Benchmarking Resource Usage of Blockchain Clients

Master Thesis

Aalborg University
Software, 10 Semester

Software
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Benchmarking Resource Usage of Blockchain Clients

Theme:
Master Thesis

Project Period:
Spring Semester 2023

Project Group:
CS-23-DS-10-11

Participant(s):
Daniel Friis Holtebo
Jeppe Krogh Laursen

Supervisor(s):
Michele Albano, mialb@cs.aau.dk
Daniele Dell'Aglio, dade@cs.aau.dk

Copies: 1

Page Numbers: 94

Date of Completion:
June 8, 2023

Abstract:

Blockchain benchmarking is often based on measuring the transaction throughput of a blockchain. This paper seeks to quantify blockchain performance by focusing on computational resource usage. The same problem has been addressed in the authors' prior work through custom blockchain implementations. This paper will focus on off-the-shelf components and known blockchains like Bitcoin and Ethereum. The proposed solution to the problem of benchmarking blockchain clients' resource usage will use industry standard software, like Docker, combined with other off-the-shelf software to create private networks. The implementation and benchmarking of blockchain clients conducted in this work creates a proof-of-concept for using off-the-shelf software for benchmarking blockchain clients, with more work to be done in the future.

Project Summary

Since the launch of Ethereum in 2015, interest in blockchain has grown, particularly with the cryptocurrency market reaching its peak in 2021. Energy consumption is a crucial factor to consider in blockchain networks, and it is influenced by both the consensus mechanisms and the specific implementation of blockchain clients. This paper builds upon previous research that aimed to benchmark custom implementations of blockchain consensus mechanisms, whereas this work focuses on benchmarking existing software for real-world blockchain clients and off-the-shelf solutions for orchestration and monitoring.

A stakeholder analysis is conducted to identify potential individuals and companies with an interest in a product for benchmarking blockchains. It is determined that such a product would be interesting for existing blockchain users for them to identify optimal blockchain clients. Furthermore, it is useful for blockchain developers to assist in pinpointing potential areas of improvement for their implementation. Following the stakeholder analysis is an overview of blockchains as a distributed ledger technology, often associated with cryptocurrencies. The overview provides a description of how the distributed ledger functions and discusses some areas of interest for blockchains. After this, consensus mechanisms are explained in more detail, with examples for Proof of Work, Proof of Stake, Proof of Elapsed Time, and Practical Byzantine Fault Tolerance. An analysis of the nodes of a blockchain network is presented, where the relationship between blockchain clients and the blockchain's consensus mechanism is covered. Following this, a detailed description of different types of blockchain network nodes is provided for full nodes, archival nodes, light nodes, authority nodes, miner nodes, and staking nodes. These node types are typically present in blockchain networks, with some types being exclusively used in certain blockchains depending on their consensus mechanism. Related work is discussed next, where it is found that most related work in this area of research centers around the transaction throughput of blockchains and not the computational resources used by different blockchain clients. As such, the focus for the paper is narrowed with the following problem statement: How can we benchmark resource usage in popular blockchain clients inside a private network?

After narrowing the scope, design of a system to satisfy the problem statement begins. First, several blockchains are considered as potential candidates for benchmarking. These are blockchains from popular cryptocurrencies, based on their market cap and blockchains used in the development of distributed applications. This discussion is rounded out with selecting three blockchains with available clients. Orchestration tools are then discussed, where containerization

is chosen to maintain the nodes used in the private network, with Docker as the container manager. Following this, resource monitoring is discussed. The concept of monitoring resource usage is first introduced, after which existing solutions are presented. Here, Glances is chosen as it provides a container mode, specifically for monitoring Docker containers. After picking a resource monitoring solution, the system's overall architecture is presented. The system will consist of some containers acting as network nodes for the private blockchain network, with Glances as an external agent to monitor the resource consumption of the containers.

Having completed the system design, an example implementation is presented. It starts with setting up Glances and ensuring the monitoring data is logged properly, where a sorting step is introduced to assist in processing the data. Then the three blockchain networks are configured for testing. Some networks failed to reach a working state, which meant that these blockchain networks were not benchmarked as intended. The Ethereum networks are, however, fully functional and are used in the system test, to provide proof-of-concept data to show the viability of the solution. To test the system, a server is provided by Aalborg University with enough processing power to maintain a Proof of Work network with 25 nodes. The chosen Ethereum client, Geth, provides two different blockchain networks with their own consensus mechanism in each. One is the Proof of Work algorithm Ethash and the other is the Proof of Authority algorithm Clique. Both algorithms are benchmarked, and their respective data is compiled into a pair of tables, with accompanying graphs. Comparing the results, shows that Proof of Authority has a significantly lower CPU consumption, but with a memory consumption that grows faster than the Proof of Work, while almost having the same network usage throughout the benchmarking tests.

It is concluded that the designed system, with its example implementation, achieves the goal of the problem statement. The system acts as a proof-of-concept for a future solution for benchmarking blockchain clients resource usage and provides a more informed discussion surrounding the efficiency of blockchain clients and blockchain networks.

Preface

Aalborg University, June 8, 2023

Daniel Friis Holtebo
<dholte18@student.aau.dk>

Jeppe Krogh Laursen
<jlau18@student.aau.dk>

Foreword

During the spring semester of 2023, our team, *CS-23-DS-10-11* from Aalborg University, dedicated our efforts to this project from February 1st to June 9th, 2023. We would like to thank our project supervisors, Michele Albano and Daniele Dell'Aglio, for their guidance and support throughout this project.

Reading Guide

This report follows the Vancouver citing and referencing method, utilizing square brackets with a numbered citation system. The corresponding numbers in the text reference entries listed sequentially towards the end of the report, on page 65, where full citation information is provided [1]. Citations are generally placed in proximity to the referenced information in the text, but if placed at the end of a paragraph, they reference the entire paragraph.

Situated on the next page, page *vii*, is a table of contents of the report where an overview is available.

Contents

Chapter 1

Introduction

In research, interest in blockchain has increased since the launch of Ethereum in 2015 [2]. Further, during 2021 the cryptocurrency market reached its all-time highest global market cap [3, 4]. The popularity of cryptocurrencies, and their respective blockchains, raises the question, what is the cost? The answer is energy. To add data to a blockchain, a notion of consensus is required. This consensus is known as a consensus mechanism. One such mechanism is called Proof-of-Work (PoW). The PoW consensus mechanism, relies on miners solving a cryptographic puzzle to trust new blocks. As such, PoW consumes a lot of energy to provide the security and reliability expected of a blockchain. For example, Bitcoin, which utilizes PoW, is estimated at an annualized energy consumption of more than 130 terawatt hours [5]. Another popular consensus mechanism is Proof-of-Stake (PoS). PoS uses a staking system, in which participants stake an amount of cryptocurrency for a chance to add a new block to the blockchain. Ethereum utilizes PoS and is estimated at 6 gigawatt hours in annualized consumption [5]. Annualized consumptions are as per May 2023.

This small comparison, clearly shows the difference between using a PoW-based blockchain and a PoS-based blockchain. However, a blockchain's overall energy consumption does not describe the blockchain clients' resource consumption. An important aspect for some blockchain participants are the minimum specifications required to participate in a given blockchain. For example, Ethereum recommends a quad-core CPU, minimum 16 GB of RAM, at least 2 TB of storage, and a stable internet connection with at least 25 Mbps download speed [6].

This paper continues the work of our previous paper [7] in benchmarking blockchain clients and their respective consensus mechanism. In the previous work, we set out to investigate the utility of blockchain technology in local energy markets. Local energy markets are markets created by smaller communities of energy consumers and prosumers, consumers who also produce energy. We found that these local markets could benefit from using blockchains to store transactional

data, to remove the reliance on a trusted third party. However, while working on the previous paper, we investigated tools for benchmarking blockchains and found a lack in research. Most papers investigate the performance of blockchains either by analytically analyzing their algorithms to describe a blockchain's relative performance, or focus entirely on how many transactions or blocks a blockchain can handle per time unit.

As a result, in our previous paper, we decided to investigate how to design and implement our own solution. We achieved this by making an implementation of a PoW and Raft blockchain client, which we then monitored to log data of its computational resource usages.

In retrospect, we noted the inefficiencies and inaccuracies compared to real-world blockchain clients of implementing blockchain clients ourselves. Therefore, in this paper we investigate existing software and off-the-shelf solutions for orchestration and monitoring, to improve on our concept of a blockchain client benchmarking solution. This eliminates the result uncertainty compared to our previous work, as existing blockchains are actively used.

1.1 Initial problem statement

The introduction above leads to the following initial problem statement:

How can we identify what blockchains have high resource consumption?

Chapter 2

Problem Analysis

In the following chapter, Section 2.1, conducts a stakeholder analysis. Next, Section 2.2 summarizes relevant topics from the problem analysis of our previous paper[7], as this paper expands on the same analysis, and Section 2.5 investigates related work to round out the chapter.

2.1 Stakeholder Analysis

This section covers the Stakeholder Analysis, identifying and categorizing individuals and organizations that may have an interest in this project's work. Stakeholders are grouped based on their needs and wants, and their most pressing issues are analyzed to determine which ones the product can address.

The stakeholder analysis consists of three steps: identifying all stakeholders, categorizing them according to their level of power and interest in the project, and gaining an understanding of their needs to determine potential investment opportunities [8].

Given that this project focuses on blockchains and on an analysis of their resource consumption, the goal of this analysis is to try to understand stakeholders and what they value, so the project's work has the highest chance of success and be as valuable as possible.

2.1.1 Identification

The first step is to identify possible stakeholders. The following list shows what possible stakeholders this project has, note that the list has already been refined, and they are each clarified further below.

- New potential blockchain users
- Existing blockchain users
- Blockchain developers
- Mining/Staking organizations

New potential blockchain users would have a big interest in this project, as they are looking into joining a blockchain network, but not knowing what each blockchain requires specifically in terms of hardware. If a potential new user has expensive energy costs or bad internet, this project may help the user to choose the most optimal blockchain network to participate in.

Existing blockchain users would have a potential interest in the success of this project, as they are typically already running one or several blockchain clients. This project could help them realize hardware bottlenecks and how the resource usages increase as more participants join the network in the future.

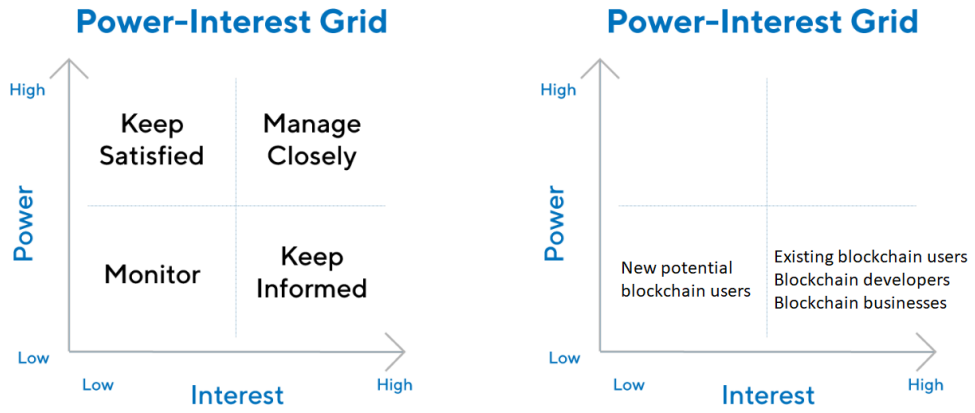
Blockchain developers could have a special interest in knowing what kind of ordinary resource usages their blockchain clients and networks have, while also assisting them in improving existing software or developing new software.

Mining/Staking organizations operate by creating large mining or validation farms where they pool their resources together, as their primary revenue stream. These

organizations would have an interest in knowing which blockchain client they can utilize to optimize their profit margin.

2.1.2 Prioritization

To assist in prioritizing the identified stakeholders, a Power/Interest grid is used. This grid consists of four quadrants, as shown in Figure 2.1a. From left to right, top to bottom, the stakeholders are positioned as either high power and low interest, high power and high interest, low power and low interest, and low power and high interest. These classifications determine how much power the relative stakeholders are given over the end product and how interested the stakeholder is in the success of the project.



(a) The Power Interest Grid. [8].

(b) The filled Power Interest Grid with stakeholders.

As shown in Figure 2.1b, none of the identified stakeholders are given high power over the project. However, they are differentiated on their interest level. New users are classified as being of low interest. This is due to their newcomer status in providing resources to blockchain networks. The remaining stakeholders are classified as low power and high interest: these stakeholders are already invested in blockchain networks, and as such, interested in comparing blockchains and blockchain clients based on their relative resource usage. However, even though they are highly interested, they do not hold much power over the product, as it is created independently of any business or company and without economic incentives from individuals.

2.1.3 Understanding

Most blockchain participants, new and existing alike, are motivated by a financial incentive to increase their profit margins and reduce operating expenses. Blockchain

developers stand out from this, as they can be motivated by other factors, such as the desire to improve particular aspects of blockchains. For example, developers could seek to make blockchain solutions more environmentally friendly. This goal could be based on minimizing their blockchain's CPU resource usage.

Since the stakeholders more likely are motivated by a financial incentive, the project should set out to evaluate as many blockchains as possible, and perform the evaluation using the actual software used by the stakeholders. Operating blockchains are mainly expensive in terms of electricity upkeep, so it would be important for this project to successfully, and accurately, monitor CPU, memory, disk storage, and network resource usages, seen from an "expenses only" perspective.

2.1.4 Summary

After the Stakeholder Analysis has been conducted, a *stakeholder communication plan* should be developed such that potential investments could be facilitated properly [8]. But as this is an academic project at Aalborg University, funding is not needed. However, if this project was done in an ordinary business or startup, potential investments are, typically, of great desire, and would be pursued further.

This analysis provides insight, as the needs and expectations of all relevant stakeholders should be considered, so the intended goal and objectives of the project can be met as closely as possible. From the analysis, we can conclude real-world software is important, as it is what the stakeholders use and provides the foundation for getting realistic performance data from the blockchain clients. To satisfy as many stakeholders as possible, we should aim for benchmarking as many blockchain systems as possible, to try to get as broad of an outreach as possible as well. Finally, resource usages such as CPU, memory, disk storage, and network metrics are important to benchmark in those systems.

2.2 Blockchains

A blockchain is a secure and transparent digital ledger that can record various types of information. It can track physical items, commodities, and intellectual properties. It is typically distributed across a network with multiple clients ranging much in size, which makes it more secure and cost-effective. Members of a blockchain share the same data view, and users can access stored data as long as they have an internet connection [7].

Blockchains provide all network participants access to the same immutable content. Each piece of data is registered only once, eliminating any duplication in the chain. Additionally, once data is added to the blockchain, it cannot be altered or tampered with by any participant. If any faulty data is added, new correct data

must be added to the blockchain to correct the error. Some blockchain make use of smart contracts. Smart contracts are pieces of software, stored on the blockchain and available to be executed upon request [7].

When new data is added to the blockchain, it is recorded as a new block containing any type of information. Each block is directly linked to the previous one, forming a chain that describes the data over time. If the blockchain is used to store digital currency, the blocks record how the currency moves and changes ownership, along with the exact time and sequence of transactions. The blocks are securely linked, making it impossible to insert new blocks in between existing ones. As more blocks are added, the trust and verification of each block increases, ensuring the immutability of the blockchain. This makes it extremely difficult for anyone to change any previously added blocks [7].

Centralized systems often require third-party validation and duplicate record-keeping, making them vulnerable to malicious intent and cyberattacks. The advent of the Internet of Things (IoT) has led to an explosion in transaction volumes, making blockchains an attractive solution. Blockchains offer greater trust, as network participants can trust new blocks to accurately and timely record data. They also provide greater security, as consensus on data accuracy is required from network members, and the data is immutable. In addition, the blockchain's distributed ledger technology and shared network eliminate the need for record reconciliation, except in the case of forks. Smart contracts can boost transaction speed by being stored on the blockchain and automatically executed when a new block is added [7].

There are four commonly known types of blockchain networks: public, private, permissioned/hybrid, and consortium. Public networks are open to anyone, while private networks have selected participants and are typically managed by an organization. Permissioned/hybrid networks place restrictions on participants, and consortium networks are often shared by multiple organizations with no single entity in control. Each type has its benefits and use cases, such as high transaction speed for private networks and flexibility for consortium networks [7].

2.3 Consensus Mechanisms

Consensus mechanisms ensure that all data on a decentralized blockchain network is genuine by obtaining security, trust, and agreement across the system. They are sets of rules that data contributions must go through to reach legitimacy. Consensus Mechanisms should be fault-tolerant and continue operating without interruption even if they encounter errors or attacks. For example, the popular cryptocurrency Bitcoin uses the PoW consensus mechanism [7].

In centralized systems, administrators typically have authority over system maintenance and data, whereas ordinary users of the system cannot perform such

tasks. In contrast, decentralized systems, such as blockchains, are self-regulating and rely on participants to contribute to data verification and authentication without any single authority. This is made possible by consensus mechanisms [7].

2.3.1 Examples of Consensus Mechanisms

Many versions of consensus mechanisms exist. In our previous work [7], we explored several mechanisms. The following list highlights a few select consensus mechanisms and offers a short explanation of how they operate.

Proof of Work

Proof of Work (PoW) is a consensus mechanism used in Bitcoin, where clients compete to solve complex cryptographic puzzles and append new blocks. The puzzle difficulty adjusts based on solving speed, and winners earn rewards such as cryptocurrency. While PoW is considered secure, it poses challenges due to its resource-intensive nature, including high participation costs, the potential for centralization through resource pooling, and significant environmental impact from energy consumption.

Proof of Stake

Compared to PoW, Proof of Stake (PoS) is an energy-efficient consensus mechanism that relies on validators staking a specific amount of cryptocurrency as collateral. Validators are chosen to create new blocks based on their stake, and their staked currency can be destroyed if they misbehave. Ethereum is an example of a blockchain that utilizes PoS, requiring a stake of 32 Ether to participate as a validator.

Proof of Elapsed Time

Proof of Elapsed Time (PoET) is a consensus mechanism where players wait a random amount of time instead of using computational power. It consists of an initialization phase, creating a genesis block, and a leader election phase where players compete to generate the next block. A Trusted Execution Environment (TEE) is used to obfuscate block creation and manage the timer. Collisions are resolved by comparing timeframes and chain lengths.

Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is a consensus mechanism derived from the Byzantine Generals Problem. It achieves consensus by having nodes participate in a voting process, with consensus reached when two-thirds agree on a new block.

PBFT does not require significant computational resources or staking. In a PBFT-enabled system, nodes are sequentially ordered as primary or secondary nodes. When a client makes a request, the primary node broadcasts it to secondary nodes, which respond to achieve success when enough non-faulty nodes provide the same result. PBFT offers benefits like low computational overhead, low latency, and high throughput, but it has high network overhead and scalability challenges.

2.4 Blockchain Node

In general, a client is a tool or piece of software that permits connection to a server in either a client-server setup or a peer-to-peer environment. This allows users to remotely communicate with other entities over a network. In blockchain technology, clients are utilized to join and interact with the blockchain network and other clients. Blockchain wallets facilitate other tasks such as, receiving, sending, and storing cryptocurrency coins and tokens. A blockchain node, however, is the combination of all three, meaning that a typical blockchain node has the capabilities of a consensus mechanism, a blockchain client, and of a blockchain wallet. Blockchain nodes serve various applications, one of which are miners, where the nodes handle the mining operation and transfer mined hashes to the rest of the blockchain network. Additionally, block explorers provide an easily readable format to access blockchain data, including block height, hash rate, transaction volume, and individual transactions. Wallets, and some nodes, usually have user-friendly graphical user interfaces (GUIs) that even inexperienced users can use to manage their crypto funds with ease [9].

2.4.1 Blockchain Clients and Consensus Mechanisms

Consensus mechanisms and blockchain clients, introduced in Section 2.3 and Section 2.4 respectively, are two different concepts in the context of blockchain technology. They are connected in that a consensus mechanism dictates the rules for how transactions and blocks are validated and added to the blockchain, while a blockchain client is used to send and receive transactions and blocks that conform to those rules. For example, if a blockchain network is using the Proof-of-Work consensus mechanism, nodes should respect the rules for solving cryptographic puzzles to validate transactions and add them to the blockchain. Without a consensus mechanism, blockchain nodes would have no way to ensure that transactions are valid and agreed upon by all nodes in the network. In summary, blockchain clients rely on the consensus mechanism to ensure the validity of transactions and maintain the integrity of the blockchain and its data.

2.4.2 Blockchain Node Types

To ensure the correctness of the blockchain and prevent malicious activity, each blockchain node typically stores its own copy of the blockchain, or parts of it, and tracks incoming transactions. However, the storage and computing requirements can be a barrier for new users, which is why many node types exist to securely synchronize with the blockchain. Some of the more well-known types are described here.

