SepChain: Separating Agreement from Execution to Improve the Scalability of Blockchains

Diogo Fernandes
Instituto Superior Técnico
Lisbon, Portugal
diogo.f.fernandes@tecnico.ulisboa.pt

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.

1 Introduction

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., Proof of Work, IBFT) will define which block will in fact be added to the chain.

Additionally, many current Blockchain systems [2, 4, 12, 15] are based on a monolithic design. As such, a Blockchain node is a single machine that conflates the logic for the previously mentioned steps. 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; similarly, it cannot be scaled out – since this would only replicate the existing problems. In addition to these faults, 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.

In this thesis, we aim to demonstrate the importance of carefully layering state machine replication systems [18], 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 [20]. Guided by these decades-old principles, we aim to demonstrate that it is possible to improve Blockchain scalability by decoupling the monolithic architecture of Hyperledger Besu [2], an Ethereum client, to introduce a separation between the agreement and execution phases. To implement these principles, we built SepChain. 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. Finally, SepChain shows these improvements with minimal changes: by building on top of the Besu code base rather than tearing it apart.

2 The Blockchain Landscape

This section provides an overview of the space of Blockchain architectures, by describing its most relevant data points.

2.1 Ethereum

In Ethereum, the ledger maintains the global state of the Ethereum Network, which is composed of a set of accounts, where each account is identified by a unique address and stores a certain amount of Ether (Ethereum's currency). An Ethereum account can be used to send and receive Ether to/from other accounts, but can also be a smart contract: pieces of reusable code that provided a certain input command, they perform an action and output a result.

The state of the Blockchain (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 sender's message and signature, the recipient, and the Ether to be sent.

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 an operation's computational effort. 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.

The world state is stored in the state machine, also called the Ethereum Virtual Machine (EVM), which also defines the state transition function: the group of selected transactions 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 select by the node and subtracts the gas used. When the iterator halts (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, 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). 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) all the system nodes verify this new block (the correctness of its timestamp, block number, among other parameters), and incorporate its state changes to their local state. 3) the block is added to the chain and becomes visible, provided that a sufficient number of nodes agree on adding the block in that position.

2.2 Hyperledger Fabric

The previous description 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 [9] was one of the first systems to think through this order, by proposing an execute-order-validate architecture.

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 the 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 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 block, establishing consensus using a deterministic protocol such as RAFT [16]. The block is then sent to a new group of peers (the committer peers) for the validation phase.

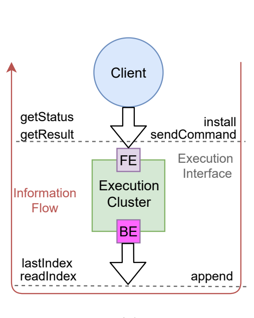
Third, in the validation phase, the peers verify if the endorsement set satisfies the endorsement policy; then, they verify each transaction sequentially, ensuring that the read set corresponds to the same write set. If in any of these steps the validation fails, the transaction is discarded. Finally,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

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 or due to concurrent execution of requests [19]. Therefore, the fact that transactions are first executed can lead to the waste of resources.

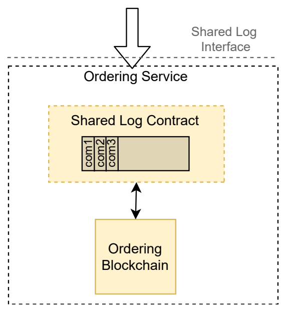
2.3 CORFU

In trying to search for a more effective layering based on better abstractions, we sought inspiration in a system called CORFU. 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.

This abstraction defines the interface between the concurrent clients and the storage layer. Thus, for the clients to interact with the storage, there are two main methods: *read* and *append*. The former reads stored data from a given index, and the latter appends new data to storage at a given index. This interface provides a clear



(a)



(b)

Figure 1: Proposed design. This figure only represents one client and one execution cluster for simplicity.

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.

Despite serving a different purpose than Blockchains, CORFU defines a fundamental concept that can be applied in other settings: a layering based on simple, precise and well-designed abstractions, with a lower layer allowing for reaching a consensus on the order of the execution of requests, and the upper layer implementing the application-specific logic.

3 A Proof-of-Concept Architecture

As seen before, modern Blockchains such as Ethereum [12] intertwine consensus and application execution into a single interdependent blob, hindering performance. Fortunately, there is a ready-made architectural solution to this dilemma: separate agreement from execution [13, 14, 20], which has already been applied in systems like Delos [10]. We now detail how SepChain applies these principles to a Blockchain setting.

