TÉCNICO LISBOA

Separating Agreement from Execution to Improve the Scalability of Blockchains

Diogo Peres Monteiro de Faria Fernandes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Rodrigo Seromenho Miragaia Rodrigues
Dr. Allen Clement

Examination Committee

September 2021

Acknowledgments

I would like to thank my supervisors for sharing their extensive knowledge with me and guiding me through the complex world of Blockchains.

I would also like to thank my parents for their everlasting encouragement and support.

And finally, I would like to thank my friends for their constant presence.

Abstract

To implement a secure, decentralized ledger of cryptographically-chained blocks, Blockchain systems must implement several steps: select a group of transactions and include them in a block in a certain order; execute the ordered group of transactions; and agree with the other Blockchain participants on the results of the execution of the transactions through a consensus algorithm. Today, almost all Blockchain systems follow a monolithic architecture, where all these components are intertwined and executed on the same physical machine, sharing the same computational resources. As a consequence, these systems suffer from severe scalability issues, even in permissioned settings. In this thesis, we propose SepChain, a system that aims to improve on these scalability issues by proposing a simple and precise separation of the system into agreement and execution components. To demonstrate the benefits of this approach, we present a novel proof of concept of this principle, which consists of taking an existing system implementation and split it into one Blockchain that only orders transactions, and into multiple other isolated Blockchains that then execute them.

Keywords

Blockchain; Agreement; Execution; Monolithic Architecture.

Resumo

Para implementar um ledger de blocos criptograficamente ligado de forma distribuída e segura, os sistemas de Blockchain têm de seguir múltiplos passos: selecionar um grupo de transações e incluí-las num bloco numa certa ordem; executar o grupo ordenado de transações; concordar no resultado dessa execução com outros participantes na rede Blockchain através de um algoritmo de consenso. Atualmente, quase todos os sistemas de Blockchain têm uma arquitetura monolítica, onde todos estes componentes estão interligados e são executados na mesma máquina física, e partilham todos os recursos da mesma. Como resultado, estes sistemas sofrem com problemas de escalabilidade severos, mesmo em Blockchains de acesso permitido. Esta tese propõe o sistema SepChain, que tem como objetivo melhorar estes problemas de escalabilidade ao propor uma separação simples e precisa do sistema em componentes de concordância e execução. Para demonstrar os benefícios desta separação, a tese apresenta uma prova de conceito deste princípio, que consiste em transformar uma implementação de um sistema preexistente e separá-la em vários componentes: uma Blockchain que apenas ordena transações, e outras Blockchains isoladas que as executam.

Palavras Chave

Blockchain; Concordância; Execução; Arquitectura Monolítica.

Contents

1 Introduction **1**

1.1 Organization of the Document 5

2 The Blockchain Landscape **7**

2.1 State Machine Replication 9

2.2 Byzantine Fault Tolerance 10

2.2.1 PBFT 10

2.2.2 IBFT 12

2.3 Shared Logs 12

2.3.1 CORFU 13

2.3.2 FuzzyLog 14

2.3.3 Delos 15

2.4 Cryptocurrencies 16

2.4.1 Bitcoin 16

2.4.2 Ethereum 18

2.4.3 Libra 19

2.5 Separate Agreement from Execution 21

2.5.1 Separating agreement from execution for byzantine fault-tolerant services 21

2.5.2 Hyperledger Fabric 22

2.5.3 Eve 23

3 Architecture **25**

3.1 Back to Basics 27

3.2 Overview 27

3.3 Design 28

3.3.1 Clients 29

3.3.1.A Install 29

3.3.1.B sendCommand 30

3.3.1.C getStatus 30

3.3.1.D getResult 31
3.3.2 Ordering 32
3.3.3 Execution 32
3.3.3.A append 32
3.3.3.B lastIndex & readIndex 32
3.4 Order of Events 33
3.5 Design Characteristics 33
3.5.1 Independent Components 33
3.5.2 Scaling Out 34
3.5.3 Parallel Execution 34
3.5.4 Execution after Agreement 34
3.5.5 Communication between layers 35
3.5.6 Post-execution simplified consensus 35

4 Implementation **37**
4.1 Overview 39
4.2 Configuration Module 41
4.3 Client 41
4.3.1 Install 42
4.3.2 Send 42
4.3.3 getStatus 42
4.3.4 getResult 43
4.4 Ordering 43
4.5 Execution 44
4.5.1 Front-end 1 45
4.5.1.A install & sendCommand 45
4.5.1.B append 45
4.5.2 Back-end 1 45
4.5.2.A lastIndex 45
4.5.2.B readIndex 46
4.5.3 Back-end 2 46
4.5.4 Front-end 2 46
4.5.4.A getStatus & getResult 46
4.6 Benefits and Downsides 47

5 Evaluation **49**
5.1 Overview . . . 51
5.2 Hardware . . . 51
5.3 Experimental Methodology . . . 51
5.4 Top-level comparison . . . 52
5.5 Impact of ordering nodes . . . 55
5.6 Impact of scaling out . . . 57
5.7 Impact of added security . . . 58
5.8 Impact of execution costs . . . 60
5.9 Impact of execution nodes . . . 61
5.10 Impact of Sharding . . . 63
5.11 Summary . . . 66

6 Conclusion **71**
6.1 Conclusion . . . 73
6.2 Current Limitations and Future Work . . . 73
6.2.1 Current Implementation . . . 73
6.2.2 Production-grade code . . . 74
6.2.3 Theoretical Concepts . . . 75
6.3 Closing Remarks . . . 75

Bibliography **77**

List of Figures

3.1 Simplified architecture in monolithic Blockchain systems. Each node holds the order, agreement and execution components. 29

3.2 Simplified SepChain architecture. Each Blockchain node holds either the order and agreement components or the execution component. The former components provide a log interface for the execution clusters to interact with. 30

3.3 SepChain 3-layer architecture design: client, execution and ordering layers. The first interface implements 4 methods that the client calls, and the second interface implements 3 methods that the execution node calls. The methods on the right send information to the next layer, while the ones on the left retrieve information from that layer; hence, the information flow. 31

4.1 SepChain implementation components overview. Shared Log refers to the implementation on the ordering layer 40

4.2 Representation of the ordering blockchain holding the replicated smart contracts as the shared log, which implements three methods. 43

4.3 Implementation components of an execution node. 44

5.1 Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 9 execution clusters. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively 53

5.2 Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 9 execution clusters. A log scale is applied to the y-axis. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively 54

5.3 Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 35 execution clusters. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively 55

5.4 Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 35 execution clusters. A log scale is applied to the y-axis. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively 56

5.5 Experiment 2: SepChain's ordering blockchain and Besu throughput values as the number of (ordering) nodes varies. The blue and orange lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The grey and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. Note that the yellow line overlaps the grey line. 57

5.6 Experiment 2: SepChain's ordering blockchain's and Besu's throughput values as the number of (ordering) nodes varies. A log scale is applied to the y-axis. The blue and orange lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The grey and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. Note that the yellow line and the grey line are overlapping. 58

5.7 Experiment 3: SepChain throughput values as the number of execution clusters varies. The orange and grey lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. The blue represents the system's contract execution upper bound as seen in Section 5.5. 59

5.8 Experiment 3: SepChain throughput values as the number of execution clusters varies. A log scale is applied on the y-axis. The orange and grey lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. The blue represents the system's contract execution upper bound as seen in Section 5.5. 60

5.9 Experiment 4: Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 50000 iterations. The lighter blue line represents Besu's performance. Every other line (from orange to dark blue) represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32]. 61

5.10 Experiment 4: Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 100 iterations. The lighter blue line represents Besu's performance. Every other line represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32]. 62
5.11 Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 50000 iterations. A log scale is applied on the y-axis. The lighter blue line represents Besu's performance. Every other line represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32] 63
5.12 Experiment 4: Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 100 iterations. A log scale is applied on the y-axis. The lighter blue line represents Besu's performance. Every other line represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32]. 64
5.13 Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 9 execution clusters. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems. 65
5.14 Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 9 execution clusters. A log scale is applied on the x-axis and on the y-axis. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems. 66
5.15 Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 35 execution clusters. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems. 67
5.16 Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 9 execution clusters. A log scale is applied on the x-axis and on the y-axis. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems. 68

5.17 Experiment 6: SepChain and Besu throughput values as the number of execution nodes varies, where the former system has 9 execution clusters. The blue line represents Besu's performance, while the orange represents SepChain's. 68

5.18 Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 9 execution clusters. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads. 69

5.19 Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 35 execution clusters. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads. . 69

5.20 Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 9 execution clusters. A log scale is applied on the x-axis and the y-axis. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads. 70

5.21 Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 35 execution clusters. A log scale is applied on the x-axis and the y-axis. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads. 70

List of Tables

5.1 Experiment 1: Besu's throughput value and SepChain's theoretical estimate of the throughput value as the number of clients varies, where the latter system has 75 execution clusters. The 50000 and 100 refer in the column names refer to the number of iterations used in each system 54

5.2 Percentage of heavy requests depending on the percentage of heavy clusters with two system settings of 9 and 35 clusters. The theoretical values show the percentage values SepChain expected to achieve. 66

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
BFT	Byzantine Fault Tolerance
DAG	Directed Acyclic Graph
DoS	Denial of Service
EVM	Ethereum Virtual Machine
IBFT	Istanbul Byzantine Fault Tolerant
IoT	Internet of Things
LAN	Local Area Network
P2P	Peer-to-Peer
PBFT	Practical Byzantine Fault Tolerance
PoA	Proof of Authority
PoW	Proof of Work
RBFT	Redundant Byzantine Fault Tolerance
RPC	Remote Procedure Call
SMR	State Machine Replication
VM	Virtual Machine

1

Introduction

Contents

1.1 Organization of the Document 5

Blockchain gained visibility in 2008 when Satoshi Nakamoto presented Bitcoin [1] to the world as a new peer-to-peer digital payment system that did not rely on a central authority and was cryptographically secure. Despite being considered the origin of Blockchain by many, the design principles of Bitcoin trace back to work developed since, at least, the early 1990s. In particular, in 1991, Haber and Stornetta [2] presented the idea of chaining blocks to timestamp a digital document. Then, in 1998, Dai proposed b-money [3], which already denoted the concept of creating a virtual coin. Subsequently, in 2002, Back published Hashcash [4], which proposed to counter an early form of Denial of Service attacks through a novel Proof of Work (PoW) service. After the introduction of Bitcoin, many other cryptocurrencies emerged, with Ethereum [5] gathering particular attention when it was released in 2014, partly due to its implementation of smart contracts – a concept that also dates back to the 1990s [6]. These systems paved the way for secure, decentralized applications following the same set of design principles, covering areas other than cryptocurrencies, such as Healthcare [7–9], Artificial Intelligence (AI) [10–12], and Internet of Things (IoT) [13–15].

All these previous systems and ideas contributed to the well-established definition that we have today: a Blockchain is a distributed and decentralized ledger that holds a number of client transactions in blocks, which are cryptographically linked, forming a chain. This ledger is shared between nodes in a distributed system, where the nodes collectively implement the task of adding new blocks to the chain. The nodes that form this distributed system all have the same, symmetric behavior and set of responsibilities, thus forming a peer-to-peer overlay network. Clients interact with the Blockchain by creating and sending transactions to the nodes of the overlay network, where a transaction is an action that alters the state of the ledger.

For a transaction to be added to the Blockchain, the nodes that implement it follow a distributed protocol with a series of steps, whose order and logic depend on the specifics of each system. For instance, in Ethereum-based systems, the following transaction processing steps occur. After clients send their transactions, these are broadcast to the Blockchain network. Then, each node that receives a transaction includes it in a pool with other pending transactions, until it is able to assemble a large enough set in a block. Afterwards, all the block transactions are executed in the order previously defined for the purpose of determining the new ledger state maintained by the Blockchain, while also validating other parameters (e.g., gas) to ensure the correct outcome of the execution. Finally, once all the transactions in the block have been executed, the consensus protocol (e.g., PoW) will define which block will in fact be added to the chain.

A characteristic that is shared by the many current Blockchain systems [1, 5, 16, 17] is that they are based on a monolithic design. As such, a Blockchain node is a single machine that conflates the logic for the previously mentioned steps, namely: receiving new transactions and grouping them in blocks, executing every transaction (while validating if there is enough gas for them to run), and finally agreeing on the outcome of the transactions that were executed. This monolithic design raises several problems, the most important of which is that it becomes a scalability bottleneck. This is because a monolithic design cannot separate the number of resources that are allocated to each of the previously mentioned steps. Moreover, transactions with different degrees of complexity share the same physical machine, which may lead to more complex transactions delaying the less complex ones. Furthermore, a monolithic design tends to increase the number of dependencies between components, without encouraging the system designers to carefully think through the constraints and redundancies that such dependencies create, and the consequent effect on the overall system behavior and performance. Finally, exactly because of how intertwined all the components are, monolithic systems cannot be scaled out – since this would only replicate the existing problems.

In addition to the faults of a monolithic design, the order by which the previously mentioned steps occur also impacts the system negatively. Taking into account that the execution phase comes before the consensus protocol, the latter is dependent on the former to finish. Moreover, since there are multiple nodes trying to get their block to be the one added to the chain, this means that every node other than the winning one wastes time and resources in a block that will not be added to the chain.

In our view, these issues are key to why Blockchains are still not up to par with the scalability and performance of centralized systems, even in the cases where these Blockchains operate in permissioned settings and are able to sidestep the expensive proof of work mechanisms. For instance, Ethereum can only achieve a throughput of 19 transactions per second [18], whereas Visa processes around 1700 transactions/second [19]. Furthermore, while some more recent Ethereum clients, such as Hyperledger Fabric [20] and FastFabric [21], have been able to reach 3000 and 20000 transactions/second respectively, these systems made design and implementation decisions with problems of their own, as discussed on the following sections.

In this thesis, we highlight that the designers of Blockchain systems should take into account the research literature from two decades ago to guide the evolution of future Blockchain designs. In particular, we should build on the lessons in the research area of state machine replication [22], an abstraction with three decades that achieves exactly the same goals that Blockchains aim to implement. In this area, one of the important design lessons from how the research systems evolved is precisely the importance of carefully layering state machine replication systems, namely by separating two important aspects: agreeing on an order among the commands, on the one hand, and executing the logic behind these commands and validating the respective output, on the other [23].

Guided by these decades-old principles, we aim to demonstrate that it is possible to improve Blockchain scalability by creating a decoupled architecture on top of Hyperledger Besu [16], an Ethereum client with a monolithic architecture, to introduce a separation between the agreement and execution phases. In simple terms, our architecture has two types of clusters: ordering and execution clusters. There is only one of the former, which is composed of ordering nodes that order the transactions by running a consensus protocol. And then there are multiple, mostly independent clusters of the latter type, each with several execution nodes. As a result, execution clusters can process transactions with a high degree of parallelism since one cluster's execution no longer interferes with the execution from other clusters. Additionally, we show how this decoupling and careful layering enables us to break several dependencies that may hamper scalability. For example, previous designs would unnecessarily execute transactions assuming an order that may not be final, leading to wasted computations.

To demonstrate these benefits, we present SepChain, a system that shows benefits with minimal changes to the implementation: by building on top of the Besu code base rather than tearing it apart. To this end, our design has two layers of nodes, both of which are instances of a Besu deployment, but configured with different smart contracts: one that is responsible for the agreement on an order for the transactions; and another that carries out the execution of those transactions and from which the client can get the results of the execution. Doing so raises several technical challenges, which we address in our system design.

1.1 Organization of the Document

The remaining of the thesis is organized as follows. Chapter 2 gives the appropriate context to past and current work in the subject, which serve as a basis to SepChain. Our proposal is then described in two parts: in Chapter 3, we detail the architecture, its features, and its constraints; in Chapter 4, we delineate how the architecture was implemented. Then, Chapter 5 presents the experimental methodology and the results we obtained. And finally, Chapter 6 draws some conclusions on the thesis and suggests future improvements to the current system.

2

The Blockchain Landscape

Contents

2.1 State Machine Replication 9
2.2 Byzantine Fault Tolerance 10
2.3 Shared Logs 12
2.4 Cryptocurrencies 16
2.5 Separate Agreement from Execution 21

In this chapter, we analyze the relevant literature. Each subsection highlights a body of work that provides the necessary background for understanding our solution and presents current systems that our approach aims to improve.

2.1 State Machine Replication

In this subsection, we examine the inner workings of State Machine Replication and its relevance for recent advances in distributed systems.

Using a central server is the simplest way to implement a client-server communication service. Yet the service can only be as fault-tolerant as the processor executing the server. This means that multiple servers (replicas) must be used for the service to continue to work.

An approach for implementing a fault-tolerant service is State Machine Replication (SMR) [22]. It replicates servers and coordinates the client's interactions among these replicas, with each replica simulating a state machine.

A state machine is composed of a group of variables and commands. While variables encode the service state at any instant, commands transform the machine's state using deterministic programs that run atomically (regarding other commands). A client makes requests to execute these commands (e.g. read, write), providing input if necessary, and receiving output either directly or to a peripheral output device.

Be that as it may, the system still needs to account for failures. The ones considered in the state machine model are either Byzantine – where the component exhibits arbitrary, inconsistent behavior – or fail-stop – where the other components can detect that a failure has occurred and then stop. A variant of the fail-stop is a crash failure, in which the components halt without the ability to detect a failure. In this context, a system component is said to be faulty when its behavior ceases to be consistent with its specification. Moreover, the system itself is said to be t fault-tolerant if it satisfies its specification under the requirement that no more than t components become faulty during a certain interval.

To implement a t fault-tolerant state machine system, the state machine must be replicated with each replica running individually. Additionally, it is required that the state machine is deterministic: starting in the same initial state and executing the same requests in the same order, each state machine must produce the same output. Since each replica will obtain an individual output – and each replica can be faulty – a voter device is necessary to combine the outputs of the replicas so they can agree on the final output for the system, decided by a majority. Considering that the system is designed under a synchronous model, $t+1$ replicas are enough for fail-stop failures (in case t fail, one still works) and $2t+1$ replicas are necessary in the case of Byzantine failures so that the correct replicas can outvote the Byzantine faulty ones. In an asynchronous model, both replication factors increase by t, for a total of $2t + 1$ and $3t + 1$, respectively. Therefore, a mandatory requirement for this implementation is to have replica coordination, which guarantees that every non-faulty replica receives the client requests in the same relative order.

In a nutshell, the state machine approach consists of simple steps to deterministically process a client request. First, the state machine is replicated and placed on several machines. Then, the client request is received and interpreted as an input. All the inputs received are subsequently ordered and immediately executed in that order. Granted that a state machine is deterministic, each non-faulty machine will produce the same output given the same input. Therefore, one or more voters will read the individual outputs and the agreed value – determined by the majority of machines – is the one returned to the client.

SMR is the foundation for fault-tolerant distributed systems and it underpins every other work analyzed in this chapter. In fact, SMR defines the transaction model adopted by current Blockchain systems [1, 5, 17]: each node first orders transactions, then executes them, and finally agrees on what was executed. Moreover, the Blockchain system itself can be seen as a state machine, where the replicated ledger is the state and the transactions are the commands that transform the state, deterministically. Additionally, this work has also paved the way for other relevant works in distributed systems such as Chubby [24] and Zookeeper [25], as well as distributed consensus algorithms like PBFT [26] and EPaxos [27].

2.2 Byzantine Fault Tolerance

In this subsection, we describe a key building block, namely the distributed consensus protocols in Byzantine [28] settings and highlight their importance to Blockchain systems. While there is a vast literature on Byzantine Fault Tolerance (BFT) consensus, we focus our discussion on PBFT and IBFT, which stand out for the following reasons. The former is the foundation for most of the research in this area, while the latter is implemented by many Blockchain systems today.

2.2.1 PBFT

Most previous byzantine fault tolerance algorithms [29, 30] assume synchrony for their systems to have safety, a property later described. However, malicious users might delay communication between non-faulty nodes and lead the system to rule them out as faulty: a Denial of Service (DoS) attack [31]. Thus, the Practical Byzantine Fault Tolerance (PBFT) paper [26] describes a state machine replication consensus algorithm that functions correctly despite byzantine faults in an asynchronous context.

The PBFT algorithm aims to provide two main properties in a byzantine context: safety and liveness [32]. The former property means that all non-faulty replicas agree on a group of requests, while the latter means that clients will eventually receive a reply to their request (e.g., read/write to a file system). However, these properties assume that $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty, where n is the total number of replicas available.

Note that while this algorithm does not rely on synchrony for safety, it must rely on it for liveness to ensure termination and make asynchronous consensus possible. This means that if there is a delay on the network when receiving a request, it must end in a finite time t. Furthermore, given that this is an asynchronous context, PBFT is optimally resilient since it uses the minimum number of replicas: $3f + 1$, with f being the faulty replicas. This number of replicas is necessary because f replicas might not respond due to an infinite delay and another f might be byzantine and, for instance, give a wrong response on purpose. Thus, the system must have $n - 2f > f$ replicas. Hence, $n > 3f$ replicas.

The protocol is implemented using the state machine replication [22] approach: each node has a state machine replica to reach consensus on the client requests, which are executed deterministically. Additionally, the replicas are split into one primary and the rest as backups. All the replicas move through numbered views, which are successive configurations of the system. The algorithm steps can be roughly described as follows: 1) the client sends a request to the primary replica; 2) the primary replica broadcasts the request to backups; 3) all the replicas execute the request and send it back to the client; 4) the client waits for $f + 1$ replies from different replicas, where f is the number of faulty replicas. The algorithm is further expanded in the following paragraph.

