École Polytechnique Fédérale de Lausanne

From Broadcast to Database; a Merkle-based approach to Universal Key-Value Stores

by Manuel Vidigueira

Master Thesis

Approved by the Examining Committee:

Prof. Rachid Guerraoui
Thesis Advisor

Maxime Monod
External Expert

Matteo Monti
Thesis Supervisor

EPFL IC DCL
INR 311 (Bâtiment INR)
Station 14
CH-1015 Lausanne

June 18, 2020

A family is not a group of relatives; it is more than the affinity of blood, it should also be an affinity of temperament. A man of genius often does not have a family. They have relatives.
— Fernando Pessoa

Dedicated to my family and relatives.

Acknowledgments

I would like to thank all of those that have helped me with this project. I thank my supervisor, Matteo Monti, for suggesting this topic and whose previous work this project is based on, as well as for providing valuable guidance during its conception and when reviewing this thesis. My adviser, Prof. Rachid Guerraoui, who gave me this opportunity and who kept in touch throughout. I thank all of my friends who have helped me review this thesis, in particular Hugo, Marco, and Kiru for taking the time to do a complete review thesis despite their busy schedules. I would also like to thank Mathias Payer for allowing me to use this template and all of its useful hints. Finally, I would like to thank my family, in particular my parents, grandparents, and sister, for their constant support before and throughout this project. It is because of them that I am able to travel to and study here at EPFL in the first place, and why I have the freedom of mind to fully dedicate myself to this project.

Lausanne, June 18, 2020 Manuel Vidigueira

Abstract

Decentralized databases rely on participant nodes keeping track of the state of the system to validate transactions. As the database grows in size, the capacity of individual nodes becomes a limiting factor if each node has to store the complete state. While some solutions manage to solve this problem in the context of a cryptocurrency, there is little completed work in generalizing them to support generic data and operations.

In this project, we design and implement a distributed, universal database for the Byzantine model that splits state across nodes while preserving security. Furthermore, it can store and execute opaque, user-defined transactions on any type of data. This is achieved by relying on client-driven communication and Merkle tree-based cryptography. By extending Merkle prefix trees to keep track of recent changes, we can handle concurrent client requests despite not knowing the semantics behind the data and operations. Our results show that our system significantly decreases storage costs per node, and can handle large numbers of concurrent client requests for a small additional memory overhead.

Contents

Chapter 1

Introduction

As of 2018, 77% of enterprises have at least one application or a portion of their enterprise computing infrastructure in the cloud [3]. The cloud enables applications and their data to be run and stored "somewhere else", removing that infrastructural concern altogether; this and other conveniences are what makes cloud computing and hosting services extremely popular. They can offer added performance, fault-tolerance, and security. Some cloud computing services even allow for high resource-demanding applications to be automatically scaled and run on multiple machines. In particular, if these machines form a computationally universal system[1], then any conceivable application could be run. Banks [30], online newspapers, social media applications [29], all types of services and their data are increasingly managed by third-parties, sparking the need for research in additional solutions.

One major concern is the increasing centralization of the cloud computing market. By the end of 2019, over 60% of the market was controlled by just 4 companies [7]. Centralized systems have many advantages, particularly when it comes to achieving higher performance and scalability. However, they are often controlled by a sole authority; this requires a certain degree of trust. Decentralized systems, on the other hand, do not have this requirement. Responsibility tends to be spread out throughout the participants, each potentially representing a different authority. They are often built to be Byzantine fault-tolerant (BFT), allowing some of the participants to be malicious (Byzantine) without affecting the system. By contrast, centralized systems often just assume that no non-client participant is malicious, as they are all controlled by the same authority. However, this added advantage of decentralization comes at a cost: BFT decentralized systems tend to be more complex, harder to design, and unable to achieve the performance of their centralized counterparts. This is particularly noticeable in universal systems, which are a big research focus as they can be used to run any type of application. One of the main challenges we see lies in designing a secure decentralized universal system with performance matching or surpassing that of a centralized one.

[1] Also known as a Turing-complete system.

In the last two decades, there has been a significant research effort put into systems that try to tackle this challenge, in part or in full, each with differing drawbacks. Some early Proof-of-work type systems, such as the original Bitcoin [27] and Ethereum[4], might use a considerable amount of resources [1] or operate in a synchronous model[2]. Each "miner" (node) in the system also needs to store the entire state of the system, e.g., all of the accounts and corresponding assets in a cryptocurrency system. Later Proof-of-Stake systems, such as Algorand [15], sought to resolve the resource (energy) and synchronicity problems, but the issue of growing state storage remained. Only with later research into compatible state sharding solutions [12, 24], where each node in the system holds only *part* of the global state, has there been significant progress in increasing storage scalability. This research is very recent, and many appealing solutions are yet to be fully investigated [8, 31].

We take many inspirations from these systems and unexplored research avenues. The main purpose of this work is to lay the foundation for a universal decentralized database that can shard storage across participant nodes, operate with generic data and transactions, and handle large numbers of concurrent client requests. With future work, we also hope to reach or surpass the state-of-the-art of cryptocurrency systems when it comes to scaling transaction throughput. To achieve this, we design a system from the ground up, implement a prototype, and evaluate it.

Our system is composed of a fixed set of nodes that store the data, i.e., key-value pairs, and execute operations (transactions), forming a distributed Key-Value Store (database). Communication is mostly client-driven: when a client wants to apply an operation, it first collects the operation's input data from the database by querying these nodes and subsequently provides it to the nodes, acting as an intermediary; this decreases stress on system nodes. To guarantee security, we rely on two core components: *Merkle tree*-based cryptography, and *Total Order Broadcast* (TOB). The former plays a fundamental role in our system, and we extend on existing work and incorporate those extensions into our design. Due to this, we dedicate a separate, theoretical chapter on the introduction and design of Merkle prefix trees. As a side-effect, we have implemented a Merkle prefix tree library in Rust that can also be used independently. The latter, Total Order Broadcast, plays a role in making our system universal, ensuring that operations are executed in the same order by all nodes. The TOB primitive should be both efficient, Byzantine fault-tolerant, and partially synchronous. As it is outside the scope of this project, we simply assume its existence[3].

In summary, given n nodes (i.e., servers) and a Total Order Broadcast primitive which is efficient, decentralized, and Byzantine fault-tolerant, we construct a Universal Key-Value Store with similar properties.

The operations can be created and defined by users with very few imposed restrictions (similarly to digital contracts) and are themselves stored as opaque data in the database. Each

[2]One needs to wait for some time, the "confirmation time", to be confident the transaction has gone through, as is the case with Bitcoin. This affects throughput and introduces vulnerabilities [21].

[3]We note that recent work by Guerraoui et al. [19] heavily hints at the existence of an $\mathcal{O}(n \log n)$ complexity TOB primitive for a system of n nodes, in these conditions.

system node only maintains a part of the database, which can be made proportional to its storage capacity. Because of this, the total storage capacity of the system is not restricted by the capacity of any given node, and can thus scale infinitely with additional nodes and capacity. By relying on Merkle proofs, communication is mostly client-driven; direct communication between system nodes is only required by the Total Order Broadcast primitive.

Our core contribution lies in the conception, design, and implementation of this system. Our proof-of-concept fully displays the end-to-end request and execution of user-defined operations on the database. Relying only on a simplified instance of Total Order Broadcast, it employs and demonstrates all the aforementioned aspects of client-driven communication, storage partitioning (sharding), decentralization, opaque user-defined operations, and universality. Our results show that our system significantly decreases storage costs per node, and can handle large numbers of concurrent client requests for a small additional memory overhead.

Chapter 2

Background

Here we present the necessary background to understand our work.

Universality. A system is computationally universal (or Turing-complete) if it can be used to simulate a Turing machine. Put simply, a universal system can simulate the work of any other real-world general-purpose computer or computer language. In a distributed database system, this can be achieved via *atomic transactions*. An atomic transaction is a sequence of instructions (including reads and writes) that, by all appearances, is executed at a single point in time (atomic). This is referred to as *Consistency* in the well-known CAP theorem [16]. As a trade-off to increase performance, many databases choose to relax their requirements and implement less constricting models [26]. Although less powerful, such systems are more than adequate to certain specific scenarios. Even the seemingly hard problem of *double-spending* in a decentralized asset transfer system [6] has been shown to be solvable by weaker, causally consistent systems, under certain conditions [18, 20]. For our system, however, we wish to allow our users to create and execute all types of operations (transactions) opaquely, i.e., without the system having any semantic understanding of the operation. An operation might transfer money between accounts, run an auction, or post a message in an online group chat, among other things. To accommodate all such scenarios in the most consistent manner possible, we need to enforce atomicity in our operations, making our system universal.

Byzantine Consensus and Total Order Broadcast. In the case of distributed systems, the problem of achieving a fault-tolerant universal system is directly tied with the problem of achieving consensus (agreement) in the presence of faulty processes (or nodes[1]) [22]. In general[2], a consensus protocol should satisfy the following set of properties [5]:

- *Termination:* Every correct process eventually decides some value.

[1] Each process can be run on a different node or server.

[2] There are many alternative definitions of this set of properties in the literature, including stronger and weaker versions of the Validity property

- *Agreement:* No two correct processes decide differently.
- *Validity:* If a process decides v, then v was proposed by some process.
- *Integrity:* No process decides twice.

Moreover, in the case of *Byzantine fault-tolerant* (BFT) consensus [23], faulty processes may fail in any arbitrary way: they might simply crash, and they might also freely deviate from established protocols, even maliciously. BFT systems typically tolerate up to a third of the total number of processes. Given our work's assumption of a *partially synchronous network*, which we introduce afterwards, and in the absence of stronger assumptions, this is known to be the limit [28].

In a distributed BFT database system, one direct way to make transactions atomic is to have each process use a BFT consensus primitive to decide on a sequence of transactions to execute. If each correct process shares its transactions, includes all other shared transactions in its proposed set, and repeatedly uses BFT consensus, all transactions are eventually executed by all correct processes in the same order [22]. Another primitive that can accomplish this is called Total Order Broadcast[3] (TOB). Total Order Broadcast is a broadcast[4] primitive that imposes a total order on message delivery: all processes using the TOB primitive to communicate will receive messages in the same order. TOB can be shown to be equivalent to consensus [9], and satisfies the following set of properties[5]:

- *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.
- *No duplication:* No message is delivered more than once.
- *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.
- *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.
- *Total order:* the messages are totally ordered in the mathematical sense, i.e., if any correct process delivers message m_1 first and message m_2 second, then every other correct process must deliver m_1 before m_2.

In much the same way, processes broadcast their transactions (include them in their message) using a Byzantine fault-tolerant TOB primitive. Through the properties above, all transactions

[3] Also known as Atomic Broadcast.

[4] In the context of a given set of nodes, a broadcast primitive allows each node to send a message to all nodes in the set (multiple times).

[5] These properties are based on [5] and [2]. The *Total order* property was altered for readability and is equivalent to that in [5].

originating from correct processes (*No creation*) are eventually executed by all correct processes (*Validity* and *Agreement*), in the same order (*Total order*) and exactly once (*No duplication*).

In our work, we make use of this TOB primitive for our theoretical correctness results. Research and implementation of a full-fledged decentralized BFT TOB primitive are left outside the scope of this project.

Partial synchrony. Three of the most common network models used in distributed systems are the synchronous, partially synchronous, and asynchronous models (in decreasing strength of assumptions). The synchronous model makes very strong assumptions on computation and communication times, such as assuming the existence of a maximum delay (Δ) between message sending and delivery and for the execution of any instruction. Fault-tolerant primitives in this model are typically easier to construct, e.g., they can make use of a Perfect Failure Detector [5, 13] to accurately detect failing nodes and remove them from the system[6]. The issue with this model lies in its practicality: it is often unrealistic, as real network delays might surpass the assumed limit, or the limit might be set too high, slowing down the system (particularly in the case of failure detection). The asynchronous model, on the other hand, makes fewer assumptions and is often more robust. The adversary can selectively delay, drop, or re-order any messages sent by honest parties. However, some problems are proven to be impossible to solve in this model, such as fault-tolerant deterministic consensus [14].

Our system operates under the partially synchronous model. The partially synchronous model is identical to the synchronous model with the difference that the maximum delay (Δ) is unknown for all correct processes. This also corresponds to the *Type 1 partial synchronous* model summarized by Wang [32] from the work of Dwork, Lynch, and Stockmeyer [10]. We note, however, that the implied assumption that messages are always delivered concerns only *high-level* messages. Packets and (low-level) messages at the transport layer level can still be dropped: as long as these omissions do not happen infinitely often, then primitives such as the *Perfect Links* described in [5] can be used to uphold the above assumption. This can be achieved in practice by using, e.g., TCP connections.

