b
UNIVERSITÄT
BERN

Implementation of an asset transfer system based on parallel blockchain instances

Bachelor Thesis

Jérémie de Faveri

from

Bern, Switzerland

Faculty of Science, University of Bern

30. August 2021

Prof. Christian Cachin
Orestis Alpos
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

The goal of this thesis is to implement an asset transfer system based on parallel blockchain instances and to evaluate its performance. An asset transfer system is a service provided by a server to transfer assets (money) between accounts owned by different people which can be modeled using a state machine. To guarantee the availability and safety of the service, the system can be decentralized on different servers, from here on called replicas, using state machine replication (SMR). The different replicas use a permissioned consensus algorithm, HotStuff, to reach consensus and thus keep a common state. Even if fewer than $1/3$ of all replicas are Byzantine faulty (behave in an arbitrary way), SMR can guarantee a common state between all correct replicas.

If SMR is restricted to state machines implementing an asset transfer system, it is not necessary to totally order all transactions in a single blockchain to guarantee a common state. More precisely, assuming m accounts $a_1, ..., a_m$, m parallelized blockchain instances $b_1, ..., b_m$ may be used, each instance b_i being responsible to totally order the transactions that withdraw money from a_i.

Let there be m clients $c_1, ..., c_m$, with client c_i being authorized to withdraw money from a_i. To transfer x assets from a_i to a_j, account c_i may send a transaction request to blockchain instance b_i on all replicas. If the transaction request is valid, blockchain instance b_i will then order it. All transactions ordered by b_i will then be executed in the correct order by modifying a state that stores the account balances of all accounts and that is common to all blockchain instances of a replica.

To benchmark the system, throughput-latency diagrams of the above described system have been recorded and compared with diagrams of the same system, however running only a single blockchain instance (and a single account), but with the same number of clients and replicas. The results show that the parallelized system has a 60,9% higher throughput compared to the totally ordered one. However, the price for that is a 103,5% higher latency. Nonetheless, as higher throughput is often more valuable than latency improvements, the results are promising and further research in this field may be worthwhile.

Acknowledgments

I would like to thank Orestis Alpos, with whom I discussed a lot fruitfully about the project, who helped me to solve complex problems encountered throughout the thesis, and who encouraged me when needed. His help was very much appreciated. I would also like to thank Christian Cachin for his in-depth expertise in the subject and his support.

For proofreading my thesis and supporting me morally throughout the thesis, I would like to thank Grazia and Lino de Faveri.

Contents

Chapter 1

Introduction

Blockchains have widened the possibilities of decentralization of services in a significant way, making gigantic decentralized asset transfer systems such as Bitcoin possible and attracting more and more companies that are interested in the benefits provided by this technology. Apart from the important security and availability benefits, blockchains can also reinforce collaboration by providing trust between multiple entities such as companies. Because of the high potential of this technology, research in this field abounds: Many papers focus on improving the consensus algorithm to accelerate the decision process, on providing better security guarantees or reducing energy consumption. One of the latest permissioned consensus algorithm, HotStuff, was introduced by "VMWare research group" in 2019 and has been used to implement the asset transfer system of this thesis [Yin+19]. The goal of this thesis is however not to ameliorate a consensus algorithm, but instead to research another way to speed up asset transfer systems, namely parallelization. Indeed, as shown in the paper "The Consensus Number of a Cryptocurrency" by Guerraoui, it is not necessary to totally order all transactions in a single blockchain to guarantee the security and especially to prevent double spending [Gue+19]. Therefore, an asset transfer system that uses one blockchain instance per account has been implemented, each blockchain instance b_i being responsible to totally order all transaction requests issued by account a_i.

Chapter 2 presents important notions used throughout the thesis and shortly introduces the HotStuff consensus algorithm. Chapters 3 and 4 precisely describe the design and implementation of the parallel asset transfer system implemented, while also pointing out what security checks and other procedures are still missing for a productive system. Chapter 3 additionally presents possible advantages and disadvantages of a parallelized asset transfer system and Chapter 4 ends by presenting difficulties encountered while implementing the asset transfer system. Finally, Chapter 5 concludes by discussing the results of the benchmarks between the totally ordered and parallelized asset transfer system.

Chapter 2

Background

2.1 State machine replication (SMR)

Services provided by servers can often be modeled using a state machine.

State machine. A state machine is a service provided by a server that can be modeled by:

- a set of inputs, called commands in the rest of this paper (or transaction requests if the state machine is an asset transfer system): These commands can be sent to the server by clients to request a state change. (e.g., a change in the account balances if the server is an asset transfer system)
- a set of states: If the state is composed of n variables (e.g., account balances) with each variable being able to store m different values (e.g., 2^{32} for an int variable), there are m^n different states.
- a state transition function: $(\text{inputs}, \text{states}) \rightarrow \text{states}$

In the rest of this chapter, we are interested to know if two states are equal, and not interested to explicitly draw state machine diagrams, which would not be feasible with so many states. With this model, it directly follows that if the same commands are executed in the same order on two different instances of a state machine, both instances will necessarily end on the same state, provided that they started on the same state [Sch90] [Wik21b].

A service (that can be modeled as a state machine) provided by a server is critical if a corruption of the sate (i.e., the state machine is on the wrong state) cannot be tolerated. To mitigate such corruptions, a state machine can be run on multiple servers (also called *replicas*) concurrently using a method called *state machine replication*. Such a system continues to work correctly even if fewer than $1/3$ of the replicas are *Byzantine faulty* (have a corrupted state).

SMR (State machine replication). State machine replication is a system to replicate a critical state machine on multiple servers, so that a command sent by a client to these replicas may be executed correctly even if some of the replicas are not working correctly (are Byzantine faulty). All non-faulty replicas running an instance of the state machine must agree on the state. This can be achieved by agreeing on the commands to execute and on their order. Traditionally, state machine replication solves this by running a single *consensus algorithm* between the replicas that *totally orders* all commands in a *single blockchain*.

Byzantine-faulty replicas. Byzantine-faulty replicas are replicas that do not behave correctly, and may in fact behave in an arbitrary way, either unintentionally (e.g., unintentional bug or power loss) or intentionally (e.g., a hacker controls the replica). Correct replicas must be able to reach consensus even if fewer than $1/3$ of the replicas are Byzantine-faulty.

Consensus algorithms and blockchains. Traditionally, replicas do not perform consensus on single commands, but rather on blocks of multiple commands (in a single block, the commands are already totally ordered) to reduce the computational power needed per command. When replicas have agreed on the next block (with the consensus algorithm), they append it to a list of already totally ordered blocks (the blockchain).

Total order vs partial order. A list of commands $\mathbb{T}$ is called totally ordered if every two commands $T, T' \in \mathbb{T}$ are comparable ($T \geq T'$ or $T \leq T'$) . If the list of commands $\mathbb{T}$ is stored on a single blockchain, then it is totally ordered.

A list of commands $\mathbb{T}$ is called partially ordered if every two commands $T, T' \in \mathbb{T}$ are either comparable ($T \geq T'$ or $T \leq T'$) or incomparable ($T \ngeq T'$ and $T \nleq T'$). If the list of commands $\mathbb{T}$ is stored in multiple blockchains, then it is partially ordered, and two commands stored on the same blockchain are comparable, two commands stored on different blockchains are incomparable.

2.2 Asset transfer systems and why they can be parallelized

An asset transfer system is a service provided by a server that can be modeled as a state machine where:

- the set of inputs are transaction requests to transfer money from an account to another.
- the state consists of the balances of all accounts.

As seen in the previous section and given that an asset transfer system can be modeled as a state machine, SMR can be accomplished by totally ordering all transactions in a single blockchain. However, as we will show in this section, if SMR is restricted to state machines that implement asset transfer systems, total order is not required to guarantee a common state [Gue+19].

Asset transfer system that allows negative balances. Applying a transaction consists of withdrawing money from an account and adding the same amount to another account. As additions and subtractions are commutative, the order in which the transactions are applied is irrelevant, and the replicas must only agree on which transactions to apply.

Asset transfer system that does not allow negative balances. Partial order is sufficient. More precisely, assuming m accounts $a_1, ..., a_m$, m parallelized blockchain instances $b_1, ..., b_m$ may be used, each instance b_i being responsible to totally order the transactions that withdraw money from a_i.

In contrary to the system that allows negative balances, certain transactions cannot yet be executed if the account balance is negative as a result of this transaction. Let's assume that account a_1 has 100 coins and has sent two transactions: The first transaction sends 100 coins to a_2, the second 100 coins to a_3. In this case, the replicas must agree on which transaction to execute first, and which transaction should not be executed until a_1 receives 100 coins. If they do not agree on the order, some replicas may execute the first transaction and some the second, which would produce an inconsistent state among correct replicas.

In this thesis, we will try to get an advantage out of this property and implement an asset transfer system that does not allow negative balances by running m parallel blockchain instances.