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.

3.1 Design

SepChain's design defines three layers: the clients, the ordering service cluster, and multiple execution clusters.

3.1.1 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 through an interface with four methods.First, clients can install their application on the execution cluster, through the *install* method, which provides the execution cluster with the name of the contract to install. This cluster then sends a transaction that creates a smart contract containing the client application on the designated execution cluster nodes. Second, as with the *install* method, the client calls their execution cluster to send a transaction with their command to be executed on their contract with *sendCommand*. Third, they can query their execution cluster for the status of the command, to see if it has already been executed or if it is still pending with (*getStatus*). If it has been executed, it returns the hash of the result, which the client considers trustworthy after receiving that same hash from a majority of execution nodes in that cluster. Fourth, they can query the contract (after receiving the hash), to get the return value of the command they sent (*getResult*).

3.1.2 Ordering. The ordering service is a Blockchain with multiple ordering nodes that provide the global order of commands. When commands are received, each node batches them into a block according to a local order. 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 [11] 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 defines three methods that the execution clusters call remotely: *append, lastIndex, readIndex*, which are defined next.

3.1.3 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*. The first method causes the clients' commands to be added to the log's map structure sequentially, on the next position available. The second method and third are always used together. The former 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.

3.2 Order of Events

The flow of the execution 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.

4 Implementation

As the codebase for an Ethereum-based implementation, we decided to use Hyperledger Besu [2], 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 [17]), and it has the option to 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 unique name: "client", "exec" or "order", followed by a unique integer indicating the instance number of that type of virtual machine. SepChain recognizes this number as the machine's id. A Besu node is started by a bash process that executes the Besu binary which is pre-installed from the source repository [3]. When executing the binary, individual aspects of each node are defined through flags, such as the P2P and RPC host and port addresses. 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 – 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, defining the nodes which participate in the network.

While Bash is essential to execute the individual node processes, the code that supports our architecture was written in JavaScript with Node.js [5]. Additionally, the smart contract themselves were written in Solidity [6], a smart contract programming languages, which is then compiled to the EVM bytecode. To support our system architecture, there are six main JavaScript modules: the client, two execution front-ends, two execution back-ends, and a general configuration module utilized by all the other ones. Finally, the logic for sending transactions uses Besu's unaltered implementation of sending a signed transaction through HTTP-RPC (using the web3 library [7]). Messages sent in transactions are always in a string format and are processed upon being received.

4.1 Configuration Module

This 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 method, which receives the encoded transaction object hex as an argument and sends this signed transaction to the indicated address.

4.2 Client

. The technology chosen for the clients to interact with the execution nodes was the WebSocket [8], 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 and command name), the client id (from the client's machine name), the request id (incremented per request), and the method name (e.g., "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. The *send* method follows the same principle, but sends the method name "send" instead.

Each client uses *getStatus* to request the status of the command execution until a majority of the replies contain the same hash. This method iterates through a list of the predefined cluster addresses and establishes a different, asynchronous web socket connection with each of themFor 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 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. Moreover, if the value string is either "Pending" or "Deprecated" – meaning the command is not yet execute or that the command is not up to date, respectively – the client calls *getStatus* again. Otherwise, the hash of the command result is received and *getResult* is called. The latter command chooses one of the execution node addresses from the achieved majority and queries it for the command result in a new web socket connection.

4.3 Ordering

To implement the ordering structure, we simply reuse the functionality of Besu; the implementation derives from the fact that each ordering Blockchain node has the log contract installedThis 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).

4.4 Execution

The execution implementation is split across a total of four JavaScript modules: two front-end modules and two back-end modules.

The first front-end program deals with relaying client requests to the ordering chain (either install requests or send command requests).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. Once the string is formed, the 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.

Then, the first back-end module deals with checking if a new contract has been added to the log. 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. 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 [1] map is set with the number of strings (i.e., the number of client commands) read as key, and the string itself as the value.

The second back-end file deals with the contract execution part. 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. 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.

The second front-end program deals with getting the status and getting the results back to the clients, following their requests. The program listens on their respective web sockets for messages (on a different port than the first front-end program). If the message received 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.