- **Full node** - Full nodes are the most standard type of blockchain node, and as its name implies, store all the blockchain's data. The node helps with verifying network rules, such as validating blocks, receiving and verifying transactions, and provides access to its complete collection of data to the rest of the network. Full nodes store the blockchains latest state, which includes all the network's users and is often represented in a type of Merkle tree [10]. Nodes adhere to a formal specification, but a network can be open to many implementations. Full nodes keep track of a lot of data and use large amounts of storage and bandwidth. An instance of this is that by early October 2020, the Bitcoin blockchain measured approximately 300 GB, while the Ethereum blockchain occupied around 500 GB. Nodes should always be available to the rest of the network, even though there generally isn't any economic incentive to run a full node, which is why many who do run them, are businesses like exchanges. Full nodes play a crucial role in securing the network by verifying all transactions and notifying other clients of invalid blocks. Some client types rely on connecting with full nodes to access the blockchain as well. Additionally, running a full node can be incentivized in certain networks. Celo [11] is an example of a blockchain that provides economic incentives for running non-validating full nodes, allowing individuals to set gateway fees for answering requests and forwarding transactions [12].

- **Archive (or archival) node** - Archival nodes store everything that a full node does, along with archives of historical states of the chain. While they do not offer any additional validation or security compared to full nodes, they require significantly more storage. As of October 2020, Ethereum's archival nodes measured over 5.3 TB of data usage and took around two weeks to synchronize. Due to the high resource requirements, very few archival nodes are operated on networks, but are usually run by block explorers, data analytics firms, and infrastructure providers [12].

- **Light node** - Running a full node requires significant storage and uptime, which is why most users prefer not to run them. Instead, light clients are used to provide high security and low computing power requirements, improving the accessibility of blockchain networks for resource-constrained de-

vices. Light clients allow users to securely synchronize with a blockchain without storing the entire blockchain. They can be used to check the state of an account, confirm a transaction, or watch for events. Light clients download and verify block headers and request other relevant information from full nodes as needed. While light clients do not require constant running, they need to be connected to full nodes to request data and interact with the blockchain. They are suitable for low-capacity users, such as those using smartphones or browser extensions, providing high security assurance about the state of a chain. Light clients do not write data to the network but make blockchains accessible to a wider range of users [12].

- **Authority node** - To participate in a public blockchain, a user downloads a client and synchronizes with the network. In private and some partially centralized blockchains, access can be restricted to a few authority nodes, which then can control and limit access for other nodes [13].

- **Miner node** - The Proof of Work consensus mechanism, used in blockchains like Bitcoin, relies on miner nodes to validate transactions by solving a complex mathematical puzzle. This process demands high computational power and has significant energy consumption. Once the miner node solves a puzzle and adds a new block to the blockchain, it is rewarded with newly minted tokens as an incentive for its contribution to the network [13].

- **Staking node** - Staking nodes are used to verify transaction validity in blockchains that use the Proof of Stake consensus mechanism. To establish a staking node, users need to have an amount of native tokens and lock it on the blockchain. The system then randomly selects a staking node to process transactions and records them on the ledger, based on some predefined rules. These rules could be the amount of locked funds or time spent on the blockchain. Compared to miner nodes, staking nodes require much less energy to validate transactions and add blocks [13].

2.5 Related Work

This section presents the related work, which focuses on measuring blockchains and their clients' resource usage. The goal is to identify existing solutions, and gain a more in-depth understanding of the problem space as described in the initial problem statement 1.1. Through a comprehensive review of the literature and analysis of existing solutions, including their strengths, weaknesses, and gaps, it may also be possible to identify new innovative solutions to effectively address the problem described in the initial problem statement.

Gromit: Benchmarking the Performance and Scalability of Blockchain Systems

Gromit is a generic framework for analyzing a blockchain solution, and has been proposed by Nasrulin et al. in [14]. The framework has been utilized to conduct some of the biggest blockchain studies to date, according to the authors, when it was published in 2022. Gromit can benchmark several performance metrics in blockchain networks such as transaction throughput, scalability, and network usage. The work involved the testing of seven representative blockchain systems. The blockchain system is benchmarked by viewing them as a transactional processing system, where an arbitrary number of clients consistently reach consensus on the submitted transactions.

In addition to the performance metrics, Gromit can output CPU and network utilization of every client in each blockchain system. The resource usages are however in relation to the blockchain network's processed transactions per second.

Hyperledger Caliper

Another existing framework which specializes in benchmarking blockchains is Hyperledger Caliper [15]. Hyperledger Caliper is presented as a "blockchain performance benchmark framework" [15] and allows its user to benchmark a limited set of supported blockchain networks: Ethereum, Hyperledger Besu, FISCO BCOS, and Hyperledger Fabric.

Hyperledger Caliper can track the following performance metrics of a blockchain, "Transaction/read throughput, Transaction/read latency (minimum, maximum, average, percentile), and Resource consumption (CPU, Memory, Network IO, . . .)" [16]. Furthermore, the framework offers two types of monitoring, Resource and Transaction monitoring, both collecting statistics during benchmarking of the System Under Test (SUT). The Resource monitoring type can be further specified in the configuration files, where either the Process, Docker, or Prometheus monitoring modules are available. The process module can monitor a specific named process on the host system, while the Docker module monitors a specific Docker container via the Docker Remote API. The Prometheus module utilizes the open-source monitoring framework, also called Prometheus [17], which scrapes and then stores monitoring data in a time series database for future query. Configuration files must be created for each blockchain network that the user wants to test. A blockchain is specified in the configuration files, and if that blockchain and its specific version is supported, Caliper handles everything and runs the network while it performs the benchmarking tests. Users can also use Caliper for non-supported blockchains, but need to implement and specify their own Connector, which can be difficult [16].

BCTMark: a framework for benchmarking blockchain technologies

Saingre et al. proposes in [18] a generic framework for benchmarking blockchain systems called BCTMark. Their work focuses on reproducibility and portability, which is why the authors tested their framework on two different test beds, a cluster of Dell PowerEdge R630 servers and a cluster of Raspberry Pi 3+'s. By focusing on these two subjects, the authors argue that their framework, and thus blockchain testing, can be repeated in different environments as long as the test bed supports the SSH protocol. Their tests were conducted with three different blockchain systems: Hyperledger Fabric, Ethereum Clique, and Ethereum Ethash. BCTMark can generate performance data based on blockchain system's energy footprint, latency, throughput, and CPU usage while accommodating varying numbers of clients. All the tests that the authors conducted were in relation to a load generation of x transactions per second on a constant number of clients [18].

BLOCKBENCH: A Framework for Analyzing Private Blockchains

In [19] Dinh et al. propose BLOCKBENCH, a framework for assessing private blockchain systems. The authors state that any blockchain system can be integrated with BLOCKBENCH, simply by connecting with its API. The framework can monitor either component-wise or overall, focusing on monitoring fault-tolerance, throughput, scalability, and latency. The framework has been based and tested on some of the most popular blockchains, namely Hyperledger Fabric, Ethereum, and Parity. These three blockchains are all able to be run in a private environment, and all support smart contracts as well. The framework's output is, as with BCTMark, in relation to a workload, typically a transaction generation load, and does not include detailed CPU, memory, or network usage data. By configuring workloads, BLOCKBENCH allows for a thorough analysis of the SUT through a range of macro and micro benchmarks. The authors conclude none of the three blockchains are suitable for processing large amounts of data at scale, as compared to more traditional database systems [19].

2.5.1 Summary

All the mentioned related works in this section can benchmark blockchains and monitor important metrics in a varying degree. The related works have all been tested on a limited set of blockchain systems, particularly Ethereum and Hyperledger blockchains. Only BCTMark focuses on portability and claims that it works with any blockchain, as long as the test bed supports the SSH protocol. Furthermore, all the work focuses, more or less, on the same benchmarking metrics, namely scalability, latency, and throughput. While these metrics are important, resource metrics such as CPU, memory, and network usages are not included, ex-

cept for Hyperledger Caliper, which can monitor those metrics by configuring the correct monitoring modules in their configuration files. The related works focus heavily on transaction throughput, and the benchmarking data presented in these are typically only in relation to the transaction throughput.

This presents an opportunity to create an open framework, which enables resource metric benchmarking of blockchain clients. By focusing on the clients rather than the blockchain as a system, it enables the possibility of comparing multiple clients for a single blockchain. This can highlight differences in implementation across clients and potentially allow developers to further improve their blockchain clients.

Chapter 3

Problem Statement

The initial problem formulation in Section 1.1 and the introduction explains how we are building upon our last work in this paper. We introduce the notion of benchmarking real-world blockchain clients instead of custom implementations, which led on to a further exploration of the problem space in the problem analysis in Chapter 2.

In the problem analysis chapter, we started off by performing a Stakeholder Analysis in Section 2.1, which helped us consider potential stakeholders for a system that benchmarks a blockchain client's resource consumption. The analysis showcased that four different stakeholders exist and to satisfy them as much as possible, the project should concern itself with the benchmarking of as many blockchain clients as possible, while benchmarking them regarding resource usages. This was followed by exploring the main concepts of blockchains, blockchain nodes, and how these are structured in Section 2.2, Section 2.3, and Section 2.4. Lastly, the chapter finished with an analysis of the related work in Section 2.5. In this analysis, an opportunity for new development was uncovered, in that most other research focuses on the transactional performance of blockchains.

Two concepts evolved from the problem analysis: First, the understanding of potential stakeholders shows that more resource consumption data is sought after for as many blockchain clients as possible. Second, that the exploration of related works illustrates a lack of frameworks and data regarding the benchmarking of blockchain clients resource consumption.

3.1 Problem statement

Based on the summarized problem analysis above, we can narrow down our focus and propose a specific objective. Our objective revolves around benchmarking the resource consumption of nodes in different blockchain networks. However, as discussed in Section 2.2, various types of blockchain networks exist. To ensure

optimal benchmarking results, we choose to conduct our resource monitoring on private blockchain networks. Private networks offer the ideal environment for benchmarking blockchain clients due to the complete control over network nodes, blocks, and transactions. Still, further investigation could be required to scale our results up to large public blockchain deployments.

With these insights, we can now present a more focused problem statement we will dedicate this project to achieve:

How can we benchmark resource usage in popular blockchain clients inside a private network?

Chapter 4

Design

This chapter will present the design considerations for creating a system which satisfies our problem statement, as presented in Chapter 3. Section 4.1 will talk about which blockchains should be compared, and Section 4.2 will talk about how blockchain clients can be compared in a controlled network. Thereafter, Section 4.3 explores potential existing solutions that can monitor a system's resource usage. Lastly, a general design of the system as a whole will be presented in Section 4.4.

4.1 Choosing Blockchains

As presented in Chapter 3, the goal of this solution is to benchmark blockchain clients in a private network. Previously, in Section 2.4, we presented what blockchain clients are and how they function in a blockchain network. In this section, we will discuss the blockchain clients we aim to benchmark.

4.1.1 Blockchains from Popular Cryptocurrencies

Looking into which clients we will benchmark, different aspects are considered. Firstly, we want to consider the clients from *popular* cryptocurrencies. To determine what blockchains are considered popular, we relied on the market caps of their different cryptocurrencies. Here, the following top ten list presents itself [20, 21, 22, 23]:

1. Bitcoin
2. Ethereum
3. Tether
4. BNB
5. USD Coin
6. XRP
7. Cardano
8. Dogecoin
9. Polygon
10. Solana

We are already familiar with Bitcoin and Ethereum, from our previous work [7]. However, most other currencies on the list are unfamiliar. Hence, here is a quick explainer of the rest:

- **Tether**
  A blockchain based token system, originally only on the Bitcoin blockchain, now is an Ethereum compatible, ERC20 token [24], and tied one-to-one with the US dollar. The primary functionality of Tether is "to facilitate the use of fiat currencies in a digital manner" [25].

- **BNB**
  Used as the primary driving force behind the Binance ecosystem, the BNB token fuels transactions on the BNB Chain similarly to the gas used on the Ethereum blockchain [26].

- **USD Coin**
  Similar to Tether, in that it is a stable coin, creating opportunities for using normal fiat currencies through blockchain networks [27]. USD Coin is backed by reserves based in the US Dollar [28]. It has a Euro-based counterpart, which is also created by Circle [29].

- **XRP**
  The token used in the XRP Ledger to settle transactions [30]. The XRP Ledger in turn enables developers to develop and deploy Web3 [31] applications ranging from crypto exchanges to XRP Ledger infrastructure and developer tooling [32].

- **Cardano**
  A self-proclaimed third generation of blockchain technology, Cardano provides the ability to develop and deploy decentralized applications, similar to XRP [33]. Cardano uses Ada as their currency of choice to fuel transactions on the blockchain [34].

- **Dogecoin**
  An "accidental crypto-movement that makes people smile" [35], Dogecoin is very close in functionality and operation to Bitcoin. The primary difference lies in the proof-of-work algorithm used, where Bitcoin relies on SHA-256, while Dogecoin uses the Scrypt algorithm [36].

- **Polygon**
  A Layer 2 scaling solution [37], Polygon operates with a number of sidechains to allow developers to deploy smart-contracts and distributed applications. Polygon attempts to "solve the scalability and usability issues [. . . of] public blockchains", like Ethereum [38].

- **Solana**
  Operating as a "single global state machine" [39], Solana provides developers with the ability to deploy smart contracts. This allows Solana to support any number of use cases, such as "finance, NFTs, payments, and gaming". [39]

Most of the respective blockchains for the cryptocurrencies mentioned above, are based on a PoS consensus mechanism. Moreover, excluding the stable coins, they have smart contract support and about half act as a layer 2 scaling solution on top of an existing blockchain. Only Bitcoin and Dogecoin's blockchains are based

on a PoW consensus mechanism, and neither support smart contracts. While a goal of our solution is to enable comparisons, we aim to target a variety of consensus mechanisms, before comparing multiple blockchains with the same consensus mechanisms. As such, we will investigate blockchains from alternative sources.

What is a Layer 2 Scaling Solution?

To understand what a layer 2 scaling solution is, we first consider where scaling is required. Blockchains, like Bitcoin and Ethereum, have a certain capacity limitation built into their consensus mechanisms [37]. For example, Ethereum has a time gap of 12 seconds between adding new blocks to their blockchain [40]. This notion of a *block time*, combined with an upper bound on the size of individual blocks, limits the maximum throughput of transactions into a blockchain.

As a result of this, a solution for scaling the block throughput above the limitations of the blockchain's consensus mechanism is required when blockchains reach a certain level of usage. One approach is to introduce changes to the fundamental protocol used in a blockchain. This is known as layer 1 scaling, where the layer described is the primary blockchain. Layer 1 scaling can introduce many challenges before the system is scaled up to the required capacity. Leveraging the strength of smart contracts, it is possible to handle some transactions off layer 1. Scaling the blockchain in this fashion is called layer 2 scaling. An alternate approach uses external systems to assist in scaling the blockchain [37]. An example would be sidechains. A sidechain is an independent blockchain, running in parallel to the main chain. The sidechain is connected to the main chain via a two-way bridge. This bridge allows for the movement of assets, e.g., funds, between the main chain and the sidechain [41, 42]. As such, transactions handled on the sidechain are equivalent to the transactions handled on the main chain.

4.1.2 Blockchains for Developing Distributed Applications

Besides blockchains with publicly traded cryptocurrencies, some blockchains exist which are primary used for developing distributed applications. Distributed applications (dApps) are applications deployed on a blockchain. They utilize smart contracts to act as a back-end, while a distributed file storage can be used to host the front-end [43]. This provides dApps with a functionality reminiscent of a classic client-server architecture.

One of the driving forces behind such solutions is the Hyperledger Foundation. They maintain and distribute a catalog of projects, ranging from enterprise-ready blockchain clients to developer tools for working with distributed applications. In Section 2.5, we discussed their project Hyperledger Caliper, which operates as a benchmarking tool for blockchains. Another of their projects, namely Hyperledger Sawtooth, allows end users to deploy a private blockchain network, using either

Practical Byzantine Fault Tolerance(PBFT) or Proof of Elapsed Time(PoET) as the consensus mechanism [15].

Another player in the market is the company R3, who owns and maintains the *Corda* project. Corda is a blockchain platform designed for digital finances [44]. It can be deployed in both a fully scalable enterprise deployment, for usage in production environments, and for development purposes on a single computer. The development setup allows for the configuration of a static network with predefined nodes. With the network running, it is possible to develop and test decentralized applications, without relying on external parties in a test blockchain network [45].

4.1.3 Summary

This section presented a set of blockchains from popular cryptocurrencies, while also highlighting a pair of blockchain solutions from the research world. In Section 4.1.1, we found that most production-ready blockchains used in the cryptocurrency market are based on a PoS solution, be it either directly as their own blockchain or as a scaling layer on top of a different blockchain. This conclusion led us to look into blockchains that are used for development and research purposes. Here we found a choice between a couple of projects. Our findings from this chapter are compiled into the following table: Table 4.1

Blockchain/cryptocurrency name	**Used in benchmarking**
Bitcoin	Yes
Ethereum	Yes
Tether	No
BNB	No
USD Coin	No
XRP	No
Dogecoin	No
Polygon	No
Cardano	No
Solana	No
Hyperledger Sawtooth	Yes
Corda	No

Table 4.1: Table presenting the blockchains slated for benchmarking in our solution.

As presented in Table 4.1, most of the blockchains from this section will not be benchmarked. The reason being that most of these blockchains have closed source software with limited configuration options. These limitations remove the possibility of setting up these blockchains in a private network, where we control

the nodes participating in the blockchain. Furthermore, in Section 4.1.2 we introduced Hyperledger Foundation's Sawtooth and R3's Corda. After an inspection of their respective documentation, we discovered that Hyperledger Sawtooth provides ready-to-go Docker files, whereas Corda would require more configuration to set up container orchestration. As such, we will be setting out to benchmark five blockchains in total: Bitcoin, Ethereum PoW, Ethereum PoS, Hyperledger Sawtooth PBFT, and Hyperledger Sawtooth PoET.

4.2 Orchestration

An important part of the problem statement from Chapter 3 is the notion of the blockchain network being benchmarked is a private network. To achieve this goal, an *orchestration* tool is required. Orchestration, being the way in which several computer systems, services, and applications are coordinated and managed to achieve some task [46]. This section will describe how we aim to orchestrate our system to produce a testing environment for our solution.

4.2.1 Containerization

Blockchain networks are built by a set of nodes, as described in Section 2.4. To build a network of nodes, using a minimal setup, we need a method for distributing software as a ready-to-go solution. One approach would be to compile a program as a portable or standalone piece of software. Such a program would be able to run without having to install it onto the system [47, 48]. Another approach, which we will be using in our solution, is known as containerization.

Containers function similarly to virtual machines. A virtual machine consists of a full operating system, a file system, and applications which can be run on the operating system. A host machine can assign a set of processors and an amount of memory to a virtual machine, limited only by what the host machine has available. Virtual machines on the host system then share access to processing time and memory with the host operating system and other virtual machines on the host system. Further, virtual machines can be bundled in packages which can be unpacked, setup, and, with a single command, run a network node. However, due to the virtual machines simulating an entire system, they create a large overhead for executing processing tasks for each virtual machine added to the host system [7].

Containers, on the other hand, allow the same flexibility of a virtual machine, but with a smaller resource overhead. They are packaged with a minimal operating system and only the applications required to execute a specific piece of software. This bundling ensures that any container can run on any host, regardless of operating system. As such, containers are more akin to portable software. For our solution, we will be using Docker to handle the orchestration of containers to build

our blockchain testing network. This is in part due to our existing knowledge of Docker, from our previous work [7], and also because alternative solutions, like Kubernetes, often depend on clustering multiple systems together [49], whereas we will be using a single system to test our solution.

Dockerfiles

The first step in creating containers for Docker to orchestrate, is to make a Dockerfile. A Dockerfile is a file containing a set of instructions Docker uses to build an image of the new container [50]. These instructions can complete numerous tasks, like copying files and running initialization commands [51]. At the point of writing a Dockerfile, it is important to distinguish between the different stages in the life of a container. Some instructions used in Dockerfiles only affect the container's file system before the image is built. As such, these changes will not be available in the final container.

Docker Compose

Docker supports Docker Compose files as well, which is another possibility for configuring, creating, and starting containers. Furthermore, Docker Compose files were specifically made for simplifying the act of defining and starting up multi container applications, and does so by the help of YAML files where services, volumes, and images can be specified. The YAML files collect all the configuration needed in one place and enables users to spin everything up, or tear it all down, with a single command [52].

4.3 Resource Monitoring

The problem statement in Section 3.1 states that blockchain clients resource usage is to be benchmarked inside a private network. And as mentioned previously in Section 4.2, the blockchain clients that are to be benchmarked are containerized and deployed by Docker. This means that Docker containers must be taken into consideration when the blockchain client's resource consumption is monitored. This section will therefore explore possible existing resource monitoring solutions that can access Docker containers, instead of developing our own custom system as we did in the previous paper [7].