2.3 HotStuff

HotStuff is a *permissioned* consensus algorithm based on PBFT (Practical byzantine fault tolerance). As explained in the Section 2.1, replicas use consensus algorithms to agree on the blocks of commands and on the order in which they are appended to the blockchain.

If f is the maximum number of Byzantine-faulty replicas that an SMR system can tolerate, then a total of $n = 3f + 1$ replicas are needed. Section 2.3.1 on the next page will, among other things, explain why $n = 3f + 1$.

2.3.1 Voting on HotStuff

HotStuff uses a voting system to perform the consensus. One replica is leader, which starts by proposing a new block to the replicas. If a replica did not already vote for another block on the same round, if all commands contained in the package are valid and if the proposal is valid (see Yin's paper to understand when a proposal is considered valid [Yin+19]), the replica sends a vote back to the leader, which can then generate a *quorum certificate* as soon as it has received a vote from more than $^2/_3$ of the replicas. The leader then starts a new round of voting by resending the proposal combined with the generated quorum. There are a total of three voting rounds per block, propose, pre-commit and commit. To understand why multiple rounds of votes are required, the reader can refer to Yin's paper. HotStuff chains the vote phases to increase performance. So the prepare phase for block 3 is also the pre-commit phase for block 2 and is also the commit phase for block 1. To be secure, the voting system should guarantee safety and liveness [Yin+19].

Liveness. Liveness means that correct replicas are guaranteed to make progress and is ensured if at least one quorum is generated in each voting round. Let v be the number of votes required per quorum. More than $f_{\text{liveness}} = n - v$ Byzantine-faulty replicas (e.g., DDoS attacked replicas) are needed for a successful liveness attack. Decreasing the number of votes needed for a quorum increases the number of Byzantine-faulty replicas needed for such an attack.

Safety. Safety means that all correct replicas agree on the same blockchain and thus on the same state and is ensured if maximum one quorum is generated in each voting round. Correct replicas will only vote for one quorum per round, Byzantine-faulty replicas on the other hand may vote multiple times in a single round. More than $f_{\text{safety}} = 2v - n - 1$ Byzantine-faulty replicas are needed for a successful safety attack.[1] Figure 2.1 shows graphically that a safety attack can succeed with at least $^1/_3$ Byzantine-faulty replicas. Increasing the number of votes v needed for a quorum would increase the number of Byzantine-faulty replicas needed for such an attack.

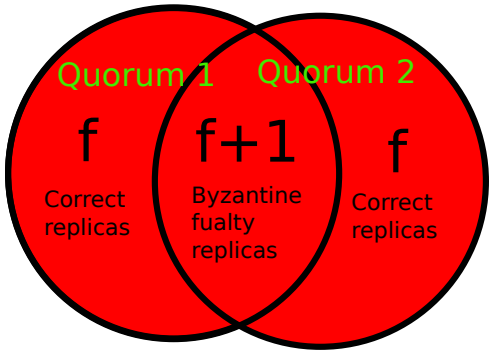


Figure 2.1. $\lceil \frac{1}{3}n \rceil = f + 1$ Byzantine-faulty replicas can succeed in a safety attack by voting for two different quorums.

Liveness and safety. $n = 3f + 1$ replicas and $v = 2f + 1$ votes per quorum (i.e., more than $\frac{2}{3}n$ votes per quorum) are required, as this is the optimum in which both liveness and safety are guaranteed with fewer than $^1/_3$ of Byzantine-faulty replicas and we can define $f \coloneqq f_{\text{liveness}} = f_{\text{safety}}$.

[1] This follows from following equation that calculates the number of Byzantine-faulty replicas $f_{\text{safety}} + 1$ needed to vote for 2 quorums:

$$\underbrace{1(n - (f_{\text{safety}} + 1))}_{\substack{\text{correct replicas} \\ \text{(vote for 1 quorum)}}} + \underbrace{2(f_{\text{safety}} + 1)}_{\substack{\text{faulty replicas} \\ \text{(vote for 2 quorums)}}} = \underbrace{2v}_{\substack{\text{votes required to} \\ \text{vote for 2 quorums}}}$$

2.3.2 Comparison of HotStuff and proof of work

This subsection will show the most important differences between permissioned and permissionless consensus algorithms by comparing HotStuff, the permissioned consensus algorithm used in this thesis, with proof of work, the most well known permissionless consensus algorithm, used for the cryptocurrency bitcoin (see Table 2.1).

Table 2.1. Comparison between HotStuff and proof of work

Definitions	HotStuff	Proof of work
number of replicas	constant	dynamic
required votes per quorum	more than $2/3$	-
Byzantine-faulty replicas	are less than $1/3$	have less than $1/2$ the of computing power

In permissioned consensus algorithms, the number of replicas must be constant and all replicas participating in the network must proof that they have the permission to participate by authenticating their messages. This is in contrast to permissionless algorithms like "proof of work" used by bitcoin, where the number of replicas can grow and shrink dynamically.

Permissionless consensus algorithms must be able to protect themselves against *Sybil attacks*, an attack in which the attacker creates a large number of Byzantine-faulty replicas to attack the system. To solve this problem, the influence of an attacker in proof of work does not depend on the amount of replicas (like with HotStuff), but on the amount of computing power available. HotStuff on the other hand can use a simple voting system that is less resource hungry.

HotStuff is secure as long as there are less than $1/3$ of Byzantine-faulty replicas, proof of work is secure as long as the Byzantine-faulty replicas have less than $1/2$ of the total computing power.

Chapter 3

Design

This chapter describes the design of the parallelized asset transfer system implemented in this thesis.

3.1 Terminology for permissioned asset transfer systems

In this section, the terminology and basic functioning of a permissioned asset transfer system is introduced in more detail, so that the parallelized system designed in this thesis may then be described using the terminology and mathematical symbols introduced here.

Permissioned asset transfer system. Let $S = (A, C, R, B)$ be a permissioned asset transfer system (see Figure 3.1) with:

- a set $A = \{a_i | 1 \leq i \leq m\}$ of accounts
- a set $C = \{c_i | 1 \leq i \leq k\}$ of clients
- a set $R = \{r_i | 1 \leq i \leq n\}$ of replicas
- a set $B = \{b_i | 1 \leq i \leq |B|\}$ of blockchain instances
- the system S using an account model to keep track of the balances

The system S is called *totally ordered* if it uses a single blockchain instance ($|B| = 1$) to totally order all the transactions.
The system S is called *parallelized* if it uses m blockchain instances ($|B| = m$), each instance b_i being responsible to totally order all outgoing transactions of a_i.
The system S is called *groupwise parallelized* if it uses $1 < |B| < m$ blockchain instances and a surjective function $f : A \to B$ that maps each account to a blockchain instance, each instance b_i being responsible to totally order all outgoing transactions of $f^{-1}(b_i)$.

Accounts and clients. Let $A = \{a_i | 1 \leq i \leq m\}$ be the set of accounts and $C = \{c_i | 1 \leq i \leq k\}$ be the set of clients. Each client is allowed to withdraw money from exactly one account, so a function $\mu : C \to A$ can be defined that maps each client to an account from which it can withdraw money. Then its fiber[1] $\mu^{-1} : A \to 2^C$ defined as $\mu^{-1}(a) := \{c \in C | \mu(c) = a\}$ is an owner map that maps each account to a set of clients that is allowed to withdraw money from that account.

Let there be an account $a \in A$ and a client $c \in \mu^{-1}(a)$. Then c may send money from account a to another arbitrary account $x \in A$ with $x \neq a$ by sending a transaction request to each replica.

Transaction request. Let $T = (c, a_i, a_j, x)$ be a transaction request from client c that transfers x assets from account a_i to a_j. The request is valid if $c \in \mu^{-1}(a_i)$.

[1] The fiber is the "inversion" of a non bijective function[Wik21a].

