DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Redesigning the ordering layer of Hyperledger Fabric with Trusted Execution Environments

Nicolas Strobl

DEPARTMENT OF INFORMATICS

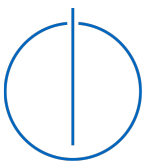
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

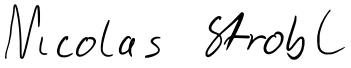
Redesigning the ordering layer of Hyperledger Fabric with Trusted Execution Environments

Umgestalten des Ordnungssystems von Hyperledger Fabric mit Trusted Execution Environments

Author:	Nicolas Strobl
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dimitra Giantsidi
Submission Date:	15.09.2022



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.



Munich, 09.09.2022 Nicolas Strobl

Acknowledgments

First of all, I want to express my thanks to my supervisor, Prof. Dr.-Ing. Pramod Bhatotia, for allowing me to explore a broad topic and to find new and creative approaches.

I especially want to thank my advisor Dimitra Giantsidi for giving me guidance through the topic and for supplying me with many exciting directions and ideas for the thesis.

Lastly, I want to thank my family and friends for all the much needed moral support in particular during the writing process.

Abstract

Blockchains are designed to generate trust between usually untrusted parties. Therefore, they are commonly used for establishing ownership in the case of cryptocurrencies, NFTs, domains, etc. or for increasing transparency for example for the supply chain of mined minerals.
Hyperledger Fabric is specialized for corporate use cases and prioritizes performance and parallelization. However, it also requires at least one centrally trusted system as part of the ordering layer which is responsible for forming the trusted log (chain of blocks).
In the pursuit of making the ordering layer trustworthy, current solutions use a distributed ordering algorithm, based on a protocol that provides similar guarantees (namely Byzantine Fault Tolerance or BFT) to other blockchains. Mainly, these algorithms strive to achieve consensus among untrusted parties by using common BFT consensus protocols. These protocols, however, are complex and their multi-phase approach leads to limited scalability and performance.
Fortunately, in recent years Trusted Executions Environments have become more common and more powerful. They create an opportunity to establish trust to a central ordering system with less communication overhead and similar guarantees as BFT systems.
We designed Orceval, a fast, parallel and scalable ordering layer for Hyperledger Fabric. It combines the ordering and the validation phase by concurrently accepting and optimistically validating transactions on different nodes and then centrally collecting and checking them to ensure correctness.
We show that Orceval has comparable throughput to the popular BFT-SMaRt even though BFT-SMaRt does not perform validation.

Contents

1 Introduction

Blockchains have gained a lot of attention in recent years. They are regarded as one of the core technologies to enable the upcoming Web 3.0 [3]. And while they are still mostly associated with cryptocurrencies like Bitcoin[32] or Ethereum[55], other applications are on the rise. Over the past years, for example Non-Fungible Tokens (NFTs)[6, 53] have increased in popularity to prove ownership of anything from digital art[17] to domain names[52]. Blockchain as a technology has proven itself as an excellent mediator between untrusted parties. Because of this unique property, specialized systems like Hyperledger Fabric[5] have unlocked many applications in the enterprise sector. Fabric's three-phase system (Execute, Order and Validate) increases throughput by executing multiple transactions in parallel and ordering them later while making key assumptions that make Fabric uniquely suited for enterprise blockchains. It is used for instance in an online marketplace for used aircraft[20], processing healthcare claims[14], establishing and managing credentials in the education sector[39], and creating transparency and traceability in the mining industry[15].
Meanwhile, Trusted Execution Environments have been steadily built into more CPUs by major manufacturers. Some of the most popular Trusted Execution Environments are Intel SGX[22] and AMD SEV[4] which were introduced within the last decade. Using isolation and encryption, they can create trust by guaranteeing that the expected code is executed without any interference even from the host system. Along with many other applications for Trusted Execution Environments like digital rights management[29] and biometric authentication[34], cloud computing is growing considerably year on year[43] and so has the interest and research in utilizing them[7, 9, 10, 45, 46, 48].
At their core, both technologies are designed to provide guarantees for running systems in untrusted environments but their combined applications as of now are limited. Research has mostly focused on either specific use cases[8, 26, 27, 38, 42] or on executing transactions or DApps off-chain[30, 54]. Therefore, we aim to introduce an ordering layer for the general purpose Hyperledger Fabric using Trusted Execution Environments. We have the following design goals:

- The ordering layer should have equivalent guarantees to Byzantine Fault Tolerant ordering layers. We want to use Trusted Execution Environments to achieve Byzantine Fault Tolerance without depending on classical consensus protocols.

- The protocol will be specifically designed for ordering between trusted machines without depending on consensus protocols.
- Hyperledger Fabric should be modified by combining the Ordering and Validation phases. Both will be handled centrally by the ordering layer without compromises to the correctness or requiring clients or peers to blindly trust it.
- We strive to achieve comparable throughput to other Byzantine Fault Tolerant ordering layers.

Our contribution is the design and evaluation of Orceval, a parallel and scalable ordering layer for Hyperledger Fabric. In this thesis, we start by explaining the background our work is based upon (§ 2). After that, we discuss our proposed changes to the structure of Hyperledger Fabric as well as the design of Orceval (§ 3). Further, after we provide an overview of the implementation of our protocol (§ 4), we compare it to BFT-SMaRt[41] and examine its performance to determine bottlenecks (§ 5). Lastly, we will discuss future work both on our specific design as well as potential other designs of ordering layers using Trusted Execution Environments (§ 6).

2 Background

2.1 Blockchain

Blockchains are a concept designed to store data in an immutable way. A blockchain is a list of blocks each containing a hash of the previous. Consequently, a block cannot be modified without editing all subsequent blocks, making it infeasible on a continuously growing list. This principle is commonly used to store a list of transactions that can only be read from and appended to. Therefore, blockchains are uniquely capable of generating trust as any data stored in them can be considered permanently committed. While the first proposal dates back to 1982 [37], the concept gained popularity after the introduction of Bitcoin[32] in 2008. Bitcoin is designed specifically for to prove and transfer ownership of digital currency but the later developed Ethereum blockchain[55] enables more use cases by being able to execute code in the form of smart contracts. Both Bitcoin and Ethereum are designed to be public, meaning they can be read and maintained by anyone. Therefore, these blockchains are generally not intended for use cases where the stored data is private or should be maintained by a restricted set of parties.

2.2 Hyperledger Fabric

Hyperledger Fabric is a blockchain system specifically designed to be used in the context of multiple untrusted organizations [5]. Firstly in fabric, peers are grouped into organizations. It is assumed that a peer from one organization trusts all peers which are part of the same organization while not trusting any other peers. This contrasts with most other blockchain systems like the previously mentioned Ethereum or Bitcoin which assume no trust between any peers. Secondly, because any peer has to be assigned to an organization, the blockchain can not be participated in or even accessed by the public. Consequently, Hyperledger Fabric is called a "permissioned" blockchain as you need to have permission to read from or append to the blockchain. Lastly, it uses an execute-order-validate paradigm as opposed to the more common order-execute paradigm to increase performance by incorporating parallelization. In the following, we provide an overview of Hyperledger Fabric.