What is Resource Monitoring

Before continuing with the exploration of existing resource monitoring solutions, it is important to fully specify what benchmarking means and how it is carried out in computer science. Benchmarking is the act of comparing performance metrics

against an accepted standard or one, or more, other benchmarks. Benchmarking typically outputs a score for the test where the higher the score the better, or, it simply outputs raw performance metrics, such as CPU, memory, or network usages. An example of a home brew benchmark test for a piece of software, could simply be running said software locally and looking at the task manager to gain an insight into the computer's performance [53].

4.3.1 Existing Resource Monitoring Solutions

This project will aim to benchmark several docker containers performance metrics, which isn't that different from the home brew benchmark example in practice. Here, the piece of software will be a blockchain network consisting of several nodes, and the task manager will be a resource monitoring software that tracks and stores the performance metrics of those blockchain nodes. Several existing downloadable solutions exist, and the rest of this section will first explore existing resource monitoring solutions that are compatible with Docker, and then choose what solution this project will utilize.

Docker Stats

We don't have to look far for an existing solution when it comes to resource monitoring for Docker containers because docker has a command named *docker stats* which displays a live stream of container(s) resource usage statistics. The stream outputs the raw usage data to STDOUT and allows the user to format the output data by using a custom template. Docker stats is, however, not able to log or store the resource usage data on its own, but would need an extension or another program to log the data [54].

Glances: An Eye on Your System

Glances is an open source, cross-platform system monitoring tool and allows for real-time performance monitoring of your system and is available to download via GitHub [55]. It supports the monitoring of various aspects, such as CPU, memory, disk, and network usage, while also covering fan speeds, voltages, temperatures, logged-in users, and Docker container monitoring. The framework presents the data in an easy to overview dashboard in STDOUT and can also output its performance data in Comma-Separated Values (CSV) or JavaScript Object Notation (JSON) file formats. Glances is written in Python and allows its users to customize what specific performance data it should monitor [55, 56].

Prometheus

Prometheus is an open-source system alerting and monitoring solution which stores its data in a time-series database and can pass the data to its data visualizer, Grafana. Prometheus can collect performance metrics from user configured targets, such as docker containers, at given intervals. Prometheus is written in the Go programming language and has its own functional query language, PromQL, that users can utilize to either show data as a graph, table, or use it with other services [57].

4.3.2 Summary

There are of course more solutions available than the three mentioned above, but a choice can already be made. Docker stats is only lacking the ability to log and store the benchmarking data on its own, and if it was able to do that, we would use the Docker stats command to monitor the Docker containers. Prometheus, on the other hand, is a big and complex monitoring and alerting system, which would be able to satisfy all of our Docker container monitoring needs, but extracting the data is not transparent. Glances provide the same capabilities as the Docker stats command with the option to log and store the data inside a CSV file, which is practical if the data is to be processed in Excel manually. Glances would not require an extension either to store and monitoring data as the docker stats solution would. As Glances provides the exact capabilities that we need without it being overly complex for this project's scope, Glances will be used to monitor and extract the relevant resource data from the blockchain node's Docker containers.

4.4 Architecture

This section will discuss the overall architecture of how blockchain clients will be benchmarked as stated in the problem statement in Section 3.1. The design chapter has covered which blockchains are to be benchmarked in Section 4.1, how these blockchain nodes are to be orchestrated with Docker in Section 4.2, and finally, how the resource consumption data is to be monitored and logged with the Glances framework in Section 4.3. Combining the knowledge gained from these sections, a simplified architecture overview can be created, which can be seen in Figure 4.1.

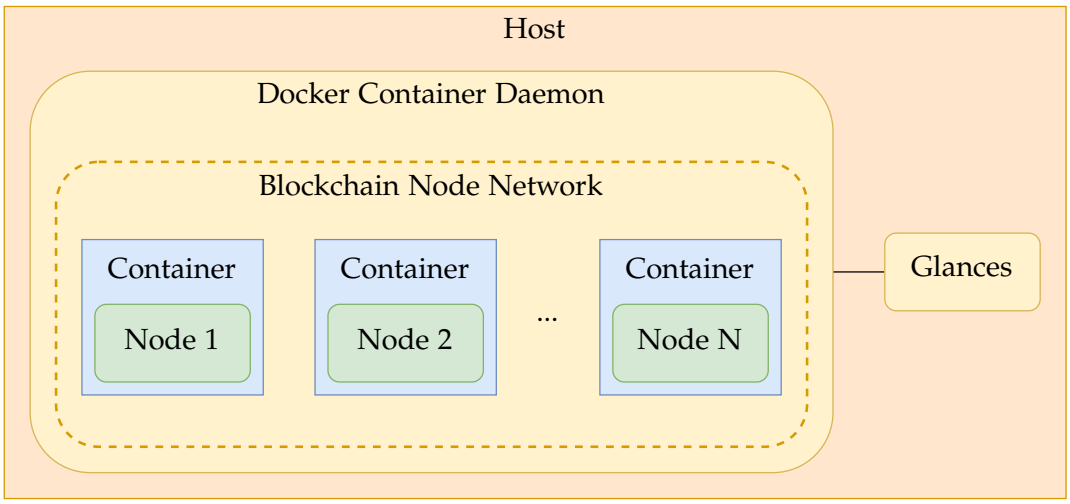


Figure 4.1: Architecture overview of our benchmarking solution.

Host

The outermost layer in Figure 4.1 is the Host, which refers to the underlying system that the benchmarking tests will be conducted on. The system will run the Docker Daemon software where the blockchain nodes are to be containerized. The host system should aspire to have as many as possible hardware resources available, such that several benchmarking tests can be conducted for each blockchain client type with growing numbers of nodes, to get the best possible results.

Docker Container Daemon

The Docker Container Daemon will be running on the host system as depicted in Figure 4.1. Its main responsibilities are handling container lifecycle management, networking, storage, and other essential tasks related to containerization.

Glances

Glances will operate on the same level as the Docker Container Daemon, as depicted in Figure 4.1. Glances will start when the Blockchain Network has been established and all Docker containers have been created and are running. Glances is going to be monitoring and collecting resource consumption data on each of the containers by contacting the Docker Daemon via the Docker Engine API, as depicted in the figure. During active tests, Glances will be specified to log all the monitoring data to a CSV file on the host, which can be easily extracted and viewed later in Excel.

Blockchain Node Network

The Blockchain Node Network layer in Figure 4.1 is a logical grouping which represents the blockchain node network and is shown in the figure by a dashed line surrounding the Docker containers. The Docker containers, regardless of the quantity, will each encompass a blockchain node, collectively forming a blockchain node network and utilizing Docker's internal network for communication.

Chapter 5

Implementation

This chapter will follow the architecture design in Section 4.4 and describe how each of the blockchain systems is to be benchmarked, both successful and unsuccessful implementations. In addition, this chapter will describe how our test campaigns for each blockchain node network were carried out with Glances. Chapter 6 will explore the results of the blockchain node network's resource monitoring.

5.1 Resource Monitoring

In this section, we will delve into the setup and monitoring process of each blockchain node network test using Glances. Additionally, we will address and explain some obstacles encountered along the way.

5.1.1 Docker Container Monitoring with Glances

Glances has, as mentioned in Section 4.3.2, been chosen to monitor the blockchain node networks. Following Glances' documentation [56], it is possible to run Glances in container mode, where it only monitors the Docker containers on the host system. During the monitoring of a blockchain node network, Glances is configured to export its container resource data into a CSV file, continuously appending data every 2 seconds. This approach facilitates convenient viewing and analysis at a later stage. Figure 5.1 displays an example of which Docker container resources Glances can monitor and extract.

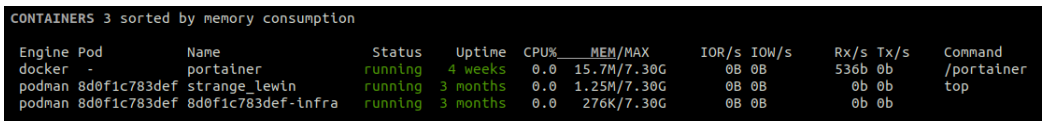


Figure 5.1: Possible Docker container resources that Glances can monitor [56].

Each resource metric that can be monitored from Docker containers using Glances will be further explained here:

- **CPU** - is shown as a percentage of a whole CPU core. Meaning that utilization of a whole CPU core is displayed as 100%, or if two cores are utilized fully, 200%.
- **Memory** - is shown as the amount of used memory, specifically calculated by subtracting the available memory from the total memory.
- **IO** - is the disk input and output (IO) throughput in Bytes per second, split into read and write throughput respectively.
- **Network** - is the network interface bit rate, displayed in bits per second (bps). It is split into two, as the IO above, which are the received and sent rate respectively.

The IO Metric

When tests with Glances are conducted, it is unfortunately clear that the IO metric isn't a useful metric for us, as it is constantly outputting zero in both read and write

columns. Our best guess is that the high amount of available RAM on the server, as mentioned in Figure 5.1 above, neglects the need for the blockchain clients to store any of its data on disks.

Reordering Glances Data

The data columns that we are interested in, namely the CPU, memory, and network read and write, are included once per blockchain node in each test run inside the output file that Glances writes to. This happens as each node's resource data is outputted by Glances every two seconds, which adds another row that must encompass every node's resource data. The problem, however, is that Glances doesn't order the new row's columns according to the previous row's columns. A bit clearer example of this phenomenon can be seen in Figure 5.2, where a test run with three nodes is conducted, where the columns have been simplified to CPU, memory, and network for the readability of the example. Notice here that the *Node 1* data is changing position in each row, making the data unintelligible without sorting it first.

Timestamp	CPU	Memory	Network	CPU	Memory	Network	CPU	Memory	Network
00:00:00	Node 1			Node 3			Node 2		
00:00:02	Node 2			Node 1			Node 3		
00:00:04	Node 1			Node 2			Node 3		
00:00:06	Node 3			Node 2			Node 1		
00:00:08	Node 2			Node 3			Node 1		

Figure 5.2: Raw unordered Glances output data with 3 nodes participating.

To circumvent this problem, and sort the output data, making it intelligible and enable processing of the resource metrics, a C# application named *ThesisDataManager* was created. The *ThesisDataManager*, TDM for short, is needed to process the files that Glances outputs, such that the data can be sorted. An example of the expected output data, after the TDM has processed the data, can be seen in Figure 5.3. The TDM adds a new column to the data, which is the combination of the network read and write throughput in bps, such that a combined network metric in bps is available.

Timestamp	CPU	Memory	Network	CPU	Memory	Network	CPU	Memory	Network
00:00:00	Node 1			Node 2			Node 3		
00:00:02	Node 1			Node 2			Node 3		
00:00:04	Node 1			Node 2			Node 3		
00:00:06	Node 1			Node 2			Node 3		
00:00:08	Node 1			Node 2			Node 3		

Figure 5.3: Ordered Glances output data with 3 nodes participating.

5.2 Bitcoin

When searching for Bitcoin clients, we investigated Bitcoin's official sources. During this, we discovered the Bitcoin Core client [58], which is developed and maintained by the developers of Bitcoin. We then set out to create a Dockerfile, to build container images for running a Bitcoin network. We started with a basic setup with two nodes. To this end, we created two individual Dockerfiles, as presented in Listing B.1 and Listing B.2. Bitcoin allows a minimal setup, with no initialization and minor changes to the configuration file, that enables the Bitcoin clients to operate in a private network. However, in the process of connecting the clients, we found that the current version of the Bitcoin Core client no longer supports actively mining Bitcoin. This change was brought about as a result of CPU mining being "useless" [59]. As such, we worked with an old release of the Bitcoin Core client, namely version 0.12 of the client. This is the last version of the client to still have an integrated CPU miner. However, when setting the application flags for automatic mining, we noticed the clients did not mine on their own. It was possible to manually request for new blocks to be mined, and after mining 100 blocks, we saw the genesis block had matured and its block reward was paid out. Following this setback, we attempted a similar setup with some previous releases of the client, but these versions provided the same result. As such, given the time limitations of this project and the possibility that the Bitcoin Core client had never been functional with our system, we decided to abandon using the Bitcoin Core client. In parallel with the setbacks of the Bitcoin Core client, we investigated alternative clients to use for a Bitcoin network, however the clients we uncovered were all implemented with GPU or ASIC mining in mind. Requiring the usage of multiple GPUs or ASICs would impact the core of our system design, and as such, we arrived at the conclusion that we were unable to implement a Bitcoin blockchain network.

5.3 Ethereum

Following our attempt at setting up the Bitcoin Core client, we transitioned to setting up an Ethereum network. Again we started off with investigating an official client. Ethereum's documentation presents several clients, with seemingly no difference between the clients, apart from the programming language used to implement the client. As such, we decided to go with the Geth client. Geth provides a Docker image, which allows a quick deployment of the client. However, this image does not allow the configuration which is needed to create a private network, and as such, we created a custom Dockerfile for our image.

5.3.1 Image creation script

Using Geth, requires a particular order of operations to initialize a network node. Geth supports both the Clique and Ethash consensus algorithms, where Clique uses Proof of Authority (PoA) and Ethash uses PoW, with minor changes between the steps for their initialization. PoA being an alternative approach to PoS, in which the participant's reputation is staked rather than staking a currency [60]. First, a genesis block must be defined. This is done using a JSON file, describing a set of hard forks, a starting difficulty in hexadecimal, a gas limit, and an initial allocation of ether. For Clique, it should also define the target block time. Following the definition of the genesis block, the client's database is initialized with the genesis block. This initialization must be done for each client, and cannot be achieved by synchronizing with other nodes. Moreover, the initialization is also used for adding future changes to hard forks. The Geth network consists of a bootstrap node and a set of participating nodes. When connecting the nodes to create the blockchain network, the bootstrap node maintains a list of the nodes in the network. New nodes are required to know the address of the bootstrap node and will receive a copy of the list of nodes. The bootstrap node is initiated by providing it with its address, and it generates a node record based on the provided address. This node record is then in turn provided to participating nodes for them to connect to the bootstrap node. To define a blockchain participant as either a miner or a signer, depending on the network, different flags are passed to the application. These flags can be seen in Listing 5.1 and Listing 5.2. An important flag for the miners is `--miner.threads=1`, as this flag limits our miners to using a single processor for mining and allows for more miners on the same host system. After this setup, the network is ready to create and validate blocks. At this point, it is possible to add transactions, however transactions are not required for the clients to create blocks [61].

Listing 5.1: The command executed to start a signer

```
geth --password password --networkid ${NETWORK_ID} --syncmode 'full' --http
    --http.api eth,net,web3,admin,debug,clique,les,txpool --bootnodes $(cat
    /eth-data/bootstrap-enode) --unlock $(cat ethereum-address) --mine
    --miner.etherbase $(cat ethereum-address) --allow-insecure-unlock
```

Listing 5.2: The command executed to start a miner

```
geth --networkid ${NETWORK_ID} --syncmode 'full' --http --http.api
    eth,net,web3,admin,debug,clique,les,txpool --bootnodes $(cat
    /eth-data/bootstrap-enode) --miner.threads=1 --mine --miner.etherbase
    $(cat ethereum-address)
```

To facilitate the node initialization and subsequent creation of images, we created a bash script. This bash script, available in full in Listing C.4, also handles the building of images, creation of containers, and starting of nodes. The Dockerfiles, as presented in Listing C.1, Listing C.2, and Listing C.3, copy the required files into the container, install the Geth client, and start the client.

5.4 Hyperledger Sawtooth

As previously mentioned in Section 4.1.3, Hyperledger Sawtooth was chosen as one of the five blockchain networks that we set out to benchmark. Furthermore, the Hyperledger Sawtooth official documentation [62] provides ready-to-go Docker Compose files for each of the two available network configurations, which are the PoET and PBFT configuration. The rest of this section provides a description of how the two Hyperledger Sawtooth network configurations were set up and initialized.

5.4.1 Configuring Hyperledger Sawtooth

In a Sawtooth blockchain network, regardless of its consensus mechanism, each Sawtooth node encompasses a set of transaction processors, a validator, an optional REST API, and a consensus engine. The first node that is initialized in a Sawtooth network must specify the genesis block, which holds configurations for the network's initial on-chain settings. When other nodes join the network, they must first access the on-chain settings inside the genesis block. The consensus engine can be custom implemented, but Hyperledger Sawtooth does provide their implementation of PoET and PBFT. All nodes on the network must run the same type of consensus engine, as well as the same types of transaction processors. It should be stated that Sawtooth doesn't have a *master node* or anything like that, the first node in a network must simply have a specified genesis block, which is then shared with other nodes as they join the network [62].

5.4.2 Hyperledger Sawtooth PBFT

The Sawtooth PBFT consensus provides a voting-based consensus algorithm with Byzantine fault tolerance (BFT) that has finality, meaning it does not fork. Following the official documentation for setting up a PBFT network, a Docker Compose YAML file can be downloaded, which handles all the required configuration to start the network on Docker for us [62].

Docker Compose files have been explained previously, in Section 4.2.1, and a shortened example of our Sawtooth PBFT YAML file can be seen in Appendix D. It has been shortened to a network configuration containing two nodes for the purposes of showing it in this report, whereas our non-shortened YAML file specifies a 5 node network. When spinning up the network with the Docker Compose file, each node consists of several Docker containers, as each node's validator, REST API, consensus engine, and transaction processors, are run in each of their own Docker containers. This does make it harder to process the monitoring data, as each node's resource consumption is spread out over multiple containers. But it is not a problem, however, as monitoring data can be combined for the respective node's containers. When starting our network, it does seem like the nodes in the network are working as intended. The nodes are communicating, and when querying each node about its known peers through the shell client, which is specified at the bottom of Listing D.1, it outputs the expected data. However, no blocks are created on the network, and even though we create transactions and send them to a node, they are not propagated through the network, regardless of what transaction batch size is configured.

After consulting online forums and some of its users, such as Sawtooth's official Discord server [63], we can conclude that we aren't the only users of Hyperledger Sawtooth PBFT that can't make it work. After countless hours of reading and problem-solving the PBFT network, we gave up on getting the network to run properly. We could have monitored the nodes, as they were technically running, but it would have been data monitoring of idle nodes, which wouldn't yield any useful data.

5.4.3 Hyperledger Sawtooth PoET

The Sawtooth PoET consensus provides a leader-election lottery system which is crash fault-tolerant [62]. Setting up a Sawtooth PoET network is similar to setting up a PBFT network, and Listing D.1 is so similar that we won't include an example of a PoET Docker Compose YAML file. The only changes in the PoET YAML file is the addition of a PoET specific transaction processor. Additionally, there are changes in the configuration of the genesis block, which is done inside the *validator-0* setup in the PBFT listing.

When spinning up the Sawtooth PoET network with our version of the Docker

Compose file, it did again seem like nodes in the network were working as intended. However, further inspection of the running nodes revealed that the genesis block was not even validated, which of course means that the network could not continue its operation. Consulting the same forums as previously mentioned, especially the official Discord channel for Hyperledger Sawtooth [63], we found that nobody could help us, and some even declared the Sawtooth PoET network as dead. We were unable to solve any of the errors of the PoET network, and, as such, unable to gather any monitoring data.

Chapter 6

Results

This chapter will present and clarify the findings derived from the benchmarks conducted on the successfully implemented blockchain node networks. A detailed explanation of the methodology employed for the resource monitoring of the networks on the server was described in Section 5.1. In Chapter 5, we elaborated on our successful implementation of the Ethereum blockchain network in two distinct modes: Clique, utilizing Proof of Authority (PoA), and Ethash, employing Proof of Work (PoW). The chapter will feature dedicated sections for each functional blockchain node network, culminating in a comparison section that evaluates and contrasts the performance of these blockchain node networks.

6.1 Testing Method

To gain insight into the potentially increasing resource consumption over time as the blockchain networks automatically generate blocks, we decided to monitor the blockchain node networks for 60 minutes each. Furthermore, we determined that each network would be characterized by a variety of network configurations. As such, each blockchain node network was run with 5, 10, 15, 20, and 25 nodes, which enables us to follow the increased resource consumption as each network's size grows.

As mentioned in Section 4.4, we planned to contact Aalborg University and acquire a cloud server instance where we could benchmark the blockchain node networks. We successfully acquired a powerful cloud Ubuntu server instance, which hardware specifications can be seen in Table 6.1. We could unfortunately not obtain any other hardware specifications of the server or our specific server instance. But as we also mentioned in our previous paper [7], we are only interested in comparing the benchmarking data with each other, and how they perform against each other. This means that it does not matter what the particular hardware configuration is, as long as we conduct all of our tests on the same server instance.

CPU Cores	Memory in GB	Disk Space in GB
32	64	100

Table 6.1: Cloud server specifications.