Each replica has a message log to append the messages it accepted and a number that denotes the replica's current view. The algorithm is as follows. First, a client sends a request to the primary, which is then multicast to all the backups using a three-phase protocol: $\underline{\text{pre-prepare}}$, $\underline{\text{prepare}}$, and $\underline{\text{commit}}$. In the $\underline{\text{pre-prepare}}$ phase, the primary assigns a sequence number to the request and multicasts a $\underline{\text{pre-prepare}}$ message to the backups. If the backup accepts the message, it enters the $\underline{\text{prepare}}$ phase by broadcasting a $\underline{\text{prepare}}$ message to all other replicas, including the primary; otherwise, nothing happens. Provided that signature, view number, and sequence number are correct, each replica accepts and adds the $\underline{\text{prepare}}$ message to their log. The replica is considered to be prepared if it has in its log the following: the message, a $\underline{\text{pre-prepare}}$ message with the correct sequence and view numbers, and $2f$ $\underline{\text{prepare}}$ messages from different backups. These two phases, together, ensure that non-faulty replicas agree on total order for all the requests within a view. Finally, replicas multicast the $\underline{\text{commit}}$ message once they are prepared. All prepared replicas that receive $2f + 1$ $\underline{\text{commit}}$ messages from different replicas, matching the $\underline{\text{pre-prepare}}$ message on the log (same view and sequence number), will execute the operation specified in the client's message and reply. Finally, the client waits for $f + 1$ replies with valid signatures; if received, the result can be accepted, despite the system possibly having f faulty replicas.

2.2.2 IBFT

The Istanbul Byzantine Fault Tolerant (IBFT) algorithm [33] stems from the original PBFT algorithm, yet it is adapted to work in a Blockchain network. It still uses the base 3-phase consensus algorithm just described, maintaining the $3f+1$ fault-tolerance property, yet with a few tweaks to work in a Blockchain setting. Additionally, instead of differentiating ”clients” and ”nodes”, as PBFT does, IBFT considers clients part of the network, and thus ”nodes”. In particular, IBFT splits nodes into a ”proposer” and multiple ”validators” and, in similarity to PBFT’s single primary and multiple backups replicas, respectively. Additionally, instead of simple read/write requests, the result of the algorithm is the consensus over a block of transactions, as expected from a Blockchain system. Moreover, IBFT uses a Proof of Authority (PoA) [34] type of consensus, since only the validators are able to create blocks. Since its conception, it has already been improved by some other works, such as [35].

The IBFT algorithm works as follows: in each view, the validators pick one of the nodes as the proposer using a round-robin algorithm. Then, the proposer proposes a new block and broadcasts it to the network with a pre-prepare message. Once the validators receive that message, they enter the pre-prepare state and broadcast a prepare message. Afterwards, when the validators receive $2f+1$ prepare messages, they broadcast the commit message, which informs that the block is ready to be entered in the Blockchain. Finally, the moment the validators receive $2f+1$ commit messages, they enter the commited state and insert the block on the Blockchain. Note that each of the validators inserts the $2f+1$ commit messages on the block header before inserting it on the chain; this works as a group signature, confirming that at least other $2f+1$ validators have validated this block.

All in all, PBFT defines how consensus can be implemented in a byzantine setting, a concept that is intrinsically tied to Blockchain systems. In fact, plenty of consensus algorithms used in current Blockchain systems are based on or improve on PBFT. These include LibraBFT [36] , HotStuff [37], Redundant Byzantine Fault Tolerance (RBFT) [38] and, of course, IBFT [35]. The latter is particularly relevant as it is a consensus algorithm implemented in multiple blockchains, in particular in Hyperledger Besu, our system’s foundational Blockchain.

2.3 Shared Logs

In this subsection, we survey the proposals for a shared log abstraction, and how this concept underpins our solution. We highlight the following three works. CORFU, as it is the seminal work on the shared log abstraction, serving as a basis to every other work in the section, as well as SepChain. FuzzyLog since improves on CORFU and provides an interesting insight to further research on how to improve the base system. Delos, as it is a prime example of a successful application of the CORFU principles into a commercial product.

2.3.1 CORFU

In trying to search for an effective layering model, we sought inspiration in a system called CORFU [39]. It defines the abstraction of a single log shared across multiple devices, implemented over a cluster of flash storage that can be accessed by multiple clients concurrently over the network. The log works as a map, linking each map position to a flash storage unit.

CORFU's main goal is to provide a simplified and inexpensive interface for running applications to interact with the underlying flash storage. With this in mind, CORFU is implemented as a client-side library that defines an Application Programming Interface (API) with the following main methods. append adds data to the log and returns its position in the log; read returns the data in the given position; trim removes out of date/invalid data from the log, at the request of the application; fill inserts ”junk” into a certain log position to prevent it from being updated in the future. This interface provides a clear separation and a high degree of independence between the two layers. Therefore, the shared log logic is simplified to the point of being faster and of easy and concurrent access, while the client logic can be as complex as necessary.

Implementing the previous interface requires three functions. First, a deterministic mapping function that projects the positions on the log to the flash pages stored in the units. For example, if clients want to read from the log, they first consult their local map to check which flash units can be used and then apply the function to calculate the page to be used, reading from it directly. In the case of any flash unit crashing, the client is automatically remapped to a different flash unit. Second, a tail-finding mechanism: to append data, a client must first ask a sequencer node to assign them the next empty position (the tail) on the log, and only then they can write to it. Third, a chaining replication protocol: if the client wants to write to a group of units, it does so sequentially, only writing to the second one after getting confirmation on the first one, thus forming a chain.

This protocol guarantees an important property: safety under contention, which guarantees that if two clients are writing at the same time in the chain, only one can append the value, while the other gets an error. Therefore, any subsequent read on that position will always return a single value: the one successfully appended – a property called single-copy semantics. In addition to this, if a client wants to read a value but they do not know if the log position was written, then they read the last position on the chain and, if it is unwritten, it returns an error. This ensures a second property: durability, which states that data is only visible if enough units in the chain have seen it (reached $f + 1$ replicas, as to tolerate f failures). Nevertheless, if they know the log position was written, they can read it from any replica on the chain.

CORFU defines the fundamental concept for our solution's implementation of the interface between the agreement and the execution layers: a shared log abstraction, where the agreement cluster provides the log and each execution cluster can interact with the log independently and concurrently. In particular, we take inspiration from its interface, yet we change the way that it is implemented to better fit the Blockchain setting we work in.

2.3.2 FuzzyLog

As previously described, CORFU [39] provides an API that is both simple and inexpensive to use by client applications. However, this abstraction implies a total order on the log that might work in smaller-scale systems, but it does not perform well at a larger scale.

The main problem with a total order is how expensive it is to enforce: ordering all updates via one single node (the sequencer in CORFU) limits throughput greatly and slows operations down. Worse still, total order can be impossible if we consider multiple geographical regions: clients might be unable to contact the sequencer due to a network partition. Fortunately, total order is not always necessary: data that is not directly related, such as different keys in a map, do not need to be ordered. As an alternative, FuzzyLog [40] provides a new abstraction that implements a partial order.

FuzzyLog is a shared log that used a Directed Acyclic Graph (DAG) [41], which implements a partial order following two key ideas: data sharding and geo-replication. In this scenario, data sharding consists of partitioning the application state across different shards where each shard has a different color, with the purpose of having nodes (updates to the log) with different colors to be updated concurrently. Every time a new update marked with a color arrives, it is added to the chain of updates (i.e. nodes) of that color. This means that there is a total order for the same color, but a partial order for the DAG.

In addition, FuzzyLog must also consider different geographical regions. Each region has a replica of each existing color. When there is a new update, it is marked with a certain color and it is added to the chain of the corresponding color locally. However, this region must be synchronized with other regions. Hence, this local chain will be added to the other regions maintaining the causal order within that color through cross-linked connections between the two chains. For example, considering Europe and Asia as two different regions, a client application in Europe appends two updates of the same color (red), which are materialized in the local chain as nodes A<–B (with B being causally after A). After that, a client in Asia appends two new up-dates (C<–D), also red. When both clients call a synchronize function, both regions will have the two different chains, A<–B and C<–D, but these chains (of the same color) will still be causally related as the update in Asia happened after the one in Europe. Thus, an extra cross-link (B<–C) joins both chains.

The importance of FuzzyLog is that it sets an interesting research challenge, which is to improve an initial CORFU-based shared log abstraction, by sharding it to provide a partial order, and gain better scalability through the parallelization of the ordering and the execution. Although SepChain does not currently implement this precise abstraction, FuzzyLog provides the foundation for an improvement on the current log implementation.

2.3.3 Delos

A great part of current systems that use consensus to replicate states are difficult to alter due to their monolithic architectures, where the system's database and consensus logic are interwoven. Thus, to take advantage of new, better consensus protocols, the system must be redesigned in its entirety. A way to simplify consensus is by implementing a shared log API such as CORFU [39], which enables the separation of application logic from consensus logic. However, to effectively improve simplicity, fault-tolerance, and speed, Delos [42] goes a step further by proposing a virtualized shared log abstraction, which aims to speed up production systems by swapping the consensus mechanism without disrupting the whole system.

A virtualized shared log approach still splits the application and consensus logic into two different layers sharing an API, but it allows another separation of the consensus logic in two other layers: a control plane and a data plane, both exposing a conventional shared log API. The former includes the logic for the reconfiguration of the protocol, such as leader election and membership changes, while the latter defines the ordering mechanism. This separation means that the system can reuse the control plane (the VirtualLog) while changing only the data plane, which is an abstraction composed of pluggable ordering protocols called Loglets. Thus, the VirtualLog abstraction implements a logical shared log that chains together the Loglets, which can be dynamically swapped. When a client sends commands to VirtualLog, these are sent to a virtual space that is mapped to each Loglet address space.

The VirtualLog is composed of two distinct components, the first being a client-side layer that receives and routes the requested operations to the underlying Loglets based on the defined mapping. Clients interact with the VirtualLog through its API by calling commands such as append to add an entry, checkTail to obtain the first unwritten position, or readNext to read the next available position. In addition to these commands, Loglets implement on their API a seal command, which ensures that a new append fails, and a modification to checkTail to return a Boolean whether the block is sealed.

The second VirtualLog component is the Metastore: a centralized component that stores the Loglet chain, of which each client maintains a local copy. If there are no changes in the chain, consistency is easily maintained by just accessing the local copy. However, if the client wants to use a different Loglet, there must be a reconfiguration. Thus, three steps are executed: sealing the old chain by sealing the last Loglet; then install the new chain on the MetaStore, which is sent by the client that triggered the change; finally, the client must fetch the newly created chain.

In a nutshell, the Delos architecture is composed of three layers. The top layer is the API layer, where the client supplies a request to an API-specific wrapper which then relays them to the layer below. The middle layer (the core) uses virtualization: it is composed of local storage and a runtime that uses the API defined on the top layer relay requests to the VirtualLog, which in turn interact with the bottom layer – the Loglets. Thus, in addition to being able to dynamically switch between Loglets, Delos can also change the API wrapper to support multiple applications while keeping the same core.

All in all, a virtual consensus approach can improve on many aspects over traditional monolithic architectures: it maintains a simple, layered architecture while improving on the fault-tolerance/downtime trade-off. This is because only the control plane is required to be fault-tolerant since the Metastore must be highly available for writes for the reconfiguration process, and therefore requires an implementation of a fault-tolerance protocol. Therefore, due to this separation, the Loglets don't need to be fault-tolerant and can be easily and quickly swapped.

Delos provides a very good example of how to benefit an existing, commercial system by implementing a clear, layered separation between consensus and execution, using a shared log abstraction. Our goal in discussing this work is to show how these principles can and should be applied to Blockchain systems, leading to several design improvements.

2.4 Cryptocurrencies

This section explores influential and current works in the field of cryptocurrencies, by detailing how they are structured, what their advantages and shortcoming are, and how our approach improves on the latter. In particular, we focus on three noteworthy systems. Bitcoin, as the pioneering work and the driving force for every other cryptocurrency. Ethereum, one of the most popular cryptocurrencies today and the starting point to many other systems, including our system of choice – Hyperledger Besu [16]. Libra, which is a permissioned blockchain that uses a BFT-based consensus protocol, thus providing a different angle than the previous two systems.

2.4.1 Bitcoin

Most internet transactions between two parties require a third party to process and mediate them in case of disputes or reversals. This intervention by a third party (such as banks) often results in these transactions becoming more expensive. Bitcoin [1] emerged as an alternative to avoid these costs and without having to trust a third party: an electronic payment system based on cryptography rather than trust and that allows the two parties to interact directly. Additionally, it uses a new, digital, and homonymous currency.

In similarity to any banking system, the users can send transactions with a certain amount of a coin. However, in the context of Bitcoin, these transactions are created and processed with a scripting language. For a user to transfer the coin, they must first digitally sign the hash of the previous transaction and the public key of the payee. Thus, the coin is bound to a chain of digital signatures, which the payee can verify for correctness. Nevertheless, the payee can only verify the ownership of the coin, but not if any of the previous owners have double spent, that is, used the same coin more than once in different transactions. One possible solution is to have a third party – a mint - validating which coins are still valid, but this would require the system to depend on a third party, which defeats Bitcoin's decentralization purpose. Therefore, Bitcoin aims to solve this by making transactions publicly announced and by making the majority of the system agree on a single group of transactions.

The solution requires two key concepts: a timestamp server and a Proof of Work (PoW) scheme. The former takes a block of transactions and publicly announces its hash to prove that the data existed at that time. Each block includes a timestamp and the hash of the previous timestamp, creating a (block) chain. Given that the Bitcoin system is decentralized, it is necessary to provide the consensus among the participants in what block should be added to the chain. Hence, the PoW consensus algorithm, which works as follows: each block also has a nonce element, which is incremented until the cryptographic hash [43] of the block has the same number of zeroes as a predefined target value. When the first participant reaches the correct hash value, their block is the one that defines the order of the transactions (the winning block). The other peers then verify (i.e. execute) the block, and if it is valid, there is a consensus on adding it to the chain.

The steps to process transactions sent to the network are the following: 1) the transactions are broadcast to all the nodes. 2) all the nodes collect multiple in a given order. 3) each node executes the transactions collected in that order. 4) PoW is executed and determines the winning node, which broadcasts the winning block to the other nodes. 5) every node validates the winning block (e.g. check if its transactions are not already spent). 6) If the block is valid, the node propagates that change to their copy of the Blockchain.

The participant that added the block to the chain in this round will receive a transaction fee, as an incentive for continuing to put in computational effort in the chain. Moreover, the block is considered secure only after another 6 blocks (confirmations) are added to the chain, as a way to prevent double-spending attacks [44]. Notwithstanding, due to network delays, two participants may arrive at a winning block at the same time and both add their block to their local chain; this is called a “fork”. The Bitcoin protocol dictates that the longest chain (i.e. with the greatest number of blocks) will represent the decision of the majority of the nodes since it is the one with the most PoW effort. Thus, even if two chains exist temporarily, only the chain to which the majority of the participants add blocks will persist, as the minority will not have enough computational power to overcome the majority's PoW effort. With this rule, nodes can always have a consensus on which chain to endorse.

2.4.2 Ethereum

Ethereum [5, 45] is a decentralized network built by following Bitcoin's footsteps, improving on the short-coming of the Bitcoin protocol and its scripting language, but, above all, having more than just a new digital currency. Thus, Ethereum developed a Blockchain system that uses a Turing-complete language [46] called EVM bytecode, which allows any programmer to arbitrarily write decentralized, consensus-based applications, financial or not.

Like Bitcoin, in Ethereum's Blockchain, each node in the network stores a copy of the full state of the Blockchain locally (the "world state"). This state is composed of user accounts, which hold the coin (ether) balance, the data storage, and a nonce (to prevent double-spending [44]). An Ethereum account can be of two types: externally owned and a smart contract. The former are deployed by the user on the Blockchain and can send and receive Ether to/from other externally owned accounts. The latter are pieces of reusable code that resemble a programming class, possibly composed of various structures and methods that run automatically and independently – provided a certain input command, they perform an action and output a result.

The world state is continuously changed through the execution of transactions, i.e., sending a certain amount of Ether from account A to B; or sending a smart contract command as input to be executed and produce a result. These transactions are in fact data blobs that contain the message to be sent, the recipient, the sender's signature, and the amount of Ether to be sent; in case the transaction creates a contract, it also contains the contract's byte code. Calls are similar to transactions, but they only query the current state, without changing it.

Executing a transaction implies using computational resources. To account for these, each transaction has a fee, which is measured in "gas" – a unit that measures how much computational effort has been put into a certain operation. Gas exists to protect the Ethereum network from attackers that try to maliciously exhaust all the nodes' available computational resources (e.g., by programming an infinite loop method in a smart contract). For this reason, each transaction has a gas limit – the maximum amount of computational effort a node can put into executing that transaction. If the transaction execution goes over the gas limit, the node halts and outputs an "out of gas" exception. Gas is also used to incentivize nodes to execute transactions. The node that adds a new block to the chain is awarded a fee (in Ether), which is the gas limit multiplied by the gas price – a value decided by the client when creating the transaction.

The world state is stored in the state machine, also called the Ethereum Virtual Machine (EVM); this includes the EVM code of every smart contract on the network. The EVM also defines the state transition function: the group of selected transactions that are executed on the current state and output the resulting state. The actual transaction execution is made possible through an iterator structure: it executes each transaction that was selected by the node and subtracts the gas used. When the iterator reaches a halt (either by reaching the end of the transaction or due to an exception, e.g., running out of gas), it gets the next transaction until all transactions are executed.

To execute transactions, a client submits them to an Ethereum node, and then these stay pending in a transaction pool. Then, for the transaction to be added to the chain, the following steps are needed. First, each Ethereum node selects some of these transactions in a certain order and batches them into a block. Then, provided that the transactions are valid (e.g., each sender's signature is correct), each of the transactions is executed, sequentially, applying the state transition function to the current state (corresponding to a tentative order that may be overturned); if the client runs out of gas, the execution halts. Third, once all transactions are executed, the consensus phase begins. This last phase entails: 1) running the consensus algorithm (e.g., Proof-of-Work) to determine which node adds their block to the chain. 2), the winner adds the transaction fee to its account and broadcasts the winning block to the rest of the network. 3) all the network nodes re-execute and validate this new block (the correctness of its timestamp, block number, among other parameters). If the block is valid, the nodes incorporate its state changes to their local Blockchain. Note that, just like Bitcoin, each Ethereum node always considers the longest chain as the correct one.

Ethereum improves over a few limitations in Bitcoin. Firstly, Bitcoin limits the users' actions to improve security. That is, programmers are not able to write loops on their code with Bitcoin's scripting language. This prevents infinite loops, which would make a contract execute indefinitely, and thus blocking the participants in non-terminating smart contract execution. However, in Ethereum, a loop would never be infinite since it would always reach the gas limit, forcing the execution to stop, thus solving the issue while maintaining Turing-completeness. Furthermore, in terms of scalability, Bitcoin is at a disadvantage. Bitcoin has higher latency since it takes 10 minutes to confirm the block through 6 confirmation blocks, as seen previously. Ethereum uses a variation of the GHOST protocol [47], as a different, more efficient way to calculate the longest chain that results in lower latency. Moreover, Bitcoin forces the size of their blocks to be 1MB, resulting in a limited amount of transactions to be included. Ethereum has no such restriction, leading to a higher throughput.

2.4.3 Libra

Libra [1] [17] presents a protocol that supports an entirely new ecosystem that offers a new global currency: the Libra coin. Its main goal is to rely on a common infrastructure to process transactions and to maintain accounts, with low entry costs and no single regulatory entity. To implement such a system, Blockchain technology is ideal since it guarantees that no central authority controls the network; hence, the Libra Blockchain. Also, and in contrast with Bitcoin's and Ethereum's permissionless Blockchains,

[1]Note that Libra was recently renamed to Diem. However, we will keep the name "Libra" to be coherent with the naming in the literature referenced.

Libra's is permissioned since the nodes in the network are chosen according to participation criteria, such as having invested in the project.

The Libra Blockchain is a decentralized network that stores a ledger of Libra coins, which are owned by accounts authenticated using public-key cryptography and identified by sequence numbers. In the ledger's state, the amount of coins each account has is stored in the form of a map (account address to account values). Furthermore, and just like before, clients interact with the Blockchain by submitting transactions. These are processed by validators (i.e. the blockchain nodes) to be included in the ledger. Libra uses the same gas mechanism as Ethereum.

A transaction takes the current ledger state as input and produces a new ledger state as well as an extra output including an execution status code (such as "error" or "success"), the gas usage, and an event list. The latter includes the side effects of a transaction, such as a confirmation of payment. Moreover, each transaction contains the sender's address and public key, the script to be executed (written in Move, Libra's bytecode language), and a sequence number (to prevent replay attacks [48]). Besides, transactions also include the gas price and the maximum gas amount (a gas limit), whose product is the transaction fee paid to the validator that inserts the block in the ledger.

For these transactions to be added to the Blockchain, validators must reach a consensus, even if there are Byzantine faults. Thus, Libra implements the LibraBFT [17] consensus protocol that works in rounds, each round with three phases. In the first phase, a validator is chosen as the leader and proposes a group of ordered transactions to all other validators. Then, the validators receive and check if those transactions should be added, according to their voting rules. If so, each validator executes the transactions, thus verifying them for correctness (i.e. a valid gas limit, public keys and sequence number). Then, in the second phase, each validator sends a signed message to the leader, voting for the approval of the block. Finally, in the third phase, once the leader receives at least $2f+1$ votes, there is a quorum on the agreement on the block to be committed; this quorum is then broadcast. A block is considered to be committed if it has a quorum on the current round k, in addition to existing two blocks with a quorum each in rounds $k+1$ and $k+2$ that confirm block k.

Libra sets itself apart from Bitcoin and Ethereum for being a permissioned Blockchain and by using a BFT consensus protocol. These result in a vast performance improvement [49] since, for instance, BFT ensures that the block is committed in a few rounds, while PoW requires waiting for confirmation blocks. However, BFT protocols have a broadcast complexity of $O(n^2)$, where n is the number of nodes [49]. Hence, to show any real improvement, BFT protocols require fewer nodes to reduce complexity, which tends to happen in permissioned Blockchains, as they have fewer participants.

