Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Using Blockchain for Tamper-Proof Broadcast Protocols

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Jonas Spenger
geboren am:
geboren in:

Gutachter/innen: Prof. Dr. Alexander Reinefeld
Prof. Dr. Björn Scheuermann

eingereicht am: ..... verteidigt am: .....

Using Blockchain for Tamper-Proof Broadcast Protocols

by Jonas Spenger

Master's Thesis

Submitted in partial fulfilment of the requirements for the degree of Master of Science (M. Sc.) in the Department of Computer Science at the Humboldt University of Berlin

First Supervisor: Prof. Dr. Alexander Reinefeld
Second Supervisor: Prof. Dr. Björn Scheuermann

May, 2020

Abstract

We present the tamper-resistant broadcast abstraction of the Bitcoin blockchain, and show how it can be used to implement tamper-resistant replicated state machines. The tamper-resistant broadcast abstraction provides functionality to: broadcast, deliver, and verify messages. The tamper-resistant property ensures: 1) the probabilistic protection against byzantine behaviour, and 2) the probabilistic verifiability that no tampering has occurred.

In this work, we study various tamper-resistant broadcast protocols for: different environmental models (public/permissioned, bounded/unbounded, byzantine fault tolerant (BFT)/non-BFT, native/non-native); as well as different properties, such as ordering guarantees (FIFO-order, causal-order, total-order), and delivery guarantees (validity, agreement, uniform). This way, we can match the protocol to the required environment model and consistency model of the replicated state machine.

We implemented the tamper-resistant broadcast abstraction as a proof of concept. The results show that the implemented tamper-resistant broadcast protocols can compete on throughput and latency with other state-of-the-art broadcast technologies. Use cases, such as a tamper-resistant file system, supply chain tracking, and a timestamp server highlight the expressiveness of the abstraction.

In conclusion, the tamper-resistant broadcast protocols provide a powerful interface, with clear semantics and tunable settings, enabling the design of tamper-resistant applications.

Acknowledgements

I wish to express my gratitude to my advisor Dr. Florian Schintke at the Zuse Institute Berlin, for the invaluable feedback, suggestions, discussions, knowledge, and guidance throughout the entirety of this project. I would also like to thank my first supervisor Prof. Dr. Alexander Reinefeld at the Humboldt University of Berlin and the Zuse Institute Berlin, for feedback on earlier versions and refereeing of this thesis. Furthermore, I wish to thank my second supervisor Prof. Dr. Björn Scheuermann at the Humboldt University of Berlin, for the involvement and refereeing of this thesis.

At last, I wish to acknowledge a selection of influential and important sources to this work. Much of the theory, notation, and style of this thesis surrounding broadcast and consensus is based on the book *Introduction to Reliable and Secure Distributed Programming* by Cachin, Guerraoui, and Rodrigues [1]. An important source for the Bitcoin blockchain was the book *Mastering Bitcoin: Programming the open blockchain* by Antonopoulos [2], and the original bitcoin paper *Bitcoin: A Peer-to-Peer Electronic Cash System* by Nakamoto [3]. Inspiration for studying blockchain abstractions stemmed from reading about the Virtualchain, described in *Blockstack: A global naming and storage system secured by blockchains* by Ali, Nelson, Shea, and Freedman [4]. The motivation for this work originated from our ongoing work on a blockchain-based file system at the Zuse Institute Berlin [5].

Contents

Chapter 1

Introduction

1.1 Motivation

Blockchain technology enables the design of decentralised and tamper-proof applications. Because of the growing demand in such technology, there is also a growing demand in understanding the fundamental technology. This was confirmed during our ongoing work at designing a blockchain-based tamper-proof file system [5]. One way to generalise blockchain-based technology is by the concept of *tamper-proof replicated state machines* and *tamper-resistant replicated state machines*. In this thesis, we study how tamper-resistant (not tamper-proof) replicated state machines can be implemented using the *blockchain* and *blockchain-based tamper-resistant broadcast protocols*.

The term *tamper-proof* generally refers to protection against byzantine behavior, such that any attempts at tampering (unauthorized changes) are not possible (c.f. byzantine fault tolerance [1, ch. 2.2.6]). In an attempt to capture the distinct characteristic of blockchains in our definition, we define tamper-proof as: 1) the protection against byzantine behavior, such that any attempts at tampering are not possible, and 2) the verifiability that no tampering has occurred (i.e. verifiable proof that it is untampered with). The second rule addresses the case when no single participating instance can be trusted (including the local instance), and enables independent third party verification of the proof.

Ideally, we would like our applications to be *tamper-proof*. The Bitcoin blockchain, however, should not be considered tamper-proof, due to its probabilistic nature. We instead choose to label it as *tamper-resistant* or *probabilistic tamper-proof*. We define this term as: 1) the probabilistic protection against byzantine behaviour, and 2) the probabilistic verifiability that no tampering has occurred. Probabilistic and probabilistically should be read as: *with high probability*. In this thesis we build upon the tamper-resistant Bitcoin blockchain, and discuss tamper-resistant protocols.

The tamper-proof and tamper-resistant property cannot be guaranteed for the general use-case of replicated state machines, unless further assumptions (restrictions) are made on the environment. For general functionalities, we require at minimum an honest majority for the secure execution of the protocols. In this work, we assume that at most one-third of the participating processes are faulty (byzantine failure model) (including the local instance).

We are motivated by the use case of tamper-resistant replicated state machines. A *replicated state machine* is a program which is replicated on several processes, such that each correct instance exhibits the same behaviour [1, ch. 6]. We characterise the tamper-resistant property of our *tamper-resistant replicated state machine* abstraction as:

- *Tamper-resistant replicated state machine:* The replicated state machine is probabilistically protected against tampering, and the output of the replicated state machine is probabilistically verifiable.

In this thesis, we study *tamper-resistant broadcast protocols* using the Bitcoin blockchain. These can be used to implement tamper-resistant replicated state machines. The tamper-resistant broadcast abstraction consists of functionality to: broadcast a message; deliver a message broadcast by another participant; and verify the authenticity of a message and the message history.

The presented work is a joining of concepts from distributed systems (replicated state machines, consensus, broadcast, byzantine fault tolerance) with concepts from blockchain. The tamper-proof property extends the concept of byzantine fault tolerance (protection against tampering), with the verifiability that no tampering has occurred (and tamper-resistance extends probabilistic byzantine fault tolerance). The verification allows for independent third party verification, and does not require local trust assumptions.

Our work differentiates from other work, by being a generalisation of the problem. The tamper-resistant (and tamper-proof) replicated state machine is a generalisation of blockchains and of blockchain-based applications. This allows us to study various environment models, and various safety and liveness properties, giving us a broader picture of the problem.

We hope to accomplish two goals: show how to design blockchain-based tamper-resistant applications and discuss important aspects thereof.

1.2 Problem Statement

The design of *blockchain-based tamper-resistant applications* is complex. It requires extensive knowledge of the properties and functions of the blockchain, and how to use these for the desired behaviour. This is often not easy as certain applications require properties and functions that are not directly derived from the blockchain, but rather from complex interactions with the blockchain.

In this thesis, we study the design of *tamper-resistant broadcast protocols* using the Bitcoin blockchain, for various environment models and safety and liveness properties.

The tamper-resistant broadcast protocols can be used to implement *tamper-resistant replicated state machines*, and, in turn, *tamper-resistant applications*. The environment model of the broadcast protocol can be matched with the required environment model of replicated state machine. The safety and liveness properties of the broadcast protocol can be matched with the required consistency model of the replicated state machine.

1.3 Outline

The following is an outline of this thesis, and lists the contributions that go towards answering the proposed problem statement:

- In Chapter 2, Background, we review previous work, and discuss how it relates to our work.
- In Chapter 3, Blockchain Ledger, we present and discuss an abstraction of the Bitcoin blockchain (Bitcoin Core API) and its properties.
- In Chapter 4, Tamper-Resistant Broadcast, we discuss and implement blockchain-based tamper-resistant broadcast protocols. We look at different environment models of broadcast protocols, in particular:
  - *Public tamper-resistant broadcast protocols:* protocol is publicly available, anyone can participate.
  - *Permissioned tamper-resistant broadcast protocols:* protocol is open to a known set of participants.
  - *Non-native permissioned tamper-resistant broadcast protocols:* protocol is permissioned and may use direct links or other protocols for exchanging messages.

  Furthermore, we look at different broadcast safety and liveness properties [1]:
  - *No creation*
  - *No duplication*
  - *Validity*
  - *Agreement*
  - *FIFO-order*
  - *Causal-order*
  - *Total-order*
  - *Uniform*

  and performance properties:
  - *Throughput*
  - *Latency*
- In Chapter 5, Evaluation, we evaluate the protocols on throughput and latency, running a workload on a distributed cluster, spanning over three physically separated locations.
- In Chapter 6, Use Cases, we show how to implement tamper-resistant replicated state machines using the tamper-resistant broadcast abstraction, and apply this to four use cases of blockchain technology [6]: data management and the secure storage of data; integrity verification such as tamper-resistant and verifiable storage of data (time stamping); blockchain virtualisation; supply chain, for example logistics providing origin tracking and identifying counterfeit products.

1.4 Significance of the Study

The presented work aids the design of tamper-resistant applications. Such applications will permeate sectors with an emphasis on trust, and become more commonplace as we transition from using human-centric and centralised forms of trust (e.g. notary offices, banks), to decentralised trust-networks (e.g. Bitcoin, Tendermint). This includes and already applies to [6]: supply chain logic and tracking; file archival of financial data; land registry. We hope that our findings will help the understanding of the problem, and help the design of new applications in these and more fields. In particular, the tamper-resistant broadcast protocols can be used to implement *tamper-resistant replicated state machines*, thus applicable to the design of arbitrary tamper-resistant applications.

1.5 Assumptions

In order to limit the scope of this thesis, we make the following assumptions:

- *Fundamental system model:* We consider the following fundamental properties of the system and environment for this project [7, ch. 2.4]:
  - *Interaction model:* Asynchronous distributed system model, i.e. no assumptions can be made about the time —process execution speeds, message transmission delays and clock drift rates are arbitrary and unknown.
  - *Failure model:*
    * Communication channel omission failure model [1, ch. 2.2.3]: A sent message may never arrive at the receivers incoming message buffer;
    * Byzantine failure model (arbitrary failure model) [1, ch. 2.2.6]: faulty processes can exhibit arbitrary behaviour, no assumptions is made on their behaviour. This is our main failure model, unless explicitly mentioned otherwise. A correct process does not crash.
    * We will also under certain circumstances (explicitly mentioned) consider the process crash-stop failure model [1, ch. 2.2.2]: faulty processes execute the protocol correctly until crashing and stopping.
  - *Security model / threat model:* The system is exposed to the threat of interactions with an adversarial process.
- *Bitcoin blockchain:* We assume that the underlying blockchain system is the Bitcoin blockchain [2, 3]. We will only consider the functionality as provided by the Bitcoin Core API [8]. Furthermore, we will only consider zero-fee transactions.
- *Verification proof in tamper-resistant protocols:* We refrain from explicitly returning a *proof* to the verification requests in the defined modules and protocols. Although returning the proof is necessary for the case when local instances cannot be trusted, we believe that adding such functionality is trivial. For example, it would be a sufficient proof to return the variables that are evaluated during the verification protocol.

Chapter 2

Background

The following is an introduction to the research on blockchain-based tamper-resistant broadcast protocols. We include an introduction to the related theory, general background information of the outlined problem, as well as a review of the related work.

2.1 Theory

This section introduces the relevant theory from distributed systems: distributed programming abstractions, the broadcast abstraction and replicated state machines.

2.1.1 Distributed Programming Abstractions

A *distributed program* consists of inter-communicating entities that perform computations [1, ch. 1.2]. The computing entities are referred to as *processes*, and the communication paths between them as *links*.

The algorithms we discuss are based on an *asynchronous event-based composition* model [1, ch. 1.4]. Each process hosts a set of components. The components communicate through the exchange of *events*. The components provide an interface, consisting of *request* events and *indication* events. The two types of events differ in direction. Request events are triggered by a second component, requesting a service to be executed at the first component. Indication events are triggered by the first component, indicating or delivering information to a second component.

Modules are made up of one or many modules [1, ch. 1.4]. They provide request and indication events. The *properties* of the module describe the properties of its behaviour. The behaviour of a module can be described as the behaviour observed by another component interacting through the module interface. One type of property are guarantees, such as the guarantee that an action is performed by all processes.

The *fundamental system model* explicitly states the assumptions made about the underlying system, and generalises on the possible interactions in the model [7, ch. 2.4]. Our assumptions of the system's interaction model, failure model, and security model, can be found in Section 1.5.

2.1.2 Distributed Agreement Problem

The *distributed agreement problem* [1, ch. 1.2] involves a set of processes to agree on certain facts. This could be finding an agreement on the order of certain events. But also, on finding a common sequence of actions to take. Distributed agreement can thus be used for the cooperation between the distributed processes.

Consensus and *broadcast* are distributed agreement problems and a focal point to this thesis. Consensus is the problem of agreeing on a value out of a set of proposed values [1, ch. 5]. The consensus module has two events: *propose*, a request to propose a value; *decide*, an indication event that the distributed processes have decided and achieved agreement on a value. Broadcast is the problem of sending a message to all processes including oneself [1, ch. 3]. It has two events: *broadcast*, a request to broadcast a message to all processes; *deliver*, an indication to deliver a message from a process. The consensus problem and the total-order broadcast problem can be shown to be equivalent [9].

2.1.3 Broadcast Abstraction

Broadcast is the problem of sending a message to all processes including oneself The *broadcast abstraction* interface is shown in Module 1 [1, ch. 3]. It has two events: *broadcast*, a request (application to module) to broadcast a message to all processes; *deliver*, an indication (module to application) to deliver a message from a process [1].

Module 1 Broadcast

Module name: Broadcast, **instance** b

Events:

E1 *Request:* < b, Broadcast | message >: Broadcast a *message* to all processes.

E2 *Indication:* < b, Deliver | p, message >: Deliver a *message* broadcast by process *p*.

Among the characteristics of broadcast protocols that we will use in later chapters, we will look at the following safety and liveness properties. Safety properties state that "something will not happen", liveness properties state that eventually "something must happen" [10]. Properties and description with minor adaptions from Cachin et al. [1]:

- *No duplication:* No message is delivered more than once.
- *No creation:* If a process delivers a message with sender s, then the message was previously broadcast by process s.
- *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

[1] By contrast, the consensus abstraction interface has the events: *propose*, a request to propose a value; *decide*, an indication event that the distributed processes have decided and achieved agreement on a value [1, ch. 5].

- *Fair Validity:* If a process repeatedly broadcasts a message, then eventually the message is delivered by every correct process.

- *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

- *Uniform agreement:* If a message m is delivered by some process (correct or non-correct), then m is eventually delivered by every correct process.

- *Total-order delivery:* Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2, then q delivers m_1 before m_2.

- *Uniform total-order delivery:* Let m_1 and m_2 be any two messages and suppose p and q are any two processes (correct or faulty) that deliver m_1 and m_2. If p delivers $m1$ before $m2$, then q delivers m_1 before m_2.

- *FIFO-order delivery (first in, first out):* If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

- *Causal-order delivery:* For any message m_1 that potentially caused a message m_2, i.e. $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1 [2].

The performance properties evaluate the implementations and the physical machines running the implementations, we will evaluate the following:

- *Latency:* The time between the broadcast request event and deliver indication event of a message (for the sending process). We define an implementation to be low latency, if the latency is comparable (within 50%) to other high-performance specialised systems for the same task.

- *Throughput:* The rate at which messages are delivered, measured in number of messages per second, or, bytes per second (total for entire network). We define an implementation to be high throughput, if the throughput is comparable (within 50%) to other high-performance specialised systems for the same task.

Lastly, we would like to define the *tamper-resistant* property:

- *Tamper-resistant:* The broadcast messages, and the delivered message history, are probabilistically[3] protected against tampering, and the delivered messages are probabilistically verifiable[4].

[2] The causal-order property specifies that messages are delivered such that the order respects the happened-before relationship. The happened-before relationship can be described as follows [1, ch. 3.9.4]: m_1 happened before m_2, i.e. $m_1 \rightarrow m_2$, if: some process broadcast m_1 before m_2; or if some process delivered m_1 before broadcasting m_2; or if there exists a message m' such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$.

[3] With high probability.

[4] A verification request returns "Valid" (with high probability), if the message has not been tampered with and is part of the untampered message history, else "NotValid".

2.1.4 Broadcast Models

We can label variations of the broadcast implementations by their properties. This is a selection of broadcast models that are mentioned throughout this document [1]:

- *Best-effort broadcast:* No creation, no duplication, validity.
- *Reliable broadcast:* No creation, no duplication, validity, agreement.
- *Uniform reliable broadcast:* No creation, no duplication, validity, uniform agreement.
- *FIFO-order (reliable) broadcast:* No creation, no duplication, validity, agreement, FIFO order.
- *Causal-order (reliable) broadcast:* No creation, no duplication, validity, agreement, causal order.
- *Total-order (reliable) broadcast:* No creation, no duplication, validity, agreement, total order.
- *Uniform total-order (reliable) broadcast:* No creation, no duplication, validity, uniform agreement, uniform total order.
- *Uniform causal-order total-order (reliable) broadcast:* No creation, no duplication, validity, uniform agreement, uniform total order, uniform causal-order.

2.1.5 Broadcast Environment Models

We can label different environments of the broadcast implementations by their properties. Of particular interest are the following four properties:

- *Public/permissioned:* The protocol is publicly available, anyone can participate (public); or, the protocol is only available to known (authorised) participants (permissioned).
- *Bounded/unbounded:* The number of participants during an epoch (voting period) is known and finite (bounded); or, the number of participants during an epoch is unknown and potentially infinite (unbounded).
- *BFT/non-BFT (byzantine fault tolerant):* The protocol tolerates arbitrary faults up to $< 1/3$ of participants (BFT); or, the protocol tolerates fail-crash-stop faults up to $< 1/2$ of participants (non-BFT). If there are more faulty processes than the limit, we define that there are no liveness or safety guarantees for the BFT model, and the safety guarantees still hold but not the liveness guarantees for the non-BFT model.
- *Native/non-native:* The messages are broadcast via the blockchain, and consensus is via blockchain (native); or, the messages are broadcast via direct links or P2P (not via blockchain), and consensus is not via blockchain (non-native).

We will study the following three broadcast environment models:

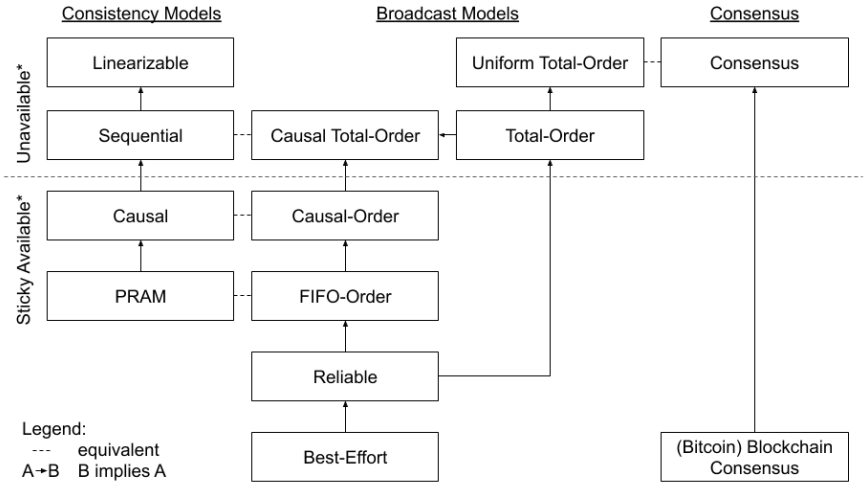

Figure 2.1: Relationship of consistency [11, 12, 13], broadcast [9], and consensus [9].