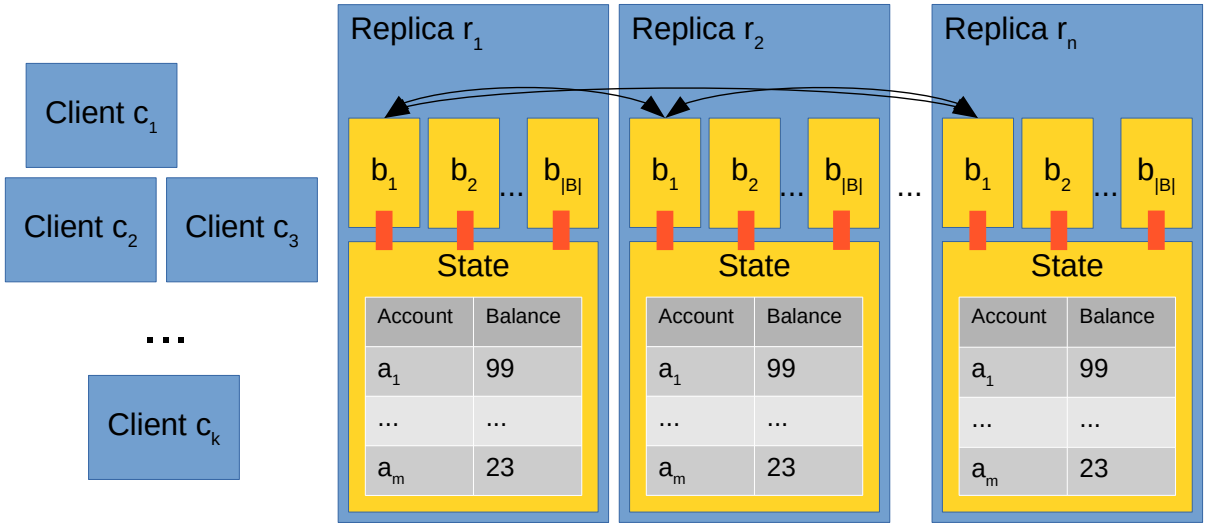


Figure 3.1. Permissioned asset transfer system

Replicas. Let $R = \{r_i | 1 \leq i \leq n\}$ be the set of replicas. Each replica $r_i \in R$ runs $|B|$ blockchain instances and has a single state. There can be a maximum of $f < \frac{n}{3}$ Byzantine-faulty replicas. The blockchain instances of all replicas run the HotStuff (permissioned) consensus algorithm to totally order the transactions. Each replica also saves the current balances of all accounts in the state.

Account model. To keep track of the account balances, cryptocurrencies can use different approaches. The most famous cryptocurrency, bitcoin, uses a model called UTXO (unspent transaction output) to model the transactions. With this model, the balances aren't directly saved, but a client can calculate its balance by adding all unspent transaction output belonging to him.

Etherum on the other hand, as well as the asset transfer system that is implemented in this thesis, uses the account model to keep track of the balances. This system is more intuitive, as, like banks, each replica has a state that keeps track of the balance of each account. After the blockchain instance has validated and ordered a transaction, the state will apply it (in the same order as ordered by the blockchain instance) by subtracting the transaction amount from the withdrawing account and adding it to the benefiting account. A client can then ask for the balance of an account by sending a request to all replicas and waiting for a response from at least $f + 1$ replicas (see Byzantine-faulty replicas). Of course, if replicas are at the moment executing transactions involving that account, the response might not be unanimous, even from correct replicas.

3.2 Transaction Request

Let $S = (A, C, R, B)$ be a parallelized permissioned asset transfer system (see definitions on Section 3.1, page 7) and let the accounts to clients owner map be defined as $\mu^{-1}(a_i) = \{c_i\}$ with $|A| = |C| = m$. Figure 3.2 summarizes the algorithm described in the rest of this section.

3.2.1 Sending a transaction

Let there be a client $c_i \in C$ and two accounts $a_i, a_j \in A$. with $a_i \neq a_j$. The client c_i wants to execute a transaction command $T = (c_i, a_i, a_j, x)$ of x assets (see "Transaction request" page 7) and therefore sends T to the blockchain instance b_i of all replicas (see Figure 3.2a on page 10). On each replica $r \in R$, blockchain instance b_i receives a transaction $T' = (c_{i''}, a_{i'}, a_{j'}, x')$ and first performs following checks:

- Blockchain instance b_i checks that $i' = i$. If this check passes, it means that the command has been sent to the correct blockchain instance.

- In a production implementation, client c_i would also digitally sign the request T. The blockchain instance b_i could then validate the transaction (check that $T' = T$) and check that $i'' = i'$ (or more generally check that $c_{i''} \in \mu^{-1}(a_i)$). This part was not implemented in this test implementation.
- It checks that $x' > 0$. An account owner should of course only be able to withdraw money from his account.
  - Note: If $x = 0$ and $j = i$, then the test implementation also accepts the command. As will be shown later, after the transaction has been executed, the replicas will respond to the client c_i by also sending the new balance of the account a_i. The implementation used in this thesis does not have a separate command to just request the balance, so a transaction request with $x = 0$ emulates such a command. Of course, a production-ready implementation should also have a separate more efficient command for that, as a consensus is not needed for a simple balance request.
- It checks that $j' \in \{1, ..., m\}$. If this check passes, the target account id is valid.

If the checks pass, b_i adds T in the queue of "decision waiting" transactions, otherwise b_i sends a negative response back to the client $c_{i''}$.

3.2.2 Ordering the transaction

Depending whether Replica r is a leader or not (for blockchain instance b_i), it behaves differently. If replica r is leader, it will read the first transaction commands from the "decision waiting" transactions queue and add them in the proposal for the next block.

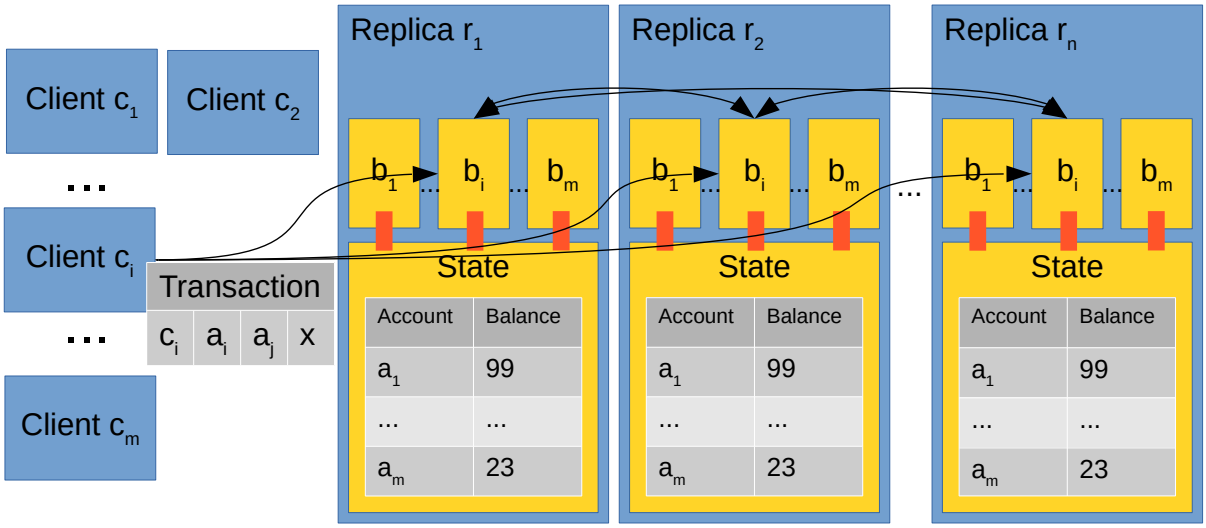
All other replicas should then, before they vote for the proposal according to the rules of HotStuff, check that they too have already added these commands to the "decision waiting" queue. This security check, which is important as the leader too could be Byzantine faulty, is needed to ensure that the proposed commands pass the checks described in Section 3.2.1 on the preceding page. It is not possible for the replicas to instead directly validate the commands in the proposal, as the proposal from the leader contains only the command hashes, but replicas need the full commands to check their validity. Note: In the current implementation, the replicas do not check that the command is in the "decision waiting" queue before voting. This is something that will need to be done in a production environment though.

According to the HotStuff protocol, the replicas will then finish to validate and totally order the block proposed by the leader. Once the block has been added to the blockchain, for each command hash in the block, the corresponding command in the "decision waiting" queue will be searched. If the command is found, it will be added at the end of the *response queue*. If the command is not found, a production-ready implementation should have a mechanism that waits for a short amount of time, as this replica r may receive the command with a certain delay, and that sends a request to the other replicas to get the command if it did not arrive within the specified delay. This mechanism is however not implemented here, so the transaction will simply not be executed in this case. This makes the system prone to state corruption at any time, but does not affect the benchmarks in a significant way, as a replica only needs to wait for f+1 responses to validate a request and create an entry for the benchmarks.