Despite their differences, these three cryptocurrencies have two very important architectural elements in common: consensus is dependent on the results of the execution; and they are monolithic systems: all the transaction executions share the same computational resources. Thus, these are fitting candidates to explore the ideas and research challenges underlying this dissertation, namely whether scalability can improve by separating agreement and execution.

2.5 Separate Agreement from Execution

In this section, we survey the systems that implement a separation between the agreement and execution phases, and analyze their benefits. The first system examined establishes the foundation concepts of separating agreement and execution, whereas the following two systems provide two possible implementations of the former concepts in a Blockchain setting.

2.5.1 Separating agreement from execution for byzantine fault-tolerant services

Byzantine Fault-Tolerant Services have as a goal to improve the trustworthiness of the systems they protect by using redundancy to strengthen: integrity – the service processes user requests correctly; availability – the service operates without interruption; and confidentiality – only the users authorized to see the information can see it. These services use state machine replication architecture, which requires combining order, execution and agreement in the same group of replicas.

However, this architecture has two serious obstacles. Firstly, enhancing integrity and availability implies hurting confidentiality: although there's a higher probability of a correct and available service, with each new replica there is also a higher probability of encountering a new bug. And as a result, this bug can be exploited to gain access to unauthorized data. Secondly, BFT systems usually require at least three-fold replication to function correctly in an asynchronous system, since there must be $3f + 1$ replicas for a maximum of f faults.

To tackle both problems, this paper [23] explores an architecture that separates the agreement phase from the execution phase. This separation is possible since the agreement phase produces verifiable proof of the ordering of a request. This architecture can bring several benefits to the replicated system.

First, this system reduces the replication cost of executing replicas in half. While the agreement phase still needs $3f + 1$ replicas to account for network delays and byzantine replicas, with this separation, once an order for the requests is decided in agreement, only $2f + 1$ replicas are needed for execution; f extra replicas to ensure the correct order of requests are no longer needed [50].

Second, this separation allows for a privacy firewall to be implemented between the two phases. Traditionally, SMR architectures include a voter that is responsible for combining the outputs of the replicas at the client. Yet, this voter may be faulty and leak information to the attacker. Although the separation between agreement and execution already provides some improvement on confidentiality per se, as agreement replicas can filter bad replies from reaching the client, this is not enough to fully protect confidentiality since faults in the replicas can leak information as well. The solution is to implement a privacy firewall: a filter node array of $h+1$ rows and $h+1$ nodes per row, which tolerate up to h faults.

This architecture ensures confidentiality without harming integrity and availability through three concepts. Firstly, redundant filters ensure that there is always filtering even in case of failure. This is because such a number of filter nodes guarantee that there is always a correct path between agreement and execution replicas, which makes sure that a reply is always available and reaches the client. Additionally, it also ensures that there is always a correct row, which does not allow for a faulty node to simultaneously access confidential data – or to change the integrity of that data - and communicate with clients. Secondly, agreement nodes outside the firewall assign sequence numbers so that attackers cannot manipulate non-determinism as a way to surreptitiously transmit confidential information. Finally, the nodes only connect to rows directly above and below, forcing a correct path to exist, and where this communication is encrypted.

This work underpins our solution's main concept – to separate agreement from execution — although with different motivations: while this paper is focused on improving the system's fault-tolerance and confidentiality, our focus is to improve the scalability. Additionally, the following works also attempt to separate the system into different components, following the concept of this paper to a certain extent.

2.5.2 Hyperledger Fabric

The previous descriptions in Section 2.4 highlighted an issue with most Blockchain designs, which is that its monolithic architecture leads to the sequence of protocol steps being formed organically, without carefully thinking through the consequences of the order of that sequence, and even leading to some wasted work. Hyperledger Fabric [20] was one of the first systems to think through this order, by proposing an execute-order-validate architecture, while also tackling the issues of sequential and deterministic executions.

The design of Fabric defines three possible roles for the Blockchain nodes: clients, who submit transaction proposals for execution and broadcast the accepted transactions for ordering; peers, who execute transaction proposals and validate transactions; and ordering nodes, who establish the total order for transactions.

An application that runs on the Fabric Blockchain has two parts: the chaincode, which is a smart contract that implements the application logic and is triggered by other transactions; and the endorsement policy attached to the chaincode, defining which peers will receive the transaction and stating the criteria for the transaction to be valid, e.g., X peers must have the same result in the end.

The processing of transactions works as follows. First, during execution, the client sends a proposal to the peers defined in the endorsement policy (also called endorsers). This transaction contains the ID of the client, the transaction payload (the operation, parameters, and ID of the chaincode), a nonce, and a transaction ID. This proposal is simulated (i.e., the chaincode operation is executed) in the endorsers' local Blockchains and they store the set of values that were read and written. These sets are then sent to the client, cryptographically signed, forming a message called ”endorsement”. Once the client collects enough endorsements that satisfy the policy, they create the transaction and send it to the ordering nodes.

Second, the ordering phase will define a total order on the received transactions by batching them into a block and atomically broadcasting the newly-formed block, establishing consensus using a deterministic consensus protocol such as Raft [51]. The block is then sent to a new group of peers (the committer peers) for the validation phase.

Third, in the validation phase, the peers verify if the endorsement set satisfies the endorsement policy; then, they verify each transaction sequentially, ensuring that the read set corresponds to the same write set. If in any of these steps the validation fails, the transaction is discarded. Finally, the block is added to the Blockchain.

With this architecture, Fabric manages to solve many limitations. First, it avoids sequential execution since only part of the peers, the endorsers selected according to the endorsement policy, execute the transaction proposal; this means that different executions can be parallelized. Second, it manages to handle non-determinism since it executes before ordering: if the client obtains different result sets (from different peers), then that transaction is rejected. Hence, if there are any inconsistencies, only the peers are affected (on their local state), but the client never is. Nonetheless, this design implies that there are multiple instances where a transaction that has already been executed can end up being discarded due to its invalidity. And while transactions can be rightfully discarded when they are invalid, works such as [52] show that valid transactions are also discarded simply due to the concurrent execution of requests. Moreover, works that improve Fabric's performance [21] do not address this issue directly. We, therefore, conclude that, despite being the first work to carefully disentangle the monolithic design of Blockchains, the fact that transactions are first executed and only then agreed upon and validated is still sub-optimal as it can lead to the waste of resources.

2.5.3 Eve

Multi-core servers pose a challenge to State Machine Replication [22] since the replicas should deterministically process the same sequence of requests so that the same states and outputs are achieved. This is because different servers execute parallelly and thus they can have divergent states and outputs. Hence the solution should be to force multi-threaded determinism, but this has a major issue: the current technology is too slow.

Eve's [53] approach does not require the replicas to execute requests in the same order. Instead, the requests are split into batches and executed by different replicas concurrently, speculating that the result of these parallel executions will be the same across enough replicas. To implement this, Eve suggests an architecture that first executes the requests and then verifies that an agreement was reached on the state and output, rather than agreeing on an order first and then executing.

The protocol has two stages. In the execution stage, the client sends their requests to a primary execution replica, which separates them into batches, assigns each a sequence number, and sends them to all other execution replicas. Then, these are split by a deterministic device (the mixer) into parallel batches: groups of requests that are speculated to produce different results. To split them, the mixer goes through each request's accessed objects and sees if they are being read or written; after going through all the objects, it looks for read/write or write/write conflicts and puts on the same batch the requests whose objects have no conflicts. Afterwards, each replica executes each parallel batch (concurrently), in the order specified by the mixer. After all batches are executed, a hash of the current state and output is generated and sent to the second and last stage: the verification stage.

While the previous stage aims to avoid divergence, it is not able to do so every time: the mixer might wrongly include conflicting requests in the same parallel batch. The verification state aims to ensure that all correct replicas that executed the same batch of requests will reach the same state and output the same results. First, an agreement protocol is run, to determine all the correct replicas' states and output results. Then, based on the hashes received from the previous stage, the protocol will decide whether the batch will be committed (if enough hashes match) or rolled back (if too many hashes differ). If the decision is to commit, then the execution replicas mark this batch as committed and send a response back to the client. Conversely, if the decision is to roll back, then the execution replicas roll back to the state of the last committed batch and re-execute it sequentially.

Compared to our approach, both Hyperledger Fabric and Eve provide a similar separation between consensus and execution, by identifying the different node roles: client, ordering/verification, and peer/execution. However, since their focus is on avoiding sequential, deterministic executions, they first execute the transactions without ordering them. This may lead to different execution results that require the nodes to re-execute until a valid group of executions is found, which can potentially result in severe latency. Thus, our approach starts from a similar principle but then follows a completely different path: in particular, we order the transactions first and immediately agree on that order, hence re-executions do not occur.

3 Architecture

Contents

3.1 Back to Basics 27
3.2 Overview 27
3.3 Design 28
3.4 Order of Events 33
3.5 Design Characteristics 33

In this chapter, we showcase SepChain's architecture: a simple but effective design that aims to demonstrate that it is possible to have a new layered architecture that significantly improves the design of current Blockchains.

3.1 Back to Basics

While modern Blockchains such as Ethereum [5], Bitcoin [1], Corda [54], have their roots in traditional state machine replication [22], the intricacies of each of their protocols lead them to pay no importance to their foundation. However, we believe that there are significant lessons from state machine replication that can and should be applied to these Blockchain systems. In fact, we believe that significant improvements in performance will only be achieved by re-examining the system architecture in the view of state machine replication, independently of the details of each of the protocols.

These modern Blockchains monolithic system architectures, as well as early Byzantine fault-tolerant state machine replication systems [26, 55] intertwine consensus and application execution into a single interdependent blob. While multiple BFT algorithms [26, 55–59] demonstrated that BFT consensus protocols could process tens of thousands of requests per second, these systems were in fact only considering trivial (e.g., *no-op*) requests [53, 60, 61].

Fortunately, there is a ready-made architectural solution to these systems: separate agreement from execution [23, 53, 61]. Under a separated architecture, one cluster of nodes provides an ordering service [62] with a shared log [39, 42], while one or more additional clusters independently execute the commands stored in that log [42, 61, 63]. Furthermore, this principle has already been applied to real-world, industry-ready systems: as we previously surveyed, Facebook successfully uses this architectural strategy to improve both performance and maintenance characteristics of their primary log processing system [42]. Now is the time to apply the same principles to Blockchains.

In the following sections, we will describe a novel architecture that applies these well-established principles, and demonstrate their viability in the current Blockchain panorama.

3.2 Overview

SepChain's main goal is to separate the Blockchain monolithic design (seen in Ethereum-based systems) into two different layers: ordering and execution. The reason for such a separation is a result of many problems inherent to monolithic systems, most importantly the inability of scaling out the system – since all the components are intricately connected, adding more resources only replicates or even heightens the already existing overheads. When translating this principle to the particular setting of Blockchain systems, adding more Blockchain nodes only means that more nodes are participating in the consensus, which ultimately slows down the performance, despite a possible increase on the system's security (due to running consensus with a larger value of f, the maximum number of tolerated faults).

Additionally, as we described in previous sections, in most monolithic Blockchain systems (e.g., Ethereum-based systems), each node has to follow three sequential steps to submit a block to the chain. First, order the transactions; then, execute them; and finally, agree on what has been executed through a consensus protocol. This order of events has a few inherent repercussions. First, the fact that the execution phase comes before the agreement phase implies that most of the transactions executed will be wasted since only one block is added to the chain in the end. Second, this order of events also implies that the consensus protocol depends on how fast the execution phase is: a node can only move on to the agreement phase once all the transactions in the block have been executed. Finally, the tight coupling between these steps running on the same nodes means that we cannot scale these functionalities independently. Our design aims to solve each of these issues.

To solve these two problems, our solution has two fundamental design choices. First, in comparison with most Ethereum-based systems, and many other current architectures, we change the order by which each phase happens. 1) Once the clients send their transactions, each node selects a few of these transactions and adds them to their local block in a certain order; 2) All the ordering Blockchain nodes agree on the transaction order; 3) only then the previously agreed transactions are executed. As a result, we only have to agree on what is going to be executed – rather than agree on what has been executed – and no execution cycles are wasted.

The second fundamental design choice is based on the fact that we depart entirely from a monolithic design, and instead, we separate agreement (i.e., consensus) and execution into different, isolated clusters of nodes, following the principles of state machine replication [23]. The architectural difference between a common monolithic Blockchain design and SepChain is depicted in Figures 3.1 and 3.2, respectively. Furthermore, we can partition the application logic by saying that each execution cluster is only responsible for executing commands from a subset of the total set of contracts. This way, there is a large degree of independence between the execution of different transactions: if they contain commands from different contracts, they can run in completely separate execution clusters and, consequently, transaction executions from different contracts can no longer slow each other down. Additionally, it becomes easier to scale out since more execution clusters can be added to parallelize execution performance.

3.3 Design

SepChain's design defines three layers: the clients, the ordering service cluster, and multiple execution clusters. This layered model works as a black box since the client only knows they are interacting with

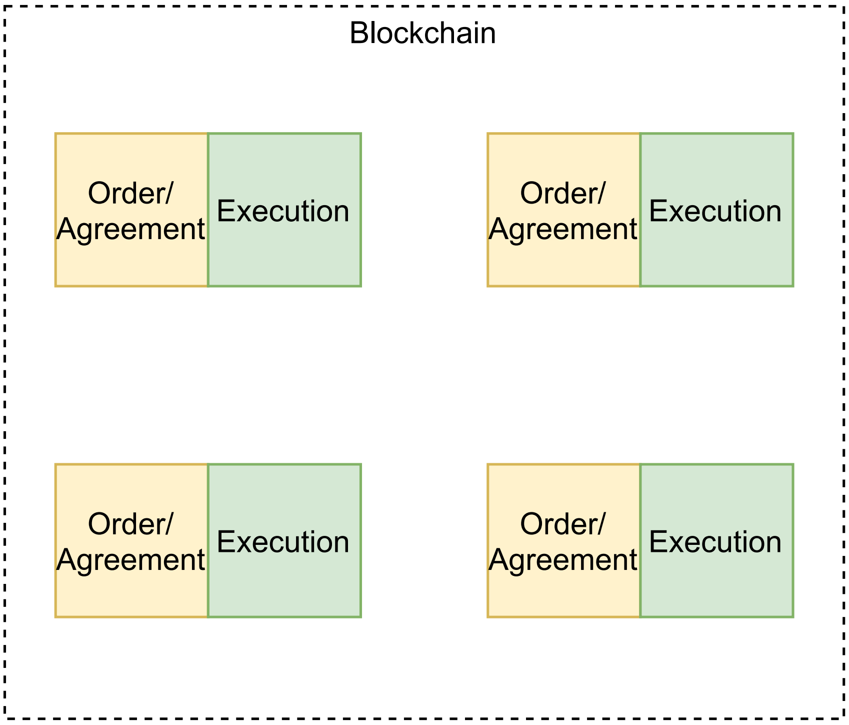

Figure 3.1: Simplified architecture in monolithic Blockchain systems. Each node holds the order, agreement and execution components.

their application; in turn, each layer can be modified without the client noticing. This structure is depicted in Figure 3.3. We now describe each of these layers in more detail.

3.3.1 Clients

The clients are simple programs that request their commands to be executed on their applications, remotely and asynchronously. To do this, they interact exclusively with one of the execution clusters (as defined in the system configuration) through an interface with four methods, which are described in the following subsections.

3.3.1.A Install

Clients can install their application in the form of a smart contract by sending that contract's data to the execution cluster through the install method. To call this method, clients only need to provide one of the execution cluster nodes (which is also predefined in the system configuration) the name of the contract they intend to install. Then, that execution cluster node itself creates and sends a transaction that creates a smart contract containing the client application on the designated execution cluster nodes.

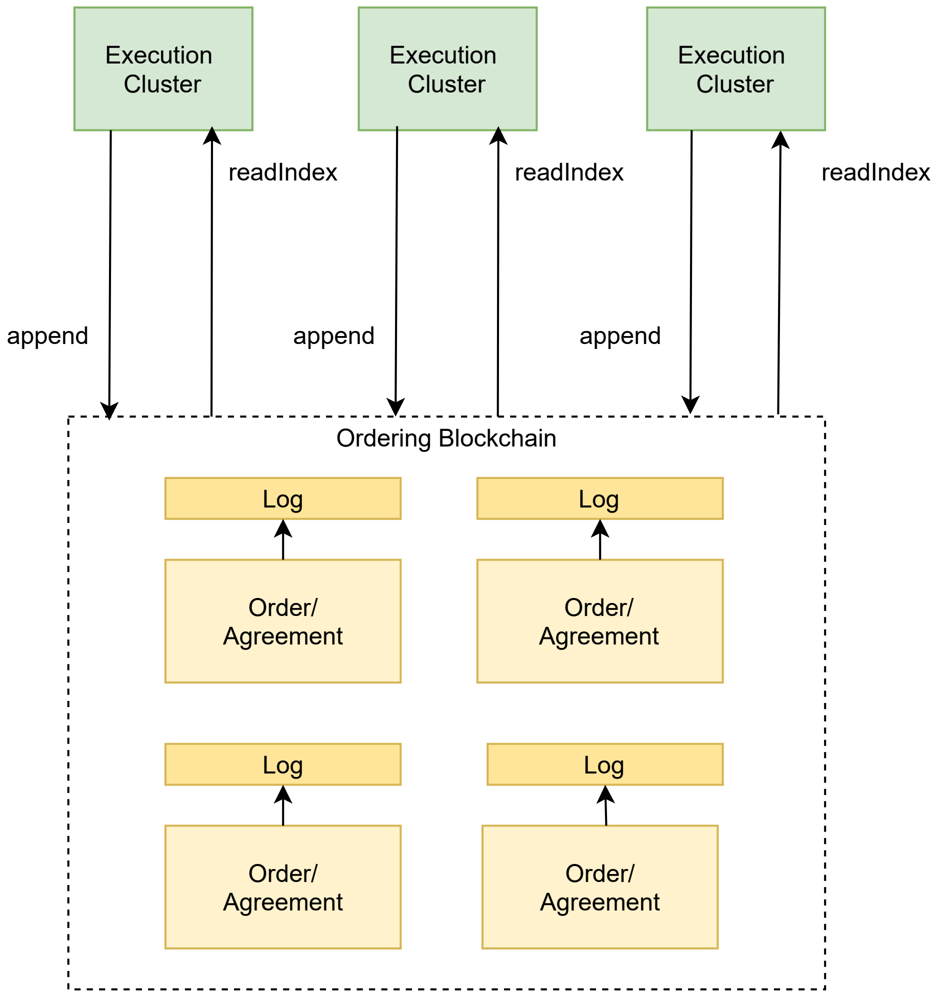

Figure 3.2: Simplified SepChain architecture. Each Blockchain node holds either the order and agreement components or the execution component. The former components provide a log interface for the execution clusters to interact with.

3.3.1.B sendCommand

Each client can send a transaction with the command to be executed on their contract with sendCommand. Similarly to the install method, clients contact the same execution cluster node providing the contract name and command name. This time, the execution cluster node creates and sends a transaction to a previously installed contract to execute the command that is provided.

3.3.1.C getStatus

Clients can query their designated execution cluster for the status of their command with (getStatus). The status returned can be one of the following three: the hash of the result of the command execution (as a String), the string ”Pending”, or the string ”Deprecated”. The former is what the client expects to

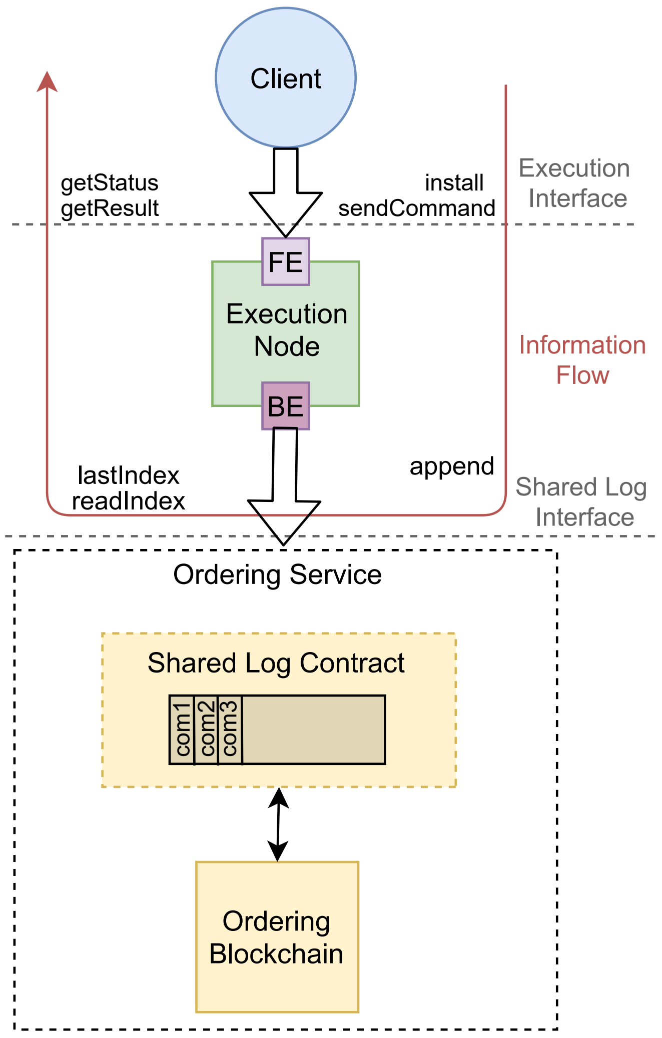

Figure 3.3: SepChain 3-layer architecture design: client, execution and ordering layers. The first interface implements 4 methods that the client calls, and the second interface implements 3 methods that the execution node calls. The methods on the right send information to the next layer, while the ones on the left retrieve information from that layer; hence, the information flow.

receive – it means that the result is ready. The latter two are Strings that represent a result not yet ready or a result that is already deprecated, respectively. The conditions in which each of these values are returned is later detailed in the implementation section. Additionally, to ensure that the string returned is trustworthy, the clients must gather the same response from a majority of the execution cluster nodes.