- *Public tamper-resistant broadcast protocols:* public, unbounded, BFT, native.
- *Permissioned tamper-resistant broadcast protocols:* permissioned, bounded, non-BFT, native.
- *Non-native permissioned tamper-resistant broadcast protocols:* permissioned, bounded, non-BFT, non-native.

We chose these environment classes as each represents a typical use-case. The *public tamper-resistant broadcast protocols* are suitable for public tamper-resistant applications, e.g. public ownership tracking, cryptocurrency. The use-case for *permissioned tamper-resistant broadcast protocols* are private tamper-resistant applications that are publicly open for auditing, but not publicly open for participation, e.g. supply chain tracking. *Non-native permissioned tamper-resistant broadcast protocols*, can be used for private tamper-resistant applications that require the use of non-native techniques to achieve higher throughput or lower latency than what is provided by the blockchain, e.g. secure storage of data, integrity verification.

2.1.6 Relations and Limits of Consistency, Consensus, and Broadcast

The relationship between consistency [11, 12, 13], broadcast [9], and consensus [9] is shown in Figure 2.1. The figure shows the equivalence of the models through dotted lines. The hierarchy of the models is shown through the arrows, pointing from weaker to stronger models. The figure is divided into sticky availability and unavailability [11, 12]. Sticky availability is defined as the ability to make progress and read previous progress under indefinitely long network partitions, if the user sticks to the same server [12]. If the user switches server, then the

ability to make progress and read previous progress is no longer guaranteed. Unavailability is defined as the impossibility to make progress due to certain network partitions.

The broadcast models which we discuss have equivalent counterparts in consistency models. The FIFO-order is equivalent to the PRAM consistency model, causal-order is equivalent to the causal consistency model, and the causal-order total-order broadcast is equivalent to the sequential consistency model. For example, the sequential consistency model requires: a single total order (SINGLEORDER), the FIFO-order (PRAM), and return value consistency [13]. This can be achieved using the causal-order total-order broadcast. The uniform total-ordered broadcast is equivalent to the consensus abstraction [9]. The (Bitcoin) blockchain consensus model is weaker in the hierarchy compared to the consensus model. The blockchain provides only probabilistic eventual consistency, compared to eventual consistency of consensus.

The concept of sticky availability and unavailability is closely related to the CAP theorem, stating that: a distributed system has to sacrifice one of consistency, availability, or partition tolerance [13]. The Bitcoin blockchain sacrifices consistency, it uses a probabilistic consensus algorithm that can cause forks and temporary inconsistencies. This is where it differs to typical consensus protocols, such as ZAB [14], which sacrifice the availability to make progress (liveness properties) rather than sacrificing consistency (safety properties). In order to achieve total-order using the Bitcoin Blockchain, we will have to either: assume that forks never exceed a certain length, which we call bounded fork length (i.e. assume that network partitions are bounded in time), this is perhaps reasonable but not strictly true; or, use other consistency preserving off-chain mechanisms to agree on a total-order.

There are further limits of what can be achieved under different broadcast environment classes. For example, no protocol can reach consensus if the number of participants is unknown and the number of faulty processes is unbounded (for BFT failure model) [15]. We could argue, that for a public environment model, we cannot bound the number of faulty and potentially byzantine processes. This is the case for the Bitcoin blockchain. The blockchain circumvents this impossibility, by providing a probabilistic eventual consensus protocol. It is also the case for our public tamper-resistant broadcast environment model. Implying, that that we cannot achieve uniform total-order broadcast with unbounded number of participants. We can argue further, that no protocol exists (for BFT and non-BFT) that can reach consensus if the number of participants during an epoch (i.e. a voting period) is unknown. This follows from the impossibility of defining a quorum for an unknown and unbounded set of participants. That is part of the motivation, why we also consider permissioned and bounded non-BFT protocols in Chapter 4, as it allows us to define quorums of correct processes during an epoch.

2.1.7 Replicated State Machines

A *replicated state machine* is a program which is replicated on several processes, such that each correct instance exhibits the same behaviour [1, ch. 6]. This can be achieved by using a total-order broadcast module (with properties: no creation, no duplication, validity, agreement, total order), that orders and delivers the same sequence of commands to all processes, and if the state machine is

deterministic.

It is not strictly necessary to have a total-order of broadcast messages for replicated state machines. Certain problems allow relaxations such that the replicated service still exhibits correct behaviour. A partial order of commands may suffice, this is the case if the commands are commutative [16, sec. 2.3]. For example, if implementing an eventually consistent replicated set, then the order of insertion would not matter. For this, a reliable broadcast (with properties: no creation, no duplication, validity, agreement) would suffice.

In later chapters, we will show that a tamper-resistant replicated state machine can be implemented using a tamper-resistant total-order broadcast.

2.2 General Background Information

This section introduces general background knowledge on blockchain, including: cryptography; the Bitcoin blockchain; saving data to the blockchain; and other blockchain technologies.

2.2.1 Cryptography

Crpytography is the study of protocols to aid the confidential communication between trusted parties [17]. Important concepts from cryptography that are used in blockchain are one-way hash functions, the Merkle tree hash, and digital signatures using public-key cryptography.

2.2.1.1 One-way Hash Functions

A *hash function*, is a function h that maps data x to a fingerprint $y = h(x)$ of the data [17, ch. 5]. The fingerprint is typically fixed length and shorter than the data. The fingerprint represents the data which produces it, much like a human fingerprint represents the person who produces it.

A *one-way hash function* is a function that is easy to compute but hard to invert. For example, given a one-way hash function and a fingerprint, it would be *practically infeasible* to find what data hashes to the fingerprint. Practically infeasible is generally understood to mean: the best strategy, to our knowledge, involves brute-force searching every possible input when searching for a specific output.

The bitcoin blockchain uses one-way hash functions SHA256 (Secure Hash Algorithm) and RIPEMD160 (RACE Integrity Primitives Evaluation Message Digest) [2, ch. 4].

A *Merkle tree* is a binary tree, in which every leaf node is the hash of some data value, and every inner node is the hash of the concatenation of its children [17, ch. 9]. The root of the Merkle tree (sometimes referred to as the Merkle tree hash) is a single hash value. The Merkle tree hash is used in Bitcoin to represent a sequence of hash values by a single hash value, the Merkle tree root. To prove that a hash value was one of the leaf nodes of the Merkle tree, one has to calculate the hash values on the path from the leaf node to the root node, this has complexity $\mathcal{O}(\log n)$ for a total of n hash values.

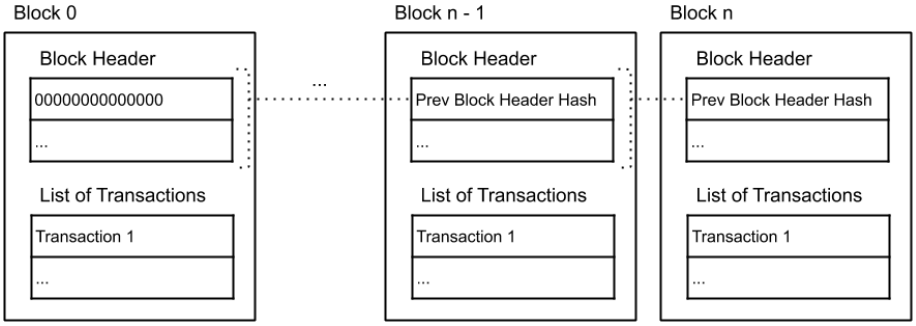

Figure 2.2: The bitcoin blockchain, forming a chain of blocks starting from the genesis block (block 0) to the most recently added block (block n).

2.2.1.2 Digital Signatures

A *digital signature* is a method for signing a message such that the authenticity of the message can be verified by the receiver [17, ch. 8]. Public-key cryptography consists of a key-pair (K, k)—a private key K and public key k—and a signing and a verification algorithm. The signing algorithm sig_K is private, whereas the verification algorithm ver_k is public.

The private key holder can sign a message m and produce a signature $y = sig_K(m)$. Any receiver of the message m and signature y can, together with the public-key verification algorithm, verify if a signature was produced by the corresponding private-key. The verification function $ver_k(m, y)$ outputs true, if $y = sig_K(m)$, else false. A message with a verified signature is considered authorized by the private key holder, because it is practically infeasible to calculate a correct signature without access to the private key.

The bitcoin blockchain uses the Elliptic Curve Digital Signature Algorithm [2, ch. 6] (ECDSA), on the basis of public-key cryptography and elliptic curve cryptography [2, ch. 4].

2.2.2 Bitcoin Blockchain

Blockchain technology and the *Bitcoin Blockchain* was first described in 2008 [3]. *Bitcoin* is a decentralised transactional system, and relies on cryptography for secure payment authorisation of the Bitcoin currency. No centralised authority or financial institution is needed for transactions and payments between clients. *Bitcoin-nodes* participate in a distributed consensus protocol to achieve this. *Full bitcoin-nodes* verify and keep the entire bitcoin blockchain in their storage, and participate in the Bitcoin consensus protocol.

2.2.2.1 Blockchain

The *blockchain* [2, ch. 8] is a growing chain of blocks, sometimes also referred to as the *ledger*. A single block consists of: the hash value of the previous block header in the chain, a timestamp, and the transaction data (see Figure 2.2). As each block references the hash of the previous block, a chain is formed. This chain goes all the way back to the first block, also known as the *genesis* block.

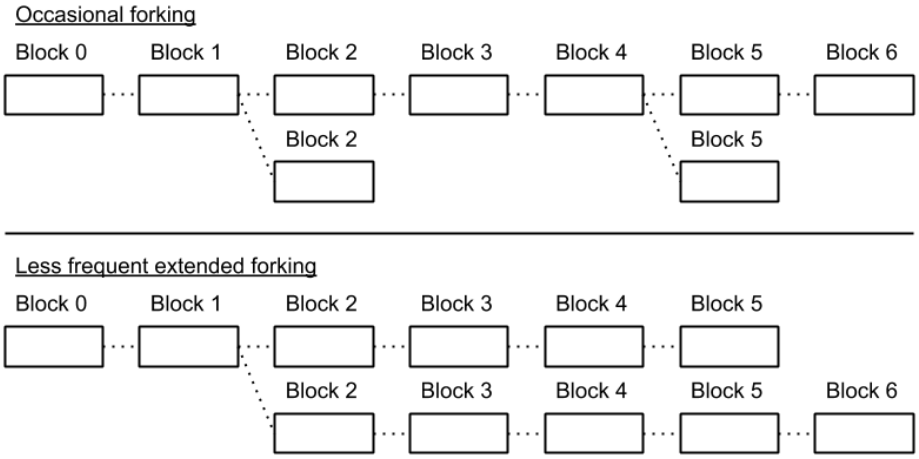

Figure 2.3: Forking on the Bitcoin Blockchain: showing the more frequent occasional forking (top), and less frequent extended forks (bottom) occurring on the blockchain (figure reproduced from [18]).

New blocks are added to the blockchain through a process called *mining* [2, ch. 10]. Mining requires solving a computationally intensive problem, so called *Proof-of-Work* (PoW): brute-force search for a random number (called the nonce) that inserted to the block header produces a hash with sufficient number of leading zeroes. The number of leading zeroes regulate the amount of work needed to participate in the block mining, as it is less likely to choose a random number that produces more leading zeroes. Once a new valid block has been mined, it is broadcast to the other participating nodes in the blockchain network.

Consensus is achieved through [2, ch. 10]: verification of each transaction; mining new blocks (aggregating transactions into blocks); verification of each block; selection of longest chain (main chain) of blocks as main chain (dealing with forks). *Forking* occurs when two mining nodes successfully mine a new block and append it to the blockchain (see example, top block five, in Figure 2.3). This will cause network to consist of two disagreeing partitions. Eventually, one of the forks will outpace the other, and become the main chain (see block six in Figure 2.3).

2.2.2.2 Block

A *block* consists of the block header and a list of transactions (see Figure 2.4) [2, ch. 9]. The block header contains information such as the time at which the block was mined, the nonce, reference to the previous block header, and it contains the Merkle root hash of all transactions. A block is considered *valid*, if all of its transactions are valid, and if the time it was mined is within two hours of the validating nodes clock. A block can be identified and accessed through its block header hash.

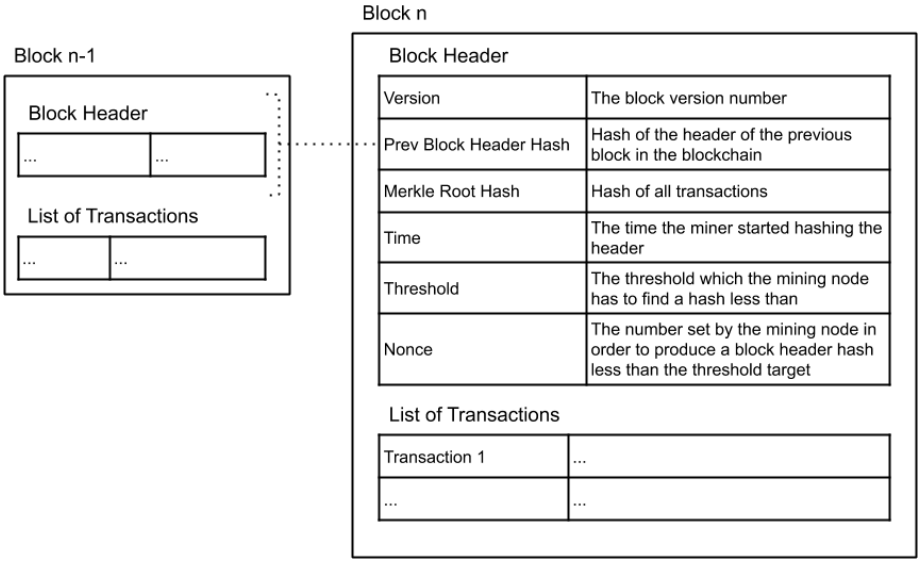

Figure 2.4: The bitcoin block data structure.

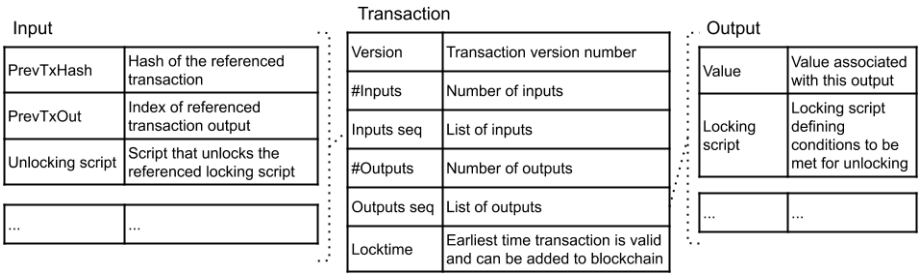

Figure 2.5: The bitcoin transaction data structure.

2.2.2.3 Transaction

A bitcoin *transaction* transfers funds from the inputs to the outputs [2, ch. 6]. A transaction consists of: version number, a list of inputs, a list of outputs, and the locktime (see Figure 2.5). It can be referenced by its *transaction id*. The transaction id, abbreviated as txid, is the hash of the transaction.

An *output* of a transaction consists of a *value* (in Bitcoin currency) and a *locking script*. The locking script defines the conditions on spending the associated value. Each input consists of a reference to a previous output (from a previous transaction), and an unlocking script. The *reference* is the concatenation of the referenced transaction id and the referenced output index of the referenced transaction. The *unlocking script* is used to unlock the inputs locking script. Unspent transaction outputs are referred to as *UTXOs*. These are transactions not yet referenced by any other transaction on the blockchain.

A transaction is *valid* if: the referenced transactions are valid; the unlocking scripts satisfactorily unlock the locking scripts; the referenced transaction outputs have not been spent; and the inputs exceed the outputs in value (the difference thereof is the transaction fee). The unlocking script is verified as follows: the

Name	Description
Version no:	1
Num. inputs:	1
Input1:	• *Previous transaction hash:* `<PREVTXHASH>` • *Previous transaction output index:* 0 • *Input unlocking script length:* sizeof(`<SIGNATURE>`) + sizeof(`<PUBKEY>`) • *Input unlocking script:* `<SIGNATURE> <PUBKEY>`
Num. outputs:	1
Output1:	• *Value:* 1 • *Output script length:* 24 • *Output script:* `OP_DUP OP_HASH160 <PUBKEYHASH> OP_EQUALVERIFY OP_CHECKSIG`
Locktime:	0

Table 2.1: Pay-to-Public-Key-Hash (P2PKH) transaction with 1 input and 1 output.

unlocking script and locking script are concatenated and executed in the bitcoin Script language, if at the end of executing the top of the stack holds the value true, then it unlocked successfully.

Pay-to-Public-Key-Hash (P2PKH) *Pay to public key hash* (P2PKH) [2, ch. 6] is the standard payment method used by Bitcoin wallets issuing a transaction. The payment is issued towards a hash of a public key, transfering bitcoins from an input to an output. Only the private key holder can unlock the output with the corresponding public key hash. The presented P2PKH transaction consists of one input and one output (see Table 2.1), but could include more inputs and outputs.

A referenced P2PKH output is signed for and verified by providing the input script consisting of the signature and the public key. The Script language then concatenates the input with the output script, such that the following script is produced: `<SIG> <PUBKEY> OP_DUP OP_HASH160 <PUBKEYHASH> OP_EQUALVERIFY OP_CHECKSIG`. The Script language reads the input from left to right, and puts any non `OP_` commands on the stack. The part `<PUBKEY> OP_DUP OP_HASH160 <PUBKEYHASH> OP_EQUALVERIFY` duplicates the public key, and checks if it matches the pubkeyhash of the output script, i.e. the address to whom the money is being sent. The `<SIG> <PUBKEY> ... OP_CHECKSIG` part, checks if the signature of the corresponding pubkey of this transaction is correct. If the signature is correct, the input is verified, and the transaction is valid.

OP_RETURN OP_RETURN [2, ch. 12] is an output type specifically for writing data to the blockchain. The OP_RETURN output marks the output as invalid and not part of the UTXOs. The data written to the OP_RETURN output is written to the output script of the output, appended after the OP_RETURN command (see Table 2.2). The maximum allowed data to be written to OP_RETURN is 80 byte according to the Bitcoin standard, and only 1

Name	Description
...	...
Output1:	• *Value:* 0 • *Output script length:* sizeof(`<DATA>`) + 1 • *Output script:* `OP_RETURN <DATA>`

Table 2.2: Transaction with OP_RETURN output.

Parameter	Default value
Mining-difficulty retargeting time:	Every two weeks [2, ch. 10]
Average block-mining time:	Ten minutes [2, ch. 10]
Max block size:	1 MB
Max data per transaction:	100 kB
Max input script size:	1650 B
Max element size on bitcoin script stack:	520 B
Max number of OP_RETURNs per transaction:	1
Max size of OP_RETURN data:	80 B

Table 2.3: Default parameters of the bitcoin blockchain [19].

OP_RETURN output is allowed per transaction. OP_RETURN is a way for bitcoin-based applications to save data and application state to the blockchain.

2.2.2.4 Default Parameters and Properties

The default parameters are shown in Table 2.3. The parameters can be changed for new networks but not for the public Bitcoin network.

2.2.3 Saving Data to the Blockchain

Regular transactions store only the necessary information for transferring funds from one locking script (output) to another. This section reviews how arbitrary data can be stored in transactions on the blockchain.

There are many possible methods for saving data to the blockchain. Out of the presented methods [19], we are only interested in methods that do not generate unspendable UTXOs and are not vulnerable to security and data integrity issues. The only qualifying methods are: OP_RETURN and Data hash (w/ sig).