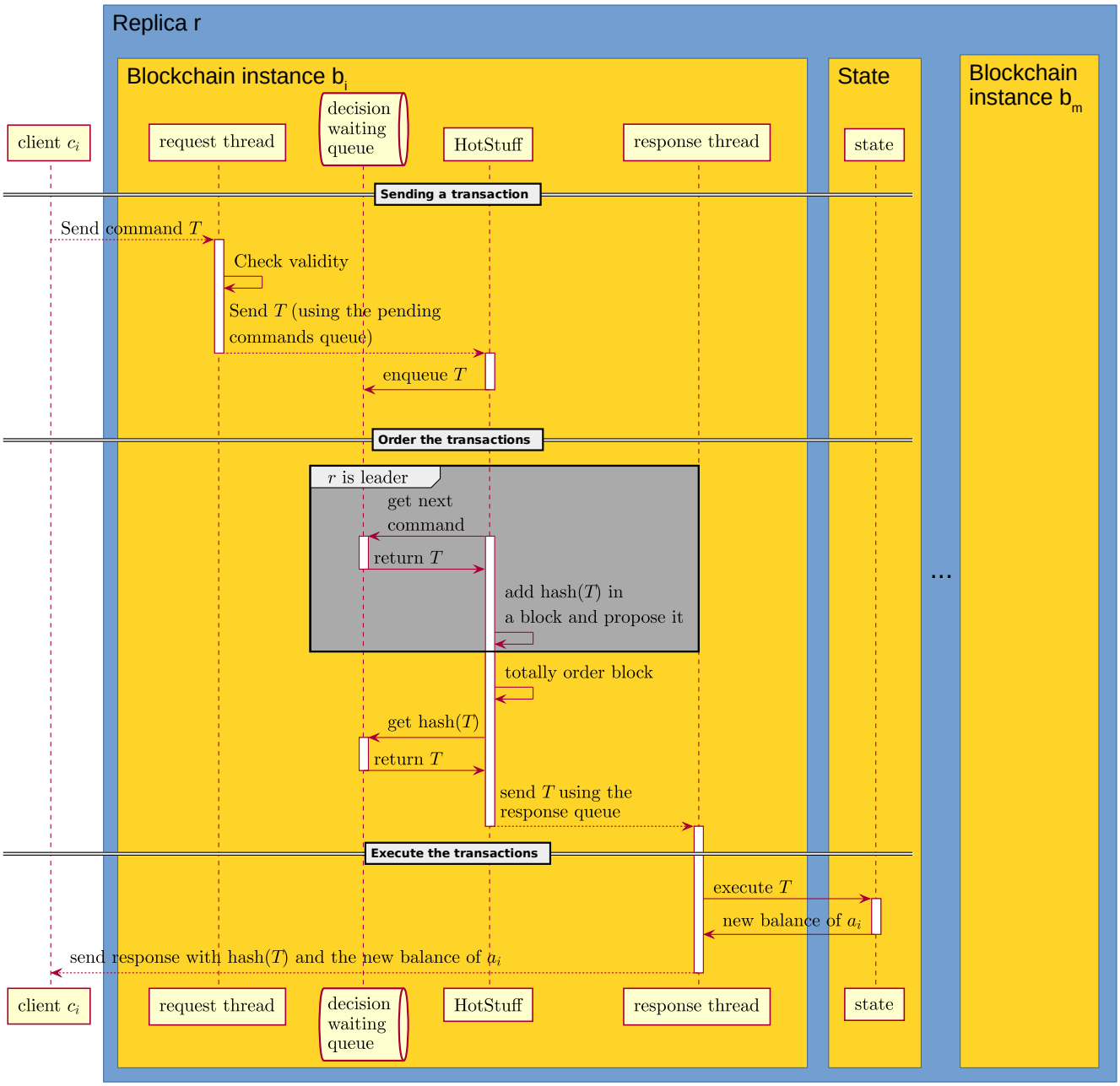
3.2.3 Executing the transaction

As explained before, the commands issued by account a_i are added to the response queue of b_i in the same order in each replica, depending on how they were ordered in HotStuff. A separate thread will then read one command at a time from the queue, execute the command and send a response to the client c_i containing the hash of T and the balance of a_i after the execution of the transaction request. Here is how command $T = (c_i, a_i, a_j, x)$ is executed (see Figure 3.3 on page 11):

1. Withdraw x from a_i



(a). Client c_i sends transaction $T = (c, a_i, a_j, x)$ to all replicas $r \in R$.



(b). Blockchain instance b_i on replica r fulfills the request of client c_i.

Figure 3.2. Parallelized permissioned asset transfer system $S = (A, C, R, B)$ with $|A| = |B| = |C| = m$ and $|R| = n$.

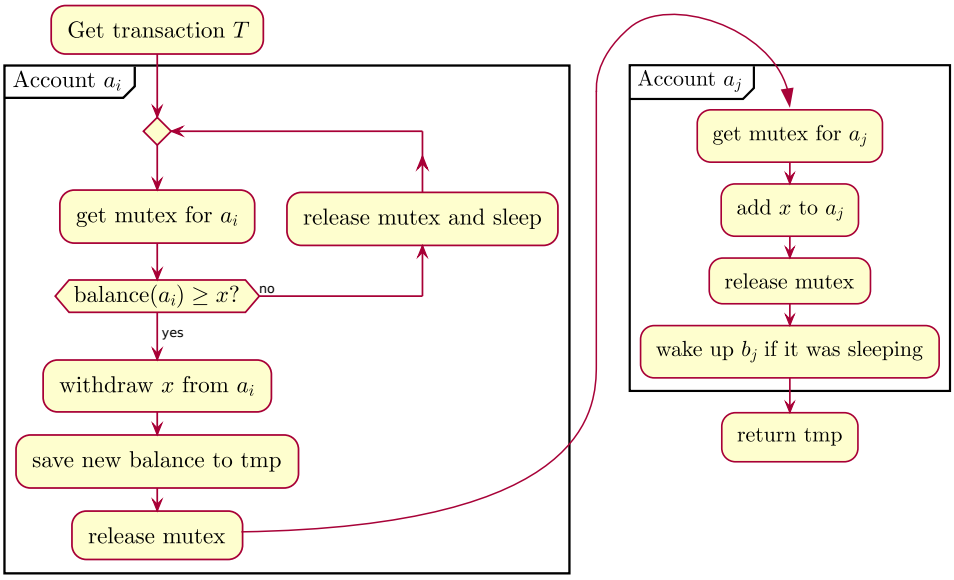


Figure 3.3. Blockchain instance b_i applies transaction $T = (c, a_i, a_j, x)$ to the state of a replica r

(a) Blockchain instance b_i gets a mutex for a_i. As on a single replica r, there are m different blockchain instances running that can access the state simultaneously, the balance variables of the accounts need to be protected by a mutex. Mutexes can only be owned by one process at a time, so acquiring it before the balance is accessed makes sure that there are not any race conditions on that variable.

(b) If $\text{balance}(a_i) < x$, b_i releases the mutex again and goes to sleep. When a correct client c has sent T, this can happen when a transaction that T depends on was not yet executed on this replica r. When another process b_x with $x \neq i$ on the same replica r has added money to this account (i.e., a transaction on which T depends has been executed), it will wake up b_i. Instance b_i will then reacquire the mutex for a_i and continue as described on item (1a). Of course, if c_i is not correct, no transaction adding money to a_i may occur, and b_i may sleep indefinitely. This causes nonetheless no security risk, as the author c_i of such an attack may only block his own account a_i by trying to send transactions where $\text{balance}(a_i) < x$.

(c) If $\text{balance}(a_i) \geq x$, b_i withdraws x from a_i, saves the new balance in a temporary variable and releases the mutex. Saving the new balance in a temporary variable will allow to return the new balance of a_i at the end of the function without the need to reacquire the mutex.

2. Add x to a_j

(a) Blockchain instance b_i gets a mutex for a_j.

(b) As x is added and not withdrawn from a_j there is no need to check that a_j has enough money. Amount x is simply added to account a_j and then the mutex of a_j is released.

(c) Wake up blockchain instance b_j on the same replica r if it was put to sleep in item (1b) (i.e., if there is a transaction in b_j that depends on this transaction T to be executed).

3. Return the new balance of a_i by returning the temporary variable created in item (1c).

3.2.4 Validating the response

Every client has a list of pending requests $\mathcal{P} \subset 2^P$ with $P = (T, v, e, N, W)$ such that:

- $T = (c_i, a_i, a_j, x)$ is the transaction sent.

- $v \in \mathbb{N}$ is the number of positive responses (transaction *valid*).
- $e \in \mathbb{N}$ is the number of negative responses (*error* messages).
- $N \subset \mathbb{N}$ is the new account balance of a_i as reported by the positive responses. N is a list of balances and not a single balance as different responses may report different balances, even if all responses are from correct replicas.
- $W \subset R$ is a list of replicas that have already sent the response.

Each correct replica r should eventually send a response $Q = (h, u, d)$ back to the client c_i, as explained in Section 3.2.3 on page 9, such that:

- h is the *hash* of the transaction request that Q responds to.
- $u \in \mathbb{N}$ is the balance of a_i after that the transaction represented by hash h has been executed by the state of replica r.
- $d \in \mathbb{Z}$ represents the *decision* taken by replica r. If $d = 1$, it means the transaction represented by hash h was executed successfully.