3.3.1.D getResult

Once the client receives the hash of the command result, they can query one of the execution nodes from the designated cluster to get the actual return value of the command they sent (getResult). The value of the contract execution is returned in a String format.

3.3.2 Ordering

The ordering service is a Blockchain with multiple ordering nodes that exclusively provide the global order of commands. When commands are received, each node batches them into a block according to a local order, and then, through a consensus protocol (e.g., PoW, IBFT), the ordering nodes reach an agreement on the order of the transactions. Once consensus is reached, the ordered commands are appended to a log structure, to be read by the execution nodes. The log is a shared log abstraction [39] in the form of a smart contract, where this contract is always installed before the clients start interacting with the system. The log stores the commands received in a map structure (mapping the log position to the command stored), and that defines three methods that the execution clusters call remotely: append, lastIndex, readIndex, which are defined next.

3.3.3 Execution

Each execution cluster is composed of a group of execution nodes, where each node is running an independent Blockchain. The execution cluster stores client smart contracts, and each cluster node holds a copy of that contract. In our proof-of-concept prototype, we statically define which clusters store which contracts; the clients only interact with the execution clusters that have their contracts. Each node of the execution cluster has a front-end, to receive client commands and relay back the result; and a back-end, to interact with the ordering service through remote method calls directed to a shared log interface with the methods append, lastIndex, readIndex, which will now be detailed.

3.3.3.A append

Every time an execution node executes their install or send methods, they will consequently create and send a transaction to call the ordering log contract's append method, which causes the clients' commands to be added to the log's map structure sequentially, on the next position available.

3.3.3.B lastIndex & readIndex

The commands to be executed are always stored on the ordering log. Therefore, the execution nodes must use the methods lastIndex and readIndex together to be able to know which contract they will be executing. The former method returns the last index on the log; if the latest index read by the node is smaller than the last index written on the log, then the node continues to read the next index until it reaches the last one. The latter method reads the log's command stored on the index provided as an argument. Note that the execution nodes only read commands that match the contracts installed on that cluster.

3.4 Order of Events

Our architecture prescribes a specific sequence of events through which these layers interact, as seen in Figure 3.3, and which can be summarized as follows.

The flow of the execution (provided that the contract is already installed) starts when a client calls sendCommand in their execution node's interface with the command to be executed. From this point on, the client can call getStatus asynchronously to check the progress of the command execution; initially, it should receive ”Pending” as a reply since the command has not been executed yet. The execution node's front-end receives the command and calls the append method on the ordering shared log contract. The ordering Blockchain receives the command and batches it with other pending commands in a certain order. Afterwards, the consensus protocol is executed and the order of these commands is agreed upon by all the nodes, and the commands are appended to the log. After that, each execution node calls lastIndex to check if there are still log positions to read. If there are, then it calls readIndex to get the command stored on that index; if the command belongs to one of its contracts, the node reads and executes it; otherwise, it moves on to the next index if it has not reached the end of the log. Eventually, the client calls getStatus and verifies if a majority of the cluster nodes has replied with the same hash (consensus). If so, then calls getResult on their execution node to get the result.

3.5 Design Characteristics

This base architecture enables us to explore several interesting characteristics that set us apart from existing systems. We list some of these unique characteristics, where the first four are features we can use to our advantage, while the last two are challenges that may handicap our system.

3.5.1 Independent Components

Normally, a monolithic design has its components overly intertwined and with a high degree of dependencies between them, possibly leading to some constraints and redundancies on the system. Having separate and independent layers allows us to avoid these redundancies and constraints by defining clear interfaces between each of these layers, and by delineating how they interact. Additionally, having two independent interface abstractions (execution and order) allows for a black-box abstraction model per layer. This means that the clients only know they are interacting with some application that will execute their command, and simplify the interface they use to interact with the whole system. Moreover, separate layers enable us to more easily maintain the system: we can change each layer individually without affecting the others. For instance, we can add more execution clusters without the clients noticing or disrupting the ordering layer, and we can change the consensus protocol on the ordering layer without changing the execution layer as well.

3.5.2 Scaling Out

In addition to the previous point, in many monolithic Blockchain systems, scaling out (i.e., increase the number of nodes) is challenging since more nodes only replicate the already existing problems – every node still has to store and execute every transaction with the adding factor of consensus becoming more complex with more nodes. Conversely, this architecture enables us to scale each element independently, and conveniently add more execution clusters to accommodate more client contracts, thus improving performance.

3.5.3 Parallel Execution

Having a shared log allows for the execution nodes to append and read from the log concurrently. Despite the global total order of the commands that the log provides, execution clusters only need to execute commands corresponding to their stored contracts. Hence, the system can parallelize the execution of multiple contracts, provided they belong to different execution clusters. This high degree of independence between execution clusters can provide direct benefits in system performance.

3.5.4 Execution after Agreement

In addition to providing a clear separation between layers, this architecture also establishes a novel order for the Blockchain transaction steps, namely the transactions are first ordered into a block, then that order is agreed upon by the ordering nodes, and finally, the transactions are executed. Since, in current systems, execution comes before agreement, the latter depends on the full execution of all the transactions in the block for the consensus protocol to start. In contrast, given that we are able to agree on the transaction order – which is cheaper than agreeing on what has been executed – execution no longer delays the consensus process. Additionally, nodes only agree on one block to be inserted on the Blockchain. And because the agreement phase is on the transactions that already have been executed, any block that was produced but ended up not being chosen to integrate the chain is wasted; and thus, every transaction execution also has been wasted. In contrast, due to choosing and agreeing on the order of the transactions first, our architecture ensures that we only execute what has already been agreed upon, and therefore no transaction execution is wasted.

3.5.5 Communication between layers

Having three layers in our system implies that extra communication must be implemented when compared to a monolithic system, which translates to a decrease in performance. In particular, as the number of execution clusters increases, more requests are exchanged between the execution layer and the ordering layer, possibly leading to significant communication overhead.

3.5.6 Post-execution simplified consensus

Since each execution cluster is in fact an independent, one-node Blockchain, we are removing the preexisting consensus that nodes must do when sharing the same Blockchain. For this reason, we must implement a simplified consensus operation (described in the following chapter) after the commands are executed, which also translates into a small overhead. This operation is, however, a must faster one when compared to the regular Blockchain consensus protocols, such as the PoW.

4 Implementation

Contents

4.1 Overview 39
4.2 Configuration Module 41
4.3 Client 41
4.4 Ordering 43
4.5 Execution 44
4.6 Benefits and Downsides 47

This chapter details the implementation for each of the architecture design aspects described in the previous chapter.

4.1 Overview

The starting point for our implementation is Ethereum, as it is one of the most popular and relevant Blockchain systems today that offers smart contract functionalities which are critical to our solution. As the codebase for an Ethereum-based implementation, we decided to use Hyperledger Besu [16], an Ethereum client written in Java. Besu provides essential features, which make it fitting to our needs: it has an EVM to deploy smart contracts, provides multiple consensus mechanisms (in particular, the IBFT 2.0 protocol [35]), and it has the option to create and test on a private Blockchain network.

To implement our architecture, our prototype is layered over Besu's codebase without changing the codebase itself. Therefore, to separate order and execution, we assigned Besu nodes, grouped as clusters, to each of these parts of the system: a single instance of a cluster of a specific type for the ordering layer and multiple same-type clusters to the execution layer. What differs between the two types of nodes is simply the smart contract they install: the ordering cluster installs the log contract, while each of the execution clusters only installs client applications – this way, the Besu codebase is completely unchanged.

Each Besu node is hosted in a different virtual machine, which has its own unique name: ”client”, ”exec” or ”order”, followed by a unique integer indicating the instance number of that type of virtual machine. Thus, for instance, the third client will be called ”client3”. This unique integer is also the machine's id as far as our implementation is concerned: it is through the machine's id that the client is chosen to communicate with a specific cluster, for instance. A Besu node is started by a bash process that executes the Besu binary which is pre-installed from the source repository [64]. When executing the binary, individual aspects of each node are defined through flags, such as the Peer-to-Peer (P2P) and Remote Procedure Call (RPC) host and port addresses. The former addresses are required for the nodes to communicate with the other Blockchain nodes, while the latter ones are necessary for interacting with external entities (e.g., nodes from other Blockchains, clients). Another relevant flag is the path for the genesis file. This JSON file contains the basic Blockchain configuration parameters such as the consensus protocol used, the gasLimit – which was set to the maximum to allow heavy transactions – and the Blockchain difficulty, among other parameters. Finally, there is a flag for the ”static nodes” JSON file, which acts as a permissioned blockchain setting: it defines the nodes that can participate in the blockchain network. All the ordering and execution nodes use this setup, only varying in specific parameter values.

The code that supports our architecture was written in JavaScript with Node.js [65] since Besu is already optimized to work with JavaScript code through libraries such as Web3.js, used to establish JSON-RPC requests. Additionally, the smart contract themselves were written in Solidity [66], one of the most accessible and well-known high-level smart contract programming languages, which is then compiled to the EVM bytecode. To support our system architecture, there are six main JavaScript modules (which are in fact .js files): the client, two execution front-ends, two execution back-ends, and a general configuration module utilized by all the other ones. This means that each execution node will have five processes running simultaneously: the four .js files and the besu Blockchain bash process that starts every Blockchain node. Each of these modules will be described in the following subsections. Finally, the logic for sending transactions uses Besu's unaltered implementation of sending a signed transaction through HTTP-RPC (using the web3 library [67]). Messages sent in transactions are always in a string format and are processed upon being received.

In figure 4.1, we can see the different components that make our implementation. In each of the following sections, we will describe each of them in more detail: section 4.3 refers to the client, section 4.4 to the execution, and section 4.5 to the ordering. Section 4.2 describes the configuration module, which is not represented in the figure since it has no real relevance for the architecture.

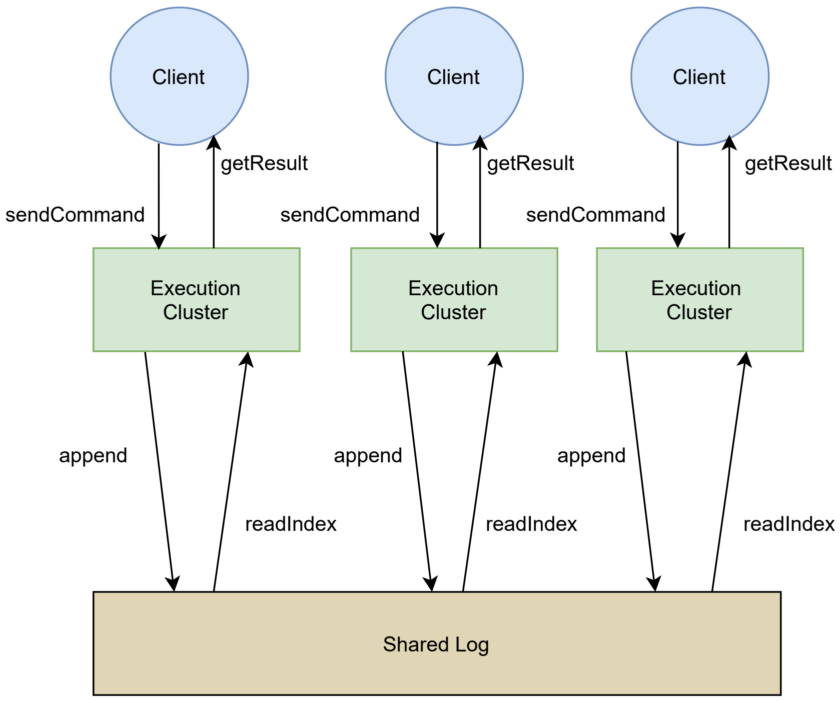

Figure 4.1: SepChain implementation components overview. Shared Log refers to the implementation on the ordering layer

4.2 Configuration Module

Having a configuration module to define static parameters common to the whole architecture is useful to automatize and organize development and testing. In our implementation of this feature, a .js file exports variables and functions which are then imported to every other .js file used. The total of code lines on this JavaScript file is 750.

The module contains fixed parameters such as the HTTP and web sockets addresses and ports used. In particular, it defines a function that returns the cluster (in the form of a list of node addresses) with which the client interacts. The pairing between the cluster and the client is based on the client's id, the total number of clusters and the number of nodes per cluster. Similarly, this module also defines a method for the client to get a specific address from a node from that cluster. Additionally, the configuration file contains a function that transforms a string to the actual command call object. And finally, it contains the functions to create and send transactions. The former creates a transaction object (i.e., a JavaScript object with a transaction's defining parameters) containing the from and to addresses, the data to send and the gas limit; the object is then signed and serialized into a hex number, which is then encoded into ASCII. The latter uses web3's sendSignedTransacton [67] method, which receives the encoded transaction object hex as an argument and sends this signed transaction to the indicated address.

4.3 Client

. The technology chosen for the clients to interact with the execution nodes was the WebSocket [68], implemented through its library for Node.js. In particular, the client uses the web-socket interface method "send" to send the required strings, and the method "addEventListener" to receive data.

The client JavaScript module defines four methods and contains a fifth method "main": a sequence of calls to the other four defined methods to simulate a simplified Ethereum client's interaction with the Blockchain: 1) install, 2) send, 3) getStatus, 4) getResult. The module itself receives as arguments the smart contract name and the command name. Every time the client sends a socket message, the message content is the string for the execution nodes to parse, separated by special characters. This string includes the two client parameters (contract name and command name), the client id (retrieved from the client's machine name), the request id (incremented per client request), and the method name ("install", "send", "getStatus", "getResult"). We will now detail each of these methods.

4.3.1 Install

Through the install method, the client sends its smart contract application to be installed. To achieve this, the client establishes a web-socket connection with the address predefined on the configuration file and uses the socket's send method to send a string with the following fields, separated by a special character: the name of the contract to be installed (the client's parameter), the method (”install”) and the client id and the request id.

4.3.2 Send

The send method follows the same principle: it sends the command to be executed on the previously installed contract. It establishes a new web-socket connection to the same address, but this time it sends as a string the contract name, the command name, the method name (”send”), the client id and the request id.

4.3.3 getStatus

Each client uses getStatus to request the status of the command execution until a majority of the replies contain the same hash. Specifically, this translates to the client staying in a loop until the getStatus method returns consensus on the hashes received. This method iterates through a list of the predefined cluster addresses and establishes a different, asynchronous web-socket connection with each of them, leveraging JavaScript's native asynchronous capabilities. For each of these cluster nodes, the client sends the string with the contract name, command name, client id, the request id and the method name (getStatus). It then waits for the execution cluster's replies and stores them in a map structure, where the key is the cluster address list index, and the value is the status received. Afterwards, in each of those asynchronous iterations, a new map structure is used to store the status received and how many times it was received. In addition, two variables store both the maximum counter and the respective status with the maximum counter. After all the asynchronous connections with all the execution cluster nodes are finalized, the client then checks for consensus: if the maximum counter registered is higher than half of the total nodes on that cluster, then there is consensus on that status. Provided that the status agreed on is a hash – since the command might have not been executed yet and the string ”pending” is returned in this case – then the client can request the command's result. It might also happen that the majority of the requests are delayed. Thus, the client keeps track of the current request id and considers ”deprecated” any requests with a smaller id. Because the client collects the reply from a majority of the nodes that may execute their command, this serves as a group signature that guarantees that whatever reply is received is trustworthy.

4.3.4 getResult

After the client has a consensus on the received hash, the client calls the getResult method. This method will choose one of the execution node addresses from the achieved majority and establish a new web-socket connection to it. The client sends a new string with the same parameters as getStatus except for the action, which is now ”getResult”. It then waits for the final result from that one execution node. The method for choosing the particular address to communicate with uses the modulo between the client id and the number of nodes on the majority as the index.

4.4 Ordering

To implement the ordering structure, we simply reuse the functionality of Besu: each blockchain node batches the command transactions received into an ordered block and agrees on that order through a consensus protocol. As a result, commands are added in the decided order to the log and become available to the execution layer. SepChain’s implementation decides how the log is implemented: each ordering Blockchain node has a log smart contract installed, as it can be observed in figure 4.2. This contract has three methods and one map structure that symbolizes the log itself, mapping from an integer – the log position – to a string – the string sent by the client, containing the contract name, and the command name. The first method is append, which receives the string, adds it to the log on the current index position, and increments the current index position afterwards. Second, the readIndex method returns the string stored in the index received as argument. Finally, lastIndex returns the last log position (i.e., the highest index).

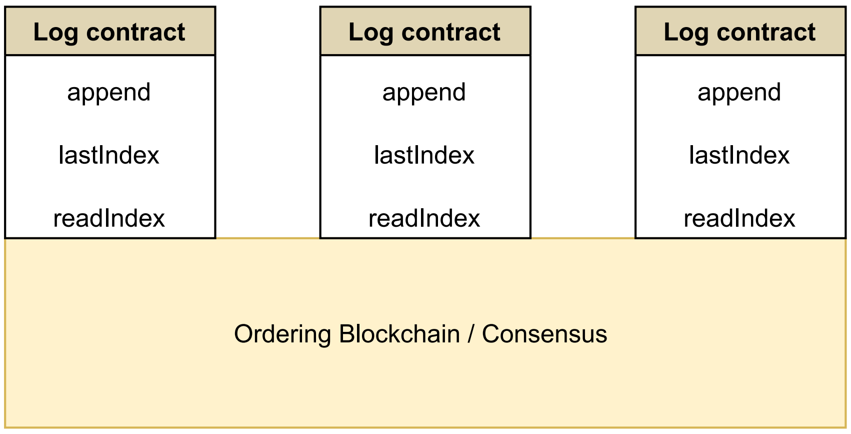

Figure 4.2: Representation of the ordering blockchain holding the replicated smart contracts as the shared log, which implements three methods.

4.5 Execution

The execution implementation is split across a total of four JavaScript modules: two belong to the front-end, and the other two to the back-end. A general view of what constitutes an execution node can be seen in figure 4.3.

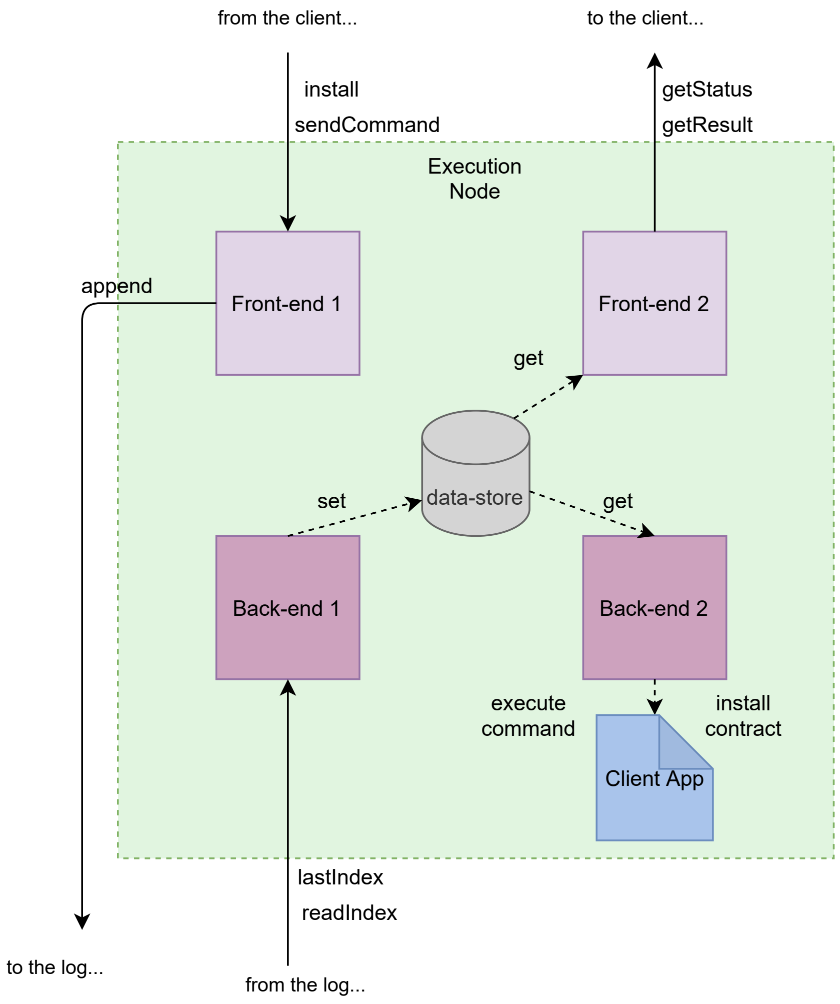

Figure 4.3: Implementation components of an execution node.

The next subsections describe each of these modules in more detail. Note that the order in which the subsections are displayed is relevant, as it represents the sequence of modules that the client request follows from the moment it is first sent. First, front-end 1 (FE1) collects the request and sends it to the ordering Blockchain. Then, back-end 1 (BE1) reads the request from the ordering Blockchain's log and writes the request on a local file. Back-end 2 (BE2) then reads and executes that request. Finally, front-end 2 (FE2) replies to the client with the hash of the result – and eventually the result itself – once BE2 finishes executing.

Although two modules for the execution would be enough to support our design – one for the back-end and another for the front-end – having two modules per back-end and two per front-end allows for a higher degree of parallelism. Both FE1 and FE2 wait for client messages – to send and get the status of a command, respectively – but since both these messages have a high degree of independence, it is beneficial to have separate modules and message queues to process them. Similarly, the system benefits from BE1 and BE2 acting independently because instead of having a single module reading and then executing, BE1 can read while BE2 executes, which is a major performance improvement.