5 Evaluation

In this section, 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. In both systems, the consensus algorithm utilized was IBFT 2.0 [17]

5.1 Experimental Methodology

To compare both systems, we measured the throughput (in transactions per second) in each experiment we conducted.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, $Throughput = (requestsPerClient * clients)/totalTime$.

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 gas consumed by the smart contract (represented in loop iterations). Additionally, the clients interact with the system by sending requests sequentially on a closed-loop.

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, each with 3 execution nodes, which demonstrate two different levels of SepChain's performance.

5.2 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.

Looking closely at Figure 2, 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.

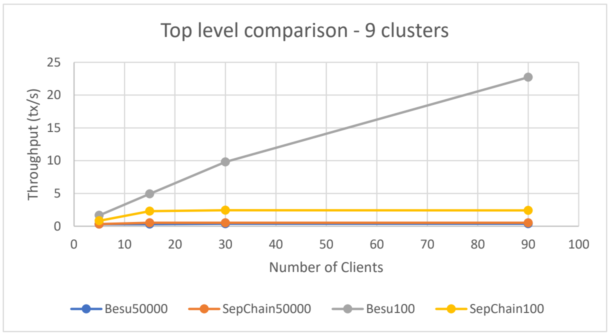


Figure 2: Throughput values as the the number of clients varies using 9 clusters

Now observing Figure 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. 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.

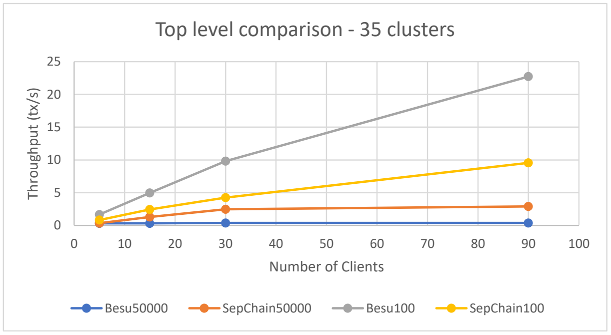


Figure 3: Throughput values as the the number of clients varies, using 35 clusters

Finally, Table 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 5.4, 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 / 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 1: Theoretical throughput values as the the number of clients varies, using 75 clusters

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 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. 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.

5.3 Impact of ordering nodes

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 and these send to the ordering Blockchain, but, when 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 4 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.

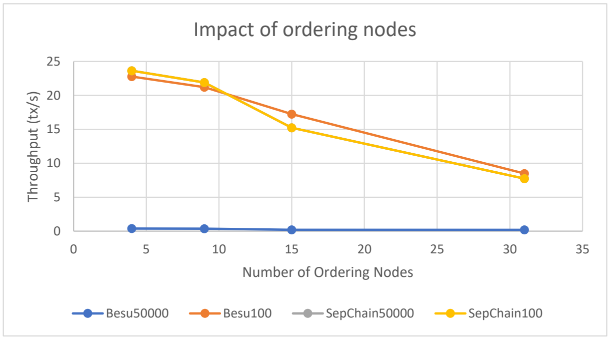


Figure 4: Throughput values as the the number of (ordering) nodes varies

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.

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.

5.4 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, 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 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.

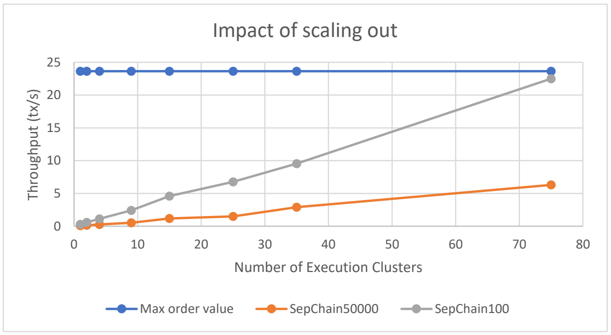


Figure 5: Throughput values as the the number of execution clusters varies

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.

5.5 Impact of added security

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.

For the experiment with 50, 000 iterations (figure 6), 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 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 7), 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.

Besu is clearly impacted by the increase of ordering nodes: 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 is also affected by the increase on the execution cluster number: more execution clusters translate to more communication overhead when interacting with the ordering log.

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.

