October 21, 2019

MASTER THESIS

Guaranteed-TX

The exploration of a guaranteed cross-shard transaction execution protocol for Ethereum 2.0.

Author:
Sjoerd Wels

Master of Computer Science
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Software Technology specialisation

Supervisors:
Dr. Maarten Everts
Dr. Ansgar Fehnker
Gert Jan Timmerman (Info-Support B.V.)

UNIVERSITY OF TWENTE.

Abstract

A major bottleneck of the state-of-the-art distributed ledgers is the limited transaction throughput. Existing ledgers lack scalability and are unable to process transactions at the speed of centralised systems. One technique used by Ethereum 2.0 to overcome these performance and scalability limitations is sharding. With sharding, the computational work is partitioned among multiple, smaller groups of validators referred to as shards. These shards operate in parallel to increase the overall transaction throughput while minimising the communication, computation and storage requirements per node. However, the interoperability between shards is yet very limited. It can take several minutes before a shard is allowed to process a cross-shard transaction while there is no guarantee that a cross-shard transaction will be processed at all. Without interoperability improvements, sharding will only benefit smart contract applications that run within a single shard.

In this study we propose Guaranteed-TX, a guaranteed cross-shard transaction execution protocol for Ethereum 2.0. Guaranteed-TX allows shards to process cross-shard transactions before being finalised in the block it was created - a property called optimistic execution - which significantly improves cross-shard transaction latencies. In addition, it provides economic guarantees that all cross-shard transaction will eventually be processed. In order to achieve both Guaranteed-TX introduces a messaging layer which records the created and processed cross-shard transactions and is shared with every shard. The messaging layer is used to finalise consistent blocks and punish validators in a shard committee for not processing cross-shard transactions. Consequently, cross-shard transactions are either processed or slowly drain the stake of the validators within the addressed shard.

Although we prove correctness of Guaranteed-TX and show that it achieves theoretical satisfying performance, the protocol is build upon assumptions which yet have to been proven. The first phase of the Ethereum 2.0 transition is set for launch on January 3, 2020. However, it may take several years before cross-shard communication will be facilitated. In the meantime, many specifications may change and empirical evidence may confirm or invalidate our assumptions. Our findings show that a guaranteed cross-shard transaction protocol is feasible and the insights of this study should be used for further protocol developments.

Contents

1 Introduction **7**
- 1.1 Research questions . . . 8
- 1.2 Approach . . . 9
- 1.3 Structure of the report . . . 9
- 1.4 Contributions . . . 9

I State-of-the-Art **11**

2 Preliminaries **13**
- 2.1 Distributed systems . . . 13
- 2.2 Cryptographic techniques . . . 15

3 Consensus **19**
- 3.1 Consensus fundamentals . . . 19
- 3.2 Traditional BFT . . . 20
- 3.3 Nakamoto consensus . . . 24
- 3.4 Detailed comparison . . . 32

4 Sharding **35**
- 4.1 Sharding fundamentals . . . 35
- 4.2 State validity . . . 41
- 4.3 Data availability . . . 43
- 4.4 Smart contracts on a sharded architecture . . . 44
- 4.5 Related work . . . 46

II Solution **49**

5 System overview **51**
- 5.1 Design Goals . . . 51
- 5.2 System Model . . . 51

6 Guaranteed transaction execution **53**
- 6.1 Transactions in a non-sharded ledger . . . 53
- 6.2 Transactions in a sharded ledger . . . 53
- 6.3 Separating transactions . . . 54
- 6.4 Rethinking Ethereum's gas mechanism . . . 55

7 Guaranteed-TX **57**
- 7.1 Overview of Guaranteed-TX . . . 57
- 7.2 High-level design . . . 58
- 7.3 Design components . . . 62

8 Security and Performance Analysis **69**
8.1 Correctness . . . 69
8.2 Performance . . . 71
8.3 Overhead . . . 73
8.4 Known limitations . . . 75

9 Sharding model modifications **77**
9.1 Different sharding approaches . . . 77
9.2 Ethereum modifications . . . 78
9.3 Discussion . . . 80

III In closing **81**

10 Conclusion **83**
10.1 Suggestions for further research . . . 83

References **88**

Glossary **89**

Acronyms **91**

Appendices **93**

A Guaranteed-TX Simulator **95**
A.1 Implementation . . . 95
A.2 Compiling the repository . . . 95
A.3 Simulation abstractions . . . 95
A.4 Usage . . . 96

Chapter 1

Introduction

While centralisation has been a major competitive advantage for many years, in recent years more attention is being paid to its negative aspects; mounting inequality [59, 63], political corruption [28], increasing monopolization [49]. The numerous scandals of highly centralised organisations have provoked populist backlash [17, 34]. The demand towards decentralisation, in which transparency and control is given back to its users, is growing. As a result, Distributed Ledger Technology (DLT), such as Bitcoin and Ethereum, has gained popularity.

A DLT, is essentially, a replicated database in the form of an immutable append-only ledger that evolves over time by applying transactions. The transactions are verified by each participant and a consensus algorithm is used to maintain a synchronised state between them. Each transaction is authenticated by the digital signatures of the involved parties. Consequently, providing an architecture that eliminates the need of a third-party and minimises the trust in a single participant.

However, the intense verification and public nature of DLT limits its potential. Existing platforms lack scalability and are unable to process transactions at the speed of centralised systems. For example, Bitcoin roughly supports 3.8 transactions per second compared to the 1700 processed by Visa [7, 18]. Moreover, the limited transaction throughput results in high transaction fees in times of congestion. At the same time, the size of the ledger is constantly growing. The complete verified transaction history of the Ethereum Network is now about 190 gigabytes [26]. The speed at which a node can sync the entire transaction history together with the required storage capabilities could possibly lead to a more centralised network.

A promising technique to address these issues is to partition the computational resources and/or data storage among multiple groups of participants. This technique is often referred to as *sharding* and is already used in the context of traditional databases. However, sharding has wide-ranging implications, among other things, on data availability, message complexity, and overall security. Finding the right balance between *decentralisation*, *scalability* and *security* is the key challenge. This problem is often referred to as the Scalability Trilemma, outlined by V. Buterin [55].

Earlier work on sharding solutions focussed on transaction sharding such that transactions could be processed in parallel [33, 36, 66]. These solutions, however, only address half of the problem. Each node still has to download and synchronise the state of the entire ledger. As a result, the increased transaction rate which goes along with a growing storage requirement would eventually lead to a storage problem. In more recent work [64], a full sharding protocol was proposed. Full sharding solution shard in terms of communication, computation and storage. However, their work is only applicable to payment transactions. Unlike smart contract transactions, payment transactions are small and can easily be partitioned into 'credit' and 'debit' operations. Full sharding solutions for smart contract platforms have additional challenges.

Ethereum, the leading smart contract platform, is actively working on a major update named Serenity which will introduce sharding. The update will be rolled out in multiple phases and the first phase is set for launch on January 3, 2020 [58]. The first phase is concerned with the construction of the beacon chain that will manage the Casper Proof of Stake protocol. The second phase is concerned with the construction, validity and consensus of the shard chains. Finally, once these phases are completed, the functionality comes together in a successive phase introducing an EWASM Virtual Machine and facilitating cross-sharding.

Up to now, very little has been set in stone regarding the distribution of smart contracts across

shards and the processing of cross-shard transactions. According to V. Buterin, the primary focus is to bring scalability and subsequently one could look into 'better' but probably more complex sharding schemes. However, the current specification poses some serious drawbacks:

- **No dynamic-balancing:** Users choose the shard at which a smart contract will be deployed. There is no balancing scheme to reallocate smart contracts and optimise the overall performance. Shards have independent *gas markets*, such that it is likely that new smart contracts are deployed on the 'cheapest' shard.
- **Slow cross-shard transactions:** Cross-shard transaction need to be finalised in the source shard before the target shard is allowed to process it. Finalisation occurs at Epoch boundaries and the time between two Epochs is currently 6.4 minutes. As a result, cross-shard transactions incur high latencies.
- **No guaranteed execution:** A cross-shard transaction is a receipt created on the source shard addressed to a target shard. However, there is no guarantee that a cross-shard transaction eventually will be processed.

One can observe that the above mentioned problems are tightly coupled. If we cannot guarantee that a cross-shard transaction will be processed eventually, we can neither guarantee that it will be processed within some upper time limit. However, we need low-latency cross-shard transactions in order to make balancing smart contracts schemes practical because slow cross-shard transactions have a negative impact on the usability of smart contracts.

In this research, we explore the problems of cross-shard transactions. We propose *Guaranteed-TX*, a guaranteed cross-shard transaction execution protocol for Ethereum 2.0. Guaranteed-TX not only ensures that all cross-shard transactions will be processed eventually, but also provides economic guarantees that a cross-shard transaction will be processed within some upper time limit. In addition, we provide several useful starting points to develop a sharding scheme in which smart contracts efficiently can execute atomic cross-shard operations.

1.1 Research questions

The aim of this study is to design a guaranteed cross-shard transaction execution protocol for Ethereum 2.0. With such scheme, one can implement almost every concurrency protocol (Lock-based, Two Phase, Validation based) to improve shard interoperability without having conflicting shards. In this research, we focus on the following questions:

Q1 How can we facilitate guaranteed cross-shard transaction execution in a sharded distributed ledger?

Q2 With the approach, found in Q1, can we design a sharding scheme that enables fast cross-shard transactions and smart contract balancing?

In traditional distributed databases guaranteed execution is typically taken for granted. The database is partitioned across separate nodes which are still maintained by the same organisation. As long as cross-shard messages are successfully delivered, they will be executed. In distributed ledgers, however, the shards are not necessarily cooperative. They face different challenges and have other requirements. We have to identify what is expected from a guaranteed execution protocol and subsequently identify which techniques could be used to ensure guaranteed execution. In order to answer the first research question we will focus on the following sub questions:

Q1.1 What would users expect from a guaranteed execution protocol?

Q1.2 Which techniques could be used to enable guaranteed execution?

The second part of this research focusses on the partitioning of smart contracts across shards and how to optimise the overall performance of the parallel execution of smart-contract transactions. To answer the second research question, we will focus on the following sub questions:

Q2.1 What problems arise with the parallel execution of smart contracts in a full sharded distributed ledger?

Q2.2 Which concurrent programming techniques are best suited to address these problems?

The findings of the above research questions led us to develop a guaranteed cross-shard transaction execution protocol named *Guaranteed-TX*. In addition, we propose a few improvements with regard to the sharding model of Ethereum 2.0.