4.5.1 Front-end 1

The first front-end program deals with relaying client requests to the ordering chain (either install requests or send command requests) and it was implemented in 90 lines.

4.5.1.A install & sendCommand

Every time this module receives a message string from a client through its web-socket connection, it checks whether the method to be executed is either install or sendCommand. In both cases, it adds the cluster id (predefined and passed as an argument when the JavaScript file is executed) to the string.

4.5.1.B append

Once the string is formed, following an install or sendCommand method, this module then creates and sends a transaction to the ordering Blockchain to execute the append method on the log contract, which will add this string to the log contract's map.

4.5.2 Back-end 1

The first back-end module deals with checking if a new contract has been added to the log. It was implemented in 118 lines.

4.5.2.A lastIndex

First, the back-end executes an infinite loop where it continuously calls the lastIndex log method. This method obtains the last index written on the log contract's map. After obtaining the last index, it is subtracted from the current index (stored locally). If the result of this operation is greater than 0, there is in fact a new command added to the log.

4.5.2.B readIndex

If in fact there is a new command added to the log, then the back-end calls the log's readIndex method with the current index as an argument, which returns the string sent by the client; the current index is then incremented. The program splits the string by its special delimiter character. If the cluster id matches the one read from the log, a data-store [69] map – which is available for every module to access – is set with the number of strings (i.e., the number of client commands) read as key, and the string itself as the value.

4.5.3 Back-end 2

The second back-end file deals with the contract execution part, and it was implemented in 217 lines. Similarly, the back-end is again in an infinite loop. First, it checks if there is a new string (i.e., command) to read. To do this, it subtracts the data-store map number of strings read from its local counter. If the difference is greater than zero, there is a new command to execute. The first thing to verify is if the request is to install a smart contract. If so, the contract is compiled locally, and a transaction is created and sent to the node's own address. Once the transaction is finalized (i.e., the contract has been added to the execution node Blockchain), the contract address is stored locally.

However, if the request is to execute a command, there are a few changes. First, a transaction is again created and sent to the node address. Then, once the transaction finishes, its result is hashed using the sha256 algorithm, using the Node.js cryptography library. Lastly, a new data-store map is created to store the client's command string mapped to a list of three elements: the result, the result's hash, and the log index.

4.5.4 Front-end 2

The second front-end program deals with getting the status and getting the results back to the clients, following their requests. It was implemented in 149 lines. The program listens on their respective web sockets for messages (on a different port than the first front-end program). Once a message is received, the program check what method to be executed – either getStatus or getResult.

4.5.4.A getStatus & getResult

If the goal is to get the status, first the program checks if the index stored locally matches the one read on the data-store result structure. If it does, it means that the request is still pending, and thus a "pending" message is sent back to the client. If the index stored is higher than the one read from the data-store, then the request is "deprecated", which is also sent as a reply to the client. Otherwise, there are no issues with the request and the program sends the hash back to the client. However, if the action is to get the result, the program will read the result stored on the data-store result structure and send it back to the client.

4.6 Benefits and Downsides

With our design and implementation, we showed that we successfully implemented the planned system architecture on top of Besu's already existing system without changing the codebase. The clients are simple modules that send requests and receive replies to/from the execution nodes via sockets. The ordering layer is implemented by writing a log smart contract as the log structure on top of the ordering Blockchain. The execution layer is the most complex layer with the largest functionality: each execution node is itself a one-node Besu Blockchain that has to interact with the log contract to execute the client commands.

In particular, by implementing the planned architecture, we also materialized its design characteristics. We were able to develop each layer as an individual component of the system, which can be changed without altering the other, or affecting the client – for instance, Besu's consensus can be changed by changing a few lines on the Blockchain genesis file. Additionally, the clients can have their commands executed through four main simple methods, without needing to know how the system works. Moreover, the execution layer can be expanded by simply adding more execution clusters (i.e., scaling out) to accommodate more clients, and to further reduce the load among the existing clusters and effectively parallelize the execution of smart contracts belonging to different execution clusters. Furthermore, because the execution clusters can constantly contact the log structure through their Back-end 1 modules, smart contract execution in each cluster becomes even more efficient for each Back-end 2 module to execute smart contracts. Finally, since the order of the commands is already agreed upon and all the commands on the log structure must be executed, the execution clusters are not slowing down the consensus protocol, nor are they wasting any command executions.

However, as predicted, some of these characteristics handicap our system. Since we defined independent layers, the communication between them is a source of overhead. Specifically, the execution nodes are constantly reading from the log in an infinite loop, which can be very inefficient. Likewise, each client establishes a socket connection every time they want to send a command or get a status, which can be resource-intensive. Added to this, there is the extra challenge of having all the clients securely learn about the result they have obtained. To this end, an extra, simplified consensus phase turned out to be required as a way to guarantee a group signature on the execution cluster's replies that the client can trust, which is, nevertheless, another source for overhead.

It is also important to point out that the choice of not changing the Besu codebase also has its advantages and disadvantages. On the one hand, we managed to implement the system in less than

$2,000$ lines of code, which is a marginally small number of lines when compared to the $500,000+$ lines on Besu's codebase [64]. Additionally, by reusing Besu's implementation, we also inherited its security and fault-tolerance features. This means that we are able to maintain a $3f+1$ fault-tolerance on the ordering Blockchain (whereas the execution layer only requires $2f+1$ nodes). Furthermore, Besu's consensus protocol also provides a group signature on each block that is created, which we can also make use of to ensure that every block produced on the ordering Blockchain is in fact correct.

On the other hand, building on top of Besu translates to significant overheads. In the execution layer, we are still using a full-fledged, one-node Besu Blockchain, while ideally we only need to strip down the EVM to execute the client contracts – we only need the Besu code that encodes only the execution of a smart contract. Similarly, on the ordering layer, the overhead from having the log as a smart contract on top of the ordering Blockchain can be avoided by stripping down the EVM to implement an independent log component.

5 Evaluation

Contents

5.1 Overview 51
5.2 Hardware 51
5.3 Experimental Methodology 51
5.4 Top-level comparison 52
5.5 Impact of ordering nodes 55
5.6 Impact of scaling out 57
5.7 Impact of added security 58
5.8 Impact of execution costs 60
5.9 Impact of execution nodes 61
5.10 Impact of Sharding 63
5.11 Summary 66

In this chapter, we showcase the results of the evaluation of SepChain in comparison to running the same contract logic running directly on Besu, which we use as a baseline system.

5.1 Overview

In both systems, the consensus algorithm utilized was IBFT 2.0 [35], which runs a PoA [34] protocol. Although we are only testing this consensus algorithm in particular, we believe that these results can be generalized to the wider array of BFT consensus – or even PoW consensus – since our architecture's principles can still be applied: we focus on changing separating execution from consensus, thus, e.g., if a better consensus protocol was used, it would only highlight further the importance of separating the expensive execution from a lighter-weight consensus.

5.2 Hardware

The following experiments were realized in INESC's Local Area Network (LAN) cluster utilizing a variable number of machines but always with the same specifications: 40GB RAM, 1TB HDD and the Intel(R) Xeon(R) CPU E5506 @ 2.13GHz - 8 Cores processor.

While there are nearly 15 machines with these specifications in the cluster, they are not nearly enough to simulate a full-fledged Blockchain system nor to have enough clients to stress the system to the desired extent. Therefore, we made use of Virtualbox [70] to create virtual machines to instantiate both multiple Blockchain nodes and multiple clients per machine, with 2GB of RAM and 1GB of RAM, respectively. This means that each physical machine is able to hold 15 Blockchain nodes and 30 clients, given the RAM constraints.

It is important to note that, because each node/client is a Virtual Machine (VM) and shares the physical machine's resources, our results may be involuntarily skewed due to these constraints.

5.3 Experimental Methodology

To compare both systems, we measured the throughput (in transactions per second) in each experiment we conducted. Since there is not a single way to measure the throughput, we measured it in a way that was both practical but that also reduced inaccuracy to a minimum. To this end, in each experiment, every client sends the same number of requests to the system, and we count the total time it takes since the first request is sent and the last one is received. Thus, $Throughput = (requestsPerClient * clients)/totalTime$. The number of requests sent is such that, for that experiment's configuration, the total time is of at least five minutes, to ensure the startup and wind down do not skew the results.

Additionally, the number of clients is always such that the system becomes saturated, i.e., if more clients are added, the throughput does not increase.

Note that although we made an effort to present the most complete and accurate set of results that we could achieve, the methodology is not perfect, and so we list possible sources of noise or imbalance in the analysis. In particular, manual testing opens the possibility for the VMs to have inconsistent delays on start-up. Additionally, we defined a minimum runtime of each experiment, which was enough to reach logical results but also allows for some experiments to run for longer than others, which might have slightly influenced the results obtained.

To evaluate SepChain's performance and compare it with Besu's, we decided on a set of experiments that tested an array of parameters: the number of ordering nodes, the number of execution clusters, the number of execution nodes per cluster, and the amount of gas consumed by the smart contract. Note that the amount of gas is represented by the number of iterations of a for loop, which is programmed into the smart contract invoked by the client. For instance, $50,000$ iterations roughly correspond to $8,900,000$ gas. Additionally, in each of the experiments, the clients interact with the system by sending requests sequentially on a closed-loop. The maximum number of clients settled upon (90) is simply due to the limited number of physical machines available.

Each of the following sections will present one of these experiments, where some of these parameters are changed, while the others are fixed. When measuring the amount of gas consumed, we use the fixed values of 100 and $50,000$ iterations. These values represent the minimum and maximum gas tested, or, in other words, light and heavy smart contract requests. Similarly, in some experiments, the values of execution clusters are fixed as 9 and 35 clusters, which means that there are 9 and 35 execution clusters, each with 3 execution nodes. These values were picked to demonstrate two levels of SepChain's performance: with 9 clusters SepChain is slow but still manages to achieve decent performance. Conversely, 35 is the maximum amount of clusters we were able to run experiments given the resources we had available, and thus show the best performance possible for the VM-based environment constraints.

5.4 Top-level comparison

The first experiment aims to establish a top-level comparison between Besu and SepChain – it compares how both systems deal with a constantly increasing system load, which is represented both by an increase in the number of clients interacting with the system as well as by two types of contract: with 100 and $50,000$ iterations. Additionally, and since SepChain greatly depends on the number of execution clusters available, we present two distinct configurations with 9 and 35 clusters, as a way to showcase a wider range of results. Furthermore, we fix three execution nodes per execution cluster and four ordering nodes. These parameters are fixed since they are not as influential, which is proven in the following experiments.

Looking closely at Figure 5.1, with 9 clusters and $50,000$ iterations, SepChain is saturated with 15 clients with a performance of 0.53 tx/s, increasing in 0.2 tx/s from the 5-client mark. Besu, in turn, only increases 0.04 tx/s from 5 clients to 30 clients and saturates with the latter at 0.38 tx/s. Therefore, we conclude that SepChain is consistently better in this setting. Conversely, with 100 iterations on the smart contract, we saturate our performance with 30 clients at 2.44 tx/s while Besu's performance cannot be saturated with only 90 clients – it reaches 22.72 tx/s and it would most likely reach an even higher number with more clients. Figure 5.2 shows the data points more clearly by applying a log scale to the y-axis. Note that it is expected that SepChain is worse than Besu when the gas values are low – we inherit substantial overhead by implementing SepChain on top of Besu. It is only when the gas values increase that our implementation is able to amortize its overheads: because we can scale out with 9 clusters, the system load (both in terms of clients and gas) is split into different clusters, showing a better performance than Besu, which has the full system load on a single Blockchain. Conversely, it is somewhat unexpected that SepChain saturates so quickly; this points towards the inherent overheads of SepChain implementation.

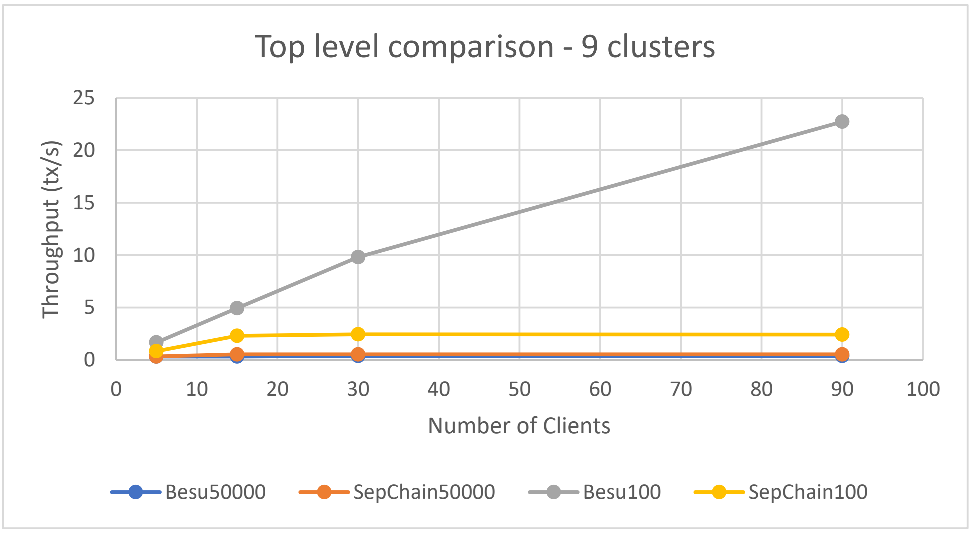

Figure 5.1: Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 9 execution clusters. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively

Now observing Figure 5.3, only SepChain changes – Besu cannot be scaled up and thus maintains the same number of Blockchain nodes. With 35 clusters, we can see a major improvement in our performance: at 90 clients we have almost 4 times the performance, sitting at 9.55 tx/s. With this configuration, SepChain cannot be saturated with 90 clients – we have an increase of more than $2x$ from the 30-client mark. Figure 5.4 shows the data points more clearly by applying a log scale to the

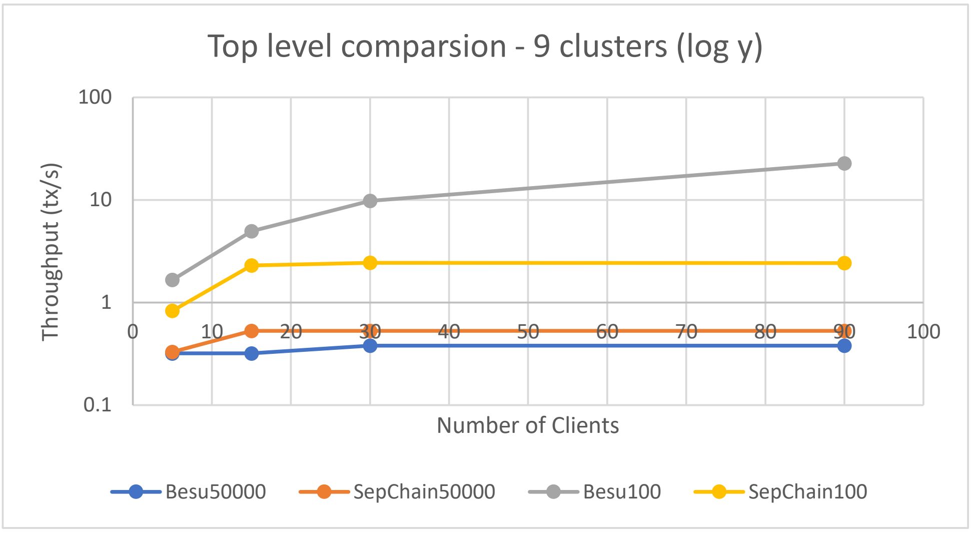

Figure 5.2: Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 9 execution clusters. A log scale is applied to the y-axis. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively

y-axis. This improvement in performance is expected since having more clusters directly translates into less system load per cluster, which in turn makes each cluster faster. The improvement over Besu's performance becomes clearer the more gas there is in the system. Additionally, it is also logical that the saturation point increases to further than 90 clients in this setting, since more clusters are able to handle a larger system load.

Finally, Table 5.1 depicts a theoretical estimate for SepChain's behavior with 75 clusters. The values at 90 clients are deduced from our performance with 1 cluster, as observed in section 5.6, multiplied by 75. We also assume there is a linear progression with the increase of the number of clients – if there are 3 times the clients, the performance is 3 times better. Of course, this would not be accurate in a real system due to inherent overheads, but it gives us a solid estimate of such a system configuration. We chose 75 clusters since it is the number of clusters from which we can match Besu's performance at 100 iterations.

System / Number of clients	Besu50000	SepChain50000	Besu100	SepChain100
5	0.32	0.33	1.66	1.25
15	0.32	1	4.95	3.75
30	0.36	2	9.81	7.5
90	0.38	6	22.72	22.5

Table 5.1: Experiment 1: Besu's throughput value and SepChain's theoretical estimate of the throughput value as the number of clients varies, where the latter system has 75 execution clusters. The 50000 and 100 refer in the column names refer to the number of iterations used in each system

By looking at the previous Figures, we can confirm that, performance-wise, the choice of the system that performs better depends on the number of iterations (i.e., the amount of gas) in the contract being

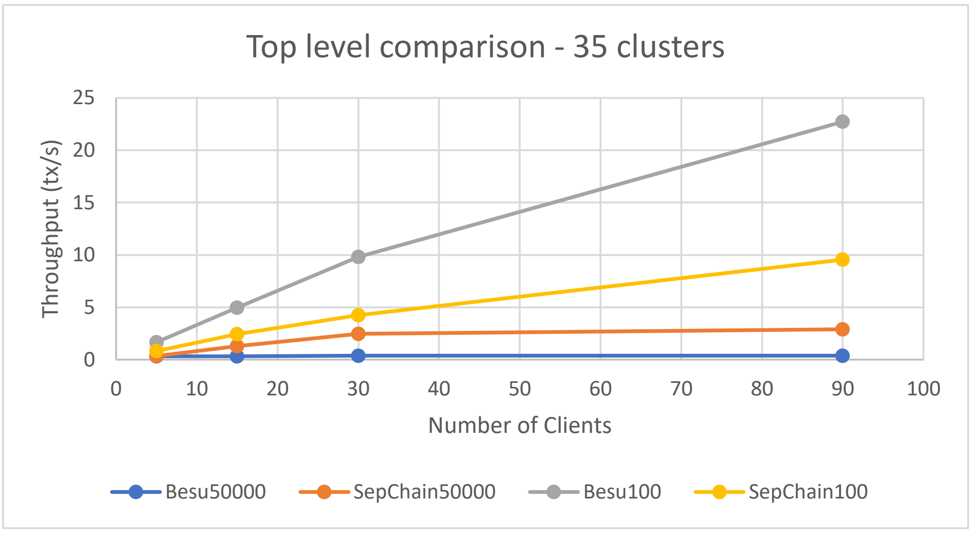

Figure 5.3: Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 35 execution clusters. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively

executed: Besu is better at 100 iterations, while we outperform Besu at $50,000$ iterations. Additionally, we can also observe that SepChain's performance improves when the number of clusters increases. Finally, the number of clients is also crucial since it has to be high enough for the system to be saturated; low client numbers reduce the throughput drastically because the load that these clients offer is insufficient to saturate the system. Furthermore, the reason why Besu is so much faster than SepChain when the number of iterations is low is simply that SepChain has extra phases of communication and consensus overheads, adding to the fact that it is implemented on top of Besu's implementation. In addition to this, Besu clients only have one Blockchain to interact with and the Blockchain nodes only communicate among the nodes on the only existing Blockchain. In contrast, our clients communicate only with one layer, and all of the layers must communicate with each other to send, receive, and reach a consensus on the requests being sent by the client. Although Besu has the upper hand when the gas values are low, this experiment shows that SepChain is able to scale out by adding more clusters to progressively match Besu's performance with low gas contracts, and outperform Besu with high gas contracts.

5.5 Impact of ordering nodes

With the assumption that the execution layer is the bottleneck in the system, we now take a step back and analyze the effects of removing the cost of execution. Thus, this second experiment intends to study how fast SepChain is able to order transactions. Therefore, in this experiment exclusively, we only test the ordering part of SepChain. That is, the clients still send their transactions to the execution clusters

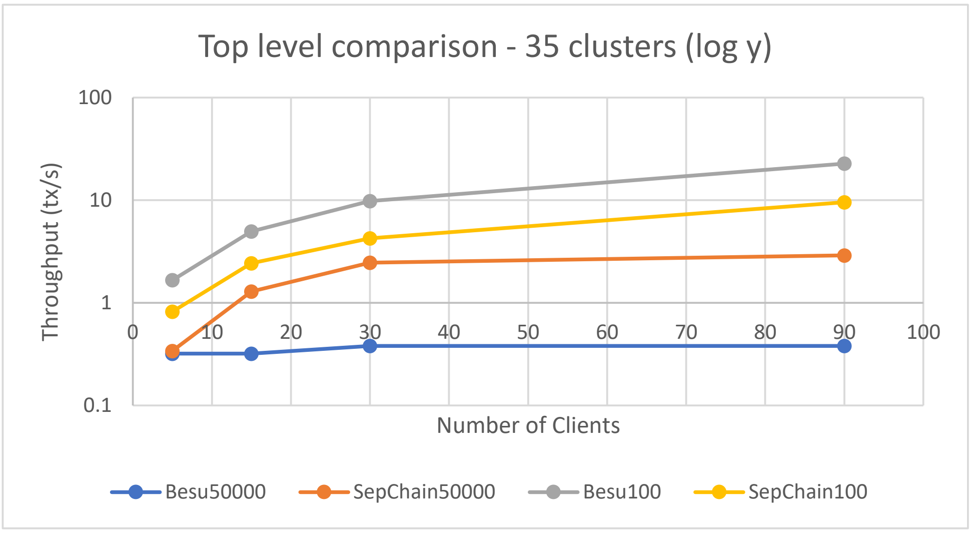

Figure 5.4: Experiment 1: Besu and SepChain throughput values as the number of clients varies, where the latter system has 35 execution clusters. A log scale is applied to the y-axis. The blue and grey lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The orange and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively

and these send to the ordering Blockchain, but, as soon as the ordering Blockchain finishes adding the transaction to the log, the result (i.e., the log position of their command) returns to the client.