6.2 Clique Results

This section will showcase and provide an explanation of the monitored results that our Ethereum Clique PoA network yielded. The Clique network was successfully run in five different configurations, where the number of participating nodes in each test was incremented by five. As mentioned in Section 5.3.1, it is possible to control the target block creation time for a Clique network in seconds. We therefore decided to set the target block creation time for our Clique networks to 12 seconds, which is the same block creation time that the Ethereum main network is operating with as well [64].

The monitored resource metrics for one node in all the Clique network configurations are displayed in Table 6.2. As mentioned in Section 6.1, all network configurations were executed for a duration of 60 minutes. Therefore, the resource metrics presented in Table 6.2 represent an average over the 60-minute period. Later, a single benchmark run is examined in more detail.

Nodes	Runtime in minutes	Blocks created	Blocks/min	Average CPU usage in percent	Average memory usage in GB	Average combined received and send rate in bps
5	60	300	5	1.417	0.063	13054.4
10	60	300	5	1.581	0.078	17225.1
15	60	300	5	2.026	0.097	28348.6
20	60	300	5	2.240	0.112	60494.1
25	60	300	5	2.697	0.148	75164.2

Table 6.2: Results of the monitored resource consumption of the Clique network.

Table 6.2 presents the data gathered from our Clique benchmarks, where each row is a different test case. The data in the three columns: *Runtime in minutes*, *Blocks created*, and *Blocks/min*, is constant. This is expected as we have set the target block creation time to 12 seconds, which corresponds with five blocks per minute. The data in the three columns: *Average CPU usage*, *Average memory usage*, and *Average combined received and send rate in bps*, grows with the number of participating nodes. This trend is also presented in Figure 6.1, which plots the data from the last three columns. The participation of additional nodes in Clique networks leads to an increase in the network's receive and send rate. This can be attributed to the PoA consensus mechanism, which incentivizes nodes to become the next validator. The CPU and memory usage increases because each node needs to process more network communication as more nodes participate in the network.

The graphs in Figure 6.1 provide a trend line over the growing resource usage data, where each resource metric's graph shares the same x-axis, which is the number of participating nodes in the corresponding test run of a Clique network. In Figure 6.3, the top graph showing the average CPU usage has been fitted with a linear tendency function, while the other two have been fitted with an exponential tendency function. Furthermore, the R^2 values exhibit a range across the metrics measured. The combined network rate demonstrates the lowest R^2 value, standing at 95.08%. In comparison, the CPU usage shows a higher R^2 value of 97.81%, while the memory usage exhibits the highest R^2 value of 99.02%. The R^2 values are a measure that tells you how well the data points fit the trend line or regression line in the graph, with a higher value indicating a stronger fit [65].

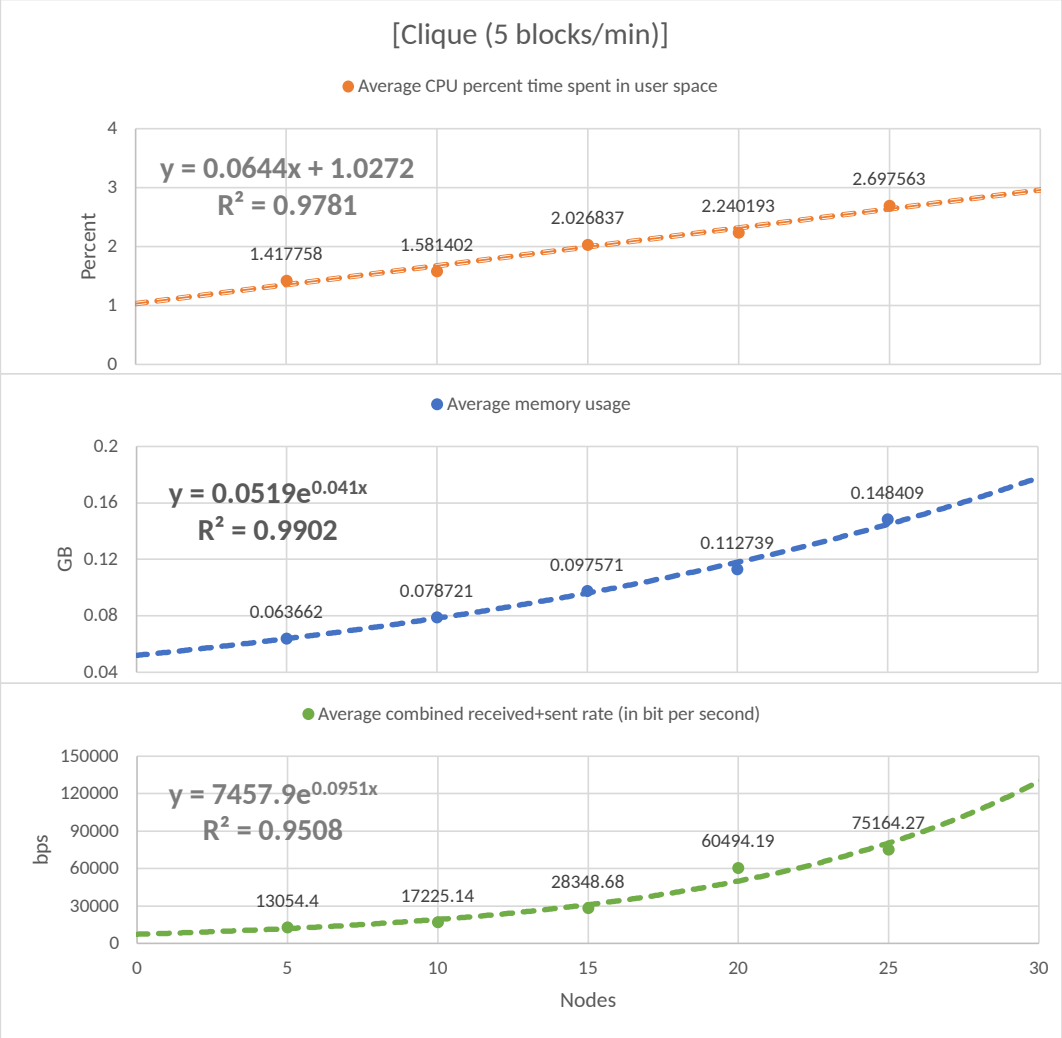


Figure 6.1: The resource consumption data from Table 6.2 showed in a three in one graph.

6.2.1 Visualizing the Growth of Resource Consumption Data

Altering our scope to investigate a single benchmarking run, so we can observe the monitoring data of a node from the Clique benchmark with a 5 node network as time progresses, is presented in Figure 6.2. Similar graphs have been made from the remaining four benchmark runs, which are available in Section A.1.

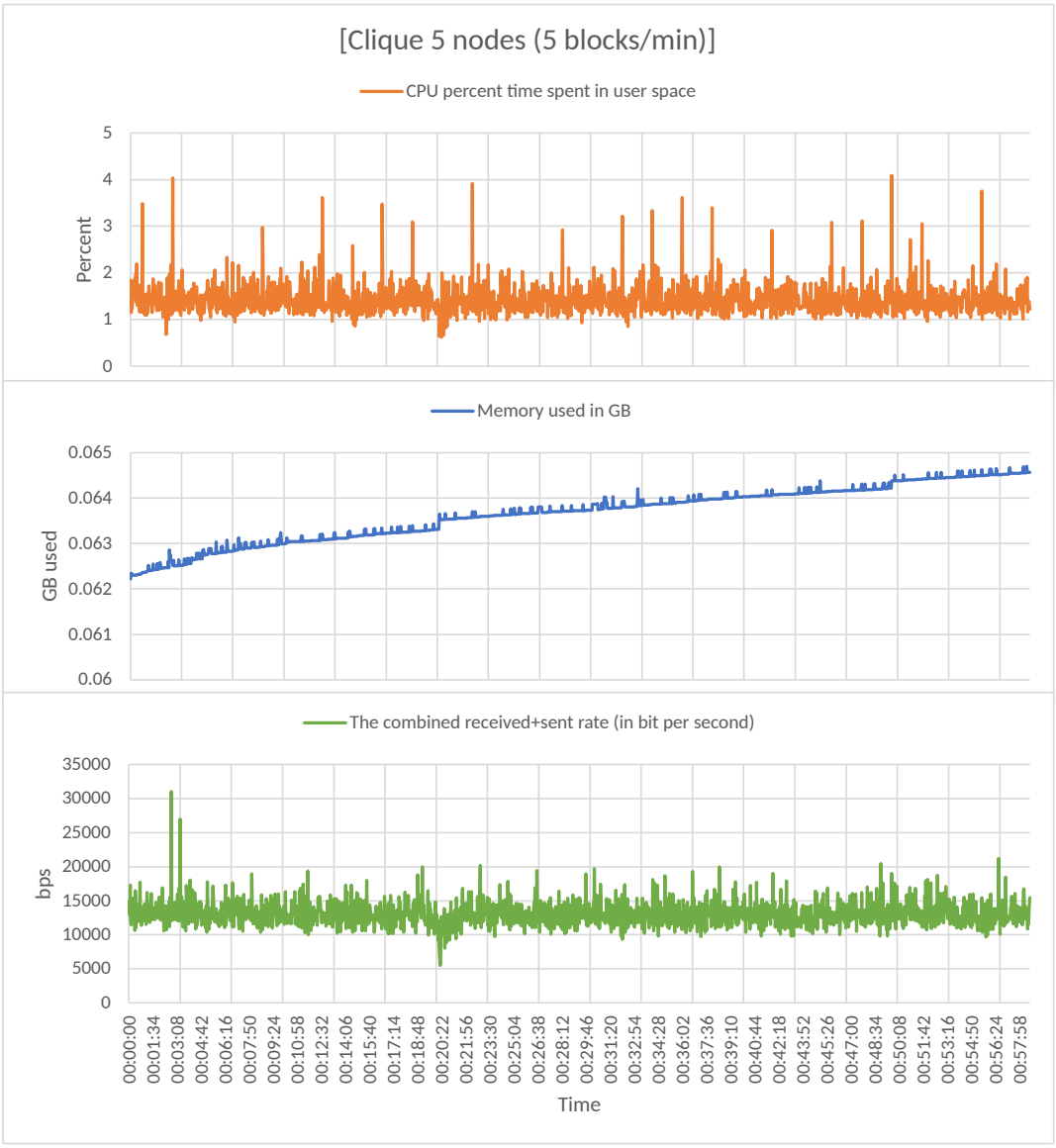


Figure 6.2: One node's resource consumption when running the Clique network with 5 nodes for 60 minutes.

The graphs' x-axis represent the elapsed time, starting from zero and up to the benchmarks ending after 60 minutes has progressed. When looking at the CPU percent time spent in user space, the top graph, the CPU usage is quite steady as time progresses, albeit having some spikes. The CPU usage is as we expected because the number of participating nodes is constant in each test run. This also means that the amount of network communication, the bottom graph, is more or less constant as well, with a few spikes. The memory usage, the middle graph, is, however, growing as time progresses. It starts at a memory usage of $\tilde{0}.062$ *GB*

before it grows to *õ.0645 GB* in the 60-minute benchmark duration. We do not know what data each node specifically stores, which could be a range of items such as blocks, transactions, and network states. However, we can conclude that each node's memory must increase in size as new blocks are created and added to the blockchain every 12 seconds.

6.3 Ethash Results

This section will showcase and provide an explanation of the monitored results that our Ethereum Ethash PoW network yielded. The Ethash network was successfully run in five different configurations, where the number of participating nodes in each test was incremented by five. It is not possible to set a block creation time as in the Clique network; however, it is possible to set a *starting* difficulty in hexadecimal for an Ethash network, as mentioned in Section 5.3.1. But regardless of what the starting difficulty is, all Ethereum PoW clients adjust their mining difficulty each block, aiming for a block creation time of 12 seconds [66]. To account for the dynamic difficulty adjustment in Ethash clients, our goal was to set the initial difficulty at a level that closely aligns with a target block creation time of 12 seconds, similar to the approach used in the Clique network. The specific starting difficulty would depend on the number of nodes participating in the Ethash network configuration. The chosen mining difficulty for our Ethash network configurations can be seen in Table 6.3. Having a block creation time of 12 seconds on the Ethash networks also meant that the results of the Clique and Ethash networks were as comparable as possible.

Nodes	Difficulty	
	Hexadecimal	Decimal
5	0x3567E0	3,500,000
10	0x6ACFC0	7,000,000
15	0x90F560	9,500,000
20	0xB71B00	12,000,000
25	0xE4E1C0	15,000,000

Table 6.3: The mining difficulties for the Ethash network configurations.

The monitored resource metrics for one node in all the Ethash network configurations are displayed in Table 6.4. As mentioned in Section 6.1, all network configurations were executed for a duration of 60 minutes. Therefore, the resource metrics presented in Table 6.4 represent an average over the 60-minute period. We will be going through a single benchmark run later for the Ethash resource consumption.

Nodes	Runtime in minutes	Blocks created	Blocks/min	Average CPU usage in percent	Average memory usage in GB	Average combined receive and send rate in bps
5	60	293	4.8	102.8	0.284	12966.5
10	60	318	5.3	102.4	0.285	17972.9
15	60	307	5.1	102.1	0.286	28707.2
20	60	308	5.1	101.6	0.287	44814.6
25	60	307	5.1	101.2	0.290	71332.48

Table 6.4: Results of the monitored resource consumption of the Ethash network.

Looking at Table 6.4 the data in the 3 columns: *Runtime in minutes*, *Blocks created*, and *Blocks/min*, are not as consistent as the Clique results. The metrics in these columns fluctuate a bit because of the randomness factor when using PoW and our starting difficulty not being 100% perfect in regard to reaching a block creation time of 12 seconds at the start of when the Ethash benchmark runs.

Examining the *Average CPU usage in percent* column and its data in Table 6.4, the CPU usage in percent is above 100% and decreasing as more nodes participate in the network. As mentioned in Section 5.1, Glances measures the Docker container CPU usage as a percentage of a whole CPU core. Meaning that a CPU usage of 102.8%, from the 5 nodes configuration, is above 100% because more than a whole CPU core is used by this node. One CPU core is used completely because of our choice in the configuration settings of the Ethash networks to limit each node's mining efforts to one CPU core, as mentioned in Section 5.3.1. Despite being primarily engaged in its mining operation, each node still needs to process additional data unrelated to mining. This additional processing requirement can account for why the value may exceed 100%. As the network expands with more participating nodes, there is a tendency for the CPU usage percentage to decrease. One possible explanation for this behavior is that the CPU cores, as mentioned in Section 6.1, are limited to 32. As the number of network nodes increases, the number of available CPU cores decreases, resulting in more nodes having to share a decreasing number of free CPU cores.

The last two columns: the *Average memory usage in GB*, and the *Average combined receive and send rate in bps* increases quite evenly as the number of participating nodes increases.

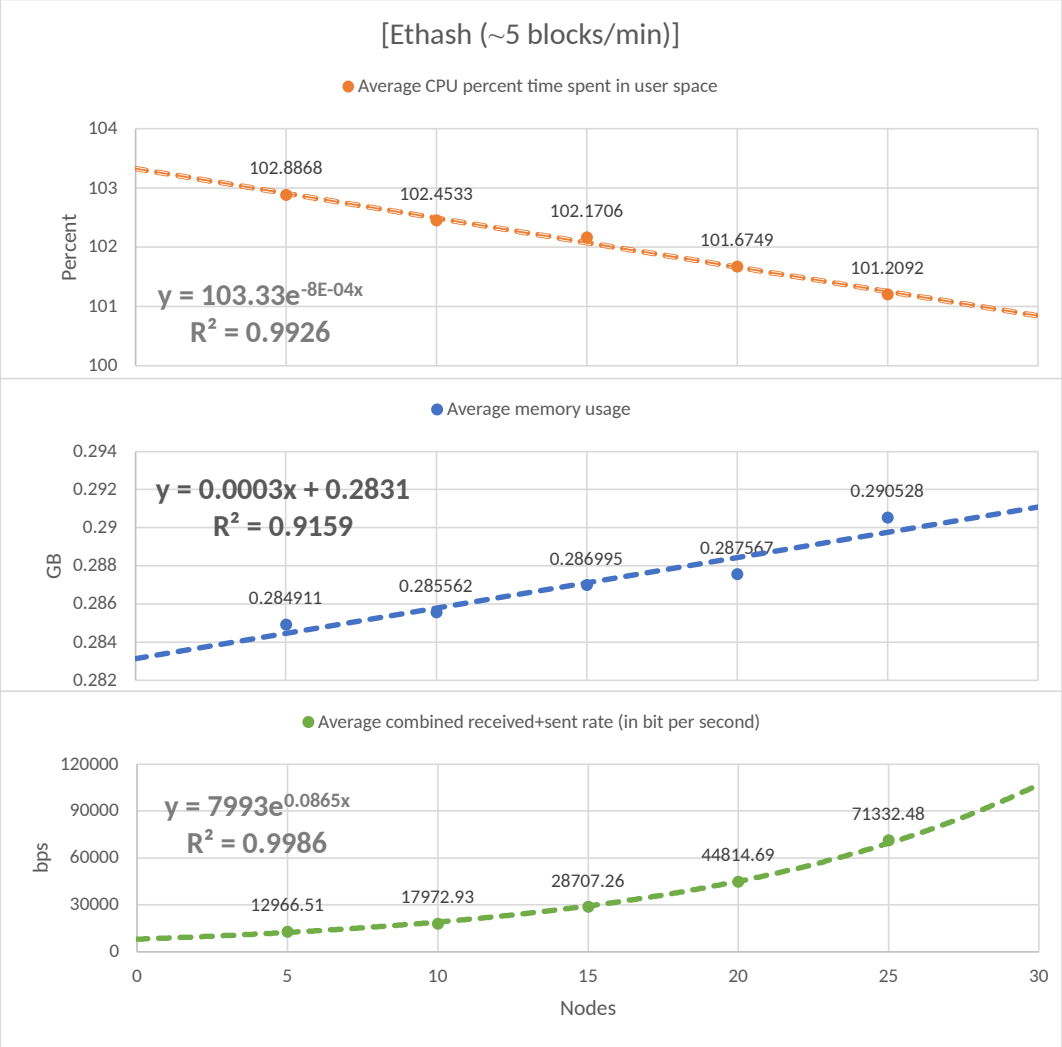


Figure 6.3: The resource consumption data from Table 6.4 showed in a three in one graph.

The data tendencies from Table 6.4 can also be seen in Figure 6.3, where each resource metric's graph shares the same x-axis, which is the number of participating nodes in the corresponding test run of an Ethash network. In Figure 6.3, each graph has been fitted with an exponential tendency function except for the memory's graph, which has been fitted with a linear tendency function. Furthermore, the R^2 values exhibit a range across the metrics measured. The memory usage demonstrates the lowest R^2 value, standing at 91.59%. In comparison, the CPU usage shows a higher R^2 value of 99.26%, while the combined network rate exhibits the highest R^2 value of 99.86%.

6.3.1 Visualizing the Growth of Resource Consumption Data

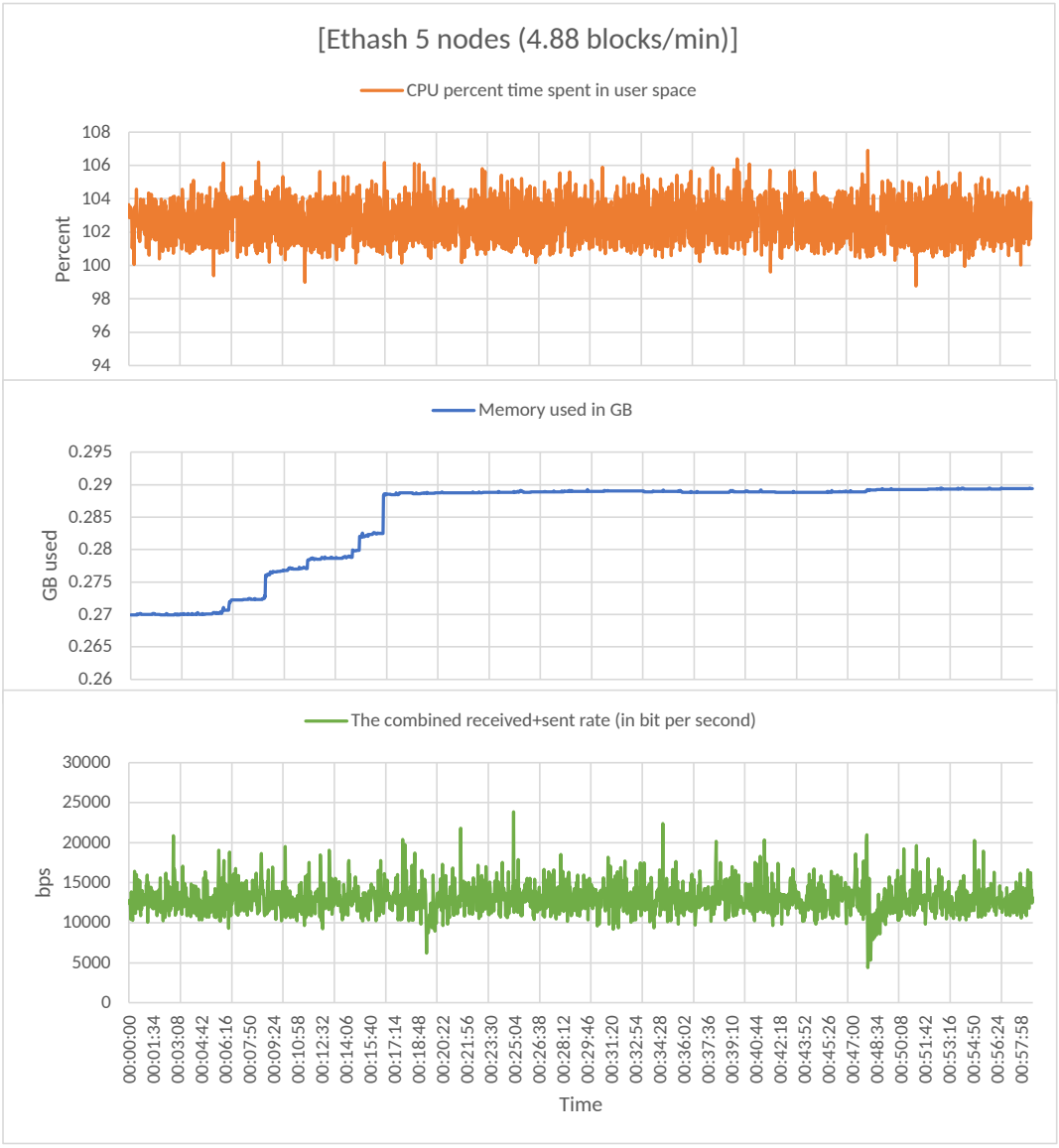


Figure 6.4: One node's resource consumption when running the Ethash network with 5 nodes for 60 minutes.