1.2 Approach

To date, there are many projects working on full sharding solutions but most of them are yet at an early stage of development. To the best of our knowledge, there is no fully operational network that is both secure and supports cross-shard transactions. For this reasons, we used a pragmatic approach.

In order to get a better understanding of the problems arising with full sharding solutions existing literature has been studied on the subjects of sharding, consensus and concurrency models. The studied materials gave us insight in the above mentioned problems and in particular, the impact of protocol properties on other aspects. Based on the analysed literature, the first version of the Guaranteed-TX was designed. We subsequently evaluated the protocol and identified that the protocol incurred high communication cost. We made modifications that significantly decreased the communication costs but traded-off the freedom of processing individual cross-shard transactions by processing batches of cross-shard transactions.

1.3 Structure of the report

This study is divided into three parts. The first part is devoted to present the various concepts relevant to this research and provides state-of-the-art of each topic. Chapter 2 provides some preliminaries that will be used throughout this document. Chapter 3 explains different distributed consensus protocols and its challenges. Then, chapter 4 discusses various sharding approaches, problems and challenges associated with smart-contract sharding.

After establishing their domain, the proposed solution Guaranteed-TX will be discussed. Chapter 5 briefly describes the system model of Ethereum 2.0. Then, chapter 6 reasons about the properties and required modifications of cross-shard transactions and chapter 7 presents Guaranteed-TX and discusses its properties and design components. The protocol is subsequently analysed in chapter 8 to proof correctness and to reason about its performance. During the study, we also identified several improvements with regard to the sharding model of Ethereum 2.0, which will be discussed in chapter 9.

Finally we will evaluate our research and the potential of a guaranteed cross-shard transaction execution protocol. Chapter 10 will discuss our findings, the meaning of those findings, and the suggestions of further research.

1.4 Contributions

To our knowledge, we present the first guaranteed cross-shard transaction execution protocol, that achieves the following properties:

- **Optimistic execution** Optimistic execution is the property of a sharded-based distributed ledger that enables a shard to process cross-shard transactions before these are finalised in the source shard. This implies that processed cross-shard transactions are reverted if the creation is reverted. Consequently, the delays of cross-shard transaction execution are significantly reduced.
- **Guaranteed cross-shard execution** Guaranteed cross-shard execution ensures that cross-shard transactions are eventually executed and ensures that the overall ledger remains in a consistent state.

Part I

State-of-the-Art

Chapter 2

Preliminaries

In this chapter we introduce some preliminaries that will be used throughout the rest of this document. The first section is devoted to the terminology of DLT used in this work. In this section, the basic concepts of, among other things, *distributed*, *decentralised*, and *permisionless* systems are explained. The second section introduces some novel cryptographic techniques, such as *Merkle trees* and *verifiable computations*. Even though we will not discuss implementation details, this section provides a general and basic understanding required in this study.

2.1 Distributed systems

DLT is a vast and complex subject. The terminology used to describe this domain poses many challenges. Understanding the variants and technology in depth is proven to be problematic when terms are misleading or out of context. Even with the current effort to establish an ISO standard, the gap between epistemic communities with their own formed ideas about DLT and the industry is hard to bridge [31]. In this research, we use the following definitions:

Distributed systems

Definition 2.1 (Distributed System)**.** A system is said to be distributed if hardware nodes or software components, located at networked computers, communicate and coordinate their actions only by passing messages.

The field of *distributed computing* has been studied extensively for many decades. In distributed computing, components interact with one another in order to achieve a common goal. These components could be physically close or geographically distant and are connected by a wide area network. The term *distributed* refers to the partitioning of computing tasks over multiple networked connected components regardless of the implementation used to reach this goal. An particular aspect of distributed systems is that, the message transmission delay is not negligible [39].

Peer-to-peer computing

Definition 2.2 (Peer-to-peer computing)**.** Peer-to-peer computing is a distributed application architecture in which interconnected peers, i.e. equally privileged nodes, share resources amongst each other without the need for central coordination by servers or stable hosts.

Peer-to-peer computing is one form of distributed computing, in which individual peers make their resources directly available to other neighbouring peers without the need of a centralised server. It requires each peer to self-organise themselves, based on whatever local information is available and through interacting with neighbouring peers. A peer simultaneously acts as both a ‘client’ and ‘server’ and is indistinguishable from other nodes regarding the view of functionality. The global state or behaviour emerges as a result of all local actions.

A peer-to-peer network is said to be *pure*, if any arbitrary chosen peer can be removed from the network without having the network suffering any loss of service. Network paths and data storages are replicated to achieve reliability, recover from failures, and provide satisfactory performance.

Peer-to-peer networks can be classified by the degree to which the network is structured. In unstructured networks, the placement of the shared resources in the network is unrelated to the network topology. Peers have no information about each other's resources and probe their neighbouring peers for their services. In structured networks, on the other hand, the topology is tightly controlled and resources are managed in specific locations. Peers typically maintain some form of distributed routing table for efficient routing and resource usages. Although unstructured networks are easy to implement and maintain, they lack scalability. As the number of participants increases, the number of message exchanged increases.

Fault tolerance

Definition 2.3 (Fault tolerance)**.** Fault tolerance is the property that enables a system to continue to operate properly in the case of a failure of one of its components.

A system is said to be fault-tolerant if it is continues to operate, possibly at a reduced level of performance, in the event of failures. Fault tolerance is typically achieved through redundancy or replication, i.e. providing functional capabilities that are unnecessary in a fault-free environment. With a redundancy approach, multiple instances of the same subsystem are initiated and one switches to one of the remaining instances in case of a failure. With a replicate approach, the multiple instances operate in parallel and the correct result is chosen on a basis of a quorum.

A particular type of fault tolerance is Byzantine fault tolerance (BFT). A Byzantine fault is any fault presenting different symptoms to different observers. With BFT, there is imperfect information on whether a component has failed, which means that a failed node can generate arbitrary data, pretending to be a correct one. Consequently, a system is said to be BFT if it is fault-tolerant to such condition.

State machine replication

State machine replication is an approach to implement fault tolerance by replicating nodes and coordinate their transitions. The idea is to replicate a service over multiple nodes to ensure availability in the case of a replica failure. To ensure consistency, each replica needs to apply the same sequence of commands, typically using a consensus protocol. An implementation of state machine replication requires the following properties to hold:

- **Initial state:** All correct replicas start on the same state.
- **Determinism:** All correct replicas receiving the same input on the same state produce the same output and resulting state.
- **Coordination:** All correct replicas process the same sequence of commands.

Replicated logs

A replicated log provides an append-only storage of 'log' entries that is replicated over multiple nodes. Each 'log' entry can contain arbitrary data. Replicated logs can provide fault tolerance for sequential growing data structures to ensure that nodes will remain consistent with one another. For example, to build a state machine replication system. We refer to the nth entry in the list as the *height* of the entry.

(De)centralisation

(De)centralisation is the process of redistributing or dispersing functions, powers, or decision making away from or to a central location or authority. With DLT, however, the term has a double meaning. V. Buterin well described the three types of decentralisation: [12].

- **Architectural (de)centralisation:** How many physical nodes do form the network?
- **Political (de)centralisation:** How many individuals or organizations participate in the network?
- **Logical (de)centralisation:** Does the system presents it self as a monolithic object or amorphous computing?

Architectural (de)centralisation refers to the level of decentralisation in the system architecture. In a centralised system, decisions are made from a centralised *physical* location and transmitted to executive components. While these systems benefit from a clear chain of command and being more efficient, they contain a single point of failure and centralised attack surface. By distributing the decision making over many separate components, eliminating the single point of failure, the system becomes fault tolerant and attack resistance.

Political (de)centralisation refers to the number of individuals / organisations controlling these physical nodes. If all the components of an architectural decentralised system are controlled by a single organisation, one is still relying on the integrity of that organisation. By redistributing the power over many independent individuals, the likeliness of these individuals colluding to act in ways that benefit them at the expense of others, reduces.

Logical (de)centralisation refers to the centralisation of the service the system provides. The service that a logical centralised system provides is based on a global centralised state. Partitioning the system into multiple subsystems is therefore not possible. A logically decentralised system, on the other hand, can endure partitioning of the service and continue to operate as independent systems.

Distributed Ledger Technology

DLT is the term used to describe the family of technologies derived from or built to support distributed ledgers. Although there are many variants with different properties, a distributed ledger is essentially a digital system of replicated state machines recording transactions over a peer-to-peer network and consensus is reached on the order of transactions.

One type of ledger, which groups transactions into blocks, is known as a blockchain. Every block contains a hash of the previous block, forming a chain from the genesis block to the current block. Once a transaction is included in a block, it is verified to be valid. Blockchains are well suited to serve as general-purpose computation ledgers, in which code uploaded by users is executed in a virtual machine in the form of a *smart contracts*. The ledger is used as a global computer.

A distributed ledger can be either *permissioned* or *permissionless*. Permissionless ledgers are open to any participant and typically comprises many participants with high churn rate. Permissioned ledgers, on the other hand, have evolved to address the need for DLT among a set of known and identifiable participants.

Most permissionless ledgers function with some underlying token as part of the ecosystem of the distributed platform. The token is used as an internal currency and contributes to the stability of the platform. The concept of using some digital value to provide guarantees that a system will operate in a particular way is known as *cryptoeconomics*.

2.2 Cryptographic techniques

Distributed ledgers are built with a range of cryptographic concepts. This section lists the ones used in this study.

Hashing

A hash function is a function that maps data of arbitrary size onto data of a fixed size. The output of the function is called the hash or hash value. A specific hash function is a cryptographic hash function which make it practically infeasible to invert the hash to some message; the hash function is one way.

A good cryptographic hash function satisfies the following properties:

- **Pre-image resistance** Given a hash value h it should be difficult to find any message m such that $h = hash(m)$.
- **Second pre-image resistance** Given an input m_1, it should be difficult to find another input m_2 such that $hash(m1) = hash(m2)$.
- **Collision resistance** It should be difficult to find two different messages m_1 and m_2 such that $hash(m1) = hash(m2)$.

Throughout this document, whenever we use the term *hash* we assume a cryptographic hash function.

Public-key cryptography

Public-key cryptography is an asymmetric cryptographic system that uses pairs of keys; *public keys* that are disseminated to the public and *private keys* which are only known by the owner of the key. In such asymmetric key encryption, anyone can encrypt some secret using a public key, but only the owner of the related private key can decrypt the generated ciphertext, i.e. the result of encryption performed on the secret. The security is based on the secrecy of the private keys.

