Overhead Measurements

To gather overhead measurements, we tested the average number of HELLO messages sent before a device receives a JOINME message. This was done since the number of HELLO messages could vary depending on whether another device is available to handle a HELLO message. Then, we used the average number of HELLO messages (UDP overhead) together with the JOINME and FETCH messages (TCP overhead) to get the total network overhead.

Network Bandwidth Usage Measurements

We performed another measurement to see the network traffic a device produces per second when joining an existing P2P network. The network traffic a device produces per second when it is already in an existing P2P network is also of interest. Therefore, we measured this as well.

6

Results

This chapter covers the results of the tests mentioned in Section 5.3.

6.1 TTC Measurements

The first test compared the TTC of five physical devices and five virtual devices, given the same starting criteria. This was performed to measure the difference in TTC between physical and virtual devices when performing the same operations. The results showed that the physical devices had a higher TTC. The physical devices were approximately 2.5 seconds or 28% slower on average, as can be seen in Figure 6.1. It was also noted that the physical devices had a higher standard deviation. Calculated based on ten respective attempts, the physical devices had a standard deviation of 2.51 compared to 1.89 for the virtual devices.

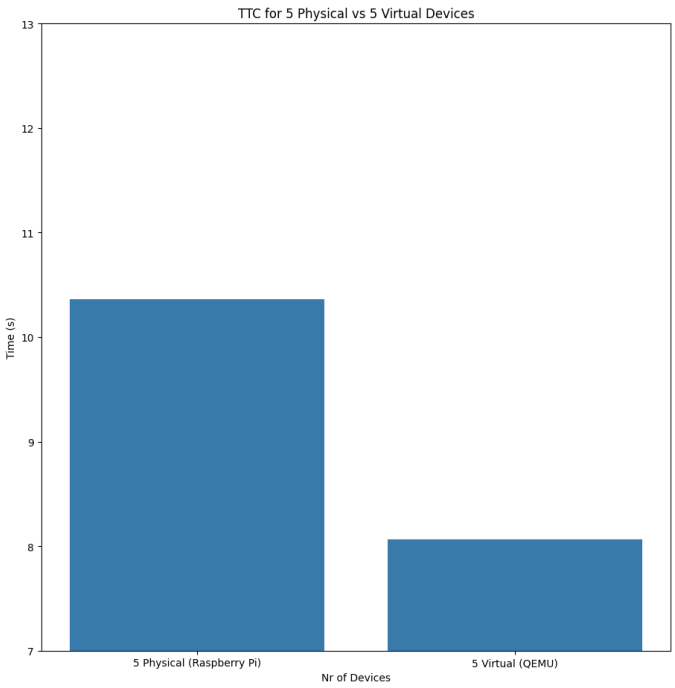


Figure 6.1: Comparison of TTC for five physical devices vs five virtual devices.

Figure 6.2 shows the results of the second TTC test, which measured the difference in TTC for an increasing number of devices starting with only one configured device. As seen in Figure 6.2, the general trend is a gradual increase in TTC when increasing the number of devices. When testing with five devices, the results showed an average TTC of approximately eight seconds with a standard deviation of roughly 1.89. When testing with ten devices, the average TTC was approximately 9.64 seconds with a standard deviation of roughly 0.61. When testing with fifteen devices, the average TTC was approximately 9.67 seconds with a standard deviation of roughly 0.77. Lastly, when testing with twenty devices, the average TTC was approximately 10.24 seconds with a standard deviation of roughly 0.50.

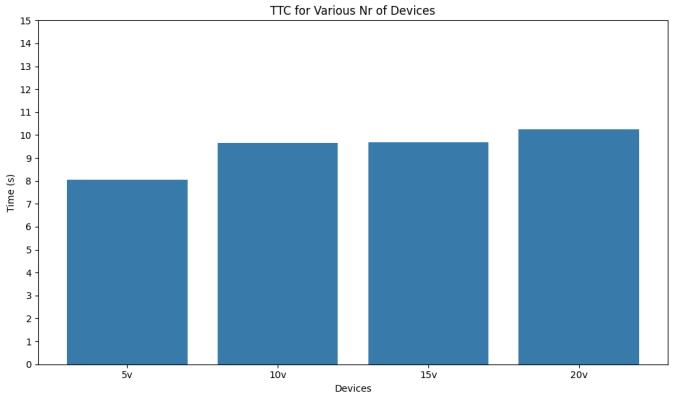


Figure 6.2: Comparison of TTC for an increasing amount of virtual devices.