When client c_i receives a response Q, it tries to find a pending request $P = (T, v, e, N, W) \in \mathcal{P}$ such that $\mathit{hash}(T) = h$. If a corresponding pending request P is found, the following is executed, if not, the response is simply ignored:

- Client c_i should ignore the response Q if its signature does not correspond to a known replica $r \in R$ or if $r \in W$. In the first case, an attacker tried to forge a response. In the second case, an attacker resent a formerly valid response (replay attack). If the check passes, the replica should be added to W ($W = W \cap \{r\}$). As the response is not signed in this test implementation, this step is ignored.
- If the decision is positive ($d = 1$):
  - The counter v is incremented.
  - The reported new balance is added to the list of new balances: $N = N \cap u$
  - If $v > f$ (there are more positive responses than the maximal number of Byzantine-faulty replicas), then a new statistics entry is created reporting when the request was sent and how long it took for the request to be fulfilled. These statistics will be used in Chapter 5 for the benchmarks. Client c_i also prints a log entry containing the command hash and all possible new balances N of a_i, and finally deletes P from the pending requests List $\mathcal{P}$.
- If the decision is negative ($d \neq 1$):
  - The counter e is incremented
  - If $e > f$ (there are more negative responses than the maximal number of Byzantine-faulty replicas), then a warning is printed in the log file and P is deleted from the pending requests list $\mathcal{P}$.

3.3 Possible advantages and disadvantages of a parallelized asset transfer system

In this section, we will discuss what possible advantages and disadvantages there could be for running a parallelized asset transfer system. Chapter 5 on page 19 will then show the actual measured performance differences between the totally ordered and the parallelized asset transfer system and discuss them.

3.3.1 Advantages

- In HotStuff, the leader is doing the most work. Whereas the other replicas just receive one message and have to send a single vote per voting round, the leader also sends n messages (one to each replica including itself) and receives n votes (including from itself), which is network intensive. Additionally, the leader must combine the votes to create a quorum certificate, which is CPU intensive. With the parallelized version, the leader roles of the different blockchain instances, and thus the load, could be evenly split among all replicas, which would also result in more evenly distributed network requests.
- As the transaction requests from different accounts are executed in parallel, this could result to less latency for the same throughput.

3.3.2 Neutral

- The amount of messages that need to be exchanged between the replicas for each validated block does not change
- The history of transactions is not perfectly reconstructible. Only if timestamps are stored in each blocks, it is possible to get a rough idea. However, this is not necessary to guarantee security, as the current state can always be reconstructed by reexecuting the transaction requests of all blockchain instances.

3.3.3 Disadvantages

- m times more network connections need to be maintained.
- If each account transmits only few transactions per minute, it may be too costly to run only one account per blockchain instance. To mitigate this issue $S = (A, C, R, B)$ can be declared as being groupwise parallelized (instead of parallelized) with $f : A \to B$ being declared as $f(a_i) = b_{(i \mod |B|)}$. With this technique, m accounts are evenly distributed between $|B|$ different blockchain instances. This method can also be used to easily increase the number of accounts in a running system without having to create new blockchain instances.

  If some accounts are expected to send way more transactions than others, the distribution of the accounts among the blockchain instances could also be done using a weight function $g : A \to \mathbb{N}$, with $g(a_i) = x \cdot g(a_j)$ meaning that a_i is expected to send x times more transactions than a_j. Each blockchain instance b would also have a weight initially defined as $G(b) = 0$. The accounts A are reordered in such a way that $g(a_1) \geq g(a_2) \geq ... \geq g(a_m)$. Then, for i from 1 to m:

  - Find a b_j such that $G(b_j) = \min\{G(b_1), ..., G(b_{|B|})\}$
  - Add a_i to the list of accounts b_j is responsible for: $f(a_i) \coloneqq b_j$
  - Add the weight of a_i to b_j: $G(b_j) \coloneqq G(b_j) + g(a_i)$

  Note that the algorithm does not give the best possible result, but instead uses a heuristic called *greedy number partitioning*, as the best possible result would be NP complete. The problem to solve, *Multi-Way number partitioning*, is similar to the more well-known problem *bin packaging* [KSM14] [Kor09].

Chapter 4

Implementation

4.1 Pseudocode

Algorithm 1 on the next page summarizes how a transaction request is executed. Whereas Figure 3.2 on page 10 focuses on distinguishing the entities and threads that are responsible for each part of the algorithm, Algorithm 1 is a higher level overview that presents only the logic actually implemented in the test implementation. The simplifications made for the test implementation are explained in detail in Section 3.2 on page 8 and are resumed in Chapter 6 on page 27.

4.2 Programming languages used

The HotStuff implementation used in this thesis was first implemented in C++ by Ted Yin. All extensions to this codebase needed to implement the parallel blockchain instances and the state were also implemented in C++. The script used to administer and benchmark HotStuff was implemented in Bash. Some scripts used by this script were implemented in Python.

4.3 System implemented

The permissioned asset transfer system $S = (A, C, R, B)$ implemented in this thesis is parallelized with the accounts to clients owner map being $\mu^{-1}(a_i) = \{c_i\}$, as defined in Section 3.2 on page 8. The identity mapping between clients and accounts simplifies the configuration script. The implementation of the server and client however does not rely on this simplification and could therefore also run parallelized systems with any owner map $\mu^{-1} : A \to 2^C$, as long as the configuration script is modified accordingly. However, to run a groupwise parallelized system, the implementation of the replica would also need to be modified, as for now each blockchain instance can only be mapped to exactly one account.

The configuration script is nonetheless able to launch the replicas and clients in a totally ordered way (only one blockchain instance), if the number of accounts is set to one, as in this case it also satisfies the definition of a parallelized asset transfer system. This trick can be used to compare the newly created parallelized system with a totally ordered system in Chapter 5. When the system is started in the totally ordered mode, it is started with the same number of clients, but with only one account. Consequently, there is only one blockchain instance, and all clients send pseudo transfer requests of 0 coins from a_1 to a_1. To summarize, the asset transfer system may be run in two different ways:

- parallelized with an identity mapping between account ids and client ids
- totally ordered: one blockchain instance, one account a_1 with $\mu^{-1}(a_1) = C$ and all clients sending pseudo transaction requests of 0 coins from a_1 to a_1.

Algorithm 1 Process transaction request on blockchain instance b_i.

Variables
    decision_waiting $\leftarrow [\perp]$: searchable queue of transactions

operation *transfer*$(T = (c, a_i, a_j, x))$ on client $c \in \mu^{-1}(a_i)$
    boradcast(T, b_i)

upon *deliver*(T') on blockchain b_i and replica r **do**
    let T' be $(c, a_{i'}, a_j, x)$
    if $(i' = i)$ **and** $(x > 0$ **or** $(x = 0 \wedge i = j))$ **and** $(j \in \{1, ..., m\})$ **then**
        decision_waiting.*add*(T')
    else
        respond("rejected")

upon *decision_waiting* is non-empty if r is leader **do**
    blk $\leftarrow$ new block
    repeat *size*(*blk*) **times**
        blk.*add*(*decision_waiting*.*remove*())
    propose(*blk*)

upon Block *blk* is totally ordered **do**
    for each command hash *hash* **in** *blk* **do**
        let T be *decision_waiting*.*get*(*hash*)
        new_balance $\leftarrow$ *execute*(T)
        respond(*new_balance*)

upon *execute*(T) on blockchain b_i and replica r **do**
    let T be (c, a_i, a_j, x)
    get_mutex(a_i)
    while $x >$ *balance*(a_i) **do**
        release_mutex(a_i)
        sleep(b_i)
        get_mutex(a_i)
    balance_a$_i \leftarrow$ *balance*$(a_i) - x$
    new_balance $\leftarrow$ *balance*(a_i)
    release_mutex(a_i)
    get_mutex(a_j)
    balance_a$_j \leftarrow$ *balance*$(a_j) + x$
    release_mutex(a_j)
    wakeup(b_j)
    return *new_balance*

4.4 Configuring the system

Every blockchain instance on every replica is started with a different combination of configuration files. The configuration script generates all needed configuration files in a directory structure:

${i} Can be any natural number between 1 and $|B|$ (Remember: $|A| = |B| = |C| = m$ if the system runs in the parallelized mode and $|C| = m, |A| = |B| = 1$ if the system runs in the totally ordered mode.).

${j} Can be any natural number between 1 and $n = |R|$

parallel_conf/ Main directory containing all configuration.

parallel_conf/account_${i-1} Directory containing all configuration files for blockchain instance b_i.

parallel_conf/account_${i-1}/hotstuff.gen.conf Main configuration file for b_i.

parallel_conf/account_${i-1}/hotstuff.gen-sec${j-1}.conf Blockchain instance b_i on replica r_j will use this private key to sign the messages.

parallel_conf/account_${i-1}/clients.txt Ip addresses of all clients $c \in C$ where $f(\mu(c)) = b_i$. So, if the system runs in the parallelized mode, this file just contains the ip address of client c_i. If the system runs in the totally ordered mode, the file contains the ip addresses of all clients, as all clients belong to account a_1. This file is used by the configuration script to connect to the clients.

parallel_conf/account_${i-1}/nodes.txt Ip addresses of all replicas followed by client and server ports used by blockchain instance b_i of the respective replica. This file is used by the configuration script to connect to the replicas.