Altering our scope to investigate a single benchmarking run, so we can observe the monitoring data of a single node from the Ethash benchmark with a 5 node network as time progresses, as presented in Figure 6.4. Similar graphs have been made from the remaining four benchmark runs, which are available in Section A.2.

The graphs' x-axis represent the elapsed time, starting from zero and up to the

benchmarks ending after 60 minutes has progressed. When looking at the CPU percent time spent in user space, the top graph, the CPU usage is in the range from 100 percent to 106 percent as time progresses. The CPU usage is as we expected because the number of available CPU cores were set before the network nodes were started, as mentioned in Section 5.3.1. The amount of network communication, which can be seen on the bottom graph, is in the range of 10,000 bps to 20,000 with few spikes because no changes are occurring to the network configuration while it is running. The node's memory usage starts at *Õ.27 GB* before it grows to *Õ.29 GB* in the 60-minute benchmark duration. We do not know what data each node specifically stores, which could be a range of items such as blocks, transactions, and network states. However, we can conclude that each node's memory usage grows as new blocks are created and added to the blockchain.

6.4 Comparing Results

In this section, we will analyze the resource consumption of the monitored Clique and Ethash networks. While no new metrics will be introduced, we will compare the results from Clique's Table 6.2 and Ethash's Table 6.4 together in Figure 6.5. By comparing these findings, we can gain insights into the resource utilization differences between the two networks. The top graph in Figure 6.5, *Average CPU percent time spent in user space*, has its y-axis scaled logarithmically with a base of four, while the other graphs are scaled normally. Each of the three graphs in the figure are sharing the same x-axis as well, which represents the number of participating nodes in the respective networks. The rest of the section will go through each graph in Figure 6.5, delving into the differences and comparisons between the two networks.

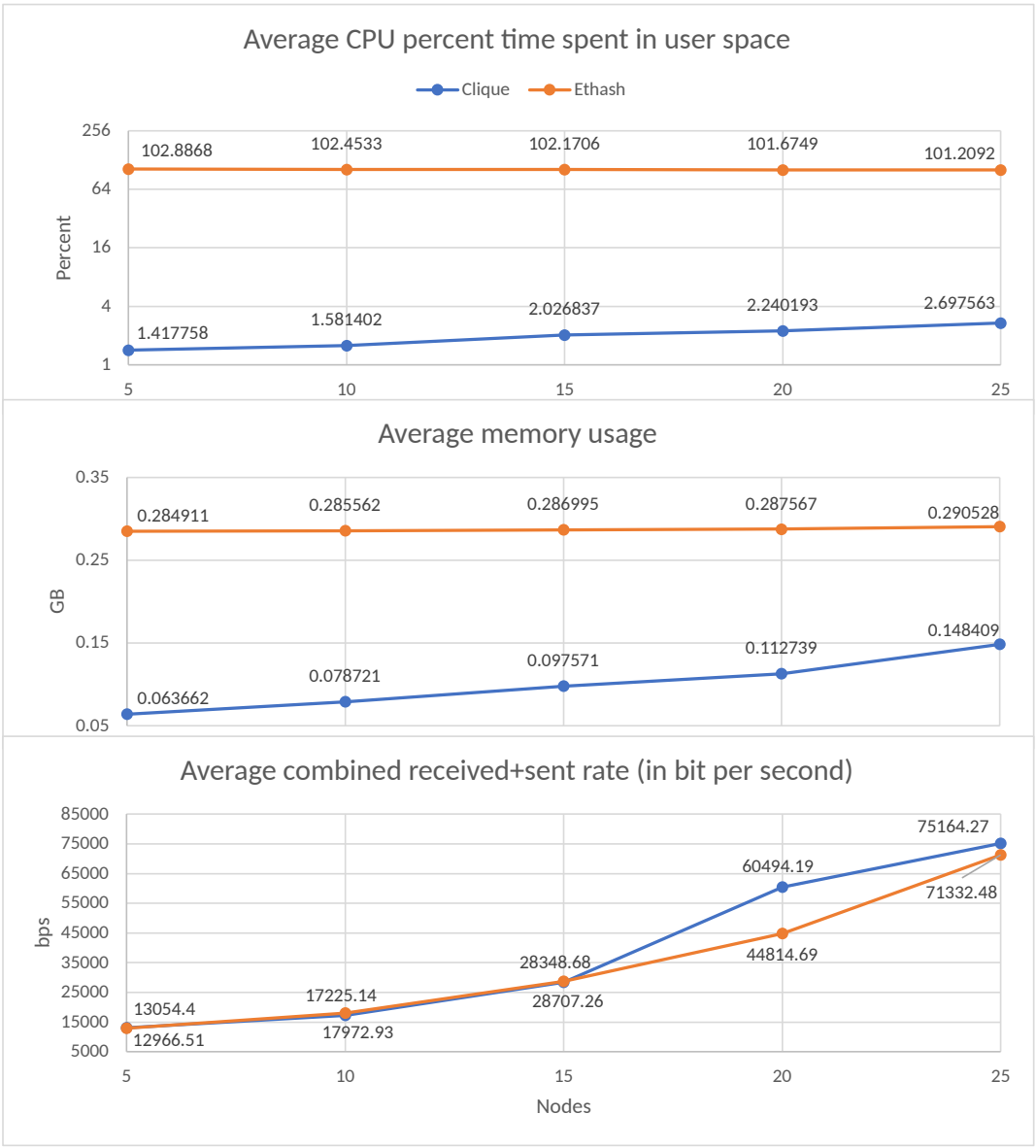


Figure 6.5: Resource monitoring results for the Clique and Ethash networks showed side by side.

6.4.1 CPU Usage

When comparing the CPU usage between the Clique and Ethash networks, we observe the difference immediately. The CPU usage for the Ethash network configurations is a lot higher than that of the Clique network configurations, even though the Ethash nodes were limited to one CPU core each for their mining operation. As previously explained in Section 6.3, the CPU usage for the Ethash nodes is decreasing at around 0.33 percent on average each time the number of participating

nodes increases by five. If the host system that the resource monitoring tests were conducted on had infinite CPU resources available, the Ethash CPU usage would most likely be constant at just above 100% instead of slowly decreasing. The Clique network node's CPU usage is, however, only increasing by around 0.25 percent on average each time the number of participating nodes increases by five.

6.4.2 Memory Usage

Comparing Clique and Ethash's memory usage, we see a clear difference in the benchmarks, where Ethash has a higher memory usage comparatively. From the tables in Table 6.2 and Table 6.4, we show that the two networks created a similar number of blocks through the runtime of their benchmarks. As such, a different explanation is required for this difference in memory usage between the algorithms. One major difference in their operation is that Ethash makes use of a Directed Acyclic Graph (DAG) to store sets of random values used in creating a block hash. This DAG can either be generated and stored before starting the mining process or it can be generated as needed, however then the client is required to wait for the generation to complete. The Geth client pre-generates the DAG before it starts mining, meaning the DAG is stored by the client. Moreover, this means the DAG needs to be moved partially into memory while the client mines blocks. To avoid the DAG generation interfering with the benchmarking data, we let the clients complete the generation before beginning the monitoring of resource usage data.

Another difference between Clique and Ethash, is that the memory usage for Clique grows faster than Ethash's usage. We are unsure as to the exact reasoning behind this growth difference. However, the average growth difference between the algorithms is 0.016 GB, where Clique grows with 0.017 GB on average per five nodes added and Ethash grows with 0.001 GB on average.

6.4.3 Network Usage

In terms of network usage, we observe that Clique and Ethash have a similar usage up to 15 nodes. Subsequently, in the observed data, Clique exhibits a noticeable increase compared to Ethash when the number of nodes reaches 20. However, towards the end, with 25 nodes, Clique's values converge back towards Ethash's values once again. Disregarding the outlier, the network usage reflects our expectations, as the nodes will most often be communicating blocks with the other nodes in the network. It is unclear what resulted in the outlier for the benchmark with 20 nodes, as the only parameter that changed between the benchmarks is the number of participating nodes in the network. On average, Clique grows by 12422 bits per second per five nodes added, with Ethash growing by 11673 bits per second on average. This average skewed by the outlier, and with more thorough testing, could reflect similar values for the two networks.

6.4.4 Summary

In summary, we can conclude that Ethash has a high CPU usage, which was expected given its PoW consensus mechanism, with Clique having a comparatively low CPU usage. Moreover, from our results, we gather that Clique grows more in memory usage, whereas Ethash will have a more conservative growth. Lastly, the algorithms have a similar network consumption, meaning the different networks provide little to no difference in terms of the amount of data required to communicate the blockchain and changes to it.

Chapter 7

Discussion

Following the implementation of our solution and producing benchmarking results, we are going to discuss the aspects of our project we could see improvements to or areas which worked against us in this chapter.

7.1 Failed Attempts with Selected Networks

As mentioned in Section 5.2 and Section 5.4, we tried and failed at setting up Bitcoin and Hyperledger Sawtooth networks and benchmark their clients. All in all, we spent nearly 3 weeks working with the Bitcoin Core client, and during that time we went through 4 versions of the client. We noticed the nodes connected to each other and when we tried sending transactions they were propagated from one node to the others. However, having spent a lot of time on attempting, and failing, at getting Bitcoin to work properly, we were surprised by the ease with which we succeeded in getting the Ethereum network up and running. This process was helped by a direct guide provided by the Geth documentation, with an end-to-end example for setting up a private network with the Geth Ethereum client. However, the documentation was not without its flaws. Often the explanation for application flags were lack luster and incomplete. Furthermore, the configuration of the genesis block was described through examples, with no clarification as to what the different parameters exactly do. Following Ethereum, we were excited to work with another blockchain client. Looking through the documentation for Hyperledger Sawtooth, we found it provided ready-to-go Docker Compose files with official Docker Images. As such, we expected Hyperledger Sawtooth to run with relative ease. However, we ran into similar issues with Hyperledger Sawtooth as we did with Bitcoin Core. The difference is that the current version of Hyperledger Sawtooth is expected to be able to create blocks automatically. This was however not an issue, as we figured that we could easily feed the network with transactions which would prompt the creation of blocks via the shell client. The problem was further exaggerated by the Hyperledger community pointing in different directions, about which version of Hyperledger Sawtooth should be functional. Further complicating the matter, is the issue of Sawtooth relying on a set of containers for a single network node, as explained in Section 5.4.1. Each of these containers uses their own Docker Images, with different version naming. As such, we followed different leads from different community members for several weeks with no luck. An issue we were unable to clear up, was using the wrong version of Docker and Compose files. We gathered that the community members with functioning networks were using a different version of Docker from the one we were using. However, when we attempted to install the same version of Docker, we found that it was only distributed for older versions of our server's operating system. Downgrading our server's operating system version and installing a different Docker and Compose version did not result in a properly functioning network. This led us to drop work on the Hyperledger Sawtooth network, and we shortly pursued alternate networks, but decided against starting work on a new network as we were nearing the end of the project.

7.2 Private Versus Public Blockchain Networks

For our solution, we decided to conduct our benchmarking with a private network. As already mentioned in Chapter 3, this provides us with control over the components of the network. However, this provided some issues in setting up these networks. In our previous work [7], we made our own implementations of the blockchain clients. This granted us control over how the nodes communicate, which made creating the private networks easier. Now, working with existing software, we were limited to the configuration tools provided by the developers of the clients. While these tools often provide a way to create a private network, they are not guaranteed to be available. Moreover, often they require predetermining fixed IP addresses for the nodes in the private network. As such, creating private networks is tedious, though not impossible. But what could public networks offer instead? If we used a public network when benchmarking blockchain clients, we would lose out on the total network control offered by private networks. Moreover, for blockchains which test smart contracts before appending them to the blockchain, it would introduce variations in the different nodes' CPU and memory usages, making the monitoring data flawed. However, public networks would remove the tedium work of setting up private networks, as they make use of seed nodes to get new nodes set up in the blockchain network. Moreover, public networks are the most used type in real deployments, and measuring them would have had a stronger impact. The seed nodes keep an up-to-date list of active nodes in the network and provide new nodes with a list of nodes they should contact to get a copy of the blockchain and get started with creating new blocks [67]. The configuration step for private networks would benefit from utilizing seed nodes. This is showcased with the Geth Ethereum client, where the bootstrap node acts as a seed node, as explained in Section 5.3.1. It could eliminate some configuration required for each blockchain network and make it easier to benchmark more blockchain clients.

7.2.1 Public Networks and Transactions

The aim of this project was to produce more realistic results, compared to our previous work [7], by testing off-the-shelf solutions. But, the blockchain networks that were benchmarked in this project did not take transactions into account. Public networks do have transactions, and they are a big part of blockchains as they affect the resource consumption of blockchain nodes even if the blockchain supports smart-contracts or not. All transactions must undergo processing by a blockchain node and have their relevant transaction data incorporated into blocks if their data is to be included on the blockchain. Consequently, the inclusion of transaction data also leads to the expansion of the block size. For example, on June 5th, 2023, the Ethereum blockchain recorded a significant daily volume of transactions,

reaching approximately 1.152 million transactions [68]. Instead of feeding our operational networks with test or empty transactions, which could potentially yield more precise benchmarking data, we primarily focused on benchmarking as many blockchains as possible instead, keeping with the private networks.

7.3 Limitations in the Server Specifications

As mentioned in Section 6.1, we used a server provided by Aalborg University for testing our solution. More specifically, this server was a virtual machine running on a server owned by the university, and we requested its creation using a web interface. As such, it was possible for us to scale the server's specifications to fit our requirements. Originally, we had a server with 16 processor cores and 32 GB of memory. This server was also used in our previous work to function as a testing bed for our previous solution. Following our previous work, we decided to investigate the possibility of getting more processing cores in our server and reached out to IT services about this issue. After a bit of communication back and forth, we arrived at having our instance limit increased to allow for the largest server available through the web interface. This would have been a server with 64 processing cores and 128 GB of memory. However, after attempting several times to have our server upgraded to the extent of our limits, we concluded that such a server was unable to be hosted on the university's servers. As such, we settled for the next largest server, which provided us with 32 processing cores and 64 GB of memory, as previously specified in Table 6.1. If we had had access to the requested server, it would have provided us the opportunity to extend our testing to networks with up to 60 nodes, in 5 node increments, similar to how we already structured testing our solution. The reason for limiting the size of our blockchain network to, at most, 30 nodes, was to allow each node an entire processing core. This is particularly important for our PoW testing, as the nodes would max out the provided processing cores.

7.4 Finding the Correct Ethash Mining Difficulties

Determining the appropriate starting difficulty for each Ethash network configuration turned out to be a time-consuming task. The lack of guidance or clues in the official documentation regarding difficulty added to the challenge. The difficulty had to be specified in hexadecimal format within the genesis block file before running the network. However, the most time-consuming aspect was the necessity to wait for a specific duration to calculate the average block creation time. This was due to the inherent randomness in block creation time caused by the PoW consensus mechanism in Ethash. When looking at the tables in Table 6.3 and Table 6.4, it

can be seen that the difficulty, in our experience at least, doesn't increase at a fixed amount each time we add five more participating nodes in an Ethash network. As an example, during the monitoring test with five participating nodes, an average of 4.8 blocks per minute were mined. However, when the difficulty was doubled for the monitoring test with ten participating nodes, the average increased to 5.3 blocks per minute. While the randomness factor in the PoW consensus mechanism could partially account for the 0.5 increase, it is worth noting that each monitoring test lasted for an entire hour. The long monitoring duration makes it challenging to attribute the entire increase solely to the randomness factor. We could, of course, have spent a bit more time to fine tune the difficulty, but at some point, after spending a lot of time on it, we decided to stop and go with the found difficulties for the moment.

7.5 Potential Project Pivot

As the spring semester progressed, and we began to configure and test potential private blockchain networks, we quickly realized just how difficult it actually was. Halfway through the semester, we had realized that the Bitcoin Core network was not viable anymore, and we were in the middle of getting the Ethereum networks up and running. Meaning that halfway through the semester, we had no proof of concept, or data to show for our work. It was at this point where we, in coordination with our supervisors, agreed on a hard deadline. If we were unable to successfully establish any functioning networks before the upcoming deadline a week later, we would pivot our focus and work on something else for our Masters, such as extending our framework from last semester [7]. We did, fortunately enough, get both of the Ethereum networks to work as intended before the deadline.

7.6 Ethereum Clients

There are more blockchain clients that can connect and operate on the Ethereum blockchain than just the Geth client. The Geth client is only used by around 66% [69] of Ethereum's users, while other clients cover the rest. Several other Ethereum clients, such as Erigon, Go Quadrans, Bor, and Nethermind, are some of the clients that fill out the rest of the gap according to [69]. We chose the Geth client as it was the most popular, has good documentation, and can run in a private network configuration, but some of these other clients could have been used as well. In our project, we decided to prioritize benchmarking multiple blockchains with a single client, rather than focusing on a single blockchain with multiple clients. However, it would be highly intriguing to conduct benchmarking on various clients operating within the same blockchain, observing and analyzing the variations in

their resource consumption.

Chapter 8

Conclusion

This project set out to accomplish the work described in the problem statement, which was presented in Section 3.1. In short, we set out to find how we could benchmark the resource consumption of popular blockchain clients inside private networks.

In Chapter 4 we designed a system to achieve the goal of the problem statement. This system uses Docker to orchestrate and manage a set of participating nodes in a private blockchain network. Furthermore, it utilizes Glances to monitor the resource usage of each Docker container running the network nodes. Having defined this system design, we create an example implementation on a server provided by Aalborg university. In this implementation, we successfully set up two private blockchain networks. The first being the Ethereum Clique, and the second being the Ethereum Ethash network, both utilizing the official Ethereum client, Geth. The implementation and configuration of both networks can be seen in Section 5.3 alongside the configuration of three other networks, namely, Bitcoin, Hyperledger Sawtooth PoET, and Hyperledger Sawtooth PBFT, which we unfortunately didn't get to a completely working state. However, as specified in the design chapter, resource consumption was monitored with the Glances framework, which enabled us to compare and benchmark the results in Chapter 6. In this chapter, we present and discuss our monitoring data, ending in a comparison between the working blockchain networks.

In Chapter 1, we describe how this project is an extension of our previous work, where we benchmarked our own implementations of consensus mechanisms. As mentioned in the introduction, we wanted to benchmark real blockchain clients in this work, instead of our own custom implementations. This achieves more realistic results as the more off-the-shelf solutions, that we utilized in this project, are actively used by the current blockchain community.

As a result of the work done in this project, we can conclude our solution achieves the goal as described in our problem statement. It presents the feasibility

of testing blockchain clients and compare them against each other, with a focus on the computational resource consumption of the different clients. The solution presented in this report acts as a proof-of-concept for using off-the-shelf software for benchmarking blockchain clients, and is not finished in its current state. The next chapter, Chapter 9, describes steps to improve the solution.

Chapter 9

Future work

Having concluded our project and a successful proof-of-concept, we would like to peer to the future of the solution proposed in this paper. As such, this chapter will present some tasks we could see as the next steps towards improving our solution.