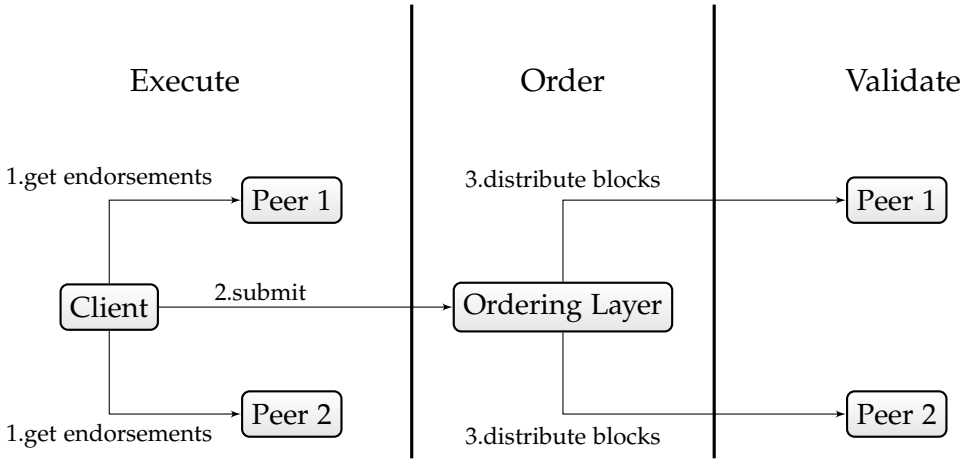


Figure 2.1: Overview of Hyperledger Fabric

The system has three roles - namely client, peer, and orderer - and a three-phase protocol consisting of Execute, Order, and Validate as illustrated by Figure 2.1. In the first phase, a client sends a transaction to a certain number of endorsing peers following a predetermined endorsement policy. These peers simulate the transaction and sign their outcome using private keys. After receiving the endorsements, the client sends the transaction and the signatures to the ordering layer. After this phase, a transaction in Hyperledger Fabric is represented by a read and write set. Reads and writes are themselves tuples of a key, the version of the key, and the value. Neither in the Order nor the Validate Phase is a transaction executed again.
In the next phase, the ordering layer brings all of the potentially conflicting transactions in an order and produces a block. The ordering layer in Hyperledger Fabric is pluggable and can be chosen to fit the needs of the given environment. Generally speaking, the ordering layer is unaware of the content of the transactions and it has to be trusted, either by providing Byzantine Fault Tolerance in its underlying system or simply by policy (for example because it is operated by a trusted third party).
The block from the ordering layer is then passed back to the peers. Each of them checks all of the containing transactions for incorrect endorsements or potential read-write conflicts which could have occurred due to parallelization or the context-unaware ordering. Transactions are then either marked valid or invalid.
Overall, in Hyperledger's Fabric transactions are first executed and endorsed concurrently and after a centralized ordering layer groups them into blocks, they are validated to resolve any potential read-write conflicts.

2.3 Trusted Execution Environments

Trusted Execution Environments (TEE) are built into CPUs. They allow for the isolated execution of code with the guarantee that neither the code nor the accompanying registers or memory can be modified by an unauthorized party which can be the host machine itself. One of the important mechanisms regarding this thesis is remote attestation. It provides cryptographic verification proving that the intended code is executed.
Trusted Execution Environments were first defined in 2009[2]. The technology was then adopted by major CPU manufacturers like Intel with Intel SGX[22] and AMD with SEV[4] in the following years. Additionally, it was implemented for many modern versions of ARMs TrustZone[44, 49]. Lately, the adoption of Trusted Execution Environments has greatly increased in part due to the rising demand for cloud computing [43].

2.4 Byzantine Fault Tolerance

A Byzantine Fault has a broad definition. A Byzantine Fault Tolerant (abbreviated as BFT) system has to be able to handle machines not acting according to their specification. This can mean that a given machine appears crashed to some machines but not to all or it can send messages not adhering to its specified protocols. Overall, a BFT system must be capable of making progress despite a certain number of faulty or even malicious machines. BFT systems usually define a specific ratio of machines or nodes that can be faulty. For example, the common BFT consensus algorithm[13, 25] requires $3f + 1$ nodes to handle up to f faulty machines while most common blockchain systems like Bitcoin[32] and Ethereum[55] require the faulty nodes to either have less than 50% of processing power in the case of Proof-of-Work systems[32] or less than 50% of the currency in Proof-of-Stake systems[36]. For Hyperledger Fabric the ratio depends on the endorsement policy[5] and can therefore be changed for different use cases. The most common endorsement policy is that a transaction has to be endorsed by a peer from either all affected or all organizations. Therefore, if any of these peers does not endorse a transaction, it is marked invalid in the third and last phase.

2.5 Byzantine Fault Tolerant ordering layers

BFT ordering layers for Hyperledger Fabric provide Byzantine Fault Tolerance. Most BFT ordering layers like BFT-SMaRt[41] or HoneyLedgerBFT[51] depend on BFT consensus protocols[31, 40] and therefore retain the same assumptions and properties.

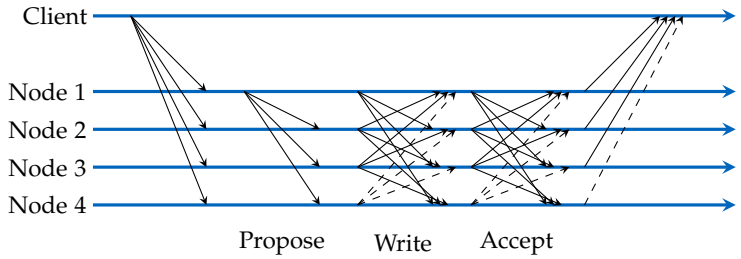


Figure 2.2: BFT-SMaRt message pattern with one client and four nodes

In the following, we take a look at BFT-SMaRt. While it requires the same $3f + 1$ nodes to handle up to f faulty machines as regular BFT consensus, it optimizes performance by utilizing batching and a leader-based approach. The consensus protocol consists of three phases: Propose, Write, and Accept[11] which are illustrated in Figure 2.2.

The Propose message is sent by the leader to all other nodes and contains the newest batch. Then the Write and Accept messages are sent from each node to all other nodes. As opposed to the Propose message, the Write and Accept messages do not transmit the entire batch but only a cryptographic hash to save on bandwidth. The four-node configuration depicted in Figure 2.2 is able to tolerate one malicious node and is therefore capable of making progress even if for instance node 4 was unresponsive or not following protocol.

3 Design

In this chapter, we introduce Orceval. The following sections present the proposed changes to Hyperledger Fabric, an overview of our protocol and its two phases, the assumptions we made, the communication, the workflow of the two phases, and the properties of the proposed design.

3.1 Modifications to Fabric

In this section, we describe how our system changes the workflow of Hyperledger Fabric. While we retain the roles of client, peer, and orderer from Hyperledger Fabric, the ordering, and the verification phase are combined and handled by the ordering layer as illustrated in Figure 3.1.

Endorsement peers: The endorsement peers do not perform validation but only execute the simulations and generate endorsements. When the ordering layer finishes a block, the endorsement peers directly apply the valid transactions to their local key values. Consequently, the total workload on the endorsement peers is lower, allowing for either higher throughput for the same number of peers or a lower number of peers to achieve similar throughput.

Ordering layer: We utilize Trusted Execution Environments to generate trust in the ordering layer. Therefore, its structure can be vastly different from common ordering layers while still offering similar guarantees as BFT systems. Thus, we propose a design that expands its responsibilities to centrally validate transactions in addition to ordering them.

However, our modifications present a drawback by introducing a computational bound to the usually mostly consensus and thus communication bounded ordering layer. The additional computations for validation lead to increased latency and consequently pose a potential bottleneck. On the other hand, this approach reduces the overall workload for the entire blockchain system as a transaction is centrally validated once as opposed to being validated by every endorsement peer individually.

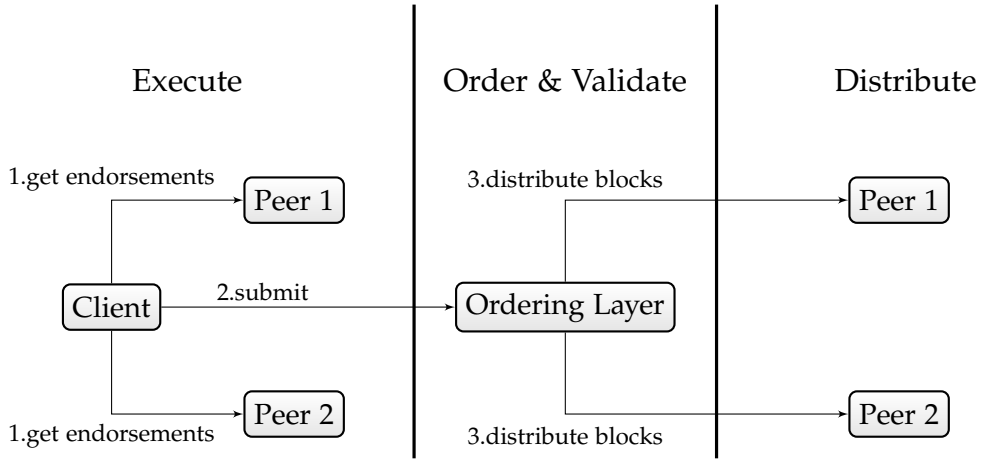


Figure 3.1: Overview of proposed design as part of Hyperledger Fabric

3.2 Protocol overview

In this section, we propose our ordering layer and provide an overview of Orceval. Our ordering layer is leaderless and consists of multiple ordering peers (in the following referred to as orderers or peers). These orderers are responsible for running a two-stage protocol.

3.2.1 Per Block Validation

The first stage is Per Block Validation. In this phase, all peers independently validate any incoming transactions optimistically. The orderer locally checks the endorsement policy and whether all reads performed by the current transaction are up-to-date. At this stage, the peer does not communicate with other orderers and assumes there are no read-write conflicts between them. As part of this phase, the peer collects the reads and writes of every - as of now considered valid - transaction to speed up later steps. The phase ends in one of three ways. Either the peer has filled a block to maximum capacity, another orderer sends its block, or it reaches an adjustable time limit. Either way, the peer sends its block to all other orderers.