WebAssembly. WebAssembly[7] (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. It was designed as a portable target for compilation of high-level languages such as C, C++, and Rust. An example code snippet can be seen in Figure 2.1. WebAssembly describes a memory-safe, sandboxed execution environment; each WebAssembly module executes in this environment, isolated from the host runtime via fault isolation techniques. Because of this, applications can only interact with the host via appropriate APIs and execute deterministically except in limited, well-defined cases. Other good features include its size-efficiency and load-time efficiency, ideal for transmitting and running modules (which would encode system operations) on-the-fly as is needed by our system, as well as its execution efficiency. Despite its recency and lack of maturity, its current aspects make it a suitable candidate

[6] Here we are referring to crash-stop (halting) failures and not Byzantine failures.

[7] `https://webassembly.org/`

Text format (.wat)

```
(module
  (func $multiply (param $lhs i32) (param $rhs
     i32) (result i32)
    get_local $lhs
    get_local $rhs
    i32.mul)
  (export "multiply" (func $multiply))
)
```

Binary format (.wasm)

```
0061 736d 0100 0000
0100 0160 027f 7f01
7f07 0300 0100 0207
0001 0661 6464 5477
6f00 000a 0a00 0100
0020 0020 016a 0b07
0900 0004 6e61 6d65
0200 0100 0200 0001
0007 0e
```

Figure 2.1: Example code snippet of WebAssembly's text format (left) and binary format (right)

for our system's default operation language. Moreover, future features and improvements such as better *dynamic linking*[8] of modules and *multiple memories* [9] might enable further system capabilities, such as a preloaded, standard system library, or nested operations.

[8] https://webassembly.org/docs/dynamic-linking/

[9] https://github.com/WebAssembly/proposals#multiple-tables-and-memories

Chapter 3

Merkle Tree Design

Merkle prefix trees are the core data structure of our system, playing a fundamental role in storage, security, and performance. For this, we dedicate a separate chapter aimed at introducing the background of Merkle prefix trees, their basic functionality, and some extensions later used by our system that might also be of independent interest.

3.1 Merkle prefix trees

A Merkle tree is a data structure that holds a set of records, i.e., associations between keys and values, similar to a map. In a normal binary Merkle tree, records are stored as leaves of the tree without any particular order. There are also empty nodes, which are leaves without a record, as well as internal nodes, which make up the "branches" of the tree; the root node of the tree, as well as all nodes between it and the leaves, are internal nodes, and all internal nodes have exactly two children (binary). Every node also has a unique *label* assigned to it. Generally, a leaf's label is equal to a cryptographic hash of its content, which includes its key and value, an empty node's label is usually set to a default value (e.g., the zeroes vector), and an internal node's label is the hash of the concatenation of the labels of its children. With a correct choice of hash function, this makes it *very* unlikely for any two different Merkle trees to have matching root labels.

The primary purpose of a Merkle tree is *not* to be used as a map, but to be able to prove the inclusion of elements in a set *without* providing the whole set. Suppose that we have stored a Merkle tree with 1 billion records. A client, Alice, is interested in knowing the value of one of the records, e.g., the record "Alice bank account". All that Alice knows is the label of the root of the Merkle tree, which uniquely identifies the set of records, but nothing else. We *could* provide Alice with the entire tree: after Alice verifies that our Merkle tree is correct and that the root label is the same, they can feel safe that we provided the real tree and obtain the value of the record. However, this is unnecessary: instead of providing the entire tree, it is sufficient to provide the

leaf corresponding to the "Alice bank account" record, all of its ancestor nodes up to the root, as well as the *labels* of all their siblings. This is called a *Merkle proof*. The siblings' labels uniquely represent the other records in the tree that Alice is not interested in while being sufficient for Alice to ascertain the authenticity of the tree. The size of this proof is only a logarithmic factor of the Merkle tree if the tree is balanced.

Because normal Merkle trees do not determine which leaf stores which record (no order), there is not an efficient way to find the leaf storing a particular record. This makes Merkle trees worse than typical radix trees for the purpose of manipulating its elements (inserting, removing, etc.). However, there is a way to combine the techniques of both Merkle and radix trees and obtain the best of both worlds, called a Merkle prefix tree.

A Merkle prefix tree is a binary Merkle tree where each record (key-value pair) can be found along a prescribed path starting from the root and depending on that record's key. Using a collision-resistant hash function H with k-bit output, we map a key to obtain its record's path. Each node in the tree represents a unique prefix, with each branch adding a 0 (left-branch) or a 1 (right-branch) to its parent node's prefix. There are three types of nodes:

Internal nodes exist for any prefix that is shared by more than one key present in the tree. Like in a normal Merkle tree, its label is dependent on the concatenation ($||$) of the labels of its children:

$$h_{\text{internal}} = H(h_{\text{left}}||h_{\text{right}})$$

Leaf nodes represent exactly one record with key k and value v. A leaf node indicates the uniqueness of its prefix. They are labelled as follows, where S is an invertible function:

$$h_{\text{leaf}} = H(S(k, v))$$

S is applied to prevent certain hash-collision attacks, like those derived from the simple concatenation of k and v.

Empty nodes represent the absence of their corresponding prefix: no key in the tree has that prefix. They are assigned a default value as a label (e.g., the "zero" element of H's codomain):

$$h_{\text{empty}} = [0]^k$$

Collision attacks. Due to our way of hashing leaf nodes, there is the possibility of a birthday attack to try to find two different nodes with the same hash (a collision), e.g., by changing the value of a record. However, we can still achieve k-bit security[1] by using a 2k-bit output collision-resistant hash function [17]. By choosing a sufficiently high k, any such attack becomes unfeasible. Throughout the rest of the work, we assume that the probability of such collisions is negligible and practically impossible. We note that there are less wasteful solutions of hashing schemes, such as the one proposed by Melara et al. [25] allowing full k-bit security using a k-bit output

[1] An attacker has to perform 2^k operations on average to find a collision.

hash function. However, due to the small impact on performance and their interference with our later extensions, we chose not to use these solutions.

Operations. Records can be obtained, added, modified, and removed from the Merkle prefix tree. All operations preserve the node type invariants defined above. The expected complexity of these operations is logarithmic in the number of records R in the tree, i.e., $\mathcal{O}(\log R)$.

3.2 Merkle proofs

A Merkle proof is essentially a pruned Merkle prefix tree that only holds information pertaining to a subset of records that we are interested in. Irrelevant branches of the tree are replaced by *placeholder nodes*:

Placeholder nodes represent any other type of node by storing and producing the same hash:

$$h_{\text{placeholder}} = h_{\text{internal}} \mid h_{\text{leaf}} \mid h_{\text{empty}}$$

They have no child nodes and are used as a constant-sized stub. Placeholder nodes are mainly included so that the hash of their parent nodes may still be computed.

To prove the *inclusion* of a record in the original tree (proof of inclusion), the full *authentication path* must be included in the Merkle proof. The authentication path includes all nodes between the root and the leaf node corresponding to the record. Siblings of nodes in the authentication path must also be included in the proof; they are stubbed out (i.e., replaced with placeholder nodes) unless they are part of the authentication path of another record also included in the proof. We can also prove the *absence* of a record, i.e. that there is no value associated with a particular key (proof of absence). Similarly, we include in the Merkle proof an authentication path between the root and an empty node or leaf node that correspond to a prefix of that key. The presence of an empty node or a leaf node corresponding to a different key implies that the record does not exist. An example can be seen in Figure 3.1. The path leading to and including the *empty* leaf corresponds to the authentication path for one or more records with prefix 10 (proof of absence), explaining why that node is not stubbed. The root's left child lies outside all authentication paths and is replaced by a *placeholder* node.

In the context of our work, Merkle proofs are sent over the network and used both for proving the inclusion and absence of records, as well as transmitting the data associated with those records (if they exist). This is why our Merkle proofs include the full value (v) of a record. Whenever a Merkle proof is received from an untrusted network, it must be validated. This can be done by checking that all leaf nodes lie in the path corresponding to their key. Any cached labels must also be recomputed[2]. For $R \ll S$, the expected space complexity of a Merkle proof is $\mathcal{O}(R \log(S))$,

[2]For example, as an optimization, our implementation caches the labels of internal nodes. These must be recomputed since they might be forged.

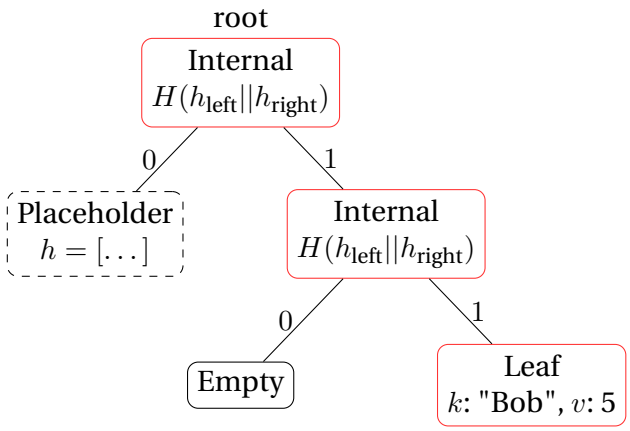

Figure 3.1: Example of a Merkle proof. In red is the authentication path for the key "Bob".

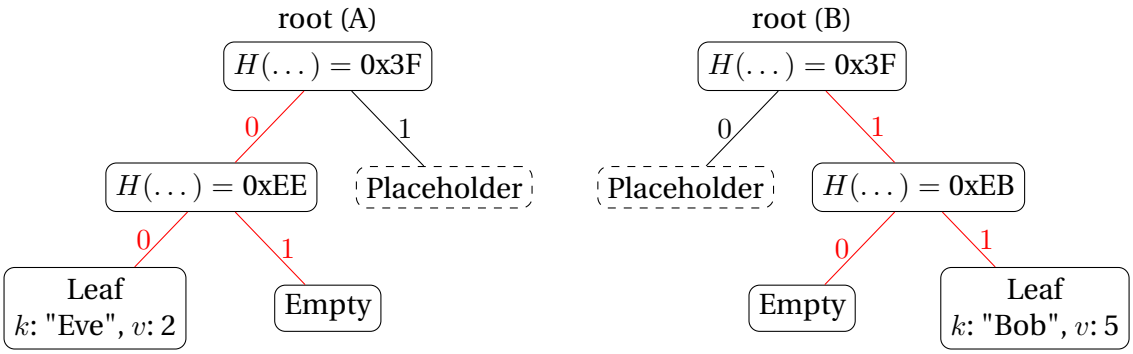

Figure 3.2: A Merkle tree split into two partitions, A and B.

where S is the total number of records in the original Merkle tree, and R is the number of records for which we include a proof of inclusion or absence[3].

3.3 Partitioning and recovering data

A Merkle prefix tree can be partitioned and stored over multiple machines. This allows us to store more records and distribute the execution load. A partition of the *base* Merkle prefix tree is defined by one or more prefixes; all nodes of the base tree under those prefixes are included in the partition. Following our previous definition of a Merkle proof, a partition corresponds exactly to a Merkle proof for all leaf nodes (proof of inclusion) and all empty nodes (proof of absence) under those prefixes.

Figure 3.2 shows an example of a Merkle prefix tree partitioned on the most significant bit. Merkle proof A includes complete information on the subset of records of the original tree with

[3]Tighter bounds may be found. For our purposes and having k-bit security, R is constant and $log(S) <= k$.

prefix 0. The same is the case for Merkle proof B, but for records with prefix 1. The hash values attributed to the internal nodes are used as an example and are not necessarily the real values. Like a normal tree, each partition can be stored on multiple machines for redundancy and fault-tolerance. In general, each prefix is directly or indirectly covered by k different machines, where k depends on the desired degree of fault-tolerance.

It is also possible to combine multiple partitions to obtain a partition covering the union of their prefixes, potentially even reconstructing the *base* tree. This is useful when changing the distribution of partitions over a system of machines, or when a single machine is performing an operation that spans multiple partitions. There is an efficient and straightforward way to achieve this:

1. Take one of the partitions and name it p_{local}.
2. Starting from the root, run a depth-first search (DFS) on p_{local}.
3. Whenever a placeholder node ph is found, if there is an alternative node n in another partition in the same position (same depth and path from the root)[4], replace ph with n in p_{local} and continue the search from n.
4. Continue until the depth-first search terminates, returning the modified p_{local}.