9.1 Generalize Network Setup

As presented in Section 7.2, seed nodes in blockchain networks make it easy to connect new nodes to the blockchain network. One issue with the solution in its current state is the need to manually configure the network through the limited configuration tools made available by the developers of each blockchain client. If we could incorporate a seed node into the private network used to benchmark the blockchain clients, it could remove the manual configuration requirement for setting up the private networks. This would allow us to provide a simple Dockerfile and have the network sort itself out with the seed node.

9.2 Add a Graphical User Interface

A graphical user interface (GUI) would improve the overall usability of the solution. It would ease the process of starting a benchmark test, by allowing users to input a Dockerfile which builds a Docker image of the blockchain client and then start a network with some number of nodes. The GUI could then be given a way to extract the blockchain length from the network and provide feedback as to the progress of the benchmark. Once the benchmark is completed, it can present the user with the results of the benchmark by outputting a CSV file and making graphs from the data.

9.3 Automate Data Processing

Throughout our resource consumption monitoring in this project, we have utilized the Glances framework to monitor the containers. After a benchmarking test was finished, we would reorder the data with the *ThesisDataManager*, as described in Section 5.1.1, and do a lot of manual work in Excel to produce the graphs and tables that are shown throughout this report. This process could be simplified with the use of another monitoring tool, such as Prometheus, as mentioned in Section 4.3, which includes the data processing and visualization add-on Grafana. We chose Glances as it seemed like the simpler monitoring tool, which it is, but we could not have predicted the need for reordering the data, and the amount of manual work to get our respective figures. We thence expect that implementing Prometheus could lighten the manual work burden for the data processing and visualization.

9.4 Get Bitcoin Core and Hyperledger Sawtooth to Work

We did, in the end, not succeed in getting either the Bitcoin, Hyperledger Sawtooth PoET, or Hyperledger Sawtooth PBFT networks to function properly. But if more

time was allocated, both networks could perhaps get to a working state.

There were several Bitcoin Core versions that we didn't try, there are a total of 65 versions [58], and we were perhaps close to picking a version which actually worked, we do not know. As described in Section 5.2, the newer versions of Bitcoin clients rely on either GPU or ASIC mining, which means that adding GPU computation to our server could potentially result in a working Bitcoin client. Another direction would be to try another Bitcoin client, instead of the Bitcoin Core client.

As previously mentioned in Section 5.4, we communicated with Hyperledger Sawtooth users, who successfully configured a PBFT network on Hyperledger's official Discord server [63]. This does point towards it being possible to get a Hyperledger Sawtooth PBFT network to function properly that can create and propagate transactions and blocks among its network nodes. Given more time, we could perhaps get the PBFT network to work, even though we are of the belief that we have tried everything already.

Moreover, even though Hyperledger Sawtooth has a PoET implementation available, it has been deemed as "dead" by the community. We are unsure as to why exactly PoET is nonfunctional, but after reading this on the Hyperledger Discord, we promptly decided not to proceed with the PoET network. If the PoET is functional at a point in the future, it would be quick to benchmark, following success with setting up Hyperledger Sawtooth with PBFT.

9.5 Test Additional Blockchains

We have successfully proven that our solution acts as a proof-of-concept for using off-the-shelf software for benchmarking blockchain clients. But we can strengthen our proof by benchmarking several blockchain clients instead of only two. A future work task would be to continue the work of testing blockchain clients and gather more data. Potential blockchain clients that could be benchmarked next are for example: EOS [70], Hyperledger Caliper [15], Hyperledger Fabric [71], and Hyperledger Besu [72].

Bibliography

[1] Monash University. *Citing and referencing: Vancouver*. URL: https://guides.lib.monash.edu/citing-referencing/vancouver. (accessed: 07.06.2023).

[2] Sergi López-Sorribes, Josep Rius-Torrentó, and Francesc Solsona-Tehàs. "A Bibliometric Review of the Evolution of Blockchain Technologies". In: *Sensors* 23.6 (2023). ISSN: 1424-8220. DOI: 10.3390/s23063167. URL: https://www.mdpi.com/1424-8220/23/6/3167. (accessed: 10.05.2023).

[3] CoinGecko. *Global Cryptocurrency Market Cap Charts*. URL: https://www.coingecko.com/en/global-charts. (accessed: 10.05.2023).

[4] Statista. *Overall cryptocurrency market capitalization per week from July 2010 to April 2023 (in billion U.S. dollars)*. URL: https://www-statista-com.zorac.aub.aau.dk/statistics/730876/cryptocurrency-maket-value/. (accessed: 10.05.2023).

[5] University of Cambridge Cambridge Judge Business School. *Cambridge Bitcoin Electricity Consumption Index*. URL: https://ccaf.io/cbnsi/cbeci. (accessed: 10.05.2023).

[6] The go-ethereum Authors. *Hardware requirements*. URL: https://geth.ethereum.org/docs/getting-started/hardware-requirements. (accessed: 10.05.2023).

[7] Jeppe Krogh Laursen and Daniel Friis Holtebo. "Consensus mechanisms for a local energy market: A benchmark". In: Aalborg University Project Library, UUID: d7b702db-56ef-4a74-8752-d4f74c588fe2. (accessed: 23.02.2023).

[8] productplan. *Stakeholder Analysis*. URL: https://www.productplan.com/glossary/stakeholder-analysis/. (accessed: 11.05.2023).

[9] coinmarketcap.com. *Client*. URL: https://coinmarketcap.com/alexandria/glossary/client. (accessed: 15.03.2023).

[10] Jake FrankenField. *Merkle Tree in Blockchain: What it is and How it Works*. URL: https://www.investopedia.com/terms/m/merkle-tree.asp. (accessed: 22.03.2023).

[11] celo.org. *Build together and prosper*. URL: https://celo.org/. (accessed: 22.03.2023).

[12] coinbase.com. *Blockchain client types*. URL: https://www.coinbase.com/cloud/discover/dev-foundations/blockchain-client-types. (accessed: 22.03.2023).

[13] originstamp.com. *The 10 Different Types of Blockchain Nodes and How They Work*. URL: https://originstamp.com/blog/the-10-different-types-of-blockchain-nodes-and-how-they-work/. (accessed: 23.03.2023).

[14] Bulat Nasrulin et al. "Gromit: Benchmarking the Performance and Scalability of Blockchain Systems". In: *2022 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. 2022, pp. 56–63. DOI: 10.1109/DAPPS55202.2022.00015.

[15] Hyperledger. *Hyperledger Caliper*. URL: https://hyperledger.github.io/caliper/. (accessed: 31.03.2023).

[16] Hyperledger. *Getting Started*. URL: https://hyperledger.github.io/caliper/v0.5.0/getting-started/. (accessed: 31.03.2023).

[17] prometheus.io. *Prometheus*. URL: https://prometheus.io/. (accessed: 16.05.2023).

[18] Dimitri Saingre, Thomas Ledoux, and Jean-Marc Menaud. "BCTMark: a Framework for Benchmarking Blockchain Technologies". In: *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)*. 2020, pp. 1–8. DOI: 10.1109/AICCSA50499.2020.9316536. (accessed: 10.05.2023).

[19] Tien Tuan Anh Dinh et al. "BLOCKBENCH: A Framework for Analyzing Private Blockchains". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, 1085–1100. ISBN: 9781450341974. DOI: 10.1145/3035918.3064033. URL: https://doi.org/10.1145/3035918.3064033. (accessed: 10.05.2023).

[20] Binance. *Markets Overview*. URL: https://www.binance.com/en/markets/overview. (accessed: 31.03.2023).

[21] Crypto.com. *Today's Cryptocurrency Prices*. URL: https://crypto.com/price. (accessed: 31.03.2023).

[22] CoinGecko. *Cryptocurrency Prices by Market Cap*. URL: https://www.coingecko.com/. (accessed: 31.03.2023).

[23] CoinMarketCap. *Today's Cryptocurrency Prices by Market Cap*. URL: https://coinmarketcap.com/. (accessed: 31.03.2023).

[24] Ethereum Community. *ERC-20 Token Standard*. URL: https://ethereum.org/en/developers/docs/standards/tokens/erc-20/. (accessed: 31.03.2023).

[25] Tether Operations Limited. *FAQ*. URL: https://tether.to/en/. (accessed: 31.03.2023).

[26] Build N Build. *BNB Chain: An Ecosystem of Blockchains*. URL: https://docs.bnbchain.org/docs/overview. (accessed: 10.04.2023).

[27] Circle Internet Financial Limited. *USD Coin*. URL: https://www.circle.com/en/usdc. (accessed: 10.04.2023).

[28] CENTRE. *Introducing USD Coin*. URL: https://www.centre.io/usdc. (accessed: 10.04.2023).

[29] Circle Internet Financial Limited. *Euro Coin*. URL: https://www.circle.com/en/euro-coin. (accessed: 10.04.2023).

[30] XRP Ledger. *Your Questions About XRP, Answered*. URL: https://xrpl.org/xrp-overview.html. (accessed: 11.04.2023).

[31] Ethereum Foundation. *Introduction to Web3*. URL: https://ethereum.org/en/web3/#what-is-web3. (accessed: 11.04.2023).

[32] XRP Ledger. *XRPL Use Cases*. URL: https://xrpl.org/uses.html. (accessed: 11.04.2023).

[33] IOHK. *Why use Cardano?* URL: https://docs.cardano.org/new-to-cardano/why-use-cardano. (accessed: 12.04.2023).

[34] Cardano. *What is Ada?* URL: https://cardano.org/what-is-ada/. (accessed: 12.04.2023)).

[35] Dogecoin Foundation. *The Dogecoin Manifesto*. URL: https://foundation.dogecoin.com/manifesto/. (accessed: 12.04.2023).

[36] The Dogecoin Foundation & Dogecoin Project. *What is a miner?* URL: https://dogecoin.com/dogepedia/articles/what-is-a-miner/. (accessed: 12.04.2023).

[37] Ethereum Community. *Scaling*. URL: https://ethereum.org/en/developers/docs/scaling/#layer-2-scaling. (accessed: 17.04.2023).

[38] Polygon Community. *What is Polygon?* URL: https://wiki.polygon.technology/docs/home/polygon-basics/what-is-polygon. (accessed: 17.04.2023).

[39] Solana Foundation. *Solana Documentaion*. URL: https://docs.solana.com/. (accessed: 17.04.2023).

[40] Ethereum Community. *Blocks*. URL: https://ethereum.org/en/developers/docs/blocks/#block-time. (accessed: 17.05.2023).

[41] Ethereum Community. *Sidechains*. URL: https://ethereum.org/en/developers/docs/scaling/sidechains/. (accessed: 17.05.2023).

[42] Ethereum Community. *Bridges*. URL: https://ethereum.org/en/developers/docs/bridges/#how-do-bridges-work. (accessed: 17.05.2023).

[43] Ethereum Community. *Introduction to dapps*. URL: https://ethereum.org/en/developers/docs/dapps/. (accessed: 25.05.2023).

[44] R3. *Corda Permissioned Distributed Ledger Technology*. URL: https://r3.com/products/corda. (accessed: 17.05.2023).

[45] R3. *Configuring the Network Participants*. URL: https://docs.r3.com/en/platform/corda/5.0-beta/developing/getting-started/configure-the-network-participants/network-participants.html. (accessed: 17.05.2023).

[46] Databricks. *What is Orchestration?* URL: https://www.databricks.com/glossary/orchestration. (accessed: 21.05.2023).

[47] Walter Glenn. *What Is a "Portable" App, and Why Does It Matter?* URL: https://www.howtogeek.com/290358/what-is-a-portable-app-and-why-does-it-matter/. (accessed: 21.05.2023).

[48] Wikipedia. *Portable Application*. URL: https://en.wikipedia.org/wiki/Portable_application. (accessed: 21.05.2023).

[49] The Kubernetes Authors. *Overview*. URL: https://kubernetes.io/docs/concepts/overview/#why-you-need-kubernetes-and-what-can-it-do. (accessed: 21.05.2023).

[50] Docker. *Containerize an application*. URL: https://docs.docker.com/get-started/02_our_app/#build-the-apps-container-image. (accessed: 29.05.2023).

[51] Docker. *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder. (accessed: 29.05.2023).

[52] Docker.com. *Use Docker Compose*. URL: https://docs.docker.com/get-started/08_using_compose/. (accessed: 29.05.2023).

[53] Tim Fisher. *What Is a Benchmark?* URL: https://www.lifewire.com/what-is-a-benchmark-2625811. (accessed: 17.05.2023).

[54] docker. *docker stats*. URL: https://docs.docker.com/engine/reference/commandline/stats/. (accessed: 17.05.2023).

[55] Nicolas Hennion. *Glances*. URL: https://github.com/nicolargo/glances. (accessed: 17.05.2023).

[56] Nicolas Hennion. *Glances*. URL: https://glances.readthedocs.io/en/latest/. (accessed: 17.05.2023).

[57] prometheus.io. *From metrics to insight*. URL: https://prometheus.io/. (accessed: 17.05.2023).

[58] Bitcoin. *Bitcoin Core*. URL: https://bitcoin.org/en/bitcoin-core/. (accessed: 31.05.2023).

[59] Bitcoin Core developers. *Bitcoin Core 0.13 release notes*. URL: https://github.com/bitcoin/bitcoin/blob/master/doc/release-notes/release-notes-0.13.0.md#removal-of-internal-miner. (accessed: 26.05.2023).

[60] Binance Acedemy. *Proof of Authority Explained*. URL: https://academy.binance.com/en/articles/proof-of-authority-explained. (accessed: 31.05.2023).

[61] The go-ethereum Authors. *Private Networks*. URL: https://geth.ethereum.org/docs/fundamentals/private-network. (accessed: 26.05.2023).

[62] Sawtooth.org. *Introduction*. URL: https://sawtooth.hyperledger.org/docs/1.2/. (accessed: 19.05.2023).

[63] Discord. *Hyperledger Foundation*. URL: https://discord.com/servers/hyperledger-foundation-905194001349627914. (accessed: 29.05.2023).

[64] Bhaskar Kashyap. *PROOF-OF-STAKE (POS)*. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/. (accessed: 31.05.2023).

[65] investopedia.com. *How Do You Calculate R-Squared in Excel?* URL: https://www.investopedia.com/ask/answers/012615/how-do-you-calculate-rsquared-excel.asp. (accessed: 05.06.2023).

[66] Donald McIntyre. *The Ethereum Classic Mining Difficulty Adjustment Explained*. URL: https://ethereumclassic.org/blog/2023-03-15-the-ethereum-classic-mining-difficulty-adjustment-explained. (accessed: 02.06.2023).

[67] Paul Valencourt Brenn Hill Samanyu Chopra and Narayan Prusty. *DNS seeds*. URL: https://www.oreilly.com/library/view/blockchain-developers-guide/9781789954722/ccb46585-403e-44dd-a0a9-253bb48f2736.xhtml. (accessed: 06.06.2023).

[68] ycharts.com. *Ethereum Transactions Per Day*. URL: https://ycharts.com/indicators/ethereum_transactions_per_day. (accessed: 06.06.2023).

[69] etherscan.io. *Ethereum Node Tracker*. URL: https://etherscan.io/nodetracker. (accessed: 06.06.2023).

[70] eos.io. *eosio*. URL: https://eos.io/. (accessed: 06.06.2023).

[71] hyperledger.org. *Hyperledger Fabric*. URL: https://www.hyperledger.org/use/fabric. (accessed: 06.06.2023).

[72] hyperledger.org. *Hyperledger Besu*. URL: https://www.hyperledger.org/use/besu. (accessed: 06.06.2023).

Appendix A

Test Results

This chapter showcases the resource monitoring graphs of all functional blockchain node networks, excluding those presented in Chapter 6, Results. The first graph is located on the next page.

A.1 Ethereum - Clique

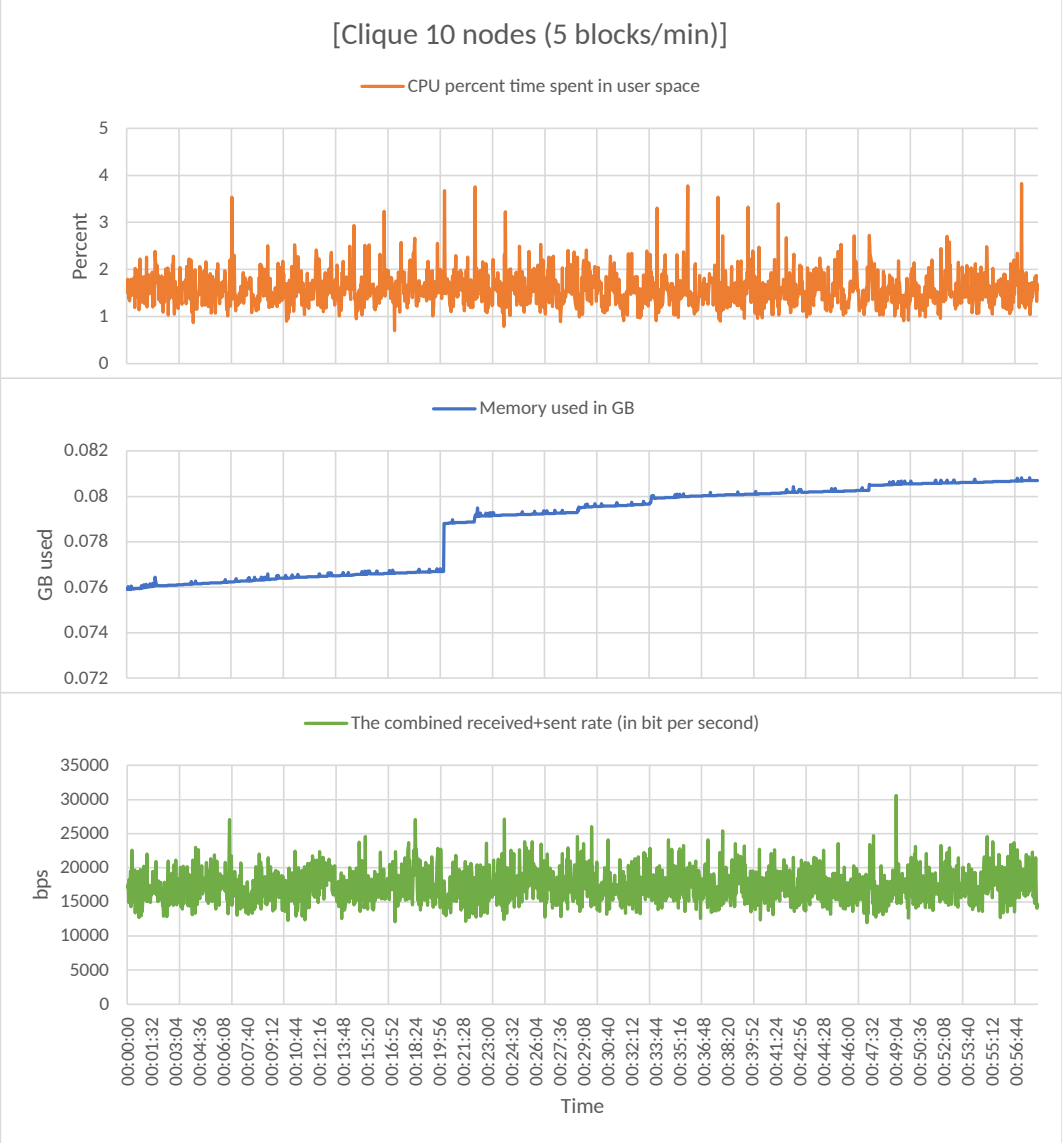


Figure A.1: One node's resource consumption when running the Clique network with 10 nodes for 60 minutes.