Figure 5.5 shows four different lines: two for Besu's performance and two for SepChain's ordering performance, each system with $50,000$ and 100 iterations. As expected, the performance of Besu is better with fewer iterations but drops as the number of nodes increases. This is expected behavior since having more nodes in a blockchain tends to slow down the consensus protocol. With 100 transactions, it drops from 22.78 to 8.49 tx/s; and with $50,000$ it drops from 0.38 to 0.19 tx/s. SepChain's lines (which are in fact the same line) also decrease as the number of nodes increases, going from 23.64 to 7.74 tx/s. Figure 5.6 shows the data points more clearly by applying a log scale to the y-axis.

SepChain shows two lines overlapping, which are in fact the same line, because SepChain's ordering is indifferent to the number of transactions that are going to be executed – the gas of the log contract is always the same. Conversely, a $50,000$ iteration contract is bound to take longer to execute than a 100 iteration contract, which still is a problem for Besu. Additionally, we can also observe that we can match Besu's performance at 100 iterations, but we can maintain that same performance for higher gas values such as $50,000$ iterations while Besu has a severe drop in performance.

In this case, because SepChain is exempt from the execution and post-execution consensus phases – it only uses the execution nodes to relay the requests to the ordering Blockchain – the performance bottleneck is on how fast the ordering phase is. For that reason, the performance of both systems is dependent on how many nodes there are on the Blockchain: more nodes equate to making consensus slower because communication becomes harder with a higher number of nodes.

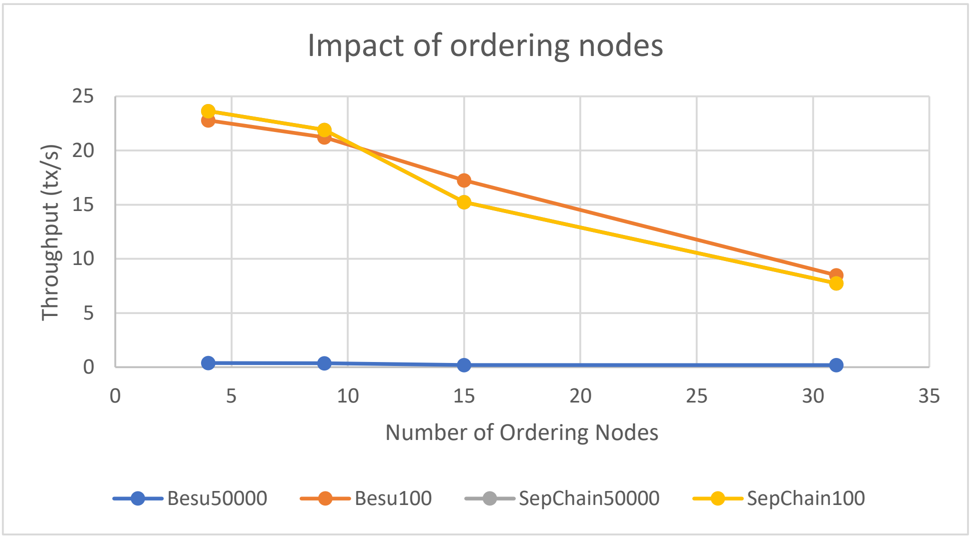

Figure 5.5: Experiment 2: SepChain's ordering blockchain and Besu throughput values as the number of (ordering) nodes varies. The blue and orange lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The grey and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. Note that the yellow line overlaps the grey line.

This experiment shows the theoretical limit for how fast we can order, which is also the upper bound for how fast we can execute. Even though it is expected that our performance always increases with more clusters added, eventually the ordering Blockchain will be saturated: the ordering section becomes the system bottleneck. Thus, from that point onward, adding more clusters is irrelevant to improving the system performance.

5.6 Impact of scaling out

The previous experiment allowed us to determine how fast we can order, which is, in other words, the upper bound to our execution layer performance. Therefore, this third experiment intends to show the effects of adding execution clusters, i.e., scaling out the system, while maintaining the number of nodes (3) per cluster, to uncover if we are able to reach that upper bound. We, again, test different cluster values while varying the number of iterations between $50,000$ and 100. Throughout this experiment, the number of order nodes is 4.

In Figure 5.7, we can verify that the throughput is affected as the number of execution clusters changes: the more clusters there are, the higher our throughput is. With $50,000$ iterations, the throughput varies from 0.084 tx/s (1 cluster) to 2.9 tx/s (35 clusters). The predicted value for 75 clusters is 6.3 tx/s, calculated by simply multiplying the throughput value for 1 cluster by 75. With 100 iterations, 1 cluster stays at 0.3 tx/s, while with 35 clusters we reach 9.55 tx/s. The predicted value for 75 clusters is 22.5 tx/s, calculated the same way as before. Note that the latter value is close to the maximum performance

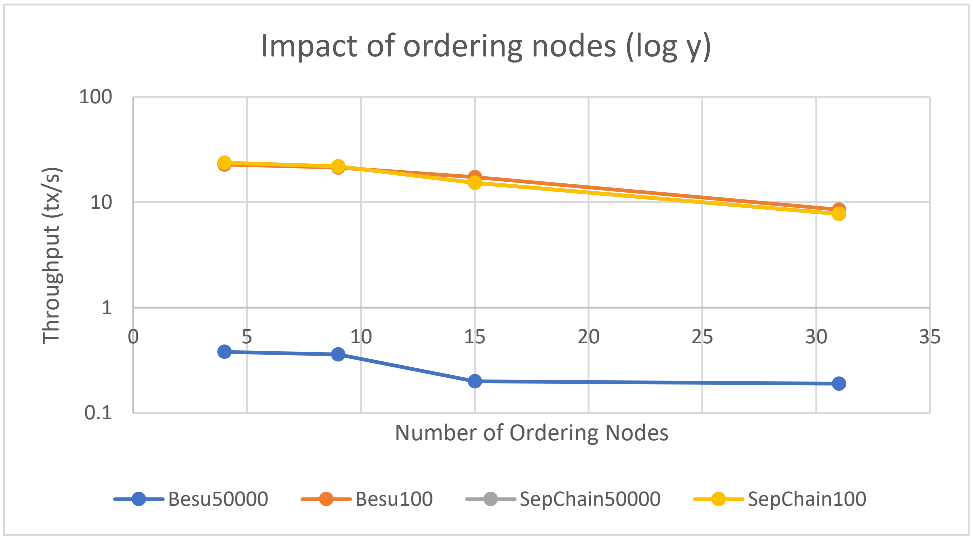

Figure 5.6: Experiment 2: SepChain's ordering blockchain's and Besu's throughput values as the number of (ordering) nodes varies. A log scale is applied to the y-axis. The blue and orange lines represent Besu's throughput with 50,000 and 100 iterations, respectively. The grey and yellow lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. Note that the yellow line and the grey line are overlapping.

value observed for the ordering layer, which suggests that 75 is a close number of clusters to what we need to show maximum performance in the execution layer. Figure 5.8 shows the data points more clearly by applying a log scale to the y-axis.

This experiment showcases the expected behavior: allocating more clusters translates to having more units to execute requests in parallel. Since each cluster is only responsible for executing a set of client contracts, as the cluster number increases there are fewer clients assigned to each cluster, which, in turn, speeds up the executions per cluster and contributes to an overall system improvement.

5.7 Impact of added security

Following the footsteps of the previous experiment, this fourth experiment intends to pinpoint if adding security to the system affects the ability to scale out. It showcases how the performance of both systems varies as the number of ordering nodes increases (4, 9 and 15), for two iteration values: 100 and $50,000$. And, in particular, SepChain is tested with multiple settings varying also the number of clusters between 1, 2, 4, 8, 16 and 32. The number of nodes per execution cluster is always 3. The emphasis on security comes from the simple fact that the more nodes there are on a Blockchain, the more secure it is since it becomes less likely that an adversary can control a sufficiently large fraction of those nodes.

For the experiment with $50,000$ iterations (figure 5.9), and looking at the 4 ordering node data points, our throughput increases from around 0.08 tx/s to 2.6 tx/s. Besu is fixed at 0.38 tx/s, which means

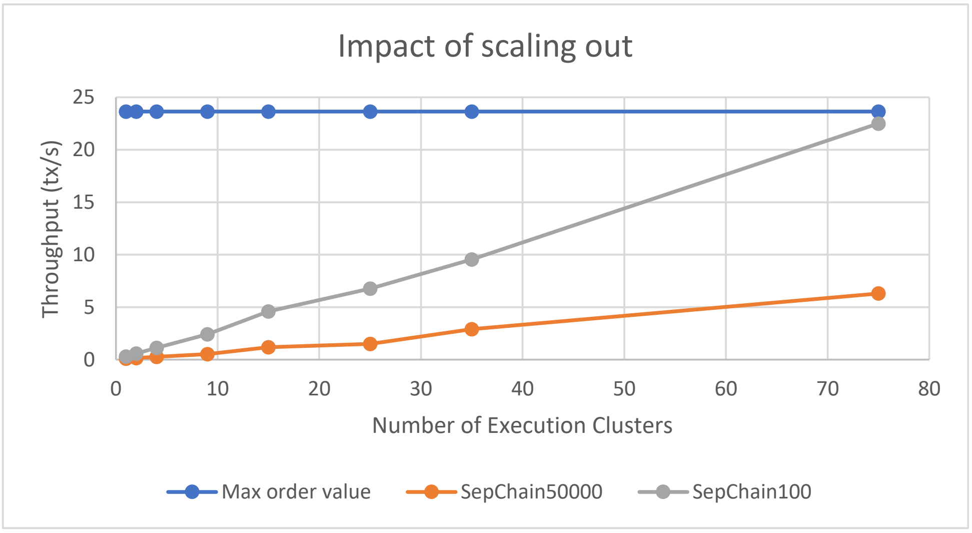

Figure 5.7: Experiment 3: SepChain throughput values as the number of execution clusters varies. The orange and grey lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. The blue represents the system's contract execution upper bound as seen in Section 5.5.

that our performance is better only from the 8 cluster mark onwards, which is set at 0.46 tx/s. For the experiment with 100 iterations (figure 5.10), looking again at 4 ordering nodes, our throughput values now range from 0.3 tx/s (at 1 cluster) to 8.61 tx/s (at 32 clusters). Besu, however, is able to reach 22.78 tx/s, as seen before. Figures 5.11 and 5.12 show the data points more clearly by applying a log scale to the y-axis.

An important aspect to notice in this set of results is how the systems are affected by the increase of ordering nodes. Besu is clearly impacted: at 15 nodes, its performance has a clear drop. In contrast, SepChain is able to show similar performance regardless of the number of ordering nodes. This is not as clear to see on the 32 cluster lines, as our performance also appears to be affected by the ordering node increase. Conversely, the reason why the performance is affected is most likely due to the increase in the execution cluster number: more execution clusters translate to more communication overhead when interacting with the ordering log.

Finally, we must also highlight the fact that as the number of clusters increases, by a factor of $2x$, our throughput also increases close to that same factor. The reason why the system is not perfectly doubling the performance as with each cluster node increase is, again, due to system implementation overheads – we are implementing SepChain on top of Besu's architecture.

This experiment allows us to demonstrate how we can in fact scale out with the added value that even if we want to increase security – by adding more ordering nodes – we are able to do so without jeopardizing performance. Again, this is something that Besu cannot achieve.

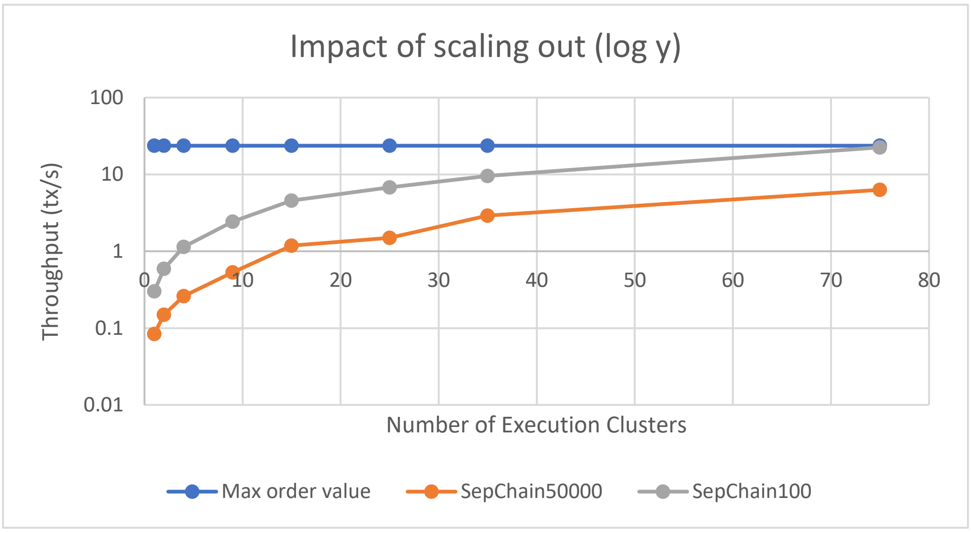

Figure 5.8: Experiment 3: SepChain throughput values as the number of execution clusters varies. A log scale is applied on the y-axis. The orange and grey lines represent SepChain's throughput with 50,000 and 100 iterations, respectively. The blue represents the system's contract execution upper bound as seen in Section 5.5.

5.8 Impact of execution costs

Sections 5.6 and 5.7 demonstrated that SepChain has the ability to scale out. This fifth experiment explores how gas affects performance, more specifically how it affects contract execution, and how much SepChain needs to scale out to match Besu's performance. To simplify these measurements, as mentioned before, different levels of gas are measured as the number of iterations in the smart contract loop. This experiment is composed of two parts, one where SepChain has 9 clusters and another where it has 35 clusters. Additionally, the number of nodes per execution cluster is always 3, and the number of ordering nodes is fixed at 4.

Figure 5.13 shows the 9 cluster experiment and has six different data points, each corresponding to a different number of iterations per system. Immediately, we can verify that the more iterations there are in a contract, the worse the throughput is, which is expected since executing a smart contract with more iterations takes longer, hence slowing down the system. Particularly, Besu's throughput ranges from around 22.72 tx/s (when at 100 iterations) to 0.38 tx/s (when at $50,000$ iterations). Similarly, SepChain suffers a drop in performance, but ranging from around 2.4 tx/s (when at 100 iterations) to 0.29 tx/s at $50,000$ iterations. The $13,000$ iteration data point of 1.21 tx/s is particularly relevant, as it shows the point from where (with this setting) SepChain outperforms Besu. Figure 5.14 shows the data points more clearly by applying a log scale to the x-axis and the y-axis.

Figure 5.15 shows the 35 cluster experiment, which also has 6 different data points per system. Besu's data points are the same, and there is a similar decrease in performance as the number of iterations increases. In particular, with 100 iterations, our throughput is 9.55 tx/s, while at $50,000$ iterations

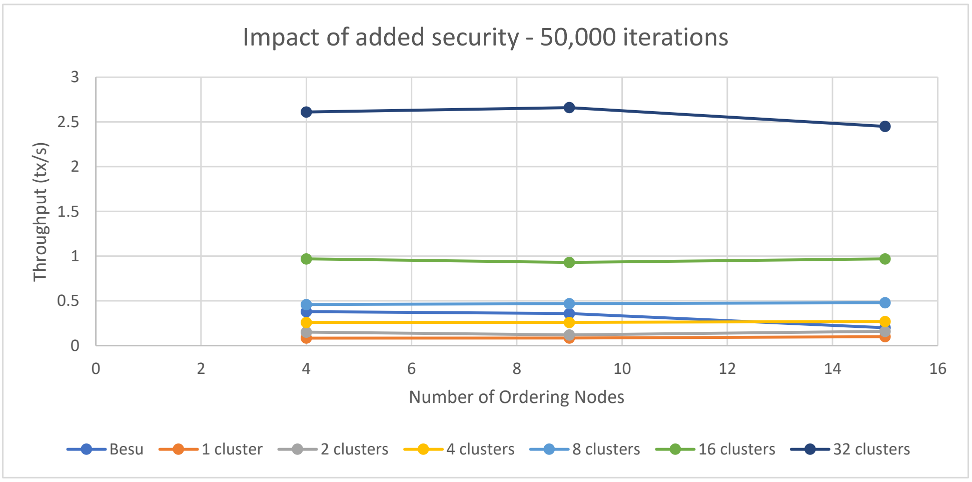

Figure 5.9: Experiment 4: Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 50000 iterations. The lighter blue line represents Besu's performance. Every other line (from orange to dark blue) represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32].

it decreases to 2.9 tx/s. With this system configuration, the intersection point is now at $4,000$ iterations, with a throughput value of 7.65 tx/s. It makes sense that the intersection is at an earlier point since Besu's performance stays the same, yet ours improves with the addition of more clusters. Figure 5.16 shows the data points more clearly by applying a log scale to the x-axis and the y-axis.

All in all, this experiment makes clear that more iterations (i.e., more gas per contract) represent a higher load on the Blockchain, leading to the contract execution taking longer and the throughput decreasing in all systems. As the number of clusters increases, we are able to have less load per cluster (working in parallel) which speeds up the overall system performance. Additionally, it would be expected that as the number of iterations doubles (i.e., the gas doubles), Besu's performance halves. However, this is only observable from around $4,000$ onward, as seen in Figures 5.14 and 5.16, where the Besu line becomes almost a straight descending line. The reason for this is the following. When a request is executed, there are two parts that make the cost: the overhead of the execution of the contract, and the complexity of the request itself. Therefore, we only observe the expected halving once the complexity of the request itself is much more prominent than the overhead – from $25,000$ to $50,000$ the performance is halved. Hence, when the number of iterations is small, the overhead is the dominant factor – there is not much of a difference between the values at 100 and 1000 iterations.

5.9 Impact of execution nodes

With the basis from the previous experiments, we now start exploring SepChain's capabilities in comparison to Besu. In particular, this sixth experiment progressively adds more execution nodes to a fixed

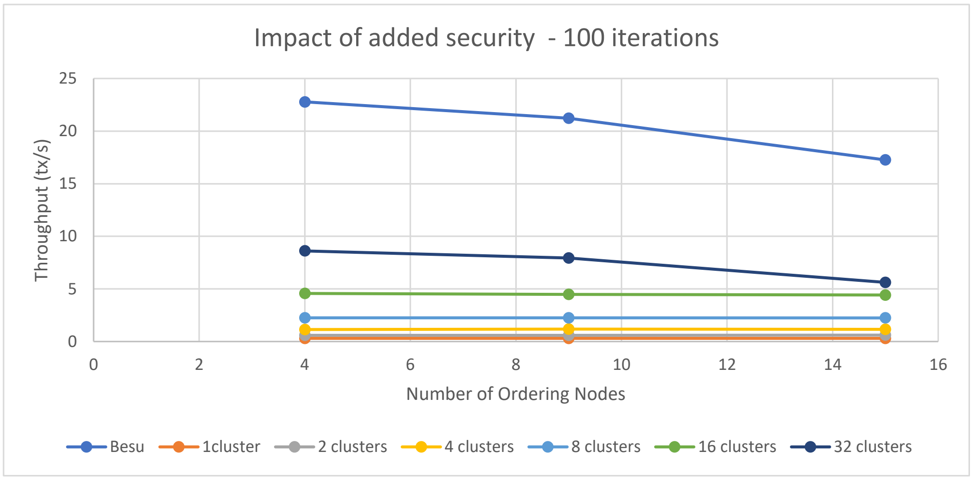

Figure 5.10: Experiment 4: Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 100 iterations. The lighter blue line represents Besu's performance. Every other line represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32].

number of 9 clusters with three nodes each, with the aim of discovering how it impacts the system. The number of ordering nodes is, again, fixed at 4.

In our case, by looking at figure 5.17, we observe that adding more nodes does not impact performance – we can stay at around the 0.53-0.6 tx/s mark. In this figure, the Besu line appears only as a way to compare the obtained values with constant Besu performance – there is no variation in performance since its number of nodes stays the same. Note that there is, however, a slight increase in SepChain's performance as the number of nodes increases. This is a reflection of increasing the quorum of nodes to accept the correct result: as the quorum size increases, the more likely it is to immediately find the correct result, which can translate into overall performance improvements. To illustrate this, consider the following. Each machine has a range of response times it can give to a certain request: it can be extremely fast if there is no other process running and taking up resources, or it can be extremely slow if there are many concurrent processes or even garbage collection running; provided that the machine is functioning properly, the likelihood of the response time being fast is higher than being slow. Therefore, if there is only one machine, we follow that machine's distribution of request response times. However, the more machines there are, the better chance there is of a sufficient number of them being fast and providing their responses earlier, providing a better performance overall, despite the extra communication overheads of having extra nodes.

This experiment shows that SepChain is able to freely add more resources to its execution layer without negatively affecting the system performance. This translates into having, for instance, better fault-tolerance – since more nodes are added to each cluster – without jeopardizing the throughput.