It is easy to see that the above algorithm is correct, assuming partitions of the same base tree. The returned tree is a partition that includes all leaf and empty nodes in every partition being combined. We prove this by contradiction. Suppose there is some node n in some partition p that is not included in p_{local} at the end. In the original p_{local} partition, node n must have either been:

1. Present and not a placeholder. If so, it would have been included in p_{local} at the end: only placeholder nodes are replaced, and placeholder nodes have no children that can be removed.
2. Represented in the form of a placeholder node. In that case, it would have been directly replaced by n, and case 1 follows.
3. Absent, and so the DFS did not reach n. For the DFS on p_{local} not to reach it, it must have retreated at some earlier point in the path to n. For that to happen, then in the path to n the DFS either:
   (a) found a leaf or empty node. This is impossible since all partitions are of the same tree, and no leaf/empty node can be found between the root and a leaf/empty node.

[4] We can simultaneously run the depth-first search on other partitions, pausing them as necessary to keep them in sync. This allows us to immediately know if there are alternative nodes in the same position.

(b) found a placeholder node with no non-placeholder alternative in the other partitions, including p. This implies that p does not include n or is an invalid partition, which contradicts the assumptions.

Note that, due to the way we define partitions as Merkle proofs, all of these methods can be applied to both.

3.4 Recovering data from old trees

The previous section explained how we can combine partitions to recover knowledge. An important assumption for this is that all partitions are of the same tree. This can be a bit restrictive, particularly when the partitions relate to trees that are only slightly different. Take the example of Figure 3.3.

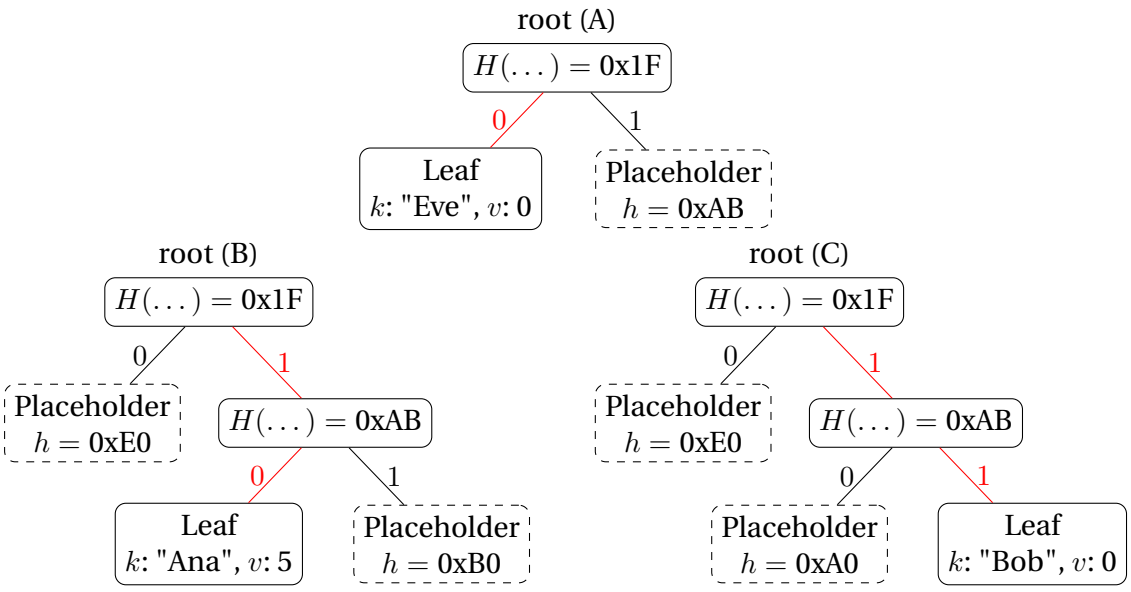

Figure 3.3: A Merkle tree split into three partitions, A, B and C. We have partition A for prefix 0 (top), partition B for prefix 10 (bottom-left), and partition C for prefix 11 (bottom-right).

Suppose that all partitions are stored in different machines that are connected in a network and that these machines are called A, B, and C respectively. Imagine a case where, for whatever reason, a client wants to construct a partition that includes both the record "Eve" (in partition A) and "Ana" (in partition B). To achieve this, the client separately requests machines A and B for their partitions so it can combine them locally. However, in the meantime, the value for "Bob" (in partition C) is modified globally, e.g. incremented, albeit at different times for each machine. Because of this, the client receives partition A (Figure 3.3), compatible with the old value of Bob, and partition B' (Figure 3.4), compatible with the new value of Bob.

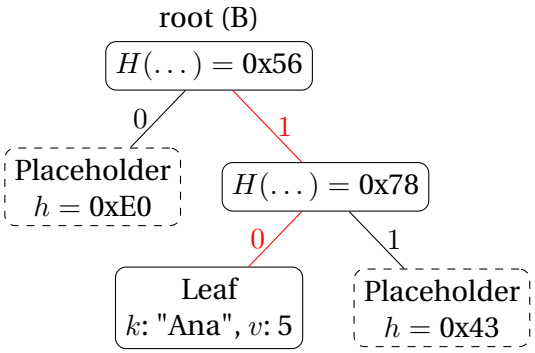

Figure 3.4: Partition B' (prefix 10), corresponding to the tree after the value of record Bob was modified.

We can see that in the new partition B, all hash values in the authentication path of Bob were modified. However, if we look at the path for "Eve" (left from the root), we can find a node that has not been changed: the placeholder ($h = 0$xE0) is still the same. This means that, when it comes to the Eve record, the old partition A is compatible with the new partition B; it can be used to extend partition B to include the Eve record. To do this, we look at all nodes in the authentication path of Eve in partition A: if any node n can substitute the placeholder (in this case, if it hashes to 0xE0 like the placeholder), then we do so. Since collisions are negligible, the placeholder must have originally replaced n, and we are simply recovering n. Since partition A included the full authentication path to Eve, partition B will now also include it.

In general, given:

- a base partition p of a tree $tree_p$ that we are extending.
- a partition q of a tree $tree_q$.
- a key k of a record for which q holds a proof of inclusion or absence.

we can attempt to extend p to include the proof of inclusion or absence of k, and which is compatible with $tree_p$. We run the following algorithm (Algorithm 1):

1. Find the last node in the authentication path of k in p.
2. If the node is an empty node or leaf node, compare it against the last node in the authentication path of k in q:
   (a) if both nodes match (both empty, or both leaves with the same key and value), return SUCCESS.
   (b) else, return FAILURE.
3. Else, the node is a placeholder with hash $h_{\text{placeholder}}$. Retrieve all nodes in the authentication path of k in q (root to leaf/empty node):

(a) if there is some node n retrieved for which $h_n = h_{\text{placeholder}}$, replace the placeholder node in p with n and return SUCCESS.

(b) else, return FAILURE.

Observation 1. *If the record corresponding to k is not the same in $tree_p$ and $tree_q$, i.e., it is not absent in both or it is not present in both with the same key and value, then the algorithm always returns FAILURE for any partition p of $tree_p$.*

Proof. If p includes k (proof of inclusion or absence), then it will execute steps $1 \to 2 \to$ b (the observation's assumption is the contrary condition of step 2.a) and return FAILURE. If p *does not* include k, then step 3 applies. According to the observation's assumption, each last node in the authentication paths of k in $tree_p$ and $tree_q$ are different and, due to collision-resistance, so are their hashes and the hashes of all their predecessors (w.h.p). This means that no nodes in the authentication paths of k in $tree_p$ or $tree_q$ have the same hash, precluding step 3.a. The algorithm will execute steps $1 \to 3 \to$ b and also return FAILURE. □

Observation 2. *The algorithm is not indifferent to the base partition. If the record corresponding to k is the same in $tree_p$ and $tree_q$, executing the algorithm using two different base partitions of $tree_p$ may yield different results (e.g., one SUCCESS, another FAILURE).*

Proof. We can see this in the original example. Let the record be "Ana", let partition q be the old partition B (Figure 3.3), and let the two base partitions be partition B' (Figure 3.4) and partition A' (Figure 3.5). Executing the algorithm with partition B' yields SUCCESS ($1 \to 2 \to$ a), while executing the algorithm with partition A' yields FAILURE ($1 \to 3 \to$ b). □

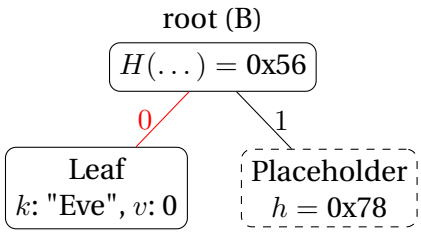

Figure 3.5: Partition A' (prefix 0), corresponding to the tree after the value of record Bob was modified.

Definition 3.4.1. A tree history is a sequence of Merkle prefix trees $t_0 \to t_1 \to \cdots \to t_T$ where every tree in the sequence is obtained by applying a single operation to the previous tree.

Definition 3.4.2. Given a tree history $t_0 \to t_1 \to \cdots \to t_N$ and a tree t_n in that history, a partition p_{t_n} of t_n is T-history-aware if every placeholder node ph in p_{t_n} has an age greater or equal to T, i.e., the node corresponding to its prefix in t and that it replaces is the same in $(t_{n-T}, t_n]$.

Theorem 1. *Take a tree history $t_0 \to t_1 \to \cdots \to t_N$ and two trees $t_{n-A}, t_n : A > 0, n > A$ in that history. For any key k, any partition q of t_{n-A} with an inclusion or absence proof for k, and any set*

of T-history-aware base partitions of t_n where $T \geq A$, the previous algorithm always yields the same result.

We prove this theorem by combining the following two lemmas:

Lemma 2. *If the record with key k was inserted, modified, or removed by an operation between $(t_{n-T}, t_n]$, then the algorithm yields the same result.*

Proof. If the record with key k was inserted, modified, or removed by an operation between $(t_{n-T}, t_n]$ then all base partitions have a proof of inclusion or absence for that record. Otherwise, it would mean that the record lies behind a placeholder with age lower than T, which would violate T-history-awareness. Therefore the algorithm will execute step 2 for all base partitions which, by referring to the same tree t_n, possess the same authentication path and yield the same result. □

Lemma 3. *If the record was last inserted, modified, or removed by an operation before t_{n-T}, then the algorithm yields the same result.*

Proof. Since the record was last changed before t_{n-T}, then whether it is absent or exists with a specific value it will be the same for all trees in $(t_{n-T}, t_n]$, and in particular for t_{n-A} and t_n. Therefore, for any base partition with a proof of inclusion or exclusion of k, the algorithm will execute step 2.a and return SUCCESS. This leaves base partitions where the node found when executing step 1 is a placeholder node. From the definition of T-history-aware, the placeholder node found in step 1 must correspond to a node that is the same in $(t_{n-T}, t_n]$, and in particular in t_{n-A}. Therefore the full authentication path in q must include this node (which has the same hash), and so the algorithm will execute step 3.a and return SUCCESS. □

In practice, Theorem 1 is used by a Merkle prefix tree partitioning scheme where the most recently manipulated data is always stored by all machines, eventually being "forgotten" and only kept by those directly responsible for it. By employing the above methodology, we can stop certain operations from interfering with each other and have them safely execute concurrently over a network, as we will see later in Section 4.3.5. Theorem 1 also works for histories of trees where each tree is obtained by applying a *bounded set of operations* to the previous tree, instead of just a single operation. The wording of the following definition and lemmas remains the same.

Chapter 4

System Design

The general goal of our system is to provide a decentralized key-value store that is flexible, safe, and highly scalable. In this chapter, we first define the system model and some common notation used throughout the rest of the work. We then specify the main goals of our system and outline how we plan on achieving them. Finally, we provide a detailed, step-by-step description of the design of the system, explaining the decisions made at each step.

4.1 System Model

Our system model includes two types of participants: system nodes, and clients.

System nodes. System nodes or just nodes are responsible for storing the global state, i.e., the data (records/key-value pairs), and responding to client requests, such as executing an operation. We assume that up to one-third of all nodes might be malicious (Byzantine), and deviate from the defined protocol. We also assume any correct node can communicate with any other correct node, and that the network operates in partial synchrony, i.e., communication between any two nodes cannot be reliably blocked. Correct nodes also compute instructions in a finite amount of time (they do not block).

Clients. Users run client-side software that interacts with the database by communicating with system nodes. Clients can request data, the execution of an operation, or even request the removal of a system node by providing proof of malicious behaviour. Similarly to nodes, we assume that any correct client can communicate with any correct system node in partial synchrony, and that correct clients compute instructions in finite time. We also assume that any client might be malicious and deviate from the protocol, but in doing so the malicious client will lose all guarantees given to correct clients.