The configuration script sends to each replica $r_j \in R$ the configuration files that it needs:

- the main configuration files of all blockchain instances "parallel_conf/account_*/hotstuff.gen.conf"
- the private cryptographic keys of replica r_j "parallel_conf/account_*/hotstuff.gen-sec${j-1}.conf"

Then it sends to each client $c_i \in C$ the configuration files that it needs:

- If the system runs in the parallelized mode, it sends the main configuration file of blockchain instance b_i "parallel_conf/account_${i-1}/hotstuff.gen.conf"
- If the system runs in the totally ordered mode, it sends the main configuration file of blockchain instance b_1 "parallel_conf/account_0/hotstuff.gen.conf"

4.5 Managing automatic administration of remote replicas

The system can be administrated using a single script. In particular, this script is able to:

- generate automatically configuration files for all replicas and clients of the system.
- automatically start, stop or initialize all replicas and clients of the system, and check if they are running or dead.
- run the system in the parallelized or totally ordered mode, locally or on digitalocean[1]
- start and stop clients and servers in parallel to save time when programming.
- copy log files back and run benchmarks on them
- selectively restart only the servers or clients that are dead.

[1] Cloud infrastructure provider: www.digitalocean.com

4.6 Encountered difficulties

This section exposes some of the biggest difficulties I had as a bachelor student working for the first time on a real codebase when implementing and benchmarking the system.

4.6.1 Understanding the codebase

The first difficulties appeared right at the beginning of my thesis, when confronted for the first time with a real codebase, the HotStuff implementation of Ted Yin. I learned how to navigate in a codebase to understand which parts were important for me to understand, and ultimately which parts needed to be modified. The comprehension of the codebase was made more difficult by:

- the lack of almost any documentation, functionwide, classwide or filewide.
- the use of custom made classes even for some low level utilities that already have an implementation in the standard library. For example, instead of ”std::shared_ptr”, the custom class ”ArcObj” was used for the same purpose.

4.6.2 Multithreading

Multithreading is a complex subject. Many points need to be considered when implementing a multithreaded system or when converting a codebase that was originally not planned for multithreading. In particular, I had to learn:

1. how to use condition variables in C++ correctly to be able to put a thread to sleep and wake it up from another thread when the conditions to resume the thread are possibly met (see Section 3.2.3 on page 9)
2. to remove global variables if they contain a state, as leaving them could lead to race conditions when accessing the state.
    - In particular, sometimes it is possible to remove global variables by replacing them with thread-local variables. Thread-local variables are like global variables, but they have different contents on each thread. At the beginning, the logger was global, so all blockchain instances logged on the same file, making it incomprehensible. By defining the logger thread-local, it was possible to save the log of each blockchain instance on a different log file.
    - A very troublesome bug finally revealed itself as being caused by a static buffer variable declared *inside* a serialization function called when serializing the quorum certificate. Such a variable exists only once: Every object and every function call accesses the same variable. As all blockchain instances serialize quorum certificates, the bug occurred whenever two blockchain instances executed the function simultaneously and was resolved by declaring the buffer as automatically allocated in the stack (removing the static keyword). Declaring a non-constant variable in a class as statically allocated is a bad practice when, like in this case, there is no obvious reason to do so, as different objects of the same class should be independent of each other.

When run in the parallelized mode, some replicas may crash at startup without any log entry, which does not create further problems, as the replica can simply be restarted in this case. The root cause of this issue however has not yet been discovered.

To be able to run 10 parallel blockchain instances per replica, the maximum number of allowed file descriptors needs to be raised at program startup. This number is by default 1024 on Linux. Probably the network library used, salticidae, needs to open lots of file descriptors.

All these difficulties were challenging but at the same time they made my work exciting and allowed me to expand my knowledge.

Chapter 5

Benchmarks

In this chapter, the totally ordered and parallelized system are compared by means of throughput-latency diagrams.

5.1 Setup

Tests were run with 10 clients and 10 replicas, all running in a data center on the same LAN[1], respectively in the totally ordered and parallelized configuration (see Section 4.3 on page 15), and with block sizes of 50, 200, 400, 600 and 800 commands. The results may be seen in Figure 5.1 on page 23. Each point corresponds to a 60 seconds test. Each diagram has two series of tests, one totally ordered and one parallelized.

Before each test series, the replicas are started and may stay turned on for multiple 60s tests, as it is not needed to restart them every time. Every test follows this procedure:

- The clients are started
- In the parallelized mode, the clients send random transaction requests (random amount x of money to random account a_j such that $a_i \neq a_j$) for 60s. In the totally ordered mode, the clients send transaction requests of 0 coins from a_1 to a_1 for 60s. The maximum number of parallelly sent transaction requests is increased for every test in a test series.
- The clients are asked to stop
- Before halting, the clients print a list with timestamp and latency of every transaction executed in their log file.
- All logs are downloaded to the server running the benchmarks.
- The logs are analyzed and the diagrams are generated.

Because the tests shall represent the best possible case where there is always enough money on an account to perform a transaction, and as the bug described in Section 3.2.2 on page 9 may corrupt the state and thus result in unpredictable balances, the balance check (see Section 3.2.3 on page 9) is deactivated for the tests, to allow the account balances to drop below zero.

Apart from errors and warnings, in the replicas all logs were deactivated for the tests, as writing millions of lines in a file is a bottleneck. Clients additionally write info log lines, as they are used to generate some statistics (see Section 5.2 on the next page).

[1] Cloud infrastructure provider: www.digitalocean.com

5.2 Verify validity

After each test has run, some general statistics are generated by parsing the log files of all clients and replicas, so that most errors that could invalidate a test may be easily recognized. Generated statistics include:

- Number of "Could not find cmd" warnings. These are warnings that are generated each time a transaction could not be executed (see end of Section 3.2.2 on page 9). Such warnings are expected in this test implementation, but the system was restarted when the count raised too much (above ca. 1'000'000) to exclude any possible slowdowns caused by that.
- Number of "network async error" warnings. These errors are normal, especially at the beginning when the replicas are establishing the connections, but there should not be too many of them.
- All other warnings are appended verbatim to the statistics.
- For each client, the number of executed transactions. All clients should have a number of the same magnitude. If this is not the case, it may indicate a failure of a particular blockchain instance or client.
- If the number of "could not find cmd" warnings is 0, the balances of all clients are collected by sending transactions of 0 coins (needs info log lines of the clients) and added together to verify that no money was destroyed or generated. The condition to run the test (no "could not find cmd" warnings) was rarely met, but the test passed when it was run.

5.3 Results

As can be seen in Figure 5.1, with the same block-size, the parallelized system is always faster and has a higher latency than the totally ordered system. When increasing the block-size, the throughput and latency generally increase. But whereas the latency increases continuously, from a certain block-size on the throughput hardly increases anymore. The parallelized system reaches the best possible throughput with a smaller block-size than the totally ordered system. The rest of this section will try to explain why.

5.3.1 Latency

As an approximation, the latency was empirically defined as follows:

$$l(|B|, \textit{throughput}, \textit{block-size}) = \frac{4.87 \cdot |B| \cdot \textit{block-size}}{\textit{throughput}} \tag{5.1}$$

A block with a bigger block-size takes longer to fill, hence increasing the latency. With a higher throughput, blocks are filled faster, hence reducing the latency. More blockchain instances means that blocks are filled slower, as the throughput must be shared among all blockchain instances, hence increasing the latency. Equation 5.1 was confirmed to be a good model of the measured latency and the constant could be defined as 4.87.

Table 5.1 on the next page shows the most representative test of each test series, i.e., one of the highest throughput tests before the latency spikes. For example, for a block-size of 50 in the totally ordered mode, test number 5 is the most representative test. Often there is more than one sensible choice, so in these cases they were chosen in the most consistent way possible. The 6th column (predicted latency) shows the latency as calculated with Equation 5.1. The last column shows the percentage error between the measured latency and the predicted latency. Apart from two exceptions, the error is always in the +-5% range.

Table 5.1. The table shows the most representative test of each test series, i.e., one of the highest throughput tests before the latency spikes.