3.2.2 Inter Block Validation

The second and last stage is Inter Block Validation. It ensures correctness by invalidating any transaction that would otherwise cause read-write conflicts between the blocks generated in the last phase.

The phase starts by collecting the blocks created by all orderers. While every peer stores the blocks from every other orderer, only one performs the validation.

3.3 Assumptions

In this section, we list our assumptions.

Trusted Execution Environments: We assume that these peers utilize Trusted Execution Environments to run the algorithm, allowing outside parties like clients or endorsement peers to verify the executed code via remote attestation.

No Byzantine Faults: Any implementation of this protocol has to guarantee that it cannot cause or result in Byzantine faults. Otherwise, the remote attestation cannot guarantee correct behavior. However, a peer can still crash.

No key leakage: We assume there is no key leakage that would compromise the correctness of authentication in the remote attestation or otherwise.

Peer awareness: All peers are aware of all other orderers at any time. Our design does not focus on how to add or remove orderers from the system. It is assumed that either the orderers remain the same or there is a separate independent coordination mechanism.

Crash recovery: After a crash, a peer can recover its storage, especially all transactions it has stored before crashing.

Reliable communication: Messages are transmitted reliably and the transmission delay is bounded.

Few write conflicts: Our system shares the assumption with Hyperledger Fabric that there are only few write conflicts. This is especially important in our case because of the increased latency mentioned in section 3.1. As validation increases the necessary computations and thus the latency, the chance of endorsement peers simulating transactions on outdated values is higher. Whenever such a transaction is submitted to the ordering layer, it has to be invalidated and therefore likely simulated again by the endorsement peers. Consequently, we focus on applications with few read-write conflicts.

3.4 Communication

In this section, we explain the communication protocol of Orceval. Generally, any communication between the orderers, from a client, or to the endorsement peers uses

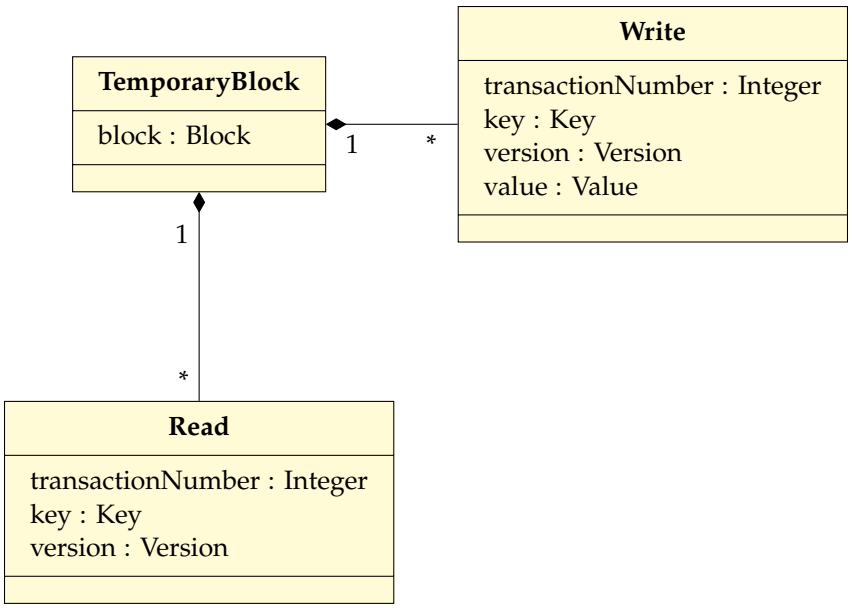


Figure 3.2: Temporary block wrapping a block with read and write information

TCP. The collective ordering network accepts transactions sent to any peer. As part of the Per Block Validation phase, it will create temporary blocks (see Figure 3.2) with combined read and write sets for all transactions part of it. The reads and writes provide a reference to the transaction performing the respective operation within the given block. These temporary blocks are then shared with all orderers but only processed by a single predetermined one. This peer performs the Inter Block Validation and creates finalized blocks containing the transactions collected by all peers in the first phase.
Generally, our protocol uses temporary and final blocks as coordination messages to minimize overhead. The message flow is illustrated in Figure 3.3. When the second peer finishes its block by either reaching the maximum number of transactions permitted in a block or by reaching a time limit. Then the peer sends its temporary block to the peers one and three which then assemble their temporary blocks from the transactions they have collected. The resulting blocks do not have to be full. After they disseminate their blocks, the second peer executes the Inter Block Validation. Once it is finished, the final blocks are sent to the peers one and three and the endorsement peers as described in section 3.1. When the orderers receive the final blocks, they resume Per Block Validation and accept new transactions.

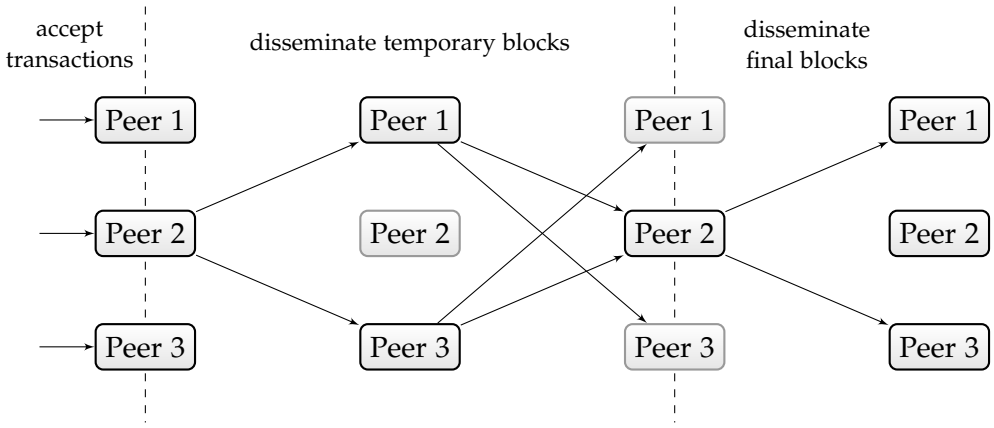


Figure 3.3: Overview of the communication flow with peer 2 finishing its temporary block first and being the leader of the Inter Block Validation (grey outline indicates inactivity)

3.5 Workflow

In this section, we explain the Per Block Validation and the Inter Block Validation in more detail.

3.5.1 Per Block Validation

Figure 3.4 shows the validation process in Per Block Validation. To determine the validity of a given transaction the peer has to perform two checks.

Read versions check: The peer checks whether the transactions reads from an outdated value. For that, the peers compare the version of the keys in the read set to a local state. This state comprises of the general state from the finished blocks in the blockchain as well as the writes from the previously validated transactions from the current phase. If a transaction reads from a key with an outdated version, the transaction is marked invalid.

Endorsements check: For endorsements to be accepted, they have to be from a valid set of endorsement peers and the signatures have to match the content of the transaction. Which set of endorsement peers are valid for a given transaction is determined by the endorsement policy.

Any transaction passing these tests is marked valid and used to update the read and write set of the current block as well as the versions of the local state.