4.2 Design goals

Our system goals are divided into security and deployability goals.

Security goals

G1. Agreement: If a correct node executes an operation, then all correct nodes will eventually execute that operation[1]. Since operations are deterministic, it is implied that if two correct nodes are at the same global state s and apply the same operation, then both nodes will move on to the same global state s'.

G2. Total Order: Operations are totally ordered; if any correct node applies operation 1 before operation 2, then all correct nodes must apply operation 1 before operation 2.

These two goals jointly give us *non-equivocation*: if all correct nodes start on the same base state (s_0), then all correct nodes must follow the same sequence of states ($s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow ...$) as operations are applied. Furthermore, if we think of applying an operation as *moving forwards in time*, then we can associate a time to each state: the base state corresponds to time 0 ($s_0 \leftrightarrow t_0$), with time increasing by 1 for each operation applied ($s_1 \leftrightarrow t_1$, $s_2 \leftrightarrow t_2$, etc.). Since all correct nodes must have the same state at the same specific time, these can also be called *global* states and the ordered sequence of global states constitutes the *history*.

G3. Correct execution of operations: As nodes might only have part of the global state (due to storage sharding), clients are responsible for providing all of the information on records that the operation might try to read. All of this information, such as the value of a record or the knowledge of its absence, must be consistent with the latest global state for the operation to be applied; otherwise, the operation is rejected and not applied. This prevents certain unintended situations, such as when an on-flight operation is delayed and no longer valid, or when a malicious user attempts to equivocate nodes by providing false information on the global state. This goal is achieved (with high probability) by relying on Merkle proofs and the collision resistance of the underlying hash function.

Deployability goals

G4. Storage scalability: The system's total storage capacity should not be constrained by the individual storage capacity of any given node. The total effective storage capacity should scale with the sum of storage capacities of all nodes.

G5. Efficiency: Computational and communication complexity should be minimized to allow

[1] All data associated with the operation used when executing (e.g., arguments, state) is also the same.

for high throughput operation processing. Computational complexity should be bound by a logarithmic factor of the number of nodes; this excludes the execution of the operation itself, as it is user-defined. Communication complexity should be bound by the complexity of the Total Order Broadcast primitive. Assuming the TOB primitive is $\Omega(n \log n)$, this is achieved if the remaining communication is $\mathcal{O}(n \log n)$, where n is the number of nodes in the system.

G6. Malicious starvation detection: A malicious node might try to provide a client with modified data that is inconsistent with the global state. Even with this false data being in the form of a valid Merkle proof, if the client is unaware of the correct global state history he might not be able to immediately detect the malicious behaviour. Because of the collision-resistance of the underlying hash function, if a malicious node provides a valid Merkle proof with modified data then the root label will not match any real global root label. If this happens, there is no risk of the operation being executed as system nodes keep track of recent global root labels and can notice the false root label. However, there is a risk of client *starvation* as its operation requests get constantly rejected. In Section 4.3.6, we detail a method to solve this which we call *lazy auditing*.

G7. Operation flexibility: We allow users to define their own operations in an opaque manner and store them in the database (they can be executed afterwards). The constraints are that operations must be deterministic, follow the system's predefined (de)serialization protocols for input/output, operate on a bounded number of records predetermined before execution, and take a bounded quantity of arguments[2]. By default, operations are assumed to be written in WebAssembly. This gives users the flexibility to code operations in several other languages that they might prefer (e.g., C++, Java, Python, Rust) and compile them to WebAssembly, without the need to learn a new domain-specific language.

4.3 System description

We provide a high-level, step-by-step description of the design process. Each step solves a separate problem or design goal while retaining the progress of the previous steps. The end result is the complete system achieving all the aforementioned design goals.

4.3.1 Domain structure

We first define the general structure of the domain and organization of data on the key-value store. Note that, while some decisions made in this section have some impact on the design of the distributed protocol, most can be taken in isolation and are not binding. Alternative domain structures can be chosen and easily implemented. This section defines a base that we use to guide our concrete implementation.

[2]These bounds are set such that they are non-restrictive for the vast majority of operations.

In our key-value store (database), records are pairs of keys and values addressable by their key. Keys are defined as UTF-8 encoded strings[3], while values are defined as generic byte vectors. This allows users to encode whatever information they desire in the most efficient way possible. There are two main system operations[4]:

- **GET.** Allows a user to read a set of records. It takes as arguments a bounded list of keys which may or may not be present in the database. It returns a Merkle proof for all of the keys (proof of inclusion or absence). If the record is present, it can be read from the Merkle proof.
- **EXECUTE.** Allows a user to apply an operation on the database. It takes as arguments an operation, a list of records used by the operation, a Merkle proof for those records, and any extra arguments taken by the operation. The Merkle proof can be obtained using the GET operation with the same list of records.

Operations are stored on the database itself as records. To execute a new operation, a user first needs to store it by running a different operation that is initially provided, and subsequently GET and EXECUTE it. Users are free to code any operation they want. Our current implementation assumes a WebAssembly binary format with specific (de)serialization of input and output, but this is not strictly necessary. The only requirement on the language used for the operations is that it can be run on an isolated, safe environment and that its execution must be deterministic.

All records are public for reading, i.e., can be returned by the GET operation. If privacy is desired, users can always encrypt the data they store. This retains generality while simplifying our system design. However, records pertaining to operations cannot be encrypted since the system needs to be able to execute them. This allows any user to GET and EXECUTE any operation, which can be problematic. This is solved in several parts. One part is that the operations themselves can be internally coded to require a cryptographic signature as an argument, which they can then verify. A random user with no access rights would not be able to produce this signature, the operation would abort, and there would be no effect. The remaining concern is that, as is, operations can modify any record. Even records of other operations are subject to change. There are multiple ways to solve this issue.

One way is the following. We impose a domain hierarchy on the record keys similar to a file system: domains and subdomains in our hierarchy are equivalent to folders and sub-folders, having the same relationship, and separated by slashes (`"/"`). The topmost domain is called the *root*. All other domains are subdomains of the root. Now, we can limit access of operations to their domain and subdomains via the following rule:

System-wide rule: Take the key of the record of an EXECUTE operation. Take a prefix of that key up to the last slash character (`"/"`), i.e., the last domain. All records inserted, modified or

[3] Compatible with the Rust String type (`https://doc.rust-lang.org/std/string/struct.String.html`).
[4] Not to be confused with the operations applied on the database, i.e., user-defined operations.

removed by this operation must have that prefix (domain). Otherwise, the EXECUTE fails and no changes are applied.

We initialize the database with a single record named ".root_rule", which is under the root domain by definition. This record encodes the operation that is responsible for "creating" new domains. It takes the name of the new domain as an argument, checking if it is a valid new sub-domain of the root: its name ($domain\_name$) must be non-empty, and the record "$domain\_name$/.base_rule" must not exist. The record "$domain\_name$/.base_rule" is then created, which encodes a template operation that can create any record (including new operations), with the limitation imposed by the rule just defined. It is used by a client to bootstrap the new domain. In combination with the previous system-wide rule, this makes it so no user has access to the top-level domain, and can create their own domain that is isolated from modification by operations in other user domains. An example of this structure can be seen in Figure 4.1.

This is only one of the possible solutions and serves to demonstrate the flexibility of the system. For example, we can also have the ".root_rule" take a public key as an additional argument and create a ".base_rule" that uses it for authentication, making it so only those with knowledge of the corresponding private key can bootstrap the domain.

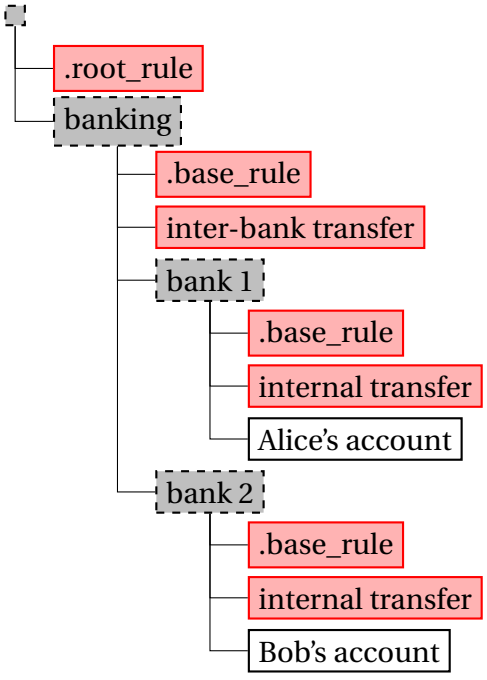

Figure 4.1: Example of the system's record structure. Nodes highlighted in grey only represent domains and are not actual records in the database. Nodes highlighted in red correspond to records that are also executable rules. White nodes are normal records. The complete keys are not displayed, i.e., "Alice's account" is actually "banking/bank 1/Alice's account", etc.

4.3.2 Single client

We start with a system of a single client and N system nodes. For now, every correct node stores a complete replica of the database in the form of a Merkle prefix tree with S records. They are able to perform all of the normal operations of Merkle prefix trees, such as getting, inserting, modifying, and removing records, generating Merkle proofs for records (proofs of inclusion or absence), and combining multiple Merkle proofs into one as described in Section 3.3. We will first describe how to perform the GET and EXECUTE operations detailed in the previous section.

To perform a GET operation, the client broadcasts a *GET request* message to $K \ll N$ nodes at random and waits for a single reply. This can be seen for a system of 3 nodes in Figure 4.2. The probability that all of the selected nodes are Byzantine and never reply is $\epsilon < (1/3)^K$, which is negligible given a sufficiently high constant K. The GET request message is made up of a list of keys (strings). These keys refer to the records that we want to read and/or obtain a Merkle proof for. Upon receiving a *GET request* message, a node produces a Merkle proof for each of the records (inclusion or absence) and combines them into a single Merkle proof. The node then sends a *GET reply* message with this Merkle proof back to the client.

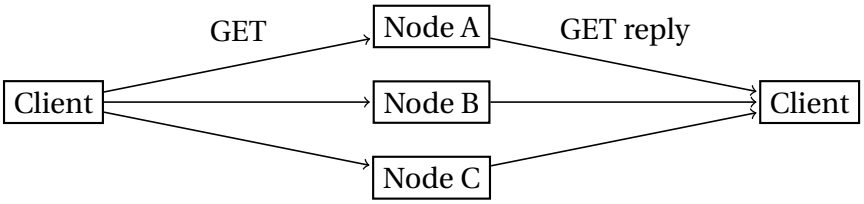

Figure 4.2: A GET operation ($N = 3, K = 3$).

This GET scheme is efficient as defined in goal 5 (**G5**, Section 4.2): there are K request messages of constant size (the number of records is bounded by a constant R), and at most K reply messages of expected size $\mathcal{O}(R \cdot \log(S)) = \mathcal{O}(1)$; the total computation complexity is $\mathcal{O}(K \cdot R \cdot \log(S)) = \mathcal{O}(K)$. However, note that the single reply obtained by the client might come from a Byzantine node and be incorrect, resulting in the issues described in **G6**.

To perform an EXECUTE operation, the client must first obtain a Merkle proof including the operation's record, as well as all the records it touches (inserts, modifies, read, or removes). This is typically achieved by performing a GET operation with this list of records immediately before. After obtaining the Merkle proof, the client then broadcasts an *EXECUTE request* message to N nodes. An example can be seen in Figure 4.3. The *EXECUTE request* message includes:

- The name of the operation (key).
- The Merkle proof including the operation and all records it touches.
- The list of the records touched (keys).
- Other arguments of the operation, jointly serialized as a vector of bytes.

Upon receiving an *EXECUTE request* message, a node must perform a series of steps before applying it on the database. If any step fails, the processing of the request is immediately aborted and no changes are applied. First, it checks that the Merkle proof is correct and is compatible with the local Merkle tree. At this step, a Merkle proof is considered compatible if and only if its root label matches with the local tree's. Second, it checks that there is a proof of inclusion or absence in the Merkle proof for every key in the *list of records touched*. Third, it checks that the operation specified in the message is present in the provided Merkle proof and is valid.

At this point, the node loads the operation into a sandboxed environment. It provides the operation with the extra arguments (serialized) in the request message. It also provides a key-value map structure that allows the operation to similarly get, insert, modify, and remove any of the records in the *list of records touched*; attempting to do so for any other record results in failure. A limit on the number of instructions can be set so that execution does not continue indefinitely. If the operation succeeds, it must return a key-value map structure like the one provided and with no records outside of the *list of records touched*. Finally, the returned map is then compared against the original and any changes (inserts, modifications, removals) are applied to the local tree.