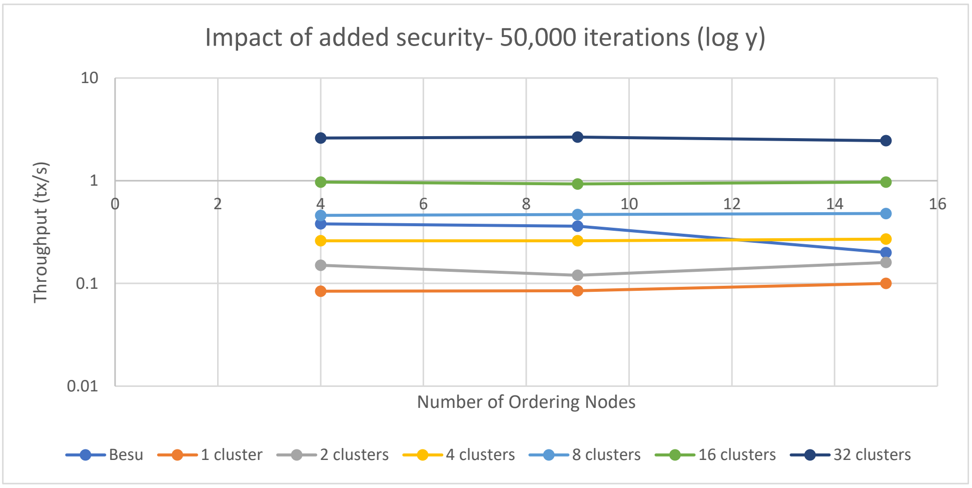

Figure 5.11: Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 50000 iterations. A log scale is applied on the y-axis. The lighter blue line represents Besu's performance. Every other line represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32]

5.10 Impact of Sharding

Finally, the seventh experiment intends to demonstrate SepChain's ability of handling transactions with different 'weights', that is, with different numbers of iterations. We, therefore, consider two types of transactions: light and heavy ones, which have 100 and $50,000$ iterations per smart contract, respectively. To test how this affects the system, we split the client pool into heavy and light clients – each client only sends one type of transaction. We, again, test two different system configurations – 9 and 35 clusters, each with 3 nodes – and, additionally, we test the differences of the system sharding its clusters or not. If the system is not sharded, then different types of clients can share clusters; if the system is sharded, then each cluster can only take one type of request.

Since there are two types of clients, we must also settle on how many clients of each type there are. To this end, we decided on the following percentages of heavy clients: 0, 1, 10, 25, 50, 75 and 100. Note that in the case of sharding, these percentages also correspond to the percentages of heavy clusters available. Conversely, when testing with these data points, we noticed that another parameter was also relevant: the percentage of heavy requests. Therefore the data points presented in the following graphs have the latter percentages as the x-axis, but still maintaining the same initial percentage of clients and clusters. For instance, the third data point in SepChain with 9 clusters has the x-axis value of 6.42. This means that even though there are 25% of heavy clients, only 6.42% of the total number of requests are in fact heavy. The percentages of heavy requests were chosen in order to reduce the tail effect of heavy clients – if, for example, there were constantly 50% of heavy requests, these would finish slower than the total number of light requests. For this reason, more light requests are added to ensure that all the

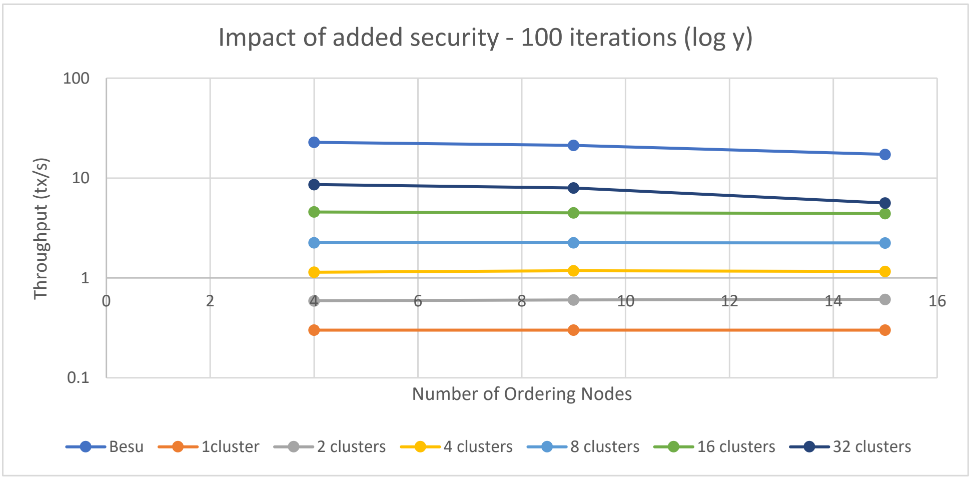

Figure 5.12: Experiment 4: Besu and SepChain throughput values as the number of ordering nodes varies, where both systems use contracts with 100 iterations. A log scale is applied on the y-axis. The lighter blue line represents Besu's performance. Every other line represents SepChain's performances with different settings where the number of clusters varies according to these values: [1, 2, 4, 8, 16, 32].

clients finish at the same time.

All the following figures (5.18, 5.19, 5.20, 5.21) show the data points for different percentages of heavy requests, one with 9 and the other 35 clusters. There are a total of 4 lines per graph: the Besu line, which maintains the same percentage of clients and requests as SepChain does; SepChain with sharding; SepChain without sharding; and the theoretical values for SepChain with sharding. The way the latter values are calculated is better demonstrated through an example. Considering that our performance with 0% of heavy requests is 2.43 tx/s, and with 100% of heavy requests is 0.53 tx/s, the theoretical value for a system with 16.66% of heavy requests where 50% of the clients are heavy is $2.43 * (100 - 16.66)/100 + 0.53 * 16.66/100$.

In both Figure 5.18 and Figure 5.19, we observe that as the percentage of heavy requests increases, the performance in all the systems decreases. We can also observe that the values where there is sharding are better than those where sharding is not being applied. This is a logical outcome since when there is no sharding applied, heavy (i.e., slower) transactions can upset light (i.e., faster) transactions when sharing the same cluster; with sharding, heavy and light transactions are completely separated.

By looking at Figure 5.18, we see that SepChain with 9 clusters is only able to outperform Besu when there are 90% of heavy clients, with a percentage of heavy requests of 75.68. With sharding, SepChain reaches 0.99 tx/s, while without sharding we only manage to get 0.87 tx/s; both these marks are better than Besu's 0.86 tx/s. It was expected that SepChain would improve over Besu's with and without sharding since we are able to increase the degree of independence between light and heavy transactions, especially with sharding enabled. However, it is unexpected that we only outperform Besu when there are 90% heavy clients and clusters – this suggests that this separation is not particularly

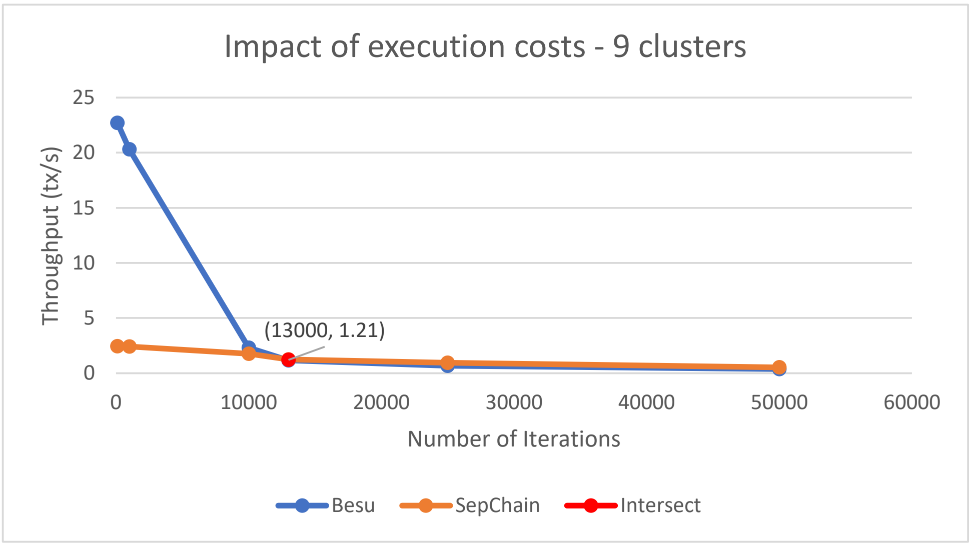

Figure 5.13: Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 9 execution clusters. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems.

effective given our inherent system overheads, especially with a low number of execution clusters.

With 35 clusters (Figure 5.19), the overall performance increases, as observed in previous experiments, which also amplifies the effect of splitting light and heavy requests. Therefore, we are able to outperform Besu at an earlier stage: with 10% of heavy clients and 5% of heavy requests, SepChain reaches 7.66 tx/s without sharding, and 8.99 tx/s with sharding; Besu stays at 6.46 tx/s. Figures 5.20 and 5.21 show the data points more clearly by applying a log scale to the x-axis and the y-axis.

Note that SepChain with sharding – regardless of the configuration – is always below the theoretical line. Since we already eliminated the possible tail effect of heavy requests by over-representing the light requests, there are still two possible reasons that explain this behavior. Firstly, there are the already-mentioned inherent system overheads of implementing SepChain on top of Besu, and these will always affect negatively any results obtained. However, there is still the possibility of having a sub-optimal percentage of heavy requests, given the percentage of clusters and clients. For instance, in the system setting with 35 clusters, with 50% heavy clients and clusters, and with 29% of heavy requests, we get 5.79 tx/s, almost 2 tx/s less than the theoretical value of 7.62 tx/s. Yet, having 29% of heavy requests might not be ideal for this setting.

We estimate the ideal percentage for both 9 and 35 clusters system settings using the following formula: $HeavyClusters/(HeavyClusters * 1 + LightClusters * LightToHeavyRatio) * 100$, where the LightToHeavyRatio corresponds to dividing the throughput of when there are only light requests by the throughput of when there are only heavy requests. These throughput values were calculated in previous experiments. For the 9 cluster setting, the ratio is $2.43/0.53$, while for the 35 cluster setting the ratio is

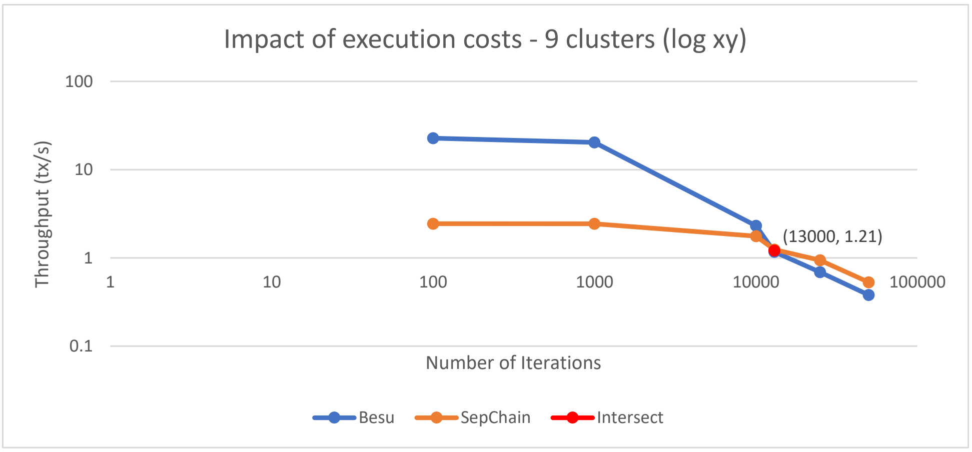

Figure 5.14: Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 9 execution clusters. A log scale is applied on the x-axis and on the y-axis. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems.

$9.55/2.99$. Looking at Table 5.2, we can see that in both cases the percentage is close but not exactly ideal. For instance, with 50% clusters, the system with 9 clusters is under the expected value, that is, there are fewer heavy requests being sent than what is ideal (according to the theoretical prediction). Conversely, the system with 35 clusters is over the expected value – there are more heavy requests than the theoretical prediction. Although percentages of heavy requests are not off by a lot, they can still influence the results obtained and justify why the throughput is below the predicted value.

Heavy Clusters \ Total Clusters	9 Clusters	9 Clusters Theoretical	35 Clusters	35 Clusters Theoretical
1%	1.11%	2.65%	1.58%	0.88%
10%	2.88%	2.65%	5%	3.77%
25%	6.42%	5.87%	11.24%	9.51%
50%	16.66%	21.42%	29%	24.33%
75%	36.81%	43.29%	44.05%	46.73%
90%	75.68%	63.57%	73.97%	70.18%

Table 5.2: Percentage of heavy requests depending on the percentage of heavy clusters with two system settings of 9 and 35 clusters. The theoretical values show the percentage values SepChain expected to achieve.

Overall, this experiment allows us to demonstrate that we are able to show performance improvement if we decide to split the system into different types of requests. This improvement is further increased if we have sharding, which has the particular advantage of isolating different contract types so that each has their desired performance without being delayed by each other.

5.11 Summary

Our experimental evaluation allows us to conclude that our implementation is able to show performance improvement over Besu's implementation in several configurations, as illustrated by Section 5.4. Given

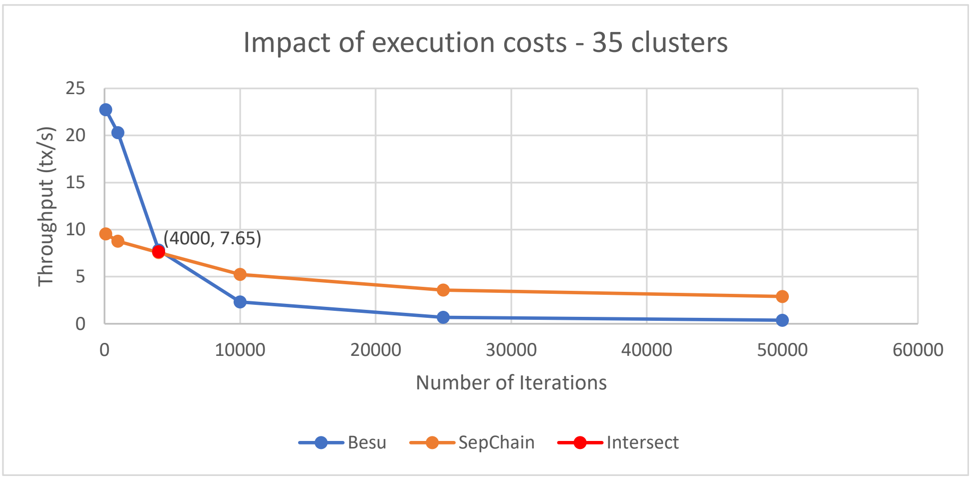

Figure 5.15: Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 35 execution clusters. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems.

that the execution layer is where the main bottleneck is, we are then able to observe, in Section 5.5, at which point this bottleneck shifts to the ordering layer, thus obtaining the limit to how much we can effectively scale out the execution clusters. Section 5.6 then confirms that we are indeed able to scale out the execution clusters and that we can do so while also improving the system security (Section 5.7). Then, in Section 5.8, we determine – for two predefined system configurations – from which gas values (represented as loop iterations) are we able to outperform Besu. Finally, given these results, we are able to further explore SepChain's capabilities. In particular, we can add more resources, for instance, to improve fault-tolerance, without negatively impacting the system, which is something that Besu cannot do (Section 5.9). Additionally, and more importantly, we are able to improve system performance by differentiating between heavy and light smart contracts, and further improve this by sharding the available execution clusters (Section 5.10).

Overall, the evaluation work confirms that our implementation is able to show performance improvement by separating the agreement and execution phases, even without changing the original codebase, and this way we are able to scale out the execution layer by adding more clusters. In addition to this, we are able to take advantage of this architecture by enabling sharding. However, the implementation is far from being perfect: while it shows good performance at a larger system scale (i.e., with higher gas and a higher number of clusters), the system still needs to improve on some implementation overheads, which are noticeable mainly at a smaller system scale, where Besu still has the upper hand.

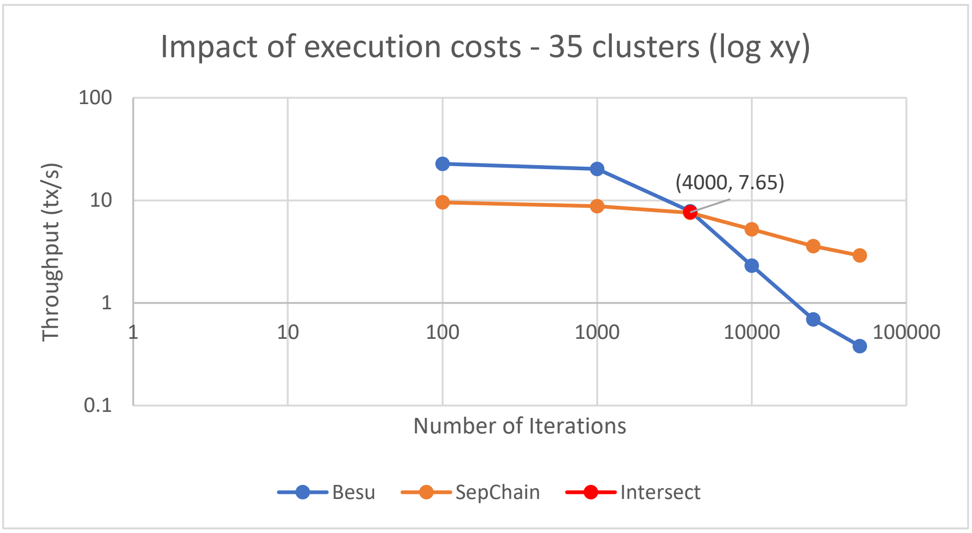

Figure 5.16: Experiment 5: SepChain and Besu throughput values as the number of smart contract iterations (i.e., gas) varies, where the former system has 9 execution clusters. A log scale is applied on the x-axis and on the y-axis. The blue line represents Besu's performance, while the orange represents SepChain's. The red dot represents the intersection of the performances of both systems.

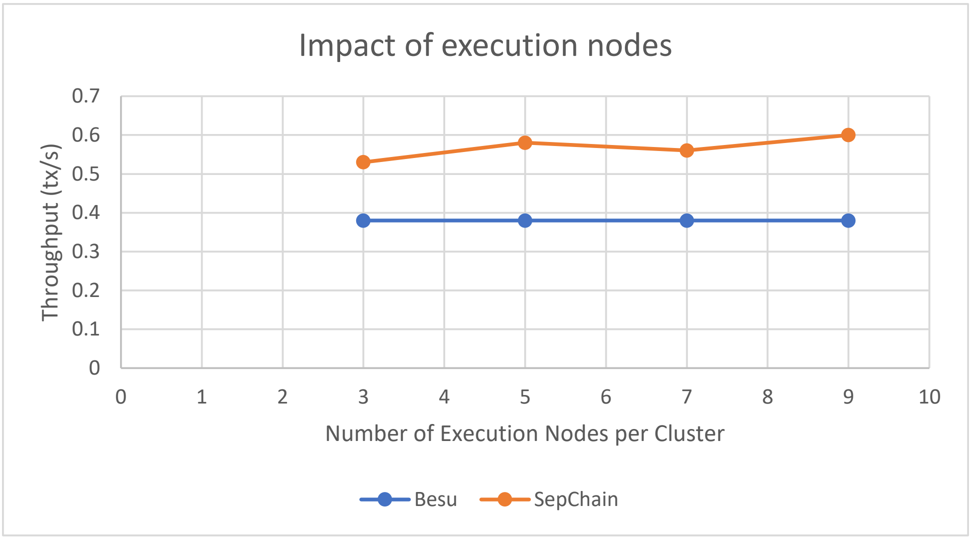

Figure 5.17: Experiment 6: SepChain and Besu throughput values as the number of execution nodes varies, where the former system has 9 execution clusters. The blue line represents Besu's performance, while the orange represents SepChain's.

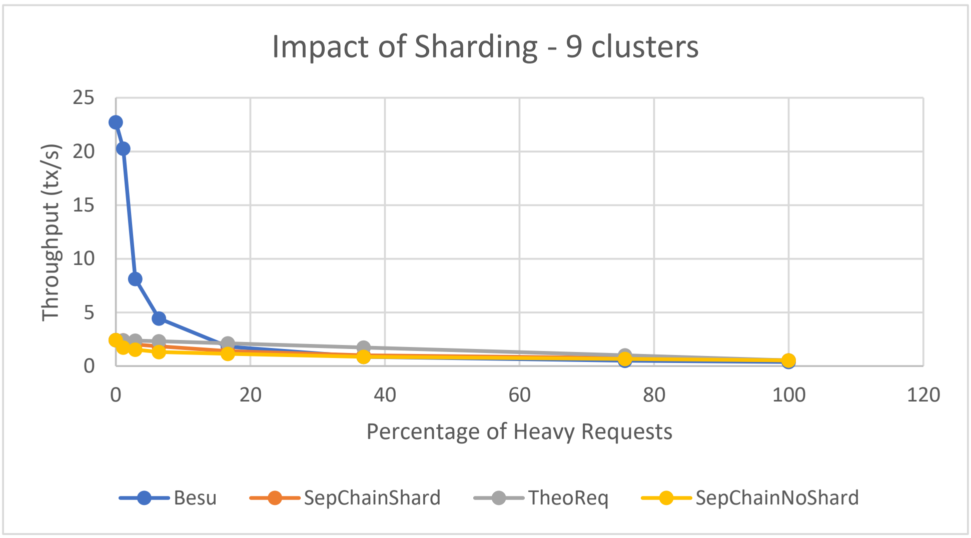

Figure 5.18: Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 9 execution clusters. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads.