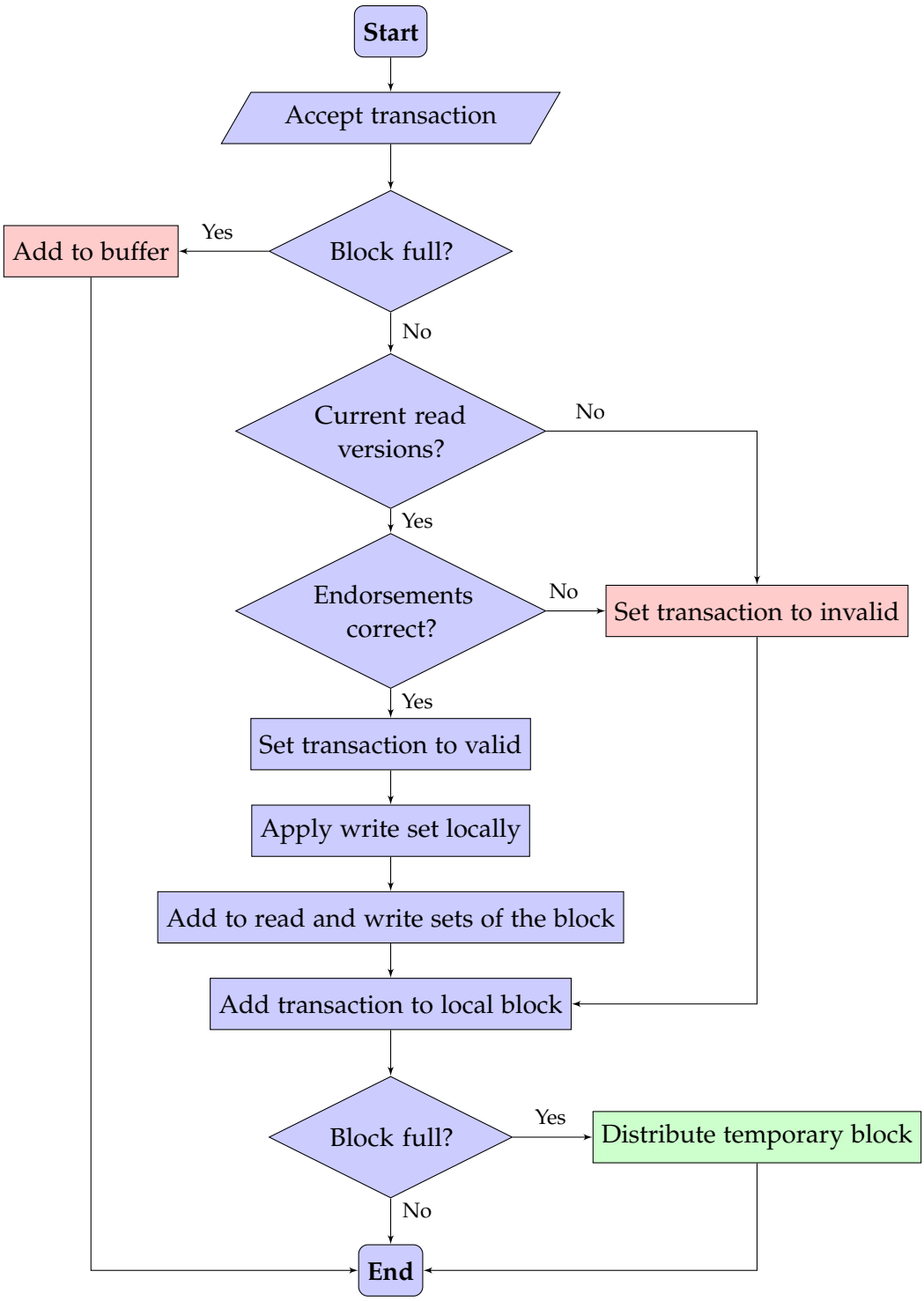


Figure 3.4: Workflow of Per Block Validation within a given peer (Flowchart)

The phase is over when either the peer has reached the maximum capacity for a block, it receives a temporary block from another orderer, or a timer, starting with the first incoming transaction, runs out. This timer limits the maximum time an orderer waits for new incoming transactions to finish their block. If any of these three conditions are met, the peer disseminates its temporary block.
Both during and after the orderers distribute their temporary blocks, they buffer incoming transactions and await the result of the next phase. When it concludes, the transactions from the buffer are validated and new incoming transactions are processed.

3.5.2 Inter Block Validation

At the start of the phase, the peer responsible for performing Inter Block Validation is picked by a deterministic function that cycles through all orderers. Therefore, no leader election takes place.
This temporary leader checks for read and write conflicts across the collected blocks. For each block the read set is scanned for incorrect versions and the according transactions are invalidated. Then any writes by a now invalidated transaction are removed from the write set. The validated write sets of this and all previous blocks of the current round are collected to a local state to check the read set of the next block. Lastly, the orderer finishes the block by adding the hash of the previous block to the current one.
Similarly to the leader, the order, in which the blocks are validated, is derived from a deterministic function that provides an equal chance for the block of any orderer to be at any position. This avoids starvation of a transaction by giving every orderer an equal chance to have write access to a key in case it is used often. When all blocks are validated, the peer distributes them to the other orderers and outside endorsement peers.
We define the deterministic function for both cases as $(B \div O) \bmod O$ with B being the current number of finished blocks in the blockchain excluding the current round and O being the total number of orderers. The result of the function is rounded down to map to an index between 0 and $O - 1$. Assuming all peers are online, a round usually produces O blocks. Consequently, the function returns a different index every round. In the case of choosing the leading orderer, this helps with load balancing, as the orderer responsible for the Inter Block Validation changes every round. For deriving the order in which the blocks are to be validated, the output of the function is used as the index of the first block to be checked in Inter Block Validation as illustrated in Figure 3.5.
Using the same function for both applications trivially leads to the index of the leading orderer and the first block being identical. Still, the function is evaluated separately for both use cases. This is necessary because if a peer crashes during Inter Block Validation, another peer can take over and produce the same result. This topic will be discussed

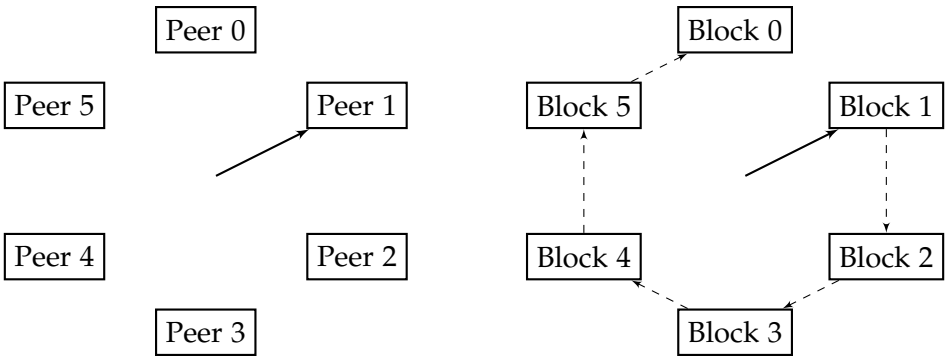


Figure 3.5: Illustration of the deterministic functions determining the orderer id (left) and the order in which the blocks are validated (right)

further in section 3.7.

3.6 Data Model

In this section, we list the guarantees Orcival achieves. The correctness of the blockchain is always ensured by our system. The valid transactions adhere to strict serializability as transactions that would result in read-write conflicts are invalidated. Additionally, for any transaction to be validated it has to have the right endorsements according to the relevant endorsement policy.
Regarding availability, the system is able to handle crash faults and make progress as long as at least one peer is online. Lastly, because we are running on Trusted Execution Environments, our system is capable to provide equivalent guarantees to Byzantine Fault tolerant ordering layers.

3.7 Safety

In this section, we describe how the guarantees from the previous section are ensured.

3.7.1 Correctness of the blockchain

To maintain a correct blockchain any given transaction is either committed entirely or invalidated. If a transaction does not include the necessary endorsements or reads an outdated value from a key, the accepting peer will invalidate the transaction in Per Block Validation. Should a transaction be modified by a malicious client after collecting the endorsements, then the signature check of the endorsement validation will detect

the change and mark the given transaction as invalid.
Lastly, any read-write conflicts occurring between different orderers are detected by the peer performing Inter Block Validation. Any transaction passing all of these tests is marked valid and all changes are committed to the blockchain.

3.7.2 Fault Tolerance

To provide similar guarantees as Byzantine Fault Tolerant ordering networks, the system has to handle Byzantine faults including crashes. A crash can occur either while performing Per Block Validation or Inter Block Validation. Crashes generally are detected by missing TCP acknowledgements during sending operations or by reaching adjustable timeouts.

Crash during Per Block Validation: This impacts a client trying to submit new transactions or other orderers expecting temporary blocks. A client detects the crash with missing TCP acknowledgements and can then send its transactions to a different orderer until the crashed peer recovers. Any partially sent transactions will be dismissed by the ordering peers and therefore an incomplete transmission has to be resubmitted by the client. Other peers can detect a crashed peer during sending their temporary blocks or by reaching a timeout while waiting for a response. In case of a crashed peer the orderers continue the Inter Block Validation without its temporary block.

Crash during Inter Block Validation: If the crashed peer is responsible for performing the Inter Block Validation that round, the next orderer according to the deterministic function (described in section 3.5.2) will execute the validation. Likewise, if the peer crashed during Inter Block Validation, the next orderer will take over. Because the order in which the blocks are validated is independent from the peer executing Inter Block Validation (as described in section 3.5.2), the resulting final blocks from any given round are identical regardless of which orderer creates them.

Crash recovery: To recover, the peer requests all new blocks from another orderer by sending the index of the last block in its blockchain. After the peer uses the new blocks to update its key value store, it revalidates any stored transactions and accepts new transactions from clients.