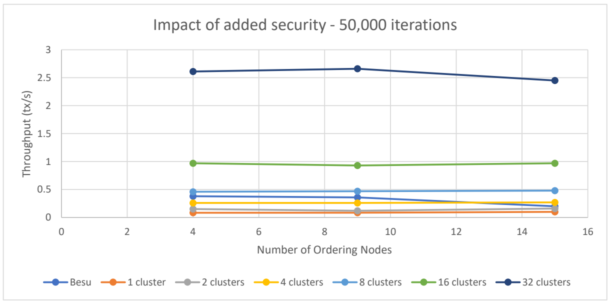


Figure 6: Throughput values as the number of ordering nodes varies, where both systems use contracts with 50000 iterations.

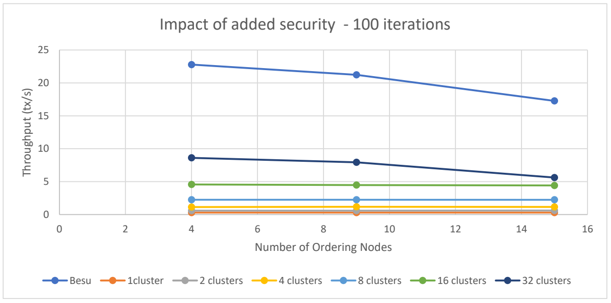


Figure 7: Throughput values as the number of ordering nodes varies, where both systems use contracts with 100 iterations.

5.6 Impact of execution costs

Sections 5.4 and 5.5 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.

Figure 8 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 9 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 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

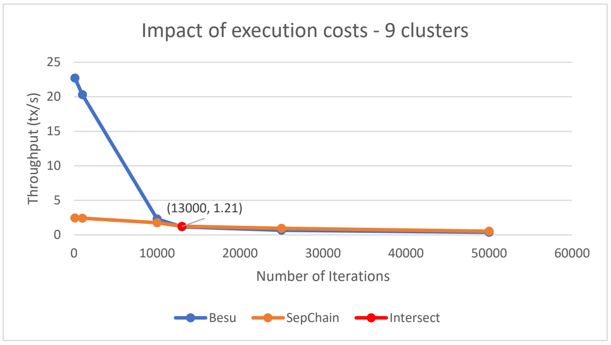


Figure 8: Throughput values as the number of smart contract iterations varies, with 9 clusters.

the intersection is at an earlier point since Besu's performance stays the same, yet ours improves with the addition of more clusters.

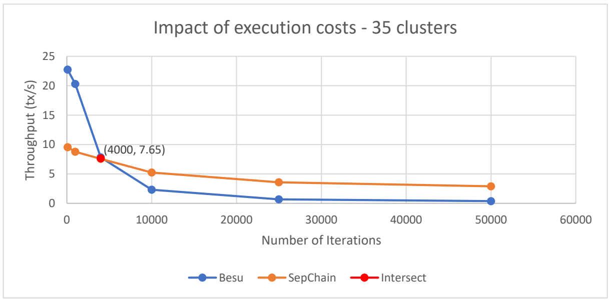


Figure 9: Throughput values as the number of smart contract iterations varies, with 35 execution clusters.

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 **??** and **??**, 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.

5.7 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 number of 9 clusters with the aim of discovering how it impacts the system.

In our case, by looking at figure 10, 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.

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.

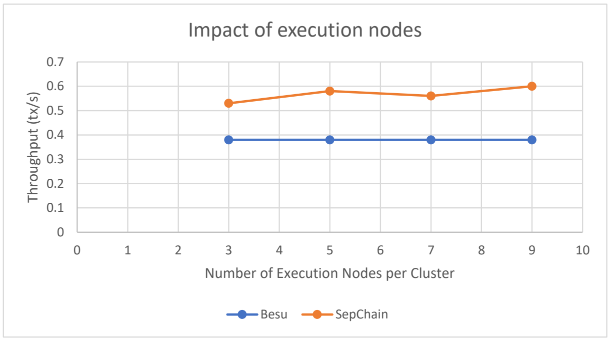


Figure 10: Throughput values as the number of execution nodes varies, with 9 clusters.

5.8 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 – 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 there 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. We must also consider another parameter: 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 following figures (11, 12) 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, SepChain with sharding; SepChain without sharding; and the theoretical values for SepChain with sharding. 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 11 and Figure 12, 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 11, 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 effective given our inherent system overheads, especially with a low number of execution clusters.