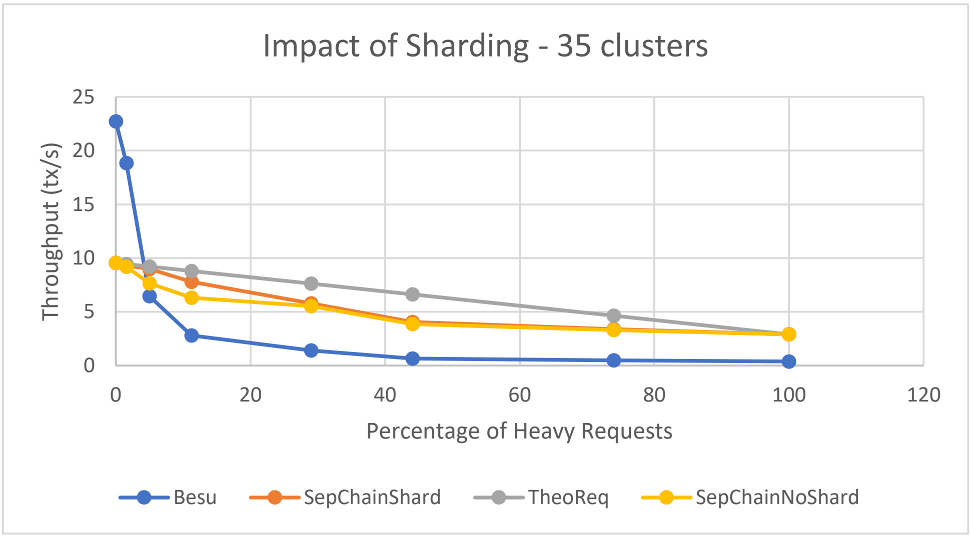

Figure 5.19: Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 35 execution clusters. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads.

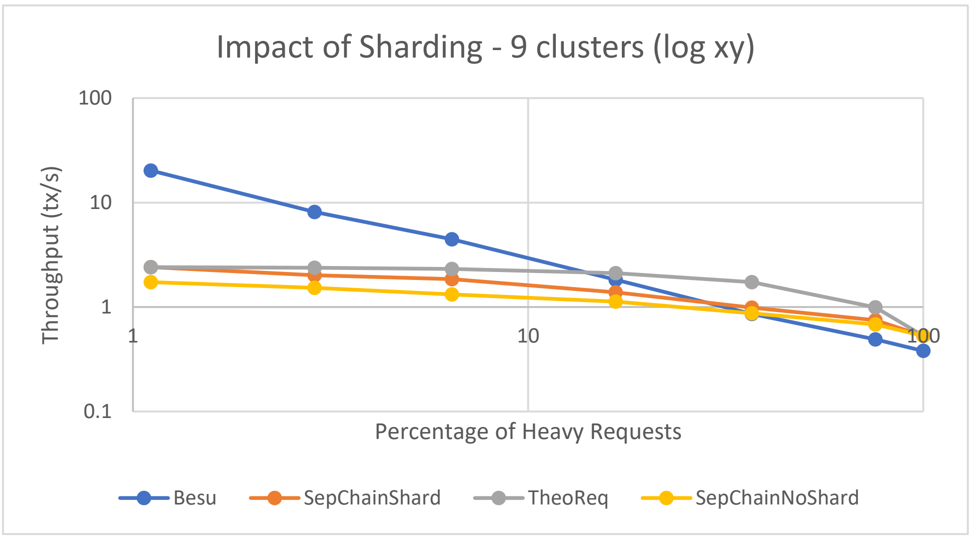

Figure 5.20: Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 9 execution clusters. A log scale is applied on the x-axis and the y-axis. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads.

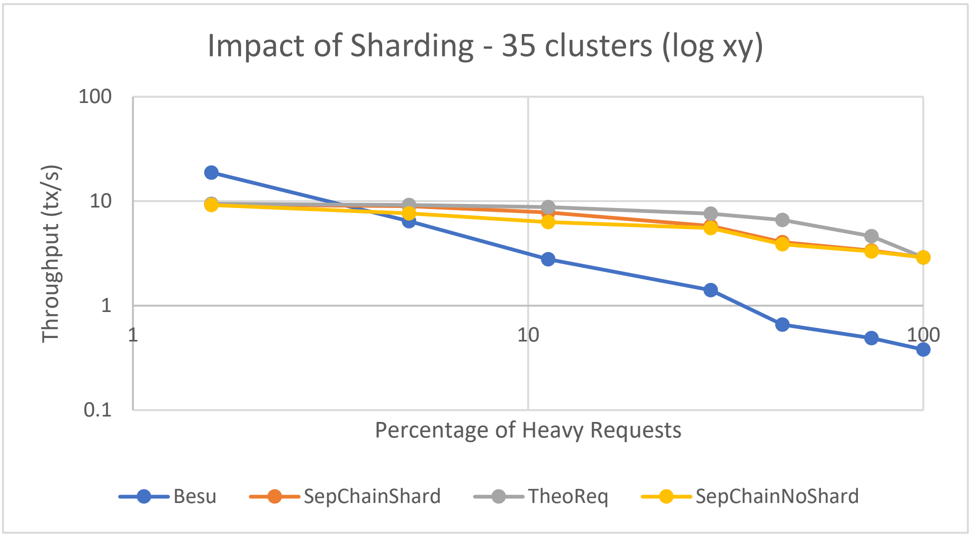

Figure 5.21: Experiment 7: SepChain and Besu throughput values as the percentage of heavy requests varies, where the former system has 35 execution clusters. A log scale is applied on the x-axis and the y-axis. The blue line represents Besu's performance, while the orange and yellow lines represent SepChain's performance with and without sharding enabled, respectively. The grey line represents the theoretical performance value that SepChain should achieve if there were no overheads.

6

Conclusion

Contents

6.1 Conclusion 73
6.2 Current Limitations and Future Work 73
6.3 Closing Remarks 75

This chapter presents the conclusion to this dissertation as well as a set of promising research directions and aspects that can be addressed to improve the current system.

6.1 Conclusion

In this thesis, we argue the need for a different Blockchain architecture to improve scalability. By applying a 20-year old proposal of separating the agreement and execution phases, we can define a clear interface between these layers and fully scale the execution layer to obtain better performance. Moreover, by carefully defining the order of the various events in the execution of a transaction, we can capitalize on the layered model and remove the implicit dependency that the agreement phase has on the execution phase in current architectures.

Through our experimental work, we confirmed that SepChain is able to achieve a better performance than a current monolithic Blockchain system, mainly in larger-scale systems. Conversely, in smaller systems, we confirmed the handicaps of implementing the system on top of Besu, without changing the codebase: an added overhead on both ordering and execution layers.

Overall, we show that creating well-defined interfaces and a comprehensive and logical order of events can lead to immediate improvements over current systems.

6.2 Current Limitations and Future Work

This section details the limitations that our system currently has and what work can be done to improve the system. In addition, we detail several promising directions for future research. The first section focuses on our main implementation decision, how it affects the system, and how changing it could improve the system performance. The following section expands on what must be done for SepChain to reach a production-grade code level. And the last section highlights new theoretical concepts that can be applied to SepChain in order to improve its performance and to develop more capabilities.

6.2.1 Current Implementation

Implementing the system architecture boiled down to an important choice: we could either crack open the existing code base to separate it into two parts, or we could implement on top of Besu and reuse its features without any changes to the code base. We chose the latter as it was the pragmatic way to showcase our architectural concept with minimal changes. However, by choosing the latter, we also inherited many overheads. We now discuss how the other choice of making changes to the code base can eliminate these overheads and natively optimize the system on two fronts: the ordering layer and the execution layer.

Firstly, the log was implemented as a smart contract on top of the already existing Besu system. This introduces unnecessary overheads since, to interact with the log, a full Blockchain transaction processing cycle must be executed. An alternative would be to design a separate structure – similar to an isolated virtual machine – that would function as or hold the log, independently from the ordering Blockchain.

Secondly, and similar to the previous limitation, our system has each execution node as a full single-node Blockchain. This also produces unnecessary overhead since we do not need a full transaction execution cycle either. The solution would be to design a structure that is able to execute smart contracts and return their result.

6.2.2 Production-grade code

While the previous section showcased the fundamental changes that would greatly increase the system performance, this section focuses on how our current code could be elevated to a production-grade level, without necessarily following those major changes (while also keeping in mind that it is fairly common for a research thesis project to not reach a production-grade level). There are, in reality, a number of challenges that need to be addressed to successfully create a commercial, production-ready system, as exemplified by [42, 71]. In the following paragraphs, we exemplify some of the challenges that we would need to address to make our system production-ready.

Firstly, our system was designed as a static system. This means that, for instance, the clients are only able to interact with the clusters defined on the configuration of the system; clients cannot make any dynamic changes to how the system is programmed to work. In the future, clients should be able to freely change which clusters of nodes they want to interact with; there should be a dynamic discovery node system that would find the most suitable execution node (and cluster) for the client to send their commands to.

Secondly, our system currently functions on string manipulation, i.e., the clients send their commands and these are transformed into strings and have their results returned as strings as well. This is an inefficiency that can be optimized by finding a different way to store the information sent by the client.

Finally, there is no garbage collection of the log. That means that the log will keep expanding indefinitely until the system stops. This is highly inefficient and can cause the system to malfunction. There should be a method that ensures that the log commands whose results have already been returned to the client can be safely removed from the log, similar to CORFU [39]'s trim API method.

6.2.3 Theoretical Concepts

Although making changes and optimizing the implementation is a logical way to improve the system, taking a more theoretical approach on how to improve the system also enables us to pinpoint the important concepts to develop in the future, as we now describe in the following paragraphs.

Firstly, we chose a consensus protocol that seemed to fit our needs and make our system as performant as possible. There should be an extra evaluation of which consensus protocol actually benefits our system the best.

Secondly, our focus was mainly on achieving a working implementation and being able to measure the performance accurately, and therefore there was no focus on defining a security and/or fault-tolerance model. In particular, we did not attempt to provide any extra security other than what Besu already provides. That being said, it is important to note that our system does provide important fault-tolerance guarantees; on the ordering layer, we maintain the $3f + 1$ that Besu already provides, while on the execution layer we have $2f + 1$, where f is the number of faults the system can suffer.

Thirdly, our system requires that all the commands are processed in a global total order that is consistent with the order in which they were sent by the clients. The system could be further improved through a log that supports a partial order as an alternative, where the nodes can process the commands in a different order than the one established to improve the overall system performance. Following FuzzyLog's [40] principle of a partially-ordered log could prove very useful to further improve the execution cluster performance: the global log would be sharded into smaller logs, each with partial-ordered commands. Each cluster could access its shard faster than the global log, which would improve contract execution over time.

Finally, our system does not take into account cross-shard interactions and transactions. For instance, a large enough command might have to be split among different clusters to be processed; or one application might have one command that interacts with two different clusters. These would not be possible in our current system and should be made possible in the future.

6.3 Closing Remarks

We contribute to the study of scalability on modern Blockchain systems by presenting SepChain, a novel approach based on a 20-year old concept of separating the agreement and execution phases of a state machine replication model into individual components, and we apply it to a Blockchain setting. Our results show that even with a simplified, non-intrusive implementation we can outperform a monolithic Blockchain system.

We hope that this work paves way for future research in the area, stimulates the discussion on what should be the right focus of the research and development efforts of the Blockchain communities to improve future systems, and challenges the current Blockchain system architecture paradigm as a whole.

Bibliography

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] S. Haber and W. Scott Stornetta, “How to time-stamp a digital document,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 537 LNCS, pp. 437–455, 1991.

[3] W. Dai, “b-money,” 1998. [Online]. Available: http://www.weidai.com/bmoney.txt

[4] A. Back, “Hashcash - A Denial of Service Counter-Measure,” *Http://Www.Hashcash.Org/Papers/Hashcash.Pdf*, no. August, pp. 1–10, 2002.

[5] V. Buterin, “A next-generation smart contract and decentralized application platform,” *Etherum*, no. January, pp. 1–36, 2014. [Online]. Available: https://ethereum.org/en/whitepaper/

[6] Nick Szabo, “Smart Contracts: Building Blocks for Digital Markets,” 1996. [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[7] L. Mertz, “(Block) Chain Reaction: A Blockchain Revolution Sweeps into Health Care, Offering the Possibility for a Much-Needed Data Solution,” *IEEE Pulse*, vol. 9, no. 3, pp. 4–7, 2018.

[8] A. Shahnaz, U. Qamar, and A. Khalid, “Using Blockchain for Electronic Health Records,” *IEEE Access*, vol. 7, pp. 147 782–147 795, 2019.

[9] A. Sharma, S. Bahl, A. K. Bagha, M. Javaid, D. K. Shukla, and A. Haleem, “Blockchain technology and its applications to combat COVID-19 pandemic,” *Research on Biomedical Engineering*, 2020.

[10] Y. Kawamoto and A. Kobayashi, “AI pedigree verification platform using blockchain,” *2020 2nd Conference on Blockchain Research and Applications for Innovative Networks and Services, BRAINS 2020*, pp. 204–205, 2020.

[11] J. D. Harris and B. Waggoner, “Decentralized and collaborative AI on blockchain,” *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019*, no. 2, pp. 368–375, 2019.

[12] K. Salah, M. H. U. Rehman, N. Nizamuddin, and A. Al-Fuqaha, “Blockchain for AI: Review and open research challenges,” *IEEE Access*, vol. 7, pp. 10 127–10 149, 2019.

[13] M. Samaniego and R. Deters, “Blockchain as a Service for IoT,” *Proceedings - 2016 IEEE International Conference on Internet of Things; IEEE Green Computing and Communications; IEEE Cyber, Physical, and Social Computing; IEEE Smart Data, iThings-GreenCom-CPSCom-Smart Data 2016*, no. January 2020, pp. 433–436, 2017.

[14] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, “Blockchain for IoT security and privacy: The case study of a smart home,” *2017 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2017*, pp. 618–623, 2017.

[15] O. Novo, “Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT,” *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.

[16] “Hyperledger Besu.” [Online]. Available: https://wiki.hyperledger.org/display/BESU/Hyperledger+Besu

[17] “Libra White paper.” [Online]. Available: https://www.diem.com/en-us/white-paper/

[18] “Blockchair Ethereum.” [Online]. Available: https://blockchair.com/ethereum/charts/transactions-per-second

[19] “Visa Net Booklet.” [Online]. Available: https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/visa-net-booklet.pdf

[20] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, “Hyperledger fabric,” pp. 1–15, 2018.

[21] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, “FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second,” *ICBC 2019 - IEEE International Conference on Blockchain and Cryptocurrency*, pp. 455–463, 2019.

[22] F. B. Schneider, “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial,” *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990. [Online]. Available: https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf

[23] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, “Separating agreement from execution for byzantine fault tolerant services,” p. 253, 2003. [Online]. Available: https://www.cs.cornell.edu/lorenzo/papers/sosp03.pdf

[24] M. Burrows, “The Chubby lock service for loosely-coupled distributed systems,” *OSDI 2006 - 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 335–350, 2006.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free coordination for internet-scale systems,” *Proceedings of the 2010 USENIX Annual Technical Conference, USENIX ATC 2010*, pp. 145–158, 2019.

[26] M. Castro, B. Liskov *et al.*, “Practical byzantine fault tolerance,” in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[27] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is more consensus in Egalitarian parliaments,” *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pp. 358–372, 2013.

[28] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[29] M. K. Reiter, “The Rampart toolkit for building high-integrity services,” 1995, pp. 99–110. [Online]. Available: http://link.springer.com/10.1007/3-540-60042-6{_}7

[30] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, “SecureRing protocols for securing group communication,” *Proceedings of the Hawaii International Conference on System Sciences*, vol. 3, no. C, pp. 317–326, 1998.

[31] Z. Chao-yang, “Dos attack analysis and study of new measures to prevent,” in *2011 International Conference on Intelligence Science and Information Engineering*, 2011, pp. 426–429.

[32] B. Alpern and F. B. Schneider, “Recognizing safety and liveness,” *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987.

[33] “IBFT documentation.” [Online]. Available: https://github.com/ethereum/EIPs/issues/650

[34] “PoA Network Whitepaper.” [Online]. Available: https://github.com/poanetwork/wiki/wiki/POA-Network-Whitepaper

[35] R. Saltini and D. Hyland-Wood, “IBFT 2.0: A safe and live variation of the IBFT blockchain consensus protocol for eventually synchronous networks,” *CoRR*, vol. abs/1909.10194, 2019. [Online]. Available: http://arxiv.org/abs/1909.10194

[36] "LibraBFT." [Online]. Available: https://developers.diem.com/docs/technical-papers/state-machine-replication-paper/

[37] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus in the lens of blockchain," *arXiv*, pp. 1–23, 2018.

[38] P. L. Aublin, S. B. Mokhtar, and V. Quema, "RBFT: Redundant byzantine fault tolerance," *Proceedings - International Conference on Distributed Computing Systems*, pp. 297–306, 2013.

[39] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, "CORFU: A shared log design for flash clusters," *Proceedings of NSDI 2012: 9th USENIX Symposium on Networked Systems Design and Implementation*, pp. 1–14, 2012. [Online]. Available: http://www.cs.yale.edu/homes/mahesh/papers/corfumain-final.pdf

[40] J. Lockerman, Y. University, J. M. Faleiro, U. Berkeley, J. Kim, U. San Diego, J. M. Faleiro UC Berkeley, J. U. Kim San Diego, S. Sankaran, D. J. Abadi, J. Aspnes, and S. Sen, "The FuzzyLog: A Partially Ordered Shared Log," *USENIX Symposium on Operating Systems Design and Implementation*, no. 18, pp. 357–372, 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/lockerman

[41] D. I. G. Amalarethinam and G. J. J. Mary, "DAGEN - A Tool To Generate Arbitrary Directed Acyclic Graphs Used For Multiprocessor Scheduling," *International Journal of Research and Reviews in Computer Science*, vol. 2, no. 3, 2011.

[42] M. Balakrishnan, J. Flinn, C. Shen, M. Dharamshi, A. Jafri, X. Shi, S. Ghosh, H. Hassan, A. Sagar, R. Shi, J. Liu, F. Gruszczynski, X. Zhang, H. Hoang, A. Yossef, F. Richard, and Y. J. Song, "Virtual Consensus in Delos," *Osdi*, pp. 617–632, 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[43] I. B. Damgård, "Collision Free Hash Functions and Public Key Signature Schemes," in *Advances*, pp. 203–216. [Online]. Available: http://link.springer.com/10.1007/3-540-39118-5{_}19

[44] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Čapkun, "Misbehavior in Bitcoin: A study of double-spending and accountability," *ACM Transactions on Information and System Security*, vol. 18, no. 1, 2015.

[45] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, pp. 1–32, 2014.

[46] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937. [Online]. Available: http://doi.wiley.com/10.1112/plms/s2-42.1.230

[47] Yonatan Sompolinsky and Aviv Zohar, “Secure High-Rate Transaction Processing in Bitcoin,” *AVN Allgemeine Vermessungs-Nachrichten*, vol. 125, no. 6, pp. 180–189, 2018.

[48] A. K. Singh and A. K. Misra, “Analysis of Cryptographically Replay Attacks and Its Mitigation Mechanism,” 2012, pp. 787–794. [Online]. Available: http://link.springer.com/10.1007/978-3-642-27443-5{_}90

[49] M. M. Jalalzai, C. Busch, and G. G. Richard, “Proteus: A scalable BFT consensus protocol for blockchains,” *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019*, pp. 308–313, 2019.

[50] L. Lamport, “Lower bounds for asynchronous consensus,” Tech. Rep. MSR-TR-2004-72, July 2004. [Online]. Available: https://www.microsoft.com/en-us/research/publication/lower-bounds-for-asynchronous-consensus/

[51] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014*, pp. 305–319, 2019.

[52] L. Xu, W. Chen, Z. Li, J. Xu, A. Liu, and L. Zhao, “Solutions for concurrency conflict problem on Hyperledger Fabric,” *World Wide Web*, vol. 24, no. 1, pp. 463–482, 2021.

[53] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, “All about Eve: Execute-verify replication for multi-core servers,” *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, pp. 237–250, 2012.

[54] M. Hearn, “Corda: A distributed ledger,” *Whitepaper*, pp. 1–73, 2019. [Online]. Available: https://www.r3.com/reports/corda-technical-whitepaper/

[55] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, nov 2002. [Online]. Available: https://dl.acm.org/doi/10.1145/571637.571640

[56] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative Byzantine fault tolerance,” *ACM Transactions on Computer Systems*, vol. 27, no. 4, 2009.

[57] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, “Making Byzantine fault tolerant systems tolerate byzantine faults,” *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009*, pp. 153–168, 2009.

[58] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie, “Fault-scalable byzantine fault-tolerant services,” vol. 39, 10 2005, pp. 59–74.

[59] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, “HQ replication: A hybrid quorum protocol for Byzantine fault tolerance,” *OSDI 2006 - 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 177–190, 2006.

[60] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, “Zz and the art of practical bft execution,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 123–138. [Online]. Available: https://doi.org/10.1145/1966445.1966457

[61] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, “UpRight cluster services,” *SOSP’09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, no. January, pp. 277–290, 2009.

[62] M. Kapritsos and F. P. Junqueira, “Scalable agreement: Toward ordering as a service,” *Proceedings of the 6th Workshop on Hot Topics in System Dependability, HotDep 2010*, 2010.

[63] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, “Tango: Distributed data structures over a shared log,” *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pp. 325–340, 2013.

[64] “Hyperledger Besu Github Repository.” [Online]. Available: https://github.com/hyperledger/besu

[65] “Node.js.” [Online]. Available: https://nodejs.org/en/

[66] “Solidity programming language.” [Online]. Available: https://docs.soliditylang.org/en/latest/index.html.

[67] “Web3 API.” [Online]. Available: https://web3js.readthedocs.io/en/v1.4.0/

[68] “WebSocket.” [Online]. Available: https://developer.mozilla.org/pt-BR/docs/Web/API/WebSockets_API

[69] “Data-Store.” [Online]. Available: https://www.npmjs.com/package/data-store

[70] “Oracle VirtualBox.” [Online]. Available: https://www.virtualbox.org

[71] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 398–407. [Online]. Available: https://doi.org/10.1145/1281100.1281103