The third test shows, as seen in Figure 6.3, that inserting one device into an already configured network had an average TTC of approximately 24 milliseconds, with a standard deviation of roughly 5.43. The attempt with the slowest TTC was roughly 72% slower than the attempt with the fastest TTC.

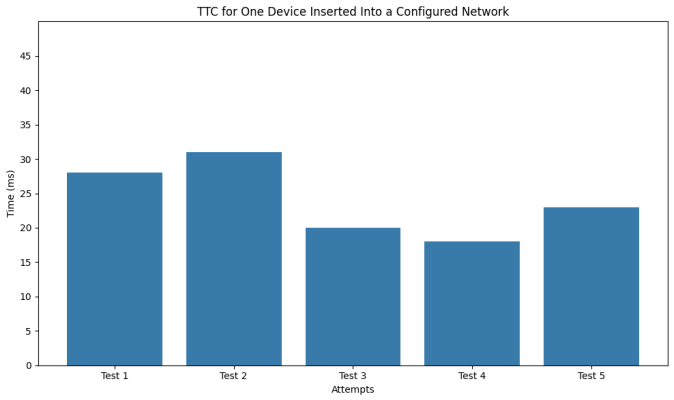


Figure 6.3: TTC for a single device inserted into a fully configured network.

6.2 Overhead and Network Usage

Onboarding

As mentioned earlier, there are three types of messages required for a device to join an existing network and become fully configured. These are HELLO, JOINME, and FETCH. Other than for the first manually configured device, all devices will start by broadcasting HELLO, receive a JOINME request from another node and finally call FETCH to its newly acquired successor.

Analyzing the messages in Wireshark with a zero drop rate, we could see that the entire procedure required, on average, 21 network packets, where 1 was the UDP HELLO message. The other 20 were the two TCP streams of 10 packets each for the incoming JOINME message and the outgoing FETCH request.

As for size, the UDP packet was 90 bytes large, with the data portion taking up 48 of those bytes. The first TCP stream was in total 745 bytes, where 62 of those were actual data. The second stream, the FETCH message, was 709 bytes in total, out of which 25 bytes were the request data and 8 bytes were the actual configuration being transferred.

The data being transferred when onboarding a new device, either as a request or as a configuration, was a total of 143 bytes. That corresponds to around 9% of the total network traffic generated, with the rest being network overhead.

Maintaining the Network

Every node sends a status request message at an even interval to maintain the network and ensure the successor list is updated. During these tests, this parameter was set to 10 seconds.

The Wireshark analysis showed that a STATUS message consisted of a TCP stream containing ten network packets. The stream was 747 bytes large, out of which 16 bytes were the request data and 39 bytes were the response data, containing everything mentioned in Table 4.1. This means that the data is around 7% of the total network traffic generated when sending this specific message.

The network overhead for one device generated by this message can vary, depending on the interval. In this case, the interval was set to 10 seconds, which meant that this message was being sent six times each minute, totaling to 4482 bytes per minute per device. The bandwidth used would increase linearly with the added devices. Thus, a P2P network with 100 devices running this protocol would have an overhead of about 448200 bytes per minute to keep the network running.

In case a device stops responding, its successor will keep trying a few times before deciding to remove it from the network. This uses the same STATUS message, as the predecessors will automatically get the information about the updated ring through that message. Thus, this process of removing unresponsive devices adds no extra overhead to the already existing STATUS message.