In summary, crashes are detected either by missing TCP acknowledgement or by timeouts when a peer is expecting temporary or final blocks. The Inter Block Validation skips temporary blocks from crashed peers and can be taken over by another orderer,

should the leading peer crash. The final blocks created are identical, whether they are generated by the original leading peer or by another one because the order, in which the blocks are validated, is deterministic.
As for other Byzantine faults, they specifically cannot occur in the implementation of the protocol (as assumed in section 3.3) and because the code is executed in a Trusted Execution Environment, any client or peer can verify correct behaviour. Therefore, this system provides the same guarantees as a Byzantine Fault Tolerant ordering layer.

4 Implementation

In this chapter, we describe the implementation of the proposed system used for the evaluation in the next chapter.

4.1 Used tools

Netty A java library for non-blocking networking [33]. It is based on channels and provides a pipeline system for easily encoding and sending blocks to other orderers as well as accepting and decoding incoming transactions or blocks from clients or orderers respectively.

Jackson API A simple java API for serializing and deserializing JSON [23]. Every transaction and block is encoded in JSON for human readability and to ensure the completeness of every transmitted message.

4.2 General components

Every orderer has a central Networking object, that keeps track of the other orderers and sets up and stores the respective channels to and from them. It offers two functions `disburseBlock(TempBlock block)` and `disburseBlocks(Block[] blocks)` to the objects `PerBlockValidation` and `InterBlockValidation`. These objects handle the logic behind both phases. Any incoming transaction is decoded and passed to the function `addTransaction(Transaction tx)` from `PerBlockValidation`. Accordingly, any incoming temporary block is decoded and passed to the `InterBlockValidation` via the function `addBlock(TempBlock block, int ordererIndex)`.

4.3 Networking with Netty

As mentioned in section 4.1, Netty is based on a channel and pipeline system. To start new channels there are two bootstrap classes. One is the `ServerBootstrap` which is bound to a local port and IP address to accept incoming connections. The second bootstrap is the simple `Bootstrap` which is bound to a remote port and IP address

to start a connection to another host. Both bootstrap classes can be used to define a pipeline with inbound and outbound handlers. In our implementation, every `Bootstrap` has a `JsonEncoder` object in its pipeline to encode every outbound message with the `ObjectMapper` from the Jackson API. Accordingly, every `ServerBootstrap` has a specific JSON decoder object to decode the message to the expected Java class. Additionally, after a JSON decoder, there is a handler for every class to pass on the received objects. As Netty is a non-blocking networking framework, the handlers as well as the JSON encoder and decoder can be executed in parallel. Therefore, if multiple transactions are received simultaneously, they can be decoded and handled concurrently. An incoming transaction is sent to the local `PerBlockValidation` object, while incoming temporary blocks are passed to `InterBlockValidation`. Incoming final blocks update the keys of the peer and unblock both validation classes to start the next round of the protocol.

4.4 Per Block Validation

As mentioned, the Per Block Validation is handled by a class by the same name. This class mainly performs the steps illustrated in Figure 3.4. This and other classes make heavy use of Javas `Streams` to check multiple elements in parallel. For example here is the implementation of checking the read set of a given transaction:

```
private Boolean checkVersions(Transaction Tx) {
    return Arrays.stream(Tx.getReads()).parallel()
            .allMatch(read ->
                read.getVersion() == getTempVersion(read.getKey()));
}
```

The function `getTempVersion(long key)` returns the most current version of the given read key from the perspective of the executing peer. This is either a new version by a transaction currently only known to this peer or the most recent version from finished blocks known to all peers. Apart from the `addTransaction(Transaction tx)` function the class also provides a `flush()` function, which is called when new final blocks have been processed and the peer can now resume validating transactions starting with the buffer.

4.5 Endorsement Validation

The endorsement validation is handled by a separate object that stores the keys of the endorsing peers grouped by their organizations. In this implementation, the keys are defined as fixed strings in the code to ensure all peers have the same set of keys. The

endorsement policy checked in our implementation is that a single peer from every known organization has to have endorsed any given transaction. The keys are stored in an array of sets called `keySets` so they can be compared against as fast as possible:

```
boolean validSignatureSet = Arrays.stream(keySets).parallel()
                            .allMatch(keySet ->
                                Arrays.stream(keys).parallel()
                                .anyMatch(keySet::contains)
                            );
```

This line of code checks whether the endorsement policy is met by the given `keys` from the transaction. If the policy is met, the signatures are verified next. For this java's own java.security.Signature class is used and verification is performed in parallel:

```
Arrays.stream(keyAndEndorsements).parallel().allMatch(keyAndEndorsement ->
{
    Signature sign;
    try {
        sign = Signature.getInstance(signatureAlgorithm);
        sign.initVerify(keyAndEndorsement.key);
        sign.update(data);
        return sign.verify(keyAndEndorsement.endorsement);
    } catch (NoSuchAlgorithmException | SignatureException |
             InvalidKeyException e) {
        throw new RuntimeException(e);
    }
})
```

The results are then passed back to `PerBlockValidation`.

4.6 Inter Block Validation

The main function `addBlock(TempBlock block, int ordererIndex)` performs the following steps:

1. Check whether the executing peer is the leader for the current round.
2. Check whether the incoming block is the next block to be verified.
3. If so, perform `validateBlock(TempBlock block)` on it and validate the following blocks in the list until one is missing.

4. If all blocks of the current round are validated, then distribute the finished blocks, otherwise, wait for the next incoming temporary block.

The validation function again uses parallel streams in java to check the currency of the versions of every read, collect the invalid transactions, mark them as invalid in the block, and remove them from the write set. Lastly, the updated write set is stored for future read set checks. Just like the `PerBlockValidation`, there is a `flush()` function to clear out the cashed temporary blocks and prepare the peer for the next round of the protocol.

4.7 Client

The client is designed for benchmarking and reuses most of the functionality from the orderers. The `NetworkClient` opens a port to receive final blocks and stores them with a timestamp. The inner class `NetworkClient.ClientThread` can be instantiated to submit random transactions to a fixed orderer. Every submitted transaction is also stored in a list in the `NetworkClient` along with a timestamp. When the threads are finished, the `NetworkClient` matches the sent transactions to the received transactions, and calculates the latencies as well as the throughput.

4.8 Trusted Execution Environments

For our evaluation, we chose not to execute our implementation in Trusted Execution Environments (TEE). On one hand, our chosen hardware does not support Intel SGX. On the other hand, the focus of our evaluation is to explore the benefits and limitations of our protocol, not our implementation. However, we acknowledge that our decision has an impact on our evaluation. Running our implementation in a Trusted Execution Environment would have reduced performance. The specific performance degradation differs depending on how the implementation is adapted. For Intel SGX, there are multiple systems for directly executing unaltered binaries in a TEE [7, 28, 35, 47, 50]. SCONE specifically claims that in their tests, they reach at least 60% performance compared to running natively [7]. Alternatively, the implementation can be written directly for SGX with the Intel SGX SDK[21] which would likely result in better performance compared to running the unaltered implementation in SGX using containers. Apart from performance, SGX had until recently a memory restriction to 256 MB and other limitations that restricted its applicability. However, in 2021 Intel released new processors[1] with enhancements to SGX which improve performance, increase maximum memory support to 512 GB per socket, and introduce multi-socket

support [19]. Generally, because Trusted Execution Environments are continuously improved upon, the performance difference between running code natively as opposed to using TEEs will likely continue to decrease in the future.
Overall, while we chose not to run our implementation using Trusted Execution Environments, we are confident that the results of our benchmarks are indicative of the benefits and limitations of Orceval.

5 Evaluation

In this chapter, we compare the throughput of our implementation with BFT-SMaRt[41], examine a bottleneck, and evaluate our system for different block sizes and different numbers of orderers.

5.1 Setup

All tests are run in Cloudlab[18] on machines designated as "c220g5"[12] on the cluster at the University of Wisconsin. Each is equipped with two Intel(R) Xeon(TM) Silver 4114, with 10 physical cores (20 threads) each and 196608 MB of memory. The nodes are connected to HP and Dell leaf switches which themselves are connected to two interconnected spine switches [16]. The resulting network has a bandwidth of 10 Gbps. Except for section 5.4 all our tests are performed on four peers.
Further, the benchmarks have multiple adjustable parameters. Both BFT-SMaRt and our implementation allow for multiple clients to run in parallel, each submitting a fixed number of requests or transactions. The requests are sent at a certain interval. The byte size of the handled value can be set in addition to whether the operations are read-only or read and write. For most tests we performed, the number of clients is 50, and the number of requests they send is 500 respectively. Each value is four bytes in size and every operation is a write operation reading one value and writing one value.
Orceval handles transactions and a lot of metadata like read and write sets, therefore the transmitted data is generally larger compared to BFT-SMaRt. Lastly, the blocks our system creates contain - unless stated otherwise - up to 25 transactions. As discussed in section 4.8, our implementation does not run on Trusted Execution Environments. Consequently, our tests of Orceval are not fully representative of the specific performance of our setup.