In addition to confidentiality, public-key cryptography could also be used to create a digital signature and guarantee that a message or document was not altered during transmit. The holder of the private key first creates a hash h of the to be signed content, i.e. the message digest, which it subsequently encrypts with its private key to create a digital signature. The message combined with the signature is send to the addressee which then can verify that the message was not altered through verifying that the signature belongs to the sender using the public key of the sender. Because the private key is only known by its sender, the sender cannot deny sending the message.

Digital signatures provide the following properties:

- **Integrity** The message cannot be altered or tampered with without being detected.
- **Authentication** The identity of the sender can successfully be confirmed.
- **Non-repudiation** The origin of the signed content is verified and the sender can not later falsely deny it.

Merkle Trees

A Merkle tree is a tree in which every leaf node is labelled with the hash of a data block and every non-leaf is labelled with the hash of its child nodes. Merkle trees or hash trees allow for efficient and secure verification of content being part of some large set of data. The verification of a data block being part of the set only requires to compute the number of hashes proportional to the logarithm of the number of leaf nodes of the tree. The *top hash* or *root hash* is typically required from some trusted source.

In DLT, Merkle trees simplify the validation of data for node types referred to as *light clients*. Light clients, in contrast to full nodes, only keep track of block headers. Data of the block body is mapped to a Merkle Tree which *top hash* is stored in the block header. In order to validate that some piece of data is part of the block, light clients acquire the related hashes in the Merkle Tree from a full node to compute the top hash and verify that the top hash equals the one in the block header.

Figure 2.1 shows an example of such Merkle tree. Assume a blockchain in which the top hash of the merkle tree of its transactions is known. In order to validate that transaction 2 is part of the tree, a user only needs to acquire hash_{00} and hash_{1}. Subsequently, the node can first calculate the hash of transaction 2, then calculate hash_{0} and subsequently the top hash. If the top hash equals the transaction Markle tree root, then transaction 2 is part of the tree.

Verifiable Computation

Verifiable computation enables a system to offload the computation of some function to other, perhaps untrusted, systems while maintaining verifiable results. Verifiable computation is useful for (i) devices which can not compute the computation themselves, or (ii) for those who want to have some guarantee that the outsourced computation is correct. In particular for the latter, it is desired that the cost of the verification process is significantly lower than the computation itself. Otherwise, it would be more beneficial to execute the computation locally.

A simple verifiable computation scheme may look like the following: a *verifier* sends the specification for some computation f, e.g. some executable code, and some input x to a *prover*. The *prover* computes output $y = f(x)$ and returns the result. Then, the *prover* should be able to convince the *verifier* that the output is correct by either answering some questions from the verifier or providing some proof that the verifier could validate.

Verifiable computation has great potential in DLT. In particular, if the cost of the verification is low. But in spite of the drastically reduced verification process cost throughout the last few years, the overhead to produce verifiable proofs of current research prototypes remains largely impractical.

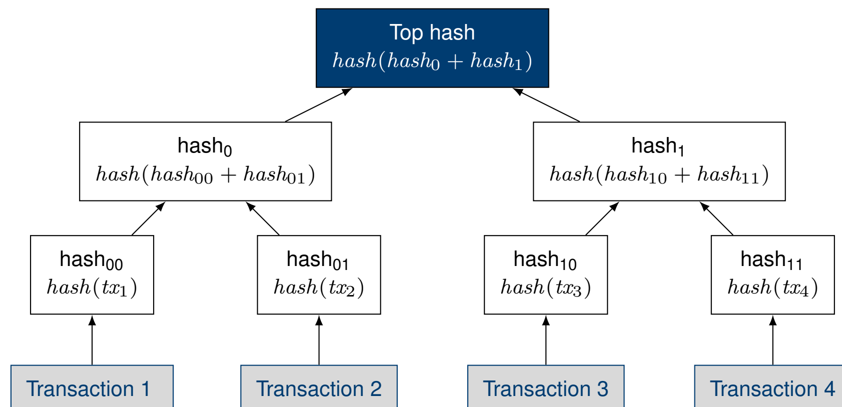


Figure 2.1: An example Merkle tree. Hash$_{00}$ and hash$_{01}$ are the hash values of transaction 1 and transaction 2. Hash$_0$ is the hash of the concatenation of hash$_{00}$ and hash$_{01}$. Then, the top hash is the concatenation of hash$_0$ and hash$_1$, respectively.

Threshold Signatures

A threshold cryptosystem is a cryptosystem that protects information by encrypting or signing messages and distributing it among a set of fault-tolerant replications. In a (k, n)-threshold signature scheme, there is a single public key that is known by all replicas and all n replicas hold a distinct private key. The i-th replica can use its private key to generate a partial signature p_i on message m. The partial signatures $\{p_i\}_{i \in I}$, with $|I| = k$ and $p_i \leftarrow tsign_i(m)$ can be combined to generate σ. Then, there is a function $tverify(m, \sigma)$ which returns true if the number of partial signature is $> k$, otherwise false.

With a threshold signature scheme, a signature can be created among a distributed system of which f are faulty. The signature schema has two major benefits. First, most of the calculations, i.e. the signing of the partial signatures, occurs at each of the replicas locally. Secondly, the combined signature σ is of fixed size, rather than having a message including a list signed signatures of each replica.

Chapter 3

Consensus

A fundamental problem in distributed systems is to achieve agreement on a value or action in the presence of faulty processes. Protocols that solve this problem are called *consensus protocols* and are typically designed to deal with a limited number of faulty processes. This chapter introduces the fundamentals of consensus protocols and in particular emphasises the differences between traditional consensus models and consensus models used in decentralised systems.

3.1 Consensus fundamentals

The consensus problem is typically defined for a set of n known processes. A protocol that solves the consensus problem is said to be t-resilient if it guarantees consensus amongst n processes of which at most t fails. Such a protocol tolerating faulty processes must satisfy the following properties:

- **Agreement:** All non-faulty processes agree on the same value.
- **Integrity:** If all non-faulty processes proposed the same value v, then any non-faulty process decides v.
- **Termination:** Eventually, every non-faulty process decides some value.

The first two are *safety* properties, i.e. properties that require that 'something bad will never happen', while the last is a liveness property, i.e. a property that states 'something good' will happen. Some variations on the *integrity* property exists, e.g. a weaker variant states that at least one non-faulty process decides v rather than all non-faulty processes. In literature, the *integrity* property is also known as *validity*.

A process is said to be correct if it follows the protocol until completion, otherwise it is said to be faulty. Traditionally, faulty behaviour was considered *crash/halt* faulty, where processes crashed or failed to deliver messages, or they are considered *Byzantine*, which means that the process behaves in an arbitrary manner. Faulty behaviour is not necessarily intentional or malicious, e.g. it could simply be caused by a bug or network failure.

Theorem 3.1. *The CAP theorem [1] states that any distributed system can have at most two out of the following three properties; availability, consistency and partition tolerance.*

From definition 3.1, it follows that a distributed system in which arbitrary network partitioning is inevitable, one has to choose between *availability* and strong *consistency*. When choosing availability over consistency, the distributed system will provide a response to each request but the response could be inconsistent across non-faulty nodes. To recover from a network partition, conflicting updates require conflict resolution to eliminate any inconsistencies. However, when choosing strong consistency over availability, each non-faulty node remains consistent but requests can not be processed during a network partition. The CAP theorem claims that network partitioning is unavoidable, resulting in a trade-off between availability and consistency.

Theorem 3.2. *The FLP Impossibility result [27] states that in an asynchronous network where messages may be delayed but not lost, there exist no deterministic consensus algorithm that is guaranteed to terminate if at least one process may crash.*

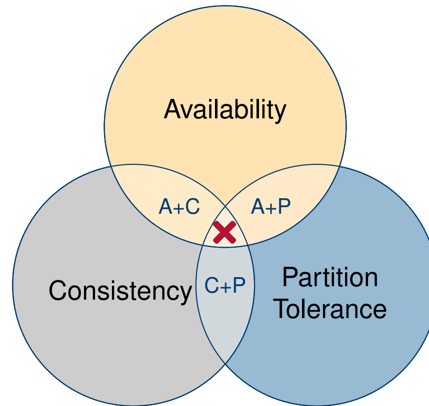


Figure 3.1: The *CAP Theorem* states that it is impossible to achieve all three of strong consistency, availability and partition tolerance at the same time.

The *FLP Impossibility* result, definition 3.2, shows that it is impossible to deterministically distinguish between processes that crashed and processes which take long to process in a fully asynchronous model. Without having an upper bound on message delivery, the consensus liveness property can not be guaranteed.

However, assuming synchrony is the real world is unrealistic. The paper by Dwork, Lynch and Stockmeyer [25] therefore introduced the concept of *partial synchrony*, in which a fixed upper bound on message delivery exists but it is not known a priori. With partial synchrony, asynchronous systems behave like synchronous systems during 'good' times and asynchronous in 'bad' times. Consensus models operating in partial synchrony guarantee liveness during good times.

To this end, a consensus protocol is said to be 'synchronously safe' if its safety is guaranteed by an assumption about timing. Such a protocol assumes that messages are delivered within some fixed, known amount of time. It is called 'asynchronous safe' if no assumptions about timing are made; messages are eventually delivered, but no strict upper bound is known (partial synchrony).

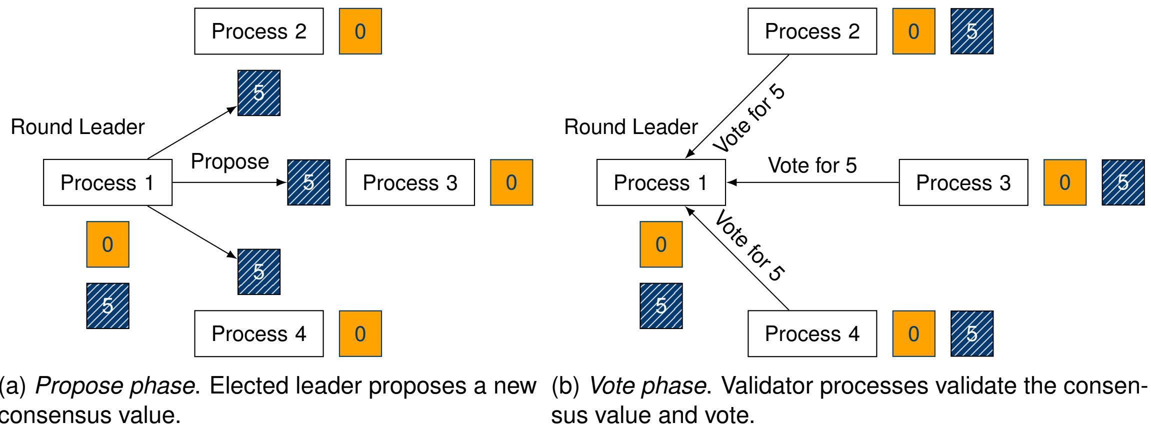
3.2 Traditional BFT

Consensus models have been well studied before the emergence of DLT. In the earlier models, consensus typically involved a leader-based approach in which agreement on a single consensus value is reached among a *static*, *bounded* number of processes. The agreement process on a single value is known as a consensus round and typically involves multiple phases and actor types.