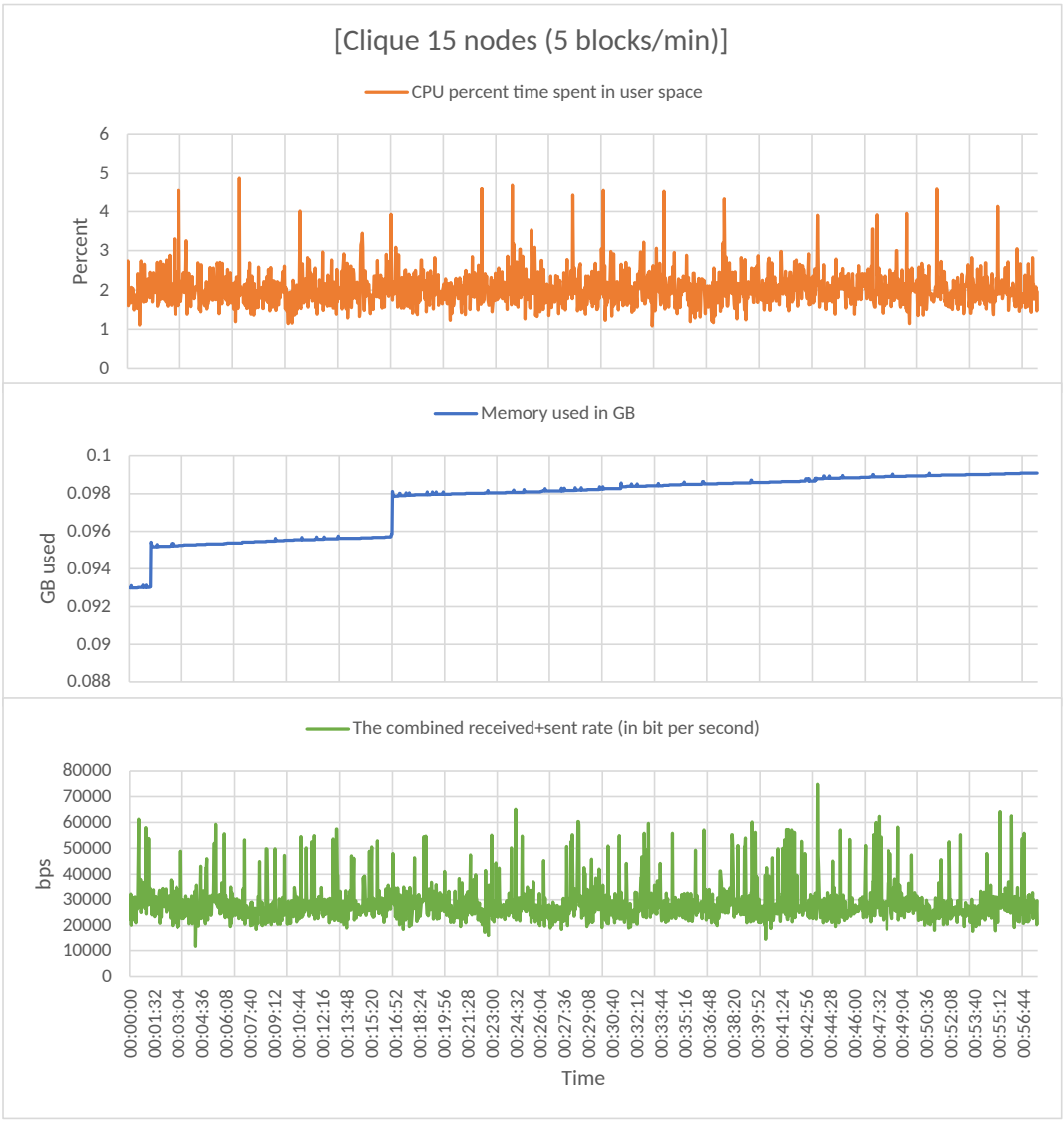


Figure A.2: One node's resource consumption when running the Clique network with 15 nodes for 60 minutes.

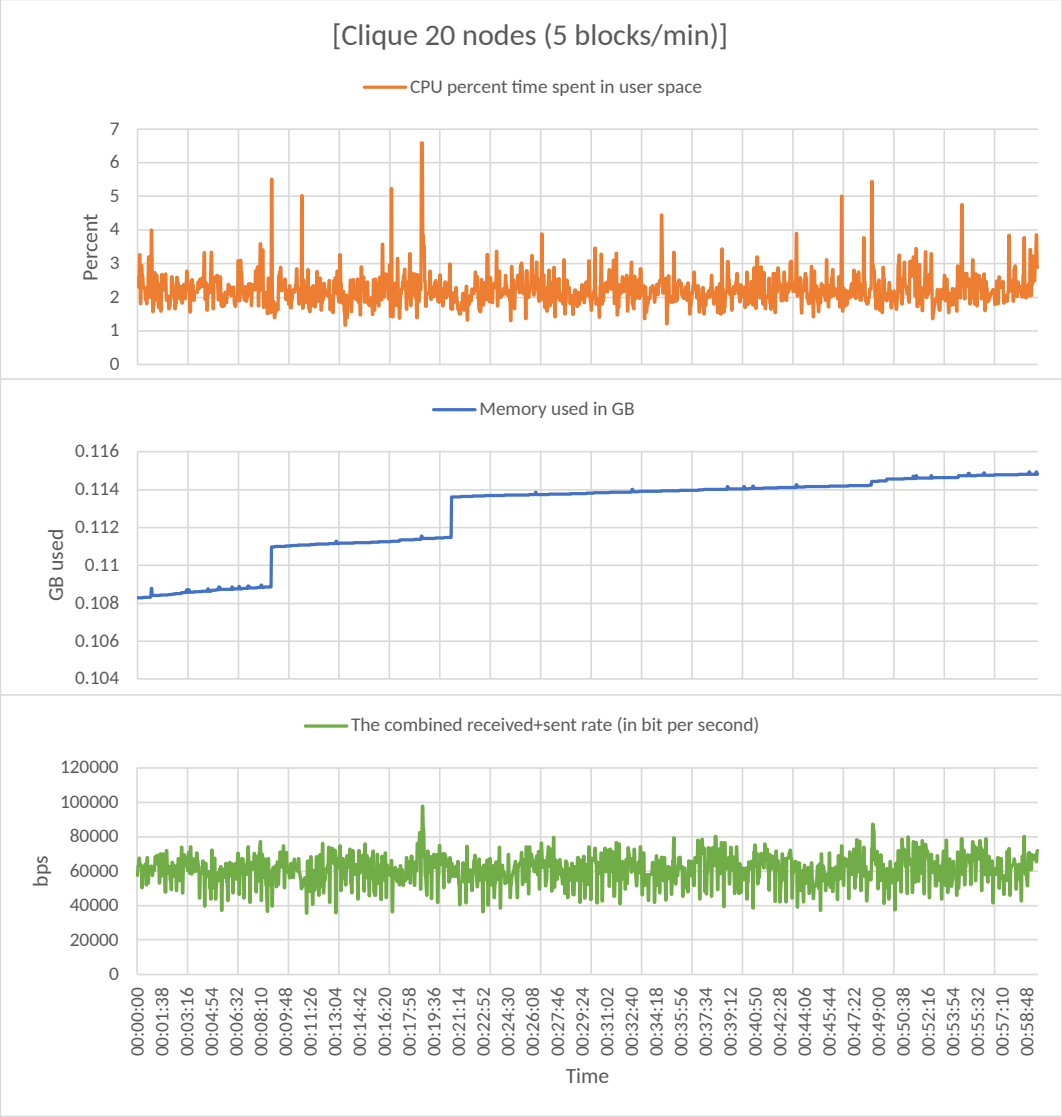


Figure A.3: One node's resource consumption when running the Clique network with 20 nodes for 60 minutes.

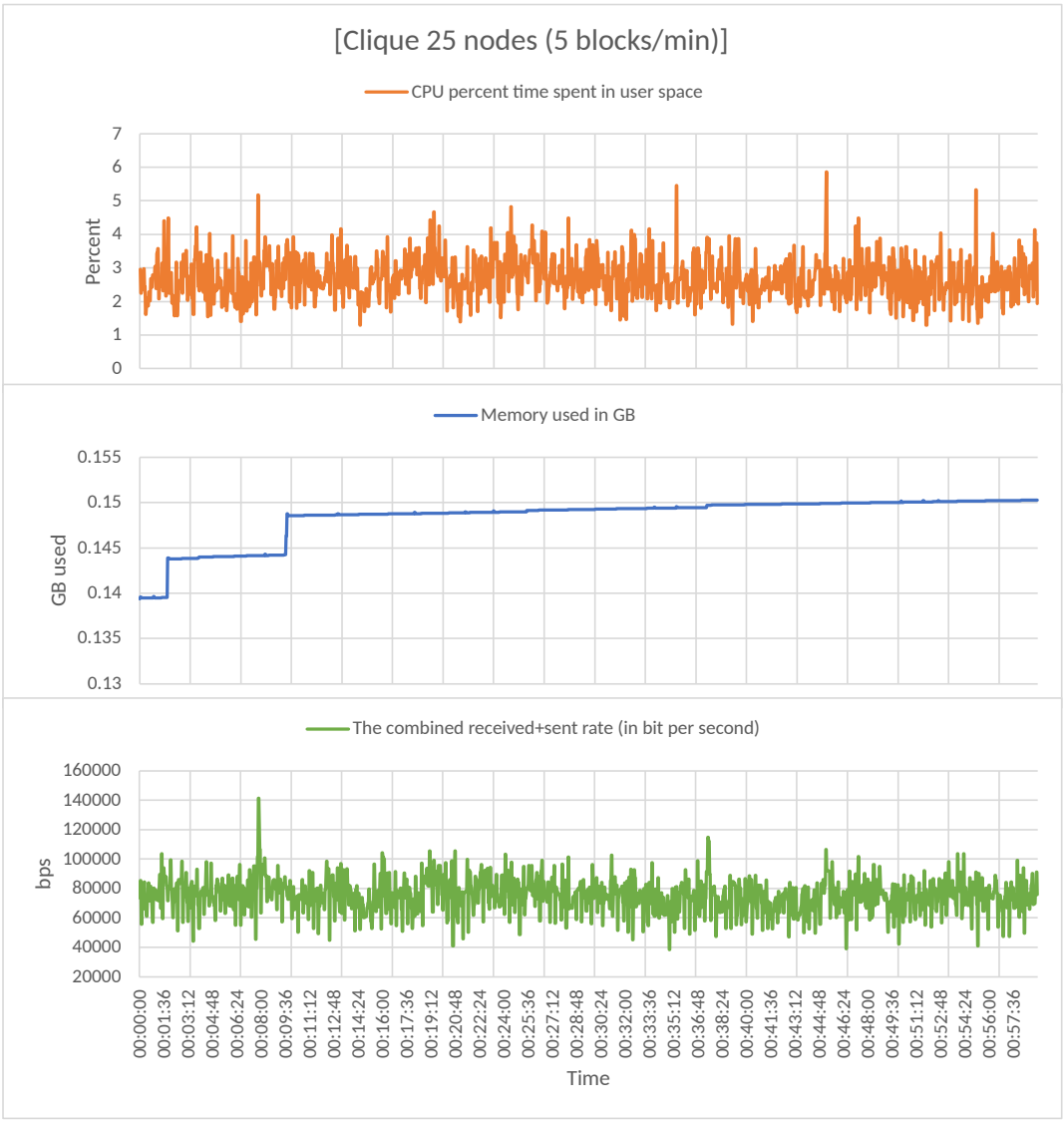


Figure A.4: One node's resource consumption when running the Clique network with 25 nodes for 60 minutes.

A.2 Ethereum - Ethash

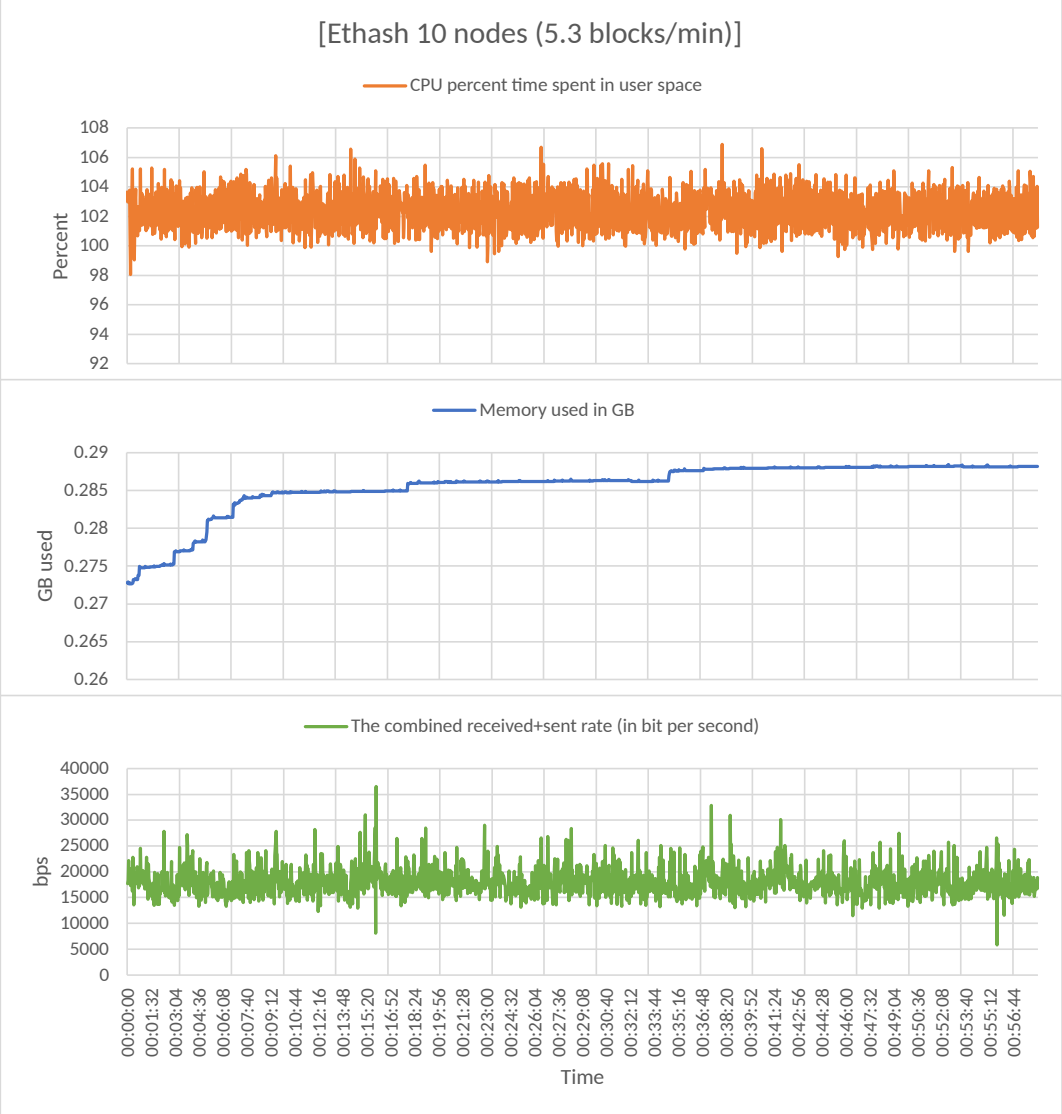


Figure A.5: One node's resource consumption when running the Ethash network with 10 nodes for 60 minutes.

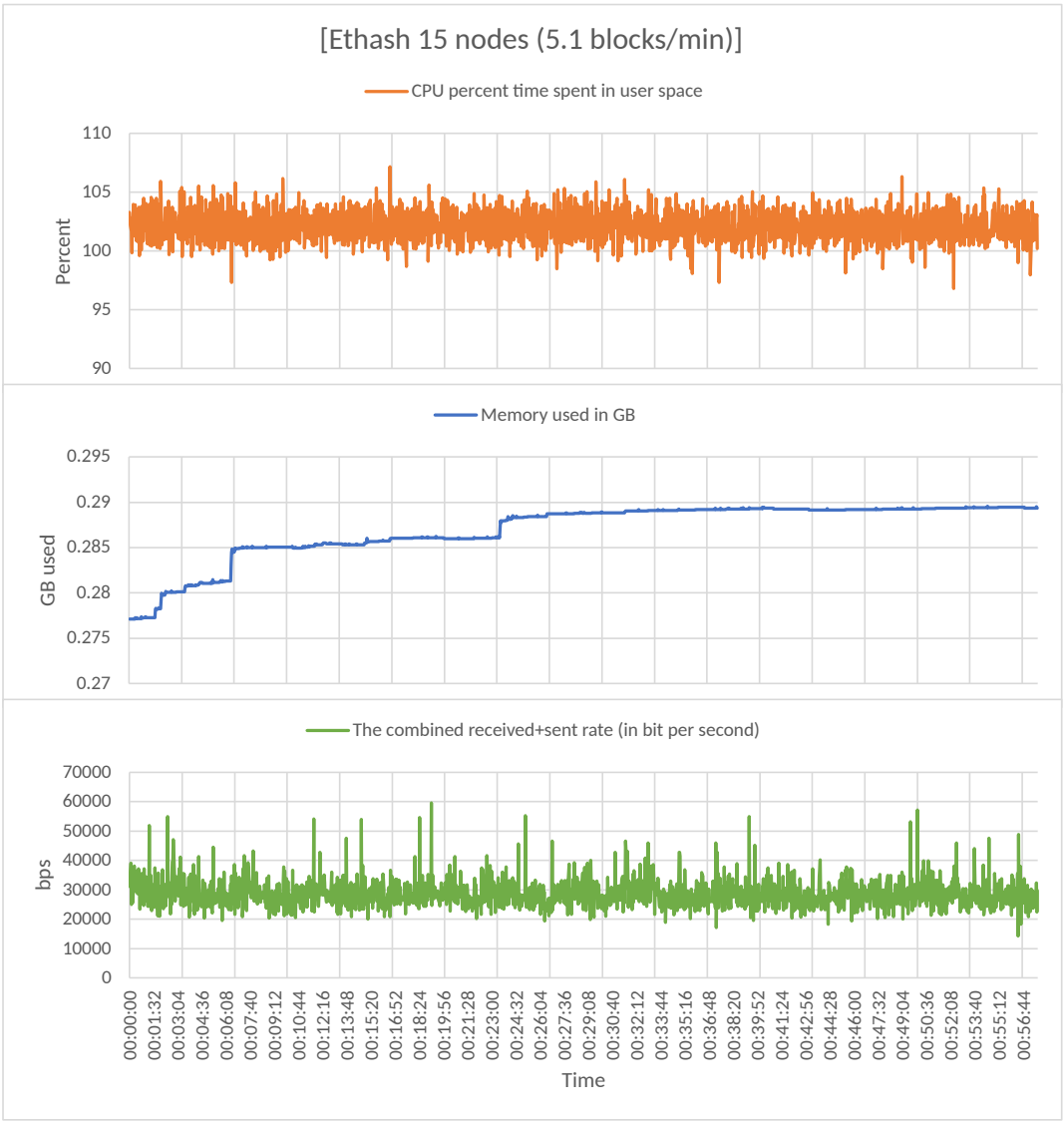


Figure A.6: One node's resource consumption when running the Ethash network with 15 nodes for 60 minutes.

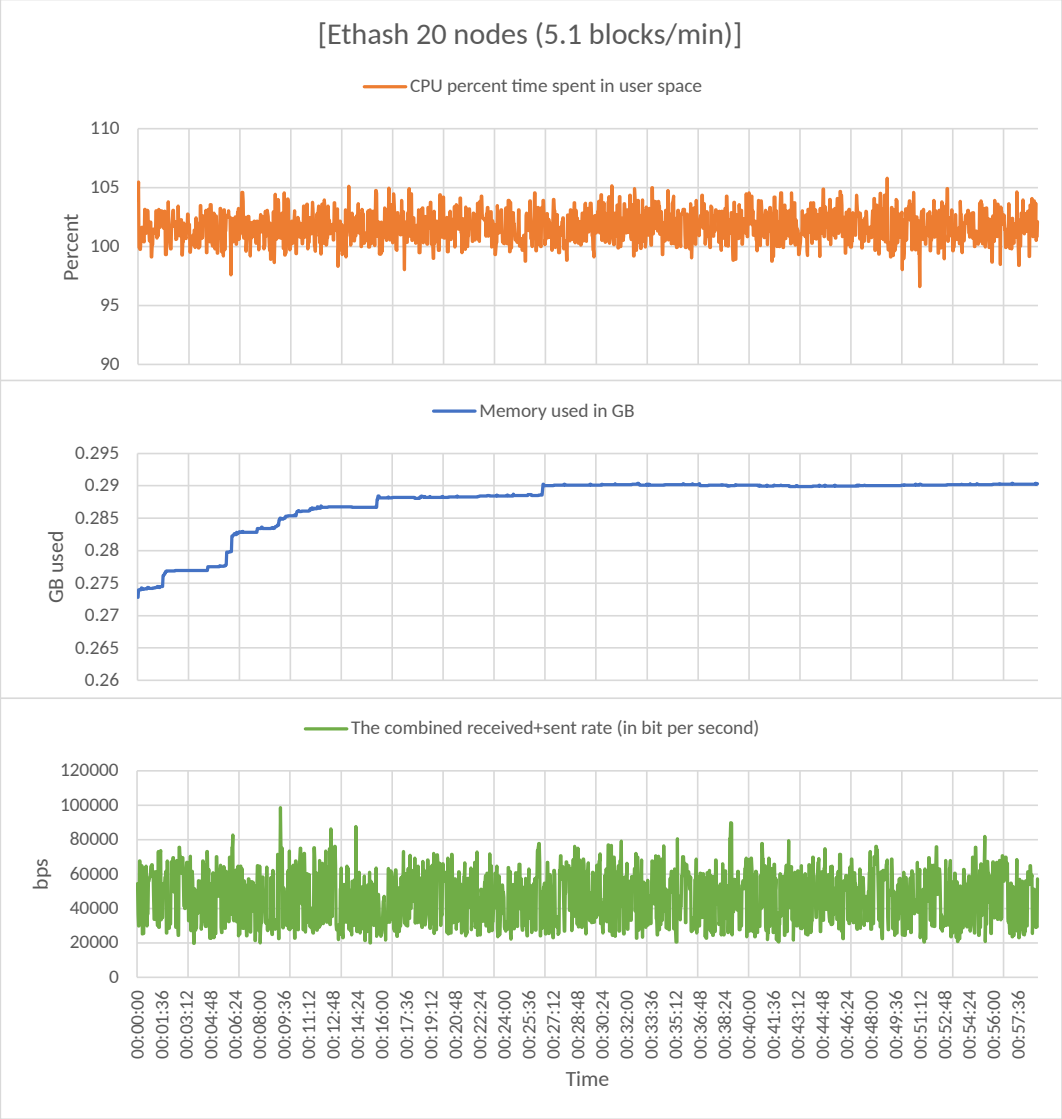


Figure A.7: One node's resource consumption when running the Ethash network with 20 nodes for 60 minutes.