7

Discussion

This chapter starts by discussing the results obtained in Chapter 6. It continues with a discussion regarding the choice of using LKMs for this type of algorithm. Lastly, the chapter ends with a discussion regarding potential future work.

7.1 Discussion of Results

The results presented in Chapter 6 show that the algorithm provides a fast and efficient way of configuring a relatively large number of devices. The following is a more in depth discussion about the results that have been obtained.

7.1.1 TTC

The first test was performed to measure the difference in TTC between five physical devices versus five virtual devices, given the same starting criteria. The results show a clear advantage for the virtual devices, see Figure 6.1. This, however, is expected since they are all running on the same machine, communicating locally instead of over a wireless interface, which has to handle more noise and slower transfer speeds. As the following tests were performed using virtual devices, applying the percentual difference between physical and virtual devices obtained through this test might give a number closer to what can be expected in a real-world scenario.

The second test measured the TTC for an increasing number of devices. As can be seen in Figure 6.2, the results showed a slight increase in time as more devices were tested, but the increase in TTC slowed down and started showing signs of converging. However, more tests with even more devices would be required to conclude that accurately. However, due to the inherent nature of the algorithm, it is logical to conclude that as more devices become configured, the configuration of other nodes speeds up as more nodes are able to share their configuration. In other words, the algorithm should see an exponential increase in speed as more devices become configured. For example, if one node is configured, it can handle only one new node at a time. If, instead, ten nodes are configured, they could theoretically handle ten new nodes at a time. However, this assumes nodes join out of order based on their IPs. In the worst-case scenario, devices will join in order, leading to a linear TTC increase based on the number of devices joining instead.

The third test was performed to measure how fast a single device could be inserted into an existing P2P network. It showed an extremely fast TTC, although varying quite a bit between different attempts, see Figure 6.3. In absolute numbers, it is a barely noticeable difference. However, looking at the percentual difference, as presented in Chapter 6, the slowest time was approximately 72% slower than the fastest, which is a big difference. Since the difference is small enough to be measured in milliseconds, it could be explained by the fact that packets end up in a buffer as other packets currently occupy the responding node, delaying the process a few milliseconds.

Overall, compared to the current process of manually configuring sensors, our proposed method showed a great improvement in TTC compared to what could realistically be achieved manually. This is true for both simultaneously configuring several devices as well as introducing an un-configured device into an already running network of devices.

7.1.2 Overhead

As more devices were added, more network overhead was created to just keep the network alive. Even though each device did not contribute to a lot of network traffic on its own, with 100 devices it started increasing. If even more devices and sensors were to be added, it would, of course, increase even more. However, if the devices connect to a "normal" WiFi router providing a 2.4 GHz band, the network could have a maximum throughput of around 600 Mbit/s. This is the same limit as the physical device, as it runs the WiFi 4 protocol, making the network the bandwidth bottleneck, not the devices. According to the results from Chapter 6, one device adds an average overhead of around 600 bit/s. That is one millionth of the theoretical max of the network. In real life, however, the messages are not divided evenly in time, which would increase the limitations a bit, but it is still far from being more data than a network could handle.

7.2 Linux Kernel Modules

The usage of LKMs was probably not the best option to implement this kind of algorithm. It is much too complex to work with compared to developing an application in user space due to its limited documentation, difficult initial setup, and slow debugging. It is also easy to introduce bugs in the code that can have disastrous consequences since the code is running in kernel space.

The biggest con of developing a kernel networking application is the risk of introducing security vulnerabilities. They risk becoming devastating for the user if some malicious actor finds the vulnerability and decides to exploit it. Because it is being run in kernel space, such an event would mean that the bad actor could potentially gain full control over the device and access all files kept on disk and all programs' memory.

Not only could any vulnerabilities become very dangerous, but they are also relatively easy to introduce because of the unsafe nature of the C language. It has no built-in memory safety, meaning that it is up to the developer to make sure any allocated memory is being released to avoid memory leaks and that buffer overflows do not happen. An alternative to using the C language could be to use a memory-safe language such as Rust. It uses techniques at compile-time to ensure that the code guarantees memory-, type-, and thread safety. There are people currently working on adding Rust support to the Linux kernel, allowing future developers to develop modules in a safer language, mitigating several high-risk vulnerabilities.