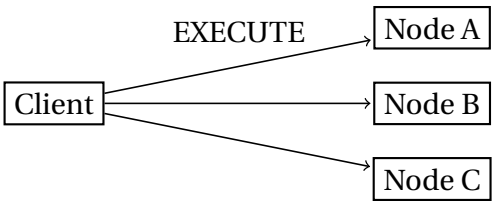

Figure 4.3: An EXECUTE operation ($N = 3$).

Similarly, this EXECUTE scheme is also efficient as defined in **G5**. In a request message, the operation name is $\mathcal{O}(1)$, the Merkle proof is $\mathcal{O}(R \cdot \log(S)) = \mathcal{O}(1)$, the list of keys is $\mathcal{O}(R) = \mathcal{O}(1)$, and the size of the serialized arguments is bounded and $\mathcal{O}(1)$. Therefore, there are exactly N request messages of $\mathcal{O}(1)$ size and the total communication complexity is $\mathcal{O}(N)$. Similarly, it is easy to see that the individual node computational complexity is $\mathcal{O}(1)$, excluding the execution of the operation itself, and the total computational complexity is $\mathcal{O}(N)$.

The steps taken by the nodes before applying the operation's changes on the database guarantee the *correct execution of operations*, as we have defined in **G3**. By using the GET and EXECUTE operations in sequence, along with the ".root_rule" record defined in section 4.3.1, users can create and execute their own generic operations; this achieves **G7**.

4.3.3 Multiple clients

Currently, the EXECUTE operation does not impose any ordering or agreement on the messages. For example, if there are network delays, the client might make two EXECUTE requests in quick

succession that arrive in different orders on different nodes, violating our total order goal. Even if a correct client were to number its messages, there's no guarantee that a malicious client would not break protocol. Part of the nodes might not even receive a client message, be it because the client crashed accidentally or because it is malicious and omitted it. These issues become even more apparent with the introduction of *multiple* clients.

To fix these and similar issues we introduce Total Order Broadcast (TOB) into our system. The goal is to have clients broadcast their EXECUTE requests via TOB to all nodes, instead of doing a simple best-effort broadcast. Figure 4.4 exemplifies this. The centre TOB node represents the communication between system nodes to achieve the properties of Total Order Broadcast on EXECUTE request messages.

As described in the Background, TOB allows a system of N nodes to broadcast messages to all of those nodes, with a set of guarantees on those messages, such as *Total Order* and *Agreement*. Some primitives might allow an outside $client$ node to be the originator of the broadcast, with the result being as if one of the N nodes had broadcast it. However, other primitives might be more restrictive and only allow nodes in the set of N to broadcast. In our system, we plan on having outside clients broadcast through TOB and so we would prefer the first. However, in case that is not possible, there is still a way we can transform the second into the first.

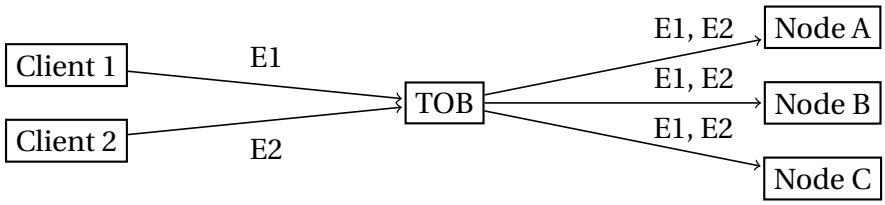

Figure 4.4: Two concurrent EXECUTE operations (E1, E2) using Total Order Broadcast to impose message ordering ($N = 3$).

We take a fault-tolerant approach similar to the GET request. The client picks K nodes at random and broadcasts the EXECUTE request to them, along with its ID (e.g. public key) and an increasing counter (can be signed). Each node that receives the message then broadcasts it via Total Order Broadcast. All messages with the same ID and count after the first are discarded.

The above construction works in theory but could be said to have some issues, namely replay attacks. However, those attacks are inevitable, even if there were encrypted channels between clients and nodes: eventually, nodes must decrypt the requests in order to be able to process them, and with some of the nodes potentially being Byzantine and able to coordinate other clients, operations might be replayed as if coming from an honest client. At the system level and with opaque operations, there is no way to distinguish the replay from an original: there can be operations that are repeatedly requested by different clients. However, if it is important for the application to avoid these attacks, this can still be done. As it is in the case of authenticating clients to make sure they have the rights to modify certain records, we leave it to the operation to detect replays. For example, as an argument of the operation, it can request a signature of a

counter and hash of the operation's arguments. The counter can be stored in the database itself. The operation fails if the value of the counter is lower than the stored value. With this, replay attacks are no longer executed by correct nodes and become about as useful as a normal client request from the perspective of an adversary mounting a Denial of Service (DoS) attack.

This achieves the remaining security goals **G1** and **G2**. Even using the construction above, the total communication complexity would increase by K times the complexity of TOB (K TOB broadcasts). Since $K \ll N$ is a constant, the expected communication complexity is still upper-bounded by a factor of the complexity of the TOB primitive, and the system is still efficient (**G5**).

4.3.4 Compatible late clients

Even though our system can now safely handle EXECUTE requests from multiple clients concurrently, this doesn't imply that it will execute them. In fact, concurrent EXECUTE requests are almost doomed to be rejected. As described in Section 4.3.2, an EXECUTE request's Merkle proof must be considered compatible with the local tree to not be rejected, and it is compatible if and only if its root label is equal to the local tree's. In the current best-case scenario, two concurrent EXECUTE requests will have a proof based on the latest Merkle tree. But if and when one of the requests succeeds and changes the state of the database, the local Merkle tree's root label will also change and no longer be compatible with the Merkle proof of the second request. Hence, for any number of concurrent EXECUTE requests that modify the database at most one is accepted. The fundamental issue is that our current notion of compatibility is too restrictive. Take the following two operations:

- Operation 1:
    1. increase = Read("monthly salary");
    2. money_bob = Read("Bob account");
    3. Write("Bob account", money_bob + increase);
- Operation 2:
    1. increase = Read("monthly_salary");
    2. money_eve = Read("Eve account");
    3. Write("Eve account", money_eve + increase);

Each operation reads and modifies the "Bob account" and "Eve account" record respectively. Both also read the "monthly salary" record but do not modify it. This is detailed in Figure 4.5. Noticing this, we realize that no matter what order these two operations execute in, the result will always be the same. This is because, when each of the operations is executed, the state they touch has not changed, even if there are records that both touch, such as "monthly salary".

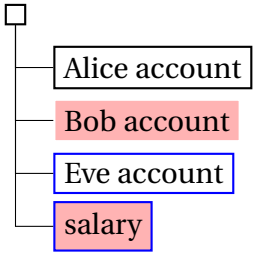

Figure 4.5: Records touched (inserted, modified, read, or removed) by two different operations. Nodes filled in red represent records touched by operation 1. Nodes bordered in blue represent records touched by operation 2.

We can generalize this. Whenever an EXECUTE request is issued by a client, the state that the operation is *expecting* is defined by the *list of touched records* and given in the Merkle proof. Even if some other part of the global state (i.e., the global Merkle prefix tree) has changed, as long as the records on the *list of touched records* have not, then the operation can still be applied as if it had just been issued. Thus, we relax our notion of compatibility. A Merkle proof is now compatible if and only if:

- the Merkle proof's root label is recent, i.e., it corresponds to a tree of at most T operations ago. This T limit is relatively large and serves to immediately reject EXECUTE requests that are too old.
- for every key in the *list of touched records*, its corresponding record in the Merkle proof must be the same as in the (latest) local tree. It must be either absent in both, or present with the same key and value in both.

The first part can be verified by storing each new root label in a FIFO queue of size T. We can check if the queue holds the Merkle proof's root label. If T becomes large and the search complexity becomes worrisome, nodes can also keep a separate $\mathcal{O}(1)$ access hash set. The second part can be currently verified because nodes store the full Merkle prefix tree locally, and so all information pertaining to all records can be checked. As we will see in the next step, this will no longer be the case once partitions are introduced.

With the changes introduced in this section, our system can now process concurrent client EXECUTE requests with a much higher acceptance rate while still doing so safely. In practice, this enables a much higher throughput of operations.

4.3.5 Storage partitioning

One of the main goals of our system is storage scalability (**G4**). For that, we need to have a way of splitting the global Merkle prefix among system nodes while retaining the capabilities required

by the previous steps. In particular, in Section 4.3.4, we rely on nodes having the full Merkle prefix tree to see if the Merkle proof is compatible.

To solve this, we will employ the methodology laid down in the Merkle tree design chapter (Chapter 3). In particular, we use Section 3.3 to partition the tree among system nodes, while the algorithm and Theorem 1 introduced in Section 3.4 are used in the verification of Merkle proof compatibility.

Each system node is assigned a set of prefixes that compose a partition of the global Merkle tree. Merkle tree nodes under these prefixes are *never* stubbed, i.e., replaced by placeholders, directly or indirectly (by having an ancestor stubbed). Prefixes can be assigned to system nodes in a multitude of ways. The main concern is that every possible prefix should be covered[5] by at least one correct node, and preferably more; this is to avoid permanent data loss. We provide one general way of partitioning and analyze its complexity. Research into better alternatives is left outside the scope of this project.

The partitioning method is the following. Let S be the total effective storage capacity of the system. Let p be the number of partitions in the system. We choose p large enough such that the average size $\frac{S}{p}$ of each partition is always constant and small enough that a system node can store on average an $\mathcal{O}(\log(p))$ number of partitions. Such p is possible to achieve, since $\lim_{p\to\infty} \frac{S}{p} \cdot \mathcal{O}(\log(p)) = 0$, and is constant for $p = \Theta(S)$. We can construct each partition by assigning prefixes such that sum of the space occupied by the underlying records is around the average size of a partition. Then, for each partition we select K nodes at random; each of the K nodes will replicate this partition[6]. Since these nodes have at least a $2/3$ chance of being correct, the probability of a partition being lost is at most $\epsilon < (1/3)^K$. With a union bound on all partitions p, the probability of any partition being lost is at most $\epsilon_{\text{total}} < p \cdot (1/3)^K = 3^{\log_3(p)-K}$. Choosing $K = \Theta(\log_3(p))$ appropriately gives us $\epsilon_{\text{total}} < C, \forall_{C>0}$. Using this partitioning method, the system would store a total of $p \cdot K = p \cdot \Theta(\log_3(p))$ such partitions of average size $\frac{S}{p}$, for a total size of $S \cdot \Theta(\log_3(p))$. For constant sized partitions ($p = \Theta(S)$), this means that the effective storage capacity S is $\Theta(1/\log S)$ of the total storage capacity.

Note that this partitioning method assumes that any node might be malicious and that the goal is to prevent *any* partition from being lost, with high probability. If we choose to relax these constraints, we can achieve much better results. For example, if there is a specific fraction of nodes that are assumed to always be correct, then we can keep those as a permanent backup, and wouldn't need $K = \Omega(\log_3(p))$. Or if the high probability guarantee is to be given only for individual partitions and not for the whole, setting a constant K would be sufficient[7], and S would scale linearly with the added capacity of new nodes.

Now that partitioning is complete, we must provide the ability for nodes to check the compatibility of the global tree with a Merkle proof. Notice that the notion of compatibility of the

[5] Covered directly or indirectly, e.g., prefix 01 covers all prefixes starting with 01.
[6] The *final partition* of a node is the union of all partitions assigned to it in this step
[7] For example, with $K = 30, \epsilon = (1/3)^K < 10^{-14}$.

Section 4.3.4 corresponds to the SUCCESS case of Section 3.4's algorithm, in the special case of the node's partition including all records in the Merkle proof. Moreover, if we make it so *every* node's partition is *T-history-aware*, then there is a direct correspondence between compatibility and the output of SUCCESS by the algorithm. Then, to check compatibility before running the operation we can simply run the algorithm, with the slight difference that we don't replace the local tree's placeholder nodes (this does not affect the result). The algorithm's *base partition* is the local tree, partition q is the Merkle proof, and the records are all those in the list of touched records. If the operation is successful, we then run the algorithm again, recovering the local tree's missing knowledge before applying the changes.

To implement T-history-aware Merkle prefix trees, we do the following. In each system node, we store a *current* timestamp counter and a T-sized FIFO queue. Whenever an operation is successful and its changes about to be applied:

1. increment the *current* timestamp counter by 1.
2. push the operation's *list of touched records* onto the queue.