Out of the two methods, OP_RETURN achieves better data efficiency (see Appendix A), data efficiency is calculated as the ratio of inserted data size to transaction size. The data efficiency of OP_RETURN is 25% at an input size of 80 byte, compared to 16% of Data Hash w/ Sig. Further, if the data size is increased, the OP_RETURN method continues to have better data efficiency, under the assumption that the OP_RETURN method is not limited to 80 byte (this is not true for the public Bitcoin blockchain, but can be set for private

Algorithm 2 Function CreateTransaction

```
uses Blockchain, instance b

function CreateTransaction( privatekey, publickey, prevtxhash, data )
    pubkeyhash = hash( publickey )
    input1 = {
        'txid': prevtxhash,                          \\ referenced transaction id
        'vout': 0 }                                  \\ referenced output number
    output1 = { pubkeyhash: 1, }            \\ bitcoin address and value in btc
    output2 = { "data": data }      \\ "data" keyword and hex-encoded data
    unsignedtransaction = b.CreateRawTransaction[a](
        [ input1 ],
        [ output1, output2 ] )
    signedtransaction = b.SignRawTransactionWithKey[b](
        unsignedtransaction,
        privatekey )
    return signedtransaction
```

[a] createrawtransaction [8]
[b] signrawtransactionwithkey [8]

networks). We limit any further discussions, and select OP_RETURN for saving data to the blockchain.

2.2.3.1 OP_RETURN

The *OP_RETURN* method [19] is achieved by adding data to an output with the following output script to a transaction:

- *Output script:* `OP_RETURN <DATA>`

It is protected against tampering as data is written to the signed outputs. Additionally, the output of OP_RETURN is defined by the bitcoin standard to not be part of the UTXO set, thus avoiding the unspendable UTXO problem.

The max data per transaction is 80 byte (for Bitcoin default, but can be set higher for other networks). The data efficiency is 25% for 80 byte of data. At most one OP_RETURN output can be added to a standard bitcoin output.

2.2.3.2 Function CreateTransaction

The following function *CreateTransaction* (Algorithm 2) details how to create a transaction containing a P2PKH input and output, and a OP_RETURN output. The P2PKH input and output is used for authorization, such that the owner of the private key is the owner of the data written to the OP_RETURN output. The function takes as arguments the private and public key, as well as the referenced transaction output, and the data which is inserted. The algorithm uses the bitcoin core API [8] functions CreateRawTransaction and SignRawTransactionWithKey in order to create a valid transaction.

2.2.4 Other Blockchain Technologies

Bitcoin was the first widely adopted blockchain. Since then (2008) many more blockchain systems have evolved. In this section, we briefly summarise other blockchain technologies and compare them to the Bitcoin blockchain.

- *Bitcoin [20]:* The Bitcoin blockchain (public blockchain), manages the Bitcoin cryptocurrency, uses a UTXO based data model (all UTXOs are saved in a data structure), and its smart contract language is not Turing complete. Bitcoin was the first to propose and use the Proof-of-Work (PoW) consensus protocol. The average block mining time is ten minutes, and average transactions per seconds are seven. The Bitcoin blockchain is considered to have the highest security, based on the amount of mining nodes participating in the PoW consensus.

- *Ethereum [20]:* The Ethereum blockchain manages the Ethereum cryptocurrency (public blockchain). The Ethereum smart contracts provide more functionality than Bitcoin smart contracts, e.g. smart contract internal state, and are Turing complete. Its data model is accounts based, this means that for each account the current state is saved. The Ethereum blockchain also uses PoW consensus, but is working on integrating other consensus algorithms. The average block mining time is 15 seconds, with an average throughput of 15-40 transactions per second.

- *Tendermint [20]:* The main use of the Tendermint blockchain (private blockchain) is as a consensus engine (for byzantine fault tolerant state machine replication), and not as a cryptocurrency. It provides a range of smart contract languages. Its consensus algorithm is a variant of PBFT (practical byzantine fault tolerance), and not PoW. The block mining latency is less than one second, and it can achieve thousand of transactions per second (for private blockchains in single data centers).

Tendermint has best performance in terms of throughput and latency, followed by Ethereum, followed by Bitcoin [20]. In terms of functionality, both Tendermint and Ethereum provide smart contracts, whereas Bitcoin does not. The advantage of using the public Bitcoin blockchain, is its superior security, due to its large adoption and slower block mining rate [4].

Although there are differences between the later blockchain implementations to the original Bitcoin blockchain, we should be able to extrapolate our work to other blockchains as they typically have more functionality than the original Bitcoin blockchain.

2.3 Related Work

We use the term related work broadly, and include a wide spectrum of other work: virtual blockchains [4, 21]; sidechains [22]; blockchain abstraction interfaces [23, 24]; blockchain replicated state machines [23, 24]; blockchain data anchoring (writing data to blockchain) [25, 26, 27]; blockchain-based applications (file system [28], database [29]); and blockchain-based tamper-resistant broadcast protocols (our work).

We have chosen six related works from literature, and present a summary thereof here and in Table 2.4:

	Problem	**Interface**	**Operations**	**Network**	**Blockchain**
[23, 24]	state machine replication	ABCI	DeliverTX; Query; Commit	pubic / private	Tendermint Core
[25]	anchoring data in blockchain	Factom	read; write; search; export	pubic / private	Bitcoin
[26, 27]	anchoring data in blockchain	Chainpoint	submit; export; verify	public	Bitcoin; Ethereum
[4, 21, 30]	blockchain virtualization	Virtualchain	-	public	Bitcoin
[29]	blockchain database	transaction manager	read; write-async; check-tx-status	pubic / private	Ethereum; Hyperledger sawtooth & fabric
[28]	blockchain file system	transaction manager	-	pubic / private	Ethereum
our work	tamper-resistant broadcast	tamper-resistant broadcast	broadcast; deliver; verify	public / private	Bitcoin

	Performance	**Direct Communication**	**Group Membership**	**Data Insertion**	**Safety and Liveness Properties**
[23, 24]	latency: $< 1s$; throughput: $> 10000tps$ (single data center) [20]	no	no	-	-
[25]	-	yes	yes (chainID)	OP_-RETURN	-
[26, 27]	-	yes	no	OP_-RETURN; smart contracts	-
[4, 21, 30]	-	no	no	OP_-RETURN	-
[29]	latency: 100ms to 1s (sequential consistency); throughput: 350tps	no	no	smart contracts	tunable consistency models
[28]	latency: 42.5s (Writing response time)	no	no	smart contracts	-
our work	-	yes / no	yes / no	OP_-RETURN	tunable

Table 2.4: Summary of related work.

- *Tendermint [23, 24]:* Tendermint (tendermint core) is used for byzantine fault tolerant state machine replication. It is a 3 layer system, decoupling the consensus layer (first layer, tendermint core) from the application layer (third layer), by the application blockchain interface (ABCI) layer (second layer).

- *Factom [25]:* Factom is an application interface to anchor, i.e. write, data to the blockchain. Written entries are grouped into blocks, and the Merkle tree root hash of the blocks are written to the blockchain (anchored). This is provided as a service to the application developer. Entries can be grouped into new chainsIDs, and the entire history of a chainID can be retrieved in reverse order.

- *Chainpoint [26, 27]:* Chainpoint, similar to Factom, proposes a scalable protocol for anchoring data to the blockchain. This is achieved by grouping writes together and writing the Merkle tree hash to the blockchain. A blockchain-verifiable receipt is generated and returned to the user that proves the data was written to a transaction. This receipt can be used to prove that the data was written independently of Chainpoint.

- *Virtualchain/Blockstack [4, 21, 30]:* Blockstack was at the time of publication a blockchain based name service, binding human readable names to public keys, using the Virtualchain. The Virtualchain allows for defining arbitrary state machines, or, arbitrary blockchains. The Virtualchain announces the state changes via the underlying blockchain. New logic can be programmed on the Virtualchain layer.

- *Endolith [28]:* Endolith is a blockchain based file system. For all changes of the file system, the blockchain is used to store the: file-hash, user, and timestamp. The system uses a transaction manager, abstracting away the blockchain, to create and submit file tracking smart contracts to the blockchain. The abstraction provides the proof that a file existed at a certain point in time (proof-of-existence), and that a file has not been tampered with (file validation).

- *BlockchainDB [29]:* BlockchainDB is a shared database on the blockchain, that uses the blockchain as a storage layer. The BlockchainDB transaction manager provides tunable consistency guarantees, such as sequential consistency or eventual consistency with bounded staleness.

- *Our work:* The work presented in this thesis: using the (Bitcoin) blockchain for tamper-resistant broadcast protocols.

The common theme to the discussed works in Table 2.4 is the: tamper-proof and tamper-resistant storage of data. Providing a tamper-proof data storage service ([25, 26, 27]), or a virtual blockchain that stores data on an underlying blockchain [4], or a blockchain database or file system that uses the tamper-resistant storage of data and metadata ([29, 28]). In a like manner, our work is using the blockchain for the tamper-resistant storage of messages.

The works in Table 2.4 describe a blockchain abstraction interface, residing somewhere between the blockchain and the application. This is similar to our work, as the tamper-resistant broadcast is layered between the application

layer and the blockchain layer. Such a three layer model for blockchain-based applications has previously been discussed in: [31, 32, 33, 23, 24]. In particular, the Tendermint architecture [23, 24], consists of three layers. Tendermint's application blockchain interface (ABCI) (layer two) provides, among a large set of functions, the broadcast abstraction functionality (the function DeliverTX [23] corresponds to the the broadcast and deliver function). The interface also provides the Query operation, which checks the validity of a transaction, and corresponds to the verify operation of our tamper-resistant broadcast model.

The challenges of blockchain-based applications are [4]: slow-writes; limited bandwidth; the endless-ledger problem; and security. The slow-writes problem is caused by the long confirmation time, such as six blocks on the Bitcoin blockchain, or seven blocks on Ethereum (causing the average write-time to be 42.5 seconds [28]). The limited bandwidth problem is caused by the one MB blocksize limit and the one block per ten minutes average mining time of the Bitcoin blockchain. We will look at solutions to the slow-writes and limited bandwidth challenges. The endless-ledger problem implies the problem which is caused by the growing size of the blockchain. The time needed to start a new node—to download and fully verify the blockchain—takes one to three days [4]. The security challenge is that the security of the application relies on the security of the underlying blockchain, however, the security of the underlying blockchain could get compromised. The Virtualchain addresses this issue, by the ability to migrate to a new blockchain, and by not changing any code of the underlying blockchain [4].

Chapter 3

Blockchain Ledger

In this chapter we abstract the Bitcoin blockchain, by introducing the blockchain ledger abstraction, an append only log (ledger) of records, and analyse its properties. In the next chapter, we use the blockchain ledger abstraction to build the tamper-resistant broadcast abstraction.

3.1 Related Work

The blockchain abstraction is also referred to as a *distributed ledger* [34, 35, 20], or public ledger [36]. A *ledger* is a double-entry record keeping system, in which each record moves funds from one account to another, similar to how (Bitcoin) blockchain transactions move funds from one address to another. The *distributed ledger abstraction*, thus suitable for our purposes, is a an append only log with two functions[35]: get( ), return the current ledger in its entirety; append(record), append a new record to the ledger (c.f. read and write [34]). We adopt this abstraction in this chapter.

We extended this abstraction with the *tamper-resistant* property, as we believe it to be practical and integral of blockchains. This functionality exists in the Bitcoin Blockchain [8]: verify historical transactions (gettxoutproof, verifytxoutproof, getrawtransaction), and verify the entire ledger (verifychain). We adopt the semantics of verifychain: a Verify request returns true if the ledger in its current state is valid and has not been tampered with.

There are disagreements on the *properties* and consistency guarantees of the Bitcoin blockchain [34]. Various statements from literature include, for example, that Bitcoin does not satisfy eventual consistency [35], consensus is never achieved on the blockchain and no transaction is ever final [15], consensus finality is not achieved but rather a probabilistic and economic consensus finality [20], Bitcoin provides eventual consistency [37], and a technical proof that the Bitcoin protocol provides consistency in the asynchronous networking model [36]. The differing opinions, most likely, arise from differing assumptions, preconditions and notation. We will have to reevaluate the properties and normalise the terminology ourselves.

Module 3 Blockchain Ledger

Module name: Blockchain Ledger **instance** bl

Events:

E1 *Request:* < bl, Append | record >: Append a *record* to the blockchain ledger.

E2 *Request:* < bl, Get >: Get the *ledger*.

E3 *Indication:* < bl, GetReturn | ledger >: Return the requested *ledger*.

E4 *Request:* < bl, Verify >: Verify the ledger.

E5 *Indication:* < bl, VerifyReturn | "Valid"/"NotValid" >: Returnvalue for verification request.

Properties:

P-1 *Tamper-resistant:* The state of the ledger, and all entries in the ledger, are protected with high probability against tampering, and the ledger is probailistically verifiable. A verification request returns "Valid" (with high probability), if the blockchain ledger has not been tampered with, else "NotValid".

3.2 Blockchain Ledger Abstraction

The *blockchain ledger abstraction*, or *tamper-resistant distributed ledger abstraction*, is shown in Module 3. It provides five events, one event for requesting to *append* a record to the ledger, and two events for requesting to *get* the current ledger, and the corresponding return event *getreturn* thereof, and two events for requesting to *verify* the current ledger, and the corresponding return event *verifyreturn*. The module provides the *tamper-resistant* property, which is defined in the Module specification. We also considered adding a property stating that the records, and the order of the records, are according to the specified *consensus rules*. We decided to omit such a property, as it is not a critical part of blockchain systems (e.g. Tendermint Core [23] has no consensus rules about the records).

3.3 Bitcoin Transaction Abstraction

The blockchain ledger abstraction discussed records rather than transactions. We need to introduce Bitcoin Blockchain transactions, before we delve into the Bitcoin Blockchain abstraction. We define a Bitcoin Blockchain transaction to contain the following fields:

- *prevtxhash:* A reference to the previous transaction whose output is used as an input. (we omit information about the output index, and assume each transaction has only one spendable output).
- *data:* The data which is saved to the transaction.
- *pubkey:* The associated public key of a transaction output, for which the corresponding private key is needed to unlock the output.

The abstraction models a bitcoin P2PKH transaction which also contains an additional OP_RETURN output (see Section 2.2.3). We omit the value, because we only consider zero-valued transactions without transaction fees (see Section 1.5). A transaction is valid, if the following holds:

- The transaction's signature of the referenced transaction is valid.
- The referenced transaction is valid.
- It is the only (i.e. the first) transaction in the log with the same prevtxhash (that references the same transaction output(s)). That is, a transaction output can only be referenced once.

3.4 Bitcoin Blockchain Ledger Abstraction

The *Bitcoin Blockchain Ledger Abstraction* is an abstraction of the Bitcoin Blockchain [3], and is an instance of the *Blockchain Ledger* abstraction.

The *Bitcoin Blockchain Ledger Module* is shown in Module 4. This module is exemplified and implemented using the bitcoin core API [8] in Algorithm 5. The correctness of the properties are discussed around the implementation.

Correctness *Probabilistic eventual consistency:* The probability of having a network partitioned into two, or more, forks, is inversely exponential to the length of the fork [3]. Thus, the probability that two forks merge over time converges to 1. From this follows, that the probability that all forks eventually merge and are consistent converges to 1.

Probabilistic finality: A transaction, once appended, can only be removed due to forks. The probability of a transaction at depth d being reverted is less than the probability of a fork of length d. From this follows, that the probability of reverting a transaction at depth d, converges to 1 over depth d.

Fair append: The fair append property is correct under the assumptions that the transaction is valid and the transaction submission rate is less than the maximum transaction mining rate (1MB per 10 minutes blockchain standard). A transaction is included to a mined block, if it is valid and if it is among the 1MB highest paying transactions (w.r.t. transaction fees) of the unconfirmed transactions. Because the average transaction rate is less than the mining rate (assumption) the average size of unconfirmed valid transactions is less than 1 MB. Thus the entire set of unconfirmed valid transactions will be mined in its entirety at infinite points in time. Which implies that the transaction is also mined if appended infinite amount of times (fairness assumption).

Transaction validity: The transaction validity follows directly from the Bitcoin consensus protocol and consensus rules, stating that, among other things, valid blocks do not contain invalid transactions.

Tamper-resistant: We would like to show that this protocol implements the tamper-resistant property. For this, we must show that (with high probability): individual transactions are protected against tampering; the history of transactions in the ledger are protected against tampering; and the verification protocol is correct.

First, we show that individual transactions are protected against tampering. This property follows from the valid signature (transaction validity) property of the blockchain.

Module 4 Bitcoin Blockchain Ledger

Module name: Bitcoin Blockchain Ledger, **instance** bl

Events:

E1 *Request:* < bl, Append | transaction >: Append a *transaction* to the blockchain ledger.

E2 *Request:* < bl, Get >: Get the *ledger*.

E3 *Indication:* < bl, GetReturn | ledger >: Return the requested *ledger*.

E4 *Request:* < bl, Verify >: Verify the ledger.

E5 *Indication:* < bl, VerifyReturn | "Valid"/"NotValid" >: Returnvalue for verification request.

Properties:

P-1 *Probabilistic eventual consistency:* With probability converging to 1 over time, every correct process eventually agrees on the same order of transactions in the ledger.

P-2 *Probabilistic finality:* With probability converging to 1 over the depth of a transaction, an appended transaction will not be reverted and removed from the ledger.

P-3 *Fair append:* If a process appends a valid transaction infinitely often, then eventually the transaction is appended to the ledger.

P-4 *Transaction validity:* Only valid transactions are appended to the ledger, this includes:

- *Authorization and authenticity:* the transaction has a valid signature.
- *Transaction order:* If a transaction ta references an output of another transaction ta' as one of its inputs, then ta' is located before ta in the ledger.
- *No double-spend:* A transaction output can be referenced by at most one transaction input.

P-5 *Tamper-resistant:* The state of the ledger, and all entries in the ledger, are protected with high probability against tampering, and the ledger is probailistically verifiable. A verification request returns "Valid" (with high probability), if the blockchain ledger has not been tampered with, else "NotValid".

The history of transactions, i.e. the ledger, is protected against tampering. Due to the probabilistic finality of the blockchain ledger, the probability that a transaction is reverted converges to 0 over the depth of the transaction. Thus, the probability that the transaction history can be tampered with, converges to 0 over the depth of the transaction history. Because the probability is larger than 0, it cannot be considered tamper-proof.

Lastly, we show the correctness of the verification protocol. This follows

Algorithm 5 Blockchain Ledger

Implements: Blockchain Ledger, **instance** bl

Uses: Bitcoin Blockchain (Bitcoin Core API [8]), **instance** b

```
upon event < bl, Init > do
    connect to blockchain b
    ledger ← [ ]

upon event < bl, Append | transaction > do
    b.submitTransaction( transaction ) [a]

upon event < bl, Get > do
    if ledger ≠ b.mainChain [b] do        \\ if not synchronized with mainChain
        ledger = b.mainChain [c]                                  \\ synchronize
    trigger < bl, GetReturn | ledger >

upon event < bl, Verify > do
    if b.verifyChain( )[d] is "Valid" do
        trigger < bl, VerifyReturn | "Valid" >
    else do
        trigger < bl, VerifyReturn | "NotValid" >
```

[a] Bitcoin Core API RPC: SendRawTransaction(transaction)

[b] This can be tested by comparing the hash of the newest blocks, e.g. using GetBestBlockHash to get the hash of the current top block of the blockchain

[c] This assignment can be made more efficient by finding the latest common block of the mainChain and the ledger, removing all orphaned blocks after that block from the ledger, and then appending all successor blocks from the mainChain. The relevant Bitcoin Core RPCs are: GetBestBlockHash; GetBlockHeader; previousblockhash and nextblockhash entries in blockheader.