The general knowledge of how to develop kernel modules can be assumed not to be as widespread as the knowledge of creating similar applications using more conventional and simpler tools, programming languages, and frameworks. Also, even though there is a lot of good documentation on how to get started with kernel modules and some basic examples of how they might look, many more resources are available for creating typical user space applications, simplifying both the development and the maintainability. To make kernel programming even more complicated, it is not possible to use the libraries one might be used to, such as the standard C library. Instead, the developer must write much of the basic code from scratch.

One benefit of implementing an application in kernel space instead of user space is how much control it can have over the host machine. It could have access to all other processes running, all network packets entering the machine, and modify all parts of the system. This could be used to analyze the network traffic to detect malicious activity, implement new network protocols, and much more. If this protocol is used to transfer new kernel module versions, there are very few limitations to what it can be used for.

Another benefit of implementing this application as a kernel module is the possibility of creating a minimalistic system. Since there are no dependencies other than the kernel itself, it is possible to remove most of the user space applications and services and run a system with almost nothing but the kernel. This would speed up boot time, reduce the processing required to keep the system alive, and thus, possibly also reduce energy consumption of the device, which is an important aspect of such devices.

7.3 Future Work

The resulting algorithm presented in this report lays a foundation for automatic configuration sharing in P2P IoT networks. While this thesis has presented a working algorithm to automatically configure a new device without any prior knowledge of the device, there are still things to improve before the algorithm could be a viable replacement in the industry. This section explores future work that needs to be done to realize an adaptation of the algorithm, as well as ideas that could be implemented to optimize and improve the overall offering of the algorithm.

7.3.1 Authentication & Encryption

Regarding security, the most essential addition to this algorithm would be the authentication of new devices and encryption of traffic between nodes in a P2P ring. By seeing the unencrypted traffic between nodes, it would be possible for this malicious device to mimic the structure of messages, impersonate other devices or gain access to sensitive information in the configuration data. Because of the lack of encryption and the fact that there is no authentication in the algorithm's current state, if a malicious device gains access to the network where the algorithm is present, said device could theoretically join the P2P ring and spread a malicious configuration.

Certificates or hardware keys could be employed to authenticate new devices. A solution based on hardware keys could be implemented with Intel Secure Device Onboard [25], an offering from Intel supporting the use of a unique hardware key for each device paired with a digital key the customer acquires upon purchase of a device. An idea could be to somehow store the digital keys of bought devices in the P2P network and use them to authenticate a new device when a HELLO message is received. Another solution is to use something similar to the patent based on certificates held by Amazon Technologies mentioned in Chapter 3.

To add encryption to network traffic between devices it could be possible to implement IPSec or TLS since these are supported in the Linux kernel. Standard TLS would not work for the HELLO messages since those are datagrams but it could be implemented to avoid leaking information during configuration sharing, which is sent over TCP.

7.3.2 Heterogeneous Networks

In a real-life industry setting, devices will likely require different configurations because of several factors, e.g., they may not be identical or need to perform different tasks. Therefore, a logical step in the future development of this algorithm would be to investigate ways of adding functionality that can handle multiple different configurations in the same P2P ring and distribute these configurations according to some predetermined pattern.

A few things need to be considered to implement this. Firstly, to know what configuration is necessary for a specific device, it must be possible to identify each device uniquely. Identifying devices can be quite challenging to do automatically, given that the goal is for the algorithm to work without any previous knowledge of a new device. A possibility could be to look into location-based configurations, i.e., defining that devices within a certain area should have a specific configuration. However, this would only enable a distinction based on their location within a factory, not the type of devices. To allow both distinctions to be made, investigating a combination of location-based configuration and making use of the fact that the solutions for authentication presented in 7.3.1 come with the added benefit of being able to identify devices uniquely. Secondly, for heterogeneous networks to become a reality a P2P ring must facilitate storing different configurations. A possibility could be to use the full Chord protocol instead of the simplified version used in this thesis.