Furthermore, every Merkle tree node in a Merkle prefix tree has an associated timestamp. The difference between it and the *current timestamp counter* is that node's *age*. When applying the changes to the local tree, we abide by the following set of rules:

- leaves and empty nodes start with an age of 0 when first created by an operation.
- internal nodes have an age equal to the minimum of the ages of their children.
- leaves or empty nodes replacing a parent node take the age of the parent (e.g., when contracting an internal node after a *remove* operation).

After the changes are applied and if the FIFO-queue has reached its maximum capacity (T), we pop its oldest entry (a list of touched records). For each key in the list, we follow its path in the tree until we reach the corresponding node. If the tree node's age is T, we replace it with a placeholder node. Placeholder nodes start with the age (timestamp) of the node they are replacing. Whenever this results in a parent internal node having both its children be placeholder nodes, the internal node itself is replaced by a placeholder node (the internal node's age must also be T, as per the previous set of rules). This mechanism of replacing old tree nodes (age greater than T) with placeholders serves to keep the local Merkle prefix tree free of old records outside its partition. Otherwise, it would slowly grow back to the full global tree. Keeping the lists of touched records by operations in a max T-sized FIFO-queue guarantees that all changes outside the local partition will be forgotten once they are old enough (age of T). This happens because every node outside the partition is considered for replacement once an operation that touched it reaches an age of T, and a node will always have the same age as the last operation that touched it.

GET operations are also affected by the introduction of partitions. Clients are supposed to obtain a single combined Merkle proof corresponding to the global state at a specific time. However, records are now spread over different system nodes which requires multiple GET replies. Suppose that all of the R records the client wants to obtain are spread over P partitions ($P \leq R$). For now, the client is assumed to know all sets of system nodes responsible for each partition[8]. We still adopt the old GET request strategy of sampling K nodes, now separately and concurrently for each of the P partitions. This will give us P different Merkle proofs, the union of which corresponds to the full proof the client wants to obtain. The difficulty lies in combining these Merkle proofs since they can be out-of-sync and have different root labels. To solve this, we again use Section 3.4's algorithm. Each system node attaches its current timestamp to its GET replies. If any two Merkle proofs obtained have an age difference greater than T, the older Merkle proofs must be replaced with more recent ones via more GET requests until this is no longer the case. afterwards, the client takes the latest Merkle proof (highest timestamp) as the *base* partition for the algorithm and attempts to extend it with the other proofs and their corresponding records. If all proofs are compatible, the client will be able to obtain a single combined proof as before, and all previous functionality of GET requests is retained.

With this section, we have achieved storage scalability (**G4**). Depending on the specific safety guarantees desired, the effective total storage S can scale with a $1/\log S$ factor or even a linear factor of the total storage capacity. All of the methods described here are efficient as per **G5**, and we retain all goals achieved by the previous steps.

4.3.6 Lazy auditing

We now describe an extension called *lazy auditing* and achieves the last and final goal of malicious starvation detection (**G7**). We note that similar mechanisms have already been proposed in other systems [25].

Even if the safety of EXECUTE operations is not affected, there is nothing stopping a node from incorrectly responding to a client's GET request. If such behaviour persists, the client might be *starved*, i.e., indefinitely prevented from reading correct data or having its EXECUTE requests pull through. To curb this malicious behaviour, we have the following:

- Every node must be able to provide a non-repudiable signature of its *GET replies* if so requested by a client (this can also be made mandatory for all GET replies).
- Every node keeps a list of all but the oldest Merkle tree root labels and their associated timestamps (a history, possibly stored as a blockchain). If a client requests it, nodes must

[8] One possible way to bootstrap clients involves storing the full node membership and associated partitions in the Merkle prefix tree and having the clients send a GET request of this information to the few nodes they know, assuming at least one correct node. The client then repeats this procedure with new nodes it becomes aware of, contrasting and corroborating the replies to obtain the true correct membership. Further detailing this mechanism is left for future work.

provide this list along with a non-repudiable signature of it.

If a client suspects it is being *starved*, it can request a constant-sized sample of nodes and obtain a sample of lists. If any two lists in the sample contradict each other, the client can present that as evidence that one of those nodes is Byzantine, via a special COMPLAIN request (a re-purposed EXECUTE request that removes the malicious nodes from the system). Similarly, if some *GET reply* message it has received in the past contradicts information in one of the lists in the sample, it can present that GET reply, its signature, and the sample of lists as evidence in a COMPLAIN request.

Whenever there is malicious starvation there must be such a contradiction (w.h.p., assuming the client's sample is of a large enough number of nodes), in which case the guilty Byzantine nodes will be detected and expelled from the system. Eventually, malicious starvation will stop, either from all Byzantine nodes being removed or having been dissuaded[9] from doing it. This achieves our last goal (**G7**).

[9] If we were to extend this to the full *permissionless* setting, we would make it cost of entry for system nodes would be so high when compared to any benefit of the temporary denial of service effect of starving a client that no Byzantine node would be inclined to do it.

Chapter 5

Implementation

Our design section provides a framework for a distributed, highly-scalable, statically-addressed key-value store based on Merkle prefix trees. To demonstrate the feasibility of our system and how real services can interact with it, we implemented a prototype in Rust and provide an example of monetary transactions in an online bank. Our implementation is divided in two parts: our Merkle prefix tree library and our reference system library. Both have been released on GitHub[1].

5.1 Merkle prefix tree library

The Merkle prefix tree library was developed to be used by our prototype but also independently by other projects. Hence, we provide a generic *Tree* structure with well-documented methods loosely following the Rust std library. For example, the signature of most methods follows that of Rust's standard hash map[2]. For our own purposes, we implement a specialized *HistoryTree* structure which transforms a normal *Tree* into a *T-history-aware* tree. The implementation is relatively straightforward and follows the design described in Section 4.3.5. We only explain how our implementation guarantees that tree nodes *outside* the partition will never *leak* and are always eventually replaced by a placeholder, without having to check the age of every node in the tree after each operation.

We have a current timestamp, a FIFO queue of size T returning the keys of records touched by operations, and a timestamp for each leaf and empty tree node allowing us to calculate its age, all as defined in Section 4.3.5. There are only two methods of the Merkle prefix tree that can be in an operation and modify the tree's topology (i.e., create or remove tree nodes). Those are the *insert* method when inserting a previously non-existing record, and the *remove* method, when removing a previously existing record:

[1] `https://github.com/mvidigueira/MasterProject`
[2] `https://doc.rust-lang.org/beta/std/collections/struct.HashMap.html`

- **Remove**: Takes a key k and a value v. Follows the path of the hash of k starting from the root of the tree until the end. If we find a leaf node n with the same key k, we replace n with a new empty node with the current timestamp. We then backtrack up to the root along the path and consider each internal node (travelling down was done by recursion, so this easily achieved). There are only three possible scenarios when considering an internal node:

    1. if the internal node has two empty nodes as children, we replace the internal node with the youngest empty node (the one with the highest timestamp).
    2. else if the internal node has a leaf and an empty node as children, we take the highest timestamp between them and replace the internal node with the leaf node but with this timestamp (possibly higher)[3].
    3. else, the internal node has another internal node as a child and we do not modify that internal node.

- **Insert**: Takes a key k and a value v. We follow the path of the hash of k starting from the root of the tree until its end. Depending on the node at the end of the path, there are three possible scenarios:

    1. If we reach an empty node, we replace it with a new leaf node with key k, value v, and the current timestamp.
    2. Else if we reach a leaf node with the same key k, we update its value to v and update its timestamp to the current timestamp.
    3. Else, we reach a leaf node n_{other} with a different key k_{other}. Let p be the longest common prefix in the hashes of k and k_{other}. We replace n_{other} with a path of internal nodes along p; the number of internal nodes on this path, when counting from the root, is equal to the length of p. The last internal node on this path has n_{other} as one of its children (same key, value, and timestamp), placed along the path of k_{other}, and a new leaf node (as in case 1) as its other child, placed along the path of k. Every other internal node created has a new empty node with the current timestamp as its other child.

Assuming that the Merkle prefix tree starts empty, and every change to the tree is applied via the provided methods (get, insert, remove), then every single internal, leaf, and empty node is created in the tree by the *insert* and *remove* methods as described above. This is the case in our system. Whenever a new operation is applied, all nodes that finally reach an age of T must be nodes that were created or had their timestamp changed by an insert or remove exactly T operations ago. This insert of remove method's argument key k must be in the *list of touched records* that was just popped from the FIFO queue, since the queue has size exactly T. The insert and remove methods also have something in common: every node they create or remove

[3] If the empty node was "younger" (had a higher timestamp), the leaf node's new age represents the more recent *proof of absence* of the removed node. We want this proof of absence to persist for T operations and not be stubbed too early, hence why the leaf node must take the higher timestamp.

from the tree (every topological change) is either a node in the path of the key (k) or its sibling. Therefore if for every key k in the popped list of touched records we check the age of every node and sibling in the path of k, we will have checked every single node in the tree that might have just reached an age of T. When doing so, if a leaf or empty node has an age of T and is not part of our partition, we replace it with a placeholder node of the same age. No node in the Merkle prefix tree that lies outside the partition ever reaches an age greater than T, and so there are no "leaks", i.e., no information stored unnecessarily that could lead to an uncontrollable growth of local memory. This justifies the statements made in Section 4.3.5.

5.2 System library

Our prototype has two main components: the client, and the server (system node). The client implements two main methods, *get_merkle_proofs* and *send_transaction_request*, which correspond respectively to the client-side part of the GET message request and the EXECUTE message request, as defined in Chapter 4. The *get_merkle_proofs* method takes the list of keys of the GET request and returns a combined Merkle proof. Similarly, *send_transaction_request* takes the arguments of the EXECUTE request and implements it. These methods can be called separately. This gives applications the freedom to, for example, perform a GET request, execute an operation locally to see its results or what records it touches, and perform another GET request before an EXECUTE request. Applications can also implement their own *get_merkle_proofs* method separately from the *send_transaction_request* method and vice-versa. The same guarantees given to correct clients are given as long as the new methods' messages follow the established protocol.

The server is able to serve multiple client requests simultaneously. Consistency is ensured via a read-write lock around the local Merkle tree. Both the client and the server make heavy use of Rust's async library[4] to parallelize both communication and execution, such as concurrently sending GET requests and waiting for GET replies on the client-side, or handling multiple GET and EXECUTE requests on the server-side. For messaging and secure communication, we make use of the *drop*[5] and *classic*[6] frameworks. We make use of the Wasmtime [7] crate's WebAssembly runtime to execute client operations in a sandboxed environment, and of the wasm-bindgen[8] crate in the generation of WebAssembly code from Rust. Our prototype includes an example of an operation written in Rust, a monetary transfer between two bank accounts, along with instructions on how to compile it to WebAssembly so that it can be used in our prototype.

In our prototype, the Total Order Broadcast (TOB) primitive is mocked by a separate centralized server (called the TOB server) which orders requests sent to it by clients and broadcasts

[4] `https://rust-lang.github.io/async-book/`
[5] `https://github.com/Distributed-EPFL/drop/tree/feature/io`
[6] `https://github.com/Distributed-EPFL/classic`
[7] `https://wasmtime.dev/`
[8] `https://rustwasm.github.io/wasm-bindgen/`

them in order to the servers handling client requests. This is sufficient for testing correctness.

Chapter 6

Evaluation

The main goal of our prototype and its evaluation is to corroborate the theoretical results of Chapters 3 and 4. We split the evaluation into two parts: safety and performance. Safety concerns whether the parts implemented by the prototype follow the specification. Performance evaluates the prototype's storage scalability and operation acceptance rates for multiple concurrent clients.

6.1 Safety evaluation

As of the writing of this document, our prototype includes 74 tests for the Merkle tree library and 24 tests for the system library. This includes:

- unit and integration tests for the regular Merkle prefix tree.
- integration tests for the *HistoryTree*.
- integration tests for multiple clients and servers.

The integration tests mainly focus on edge cases and the functionality introduced by the new T-history-aware trees (*HistoryTree*). The lack of a realistic TOB primitive precludes any meaningful end-to-end performance testing of our prototype. We cover an exhaustive list of scenarios of concurrent client operations, including operations on:

- non-intersecting sets of records. No operation should be rejected.
- intersecting sets of records that are not modified. No operation should be rejected.

- intersecting sets of records that are modified and make the followings operation invalid. The following operations should be rejected.
- intersecting sets of records that are modified by the first operation but restored to the previous state by a second, allowing a third operation that is expecting those records to be unmodified to still be accepted. The third operation should be accepted.