5.2 Throughput and latency

While the interval is at or above 40ms, the throughput of our system - as seen in Figure 5.1 - is comparable to BFT-SMaRt, even though our system performs endorsement validation and read-write conflict detection in addition to ordering. Below an interval

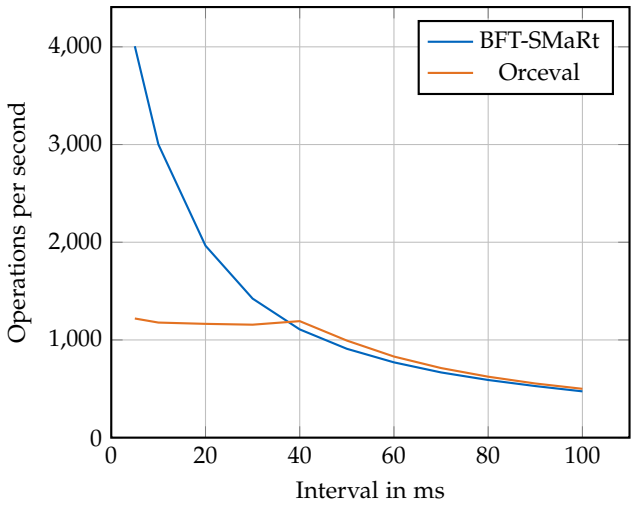


Figure 5.1: The throughput of BFT-SMaRt and Orceval for different input intervals

of 40ms, the throughput is constant at circa 1200 operations per second. At that point, any incoming transactions are buffered and wait for validation. This is reflected by the latency as seen in Figure 5.2. At an interval of 100ms, the maximum latency is elevated as a transaction has to be stored until there are enough transactions to fill a temporary block. On the other hand, at or below 40ms the latency elevated again, as the limit for throughput has been reached and incoming transactions have to await validation. The upper bound of the throughput suggests the existence of a bottleneck in Orceval.
This bottleneck is the computation of the endorsement validation as part of the Per Block Validation. Figure 5.3 plots the ratio of time spent in endorsement validation divided by the total time spend in Per Block Validation including waiting for transactions. At an interval of 40ms, the endorsement validation takes up 99.8% of the Per Block Validation. Thus, any lower interval is bound by the computation of the endorsements and the system cannot reach a higher throughput. However, verifying the endorsements is central to the validity of any given transaction and cannot be left out. On one hand, it is needed to verify the integrity of the transaction and defend against malicious clients (as discussed in section 3.7) and on the other hand, the overarching system relies on endorsements to generate trust between untrusted parties. Therefore, the Per Block Validation can mainly be accelerated by more efficient signature checking algorithms or by using more powerful processors.
In summary, our system is comparable to BFT-SMaRt in terms of throughput despite performing validation as well as ordering. However, the validation - in particular the endorsement validation - limits the maximum throughput and consequently directly

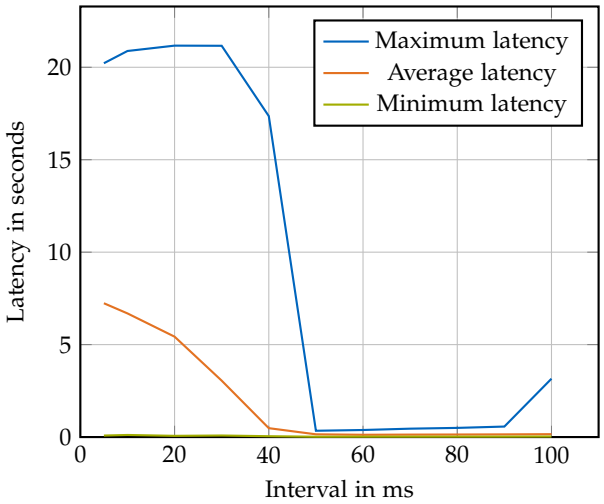


Figure 5.2: The latency of Orcival for different input intervals

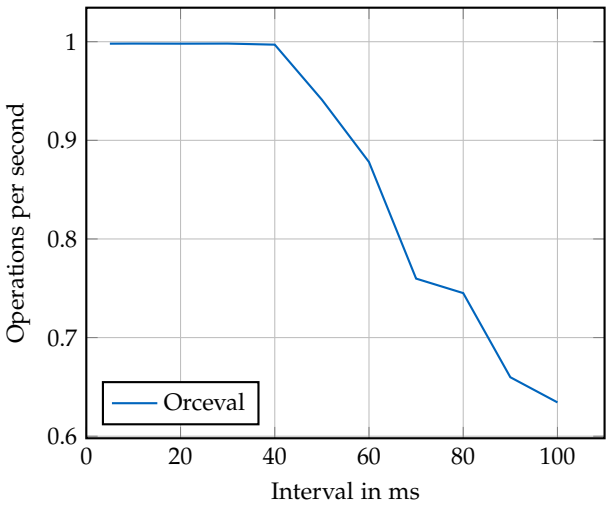


Figure 5.3: Average ratio of time spent in endorsement validation of the total time in Per Block Validation for different input intervals

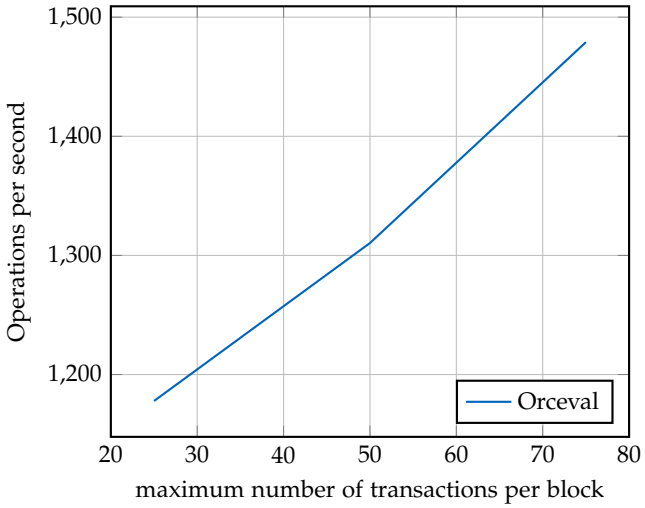


Figure 5.4: Comparison of the throughput of Orceval for different block sizes

increases latency once it is exceeded.
Lastly, we do not include BFT-SMaRt in Figure 5.2, because the latency of Orceval is not easily comparable to BFT-SMaRt, as it does not validate any transactions. Additionally, while BFT-SMaRt distributes the ordered blocks one by one, Orceval releases four at a time as all orderers generate them mostly independently. Consequently, BFT-SMaRt generally has significantly lower latency. When comparing both systems at an interval of 50 ms, the average latency of Orceval is 45 times higher at 149 ms compared to BFT-SMaRt at 3.3 ms. Overall, the highest latency reported by BFT-SMaRt was 6.4 ms at an interval of 5ms. This is less than half of the lowest latency reported by Orceval at 18 ms with a 50 ms interval.

5.3 Performance for different block sizes

In this section, we will take a brief look at how different block sizes affect the throughput of our system. For this test, the interval is set to 10ms to observe the maximum throughput and otherwise the default parameters were used, as explained at the beginning of this chapter. In addition to the default block size of 25, sizes of 50 and 75 were tested. The results for 50 and 75 in Figure 5.4 are averages of three runs for each size. The increased block size allows the peers to spend more time independently validating transactions and decreases the necessary coordination between them. Additionally, as the orderers wait after the Per Block Validation for the Inter Block Validation to finish, the peers have to enter a blocked state less often.