The Chord protocol could be a great option as it introduces efficient searching for stored configurations through the use of a finger table and fault tolerance by storing a node's configuration at its successors in its successor list.

7.3.3 Tailor-Made Linux Distribution

The operating system on which these modules were inserted and tested on was pre-built by the creators of the hardware. It comes pre-installed with lots of programs and services that simplify the usage of the device for a normal (non-expert) user. However, this slows down the device in terms of both boot time and run time.

One change that would benefit the performance the most would be to use a custom Linux distribution tailor-made for the specific use cases of these sensors. The user space part of the distribution could be made as minimalistic as possible, using something like Busybox, which is a compiled binary, combining many of the common utilities that are often found in Linux-based systems [26]. It is optimized for size and minimal usage of resources, which makes it a great candidate for resource-limited devices such as the ones mentioned in this thesis.

When using something like Busybox running on the device, the algorithm could fully utilize the fact that it is implemented as a kernel module and, thus, does not require any additional user-space applications or libraries to function. This would both speed up the boot time and reduce the energy consumption, as fewer computations are required to keep the system up and running.

8

Conclusion

In conclusion, this thesis has presented the development, implementation, and testing of a novel algorithm enabling automatic configuration of IoT devices. The testing performed showed that the algorithm is both fast and efficient. The algorithm also proved scalable as there was a relatively small increase in TTC when increasing the number of devices.

The algorithm can potentially remove the need for manual configuration of IoT devices altogether in some instances by providing a fast and efficient configuration process. With its ability to configure a new device joining an existing P2P network in as little as 18 milliseconds, the algorithm far outperforms what could realistically be achieved when manually configuring a new device.

However, the algorithm comes with its limitations. Since security and heterogeneous networks were deemed out of the scope, it would not be realistic to think that the algorithm could replace any manual process in its current state. For it to become a viable replacement, future research is necessary.

It is suggested that future research should explore security topics such as authentication and encryption and heterogenous networks to allow for multiple different configurations and devices to exist in the same network. Investigating a tailor-made Linux distribution could also be beneficial in making the complete offering as lightweight as possible.

In summary, this algorithm is a strong proof-of-concept of automatic configuration using knowledge-sharing and a P2P network architecture. It shows that this type of automatic configuration algorithm could be a promising solution to the problem of manual configuration, as it is both fast and efficient. However, it is yet to be a viable replacement. Still, with more research on the security aspects and support of heterogenous networks, the algorithm could replace the manual configuration process some of Combitech AB's customers use today.

Bibliography

[1] IBM, *What is industry 4.0?* https://www.ibm.com/topics/industry-4-0, Accessed: 2024-02-06.

[2] V. P. Gupta, "Smart sensors and industrial iot (iiot): A driver of the growth of industry 4.0," in *Smart Sensors for Industrial Internet of Things: Challenges, Solutions and Applications*, D. Gupta, V. Hugo C. de Albuquerque, A. Khanna, and P. L. Mehta, Eds. Cham: Springer International Publishing, 2021, pp. 37–49, ISBN: 978-3-030-52624-5. DOI: 10.1007/978-3-030-52624-5_3. [Online]. Available: https://doi.org/10.1007/978-3-030-52624-5_3.

[3] M. Weiser, "The computer for the 21st century," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, Jul. 1999, ISSN: 1559-1662. DOI: 10.1145/329124.329126. [Online]. Available: https://doi.org/10.1145/329124.329126.

[4] A. Sunyaev, *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. Jan. 2020, ISBN: 978-3-030-34956-1. DOI: 10.1007/978-3-030-34957-8.

[5] IBM, *What is the internet of things?* https://www.ibm.com/topics/internet-of-things, Accessed: 2024-02-09.

[6] P. Sethi and S. Sarangi, "Internet of things: Architectures, protocols, and applications," *Journal of Electrical and Computer Engineering*, vol. 2017, pp. 1–25, Jan. 2017. DOI: 10.1155/2017/9324035.