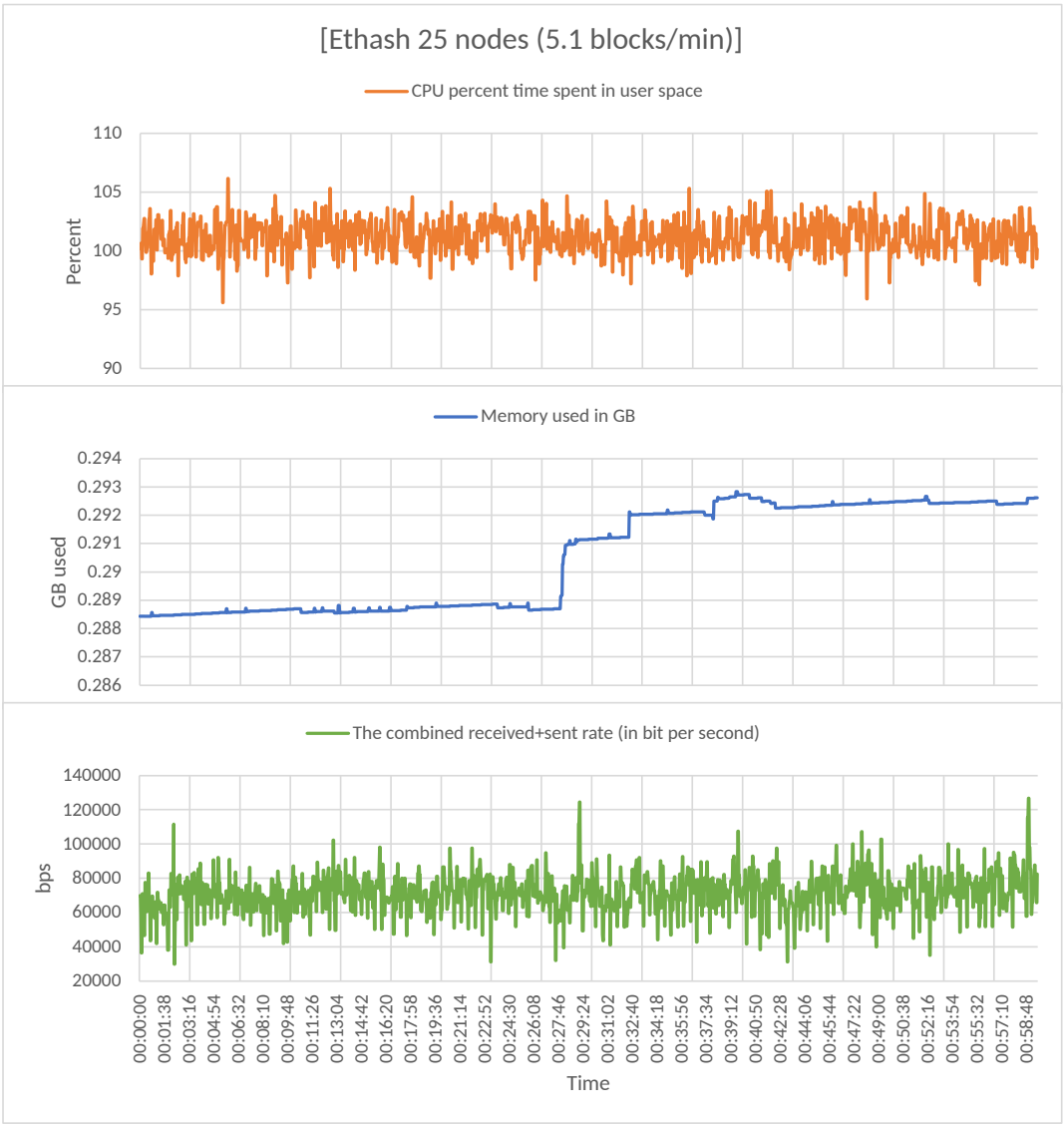


Figure A.8: One node's resource consumption when running the Ethash network with 25 nodes for 60 minutes.

Appendix B

Bitcoin Resources

This appendix chapter will showcase the Dockerfile which was used for the Bitcoin Core client.

Listing B.1: This Dockerfile shows the first Bitcoin Core Dockerfile.

```
FROM debian:bullseye-slim

RUN useradd -r bitcoin \
  && apt-get update -y \
  && apt-get install -y curl \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ENV BITCOIN_VERSION=0.12.1
ENV BITCOIN_DATA=/home/bitcoin/.bitcoin
ENV PATH=/opt/bitcoin-${BITCOIN_VERSION}/bin:$PATH

RUN curl -SLO https://bitcoincore.org/bin/bitcoin-core-${BITCOIN_VERSION}/
bitcoin-${BITCOIN_VERSION}-linux64.tar.gz \
  && tar -xzf *.tar.gz -C /opt  \
  && rm *.tar.gz \
  && rm -rf /opt/bitcoin-${BITCOIN_VERSION}/bin/bitcoin-qt

COPY bitcoin-conf1.conf /root/.bitcoin/bitcoin.conf

EXPOSE 8332 8333 18332 18333 18443 18444 38333 38332

CMD ["bitcoind"]
```

Listing B.2: This Dockerfile shows the second Bitcoin Core Dockerfile.

```
FROM debian:bullseye-slim

RUN useradd -r bitcoin \
  && apt-get update -y \
  && apt-get install -y curl \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ENV BITCOIN_VERSION=0.12.1
ENV BITCOIN_DATA=/home/bitcoin/.bitcoin
ENV PATH=/opt/bitcoin-${BITCOIN_VERSION}/bin:$PATH

RUN curl -SLO
    https://bitcoincore.org/bin/bitcoin-core-${BITCOIN_VERSION}/bitcoin-
${BITCOIN_VERSION}-linux64.tar.gz \
  && tar -xzf *.tar.gz -C /opt  \
  && rm *.tar.gz \
  && rm -rf /opt/bitcoin-${BITCOIN_VERSION}/bin/bitcoin-qt

COPY bitcoin-conf2.conf /root/.bitcoin/bitcoin.conf

EXPOSE 8332 8333 18332 18333 18443 18444 38333 38332

CMD ["bitcoind"]
```

Appendix C

Ethereum Resources

This appendix chapter will showcase the Ethereum Dockerfiles for the bootstrap node in addition to both a signer and a miner node, as well as the bash script that can start and set up an Ethereum network.

Listing C.1: This Dockerfile shows the BootstrapDockerfile.

```
1  FROM debian:bullseye
2
3  ARG GETH_VERSION
4
5  ENV PATH=/opt/geth-linux-amd64-${GETH_VERSION}:$PATH
6  ENV NETWORK_ID=12345
7
8  RUN mkdir /eth-data
9  RUN mkdir /download-data
10
11 COPY ./download/ /download-data/
12 COPY genesis.json /eth-data/genesis.json
13 COPY password password
14 COPY bootstrap.sh /bootstrap.sh
15 COPY node-0 /root/.ethereum
16
17 RUN chmod +x /bootstrap.sh
18
19 RUN tar -xzf /download-data/*.tar.gz -C /opt \
20   && rm /download-data/*.tar.gz \
21   && geth init /eth-data/genesis.json \
22   && ls /root/.ethereum/keystore | grep -Eo 'Z--.*' | cut -d'-' -f 3 >>
       ethereum-address
23
24 EXPOSE 8551 8545 30303
```

```
25
26 CMD ["sh", "-c", "/bootstrap.sh"]
```

Listing C.2: This Dockerfile shows the SignerDockerfile.

```
1  FROM debian:bullseye
2
3  ARG INDEX
4  ARG GETH_VERSION
5
6  ENV PATH=/opt/geth-linux-amd64-${GETH_VERSION}:$PATH
7  ENV NETWORK_ID=12345
8
9  RUN mkdir /eth-data
10 RUN mkdir /download-data
11
12 COPY ./download/ /download-data/
13 COPY genesis.json /eth-data/genesis.json
14 COPY password password
15 COPY signer.sh /signer.sh
16 COPY node-${INDEX} /root/.ethereum
17 COPY bootstrap-enode /eth-data/bootstrap-enode
18
19 RUN chmod +x /signer.sh
20
21 RUN tar -xzf /download-data/*.tar.gz -C /opt \
22   && rm /download-data/*.tar.gz \
23   && geth init /eth-data/genesis.json \
24   && ls /root/.ethereum/keystore | grep -Eo 'Z--.*' | cut -d'-' -f 3 >>
     ethereum-address
25
26 EXPOSE 8551 30303
27
28 CMD ["sh", "-c", "/signer.sh"]
```

Listing C.3: This Dockerfile shows the MinerDockerfile.

```
1  FROM debian:bullseye
2
3  ARG INDEX
4  ARG GETH_VERSION
5
6  ENV PATH=/opt/geth-linux-amd64-${GETH_VERSION}:$PATH
7  ENV NETWORK_ID=12345
```

```
8
9  RUN mkdir /eth-data
10 RUN mkdir /download-data
11
12 COPY ./download/ /download-data/
13 COPY genesis.json /eth-data/genesis.json
14 COPY password password
15 COPY miner.sh /miner.sh
16 COPY node-${INDEX} /root/.ethereum
17 COPY bootstrap-enode /eth-data/bootstrap-enode
18
19 RUN chmod +x /miner.sh
20
21 RUN tar -xzf /download-data/*.tar.gz -C /opt \
22   && rm /download-data/*.tar.gz \
23   && geth init /eth-data/genesis.json \
24   && ls /root/.ethereum/keystore | grep -Eo 'Z--.*' | cut -d'-' -f 3 >>
       ethereum-address
25
26 EXPOSE 8551 30303
27
28 CMD ["sh", "-c", "/miner.sh"]
```

Listing C.4: Bash script which starts an Ethereum Clique network with a custom amount of nodes.

```
1  GETH_VERSION=1.11.5-a38f4108
2
3  docker ps --filter name=bootstrap -aq | xargs docker stop | xargs docker rm
4  docker ps --filter name=node-* -aq | xargs docker stop | xargs docker rm
5
6  echo -----Input number of nodes:
7  read nodes
8
9  cd
10 cd thesis/blockchainClientStorage/Ethereum/Clique
11 mkdir -p data
12 cd
13 rm -rf thesis/blockchainClientStorage/Ethereum/Clique/data/*
14 cd
15 cd thesis/blockchainClientStorage/Ethereum/Clique/data
16 echo password > password
17
18 echo -----Cleaned and setup data folder
19
```

```
mkdir -p download
cd download

curl -SLO https://gethstore.blob.core.windows.net/builds/geth-linux-
amd64-$GETH_VERSION.tar.gz

cd ..
echo -----Downloaded blockchain client

for index in $(seq 0 $nodes)
do
    mkdir -p node-$index
    geth --datadir node-$index account new --password password | grep -Eo
    '0x.*' | cut -d'x' -f 2 >> ethereum-addresses
done

cp ../clique-genesis.json genesis.json
sed -i "s/ETH_ADDR/$(cat ethereum-addresses | tr -d '\n')/g" genesis.json

echo -----Added addresses to genesis file

cd ..
cp bootstrap.sh data/bootstrap.sh
docker build --build-arg GETH_VERSION=$GETH_VERSION -t bootstrap-image -f
    BootstrapDockerfile data
docker create --network p10network --ip 192.168.0.2 --name bootstrap
    bootstrap-image
docker start bootstrap

echo -----Finished building, creating, and starting the bootstrap node

sleep 3
docker exec bootstrap geth attach --exec admin.nodeInfo.enr | cut -d'"' -f
    2 > data/bootstrap-enode

echo -----Finished retrieving the bootstrap nodes enode info

cp signer.sh data/signer.sh
for index in $(seq 1 $nodes)
do
    docker build --build-arg INDEX=$index --build-arg
    GETH_VERSION=$GETH_VERSION -t node-$index-image -f SignerDockerfile data
    docker create --network p10network --name node-$index node-$index-image
```

```
58      docker start node-$index
59      echo -----Started node-$index
60  done
61
62  echo -----Finished
```

Appendix D

Hyperledger Sawtooth Resources

This appendix chapter will showcase the Hyperledger Sawtooth PBFT docker-compose YAML file.

Listing D.1: This YAML file showcases the docker-compose yaml file for starting a two node Hyperledger Sawtooth PBFT network.

```
version: '3'

volumes:
  pbft-shared:

services:

# -------------=== rest api ===-------------
  rest-api-0:
    image: hyperledger/sawtooth-rest-api:chime
    container_name: sawtooth-rest-api-default-0
    expose:
      - 8008
    ports:
      - '8008:8008'
    depends_on:
      - validator-0
    entrypoint: |
      sawtooth-rest-api -vvv
        --connect tcp://validator-0:4004
        --bind rest-api-0:8008
    stop_signal: SIGKILL

  rest-api-1:
    image: hyperledger/sawtooth-rest-api:chime
```

```
26      container_name: sawtooth-rest-api-default-1
27      expose:
28        - 8009
29      ports:
30        - '8009:8009'
31      depends_on:
32        - validator-1
33      entrypoint: |
34        sawtooth-rest-api -vvv
35            --connect tcp://validator-1:4005
36            --bind rest-api-1:8009
37      stop_signal: SIGKILL
38
39  # --------------=== validators ===--------------
40
41    validator-0:
42      image: hyperledger/sawtooth-validator:chime
43      container_name: sawtooth-validator-default-0
44      expose:
45        - 4004
46        - 5050
47        - 8800
48      volumes:
49        - pbft-shared:/pbft-shared
50      command: |
51        bash -c "
52          if [ -e /pbft-shared/validators/validator-0.priv ]; then
53            cp /pbft-shared/validators/validator-0.pub
      /etc/sawtooth/keys/validator.pub
54            cp /pbft-shared/validators/validator-0.priv
      /etc/sawtooth/keys/validator.priv
55          fi &&
56          if [ ! -e /etc/sawtooth/keys/validator.priv ]; then
57            sawadm keygen
58            mkdir -p /pbft-shared/validators || true
59            cp /etc/sawtooth/keys/validator.pub
      /pbft-shared/validators/validator-0.pub
60            cp /etc/sawtooth/keys/validator.priv
      /pbft-shared/validators/validator-0.priv
61          fi &&
62          if [ ! -e config-genesis.batch ]; then
63            sawset genesis -k /etc/sawtooth/keys/validator.priv -o
      config-genesis.batch
```

```
64          fi &&
65          while [[ ! -f /pbft-shared/validators/validator-1.pub ]];
66          do sleep 1; done
67          echo sawtooth.consensus.pbft.members=\\['\"'$$(cat
    /pbft-shared/validators/validator-0.pub)'\"',\"'$$(cat
    /pbft-shared/validators/validator-1.pub)'\"'\\] &&
68          if [ ! -e config.batch ]; then
69            sawset proposal create \
70              -k /etc/sawtooth/keys/validator.priv \
71              sawtooth.consensus.algorithm.name=pbft \
72              sawtooth.consensus.algorithm.version=chime \
73              sawtooth.consensus.pbft.members=\\['\"'$$(cat
    /pbft-shared/validators/validator-0.pub)'\"',\"'$$(cat
    /pbft-shared/validators/validator-1.pub)'\"'\\] \
74              sawtooth.publisher.max_batches_per_block=1200 \
75              -o config.batch
76          fi &&
77          if [ ! -e /var/lib/sawtooth/genesis.batch ]; then
78            sawadm genesis config-genesis.batch config.batch
79          fi &&
80          if [ ! -e /root/.sawtooth/keys/my_key.priv ]; then
81            sawtooth keygen my_key
82          fi &&
83          sawtooth-validator -vvv \
84            --endpoint tcp://validator-0:8800 \
85            --bind component:tcp://eth0:4004 \
86            --bind consensus:tcp://eth0:5050 \
87            --bind network:tcp://eth0:8800 \
88            --scheduler parallel \
89            --peering static \
90            --maximum-peer-connectivity 10000
91          "
92    validator-1:
93      image: hyperledger/sawtooth-validator:chime
94      container_name: sawtooth-validator-default-1
95      expose:
96        - 4005
97        - 5051
98        - 8801
99      volumes:
100       - pbft-shared:/pbft-shared
101     command: |
102       bash -c "
```

```
        if [ -e /pbft-shared/validators/validator-1.priv ]; then
          cp /pbft-shared/validators/validator-1.pub
    /etc/sawtooth/keys/validator.pub
          cp /pbft-shared/validators/validator-1.priv
    /etc/sawtooth/keys/validator.priv
        fi &&
        if [ ! -e /etc/sawtooth/keys/validator.priv ]; then
          sawadm keygen
          sawtooth keygen my_key
          mkdir -p /pbft-shared/validators || true
          cp /etc/sawtooth/keys/validator.pub
    /pbft-shared/validators/validator-1.pub
          cp /etc/sawtooth/keys/validator.priv
    /pbft-shared/validators/validator-1.priv
        fi &&
        sawtooth-validator -vvv \
          --endpoint tcp://validator-1:8801 \
          --bind component:tcp://eth0:4005 \
          --bind consensus:tcp://eth0:5051 \
          --bind network:tcp://eth0:8801 \
          --scheduler parallel \
          --peering static \
          --maximum-peer-connectivity 10000 \
          --peers tcp://validator-0:8800
      "

# --------------=== pbft engines ===--------------

  pbft-0:
    image: hyperledger/sawtooth-pbft-engine:chime
    container_name: sawtooth-pbft-engine-default-0
    depends_on:
      - validator-0
    entrypoint: |
        pbft-engine -vvv --connect tcp://validator-0:5050
    stop_signal: SIGKILL

  pbft-1:
    image: hyperledger/sawtooth-pbft-engine:chime
    container_name: sawtooth-pbft-engine-default-1
    depends_on:
      - validator-1
    entrypoint: |
```

```
142       pbft-engine -vvv --connect tcp://validator-1:5051
143     stop_signal: SIGKILL
144
145 # -------------=== settings tp ===-------------
146
147   settings-tp-0:
148     image: hyperledger/sawtooth-settings-tp:chime
149     depends_on:
150       - validator-0
151     command: settings-tp -vv --connect tcp://validator-0:4004
152
153   settings-tp-1:
154     image: hyperledger/sawtooth-settings-tp:chime
155     depends_on:
156       - validator-1
157     command: settings-tp -vv --connect tcp://validator-1:4005
158
159 # -------------=== xo tp ===-------------
160
161   xo-tp-0:
162     image: hyperledger/sawtooth-xo-tp-python:chime
163     container_name: sawtooth-xo-tp-python-default-0
164     expose:
165       - 4010
166     depends_on:
167       - validator-0
168     command: xo-tp-python -vv -C tcp://validator-0:4004
169     stop_signal: SIGKILL
170
171   xo-tp-1:
172     image: hyperledger/sawtooth-xo-tp-python:chime
173     container_name: sawtooth-xo-tp-python-default-1
174     expose:
175       - 4010
176     depends_on:
177       - validator-1
178     command: xo-tp-python -vv -C tcp://validator-1:4005
179     stop_signal: SIGKILL
180
181 # -------------=== intkey tp ===-------------
182
183   intkey-tp-0:
184     image: hyperledger/sawtooth-intkey-tp-python:chime
```

```
185     container_name: sawtooth-intkey-tp-python-default-0
186     expose:
187       - 4004
188     command: intkey-tp-python -C tcp://validator-0:4004
189     stop_signal: SIGKILL
190
191   intkey-tp-1:
192     image: hyperledger/sawtooth-intkey-tp-python:chime
193     container_name: sawtooth-intkey-tp-python-default-1
194     expose:
195       - 4004
196     command: intkey-tp-python -C tcp://validator-1:4005
197     stop_signal: SIGKILL
198
199 # --------------=== shell ===--------------
200
201   shell:
202     image: hyperledger/sawtooth-shell:chime
203     container_name: sawtooth-shell-default
204     entrypoint: "bash -c \"\
205         sawtooth  keygen && \
206         tail -f /dev/null \
207         \""
```