[d] Bitcoin Core API RPC: VerifyChain(4, 0), thorough check of all blocks.

from the correctness of the blockchain implementation. The verification protocol returns "Valid", if the ledger in its entirety is valid, by checking every block down to the genesis block. This implies that (with high probability) every transaction in the ledger is valid, as well as the current state of the blockchain is valid and has not been tampered with.

Chapter 4

Tamper-Resistant Broadcast

This chapter introduces the tamper-resistant broadcast abstraction. It is divided into sections relating to broadcast environment models: public tamper-resistant broadcast protocols; permissioned tamper-resistant broadcast protocols; permissioned non-native tamper-resistant broadcast protocols.

4.1 Related Work

The operations discussed in the works of Table 2.4 can be generalized and interpreted as the broadcast module events—broadcast, deliver—and, an additional operation to verify the authenticity of data: *verify*.

The *broadcast* operation, writes a message to the blockchain (direction: application to blockchain), either via smart contracts [28, 26, 27, 29], or via OP_RETURN [25, 26, 27, 4]. The write operation can be asynchronous or synchronous, i.e. blocking or non-blocking [23, 24, 29].

The *deliver* operation, delivers messages previously broadcast via the blockchain (direction: blockchain to application). This is the semantic of our work and other work [23, 25]. Other works [29] define a read operator with the semantic of reading a variable.

The *verify* operation lets the application validate if a message was previously written to the blockchain (part of the message history), e.g. Query [23] and verify [26]. A message can be verified by checking if the transaction that stored the message (in complete form or as a hash representation) is stored on the blockchain [26]. Transaction verification is also part of the Bitcoin Blockchain functionality (gettxoutproof, verifytxoutproof) [8].

By studying the previous literature, we can study examples on how certain broadcast properties are realised when using the blockchain as a communications medium, but also for solutions that do not rely on blockchain. A summary of examples of solutions from previous research to the discussed properties can be found in Appendix B.

Module 6 Tamper-Resistant Broadcast

Module name: Tamper-Resistant Broadcast, **instance** tb

Events:

- **E1** *Request:* < tb, Broadcast | message >: Broadcast message to all processes.
- **E2** *Indication:* < tb, Deliver | p, message >: Deliver message broadcast by process p.
- **E3** *Request:* < tb, Verify | p, message >: Verify if a message has been tampered with.
- **E4** *Indication:* < tb, VerifyReturn | "Valid"/"NotValid" >: Returnvalue for verification request.

Properties:

- **P-1** *Tamper-resistant:* The broadcast messages, and the delivered message history are probabilistically (with high probability) protected against tampering, and the delivered messages are probabilistically verifiable. A verification request returns "Valid" (with high probability), if the message has not been tampered with and is part of the untampered message history, else "NotValid".

Algorithm 7 Tamper-Resistant Broadcast

Implements: Tamper-Resistant Broadcast, **instance** tb

Uses: Blockchain Ledger, **instance** bl

```
upon event < tb, Verify | p, txid, message > do
    if message signed by p in bl.Get() ∧ bl.Verify() is "Valid" do
        trigger < tb, VerifyReturn | "Valid" >
    else do
        trigger < tb, VerifyReturn | "NotValid" >
```

4.2 Tamper-Resistant Broadcast Abstraction

The tamper-resistant broadcast abstraction is shown in Module 6. It consists of: a broadcast request event, i.e. send a message to all others; a deliver event, i.e. deliver a message; and a verify and verifyreturn event, probabilistically validate that a message has not been tampered with and is part of the untampered message history.

The implementation is shown in Algorithm 7. We omit showing implementations for the broadcast and delivery events, as these will be defined later. A message is verified if it exists as a historical transaction on the (untampered) blockchain ledger (as common in other discussed protocols [26, 27, 25, 28], the

transaction on the blockchain is used as proof). The implementation uses the Get request and the Verify request of the Blockchain Ledger as a synchronous function call. In a real implementation using the Bitcoin Blockchain, we could directly use the Bitcoin Blockchain native functions for verification of a transaction: gettxoutproof, verifytxoutproof [8].

Correctness *Tamper-resistant:* We would like to show that this protocol, together with the subsequent protocols, implements the tamper-resistant property. As we have not defined the broadcast and delivery events, we must make assumptions about their implementation. These assumptions will hold for all subsequently defined protocols, thus this correctness argument will subsequently hold. We assume that broadcast messages are appended to the blockchain ledger (directly or indirectly as a verifiable hash value), signed for by the broadcasting process. Messages are not delivered before they have been appended to the blockchain ledger.

The correctness follows from the correctness of the tamper-resistant property of the blockchain ledger. The messages and message history are protected against tampering (with high probability), because they are part of the blockchain ledger, and the blockchain ledger is protected against tampering. The verification request checks if the message is in the blockchain ledger, and if the blockchain ledger is valid, if both are true then the message has not been tampered with and is part of the message history.

Given the assumptions, we shall be able to assume the tamper-resistant property of the following broadcast protocols in this chapter.

4.3 Public Tamper-Resistant Broadcast Protocols

We will start with studying public tamper-resistant broadcast protocols. These protocols are: public (open to any participant), unbounded (unlimited participants), BFT (resistant to byzantine faults), and native (messages written to blockchain, consensus via blockchain). The benefit of this configuration, is that it is publicly verifiable and open for any party to participate. This can be useful for certain applications, such as public and open registration systems. We will study a selection of broadcast models, starting with the best-effort tamper-resistant broadcast.

4.3.1 Best-Effort Tamper-Resistant Broadcast

The best-effort tamper-resistant broadcast provides validity, no duplication, and no creation (see Module 8). The implementation is shown in Algorithm 9. The algorithm adds any new broadcast messages to a set of waiting transactions. All transactions that are not part of the blockchain are continually (upon a timeout) re-appended. The algorithm regularly (upon timeout) checks if the ledger has changed (i.e. new transactions). If the ledger has changed, then all transactions of the ledger, that have not previously been delivered (comparison to a set of delivered transactions), are delivered.

Module 8 Best-Effort Tamper-Resistant Broadcast

Module name: Best-Effort Tamper-Resistant Broadcast, **instance** btb

Properties:

P-1 *Tamper-resistant*

P-2-4 *Validity, no duplication, no creation*

Algorithm 9 Best-Effort Tamper-Resistant Broadcast

Implements: Best-Effort Tamper-Resistant Broadcast, **instance** btb

Uses: Blockchain Ledger, **instance** bl

```
upon event < btb, Init | privkey, pubkey, prevtxhash > do
    privkey = privkey
    pubkey = pubkey
    prevtxhash = prevtxhash
    prevledger = [ ]
    waiting = ∅
    delivered = ∅
    starttimer( Δ )

upon event < btb, Broadcast | message > do
    transaction = CreateTransaction( privkey, pubkey, prevtxhash, message )
    waiting = waiting ∪ transaction
    prevtxhash = hash( transaction )

upon event < Timeout > do
    for transaction ∈ waiting ∧ transaction ∉ prevledger do
        trigger < bl, Append | transaction >
    trigger < bl, Get >

upon event < bl, GetReturn | ledger > do
    for transaction ∈ ledger \ prevledger ∧ transaction ∉ delivered do
        trigger < bb, Deliver | transaction.pubkey, transaction.message >
        delivered = delivered ∪ transaction
    prevledger = ledger
    starttimer( Δ )
```

Correctness *Validity:* The validity property is inherited from the fair append property. If a correct process broadcasts a message m, then m is added to the waiting set of transactions. This set of transactions is repeatedly appended to the blockchain ledger, if the transaction is not yet on the ledger (see e.g. [1, ch. 2.4] or [28]). Thus, eventually it will be appended to the blockchain, and delivered by the other processes.

No duplication: This is achieved by saving all delivered messages in a set,

and comparing any new messages with the set of already delivered messages (see e.g. [1, ch. 2.4]).

No creation: Assuming that every blockchain broadcast instance is instantiated with a unique public key (assumption). And assuming that every delivered message is valid. Then this implies that every message also must have been signed for by the private key holder. Thus the authenticity of the message is provided (see e.g. [4]).

4.3.2 Reliable Tamper-Resistant Broadcast

The reliable tamper-resistant broadcast (Module 10) provides in addition the agreement property. The agreement property states, if a message m is delivered by some correct process, then m is eventually delivered by every correct process.

The reliable tamper-resistant broadcast protocol is shown in Algorithm 11. The protocol achieves agreement by: if a process delivers a message, then this message is added to a set of delivered messages; if, due to a fork, a previously delivered message is no longer part of the blockchain, then this message (transaction) is continually re-appended to the blockchain until it is again part of the ledger.

We must, however, consider the following scenario. Consider an adversarial process that double-spends a UTXO on two different forks, such that two different messages are delivered by the processes on the different forks. In this case, we must be able to re-append (one or both of them) to the blockchain, in order to achieve agreement. We cannot add both transactions to the blockchain (using same input UTXO). Instead, we could write the lost transaction as part of the data of a new transaction (as is done in Algorithm 11).

Correctness *Tamper-resistant, validity, no duplication, no creation:* This follows from the same argument as for best-effort broadcast.

Agreement: The agreement property is achieved by each process keeping track of its delivered messages, in such a way that if a delivered message/transaction is removed from the blockchain due to forking, then the process adds this transaction to its own waiting set, even if the process did not originally broadcast the message. This ensures that any correct process that delivers a message repeatedly ensures that the message is part of the blockchain and thus also eventually delivered by every other correct process (see e.g. [1, ch. 3.3]).

4.3.3 FIFO-Order Tamper-Resistant Broadcast

The FIFO-order (first-in first-out) property specifies the order of delivery of messages. It guarantees that the order of messages broadcast from a sender s is the same as the order of delivered messages from s. The FIFO-order (reliable) tamper-resistant broadcast is shown in Module 12.

The idea behind the FIFO-order algorithm (Algorithm 13), is that each broadcast transaction references the previously broadcast transaction in the prevtxhash field. This produces a linked chain of transactions, and the blockchain ledger must adhere to this order (transaction validity property). From this follows that, if transactions are delivered in the order of the blockchain, then this is also the order of broadcast messages. The reliable tamper-resistant broadcast

Module 10 Reliable Tamper-Resistant Broadcast

Module name: Reliable Tamper-Resistant Broadcast, **instance** rtb

Properties:

P-1-4 *Tamper-resistant, validity, no duplication, no creation*

P-5 *Agreement*

Algorithm 11 Reliable Tamper-Resistant Broadcast

Implements: Reliable Tamper-Resistant Broadcast, **instance** rtb

Uses: Blockchain Ledger, **instance** bl

upon event $<$ rtb, Init | privkey, pubkey, prevtxhash $>$ **do**
    same as Algorithm 9

upon event $<$ rtb, Broadcast | message $>$ **do**
    same as Algorithm 9

upon event $<$ Timeout $>$ **do**
    same as Algorithm 9

upon event $<$ bl, GetReturn | ledger $>$ **do**
    for transaction $\in$ ledger $\setminus$ prevledger $\wedge$ transaction[a] $\notin$ delivered **do**
        trigger $<$ rtb, Deliver | transaction.pubkey, transaction.message $>$
        delivered = delivered $\cup$ transaction
    for transaction $\in$ delivered $\setminus$ ledger **do**
        if transaction not valid[b] **do**
            trigger $<$ rtb, Broadcast | transaction $>$[c]
        waiting = waiting $\cup$ transaction
    prevledger = ledger
    starttimer(Δ)

[a] If the transaction contains a transaction as data, then check if is valid and if it has already been delivered, if valid then deliver the transaction.data.pubkey, transaction.data.message.
[b] Adversarial process has double-spent the UTXO input of this transaction.
[c] Broadcast the transaction as data payload instead.

algorithm (Algorithm 11) did not specify the order of delivery, whereas this is specified for the FIFO-order algorithm.

Correctness *Tamper-resistant, validity, no duplication, no creation, agreement:* Follows from reliable broadcast.

FIFO-order: The correctness of the FIFO-order property follows from the blockchain ledger transaction validity property, which encompasses the transaction order property: if a transaction ta references an output of another transaction ta' as one of its inputs, then ta' is located before ta in the ledger. As

Module 12 FIFO-Order Tamper-Resistant Broadcast

Module name: FIFO-Order Tamper-Resistant Broadcast, **instance** ftb

Properties:

P-1-5 *Tamper-resistant, validity, no duplication, no creation, agreement*

P-6 *FIFO-order*[a]

[a]We consider messages broadcast by an adversarial process to be concurrent, and not in any order. For example, if an adversarial process double-spends a UTXO, and appends two different messages on two different forks, then these are two concurrent broadcasts and not strictly in FIFO-order.

Algorithm 13 FIFO-Order Tamper-Resistant Broadcast

Implements: FIFO-Order Tamper-Resistant Broadcast, **instance** ftb

Uses: Blockchain Ledger, **instance** bl

```
upon event < ftb, Init | privkey, pubkey, prevtxid > do
    same as Algorithm 9

upon event < ftb, Broadcast | message > do
    same as Algorithm 9

upon event < Timeout > do
    same as Algorithm 9

upon event < bl, GetReturn | ledger > do
    for transaction ∈ ledger \ prevledger                \\ in sequence of ledger
    ∧ transaction^a ∉ delivered do
        trigger < rtb, Deliver | transaction.pubkey, transaction.message >
        delivered = delivered ∪ transaction
    for transaction ∈ delivered \ ledger do
        if transaction not valid^b do
            trigger < rtb, Broadcast | transaction >^c
        waiting = waiting ∪ transaction
    prevledger = ledger
    starttimer( Δ )
```

[a]If the transaction contains a transaction as data, then check if is valid and if it has already been delivered, if valid then deliver the transaction.data.pubkey, transaction.data.message.

[b]Adversarial process has double-spent the UTXO input of this transaction

[c]Broadcast the transaction as data payload instead.

the submitted transactions build a chain of submitted transactions by input references, this FIFO-order also has to be reflected on the blockchain ledger. Because previously undelivered messages are delivered in the order of the blockchain, the order of the delivered messages are in FIFO-order. The FIFO-order is not required of transactions by an adversarial process.

Module 14 Causal-Order Tamper-Resistant Broadcast

Module name: Causal-Order Tamper-Resistant Broadcast, **instance** ctb

Properties:

P-1-6 *Tamper-resistant, validity, no duplication, no creation, agreement, FIFO-order*

P-7 *Causal-order*

Algorithm 15 Causal-Order Tamper-Resistant Broadcast

Implements: Causal-Order Tamper-Resistant Broadcast, **instance** ctb

Uses: FIFO-order Tamper-Resistant Broadcast, **instance** ftb

```
upon event < ctb, Init | privkey, pubkey, prevtxhash > do
    ftb.Init( privkey, pubkey, prevtxhash )
    DAG = ∅
    delivered = ∅
    root = ⊥
    add root to DAG

upon event < ctb, Broadcast | message > do
    leafs = set of reachable leafs from root in DAG
    msg = pack( leafs, message )
    for tx in leafs do
        add edge (tx, msg.txid) to DAG
    trigger < ftb, Broadcast | msg >

upon event < ftb, Deliver | pid, message > do
    leafs, msg = unpack( message )
    tx1, tx2, ..., txn = leafs
    add node (pid, tx, message) to DAG
    add edges (tx1, tx), ..., (txn, tx) to DAG
    for all reachable[a] nodes that are not in delivered do
        trigger < ctb, Deliver | node.pid, node.txid, node.msg >
        delivered = delivered ∪ node
```

[a] We define a node as reachable, if all of its predecessors are in the DAG and reachable.

4.3.4 Causal-Order Tamper-Resistant Broadcast

The causal-order property specifies that messages are delivered such that the order respects the happened-before relationship: for any message $m1$ that potentially caused a message $m2$, i.e., $m1 \rightarrow m2$, no process delivers $m2$ unless it has already delivered $m1$. The module is shown in Module 14.

The causal-order can be achieved by: using vector clocks [1, ch. 3.9], or appending the causal message history to the message [1, ch. 3.9]. Because the

number of participants is unbounded, we cannot use the vector-clock methodology. The implementation which we present, is a derivation of the message history method.

For every message, we must uniquely (and immutably) assign its causal past. The protocol (Algorithm 15), broadcasts together with the message, a set of transaction ids that were the causal predecessors of the transaction. From these references, we can build a directed acyclic graph (DAG). From the DAG, the causal past of a transaction can be inferred. Thus, we wait to deliver a transaction, until its causal past has been delivered.

Correctness *Tamper-resistant, validity, no duplication, no creation, agreement, FIFO-order* These properties follow directly from the use of the FIFO-order (reliable) broadcast. The FIFO-order follows from the causal-order.

Causal-order: Messages are delivered in the order of the DAG. And, only messages for which all predecessors are present in the DAG are delivered. From this follows the causal-delivery, if the DAG is causally ordered. The DAG respects the causal-order by construction, as processes which broadcast messages assign their causal predecessors correctly. This is achieved by the algorithm choosing all leafs of the DAG (leafs defined as the set of reachable nodes that have no reachable successor node; we define a node as reachable, if all of its predecessors (references) are in the DAG and reachable), and append these leafs to the message.

4.3.5 Uniform Causal-Order Total-Order Tamper-Resistant Broadcast

As noted in the background chapter, we cannot design a public uniform total-ordered broadcast protocol, as it would imply a solution to the (impossible) public uniform consensus problem, for an unbounded number of participants. The standard practice is to assume a bounded fork length: Bitcoin six blocks [38]; Bitcoin ten blocks [30]; Ethereum seven blocks [28]. Thus, any transaction that is at least six blocks deep on the Bitcoin blockchain can be considered *confirmed*, or, *final*, i.e. it will not be re-ordered or removed from the ledger.

The blockchain uniform causal-order total-order tamper-resistant broadcast is shown in Module 16, the implementation is shown in in Algorithm 17. The uniform agreement and total-order follows from only delivering transactions that are final, i.e. confirmed (according to assumptions). From the total-order and FIFO-order, follows the causal-ordered delivery of messages. The difference of this protocol to previous protocols, is that we wait until a transaction is six blocks deep (confirmed, finalized) before delivering it to the application.

Correctness *Tamper-resistant, validity, no duplication, no creation, FIFO-order:* Same as for FIFO-order broadcast.

Causal-order: FIFO-order + total-order implies causal-order. Let us assume that the order is both FIFO-order and total-order. We must show that the order respects the causal-order according to the happened-before relationship (see Section 2.1, denoted by $\rightarrow$). The happened-before relationship is described by the disjunction of 3 rules, we will show how each rule is satisfied by our assumptions. (1) If some process broadcast m_1 before m_2, then $m_1 \rightarrow m_2$. Because of the

Module 16 Uniform Causal-Order Total-Order Tamper-Resistant Broadcast

Module name: Uniform Causal-Order Total-Order Tamper-Resistant Broadcast, **instance** ucttb

Events:

- **E1** *Request:* < ucttb, Broadcast | message >: Broadcast message to all processes.
- **E2** *Indication:* < ucttb, Deliver | p, message >: Deliver message broadcast by process p.

Properties:

- **P-1-4** *Tamper-resistant, validity, no duplication, no creation*
- **P-5** *Uniform agreement*
- **P-6** *Uniform total-order*
- **P-7** *Uniform causal-order*

Algorithm 17 Uniform Causal-Order Total-Order Tamper-Resistant Broadcast

Implements: Uniform Causal-Order Total-Order Tamper-Resistant Broadcast, **instance** ucttb

Uses: Blockchain Ledger, **instance** bl

upon event < ucttb, Init | privkey, pubkey, prevtxhash > **do**
    same as Algorithm 9

upon event < ucttb, Broadcast | message > **do**
    same as Algorithm 9

upon event < Timeout > **do**
    same as Algorithm 9

upon event < bl, GetReturn | ledger > **do**
    for transaction ∈ ledger \ delivered \\ in sequence of ledger
    ∧ transaction is confirmed [a] **do**
        trigger < ucttb, Deliver | transaction.pubkey, transaction.message>
        delivered = delivered ∪ transaction
    prevledger = ledger
    starttimer(Δ)

[a] Bitcoin transactions are commonly considered finalized once they are six blocks deep [38].