We can simplify such BFT consensus protocol as follows; each round starts with a *propose phase*, in which a leader elected process proposes a new consensus value, e.g. '0' or '1' in case of a binary consensus protocol. The propose phase is followed by a *vote phase*, in which the non-faulty processes receive the proposal, validate it, and vote for it as next accepted consensus value. Processes that validate and vote for the proposal are known as *validators*. After voting, the validators must come to an agreement. If a quorum number of validators vote in favour of the proposal, i.e. a threshold of votes is received, agreement on the consensus value is reached. In case of non-agreement, the process is restarted, typically with a new leader. This phase is known as the *decide phase*. In addition to the *leader* and *validator* processes, we can distinguish *learners*, which do not actively participate in the consensus process but only learn about the decided consensus value. Figure 3.2, visualises this process.

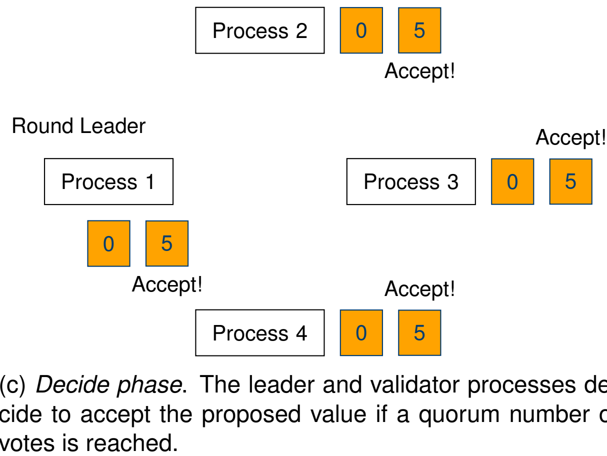
Traditional BFT protocols are typically evaluated on the following aspects: *fault tolerance*, *running time*, and *message complexity*. Fault tolerance is expressed as the type and number of faults the protocol can deal with, running time is expressed as the number of rounds of messages being exchanged and message complexity as the amount of message traffic that is generated by the protocol.

To this end, we classify consensus protocols as *traditional* if a quorum number of nodes need to vote in favour in order to reach consensus on a single consensus value.



(a) *Propose phase*. Elected leader proposes a new consensus value.

(b) *Vote phase*. Validator processes validate the consensus value and vote.



(c) *Decide phase*. The leader and validator processes decide to accept the proposed value if a quorum number of votes is reached.

Figure 3.2: *Traditional BFT consensus*. Every consensus round, a leader is elected to propose a consensus value. The consensus value is accepted when a quorum number of validators vote in favour of the proposed value.

3.2.1 Consensus models

The earlier BFT models targeted high performance while tolerating a few failures and replicas communicated over a LAN. In 1999, M. Castro and B. Liskov published Practical Byzantine Fault Tolerance (pBFT) [13], a BFT algorithm with optimisations to implement the algorithm into-real world systems. This led to a whole new line of research in BFT, e.g. exploiting optimism [37], WAN optimalisations [5] and better fault tolerance [15]. However, most of these algorithms either focus on small clusters or used aggressive batching techniques. More recently, algorithms were designed that are optimised for DLT systems. These algorithms are optimised to work with a group of hundred replicas in a chained execution environment while ensuring low latencies and high throughput.

We discuss pBFT as a base protocol together with the more recent protocols optimised for DLT. The protocols below all tolerate at most f faulty nodes for a set of $n = 3f + 1$ nodes in a partial synchronous network; safety is guaranteed under asynchrony and liveness in times of synchrony.

Practical Byzantine Fault Tolerance

pBFT [13] is a three-phase consensus protocol in which replicas move through a succession of configurations called *views*. In order to decide on some value in a view, a replica is assigned as a *primary*, i.e. leader during the view. Figure 3.3 shows the operation of the protocol in the case of no Byzantine faults. The leader of current view v receives a request message m from a client, it assigns a sequence number n to the requests, and multi-cast the request to the other replicas in the form of a *pre-prepare* message. The replicas confirm the request by multi-casting a *prepare* message for the related pre-prepare message. A replica reaches the *prepared* state if it received $2f$ pre-prepare/prepare messages for message m with sequence number n in view v. In that case, the majority of the non-faulty replicas $(f + 1)$ assigned sequence number s to message m in view v, reaching total order in that view. The replicas then send a *commit* message to reach total order between views. If a replica receives $2f$ commits, the decision is final and the the request is executed.

The sequence number assigned to a request defines the total order of all request and the view number determines the leader. If a leader becomes faulty, a view-change from v to $v + 1$ is required to guarantee liveness. View-changes are triggered by time-outs of replicas waiting for new request to be executed.

pBFT is able to deal with concurrent requests, in which a *window* of open slots, i.e. sequence numbers, are concurrently assigned to requests. This method increases the request throughput. However, with DLT, the validity of a block at height $h + 1$ could depend on block h. Therefore, block $h + 1$ could never be committed before block h.

Although pBFT works well for a small number of known nodes, it is not suitable for DLT. The protocol does not scale well; the message complexity involves $O(n^2)$ messages for a normal round and $O(n^3)$ for a view-change. The protocol further more assumes point-to-point connections between each replica.

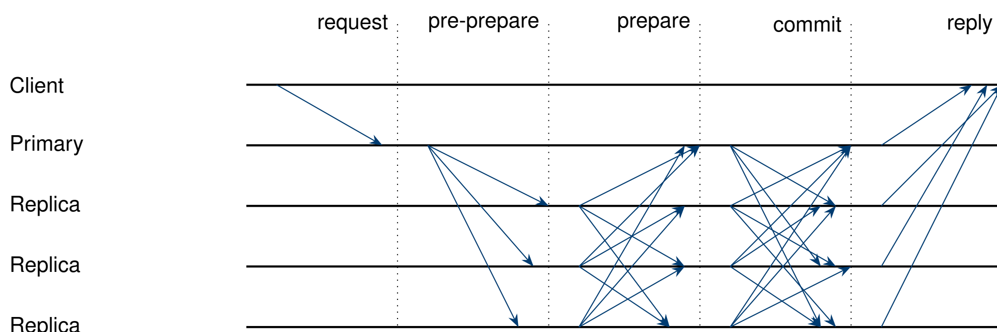


Figure 3.3: pBFT Normal Case Operation.

Tendermint

In 2016, E. Buchman proposed Tendermint; a BFT consensus model optimised for distributed ledgers. Tendermint is a simplified pBFT algorithm for replicated log systems. Rather than using

sequence numbers, Tendermint uses rounds. In each a round a proposer, i.e. round leader, sends a block proposal to all validators, i.e. replicas. The validators validate the proposal and broadcast a signed $prevote(proposal)$ message if the proposal is valid and a $prevote(nil)$ message otherwise. In the latter case the proposal may be valid but the proposal did not arrive in time. Then, the validator waits for pre-votes from the other validators. Once a validator receives $2f + 1$ pre-votes in favour of the related proposal, it will sign and broadcast a $precommit(proposal)$, otherwise broadcast a $precommit(nil)$. After receiving $2f + 1$ pre-commits in favour of the proposal, the proposal is accepted and executed.

Tendermint simplifies the pBFT protocol by removing the view-change complexity. Instead of requesting a view-change, the algorithm relies on time outs for voting phases. For every voting phase, a validator sets a timer. Once the timer expires and the validator did not receive enough valid pre-votes or a valid proposal respectively, it will vote in favour of a nil message and go to the next phase. Consequently, the proposal will not reach a quorum and will not be committed. The validators start a new round with a replaced leader elected in a round-robin fashion. A faulty leader therefore results in $O(n^2)$ complexity, equal to normal round complexity.

Another improvement of the Tendermint consensus algorithm is its *Gossip protocol*. Instead of using point-to-point connections, messages are spread via 'gossip'. A *Gossip* protocol is a peer-to-peer communication protocol based on the way epidemics spread. Nodes periodically spread the latest information they are aware of to a random number of peers they are connected with. Information is propagated between nodes via random gossiping. On average, this reduces the message complexity to $O(n \log n)$. While 'gossiping' reduces message complexity and the number of peers a validator has to connect with, it does not reduce consensus complexity. That is, message dissemination protocols can independently be replaced. For example, pBFT could also use Tendermint's gossip protocol.

SBFT

SBFT [30] is a BFT consensus model that uses optimism to reduce best-case latency to 1 round-trip in a fast track. The primary broadcasts a proposal to all replicas and a decision is reached when $3f + 1$ distinct replicas voted in favour of the proposal. If the fast track fails, the consensus scheme falls back on a common-track which has the identical 2 phases as pBFT and Tendermint.

SBFT uses a threshold signature scheme, in which each participant creates a partial message signature which is send to a collector replica. The collector combines the partial signatures to compute a threshold signature as proof that $2f+1$ validators agreed to the initial message. The combined signature is disseminated to all validators and serves as proof that a quorum voted in favour of some message. The threshold signature scheme reduces the all-to-all message complexity $O(n^2)$ to linear complexity $O(n)$. Figure 3.4 shows the common-track operation of SBFT.

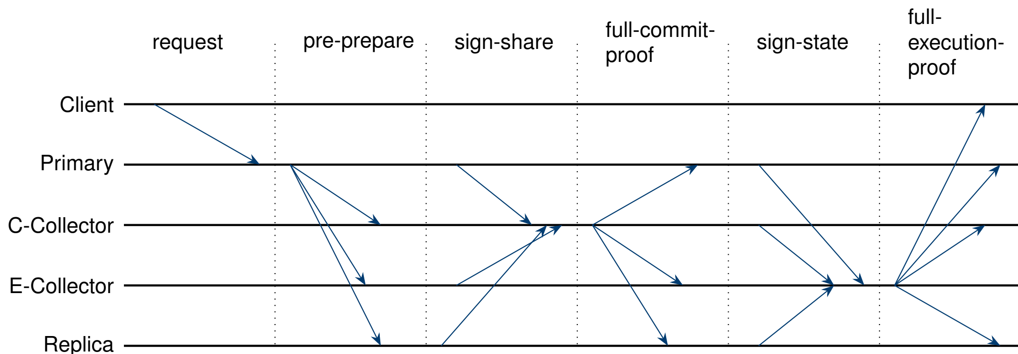


Figure 3.4: Common track SBFT operation.