Block-size	Mode	Test number	Throughput (Kops/sec)	Measured latency (msec)	Predicted latency (msec)	Percentage error
50	totally ordered	5	13.2	18.2	18.5	+1.9%
50	parallelized	5	50.4	43.7	48.3	+10.7%
200	totally ordered	12	36.7	27.6	26.6	-3.7%
200	parallelized	10	94.6	105.0	103.0	-1.9%
400	totally ordered	10	52.6	37.0	37.0	+0.0%
400	parallelized	9	98.2	195.0	198.3	+1.7%
600	totally ordered	12	58.8	51.6	49.7	-3.7%
600	parallelized	7	98.6	283.0	296.5	+4.8%
800	totally ordered	15	59.6	74.9	65.3	-12.8%
800	parallelized	6	103.0	368.6	378.1	+2.6%

5.3.2 Throughput

Each transaction can be associated with an execution cost, that represents, among others, CPU and network load. Reducing the cost per transaction would increase the total throughput of the system. Let x be the cost to totally order a block and y be the individual cost of a command. Then the cost per transaction T can be defined as:

$$c(T) = \frac{x}{\textit{block-size}} + y \tag{5.2}$$

The cost per block x is much higher than the individual cost of a transaction y, so at the beginning increasing the block-size strongly decreases $c(T)$ and thus increases the throughput. But from a certain block-size the first term of Equation 5.2 becomes negligibly small, so $c(T) \approx y$. From then on, it is useless to further increase the block-size. This model can explain the general trends observed.

However, it does not explain why the parallelized system reaches the best possible throughput with a smaller block-size than the totally ordered system. One possible explanation for that could be the increase of latency. Indeed, a higher latency means that the system idles for longer periods of time, wasting system resources until enough transactions have arrived to fill a block. So if the resources wasted because of the higher latency compensate the resources gained because of the lower cost per transaction $c(T)$, increasing the block-size has no effect. As the latency increase is much higher for the parallelized system than for the totally ordered system, this could explain why the parallelized system reaches the best possible throughput with a smaller block-size.

5.3.3 Ratio throughput latency

The parallelized system does not improve nor worsen the ratio between latency and throughput (which disproves the second point in Section 3.3.1 on page 13): As can be seen, a parallelized system with a block-size of 50 has approximately the same latency and throughput as a totally ordered system with a block-size of 400.

5.3.4 Global performance improvement

As defined is Section 5.3.2, from a certain block-size on the throughput hardly increases anymore. Further throughput increase is considered negligible when the latency increase is more than 10 times the throughput increase[2]. This definition is subjective and depends on how much throughput is valued over

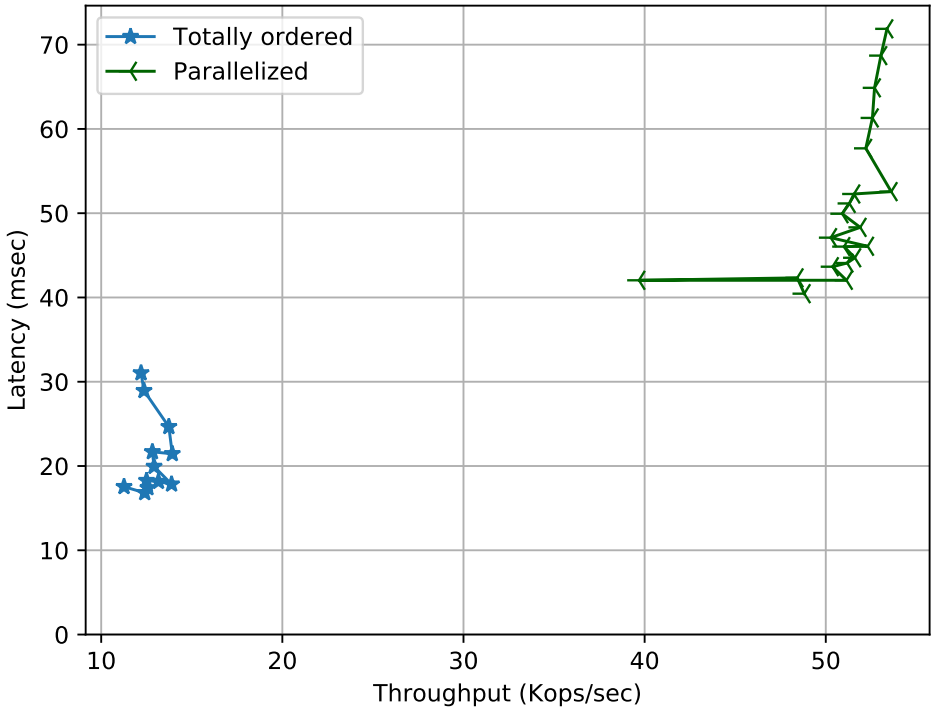
[2]alternative definition: when the throughput increase is less than 10 times the throughput increase measured between block-size 50 and 200.

latency. Based on the above definition, the parallelized system reached its maximal thoroughput of 94.6 Kops/sec with a block-size of 200 and a latency of 105.0 msec, whereas the totally ordered system reached its maximal throughput of 58.8 Kops/sec with a block-size of 600 and a latency of 51.6 msec (see Table 5.1 on the preceding page). Consequently, the parallelized system has a 60,9% better throughput and a 103,5% worse latency compared to the totally ordered system.

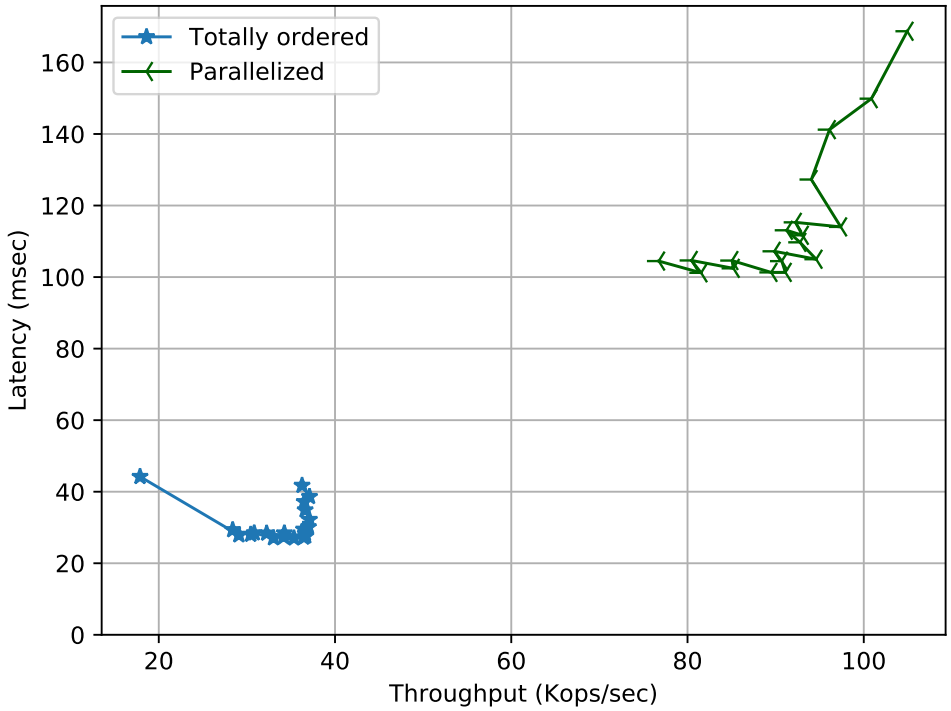
5.4 Further tests

The results seem promising, but may benefit from further tests to confirm them, and in particular, to see if it is possible to decrease the latency or further increase the throughput. The following further tests may be run:

- Tests with a varying number of blockchain instances. These tests may be useful to find an optimum between latency increase and throughput increase.
- 10 replicas, but only 1 client. This test would be helpful to better understand how the number of clients influence the speed.
- Tests with completely deactivated logs. At the moment, there are still lots of warnings and the clients still generate lots of log lines. A complete deactivation could fasten the system a bit further.
- Tests with a varying number of clients per blockchain instance, not only one. Maybe the maximum throughput is reached only when there is more than one client per blockchain instance.
- Tests that also measure cpu and network usage:
  - to detect bottlenecks
  - to guarantee that the bottleneck is not the result of a programming mistake.
  - to prove or disprove the first point in Section 3.3.1 on page 13.