FIFO-order property, m_1 will always be before m_2 in the total-order. (2) If some process delivered m_1 before broadcasting m_2, then $m_1 \rightarrow m_2$. Because of the total-order property, if m_1 is delivered, then the prefix of messages in the total-order up to m_1 is fixed. Thus, no message m_2 broadcast after the delivery of m_1 can be inserted before m_1 in the total-order. Thus, m_1 must be before m_2 in the total-order. (3) If there exists a message m' such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$, then $m_1 \rightarrow m_2$. The transitivity rule follows from the transitivity of the total-order property. If m_1 is before m' and m' is before m_2 in the total-order, then m_1 is before m_2 in the total-order.

The reverse direction (causal-order + total-order implies FIFO-order + total-order) is trivial to show, thus FIFO-order + total-order is equivalent to causal-order + total-order. Similarly, as shown in Hadzilacos and Toueg [9, ch. 3.3], causal-order is equivalent to FIFO-order + local-order (local-order: if a process delivers a message m_1 before broadcasting a message m_2, then no other process delivers m_2 before m_1). As an alternative argument, we could have shown that total-order implies the local-order, thus total-order and FIFO-order implies the local-order and FIFO-order, which implies the causal-order.

Uniform agreement: Transactions are not delivered until they have been confirmed on to the blockchain. This ensures that if a transaction is delivered by a process, then it has been confirmed, which implies that the transaction is immutably stored on the blockchain and eventually is delivered by all correct processes, even if the process fails shortly after delivering (see e.g. [30, 28]).

Uniform total-order: The uniform total-order property is ensured, because each process delivers the messages of the confirmed transactions in the same order as the ledger. Because of the bounded fork length assumption, the confirmed transactions order on the ledger is final. Thus, the order in which the messages are delivered is the same for any two processes (see e.g. [30, 28]).

4.3.6 Implications

This section has shown the possibility for achieving causal-ordered reliable broadcast, for the public, unbounded, BFT model. We noted the impossibility of uniform reliable broadcast (and stronger models), for the public, unbounded, BFT model. Many current implementations rely on the assumption that the fork-length is bounded for realising a uniform total-ordered tamper-resistant broadcast. In the next section, we will use the fact that we have a known and bounded set of non-byzantine participants, and can therefore design stronger models (without assuming bounded fork length).

4.4 Permissioned Tamper-Resistant Broadcast Protocols

There are two reasons why we are interested in permissioned broadcast protocols. The first reason, is that group communication and group membership abstractions can be very useful for the application developer. In particular, through disjoint permissioned group communication channels, we can limit the group to known participants and non-byzantine participants. Second, by introducing group communication channels, we can also limit the number of participants to a known set and bounded number. Thus, we can use protocols that rely on

quorums (e.g. majority voting), and we can also assume the non-byzantine behaviour of the participants.

4.4.1 Fixed Group Tamper-Resistant Broadcast

The *fixed group tamper-resistant broadcast* protocol, has a fixed and finite set of members (participants). The set of members are decided before the initialisation of the protocol. The messages are publicly broadcast via the blockchain, this provides the tamper-resistant property.

The fixed membership tamper-resistant broadcast module is shown in Module 18. The group membership property, implies, that no message is delivered unless it was broadcast by a process from the group (c.f. view inclusion of view-synchronous communication for a static view [1, ch. 6.8]).

The implementation in Algorithm 19 identifies the group members through their public keys. The protocol uses an underlying instance of a tamper-resistant broadcast protocol. Messages that are delivered by the underlying protocol are filtered, such that only messages from known group members are delivered to the application. Because of this, the properties of the underlying broadcast protocol are wholly inherited, whilst also providing the group membership property.

Correctness *Group membership:* The correctness of the algorithm relies on the assumption that, if the underlying broadcast protocol delivers a message m with sender p, then this message was previously broadcast by p. This follows from the no creation property of the tamper-resistant broadcast protocols discussed in previous section. Thus, by filtering for messages that are from group members, we only deliver messages that are from group members.

4.4.2 Implications

Permissioned tamper-resistant broadcast allows us to design protocols and solve problems that are not possible when using the public tamper-resistant broadcast abstraction. This is because we can assume a known and bounded set of group members; and assume the non-byzantine failure model of the group members. The security of the application is still backed by the tamper-resistant (possibly public) blockchain. Thus the message history is protected from byzantine faults.

In this short section, we left out many other aspects, notably: dynamic group membership and view-synchronous communication [1, ch. 6.8].

We could build many more protocols on the fixed membership model. Here are some that we thought are worth a brief mention:

Byzantine, and non-byzantine, fault-tolerant consensus: Byzantine fault-tolerant consensus can be achieved through a best-effort broadcast (ByzantineEpochConsensus [1, ch. 5.6]). The same is true for non-byzantine fault-tolerant consensus. We find that fixed group tamper-resistant consensus is an interesting possibility, because the protocol is transparent and publicly inspectable. However, because of the high message complexity of consensus, we think that in the majority of cases, using a non-native protocol for consensus (if possible) is a better alternative, as we explore in the next section.

Uniform total-order (reliable) broadcast: Byzantine, and non-byzantine uniform, total-order broadcast can be achieved by using reliable broadcast and consensus [1, ch. 6.1, ch. 6.2].

Module 18 Fixed Group Tamper-Resistant Broadcast

Module name: Fixed Group Tamper-Resistant Broadcast, **instance** fgtb

Events:

E1 *Request:* < fgtb, Broadcast | message >: Broadcast message to all processes.

E2 *Indication:* < fgtb, Deliver | p, message >: Deliver message broadcast by process p.

Properties:

P-1 *Group membership:* If a process delivers a message m broadcast by process p, then p is a member of the fixed group.

Algorithm 19 Fixed Group Tamper-Resistant Broadcast

Implements: Fixed Group Tamper-Resistant Broadcast, **instance** fgtb

Uses: Tamper-Resistant Broadcast, **instance** tb

upon event < fgtb, Init | privkey, pubkey, prevtxhash, groupmembers > **do**
    groupmembers = groupmembers
    tb.Init(privkey, pubkey, prevtxhash)

upon event < fgtb, Broadcast | message > **do**
    trigger < tb, Broadcast | message[a] >

upon event < tb, Deliver | p, message > **do**
    if p ∈ groupmembers **do**
        trigger < fgtb, Deliver | p, message >

[a] It is also possible to encrypt the message, such that only the group members could decrypt the message. For this, a (1,n) threshold encryption scheme could be used, such that any (1) of the recipients (n) can decrypt the message [17, ch. 11.5]. For this, the set public-keys are needed for encryption, and one private key for decryption

Uniform causal-order (reliable) broadcast: Using the causal-order tamper-resistant broadcast implementation in previous section, we could wait to deliver a message until it has at least $> n/2$ many leafs (majority of processes have acknowledged the transaction).

Uniform causal-order total-order broadcast: By combining the total-order and FIFO-order, we achieve causal-order. For example, we could use a FIFO-broadcast protocol to broadcast the messages. Using consensus, we could decide on which order to broadcast the messages (whilst keeping the FIFO-order).

Performance gains of permissioned tamper-resistant broadcast protocols versus public tamper-resistant broadcast protocols: An interesting aspect are potential performance gains possible for permissioned protocols compared to public

protocols. The discussed fixed group tamper-resistant broadcast protocol in Algorithm 19, reduces itself to using a public tamper-resistant broadcast protocol for the same non-group-membership problem. This implies that no performance gains are achieved when comparing the public and permissioned counterparts. Performance gains, however, are possible as we reduce the problem instance space.

To our knowledge, there are no significant performance gains possible from optimising the discussed best-effort, reliable, FIFO-order, and causal-order broadcast protocols for the permissioned environment model. In the next section we discuss how performance gains can be achieved by decoupling the performance from the underlying blockchain using non-native permissioned tamper-resistant broadcast protocols.

4.5 Permissioned Non-Native Tamper-Resistant Broadcast Protocols

In this section, we look at permissioned protocols (bounded), assuming the fail-crash failure model (non-BFT), that may use direct links or P2P for exchanging messages (non-native). The difference to previous sections, is that we can use external means for communication, and not depend on the blockchain for exchanging messages. In effect, we are not limited to blockchains throughput and latency for our protocol. This is one of the benefits of the non-native environment, and we will look at commonly employed techniques for achieving: high throughput, low latency, and arbitrary message sizes.

4.5.1 High Throughput Low Latency Non-Native Uniform Total-Order Tamper-Resistant Broadcast

The broadcast module is shown in Module 20, it defines the high throughput and low latency properties as comparable to other high-performance systems. High throughput can be achieved by grouping writes together and writing the Merkle root hash thereof on the blockchain [25, 26, 27]. Low latency can be achieved by buffering broadcast messages as an asynchronous write [28, 25, 26]. Arbitrary message size is enabled through a constant sized hash.

The implementation is shown in Algorithm 21. It uses a high-throughput and low-latency total-ordered broadcast (htlltb) primitive for the ordering and dissemination of messages to all processes. By construction, the protocol inherits the properties of the total ordered broadcast (htlltb), any total-ordered broadcast model could be used for this. The tamper-poof property arises from anchoring (writing) a group of messages to the blockchain through the Merkle-root hash. This is regularly triggered by a timeout.

Correctness *High throughput and low latency:* The throughput and latency are only limited by the used uniform total-order broadcast, as well as the local operations *log.append* and *signature is valid*. Assuming that the local operations are implemented efficiently, then the latency and throughput should reflect that of the used high-performance uniform total-ordered broadcast, and achieve high throughput and low latency.

Module 20 High Throughput Low Latency Tamper-Resistant Broadcast

Module name: High Throughput Low Latency Tamper-Resistant Broadcast, **instance** htlltb

Events:

E1 *Request:* < htlltb, Broadcast | message >: Broadcast message to all processes.

E2 *Indication:* < htlltb, Deliver | p, message >: Deliver message broadcast by process p.

Properties:

P-1 *High throughput:* The throughput is comparable (within 50%) to other high-performance specialized systems for this task.

P-2 *Low latency:* The latency is comparable (within 50%) to other high-performance specialized systems for this task.

Other broadcast properties The protocol inherits the broadcast properties of the underlying uniform total-order broadcast protocol by construction. This is because, if the application broadcasts a message, then the message is broadcast via hltob, and, if hltob delivers a message, then our algorithm also delivers the message.

Tamper-resistant: In order to prove the tamper-resistant condition,we must show that the broadcast messages are protected against tampering, and that the message history is protected against tampering. The broadcast messages are protected against tampering because they are signed by the private key holder, this signature is appended to the message. The message history is protected by the tamper-resistant broadcast. The Merkle root of the entire message history is regularly broadcast via the tamper-resistant broadcast. Thus, the Merkle root of the history of messages (the log) is tamper-resistant, which implies that the history of messages is tamper-resistant.

4.5.2 Implications

The presented implementation inherits its properties of the used subcomponents. For example, if we required a non-BFT causal-ordered total-ordered tamper-resistant broadcast, we can use the zookeeper atomic broadcast [14], which has the properties of a uniform causal-ordered total-ordered (reliable) broadcast. This composition would tolerate up to $< n/2$ faulty (non-byzantine) failures.

With the non-native permissioned tamper-resistant broadcast environment model, high-performance tamper-resistant broadcast models can be designed matching the desired properties.

Algorithm 21 High Throughput Low Latency Tamper-Resistant Broadcast

Implements:
High Throughput Low Latency Tamper-Resistant Broadcast, **instance** htlltb

Uses:
Fixed Group Reliable Tamper-Resistant Broadcast, **instance** fgrtb
High Throughput Low Latency Uniform Total-Ordered Broadcast, **instance** hltob

upon event < htlltb, Init | privkey, pubkey, prevtxhash, groupmembers > **do**
fgrtb.Init(privkey, pubkey, prevtxhash, groupmembers)
map = $\emptyset$
log = [ ]
timeout.start(Δ)

upon event < htlltb, Broadcast | message > **do**
trigger < hltob, Broadcast | message, sign(message, privkey) >

upon event < hltob, Deliver | p, message, signature > **do**
if signature is valid **do**
trigger < htlltb, Deliver | p, length(log), message >
log.append(p, message, signature)

upon event Timeout **do**
mth = MerkleTreeHash(log)
trigger < fgrtb, Broadcast | length(log) $\rightarrow$ mth >
timeout.start(Δ)

upon event < fgtb, Deliver | p, txid, length $\rightarrow$ mth > **do**
map.add(length $\rightarrow$ (txid, mth))

upon event < htlltb, Verify | p, lsn, message, signature > **do**
if p, message, signature in log at position lsn **do**
length = min$\{length \in map : length \geq lsn\}$
txid, mth = map.get(length)
trigger < hltb, VerifyReturn | fgrtb.Verify(txid, mth) >
else
trigger < hltb, VerifyReturn | "NotValid" >

Chapter 5

Evaluation

We implemented and evaluated the performance of a selection of the protocols, and experimentally tested the correctness of the protocols. The implementation of the protocols, together with all code used for running the discussed experiments can be accessed online [39].

Broadcast protocols are commonly evaluated on throughput and latency [40, 41]. This is also true of the related work in Section 2.3 [28, 29], and of Blockchain systems [20]. We define the latency as time between broadcast and delivery at the sending process, as similar to broadcast-related work [40]. In contrast, blockchain systems measure latency typically as the average block mining time [20, 42]. The throughput is described as messages per second [41], or MB per second [41, 40], or measured as transactions per seconds (tx/s) [20, 29, 42]. We use the definition of transactions per second (tx/s).

Besides the performance metrics, other interesting aspects to consider are: number of processes for scalability [40, 41]; message size; packet loss; fairness (percentage of completed broadcasts); overhead of the various subsystems as a dimension [28], various consistency models [29]. We cannot investigate every aspect, but we do investigate varying the number of processes, varying the consistency models, and test for the subsystem overhead of one of the protocols.

5.1 Methodology

We implemented four protocols and deployed them on a distributed cluster, spanning over three physically separated locations in the Frankfurt area[1].

The benchmark workload consisted of continually alternating between broadcasting a message and delivering messages for a time of 180 seconds (3 minutes). This was executed in parallel on three, six, nine, and twelve processes. We set up a fresh execution environment for each benchmark run, using Google Kubernetes Engine. This comprised of a six-node ETCD cluster, a six-node MultiChain cluster, and a twelve-node Broadcast cluster. The nodes were evenly deployed over three physically separated locations in the Frankfurt area. From these workloads, we collected the history of each process executing the benchmark. A

[1] Refer to [39] and in particular the evaluation execution script evaluation/deployment/run_-all.sh, and the benchmark executor tamperproofbroadcast/tests/benchmarks/benchmark.py for exact details [39].

history consists of a sequence of: operation (broadcast or deliver); process id; message; time. From this history, the throughput and latency of each individual process was calculated. Further statistics, such as average throughput and average latency, and the 5th and 95th percentile were calculated from this. The histories were also used to check the correctness of each execution, with respect to the safety properties of the broadcast protocols.

5.1.1 Implementation

We implemented three of the discussed protocols from Chapter 4, and a non-tamper-resistant broadcast protocol for comparison. Here follows a short description of each implementation and the identifier which refers to it:

- *fotb (identifier used in this chapter):* FIFO-Order Tamper-Resistant Broadcast (Algorithm 13)
- *totb:* Uniform Causal-Order Total-Order Tamper-Resistant Broadcast (Algorithm 17)
- *htlltb:* High Throughput Low Latency Tamper-Resistant Broadcast (Algorithm 21)
- *htlltbtest:* Not described in previous chapters. This is an implementation of a non-tamper-resistant uniform causal-ordered total-ordered broadcast. The purpose of it is to allow the comparison of our proposed algorithms to other state-of-the art algorithms. For this, we use the implementation of *htlltb*, and strip away all blockchain and tamper-resistant related components. What we are left with, is a broadcast abstraction of the used high throughput low latency broadcast software.

We use a Bitcoin Core [8] compatible blockchain, MultiChain version 2.0.2 [43], as the blockchain ledger, for fotb, totb and htlltb. We use ETCD version 3.4.5 [44], a raft-based consensus program (linearizable replicated key-value storage), for the high throughput and low latency causal-order total-order broadcast of htlltb and htlltbtest.

The implementations were translated from pseudocode into the programming language Python. We tried to use the Python built-in data types, such as sets and lists, where possible.

We use batching to increase the throughput, as otherwise the performance suffers due to RPC round trip times. For fotb and totb, broadcast messages during the latest 0.1 seconds (or if cumulatively larger than 2048 byte), are batched together before written to the blockchain. For htlltb and htlltbtest, broadcast messages are batched together in groups of 128 messages, before being broadcast via the underlying broadcast system.

5.1.2 Software

We used the following software and settings:

- *Broadcast protocols:* The implemented tamper-resistant broadcast protocols are available online [39].

- *MultiChain:* We used the MultiChain blockchain serving as the Bitcoin Core compatible backend.
  - *Consensus:* Proof-of-work, 10 seconds per block, max 8MB per block, no transaction costs.
  - *Version:* 2.0.2 [45]
- *ETCD:* Version: 3.4.5, bitnami/etcd [44]

5.1.3 Deployment

We deployed the benchmark on Google Kubernetes Engine with the following settings. The code for the deployment is available online [39].

- *Region:* europe-west-3 (Frankfurt, Germany)
- *Zones:* a, b, c (physically separated locations)
- *Machine type (n1-standard-2):* 2 vCPU, 7.5 GB memory, 100 GB disk size, 10 Gbps network egress bandwidth (24 machines in total, 1 node per machine) [2]
- *Nodes:* 6 MultiChain nodes, 2 per zone; 6 ETCD nodes, 2 per zone; 12 broadcast nodes, 4 per zone.

5.1.4 Benchmark

The benchmark consisted of the following workload:

- *Workload:* 180 seconds (3 minutes) of alternating between: broadcasting 1 message; delivering up to 1024 messages or until local queue is exhausted.
- *Message size:* 256 byte random data.

We repeated the benchmark for the following configurations:

- *Number of processes:* 3, 6, 9, and 12.
- *Protocols:* fotb, totb, htlltb, and htlltbtest.

5.1.5 Statistics

We collected the statistics of the benchmark runs. This is how we caluclated the throughput and latency:

- *Throughput:* Total delivered messages per second (tx/s). This is calculated as the average (over all histories) number of delivered messages per second.
- *Latency:* Time between broadcasting and delivering a message, from the perspective of the broadcasting process. This is calculated as the average (over all histories) of all messages that were broadcast and delivered.

[2] A vCPU is a single hardware Hyper-thread on one of the following Intel Xeon CPU platforms: Skylake, Broadwell, Haswell, Sandy Bridge, or Ivy Bridge [46].

- *Broadcast Throughput:* Total broadcast (not delivered) messages per second (tx/s). This is calculated as the average (over all histories) number of broadcast messages per second.

- *5th and 95th percentile:* The 5th percentile is the value which is greater than 5% of the observed values and less than 95%; the 95th percentile is the value which is greater than 95% of the observed values.

5.2 Results

We executed the benchmark for each protocol, with three, six, nine, and twelve nodes. From the histories, we analysed the throughput and latency, and checked the histories for correctness.

5.2.1 Correctness

We collected the histories from each workload. A history, is a sequence of broadcast and delivered messages from the view of a process. We evaluated these histories [39], and could not find any inconsistencies to the proposed safety properties: FIFO-order; total-order; causal-order; no-duplication; no-creation. We did, however, confirm that fotb does not provide the total-order property, as fotb-6 and fotb-9 did not deliver in a total-order accross nodes (this is due to blockchain forks).