HotStuff

HotStuff [62] is a consensus model that achieves linear message complexity for both the normal-case and view-change operations. Similar to SBFT, HotStuff uses a threshold signature scheme.

However, in HotStuff the primary carries out the computation of the threshold signature. The computed threshold signature is referred to as a *quorum certificate (QC)*, which is used as proof that a quorum of validators voted in favour during some message proposal phase. Hotstuff also works in a succession of *views*, with each view having a unique leader referred to as the primary. The primary drives the protocol by proposing new consensus values.

Figure 3.5 provides an overview of the normal-case operation. In contrast to SBFT and Tendermint, HotStuff derives around three phases; *prepare*, *pre-commit*, and *commit*. In each phase, the primary collects a quorum of votes to compute its QC. Once the primary moves to the next phase, it includes the QC of the previous phase. The extra phase allows a new leader to simply pick the highest QC it knows, with a small price in latency in return, removing the view-change complexity of *pBFT*.

In order to minimise message complexity, Hotstuff introduced an efficient pipelining protocol. Similar to the concurrent approach of pBFT, pipelining increases throughput. However, pipelining minimises message complexity by aggregating messages of multiple phases. With pipelining, Hotstuff is able to commit one consensus value per round-trip of messages. The pipeline approach works as follows; a primary sends a *prepare* message and waits for a quorum of partial signatures that can be combined to generate a $prepare^{QC}$ for view v. However, instead of sending a $precommit$ message with the $prepare^{QC}$ proof, it sends a new $prepare$ request for the view $v + 1$ including the $prepare^{QC}$ of view v. The $prepare$ message for view $v+1$ thus simultaneously serves as a $precommit$ message for view v. Instead of having a $prepare^{QC}$, $precommit^{QC}$ or $commit^{QC}$ as proof, the primary generates a $generic^{QC}$ which aggregates the proofs of multiple pipelined views. Note that the authors state that pipelining is not unique to HotStuff and could also be applied to Tendermint and pBFT.

Finally, HostStuff has *optimistic responsiveness*, a property defined by the authors themselves. According to the paper, *responsiveness* is the property that enables a correct leader to drive the protocol to consensus at the pace of actual network delay once the network becomes synchronous. Tendemint, for example, is not *responsive*, as a view-change requires the validators to move through the *pre-commit* and *commit* phase before going to the next round. The responsiveness of HotStuff therefore provides speed-ups in cases where validators could not receive enough votes in time.

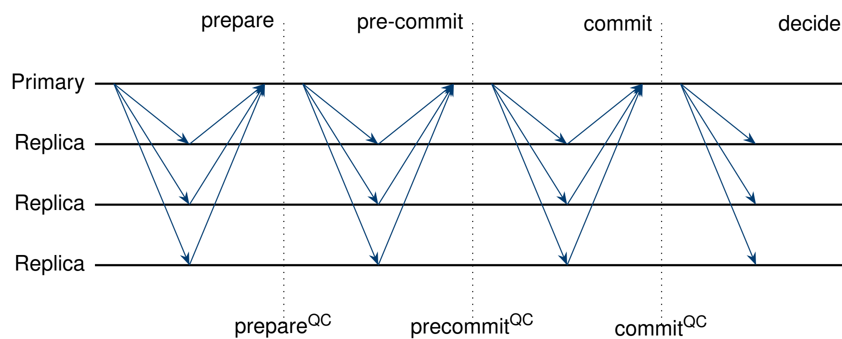


Figure 3.5: Normal-case HotStuff operation.

3.3 Nakamoto consensus

On 31 October 2008, a link to the Bitcoin whitepaper was posted to a cryptography mailing list [41]. The paper discussed a novel consensus mechanism, now known as 'Nakamoto consensus', in which a high-volume number of nodes with high-churn rate and open membership could reach consensus in a decentralised way. This work served as the groundwork for a new type of consensus protocols.

Nakamoto consensus utilises the growing aspect of replicated logs, i.e. the property that in replicated logs only new entries can be added. With Nakamoto consensus, a valid proposal is not only an additional entry, but also a vote in favour for all previous entries. Every additional 'vote', i.e. additional 'log' entry, provides some confirmation that others accepted the previous proposed entries. This minimises the message complexity to $O(n)$.

The protocol uses some *leader election scheme* to limited the number of participants that are

allowed to vote at a specific time. However, it may be possible that multiple participants propose a valid vote at the same time, which implies that there could be multiple entries at height $h + 1$ which extends entry n at height h. Therefore, the nodes agree upon some chain referred as the canonical chain. The rule determining the *canonical chain* out of all candidate chains is known as the fork choice rule.

Nakamoto consensus offers probabilistic finality, which means that the probability that a log entry will be reverted decreases with the number of additional votes, i.e. additional log entries. After x additional votes, a log entry is considered final. In order to guarantee probabilistic finality (and safety in general), the protocol requires nodes to extend the canonical chain, which is economically incentivised.

3.3.1 Properties

We will now discuss the main properties of Nakamoto consensus.

Leader election scheme

Nakamoto consensus is a consensus protocol with unknown and anonymous participants. It potentially supports a large number of validators participating with a high-churn rate. Instead of assigning a single leader, Nakamoto consensus uses a lottery mechanism to decide who is able to generate a valid block at a specific time. This mechanism is sometimes referred to as the 'leader election scheme' or 'Sybil resistance scheme'. In other studies, the authors just use the name of the election leader scheme to refer to the consensus model, for example Proof-of-Work (PoW).

The lottery mechanism ensures that entries are generated at a frequent rate and that the time between two valid generated blocks is greater than the propagation time of a block proposal in the network. If the propagation time is larger, new generated blocks would not extend the latest generated blocks as validators/miners are not aware of the latests blocks. This results in a high fork rate and progress would stagnate. On the other hand, if the time between two blocks is too long, progress would stagnate and result in liveness problems. These timing assumptions make Nakamoto a synchronous protocol.

A leader election mechanism must ensure fairness. We say a consensus model is fair if the number of resources a participants invests in the system is equal to their voting-power capacity. Fairness ensures that every participants that can fairly contribute, which enables decentralisation.

Although there have been many 'leader election schemes' proposed in literature, we highlight the most popular ones:

Proof-of-Work PoW is a lottery election based on a cryptographic puzzle. A participant has to find a nonce such that the hash of a block with (i) a set of transactions o, (ii) the public key i_{pub} of the participant and, (iii) the hash of previous block h, satisfies hardness H. This voting scheme is based on 1-CPU-1-vote principle and the hardness parameter H is tuned such that the average block generation time remains the same. PoW can be formalised as follows:

$$SHA256(h|i_{pub}|o|nonce) < H$$

With PoW, the *canonical chain* serves as proof that it came from the largest pool of CPU power. The awards are some what proportional to the invested effort, i.e. CPU/GPU power. The major drawback of PoW however, is the highly wasted energy consumption. Although the existing numbers are annual estimates, it is calculated that Bitcoin consumes more energy than that is used in the country of Switzerland [14]. Moreover, other studies show that PoW in practise is not as fair as one expected it to be [29].

Proof-of-Elapsed-Time Proof-of-elapsed-Time (PoET) [65] was proposed as an efficient PoW alternative. Instead of spending wasted CPU cycles, each validator runs some trusted piece of code that idles for a randomly determined interval of time. The validator which is first awake, is the assigned consensus round leader. Similar to PoW, PoET is based on a 1-CPU-1-vote principal. The trusted code however, needs to be executed in *trusted hardware*. PoET thus relies on the trusted hardware manufacturer and assumes the hardware can not be hacked.

Proof-of-Stake Proof-of-Stake (PoS) is a lottery election based on invested stake. In permisionless systems, validators have typically deposited some cryptocurrency. The probability of being elected to create a new block is then proportional to the amount of invested stake, based on a 1-coin-1-vote principle.

In contrast to PoW, PoS does not require validators to exhaust computational resources. There is no time-intensive operation required to rewrite a new chain. It is therefore possible to have multiple correct looking chains for a particular ledger making the consensus algorithm *weakly subjective*.

This introduces a new attack type known as a *long range attack*. A malicious adversary could rewrite a long chain in favour of the adversary and convince others that the malicious chain is the main chain in order to execute double-spend attacks. New validators that join the network or validators that have been off-line for a while are in particular vulnerable to long range attacks. One populair way to protect against long-range attacks is to make use of checkpoints, i.e. agree with all validators that some block is part of the canonical chain which can not be reverted any more.

Another issue of PoS is the *nothing at stake* problem. In the event of some fork, whether accidental of malicious, the optimal strategy of the validator is to contribute to both chains such that the validator gets their reward no matter which fork wins. Because the validators are economically incentivised to contribute to both chains, a small group of validators, e.g. 1%, could simply execute a double-spend attack by first mining on both chains and subsequently on the malicious chain. In order to protect against the nothing at stake attack, PoS systems typically implement some *slashing* mechanism. Validators that equivocate will be punished by slashing their deposits.

A requirement of PoS is that the validators have some ‘stake’ of value to deposit. The value of the deposit determines to some extend the security of the system. This could be problematic for new systems because the inherent token does not have a stable value yet. If the value of the coin is initially too low, then an attacker can gain early monopoly power. If the value is to high, then the system may not have enough joining validators causing liveness problems [2].

In contrast to PoW and PoET, the validators in PoS become somewhat known as it requires users to deposit an amount of their stake. Consequently, PoS enables the use of traditional BFT consensus models in permissionless environments.

Fork choice rule

Traditional BFT protocols use a consensus mechanism in which block proposals go from zero confirmations to a level in which a block is fully confirmed and thus final. A successive proposal, can only be confirmed if the previous block is confirmed. This mechanism is called fork-free which means that a block proposal has only one child proposal.