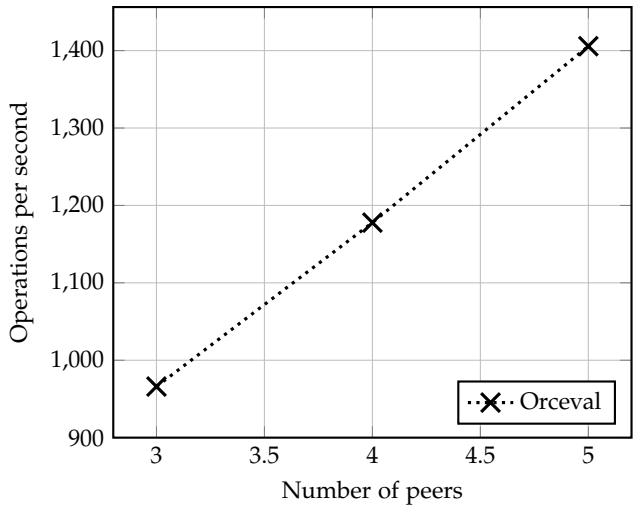


Figure 5.5: Comparison of the throughput of Orceval for different numbers of peers

However, because the blocks are bigger, the peer performing the Inter Block Validation has to check larger read and write sets for conflicts. And while the number of messages sent between the peers decreases, the size of the messages increases.
Therefore in general, a bigger block size has the benefit of reducing the total time a peer spends coordinating with other orderers and waiting for finished blocks from the Inter Block Validation. Thus, the peer can spend more time in Per Block Validation which due to Endorsement Validation is the most acute bottleneck.

5.4 Scalability

Another important matter is how our system scales to different numbers of orderers. In this section, we compare our four-peer network from previous sections to networks of three and five orderers. Similarly to the last section, we use the default parameters from the beginning of the chapter, an interval of 10ms, and the previously mentioned numbers of peers. The results for three and five nodes in Figure 5.5 are again averages of three runs each. The three-peer setup achieves circa 85% of the performance of the four-peer setup while the five-peer setup marks a circa 19% improvement over that same four-peer setup.
Having more peers in the network allows for more transactions and especially more computationally intensive endorsements to be checked in parallel. On the other hand, the Inter Block Validation now has to validate more blocks each round. Overall, because of the time-intensive signature checks, more peers will improve performance. But

at a certain amount of peers, the increased coordination and Inter Block Validation overheads can potentially outweigh the performance gain from a more distributed Per Block Validation.
In summary, Orceval is quite scalable and while there is a potential limit, the endorsement validation as the main bottleneck is perfectly parallelizable.
In contrast, BFT-SMaRt is shown to decrease throughput as the number of peers increases [41]. The main reason is that for a higher number of peers the system has to guarantee tolerance to more faulty nodes following the $3f + 1$ ratio. To ensure higher tolerance, BFT-SMaRt has to distribute more coordination messages and therefore its performance decreases.

6 Future Work

6.1 Possible improvements to our design

As alluded to in chapter 5, the main bottleneck of our implementation is signature checking as part of the endorsement evaluation. Consequently, one intuitive improvement could derive from exploring different signature algorithms or more efficient implementations with better utilization of specific hardware.
However, there is a more general improvement which does not depend on changing the endorsement validation. Our design and implementation require the orderers to pause Per Block Validation while the Inter Block Validation takes place. Consequently, especially the peers not performing the Inter Block Validation do not fully utilize their available capacity. Hence, as an improvement, all peers could continue running Per Block Validation even while Inter Block Validation takes place. Although the orderers would now risk validating the read set on outdated versions, the Inter Block Validation would detect the conflict and therefore still ensure correctness. This improvement could significantly increase throughput, especially because this allows the peers to perform the endorsement validation at times when in our current design they would be blocked.
Lastly, our design could be adapted to use cases with more read-write conflicts. Our system could be stripped of its validation capabilities and used as an ordering layer in the original Hyperledger Fabric. If the peers do not perform endorsement validation or read set checks, trivially, the latency will be significantly lower. Because our system is designed to have minimal coordination between the peers, this modification has the potential to have similar or lower latency compared to other Byzantine Fault Tolerant ordering layers. This however has some significant drawbacks. Firstly, this requires each individual peer to perform validation which our design would execute once. And secondly, the endorsement peers, therefore, have to spend time on validation thus decreasing throughput for incoming new transactions. Overall, this modification would increase power consumption and either require more endorsement peers or reduce throughput with the same amount of peers. But it would provide the benefit that endorsement peers get notified about changes to the versions and values sooner and therefore prevent them from performing simulations with outdated values.

6.2 Exploration of other designs

In this section, we introduce two alternative designs we explored. Intuitively, adding validation to the ordering process gives the ordering layer awareness of the content of the transactions. Therefore, the order itself can be influenced by the validation and transactions rearranged to resolve potential read-write conflicts. In the following, this idea is referred to as intelligent or context-aware ordering.
A simple way to apply this concept is to have a replicated leader algorithm as the ordering layer. This leader creates a fixed list of transactions and searches for the optimal position to place a new transaction. If there are no conflicts, the transaction is appended to the end of the list. Otherwise, the possible positions can be determined by an upper and a lower bound. The upper bound is determined from the first transaction which overwrites any keys read by the current transaction. Conversely, the lower bound is determined by the last transaction which reads from a key the current transaction writes to. If the calculated lower bound is higher than the upper bound, then the transaction cannot be positioned and has to be invalidated. Otherwise, the transaction can be placed at any position within the bounds. This idea is limited mainly by having a central transaction list maintained by the leader. This leader consequently is the bottleneck.
Another approach is to maintain a distributed dependency graph (inspired by the Zeus protocol[24]). The graph can be split into components that are independent of each other. Each component can be "owned" by a different peer which internally maintains a dependency graph for each component. When a transaction creates a dependency between two or more components, ownership of them has to be transferred to a single peer which then generates the new dependency graph for the given component. This protocol is more intensive, as it builds the actual dependency graph, but it allows for an independent, distributed calculation of the dependencies and thus the order of transactions.
However, intelligent ordering in general is conceptually conflicting. Because of the added computation context-aware ordering entails, both approaches necessarily increase latency. As described in section 3.3, intelligent ordering should then mainly be applied to use cases with few write conflicts which would therefore make the computations irrelevant. Further research is required to determine how much latency intelligent ordering adds and whether the resulting trade-off might be beneficial in certain applications.

7 Conclusion

We presented Orceval, a leaderless, scalable, and trusted ordering layer for Hyperledger Fabric. We discussed the three-phase protocol of Hyperledger Fabric and how it achieves high throughput by utilizing parallelization. Then we proposed combining the last two phases of Fabric - namely ordering and validation - and performing them in the central ordering layer. Using Trusted Execution Environments, Orceval achieves equal guarantees to BFT ordering layers and is able to perform validation centrally as a verifiably trustworthy party. Orceval uses a two phase system to minimize communication overhead. In the first phase the individual peers validate any incoming transactions optimistically and in the second phase a temporary leader collects the transactions, resolves any conflicts, and distributes the finished blocks. After discussing our implementation, we compared the performance of Orceval to BFT-SMaRt. Orceval achieves similar throughput to BFT-SMaRt even though it does not perform validation. However, the signature checks of Orceval prevent it from reaching the same maximum throughput as BFT-SMaRt. Lastly, we discussed possible ways to improve the performance of Orceval and introduced two other concepts for using Trusted Execution Environments in ordering layers.

List of Figures

List of Tables

Bibliography

[1] *3rd Gen Intel(R) Xeon(R) Scalable Processors*. URL: https://www.intel.com/content/www/us/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html (visited on 08/31/2022).

[2] *ADVANCED TRUSTED ENVIRONMENT:OMTP TR1*. Tech. rep. OMTP Limited, 2009. URL: http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf (visited on 08/31/2022).

[3] F. A. Alabdulwahhab. "Web 3.0: The Decentralized Web Blockchain networks and Protocol Innovation." In: *2018 1st International Conference on Computer Applications & Information Security (ICCAIS)*. 2018, pp. 1–4. DOI: 10.1109/CAIS.2018.8441990.

[4] *AMD Secure Encrypted Virtualization (SEV)*. URL: https://developer.amd.com/sev/ (visited on 08/31/2022).

[5] 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." In: *Proceedings of the Thirteenth EuroSys Conference*. ACM, Apr. 2018. DOI: 10.1145/3190508.3190538. URL: https://doi.org/10.1145%5C%2F3190508.3190538.

[6] L. Ante. "Non-fungible token (NFT) markets on the Ethereum blockchain: Temporal development, cointegration and interrelations." In: *Available at SSRN 3904683* (2021).

[7] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, et al. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[8] G. Ayoade, V. Karande, L. Khan, and K. Hamlen. "Decentralized IoT data management using blockchain and trusted execution environment." In: *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE. 2018, pp. 15–22.

[9] M. Bailleu, D. Giantsidi, V. Gavrielatos, V. Nagarajan, P. Bhatotia, et al. "Avocado: A Secure In-Memory Distributed Storage System." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 65–79.

[10] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution." In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 2019, pp. 173–190.