We should note that, because a history is specific to an execution, the results of this analysis cannot generalise to every execution. Thus, the analysis cannot prove the correctness of the implementation. Rather, the analysis can show

Process 1	**Process 2**
Broadcast • Process 1 • MessageNr 0 • Time 1584566082.8416443 Broadcast • Process 1 • MessageNr 1 • Time 1584566082.8416843 ... Deliver • Process 1 • MessageNr 0 • Time 1584566083.0866983 Deliver • Process 1 • MessageNr 1 • Time 1584566083.0867066 ... Deliver • Process 2 • MessageNr 0 • Time 1584566083.7022316 Deliver • Process 2 • MessageNr 1 • Time 1584566083.7022405 ...	Broadcast • Process 2 • MessageNr 0 • Time 1584566083.6448305 Broadcast • Process 2 • MessageNr 1 • Time 1584566083.644878 ... Deliver • Process 1 • MessageNr 0 • Time 1584566084.357182 Deliver • Process 1 • MessageNr 1 • Time 1584566084.0794506 ... Deliver • Process 2 • MessageNr 0 • Time 1584566084.357182 Deliver • Process 2 • MessageNr 1 • Time 1584566084.3571942 ...

Table 5.1: Abbreviated history of process 1 and process 2 of htlltbtest-3.

if there exists an execution that contradicts and violates any safety property. The correctness tests did help us, however, detecting an error in earlier versions causing long forks, causing the total-order property to be violated for the total-order broadcast (because it assumes a bounded fork length).

The total-order property was assessed by asserting that, for each pair of histories (of delivered messages), one is a prefix of the other. The FIFO-order was asserted by checking that the order of delivery was the same as the order of the broadcast. The no duplication property was asserted by checking for any duplicate entries in any of the histories. The no creation was asserted by checking every delivered message was broadcast in one of the histories. We also evaluated the correctness of the implementations through writing tests that would run a similar broadcast-deliver workload, and check the history in retrospective for inconsistencies.

We show an excerpt of the the history of two processes of the htlltbtest-3 execution in Table 5.1. Each process starts broadcasting messages before it starts delivering. The messages of process 1 and process 2 are broadcast concurrently, and not strictly ordered, but the messages are delivered in the same order by both processes (total-order), and delivered in the same order as they were broadcast (FIFO-order).

5.2.2 Throughput

The average measured throughput over the 180 second period is shown in Figure 5.1. The outcome suggests that we can order the protocols in terms of throughput from lowest to highest: totb (lowest throughput), fotb, htlltb, htlltbtest (highest throughput).

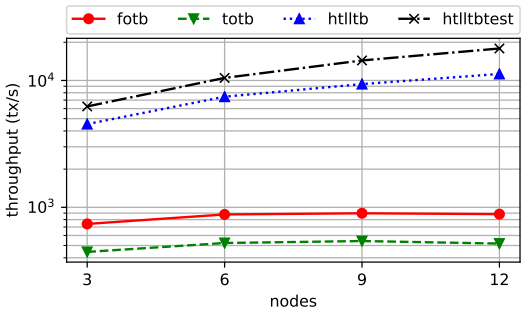

Figure 5.1: Throughput versus number of participating nodes.

The throughput rounded to two significant digits is shown in Table 5.2. The relative throughput achieved by htlltb compared to fotb and totb was 6.1, 8.4, 10, 13 times higher and 10, 14, 17, 21 higher for 3, 6, 9, and 12 nodes respectively.

The throughput of htlltb as relative to htlltbtest was: 0.73, 0.74, 0.67, 0.61. This would qualify htlltb for our definition of "high throughput" (see Section 2.1.3), if we use htlltbtest as the benchmark comparison.

The average throughput (averaged over nodes) is displayed as a time series in Figure 5.2. The figure suggests that the throughput over time is more stable for the htlltb and htlltbtest when compared to totb and fotb. This spiking effect

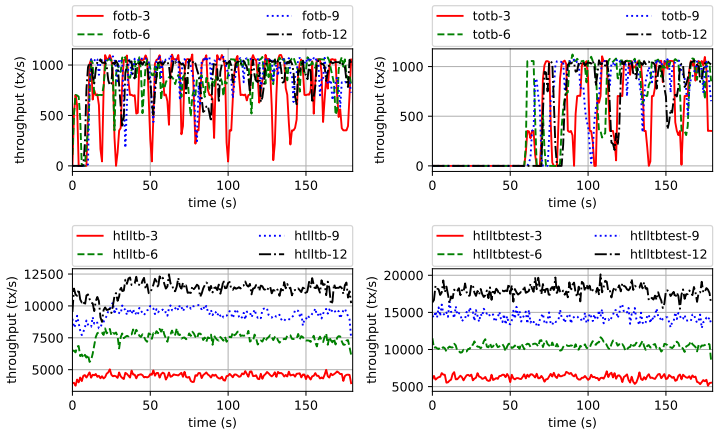

Figure 5.2: Throughput time series.

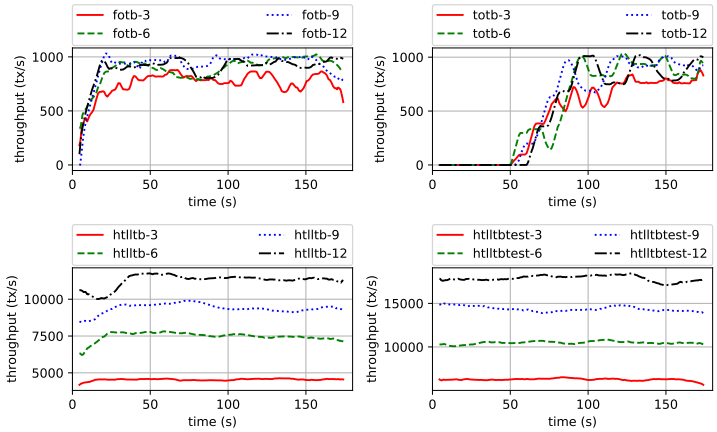

Figure 5.3: Throughput time series of rolling average over 30 second window.

for totb and hotb seems to occur at regular ten-second intervals, suggesting a connection to the block mining frequency.

Figure 5.3 shows the rolling average over a 30 second window of the time series. Herein our attention is drawn to the slow start of fotb and totb. There is a 60 second delay before totb starts delivering messages, due to the protocol

nodes	fotb	totb	htlltb	htlltbtest
3	(tx/s) 740	450	4,500	6,200
6	880	520	7,400	10,000
9	900	540	9,400	14,000
12	880	520	11,000	18,000

Table 5.2: Average throughput (tx/s).

waiting for 6 blocks confirmation, each block taking on average 10 seconds to mine. For fotb, the delay would be 1 block waiting time. This artificially causes the perceived average throughput of totb to decrease, as the protocol delivers messages two-thirds of the workload time (120s out of 180s). Yet, if we adjust for this, by adding 50% to the average throughput of totb, it still is less than that of fotb (0.91, 0.89, 0.90, 0.89).

The theoretical maximum throughput of totb and fotb is 2,700tx/s, if a block of maximum size 8 MB gets mined every 10 seconds, and transactions have a payload of 256 byte. Neither totb nor fotb achieve a peak throughput during a 1 second interval (Figure 5.2), but rather peak at around 1,000tx/s. This hints at inefficiencies of the implementation.

ETCD reportedly can achieve 50,000 writes per second, and 186,000 reads per second [47]. Albeit, this was achieved on a different configuration to ours on Google Cloud Engine, with higher number of CPUs per node, and with a total of 1,000 clients (processes). Our comparative benchmarks, htlltb-12 and htlltbtest-12, achieved 22% and 36% thereof. The reported benchmark achieves 600 writes per second with one client (c.f. 50,000 with 1,000 clients), thus we should expect that there is plenty of room to increase the throughput as the number of processes are increased.

5.2.3 Latency

The average latency in milliseconds of the benchmarks are shown in Figure 5.4. The figure suggests that the latency of the protocols are ordered from highest to lowest: totb (highest latency), fotb, htlltb, htlltbtest (lowest latency).

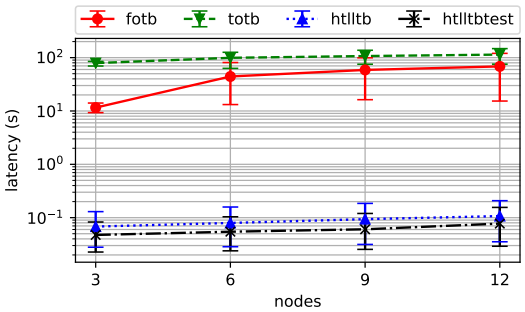

Figure 5.4: Latency.

nodes	fotb	totb	htlltb	htlltbtest
3	(ms) 12,000	79,000	68	47
6	44,000	99,000	80	55
9	59,000	110,000	94	61
12	69,000	110,000	110	77

Table 5.3: Average latency (ms).

The average latencies are shown in Table 5.3, rounded to two significant digits. Both fotb and totb have a comparatively high latency to htlltb. For example, for three nodes, the average latency of fotb and totb is 630 and 1,000 times higher compared to htlltb. The comparatively high latency of totb-3 is explained by the 6-blocks confirmation wait time at 10 seconds per block, similarly for fotb depends on the block mining time. Again, this highlights the benefit of decoupling the latency from the underlying blockchain.

When comparing htlltb to htlltbtest at 3, 6, 9, and 12 nodes, the relative latencies are: 1.45, 1.45, 1.54, 1.43. This partly supports our definition of "fast latency" (see Section 2.1.3).

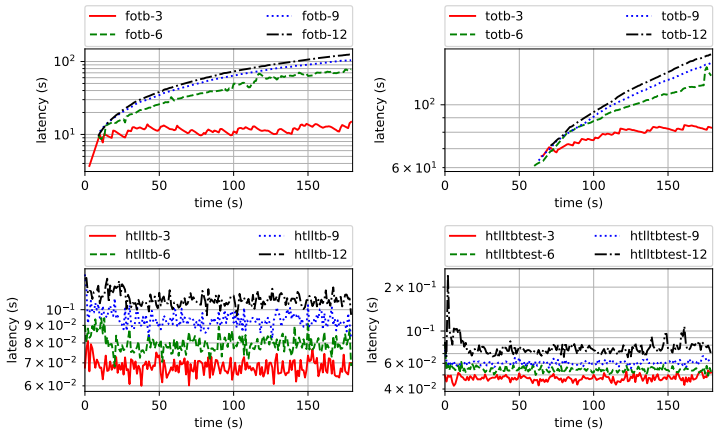

Figure 5.5: Latency time series.

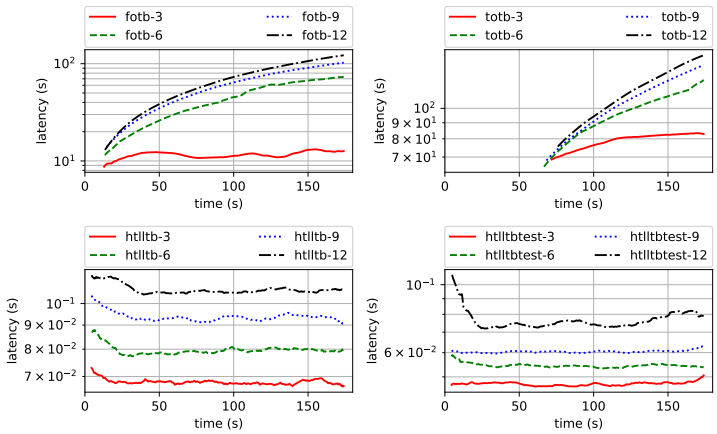

Figure 5.6: Latency time series of rolling average over 30 second window

The latency time series is shown in Figure 5.5. The latency of a message is plotted at the time of delivering the message. This is why there is a 60 second delay before any data points of the totb time series, and 10 seconds of the fotb time series.

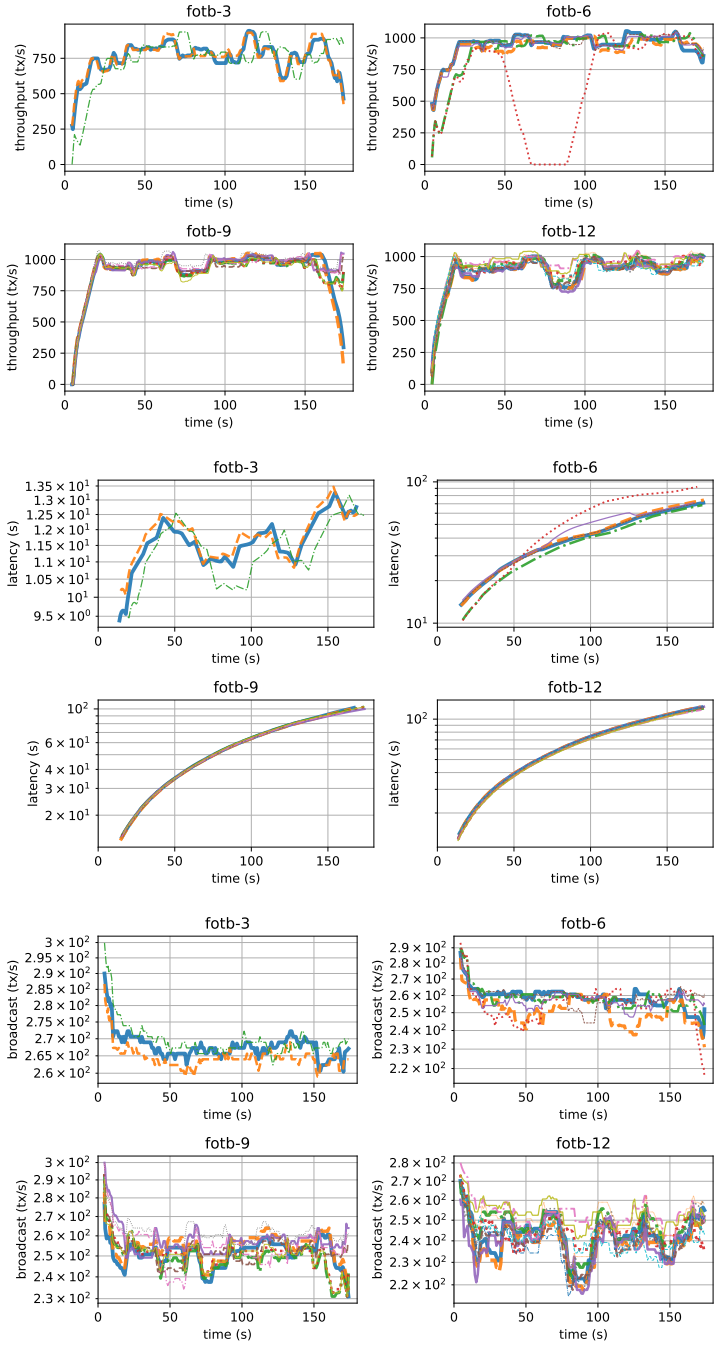

Figure 5.7: Throughput, latency, and broadcast throughput time series of fotb.

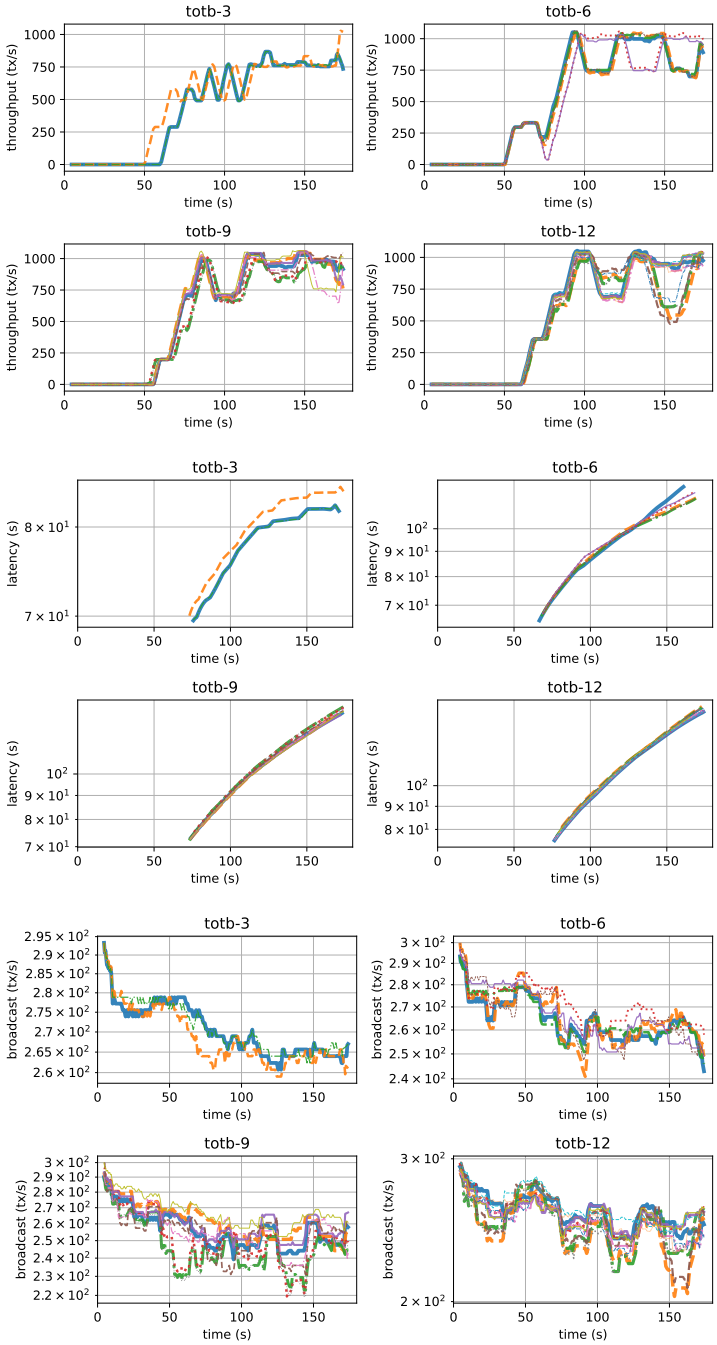

Figure 5.8: Throughput, latency, and broadcast throughput time series of totb.

Again, we notice that the latency of htlltb and htlltbtest appear to be more stable accross the 180 seconds. In particular, the increasing latency over time for fotb-6-9-12 and totb-6-9-12 is surprising. The latency surpasses 100 seconds for totb-6-9-12, and for fotb-9-12, causing no broadcast message after 80 seconds to be delivered before the end of the workload (180 seconds). Figure 5.7 and 5.8 show the throughput, latency, and broadcast throughput (the number of broadcast messages (delivered and un-delivered)), of each individual node of fotb and totb. The figure shows that the throughput and broadcast remain relatively stable compared to the latency. Whereas the latency steadily increases. The broadcast throughput (Table 5.4) of fotb is 1.08, 1.70, 2.56, 3.30 times larger compared to the throughput (Table 5.2). This suggests that the system cannot deliver the messages at the same rate as they are being broadcast, causing longer queues and thus increasing latency.

nodes	fotb	totb	htlltb	htlltbtest
3	(tx/s) 800	810	4,500	6,200
6	1,500	1,600	7,500	10,000
9	2,300	2,300	9,400	14,000
12	2,900	3,100	11,000	18,000

Table 5.4: Average broadcast throughput (broadcast tx/s).

The reported (write) latency of ETCD is 20ms for 1,000 concurrent clients, and 1.6ms for 1 client [47]. Compared to this, our protocol htlltb-12 is 5.5 times slower, and htlltbtest-12 3.9 times slower. This shows that our implementation might have inefficiencies which could be optimised and corrected.