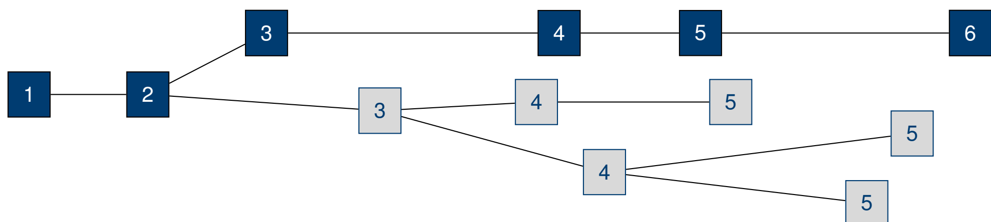
Nakamoto consensus, on the other hand, is a forkful protocol [9]. Forkful protocols allow multiple block proposals to point to the same parent block, introducing forks. The validators have to agree on the same fork to ensure consistency. The rule that deterministically determines the correct fork, i.e. canonical chain, is refereed to as the *fork choice rule*.

Forkful protocols can be visualised as directed trees in which the root of the tree represents the *genesis* block. Each new proposal points to some node in the tree, which allows the tree to grow. Then, the fork choice rule determines the canonical chain and every node in the tree that is not part of the canonical chain is referred to as a stale block.

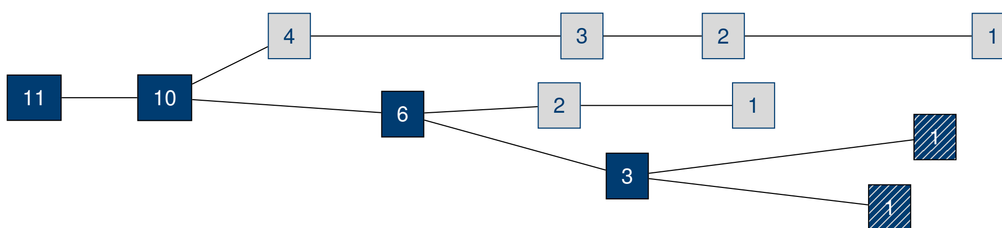
Figure 3.6 visualises three popular fork-choice rules. The longest chain rule, used in Bitcoin, determines the canonical chain by choosing the ‘heaviest’ worked tree, i.e. the longest path from the genesis block to the leaf block. Blocks are ranked by the chain length and the longest chain is considered the canonical chain.

However, the longest chain rule assumes some synchrony. In times of network asynchrony, in which blocks proposals have high-latencies and the overall fork rate is increased, the longest chain is not necessarily the heaviest worked chain. Validators that have produced blocks pointing to the same parent are considered as a single vote, rather than two votes for the same parent block. The Greedy Heaviest-Observed Sub-Tree (GHOST) rule takes this nuance into account by choosing the *heaviest observed* subtree, such that each produced block counts as a single vote. Ethereum 1.0 uses a simplified version of the GHOST rule.

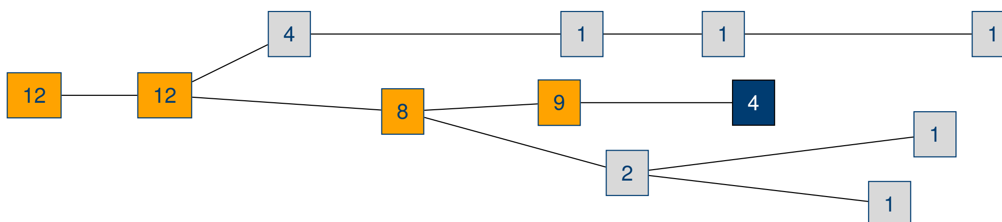
More recently, with the developments of Ethereum 2.0 and the correct-by-construction Casper, a new fork choice rule was introduced that is based on proposal attestations. Similar as before, blocks are produced at regular time intervals, but the consensus model is extended by letting other



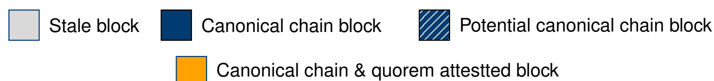
(a) The *longest chain rule*. Blocks are ranked by the chain length and the longest chain is considered the canonical chain.



(b) The *Greedy Heaviest-Observed Sub-Tree rule*. Blocks are ranked by the number of blocks of its subtrees and the heaviest observed subtree is considered part of the canonical chain.



(c) The *Latest-Message-Driven Greedy Heaviest-Observed Sub-Tree rule*. Blocks are ranked by the number of latest received attestations and finalised blocks. A finalised block is a block for which a quorum number of attestations is received. Finalised blocks are always considered part of the canonical chain.



Legend

Figure 3.6: Block tree models determining the canonical chain using different fork choice rules.

validators 'attest' for block proposals by sending attestations in favour of a particular block. This new rule, referred to as *Latest-Message-Driven (LMD) GHOST*, picks the fork with the most attestations. The LMD variant only considers the *latest* attestation of each validator. Alternatively, the Immediate-Message-Driven (IMD) variant considers all attestations. However, IMD does not satisfy the property that if a fork choice rule favours chain b at time t, then the extend in which it favours b should naturally increase over time [8]. This results in instability, as a malicious attacker could switch forks for every attestations.

Moreover, Nakamoto consensus assumes weak synchrony to ensure 'safety' because every validator maintains a block tree locally. A validator independently determines the *canonical chain* based on all block proposals it is aware of. In order for validators to be consistent, they require to have received the same block proposals.

Cryptoeconomics

In collaborative distributed systems in which nodes provide a service without a central authority controlling these nodes, i.e. political decentralised systems, a node may deviate from the protocol due to selfish beneficial behaviour. In 2005, the *B*yzantine-*A*ltruistic-*R*ational model [3] therefore classified nodes in the following categories:

- **Byzantine** nodes may deviate arbitrarily from the suggested protocol for any reason.
- **Altruistic** nodes follow the suggested protocol exactly and reflect the existence of good Samaritans.
- **Rational** nodes are self-interested and seek to maximize their own benefit. Rational nodes deviate from the protocol if and only if doing so increases their net utility from participating in the system.

The innovation of Bitcoin was to assume rational nodes and use economic incentive to create a sustainable system. This resulted in an emerging research area; economic mechanism design, known a *cryptoeconomic* design.

Cryptoeconomics is the use of incentives and cryptography to design new kinds of systems, applications, and networks. These designs rely on economic incentives and punishments. By applying economic theory to design, one can design protocols or mechanisms that produce a certain equilibrium outcome. If the dominant strategy of all nodes is to follow the protocol regardless of what the other nodes do, i.e. the dominant strategy equilibrium, one can provide strong cryptoeconomic security guarantees.

3.3.2 Consensus models

Nakamoto consensus is rather a consensus family than a consensus model on its own. The properties of a particular consensus model heavily depends on the utilised *leader election scheme*, *fork choice rule* and *cryptoeconomic* incentives. We will discuss the implementation details of two popular networks; Bitcoin and Ethereum 2.0.

Bitcoin

Bitcoin, the pioneering work of S. Nakamoto, uses *proof-of-work* and the *longest chain rule*. The hardness parameter is frequently adjusted such that the average block creation time is about 10 minutes. With a block-size of 1 MB, this results in an average transaction rate of 7 transactions per second. Figure 3.7 provides an overview of the Bitcoin network with 4 nodes.

Ethereum 2.0

Ethereum 2.0, also known as the *Serenity* update, is set for launch on January 3, 2020 [58]. The successor of Ethereum brings major improvements with regard to energy efficiency and scalability. The release is scheduled to take several years in seven distinct phases of which the majority are yet in the earliest stage of research [54]. The first phase includes, among others; (i) the Beacon

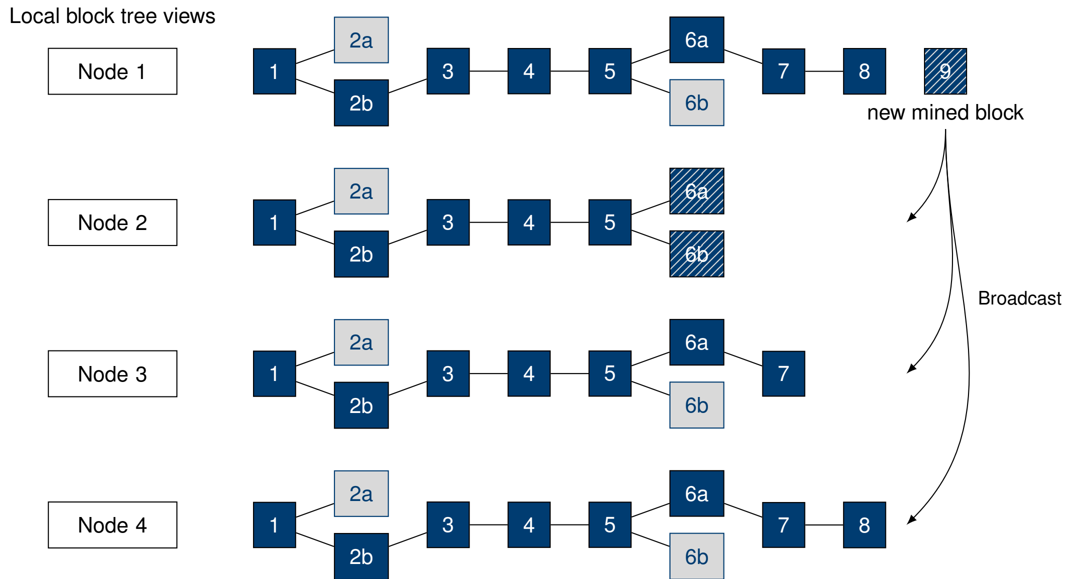


Figure 3.7: *Bitcoin (Nakamoto consensus, Proof-of-Work, longest chain rule)*. Each node maintains a block tree and determines the canonical chain using a fork choice rule locally. Node 1 proposes a new block that, using PoW, satisfies the difficulty parameter. Note that, block trees could differ due to network propagation latencies.

chain, (ii) validator deposits, and a (iii) Proof-of-Stake consensus protocol. Note that, even though the launch is set for January 3, the specification of phase 0 is still work in progress and subject to change. The discussed specification below may change over time.

The new design seamlessly integrates sharding with a proof-of-stake consensus protocol. Although we discuss sharding extensively in the next chapter, this section already discusses some sharding specifics. A major difference from Ethereum 1.0 is the introduction of the beacon chain. The beacon chain, at its core, is a system chain which keeps records of all validators. The validators are randomly sampled into committees. The committees are responsible for the consensus protocol on the beacon chain and its shard chains. In addition to the validator registry, the beacon chain provides cross-links between shard chains and some other basic functionality such as a randomness generator and reporting mechanisms for malicious behaviour. Figure 3.8a provides an overview of the beacon chain and the relation with its shard chains.