We similarly test *stubbing*, i.e., the replacement of Merkle tree nodes of age T with placeholders, checking that:

1. Nodes are not replaced too early (before age T). This is done querying the tree for proofs of inclusion or absence immediately before that proof is about to expire and be replaced by a placeholder.
2. Nodes outside the partition are replaced once their age is T. This can be checked in tandem with the previous test, by checking that nodes are replaced as soon as they reach an age of T.
3. Nodes belonging to the local partition are never replaced. Similarly done by querying the tree for all nodes in its partition after T or more operations have elapsed.

These tests confirm that the *HistoryTree* has an asymptotic memory complexity equal to that of a normal partition of Merkle prefix tree, for a constant T. In summary, our test results are in line with our theoretical results.

6.2 Performance evaluation

The primary goal of our performance evaluation is to estimate how effective our system is in storage scaling and handling concurrent client requests. Some of the questions we want to answer are the following:

1. Memory usage
    - What is the memory overhead of the *HistoryTree*?
    - How efficient is the storage partitioning in terms of stored data per node?
2. Operation acceptance rate
    - What are the effects of the history length T on the operation acceptance rate?

- What is the ideal history length T?

Due to the lack of a realistic TOB primitive, we do not perform any time-sensitive end-to-end experiments. Instead, we focus on testing memory usage, as well as operation acceptance rates in a scenario of multiple concurrent client requests.

For reference, all tests were conducted on a single 2.60GHz Intel i7-4720HQ CPU, running Ubuntu 18.04. Since no performance test is time-sensitive, all performance tests are run on a single thread using the Rust *async* library and the *tokio* crate[1]. This guarantees deterministic execution that is independent of the machine used. In particular, for tests with concurrent clients, we guarantee that exact number of concurrent clients by alternating between each client thread in succession, increasing the accuracy of our results.

6.2.1 Memory usage

We instantiate our system with the following parameters:

- The *history length* (T) is set to 4.
- System nodes are hashed to a random prefix and are attributed all prefixes to which they are closer. This approximately balances the number of records per partition while covering the entire tree.
- Every prefix is covered by exactly one node (no data loss tolerance).

All experiments start by creating 1.000 records, numbered 0 to 999, each associated with the 32-bit integer value 0. *Data* corresponds to the total size in bytes of the Merkle prefix tree in each node, including all internal, empty, placeholder, and leaf nodes. *Overhead* corresponds to all of the remaining structures of the *HistoryTree*, e.g., the history queue lists of touched records and the history queue of root labels.

The first experiment is shown on Figure 6.1. All 6 operations touch a single record, 0. We can see that the overhead for the first 3 steps remains relatively high for every node. This is due to the memory cost of the references of the 1.000 inserted records in the queue of *touched records list*. Each operation also pushes a single record (0) onto the queue, explaining the slight increase in overhead. Once 4 operations have elapsed since those records were inserted, the corresponding *list of touched records* is popped from the back of the queue and the memory is recovered, leading to the drop in overhead seen in the transition of $3 \rightarrow 4$.

Similarly, all nodes start with the full Merkle prefix tree, which is around 32KB in size. Upon the fourth operation, each node replaces all records of the starting 1000 outside its partition,

[1] `https://github.com/tokio-rs/tokio`

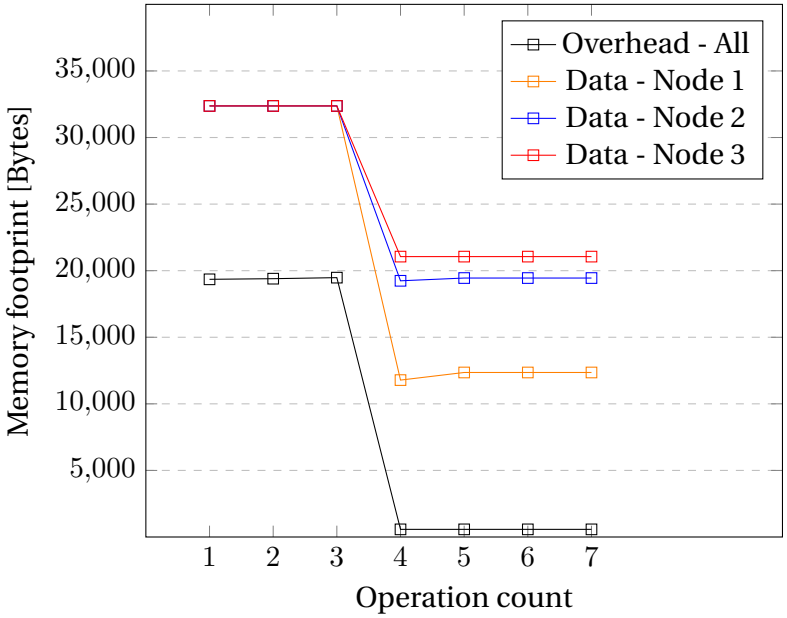

Figure 6.1: Effect of partitioning on memory usage (N=3, T=4).

except for record 0, inducing the drop in data stored. After the drop, each node's Merkle prefix tree occupies 17.6KB of space on average. The discrepancy in storage between nodes can be explained by the difference in the distribution of prefixes per node, as well as the number of records per partition, due to the uniform spread of each. Summing the sizes of the Merkle partitions in each node, we can see that it exceeds the size of the original tree (32KB). This is expected and is due to the repetition of internal nodes and placeholder nodes, i.e., the overhead of the Merkle prefix tree itself. It is particularly noticeable in this case since the values stored in each leaf are of small size (4 Bytes), and the memory cost of internal, empty, and placeholder nodes becomes apparent. In practice, we expect records to hold data of over 1KB large on average, and this would not occur.

Figure 6.2 corresponds to the same experiment but using 5 nodes instead of 3. The main difference lies in the amount of data stored on average per partition: after the transition of $3 \rightarrow 4$, each node's Merkle tree uses between 8KB and 15KB of space, averaging 11.7KB, which is significantly lower than the average of 17.6KB for 3 nodes.

Finally, we experiment with heavy and selective operations, i.e., operations that update a small set of records with relatively large values, shown in Figure 6.3. The fifth operation updates a single record belonging to node 3's partition with a 10KB value. All other operations are the same as previously. We can see that the data stored by all nodes increases by the same amount, 10KB, and remains approximately constant. After $T = 4$ operations have elapsed, all nodes except for node 3 replace that record with a placeholder, leading to the drop in data seen at ninth operation, while node 3 retains that data.

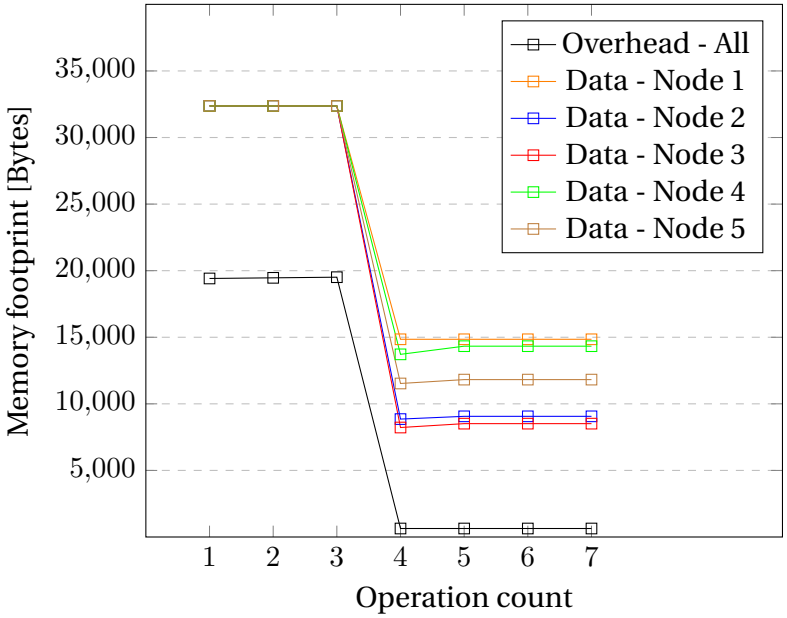

Figure 6.2: Effect of partitioning on memory usage (N=5, T=4).

We can approximate the storage overhead of the *HistoryTree* with the formula

$$\text{storage overhead} = T \cdot r \cdot (\bar{S} + o)$$

where T is the history length, r is the average number of records touched per operation, $\bar{S}$ is the average size of each record, o is the average overhead of each record in the HistoryTree, and R is the total number of records in the database. Using the limit of $r = 400$, a high estimate of $o = 200$B, and assuming $\bar{S} = 1$KB, supporting $T = 10.000$ concurrent client requests can be done using just 4.8GB of additional overhead. As an example, in a typical cryptocurrency with one-to-one account transfers, we would touch two records, one for each account, plus the record of the operation, corresponding to $r = 3$. Using the same values for $\bar{S}$ and o and the same additional memory cost of 4.8GB, the system would be able to handle up to 1.3 million concurrent clients.

6.2.2 Operation acceptance rate

Next, we evaluate the difference in acceptance rates of concurrent *GET and EXECUTE* operations under different history lengths (T). In order to measure this accurately, we complete every client's GET request before the first EXECUTE request is sent to the nodes.

In the first experiment (Figure 6.4), we initialize the system with 100 records. Each client operation picks 1 record (0 to 99) at random and changes it to a different value. Note that it is

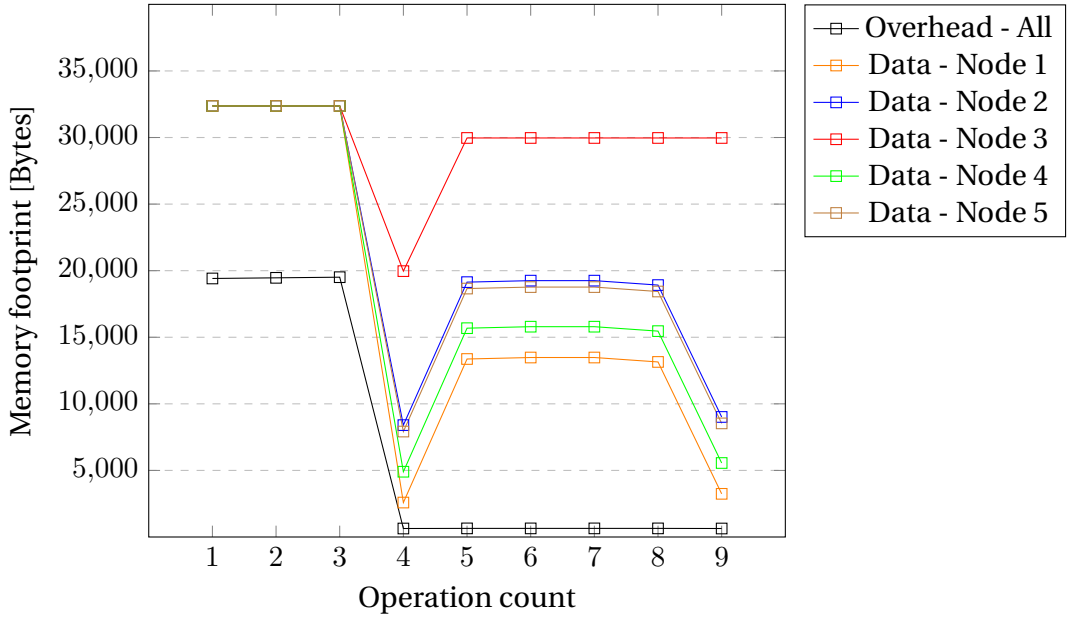

Figure 6.3: Evolution of memory usage with large and selective operations (N=5, T=3). The operation at time 5 updates a single record with a 10KB value.

inevitable for an operation to be rejected if it chooses a repeated value. In this particular case, the expected number of collisions for 100 operations and 100 possible records is approximately 36.6.

From the figure, we can see that the value of T caps the number of concurrent operations that can be accepted. This is expected since the root label of the Merkle tree of later operations will no longer be valid and the operations will be rejected. We can also see that the acceptance rate stabilizes around 63%, starting from $T = 80$. This is due to the limiting factor no being longer T, but the number of expected collisions between operations: the expected rate of successful operations is $\frac{100-36.6}{100} \approx 0.63$. Compare this to the second experiment, where operations choose records from a pool of 1000 (Figure 6.5). The expected number of collisions becomes significantly lower, approximately 4.8 for 100 operations. This is reflected by the acceptance rate stabilizing around $0.97 \approx \frac{100-4.8}{100}$, starting from $T = 100$.

Choosing T. In practice, we expect the collision rate to be low, and so the number of concurrent operations that are also *compatible* (i.e., do not collide) will also be close to the number of concurrent operations C. For any $T \geq C$, none of the compatible operations will be rejected, while if T is less than C, at most T will be accepted. Therefore, We should choose $T \leq C$ as high as possible, so long as there is available memory.