Chapter 6

Use Cases

Blockchain has the possibility to impact a wide range of problems and fields, including [6]: data management and the secure storage of data; supply chain, for example logistics providing origin tracking and identifying counterfeit products; integrity verification such as tamper-resistant and verifiable storage of data (time stamping). Among future trends is the continued work on blockchain adoption and interoperability, this would include, for example, blockchain virtualisation. We will review these use cases, and how they can be implemented with the tamper-resistant broadcast abstraction.

6.1 Design Pattern: Tamper-Resistant Replicated State Machine

The *tamper-resistant replicated state machine* is a generalisation of how to implement a tamper-resistant (replicated) application, and shows the general technique employed. We have already discussed the replicated state machine in Section 2.1 but as a non-tamper-resistant variant. What makes this different, or, what makes a replicated state machine tamper-resistant? The difference to the replicated state machine, is the addition of the tamper-resistant property and verification protocol.

The Tamper-Resistant Replicated State Machine is shown in Module 22. The interface is based on the the replicated state machine interface as defined Cachin et al. [1, Module 6.12]. The tamper-resistant property requires the current state (and history of states) of the replicated state machine to be the probabilistically (with high probability) correct state (untampered). This has to be verifiable. The verification entails, in general, either proving that the history of operations that caused the state is untampered, or that the state (and history of states) is untampered with. We opt for the former. The verification request will (with high probability) prove that the untampered history of operations produce the current state.

The implementation of the tamper-resistant replicated state machine is shown in Algorithm 23. The implementation uses an instance of the tamper-resistant broadcast. A uniform reliable total-order broadcast is sufficient for implementing a replicated state machine [1, ch. 6].

The algorithm broadcasts any incoming commands via the tamper-resistant

Module 22 Tamper-Resistant Replicated State Machine

Module name: Tamper-Resistant Replicated State Machine, **instance** trsm

Events:

E1 *Request:* < trsm, Execute | command >: Execute *command* on state machine.

E2 *Indication:* < trsm, Output | response >: : Return *response* from *command* executed on state machine.

E3 *Request:* < tb, Verify >: Verify current *state* of replicated state machine.

E4 *Indication:* < tb, VerifyReturn | "Valid"/"NotValid" >: Returnvalue for verification request.

Properties:

P-1 *Tamper-resistant:* The replicated state machine is probabilistically (with high probability) protected against tampering, and the output of the replicated state machine is probabilistically verifiable. A verification request returns "Valid" (with high probability), if the replicated state machine has not been tampered with, else "NotValid".

Algorithm 23 Tamper-Resistant Replicated State Machine

Implements: Tamper-Resistant Replicated State Machine, **instance** trsm

Uses: Tamper-Resistant Broadcast, **instance** tb

```
upon event < trsm, Init > do
    state = initial state
    log = [ ]

upon event < trsm, Execute | command > do
    trigger < tb, Broadcast | command >

upon event < tb, Deliver | pid, txid, command > do
    returnvalue, state = execute(command, state)
    trigger < trsm, Output | returnvalue >
    log.append( command )

upon event < trsm, Verify > do
    if tb.Verify( command ) is "Valid" for each command in log
    ∧ log produces current state do
        trigger < trsm, VerifyReturn | "Valid" >
    else
        trigger < trsm, VerifyReturn | "NotValid" >
```

broadcast. A command consists of the operation and arguments to be executed on the state machine. Delivered commands via the tamper-resistant broadcast are added to a log of commands, and executed on the state machine.

Correctness The tamper-resistant property follows from the use of the tamper-resistant broadcast. The broadcast commands are protected against tampering, and the Verify request returns "Valid", if the history of commands that produce the current state are verified to not having been tampered with, else "NotValid".

If a uniform total-order broadcast is used, the implementation also provides the agreement and termination properties [1, ch. 6].

Implications The tamper-resistant replicated state machine is a general approach of creating tamper-resistant applications. The verification functionality could be extended to include the verification of historical states, the verification of a state passed as an argument, the verification of an operation, etc.

6.2 Use Case: Tamper-Resistant File System

The first use case we study is the secure storage of data using blockchain. We also refer to it as a *blockchain-based file system* or a *tamper-resistant file system*. This includes our ongoing work on a blockchain-based file system [5]. Other examples from literature include a data provenance system for cloud data storage [28], and a Ethereum-blockchain-based file system [48]. A tamper-resistant file system is essential for certain applications that require the data to be securely stored, and for which it is necessary to produce a verifiable proof. An example of this is the storage of financial data of business entities in Germany [5]. This requires providing tamper-resistant record-keeping, motivating the use of blockchain technology.

For this use case, we present an implementation of a multi-user file system (Algorithm 24). We use the tamper-resistant replicated state machine design pattern. The local file system is the replicated state machine, and the commands executed are the file system operations that the clients request to execute.

The correctness of the implementation relies on the correctness of the local file system, and on the properties of the tamper-resistant broadcast. We would recommend using the high throughput low latency total-order causal-order tamper-resistant broadcast. This would guarantee the sequential-consistency on the file system (see Section 2.1). Other models could also be used. For example, if we can guarantee that the files and directories of different users are disjoint, it would be sufficient to require the FIFO-order, rather than total-order and causal-order.

6.3 Use Case: Timestamp Server

Providing a means for verifying the integrity and existence of data, also known as timestamping, is a service provided by Factom [25] and Chainpoint [26]. This involves providing a verifiable proof that the data existed at, or before, a specific date and time. Here, we show a minimal example of how this can be implemented using the tamper-resistant replicated state machine approach.

Algorithm 24 Multi-User Tamper-Resistant File System

Implements: Multi-User Tamper-Resistant File System, **instance** mufs

Uses:
Tamper-Resistant Broadcast, **instance** tb
Local File System, **instance** lfs

upon event < mufs, Init > **do**
lfs = initialise local file system
log = [ ]

upon event < mufs, Execute | command > **do**
trigger < tb, Broadcast | command >

upon event < tb, Deliver | pid, txid, command > **do**
returnvalue = lfs.execute(command)
if pid is my pid **do** \\if it was my command
trigger < mufs, Output | returnvalue >

upon event < mufs, Verify > **do**
Same as Algorithm 23

Algorithm 25 Timestamp Server

Implements: Timestamp Server, **instance** ts

Uses: Tamper-Resistant Broadcast, **instance** tb

upon event < ts, Init > **do**
map = { }

upon event < ts, Timestamp | file > **do**
trigger < tb, Broadcast | hash(file) >

upon event < tb, Deliver | pid, txid, filehash > **do**
map.add(filehash → txid)

upon event < ts, Verify | file > **do**
txid = map.get(hash(file))
trigger < ts, VerifyReturn | tb.Verify(txid) >

The implementation is shown in Algorithm 25. The Timestamp Server abstraction consists of two operations: Timestamp and Verify. When a user requests to timestamp a new file, the filehash (hash value of the file) is broadcast. Upon delivering a filehash, we add a new entry to the map, mapping the filehash to the transaction id. When a user attempts to verify a file, the filehash is calculated, the transaction id is recovered from the map, and we return the proof from the tamper-resistant broadcast. From the returned proof, the age of the transaction id can be inferred, and by this the age of the file.

We would suggest to use an instance of high throughput low latency uniform reliable tamper-resistant broadcast, to service the needs for a large number of

Algorithm 26 Virtual Blockchain

Implements: Virtual Blockchain, **instance** vb

Uses: Tamper-Resistant Broadcast, **instance** tb

```
upon event < vb, Init > do
    state = initial state
    ledger = [ ]

upon event < vb, Append | transaction > do
    trigger < tb, Broadcast | transaction >

upon event < tb, Deliver | pid, txid, transaction > do
    if transaction conform with consensus rules at state do
        returnvalue, state = execute( state, transaction )
        ledger.append( transaction )

upon event < vb, GetLedger > do
    trigger < vb, GetLedgerReturn | ledger >

upon event < bv, Verify > do
    if tb.Verify( transaction ) is "Valid" for each transaction in ledger
    ∧ ledger produces current state do
        trigger < trsm, VerifyReturn | "Valid" >
    else
        trigger < trsm, VerifyReturn | "NotValid" >
```

users. It is not necessary to restrict the ordering guarantees on the broadcast abstraction. The uniform reliability property, guarantees that if a filehash has been timestamped (delivered), then it is also eventually timestamped by all other correct servers.

We are uncertain about the possibility of bounding the time between the Timestamp request and the finalised append to the blockchain. This implies that we cannot put an upper bound on the time difference between the Timestamp request and the latest creation time of the returned timestamp proof. We do note, however, that larger time differences are increasingly unlikely, with probability converging to 0 (see probabilistic finality property of Module 4).

6.4 Use Case: Virtual Blockchain

The third use case is blockchain virtualisation. The Virtual Blockchain design pattern can be used to define an arbitrary blockchain. The implementer can arbitrarily define the consensus rules and the state machine behaviour. The benefit is that new applications (blockchains) can be implemented without changes to the code of the underlying blockchain layer, decoupling the security from the application logic [4]. Tendermint Core uses a similar approach, decoupling the application logic (state machine and consensus rules) from the consensus mechanism, to enable user defined blockchains [23].

The blockchain virtualisation implements the Blockchain Ledger interface

(Module 3). It consists of five events: Append; Get; GetReturn; Verify; and VerifyReturn.

The Virtual Blockchain is implemented in Algorithm 26. It uses a tamper-resistant broadcast instance. When the user requests to append a transaction, it is broadcast via the tamper-resistant broadcast. The transaction is delivered, and if it conforms to the consensus rules at the current state of the state machine, the transaction is executed on the state machine and added to the ledger. The GetLedger request returns the latest ledger. The implementation is similar to the Tamper-Resistant Replicated State Machine implementation. The difference, is that transactions are checked for conformity with the consensus rules at the current state before being executed by the state machine and added to the log.

Correctness Because every append triggers a broadcast, and every deliver causes a transaction to be executed (if the transaction is conform), the consistency, finality, append properties are inherited from the used tamper-resistant broadcast model. The tamper-resistant property is inherited from the tamper-resistant broadcast model. The transaction validity property holds true (according to the consensus rules), because every transaction is checked for conformity with the consensus rules, thus only valid transactions are appended to the ledger.

Implications The blockchain virtualisation can be used to create new blockchains and new blockchain applications. For example, it can be used to create virtual currencies, i.e. a cryptocurrency that is not native to a blockchain.

6.5 Use Case: Supply Chain Tracking

The last use case we study is supply chain tracking. We will use the blockchain virtualisation design pattern which requires us to specify the state machine and the consensus rules.

For this example we consider two roles, or types of end-users, that are represented by the blockchain:

- *Businesses:* Businesses are producers, merchants and other intermediaries that own and modify the product at some point from moment of production to consumption. We denote a business by the letter B with a subscript identifier, e.g. B_1.
- *Consumers:* Consumers consume a product and become the final owners of it. We denote consumers by the letter C with a subscript identifier, e.g. C_1.

The state machine can execute the following transactions. The consensus rules define the preconditions for a valid transaction. The outcome describes the effect of a valid transaction:

- Transaction: Business B_1 produces a new product p.
  - Precondition: p has never been produced before.
  - Precondition: Transaction has been signed for by B_1.
  - Outcome: B_1 becomes new owner of p.

- Transaction: Business B_1 transfers a product p to a new owner B_2.
  - Precondition: B_1 is owner of p.
  - Precondition: Transaction has been signed for by B_1 and B_2.
  - Outcome: B_2 becomes new owner of p.
- Transaction: Consumer C consumes product p from business B_1.
  - Precondition: B_1 is owner of p.
  - Precondition: Transaction has been signed for by B_1.
  - Outcome: C becomes final owner of p.

The proposed supply chain tracking protocol can be implemented by using the Virtual Blockchain Algorithm 26, through substituting in the aforementioned state machine specification and consensus rules.

Chapter 7

Discussion

We have now presented a variety of *tamper-resistant broadcast* protocols, and described how they can be used to realise *tamper-resistant replicated state machines*. Our study was motivated by the topic of how to implement tamper-resistant replicated state machines. We defined tamper-resistance as 1) the probabilistic (with high probability) protection against byzantine behaviour, and 2) the probabilistic verifiability that no tampering has occurred. We chose to study blockchain-based tamper-resistant broadcast protocols. We studied various *environment models* and *safety and liveness properties* of the tamper-resistant broadcast protocols in order to support various environment models and consistency models of tamper-resistant replicated state machines. We evaluated the protocols and showed how they can be used for four use cases.

The tamper-proof and tamper-resistant definitions are derived from byzantine fault tolerance and probabilistic byzantine fault tolerance respectively, but differ by the addition of an explicit verification protocol. We believe our definitions of tamper-proof and tamper-resistant are useful, especially for web-applications in which a client cannot trust the web server. The definitions may, however, be criticised of being too vague. For example, it is not clear what tampering is. An alternative, more specific, definition could use the *universal composable security*, a framework for analysing cryptographic protocols [49]. With it we can describe the correct behaviour of the protocol for all correct processes, protected from actions by byzantine processes. Similar to our definition, in the universal composable security framework, assumptions are required for general functionalities in the so called plain model (requiring an honest majority) [50]. Assumptions are integral to make the definition practical and feasible. Besides the design of *mathematically* tamper-proof applications (as done with the universal composable security framework), it remains a challenge to design *legally* tamper-proof applications, from a legal viewpoint.

We found that the Bitcoin Blockchain cannot be considered tamper-proof, but rather tamper-resistant. This is in part due to blockchain forking, which can cause transactions to be reordered, thus invalidating any previously exported proofs. It also causes the latency between a message being broadcast, and the time it is written to the blockchain ledger, to be unbounded. Both issues affect the verification protocol, but do not affect the safety or liveness properties of the presented protocols. It may be possible to circumvent these difficulties by using blockchain technologies other than Bitcoin.

In general, the tamper-proof property is not achievable in the public environment model. The tamper-resistance property of the Bitcoin blockchain is a compromise for this particular environment. For the permissioned environment model, however, it is possible to define tamper-proof protocols. This includes Tendermint [23], that implements a byzantine fault tolerant blockchain with transaction finality. The protocols which we discussed would be tamper-proof, if we ran them on a permissioned Bitcoin compatible blockchain with transaction finality. We should note that the distinction between public and permissioned may not be that clear, as some may argue that proof-of-stake blockchains such as Tendermint [23] could be considered public.

The security aspects of real-world tamper-proof and tamper-resistant blockchains should also be considered. For example, a tamper-proof blockchain on a permissioned network is associated with risks such as centralisation and censorship, and the risk of shut-down. This is not the case for tamper-resistant blockchains on public networks. On a public network there are no risks of centralisation and censorship, but tamper-resistant blockchains provide weaker consistency semantics. There may be certain applications for which transaction finality and tamper-proof are necessary properties, thus rendering tamper-resistant public blockchains unsuitable. Other applications, however, may be sufficiently protected by the tamper-resistance guarantee, and require censorship resistance. This includes the Bitcoin fork namecoin [4], a decentralised and censorship resistant domain name system.

The discussed tamper-resistant protocols did not explicitly return the proof, albeit this is necessary to support local distrust. As we noted in the Introduction, Section 1.5, we assumed that generating the proof is trivial if the verification protocol is correct.

The blockchain ledger abstraction consists of the events: Append, Get, GetReturn, Verify, and VerifyReturn. We found that there are disagreements on the properties and consistency guarantees of the Bitcoin blockchain in literature [34]. We characterise the Bitcoin blockchain by the properties: probabilistic eventual consistency, probabilistic finality, fair append, transaction validity, and tamper-resistant. These properties rest on the assumption that less than 1/3 of participants are byzantine. If we would like to derive that the blockchain is eventual consistent, we would need to assume the bounded fork length, which is in contradiction to the asynchronous distributed system model. Future work should consider extending the work to include other blockchains and transaction costs.

The tamper-resistant broadcast consists of the events: Broadcast, Deliver, Verify, and VerifyReturn. We found a large corpus of research that applied to tamper-proof and tamper-resistant broadcast, and we believe that blockchain-based tamper-resistant broadcast was a suitable way to study the problem. In hindsight, however, we would have chosen to study tamper-resistant *distributed ledger* protocols, i.e. a simplified blockchain abstraction consisting of a ledger of records, and the operations append a record; get the ledger; and verify the ledger. The benefit of this abstraction we think is the less awkward transformation of the Blockchain Ledger to the Distributed Ledger (compared to Blockchain Ledger to Broadcast). Additionally, the distributed ledger abstraction would allow us to describe notions such as eventual consistency, which is very awkward, or not possible, to describe on the broadcast abstraction. An alternative would be to add an *Undo-Delivery* event to the tamper-resistant broadcast abstraction, to

undo any transactions that got lost due to forking. This may be an opportunity for further exploration.

We considered three different environment models, which we named: public tamper-resistant broadcast protocols; permissioned tamper-resistant broadcast protocols; non-native permissioned tamper-resistant broadcast protocols. It is impossible to achieve consensus in the public environment, assuming that the number of participants is unknown and that the number of faulty processes is unbounded [15]. We did not explore other possibilities such as partial synchrony, or bounding the number of faulty processes. We did, however, deal with this issue by introducing permissioned protocols, to bound the number of processes to a known set of processes. These protocols could be extended with the notion of dynamic groups and epochs. The non-native permissioned broadcast protocols decoupled the broadcasting from the blockchain, and allowed us to design models that achieved high throughput and low latency.

We executed a benchmark, consisting of 180 seconds of alternative broadcast and delivering messages, on a cluster spanning three physically separated locations, varying the number of participating processes from three to twelve.

The results should only act as a guide on how each protocol performs, rather than a precise evaluation thereof. We did not evaluate the benchmark repeatedly, thus we cannot judge what statistical deviations would have influenced the outcome. Furthermore, we should have implemented our protocols without batching, in order to make the results more reproducible and comparable.

The correctness of the protocols were evaluated, using the history of broadcast and delivered messages from the benchmark. We could not find any violations of the proposed safety properties, but this does not guarantee the correctness of the protocols.

The throughput and latency was evaluated, and the results were as expected. In order of increasing throughput and decreasing latency, the protocols are ordered: totb (lowest throughput, highest latency); fotb; htlltb; htlltbtest (highest throughput, lowest latency). We found that the High Throughput Low Latency Tamper-Resistant Broadcast (htlltb) improved the throughput by 6.1 times and latency by 176 times compared to FIFO-order Tamper-Resistant Broadcast (fotb), and throughput by 10 times and latency by 1,200 times compared to (Uniform Causal-Order) Total-Order Tamper-Resistant Broadcast (totb). The results also showed that htlltb caused negligible overhead (i.e. less than 50% difference) compared to its non-tamper-resistant counterpart (htlltbtest). This supports our statement that the htlltb can achieve high throughput and low latency, according to our definition (Section 2.1.3).

The performance of htlltb (throughput 11,000 tx/s, latency 110 ms) is favourable when compared to blockchains: the public Bitcoin Blockchain attains 600 seconds latency and 7tx/s throughput; the public Ethereum Blockchain attains 15 seconds and 15tx/s to 40tx/s; the private Ethereum Blockchain attains 1'000tx/s; the Tendermint Blockchain <1s and "tens of thousands" tx/s [20]. The MultiChain instance we used has a peak throughput of 2,700 tx/s and average latency of 5 seconds (half the block mining time). This highlights the benefit of decoupling the performance aspects of the tamper-resistant broadcast protocol from the blockchain performance.

We expect the native protocols (fotb and totb) to reach the peak throughput (2,700 tx/s) if we increase the number of participants. If we increase the number of participants of the non-native protocols (htlltb and htlltbtest), we expect to

reach the peak throuhput of ETCD (50,000 tx/s [47]). If we increase the number of ETCD nodes in the cluster, then we expect a drop in the throughput. This is discussed in a study that compared consensus protocols on scalability, ETCD was reported to achieve 5700 tx/s for 3 nodes, and 490 tx/s for 100 nodes [51].