To become a validator, a participant deposits a certain amount of ETH into a smart contract on the beacon chain. Initially, the only mechanism to become a validator is to make an one way 32 ETH deposit into a smart contract located on Ethereum 1.0. After depositing, the validator is assigned to a committee. Leaving the validator pool is either voluntarily or forced as a penalty for misbehaviour. Once the validator leaves the validator pool, it is placed upon a withdraw queue to withdraw his deposit. The minimum withdraw time is 18 hours and adjusts dynamically depending on then number of validators withdrawing at that time. The withdraw queue prevents malicious validators to withdraw their stake directly after having acted maliciously.

Serenity splits the block generation period into *epochs* and each epoch being further split into 64 *slots*. At the beginning of each epoch, the validator committees are slowly rotated. Rotating the entire committee at once is inconvenient as the whole new committee requires to download the latest state before the committee can validate new generated blocks. In each slot, an active validator of the committee is assigned to propose a new block; the block proposer. The other validators of the committee, the attesters, are required to attest for the proposed block or attest for a ‘skip-block’.

Figure 3.8b provides an overview of the block proposal mechanism. The current specification defines the slot period of 6 seconds, which results in a epoch duration time of 6.4 minutes. Cross-linking each block of each shard chain is infeasible because it involves many attestations. Therefore, shard chains get cross-linked at the end of each epoch. A cross-link is a set of signatures from a shard committee attesting to a block in the shard chain. Once a cross-link is included in a block in the beacon chain and that block is finalised, then the shard chain of the cross-link is finalised up to the

block the shard committee attested for. Finalising cross-links protects against *long range attacks*. The consensus model used to finalise blocks on the beacon chain is Casper the Friendly Finality Gadget, which will be discussed in section 3.3.2.

In this design, forks only arise due to network latencies and malicious behaviour. The leader election scheme prevents block from being proposed at the same time. However, the fixed time slots in which block producers create new blocks enables a new type of censorship attack. The problem is that a honest validator cannot distinguish between produced blocks which were deliberately delayed by a malicious validator and those that were delayed due to network latencies. A malicious block producer could take advantage of this ignorance and censor blocks of the preceding or succeeding block producer depending on the fork choice rule.

Take for example figure 3.8b in which a block at slot 2 and slot 3 are both pointing to the block at slot 1. If the honest validators would accept the block of the first successive slot pointing to the latest canonical block, i.e. slot 2, then a malicious block producer could deliberately delay the dissemination of his block to censor the produced block of the next slot. On the other hand, if the honest validators accept the latest block pointing to the penultimate canonical block, then a malicious block producer could censor blocks of preceding block producers. In order to prevent both cases, Ethereum 2.0 requires the validators in the shard committee to attest for a proposed block in the same time frame that it was proposed, i.e. the 6 seconds slot time frame. If the block was malicious or not received by an attester, they attest for a 'skip-block'. The validators use the *LMD GHOST* fork choice rule to determine the canonical chain.

The consensus protocol involves many attestations, which can efficiently be aggregated using BLS signatures [32] to reduce message and processing complexity. Note that, the protocol assumes synchronised clocks.

Casper Research

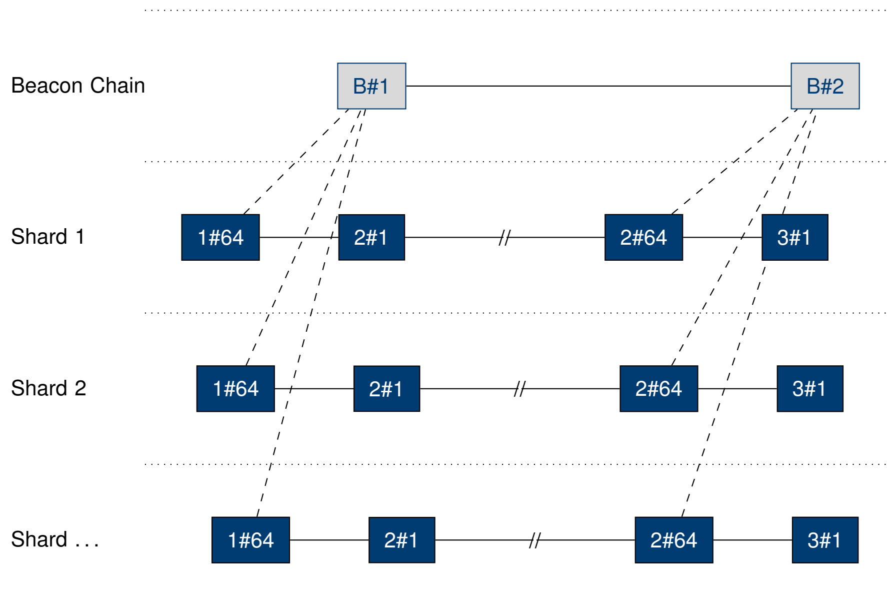
Casper is the code name of Ethereum's PoS consensus protocols. Their work lead two co-developed Casper implementations; (i) Casper Correct-by-Construction (Casper CBC), and (ii) Casper the Friendly Finality Gadget (Casper FFG).

Casper CBC was initially proposed as a protocol for distributed ledgers, but it evolved into a wider field of study comprising a family of PoS models. CBC casper defines a general protocol space, that each successive refinement can build upon. This means that, CBC Casper can be used to derive new protocols which inherit all proofs and guarantees of the framework, but it is by itself not a complete protocol. Correctness of these protocols is thus guaranteed by their construction, hence correct-by-construction.

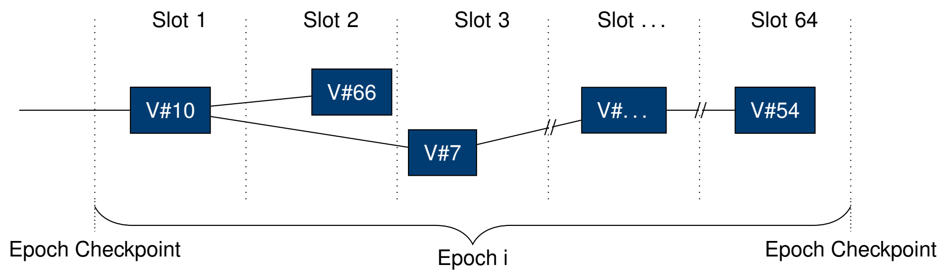
A key property of CBC Casper is that finality is determined client-side. There is no 'threshold' defined in the protocol itself which determines that a consensus value if final. Each client independently can determine at which point a consensus value reaches finality. This means that a client could decide that for some 'low valued' decisions, a majority of 51% is sufficient, while for 'high value' decisions, a majority of 90% is required. Note that, the client-side finality threshold impacts the fault-tolerance threshold.

There are however, some serious notes about liveness guarantees [48]. More importantly, CBC Casper results in high message complexity and requires more research to improve efficiency [11]. Therefore, CBC Casper is to be integrated in the 3rd phase of the Serenity updated and has to replace Casper FFG.

Casper FFG implements a PoS mechanism as an overlay on top of a PoW ledger to provide economic finality, creating a hybrid consensus model. Casper FFG extends the PoW block proposal mechanism by reaching agreement on 'checkpoints' in a traditional BFT approach. The fork choice rule of the PoW protocol is adapted such that finalised blocks are always part of the canonical chain which can not be reverted. This way, validators can produce new blocks without having to wait for many confirmations but still benefit of reaching economic finality with some delay. As previously mentioned, Casper FFG is used to finalise blocks at epoch boundaries in the beacon chain in phase 0.



(a) *The beacon chain and its shard chains.* Validators are registered and managed in the beacon chain. Beacon blocks provide cross-links between shard chains as an infrastructure for asynchronous cross-shard communication. Blocks are labelled by Epoch#SlotNumber and forks are emitted for simplicity.



(b) *Epocs, slots and cross-sharding checkpoints.* Validators are randomly assigned to create a block in a specific slot. A block producers proposes a new block and the other validators attest for that block. If the attesters did not receive a valid block, they attest for a 'skip-block'. Validators use the *LMD GHOST* rule to determine the canonical chain. Blocks are labelled by block proposer validator id.

Figure 3.8: *Ethereum 2.0 (Nakamoto consensus, Proof-of-Stake, LMD GHOST rule)*.

3.3.3 Protocol parameters

The maximum rate at which a distributed ledger can process transactions highly depends on two parameters; *block size* and *block interval*. Tuning these parameters have been widely discussed for PoW ledgers [6, 16], however, the affects of tuning these parameters slightly change for the PoS system proposed by Ethereum 2.0.

Proof-of-Work

The difficulty parameter of PoW ensures that every t minutes a new solution is found. In Bitcoin, the difficulty parameter is adjusted to maintain a block creation time of 10 minutes. That is, the average time needed for a validator to find a correct nonce that satisfies the correct hardness parameter. Decreasing the block creation time t would increase the produced block rate. However, it would also decrease the time δ at which another validator finds a block after time $t + \delta$.

Let us assume a block produced by validator v_1 after time t and a new block produced by validator v_2 after time $t + \delta$. If δ is to small, then validator v_2 will solve the solution before the produced block of validator v_1 is well propagated through the network. As a result, the orphan rate will grow and lead to an unstable block generation process. Moreover, it will lead to an inaccurate evaluation of the difficulty target of target.

The propagation time depends on the block size. Larger blocks require more time to get disseminated across the network than smaller blocks. In addition, nodes require more powerful hardware to process larger blocks. The number of 'full' nodes could decrease due to the increased hardware requirements which will lead to centralisation of the network.

In other words, increasing the block size or decreasing the block generation time for PoW systems not necessarily increases the transaction throughput.

Proof-of-Stake (Ethereum 2.0)

The PoS consensus model of Ethereum 2.0 defines fixed time slots in which blocks are proposed and attested for. The block creation time is fixed and based on the time needed to propagate a block, the time needed to propagate the attestations and probably some additional margin to deal with clock variations. There is no δ to take into account.

In 2013, a research by C. Decker and R. Wattenhofer measured the Bitcoin network latency and determined the it takes on average 12.6 seconds before a block is propagated to 95% of the nodes [19]. The defined time slot interval of 6 seconds in Ethereum 2.0 is significantly less. However, the number of nodes the blocks need to be disseminated to is significantly less. The value of 6 seconds is determined by simulation and has yet to prove itself in practise.

3.4 Detailed comparison

This section provides a comparison of the BFT consensus algorithms and the Nakamoto consensus protocol. First, we compare the state-of-the-art BFT models to select the one which is most suitable in a *permissionless* environment. Subsequently, we compare the advantages and disadvantages of both BFT and Nakamoto consensus.