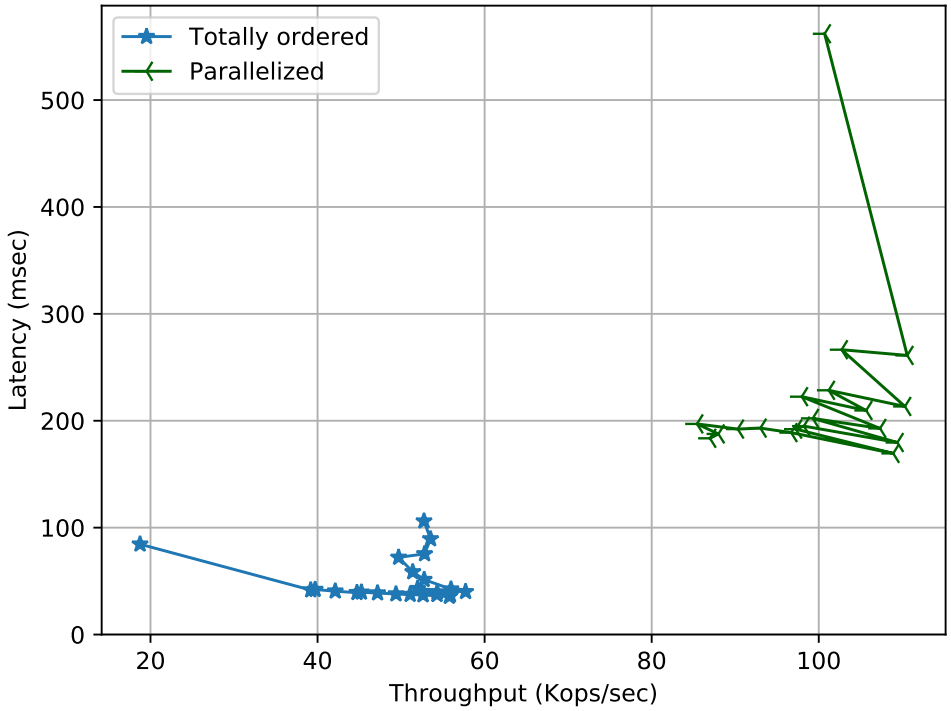


(a). Block-size 50

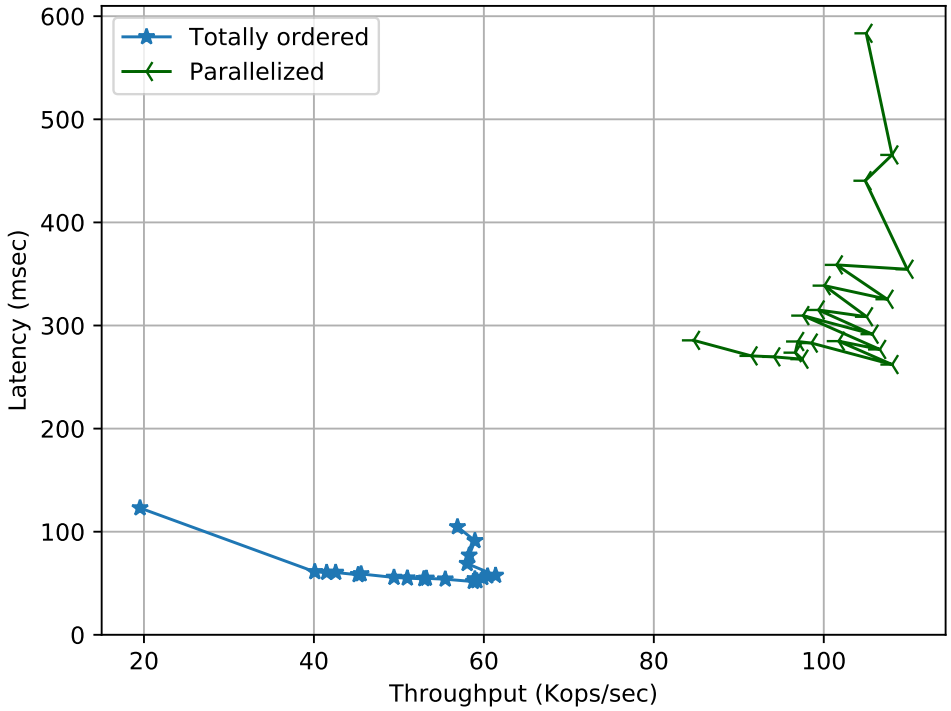


(b). Block-size 200

Figure 5.1. The figure shows throughput latency diagrams for different block sizes. Each diagram has two series of tests, one totally ordered and one parallelized. Each point corresponds to a 60 seconds test.

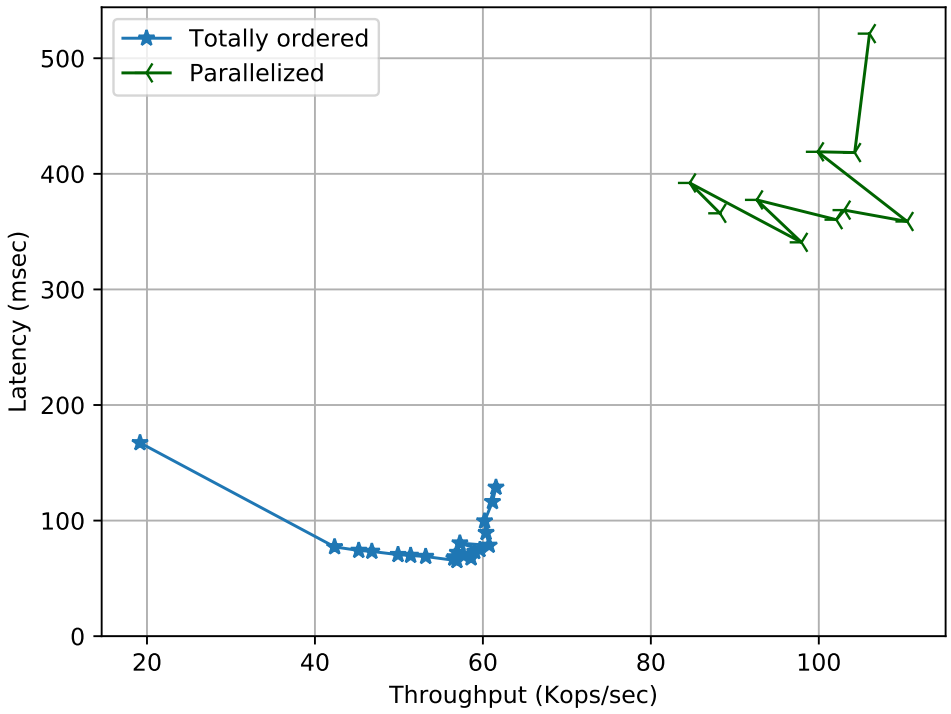


(c). Block-size 400



(d). Block-size 600

Figure 5.1. The figure shows throughput latency diagrams for different block sizes. Each diagram has two series of tests, one totally ordered and one parallelized. Each point corresponds to a 60 seconds test.



(e). Block-size 800

Figure 5.1. The figure shows throughput latency diagrams for different block sizes. Each diagram has two series of tests, one totally ordered and one parallelized. Each point corresponds to a 60 seconds test.

Chapter 6

Conclusion

In this thesis, a parallelized asset transfer system using HotStuff as a consensus algorithm was implemented that has a 60,9% better throughput (and a 103,5% worse latency) than the totally ordered asset transfer system it was compared with.

The results have shown that parallelization can indeed be used to increase the throughput of an asset transfer system, albeit at the expense of a higher latency. Nonetheless, as higher throughput is often more valuable than latency improvements, and as the parallelized system does not seem to worsen the ratio between throughput and latency either, the results are promising and further research in this field may certainly be worthwhile.

There is lots of exciting further research, programming and measuring that can be done to ameliorate this prototype or the benchmarks, mainly (in decreasing order of importance):

- Implement both algorithms in Section 3.2.2 on page 9 that are not yet implemented in this prototype, but are strictly necessary in a productive environment.
- Run further tests (see Section 5.4 on page 22).
- Implement client authentication on the servers (see Section 3.2.1 on page 8) and server authentication on the clients (see Section 3.2.4 on page 11).

Bibliography

[Gue+19] Rachid Guerraoui et al. “The Consensus Number of a Cryptocurrency”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 307–316. DOI: 10.1145/3293611.3331589. URL: https://doi.org/10.1145/3293611.3331589.

[Kor09] Richard E. Korf. “Multi-Way Number Partitioning”. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 538–543. URL: http://ijcai.org/Proceedings/09/Papers/096.pdf.

[KSM14] Richard E. Korf, Ethan L. Schreiber, and Michael D. Moffitt. “Optimal Sequential Multi-Way Number Partitioning”. In: *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2014, Fort Lauderdale, FL, USA, January 6-8, 2014*. 2014. URL: http://www.cs.uic.edu/pub/Isaim2014/WebPreferences/ISAIM2014_Korf_etal.pdf.

[Sch90] Fred B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319. DOI: 10.1145/98163.98167. URL: http://ecommons.library.cornell.edu/bitstream/1813/6640/2/86-800.ps.

[Wik21a] Wikipedia, ed. *Fiber (mathematics)*. Aug. 19, 2021. URL: https://en.wikipedia.org/w/index.php?title=Fiber_(mathematics)&oldid=1039563985.

[Wik21b] Wikipedia, ed. *State machine replication*. Apr. 19, 2021. URL: https://en.wikipedia.org/w/index.php?title=State_machine_replication&oldid=1018712180.

[Yin+19] Maofan Yin et al. “HotStuff: BFT Consensus with Linearity and Responsiveness”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 347–356. DOI: 10.1145/3293611.3331591. URL: https://doi.org/10.1145/3293611.3331591.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Wohlenfen, der 30.08.2021 — Ort/Datum

J. de Faveri — Unterschrift