Estimating C. For a GET and subsequent EXECUTE operation, let:

1. R be the number of EXECUTE operations requested per second (rate).

2. L be the average time in seconds between the moment the first GET request message arrives at a node until the last node applies the changes of the EXECUTE request (read-write latency).

the average number of concurrent operations is given by $C = R \cdot L$. We recommend overestimating this value when choosing T to give leeway to slower than average clients or better handle periods of increased rate, if the memory is available.

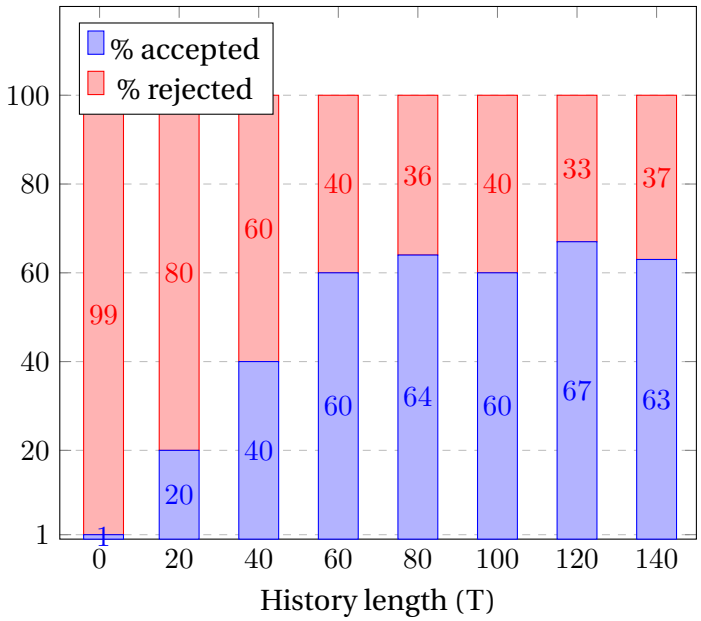

Figure 6.4: Variation of operation acceptance rate with history length. There are 100 concurrent operations each modifying 1 out of 100 possible records at random (high probability of collision).

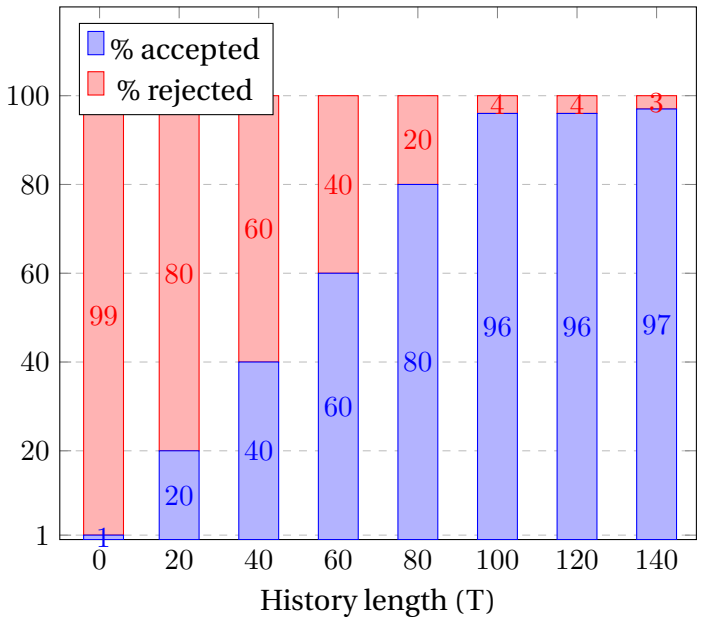

Figure 6.5: Variation of operation acceptance rate with history length. There are 100 concurrent operations each modifying 1 out of 1000 possible records at random (low probability of collision).

Chapter 7

Related Work

The vast majority of recent work in decentralized database systems has been in the context of *cryptocurrencies*. This is due to historical reasons as well as the financial interest in this type of application. However, cryptocurrencies make up a very small subset of the possible applications that our system can run. Furthermore, there are many assumptions and optimizations that can be done in that context that cannot be generalized to other types of data and operations, and so comparing performance directly can be unfair. In any case, we find it important to draw some comparisons and distinctions with some systems due to the similarity of some of the techniques involved. We are particularly interested in comparing storage scaling techniques, as that is the focus of this work.

Vault [24] is a recent cryptocurrency design based on Algorand [15] that employs many techniques similar to ours in order to minimize storage costs per participant. For example, checking if a transaction is valid is also done by verifying if its inputs are valid. Vault's transactions also have an *expiration date*, an interval in "time" during which they can be accepted, and after which they are unilaterally rejected. The key difference between Vault and our system is that it *completely* decouples recent transaction history from state storage. In order to handle a late concurrent client request, it keeps track of all recent operations, applies them to the Merkle proof (witness) of the request, and then verifies its validity. This implies that, instead of being executed once for the local state, operations might be executed multiple additional times to update late proofs. In the context of a cryptocurrency with relatively cheap, one-to-one account transfers, this is not a big problem. Most importantly, it allows Vault to rid itself of the overhead of storing state touched by recent transactions, which can be considered a worthwhile trade-off. However, our system handles generic data and operations, and repeating the execution of a computationally heavy operation would be unbearable. Furthermore, while Vault's algorithm to update witnesses works for witnesses with different leaf *values* (the record's value was modified), it does not specify if it handles absent leaves due the removal or insertion of a record between operations.

Ethereum [4] is one of the more famous decentralized application platforms and is closer in context to our work due to its ability to run *digital contracts*, similar to our operations. As far as we are aware, Ethereum does not have a final published version of its storage sharding mechanism. However, we find it worthwhile to mention that a similar approach to ours was listed in the proposals [12], but was ultimately abandoned due to censorship vulnerability concerns when combined with Ethereum's Proof of Work [11]. Another key difference between both systems is that Ethereum's operations access memory *dynamically*: they are not restricted to reading and writing records from a predefined, *static* set or range of addresses. This is a fundamental requirement of our system in order to curb the size of the Merkle proofs, which makes our technique incompatible with the current version of Ethereum.

Chapter 8

Conclusion

We have introduced, specified, and implemented a prototype for a Merkle-based, distributed, and universal key-value store. We have also defined a set of theoretical extensions and an algorithm for Merkle prefix trees and corresponding proofs (Chapter 3). These are implemented into our system, playing a fundamental role in storage scalability and throughput, and can also be used independently in other systems and scenarios. The system as stated in the Design (Chapter 4) is efficient, byzantine fault-tolerant, and flexible to the user. The prototype does not implement all of the features stated, in particular, K fault-tolerant GET requests, *lazy auditing*, and a full-fledged Total Order Broadcast primitive, making it impossible to obtain meaningful end-to-end performance results. However, as it stands, it is completely capable of demonstrating all of the remaining features and, more importantly, it fully corroborates our theoretical results (Section 3.4).

Future work includes completing the prototype with the remaining features as described, as well as implementing a competitive Total Order Broadcast primitive in order to obtain accurate performance measurements that are directly comparable to other systems'. Other medium-term goals are to research and extend the system to the permissionless setting, including client bootstrapping and Sybil resistance mechanisms. Bootstrapping clients that know only a small set of system nodes has been part of ongoing work and its design is nearly complete. Although there are some remaining problems which are yet to be fully researched, we are confident that the core mechanisms of our system are compatible with the permissionless setting, and that extension should be possible. High-priority long term goals include introducing fast *transaction sharding* for throughput scalability, i.e., splitting the transactions for processing over different shards instead of a single one, as is being researched for systems like Ethereum [12]. So far, our system stores all records under a single Merkle prefix tree, hashing the full key. Another way to store those records is to hash different parts of the key, e.g., the slash-separated domains defined in Section 4.3.1, and have a hierarchy of Merkle trees (Merkle prefix trees as nodes of other Merkle prefix trees). This could allow records sharing a similar prefix on their keys to also share a long prefix on their hash, making them more likely to be stored in the same system node, which has

some advantages. It would also be interesting to research and analyze such a scheme.

Finally, we believe that extending our system with future work would make it competitive with the current state-of-the-art. Should a sufficiently fast Total Order Broadcast primitive arise, our approach's flexibility, efficiency, and potential for scaling might perhaps allow it to reach a planetary scale and compete with traditional cloud services for some types of online businesses. As such, we think that it is a worthwhile research direction to pursue.

Bibliography

[1] https://digiconomist.net/bitcoin-energy-consumption. [Online; accessed 19-May-2020].

[2] https://en.wikipedia.org/wiki/Atomic_broadcast. [Online; accessed 29-May-2020].

[3] *2018 Cloud Computing Survey*. Tech. rep. IDG Communications, Inc., 2018.

[4] Vitalik Buterin. *Ethereum: A next-generation smart contract and decentralized application platform*. Accessed: 2016-08-22. 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[5] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[6] Usman W Chohan. "The double spending problem and cryptocurrencies". In: *Available at SSRN 3090174* (2017).

[7] *Cloud market share Q4 2019 and full-year*. Tech. rep. Canalys, 2019.

[8] Deadalnix. *Using Merklix tree to shard block validation*. https://www.deadalnix.me/2016/11/06/using-merklix-tree-to-shard-block-validation/. [Online; accessed 19-May-2020].

[9] Xavier Défago, André Schiper, and Péter Urbán. "Total order broadcast and multicast algorithms: Taxonomy and survey". In: *ACM Computing Surveys (CSUR)* 36.4 (2004), pp. 372–421.

[10] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[11] *Ethereum Sharding FAQ*. https://github.com/ethereum/wiki/wiki/Sharding-FAQ#can-we-force-more-of-the-state-to-be-held-user-side-so-that-transactions-can-be-validated-without-requiring-validators-to-hold-all-state-data. [Online; accessed 19-May-2020].

[12] *Ethereum Sharding roadmap*. https://github.com/ethereum/wiki/wiki/Sharding-roadmap/. [Online; accessed 19-May-2020].

[13] *Failure Detectors*. http://www.cs.yale.edu/homes/aspnes/pinewiki/FailureDetectors.html. [Online; accessed 29-May-2020].

[14] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[15] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. “Algorand: Scaling byzantine agreements for cryptocurrencies”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 51–68.

[16] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[17] S. Goldwasser and M. Bellare. *Lecture Notes on Cryptography*. `http://cseweb.ucsd.edu/~mihir/papers/gb.html`. 1996-2001.

[18] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. “AT2: asynchronous trustworthy transfers”. In: *arXiv preprint arXiv:1812.10844* (2018).

[19] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. “Scalable Byzantine Reliable Broadcast”. In: *33rd International Symposium on Distributed Computing (DISC 2019)*. Ed. by Jukka Suomela. Vol. 146. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 22:1–22:16. ISBN: 978-3-95977-126-9. DOI: 10.4230/LIPIcs.DISC.2019.22. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/11329`.

[20] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. “The consensus number of a cryptocurrency”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 307–316.

[21] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. “Eclipse Attacks on Bitcoin’s Peer-to-Peer Network”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 129–144. ISBN: 978-1-939133-11-3. URL: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman`.

[22] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[23] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine generals problem”. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 203–226.

[24] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. *Vault: Fast Bootstrapping for the Algorand Cryptocurrency*. Cryptology ePrint Archive, Report 2018/269. `https://eprint.iacr.org/2018/269`. 2018.

[25] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. “{CONIKS}: Bringing Key Transparency to End Users”. In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 383–398.

[26] David Mosberger. “Memory consistency models”. In: *ACM SIGOPS Operating Systems Review* 27.1 (1993), pp. 18–26.

[27] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Accessed: 2015-07-01. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[28] Marshall Pease, Robert Shostak, and Leslie Lamport. "Reaching agreement in the presence of faults". In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 228–234.

[29] Ben Saunders. *Who's Using Amazon Web Services?* https://www.contino.io/insights/whos-using-aws. [Online; accessed 18-May-2020]. 2020.

[30] Bill Streeter. *New Tech Platforms Hold the Key to Retail Banking's Future*. https://thefinancialbrand.com/93779/traditional-bank-it-cloud-computing-real-time-fintech/. [Online; accessed 18-May-2020]. 2020.

[31] *The Stateless Client concept*. https://ethresear.ch/t/the-stateless-client-concept/172. [Online; accessed 19-May-2020].

[32] Yongge Wang. "Byzantine Fault Tolerance in Partial Synchronous Networks". In: (2020), pp. 2–3.