[7] I. Stoica, R. Morris, D. Liben-Nowell, *et al.*, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003. DOI: 10.1109/TNET.2002.808407.

[8] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proceedings First International Conference on Peer-to-Peer Computing*, 2001, pp. 101–102. DOI: 10.1109/P2P.2001.990434.

[9] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek, and A. Rowstron, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65, ISBN: 978-3-540-45748-0.

[10] E. Foundation, *2023 iot & edge developer survey report*, Accessed: 2024-05-15, 2023. [Online]. Available: https://5413615.fs1.hubspotusercontent-na1.net/hubfs/5413615/Eclipse%20IoT%20White%20Papers%20and%20Case%20Studies/2023%20IoT%20&%20Edge%20Developer%20Survey%20Report.pdf.

[11] *What is linux*, https://www.linux.com/what-is-linux/, Accessed: 2024-02-09.
[12] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Boston, MA: Pearson, 2014, ISBN: 978-0-13-359162-0.
[13] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, 10th Edition*. Wiley, 2018, ISBN: 978-1-118-06333-0. [Online]. Available: http://os-book.com/OS10/index.html.
[14] T. L. Foundation, *The linux kernel archives*, https://www.kernel.org/, Accessed: 2024-05-27, 2024.
[15] R. P. (Ltd, *Raspberry pi zero 2 w*, Accessed: 2024-03-19. [Online]. Available: https://datasheets.raspberrypi.com/rpizero2/raspberry-pi-zero-2-w-product-brief.pdf.
[16] S. Madsen, A. Santos, and B. Jørgensen, "A qr code based framework for auto-configuration of iot sensor networks in buildings," *Energy Informatics*, vol. 4, p. 46, Sep. 2021. DOI: 10.1186/s42162-021-00152-w.
[17] I. Chatzigiannakis, H. Hasemann, M. Karnstedt, *et al.*, "True self-configuration for the iot," in *2012 3rd IEEE International Conference on the Internet of Things*, 2012, pp. 9–15. DOI: 10.1109/IOT.2012.6402298.
[18] G. Reinhart, S. Krug, S. Hüttner, Z. Mari, F. Riedelbauch, and M. Schlögel, "Automatic configuration (plug & produce) of industrial ethernet networks," in *2010 9th IEEE/IAS International Conference on Industry Applications - INDUSCON 2010*, 2010, pp. 1–6. DOI: 10.1109/INDUSCON.2010.5739892.
[19] I. Boskov, H. Yetgin, M. Vunik, C. Fortuna, and M. Mohori, "Time-to-provision evaluation of iot devices using automated zero-touch provisioning," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–7. DOI: 10.1109/GLOBECOM42002.2020.9348119.
[20] R. Loladia, R. Bhattacharyya, A. Thakur, and A. S. Beheray, "Zero-touch provisioning of iot devices with multi-factor authentication," U.S. Patent 10447683B1, Oct. 2019. [Online]. Available: https://patents.google.com/patent/US10447683B1/en.
[21] *The linux kernel module programming guide*, Accessed: 2024-02-09, 2024. [Online]. Available: https://sysprog21.github.io/lkmpg/.
[22] *Qemu*, https://wiki.qemu.org/, Accessed: 2024-02-09.
[23] T. L. Foundation, *Rmmod(8) - linux manual page*, Accessed: 2024-03-21, 2024. [Online]. Available: https://man7.org/linux/man-pages/man8/rmmod.8.html.
[24] W. Foundation, *Wireshark: Network protocol analyzer*, https://www.wireshark.org/, Accessed: 2024-05-27, 2024.
[25] Intel, *Intel Secure Device Onboard (Intel SDO)*, Accessed: 2024-05-23. [Online]. Available: https://www.intel.com/content/www/us/en/internet-of-things/secure-device-onboard.html.
[26] *About busybox*, https://busybox.net/about.html, Accessed: 2024-05-27, 2024.