Implementing the protocols was not easy, and would require several small protocol extensions for making it suitable for real-world applications. This has been reported of Paxos implementations [52]. We shoud expect that the performance of our implementation could still be improved. For fotb and totb, this is evident because of the issues with increasing latency. The protocols (fotb, totb) peaked at 1,000 tx/s, less than half of the theoretical peak performance 2,700 tx/s. For htlltb and htlltbtest, we would also expect possibility to increase the throughput and decrease the latency, as when compared to an ETCD benchmark [47], htlltb and htlltbtest achieved only 22% and 36% throughput and 5.5 and 3.9 times slower latency respectively.

We showed on three use-cases, a multi-user file system, a supply chain tracking application, and timestamp server, how we can design tamper-resistant applications using the tamper-resistant broadcast abstraction. The use-cases showed the direct relationship between our work and related work. We showed simplified protocols for the Virtual Blockchain, in parallel to the VirtualChain of Blockstack [4]; the Timestamp Server with correspondence to Factom [25] and Chainpoint [26]; and the tamper-resistant file system as similar to Endolith [28] and our ongoing work [5]. We hope to have conveyed the shared principles between different implementations.

In summary, we have discussed what we consider to be important aspects and hope to have shown how to design tamper-resistant applications through the use of the presented tamper-resistant broadcast protocols.

Chapter 8

Conclusion

We studied blockchain-based *tamper-resistant broadcast* protocols and how they can be used to implement *tamper-resistant replicated state machines*. We studied various environment models, and safety and liveness properties of the tamper-resistant broadcast, this allows us to match the consistency and environment requirements of the application. The presented protocols, and provided implementation, should provide a sufficient base for the development of tamper-resistant applications using the Bitcoin blockchain.

We found it important to distinguish between the environment models. For example, total-order broadcast, consensus, and tamper-proof cannot be implemented in the public environment model, whereas they can be implemented in the permissioned environment model. Furthermore, we found it necessary to decouple the throughput and latency from the blockchain, and use non-native (i.e. not via blockchain) techniques for consensus and communication in order to achieve high throughput and low latency. We implemented three of the protocols as a proof of concept, and showed that the non-native permissioned protocols can compete on performance with other state-of-the-art technologies.

We chose to limit the study to a Bitcoin compatible blockchain with zero-fee transactions. The tamper-resistant broadcast protocols inherit the tamper-resistant property from the Bitcoin blockchain through composition. In general, tamper-proof cannot be achieved under the public environment model, whereas tamper-resistant can be achieved. The reported results are limited by not executing the experiments in a repeated fashion.

We recommend further work on the definitions of *tamper-proof* and *tamper-resistant*, and on such replicated state machines and broadcast protocols. For work on *tamper-resistant broadcast protocols*, we recommend to focus on the aforementioned limitations, continued work to include support for more blockchains, and improving the implementations for robustness and efficiency. In hindsight, we would suggest using the tamper-resistant distributed ledger abstraction instead of tamper-resistant broadcast, as it more naturally fits the task. We believe that the presented ideas can be extrapolated to the tamper-proof property.

We hope to have demonstrated the tamper-resistant broadcast as a powerful interface with clear semantics and tunable settings. We believe our work helps the design of tamper-resistant applications, and contributes towards the transition from human-centric and centralised forms of trust, to decentralised forms of trust-networks.

Bibliography

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

[2] Andreas M Antonopoulos. *Mastering Bitcoin: Programming the open blockchain.* O'Reilly Media, Inc., 2017.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[4] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *2016 USENIX Annual Technical Conference*, pages 181–194, 2016.

[5] Pacio – audit-proof, distributed archival service, Last visited 2019-05-13. URL `https://www.zib.de/projects/pacio`.

[6] Fran Casino, Thomas K Dasaklis, and Constantinos Patsakis. A systematic literature review of blockchain-based applications: current status, classification and open issues. *Telematics and Informatics*, 36:55–81, 2019.

[7] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design.* Pearson Education, 2005.

[8] Bitcoin developer reference: Bitcoin core apis, Last visited 2020-01-17. URL `https://bitcoin.org/en/developer-reference#bitcoin-core-apis`.

[9] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[10] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, SE-3(2):125–143, 1977.

[11] Consistency models, Last visited 2020-01-21. URL `https://jepsen.io/consistency`.

[12] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: virtues and limitations (extended version). *arXiv preprint arXiv:1302.0309*, 2013.

[13] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):1–34, 2016.

[14] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[15] Kenji Saito and Hiroyuki Yamada. What's so different about blockchain? — blockchain is a probabilistic state machine. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 168–175. IEEE, 2016.

[16] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Consensus in the cloud: Paxos systems demystified. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–10. IEEE, 2016.

[17] Douglas Robert Stinson and Maura Paterson. *Cryptography: Theory and Practice*. CRC Press, 4 edition, 2018.

[18] Blockchain guide, Last visited 2020-01-17. URL `https://bitcoin.org/en/blockchain-guide`.

[19] Andrew Sward, Ivy Vecna, and Forrest Stonedahl. Data insertion in bitcoin's blockchain. *Ledger*, 3, 2018.

[20] Marianna Belotti, Nikola Božić, Guy Pujolle, and Stefano Secci. A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838, 2019.

[21] Muneeb Ali, Ryan Shea, Jude Nelson, and Michael J Freedman. Blockstack technical whitepaper, 2017, Last visited 2020-01-17. URL `https://pdos.csail.mit.edu/6.824/papers/blockstack-2017.pdf`.

[22] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains, 2014, Last visited 2020-01-17. URL `https://blockstream.com/sidechains.pdf`.

[23] What is tendermint?, Last visited 2020-01-17. URL `https://github.com/tendermint/tendermint/blob/master/docs/introduction/what-is-tendermint.md`.

[24] Jae Kwon and Ethan Buchman. Cosmos whitepaper, Last visited 2020-01-17. URL `https://cosmos.network/resources/whitepaper`.

[25] Factom business processes secured by immutable audit trails on the blockchain, Last visited 2020-01-17. URL `https://www.factom.com/assets/docs/Factom_Whitepaper_v1.2.pdf`.

[26] W Vaughn, Jason Bukowski, R Shea, C Allen, P Storz, and J Nelson. Chainpoint–a scalable protocol for anchoring data in the blockchain and generating blockchain receipts v2.1.0, 2016, Last visited 2020-01-17. URL `https://github.com/chainpoint/whitepaper/blob/master/chainpoint_white_paper.pdf`.

[27] W Vaughn, Jason Bukowski, and Glenn Rempe. Tierion network–a global platform for verifiable data, 2017, Last visited 2020-01-17. URL `https://tokensale.tierion.com/TierionTokenSaleWhitePaper.pdf`.

[28] Thomas Renner, Johannes Müller, and Odej Kao. Endolith: A blockchain-based framework to enhance data retention in cloud storages. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 627–634. IEEE, 2018.

[29] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. Blockchaindb: a shared database on blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019.

[30] Jude Nelson, Muneeb Ali, Ryan Shea, and Michael J Freedman. Extending existing blockchains with virtualchain. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[31] Blockchain middle layer services will be as valuable and free as air, Last visited 2020-01-17. URL `https://dailyfintech.com/2018/08/30/blockchain-middle-layer-services-will-be-as-valuable-and-free-as-air/`.

[32] Layer 2 blockchain technology: Everything you need to know, Last visited 2020-01-17. URL `https://golucidity.com/layer-2-blockchain-technology/`.

[33] Léo Besançon, Catarina Ferreira Da Silva, and Parisa Ghodous. Towards blockchain interoperability: Improving video games data exchange. In *IEEE International Conference on Blockchain and Cryptocurrency*, 2019.

[34] Emmanuelle Anceaume, Romaric Ludinard, Maria Potop-Butucaru, and Frédéric Tronel. Bitcoin a distributed shared register. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 456–468. Springer, 2017.

[35] Antonio Fernández Anta, Kishori Konwar, Chryssis Georgiou, and Nicolas Nicolaou. Formalizing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.

[36] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

[37] Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, page 13. ACM, 2016.

[38] Number of confirmations, Last visited 2020-04-01. URL `https://en.bitcoin.it/wiki/Confirmation#Number_of_Confirmations`.

[39] Source code, Last visited 2020-04-07. URL `https://github.com/jonasspenger/mscthesis`. Used commit short hash 1db6931.

[40] Rachid Guerraoui, Ron Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems*, 28(ARTICLE):5, 2010.

[41] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 527–536. IEEE, 2010.

[42] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[43] Gideon Greenspan. Multichain private blockchain–white paper. *URl: https://www.multichain.com/download/MultiChain-White-Paper.pdf*, 2015, Last visited 2019-05-13.

[44] Etcd download, Last visited 2020-04-01. URL `https://bitnami.com/stack/etcd/containers`.

[45] Multichain download, Last visited 2020-04-01. URL `https://www.multichain.com/download-community/`.

[46] Machine types, Last visited 2020-04-01. URL `https://cloud.google.com/compute/docs/machine-types`.

[47] Etcd benchmarks, Last visited 2020-04-01. URL `https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/performance.md#benchmarks`.

[48] Xueping Liang, Sachin Shetty, Deepak Tosh, Charles Kamhoua, Kevin Kwiat, and Laurent Njilla. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *Proceedings of the 17th IEEE/ACM international symposium on cluster, cloud and grid computing*, pages 468–477. IEEE Press, 2017.

[49] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[50] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 115–128. Springer, 2007.

[51] Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi, and Marko Vukolic. Can 100 machines agree? *arXiv preprint arXiv:1911.07966*, 2019.

[52] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

Appendix A

Data Insertion Methods

Regular transactions store only the necessary information for transferring funds from one locking script to another. This is a review of how arbitrary data can be stored in transactions on the blockchain.

A.1 Characteristics

Blockchain data insertion methods can be compared on the following characteristics [19]:

- *Data efficiency:* The ratio of the data size to the transaction size.
- *Cost efficiency:* The cost (transaction fee) per byte data that is saved.
- *Max data per transaction:* Maximum amount of data that can be saved in a transaction.
- *Security and data integrity:* Transactions should be resistant to unintended modification of the transaction. For example, sniping is the process of changing the unsigned outputs of a transaction to new outputs, this can also include reordering of outputs. Transaction malleability, includes possibility of blockchain nodes to modify transactions whilst preserving the validity of the transactions.
- *Unspendable UTXOs:* Unspendable UTXOs are caused by writing data instead of a valid public key hash to the locking script. This causes the output to be practically unspendable, and bloating the set of UTXOs.

We are only interested in methods that do not generate unspendable UTXOs and are not vulnerable to security and data integrity issues. The only qualifying methods out of the presented methods [19] are: OP_RETURN and Data hash (w/ sig), which we will elaborate further.

A.2 OP_RETURN

The *OP_RETURN* method [19] is achieved by adding data to an output with the following output script to a transaction:

- *Output script:* `OP_RETURN <DATA>`

It is protected against tampering as data is written to the signed outputs. Further, the output of OP_RETURN is defined by the bitcoin standard to not be part of the UTXO set, thus avoiding the unspendable UTXO problem.

The max data per transaction is 80 byte. The data efficiency is 25% for 80 byte of data. At most one OP_RETURN output can be added to a standard bitcoin output.

A.3 Data Hash w/ Sig

The *Data Hash w/ Sig* method [19] requires two transactions to write one data. The first transaction writes in its output scripts the hash of the data. The second transaction writes in its input scripts the un-hashed data.

The following output script is used in the first of two separate transactions. It contains a redeem-script-hash, which is the hash of the redeem script used by the input script (see below).

- *Output Script:* `OP_HASH160 <REDEEMSCRIPTHASH> OP_EQUAL`

The following input references the just mentioned output. The input script contains the data, which is securely verified by the hashed redeem script in the output script.

- *Input Script:* `<SIG> <DATA1> <DATA2> <DATA3> <REDEEMSCRIPT>`
- *Redeem Script:* `OP_HASH160 <DATA3HASH> OP_EQUALVERIFY OP_HASH160 <DATA2HASH> OP_EQUALVERIFY OP_HASH160 <DATA1HASH> OP_EQUALVERIFY <PUBKEY> OP_CHECKSIG`

Data is protected from modification by writing a hash of the data in the redeemscript. Reordering of the inputs is avoided by signing each input. Because of this, Data Hash w/ Sig is not vulnerable to sniping or transaction malleability. Further, it does not generate any unspendable UTXOs, as the generated UTXO by the output script is spent by the input script in a following transaction.

The max data per transaction is 86,087 byte. The data efficiency is 95% for 86,087 byte of data, and 16% for 80 byte.

A.4 Data Efficiency

Data efficiency is calculated as the ratio of the data size to the transaction size. If the method has higher data efficiency it can achieve higher throughput of data stored to the blockchain. We compare the data efficiency of the methods *OP_RETURN*, *Data Hash w/ Sig*, as well as two derivations *"var OP_RETURN"* and *"max OP_RETURN"*:

- *Data efficiency:* The data efficiency is calculated as the ratio of the data size to the transaction size.
- *Data size:* The amount of data to be saved to the blockchain in a transaction.

- *Transaction size:* The transaction size is calculated from a transaction, which additionally to the input and output scripts of the data saving method also contains a standard P2PKH transaction input and output.
  - *P2PKH:* The transaction metadata and P2PKH input and output account for: 204 byte[1].
  - *OP_RETURN:* The OP_RETURN output accounts for: data size byte + 18 byte[2].
  - *Data Hash w/ Sig:* The input and output roughly account for: data size + ROOF(data size / 1461) * 277 byte[3].
  - *var OP_RETURN:* We also consider the possibility of allowing more than a single OP_RETURN output per transaction, each writing max 80 byte of data.
  - *max OP_RETURN:* We also consider the possibility of allowing a single OP_RETURN to write an unlimited amount of data.

We compare the methods *OP_RETURN*, *Data Hash w/ Sig*, as well as two derivations *"var OP_RETURN"* and *"max OP_RETURN"*. Var OP_RETURN is a scenario in which there is no limit on the number of OP_RETURN outputs per transaction. The bitcoin standard limit is one. Max OP_RETURN is a scenario in which there is no limit the size of data written to OP_RETURN. The bitcoin standard limit is 80 byte.

The data efficiency comparison is shown in Table A.1. OP_RETURN is most efficient for data up to 80 byte. For data larger than 80 byte, the most data efficient method would be max OP_RETURN (no limit on OP_RETURN data size), or also a mix of var OP_RETURN and max OP_RETURN. Data Hash w/ Sig achieves high comparable data efficiency whilst conforming to the bitcoin standard.

[1] P2PKH transaction size: 204 byte = 107 byte (P2PKH input script) + 36 byte (referenced output) + 1 byte (input script length) + 25 byte (P2PKH output script) + 1 byte (output script length) + 8 byte (output value) + 26 byte (max transaction metadata)

[2] OP_RETURN output size: data size + 18 byte = data size + 17 byte (max output metadata) + 1 (OP_RETURN script operator code).

[3] Data Hash w/ Sig inputs and outputs are of size: metadata byte + input script byte + output script byte. Where metadata byte = ROOF(data size / 1461) * (17 (max output metadata) + 49 (max input metadata)), and input script byte = data size + (ROOF(data size / 1461) - 1) * 189 (input script metadata) + 111 (input script metadata) + ROOF((data size % 1461) / 520) * 26 (input script metadata), and output script byte = ROOF(data size / 1461) * 22 byte

Data Size (byte)	**OP_R**	**var OP_R**	**max OP_R**	**Data Hash w/ Sig**
2	1%	1%	1%	0%
4	2%	2%	2%	1%
8	3%	3%	3%	2%
16	7%	7%	7%	4%
32	13%	13%	13%	7%
64	22%	22%	22%	13%
80	**26%**	**26%**	**26%**	**16%**
128		35%	37%	23%
256		48%	54%	37%
512		61%	70%	54%
1'024		70%	82%	69%
2'048		75%	90%	74%
4'096		**78%**	**95%**	**80%**

Table A.1: Data efficiency as percentage versus data size for various data insertion methods. OP_RETURN is abbreviated as OP_R.

Appendix B

Broadcast Properties

There are many ways in which broadcast properties can be realised. This is a summary of how the discussed properties can be realised, using blockchain and non-blockchain techniques, with the techniques employed in our work highlighted, followed by a discussion on the definition of *confirmed transactions*:

- *Validity:* Resubmitting rejected transactions to the blockchain until confirmed [28]. Retransmit forever, using fairness property [1, ch. 2.4] (our work).
- *No duplication:* Waiting for blockchain confirmation before delivering a message [28, 30]. Detects and invalidates stale transactions by the use of consensus hashes [21]. Keep a record of all messages that have been delivered in the past, when a message is received, it is delivered only if it is not a duplicate [1, ch. 2.4] (our work).
- *No creation:* Private key holder is the only one able to sign for a message with the corresponding public key [4] (our work). That is, if each process has a unique pubkey, then the sender can be verified by the message signature.
- *Agreement:* Waiting for blockchain confirmation before delivering a message [28, 30]. Every correct process continually resubmits every message it delivers (using the validity property) [1, ch. 3.3] (our work).
- *FIFO-order:* [28] enforces a per-file FIFO-order. This is achieved by waiting for submitting the following transaction before the previous one has been confirmed on the blockchain. The FIFO property can also be achieved by assigning sequence numbers to messages, and delivering the messages in order of the sequence numbers [1, ch. 3.9]. Link two consecutive transactions, by using the output of the first transaction as the input of the subsequent transaction (our work).
- *Causal-order:* This can be achieved by using vector clocks, i.e. appending a vector of timestamps to the broadcast messages (waiting, smaller message size), or, by appending entire causal past to the message (no waiting, large message size) (our work) [1, ch. 3.9]. The causal-order is implied by the combination of FIFO-order and total-order, as is used in one of our protocols (our work).

- *Uniform agreement:* Deliver only confirmed transactions [28, 30] (our work). Uniform agreement cannot be achieved if the number of participants is unbounded (because of equivalence of uniform reliable broadcast and consensus) [15]. The solution to this is to either assume the bounded fork length. Another solution is to limit the protocol to a bounded set of participants (permissioned) (our work). If the set of participants is bounded in number, quorum and majority based protocols can be used, such as [1, ch. 3.4].

- *Uniform total-order:* Deliver only confirmed transactions [28, 30] (our work). Confirmed transactions are in a total-order, thus, delivering messages in the same order as the confirmed transactions achieves a total-order. Similarly to uniform agreement, in order to achieve uniform total-order, we must either bound the number of participants, or assume bounded fork-length.

- *High throughput / message size / low latency / group membership:* High throughput is achieved by grouping many messages together, calculating the Merkle tree root hash thereof, and write the Merkle root hash to the blockchain [25, 26, 27] (our work). This method is compatible with arbitrary message size (our work). Low latency is achieved by providing an asynchronous write, that is, buffering the incoming messages [28, 25, 26, 29], and providing functionality to retrieve the proof that it was written to the blockchain at a later point (our work). This might require direct or P2P communication protocols between the nodes, to exchange information that is not written to the blockchain [25]. The performance, unless using such means, will rely part on the underlying blockchain. The transactions can also be grouped, e.g. [25] uses chainIDs, to group transactions into disjoint chains. In our work, we simply filter messages, such that only messages from group members are let through.

Confirmed Transactions Many of the discussed approaches rely on the concept of *confirmed transactions* or *final transactions*. Confirmed transactions can be described as transactions that are stored immutably and not to be reordered, i.e finalised. Because of the probabilistic nature of blockchain, we can only consider transactions as confirmed or final under the assumption of bounded fork length, i.e. that there are no forks over a certain length (in reality, no transaction is ever fully finalised on the Bitcoin blockchain). As an example, transactions are confirmed after: six blocks on the Bitcoin blockchain [38]; ten blocks on the Bitcoin blockchain [30]; and seven blocks on the Ethereum blockchain [28].

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den May 22, 2020 .....