[11] A. Bessani, J. Sousa, and E. E. Alchieri. "State machine replication for the masses with BFT-SMART." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 355–362.

[12] *c220g5 node in cloudlab*. URL: `https://www.wisc.cloudlab.us/portal/show-nodetype.php?type=c220g5` (visited on 08/31/2022).

[13] M. Castro. "Practical Byzantine fault tolerance." In: *OSDI '99*. 1999.

[14] *Change Healthcare using Hyperledger Fabric to improve claims lifecycle throughput and transparency*. URL: `https://www.hyperledger.org/learn/publications/changehealthcare-case-study` (visited on 08/31/2022).

[15] *Circulor achieves first-ever mine-to-manufacturer traceability of a conflict mineral with Hyperledger Fabric*. URL: `https://www.hyperledger.org/learn/publications/tantalum-case-study` (visited on 08/31/2022).

[16] *CloudLab Hardware*. URL: `https://docs.cloudlab.us/hardware.html` (visited on 08/31/2022).

[17] *Digital Art NFTs*. URL: `https://www.nft.com` (visited on 08/31/2022).

[18] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. "The Design and Operation of {CloudLab}." In: *2019 USENIX annual technical conference (USENIX ATC 19)*. 2019, pp. 1–14.

[19] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig. "Benchmarking the Second Generation of Intel SGX Hardware." In: *Data Management on New Hardware*. 2022, pp. 1–8.

[20] *Honeywell Aerospace creates online parts marketplace with Hyperledger Fabric*. URL: `https://www.hyperledger.org/learn/publications/honeywell-case-study` (visited on 08/31/2022).

[21] *Intel(R) Software Guard Extensions SDK for Linux OS*. URL: `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html` (visited on 08/31/2022).

[22] *Intel(R) Software Guard Extensions(Intel(R) SGX)*. URL: https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/software-guard-extensions.html (visited on 08/31/2022).

[23] *Jackson Project*. URL: https://github.com/FasterXML/jackson (visited on 08/31/2022).

[24] A. Katsarakis, Y. Ma, Z. Tan, A. Bainbridge, M. Balkwill, A. Dragojevic, B. Grot, B. Radunovic, and Y. Zhang. "Zeus: Locality-aware distributed transactions." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 145–161.

[25] L. Lamport. "Byzantizing Paxos by Refinement." In: *Distributed Computing: 25th International Symposium: DISC 2011, David Peleg, editor. Springer-Verlag.* (June 2011), pp. 211–224. URL: https://www.microsoft.com/en-us/research/publication/byzantizing-paxos-refinement/.

[26] Y. Liang, Y. Li, and B.-S. Shin. "Faircs—blockchain-based fair crowdsensing scheme using trusted execution environment." In: *Sensors* 20.11 (2020), p. 3172.

[27] J. Lind, I. Eyal, F. Kelbert, O. Naor, P. Pietzuch, and E. G. Sirer. "Teechain: Scalable blockchain payments using trusted execution environments." In: *arXiv preprint arXiv:1707.05454* (2017).

[28] *Linux Stacks for Intel(R) Software Guard Extensions (Intel(R) SGX)*. URL: https://www.intel.com/content/www/us/en/developer/topic-technology/open/sgx-stacks/overview.html (visited on 08/31/2022).

[29] D. Livshits, A. Mikityuk, S. Pham, and A. Shabtai. "Towards Security of Native DRM Execution in HTML5." In: *2015 IEEE International Symposium on Multimedia (ISM)*. IEEE. 2015, pp. 411–416.

[30] L. P. Maddali, M. S. D. Thakur, R. Vigneswaran, M. Rajan, S. Kanchanapalli, and B. Das. "VeriBlock: A novel blockchain framework based on verifiable computing and trusted execution environment." In: *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE. 2020, pp. 1–6.

[31] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. "The honey badger of BFT protocols." In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 31–42.

[32] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Tech. rep. 2009. URL: http://bitcoin.org/bitcoin.pdf (visited on 08/31/2022).

[33] *Netty project*. URL: https://netty.io (visited on 08/31/2022).

[34] T. A. T. Nguyen and T. K. Dang. "Privacy preserving biometric-based remote authentication with secure processing unit on untrusted server." In: *IET Biometrics* 8.1 (2019), pp. 79–91. DOI: https://doi.org/10.1049/iet-bmt.2018.5101. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-bmt.2018.5101. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-bmt.2018.5101.

[35] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. "SGX-LKL: Securing the host OS interface for trusted execution." In: *arXiv preprint arXiv:1908.11143* (2019).

[36] F. Saleh. "Blockchain without Waste: Proof-of-Stake." In: *The Review of Financial Studies* 34.3 (July 2020), pp. 1156–1190. ISSN: 0893-9454. DOI: 10.1093/rfs/hhaa075. eprint: https://academic.oup.com/rfs/article-pdf/34/3/1156/36264598/hhaa075.pdf. URL: https://doi.org/10.1093/rfs/hhaa075.

[37] A. T. Sherman, F. Javani, H. Zhang, and E. Golaszewski. "On the Origins and Variations of Blockchain Technologies." In: *IEEE Security & Privacy* 17.1 (2019), pp. 72–77. DOI: 10.1109/MSEC.2019.2893730.

[38] D.-H. Shih, T.-W. Wu, M.-H. Shih, W.-C. Tsai, and D. C. Yen. "A Novel Auction Blockchain System with Price Recommendation and Trusted Execution Environment." In: *Mathematics* 9.24 (2021), p. 3214.

[39] *Sony Global Education Chooses Hyperledger Fabric for a Next-Generation Credentials Platform*. URL: https://www.hyperledger.org/wp-content/uploads/2017/12/Hyperledger_CaseStudy_Sony.pdf (visited on 08/31/2022).

[40] J. Sousa and A. Bessani. "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation." In: *2012 Ninth European Dependable Computing Conference*. IEEE. 2012, pp. 37–48.

[41] J. Sousa, A. Bessani, and M. Vukolic. "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform." In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 51–58. DOI: 10.1109/DSN.2018.00018.

[42] G. Su, W. Yang, Z. Luo, Y. Zhang, Z. Bai, and Y. Zhu. "BDTF: A blockchain-based data trading framework with trusted execution environment." In: *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*. IEEE. 2020, pp. 92–97.

[43] V. Sumina. *26 Cloud Computing Statistics, Facts & Trends for 2022*. URL: https://www.cloudwards.net/cloud-computing-statistics/ (visited on 08/31/2022).

[44] H. Sun and H. Lei. "A design and verification methodology for a trustzone trusted execution environment." In: *IEEE Access* 8 (2020), pp. 33870–33883.

[45] S. Tamrakar. "Applications of Trusted Execution Environments (TEEs)." English. Doctoral thesis. School of Science, 2017, 119 + app. 105. ISBN: 978-952-60-7463-4 (electronic), 978-952-60-7464-1 (printed). URL: http://urn.fi/URN:ISBN:978-952-60-7463-4.

[46] J. Thalheim, H. Unnibhavi, C. Priebe, P. Bhatotia, and P. Pietzuch. "rkt-io: a direct I/O stack for shielded execution." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 490–506.

[47] D. Tian, J. I. Choi, G. Hernandez, P. Traynor, and K. R. Butler. "A practical intel SGX setting for linux containers in the cloud." In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 2019, pp. 255–266.

[48] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. "Shieldbox: Secure middleboxes using shielded execution." In: *Proceedings of the Symposium on SDN Research*. 2018, pp. 1–14.

[49] *TrustZone for Cortex-A*. URL: https://www.arm.com/technologies/trustzone-for-cortex-a (visited on 08/31/2022).

[50] C.-C. Tsai, D. E. Porter, and M. Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 2017, pp. 645–658.

[51] H. Turki, F. Salgado, and J. M. Camacho. "HoneyLedgerBFT : Enabling Byzantine Fault Tolerance for the Hyperledger platform." In: 2017.

[52] *Unstoppable Domains*. URL: https://unstoppabledomains.com (visited on 08/31/2022).

[53] Q. Wang, R. Li, Q. Wang, and S. Chen. "Non-fungible token (NFT): Overview, evaluation, opportunities and challenges." In: *arXiv preprint arXiv:2105.07447* (2021).

[54] Y. Wang, J. Li, S. Zhao, and F. Yu. "Hybridchain: a novel architecture for confidentiality-preserving and performant permissioned blockchain using trusted execution environment." In: *IEEE Access* 8 (2020), pp. 190652–190662.

[55] G. Wood. *Ethereum: A secure decentralised generalised transaction ledger*. URL: https://gavwood.com/paper.pdf (visited on 08/31/2022).