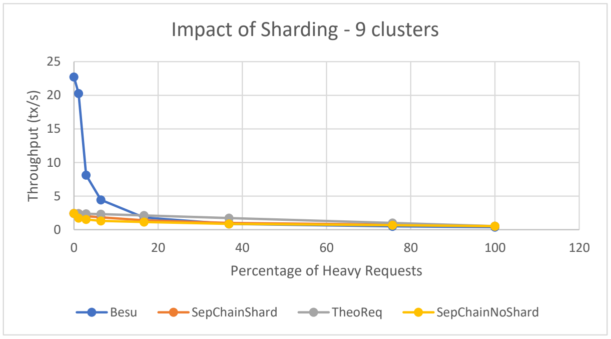


Figure 11: Throughput values as the percentage of heavy requests varies, with 9 clusters.

With 35 clusters (Figure 12), 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.

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

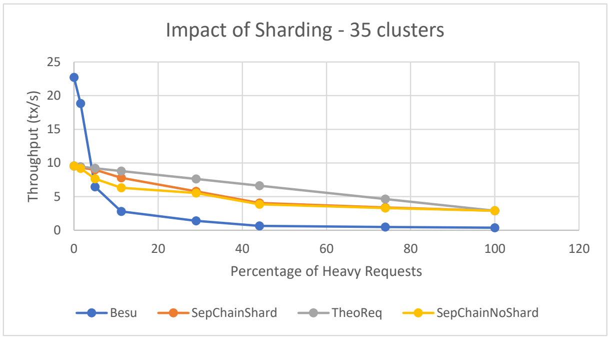


Figure 12: Throughput values as the percentage of heavy requests varies, with 35 clusters.

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 also 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, which might have skewed the results.

All in all, 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.

5.9 Summary

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.

6 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 simultaneously lead to improvements in current systems, stimulate the discussion on how to improve future systems, and challenge the blockchain systems paradigm as a whole.

References

[1] [n.d.]. Data-Store. ([n. d.]). https://www.npmjs.com/package/data-store
[2] [n.d.]. Hyperledger Besu. ([n. d.]). https://wiki.hyperledger.org/display/BESU/Hyperledger+Besu
[3] [n.d.]. Hyperledger Besu Github Repository. https://github.com/hyperledger/besu
[4] [n.d.]. Libra White paper. https://www.diem.com/en-us/white-paper/
[5] [n.d.]. Node.js. https://nodejs.org/en/
[6] [n.d.]. Solidity programming language. ([n. d.]). https://docs.soliditylang.org/en/latest/index.html.
[7] [n.d.]. Web3 API. https://web3js.readthedocs.io/en/v1.4.0/
[8] [n.d.]. WebSocket. https://developer.mozilla.org/pt-BR/docs/Web/API/WebSockets_API
[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger fabric. (2018), 1–15. https://doi.org/10.1145/3190508.3190538
[10] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. *Osdi* (2020), 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan
[11] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A shared log design for flash clusters. *Proceedings of NSDI 2012: 9th USENIX Symposium on Networked Systems Design and Implementation* (2012), 1–14. http://www.cs.yale.edu/homes/mahesh/papers/corfumain-final.pdf
[12] Vitalik Buterin. 2014. A next-generation smart contract and decentralized application platform. *Etherum* January (2014), 1–36. https://ethereum.org/en/whitepaper/
[13] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. UpRight cluster services. *SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* January (2009), 277–290. https://doi.org/10.1145/1629575.1629602
[14] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-verify replication for multi-core servers. *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012* (2012), 237–250.
[15] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008). https://bitcoin.org/bitcoin.pdf
[16] Diego Ongaro and John Ousterhout. 2019. In search of an understandable consensus algorithm. *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014* (2019), 305–319.
[17] Roberto Saltini and David Hyland-Wood. 2019. IBFT 2.0: A Safe and Live Variation of the IBFT Blockchain Consensus Protocol for Eventually Synchronous Networks. *CoRR* abs/1909.10194 (2019). arXiv:1909.10194 http://arxiv.org/abs/1909.10194
[18] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319. https://doi.org/10.1145/98163.98167
[19] Lu Xu, Wei Chen, Zhixu Li, Jiajie Xu, An Liu, and Lei Zhao. 2021. Solutions for concurrency conflict problem on Hyperledger Fabric. *World Wide Web* 24, 1 (2021), 463–482. https://doi.org/10.1007/s11280-020-00851-6
[20] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating agreement from execution for byzantine fault tolerant services. (2003), 253. https://doi.org/10.1145/945469.945470