3.4.1 Comparing traditional BFT models

Table 3.1 provides an overview of the state-of-the-arts BFT consensus models optimised for DLT. Latency is measured as the number of round trips before a consensus value is finalised. Message authenticator complexity, defined by HotStuff [62], is the sum, over all replicas $i \in [n]$, of the number of authenticators received by replica i in the protocol to reach a decision. Message authenticator complexity, unlike message complexity, abstracts unnecessary details about network topology. For example, 1 message holding m authenticators counts the same as m messages holding 1 authenticator. An *authenticator* is either a partial signature or a signature. In the same paper, the authors defined the property *responsiveness*. Responsiveness requires that a non-faulty leader can drive the protocol to consensus in time depending only on the actual message delays. Furthermore, all

	Latency[a]	Authenticator complexity		Responsiveness	Leader Paradigm
		Correct Leader	f Leader Failures		
PBFT	2	$O(n^2)$	$O(n^2)$	Yes	Stable
Tendermint	2	$O(n^2)$	$O(n^2)$	No	Rotating
Tendermint[b]	2	$O(n)$	$O(n^2)$	No	Rotating
Casper	2	$O(n^2)$	$O(n^2)$	No	Rotating
Casper [b]	2	$O(n)$	$O(n^2)$	No	Rotating
SBFT	1[c]	$O(n)$	$O(n^2)$	Yes	Stable
HotStuff	3	$O(n)$	$O(fn)$	Yes	Rotating

[a] Measured in round-trips
[b] Signatures combined using a threshold signature scheme.
[c] Best-case latency for fast-track. Common track is 2 round-trips.

Table 3.1: Comparison of BFT consensus models.

consensus protocols have some mechanism to replace view or round leaders. A *rotating* leader means that the leader is changed every single proposal while a *stable* leader means that the leader is only replaced when a problem is detected.

The first observation is that 'there is no such thing as a free lunch'. There is a trade-off between latency, authenticator complexity and responsiveness. Each consensus protocol comes with its own set of trade-offs. SBFT, for example, has an optimal latency of one round-trip, but in the leader fails it has $O(n^2)$ authenticator complexity. HotStuff, on the other hand has $O(fn)$ authenticator complexity in case of f leader failures, but as a result it takes 3 round trips for a consensus value to be finalised.

Permissionless networks benefits from having *low authenticator complexity*. With a low authenticator complexity more nodes can participate in the consensus model, resulting in a more decentralised network. At the same time, nodes could be off-line or malicious, which could lead to situations where the leader fails or situations in which a quorum of nodes did not vote in time. Therefore, having a consensus protocol with *low authenticator complexity* being *responsive* would optimize the overall performance. HotStuff seems to suit these requirements best.

Furthermore, the *rotating* leader paradigm is preferred above the *stable* leader paradigm to prevent transaction censorship. A stable leader could create valid proposals for over a long period of time, but exclude specific transactions all the time. Detecting censorship is hard because it is hard to proof that the stable leader did receive some transaction but deliberately did not process it. Rotating leaders frequently would provide better censorship resistance.

3.4.2 Comparing traditional BFT and Nakamoto consensus

Nakamoto consensus and traditional BFT both take another position on the consensus 'tradeoff' triangle [53]. While BFT protocols have low latency, they incur high overhead for a high number of nodes. Nakamoto consensus, on the other hand, has high (probalistic) latency, but incur low overhead for a high number of nodes. As a result, the transaction rate decreases as the number of participants increases for BFT protocols, while Nakamoto consensus has a fixed low transaction rate independent of the number of participants. Figure 3.9 visualises the relation between the transaction rate and number of nodes per consensus model.

Traditional BFT models assume *static* and *known* participants. In order to use BFT in *permissionless* setting, one could use PoS. The nodes who deposited stake are the valid participants and the economic incentives discourage participants to deviate from the protocol.

There is another important difference between both. Traditional BFT, unlike Nakamoto consensus, considers off-line nodes to be malicious. Fault-tolerance of traditional BFT protocols assume $2f+1$ nodes to be honest and therefore on-line. If $f+1$ nodes are off-line and all other nodes are honest, traditional BFT will not make any progress. Nakamoto consensus, on the other hand, will experience a decreases performance due to off-line nodes, but could still make progress. Nakamoto consensus tolerates f malicious nodes of the $3f+1$ on-line nodes.

Table 3.2 evaluates the most important properties of both consensus models.

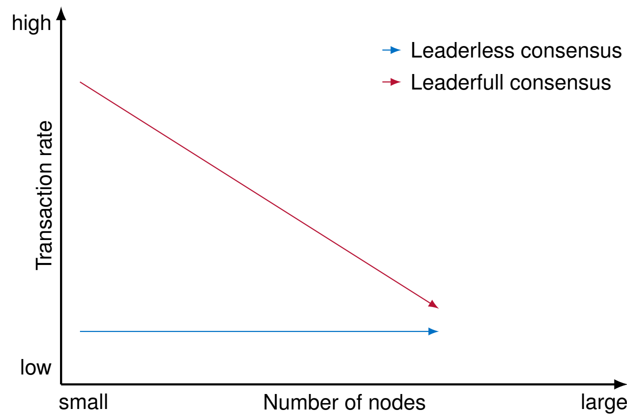


Figure 3.9: Relation between the transaction rate and the number of validators per consensus model.

	BFT	**Nakamoto Consensus**
Participants	< 100	> 100
Authenticator complexity	$\mathcal{O}(n)$ / $\mathcal{O}(fn)$[a]	$\mathcal{O}(n)$
Finality type	Immediate	Probabilistic / economic
Finality time	Fast	Slow
Scalability participants	No	Yes
Scalability throughput	No	No
Throughput	High	Low
Membership model	Permissioned	Permissionless
Network model	Partial synchrony	Synchrony
Adversary model	Static	Adaptive

[a] f Leader failures complexity

Table 3.2: Comparison of BFT (HotStuff) and Nakamoto consensus.

Chapter 4

Sharding

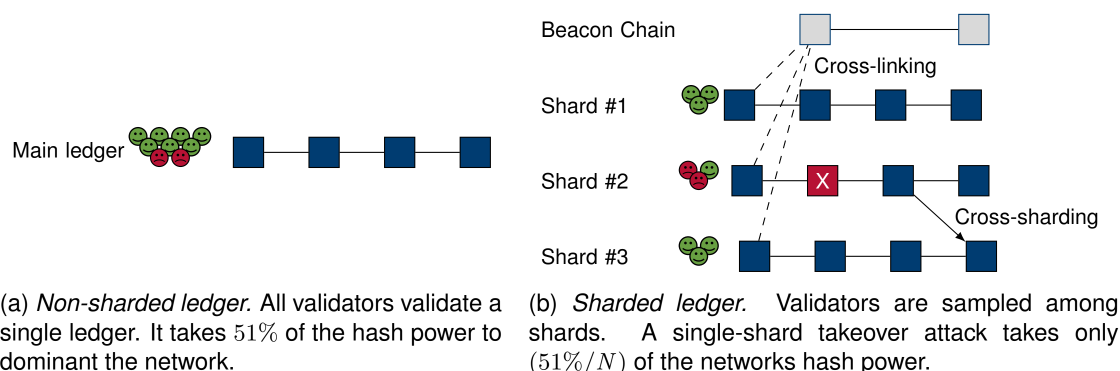
Sharding is a promising technique to improve scalability at the protocol level. With sharding, the computational work is divided among the validators such that every node in the network only has to do a subset of the total computation. This section outlines the state-of-the-art sharding techniques and the major problems that sharding solutions are facing today.

4.1 Sharding fundamentals

The basic idea of sharding is to divide the computational work of a distributed ledger among a subset of nodes. Instead of having a single ledger, there will be multiple ledgers referred to as *shards*. Each shard is having its own set of validators that verify transactions and run a consensus protocol. For now, assume that the shards do not communicate with each other.

4.1.1 Validator partitioning

The first challenge of a sharded ledger is to split the total number of validators among the number of shards while maintaining adequate security. Let's first assume a non-sharded ledger with a Nakamoto Proof-of-Work consensus model. In order to execute a 51% attack, i.e. an attack in which the adversary takes control over the majority of network hash rate to revise transaction history, the adversary needs 51% of the total hash power in the network. Now assume the ledger compromises 10 shards. Given that the hash power is equally distributed over the shards, it takes only $51\%/10 = 5,1\%$ of the network's hash power to dominate a single shard. Thus, the security of the overall system degrades with the number of shards. This type of attack, in which a group of validators collaborate and take over a single shard, is called the *single-shard takeover attack* [60].



(a) *Non-sharded ledger.* All validators validate a single ledger. It takes 51% of the hash power to dominant the network.

(b) *Sharded ledger.* Validators are sampled among shards. A single-shard takeover attack takes only $(51\%/N)$ of the networks hash power.

Figure 4.1: A sharded versus non-sharded ledger

Figure 4.1 visualises a sharded and non-sharded ledger. For now, we focus on validator partitioning. In both ledgers there are 9 validators of which 2 are malicious and collaborate to execute

an attack. In the non-sharded ledger, the malicious validators form a minority and the majority of honest nodes will not accept the invalid blocks produced by the malicious validators. However, the malicious validators in the sharded ledger are all assigned to shard 2 and form a clear majority. The malicious validators could accept a block with an invalid state transition that creates coins out of thin air, e.g. accept a transaction that 'credits' coins to an account without 'debiting' them from another. The validators in the shard committees of the other shards do not validate the produced blocks of shard 2 and are unaware of the compromise.

However, compromising a single shard is only possible if all the collaborating 5,1% of the validators operate in the same shard. Therefore, most sharding designs rely on a source of randomness to assign validators to shards. If validators can not choose which shard they get to validate, it is highly unlikely that all the malicious participants end up in the same shard. Random sampling validators from a *validator pool* into *shard committees* protects against (static) single-shard take over attacks.

Using a binomial distribution, one can calculate the probability of an attacker having compromised more than 50% of the nodes in a single shard. Let us assume a sample size of $N = 100$ and the attack controls $p = 25\%$ of the validator pool. Then, the probability of having more than 50% of the validators in a single shard is = $6.63 * 10^{-8}$ [55].

The beacon chain

There are several tasks and computations in a sharded ledger that are not specific to any particular shard. Many sharding designs build upon the concept of having a separate chain responsible for these computations and linking all the shard chains together, i.e. cross-linking shards. Such a chain is called *the beacon chain* in Ethereum 2.0 [55], a *relay chain* in PolkaDot [22] and a *main chain* in Near Protocol [45]. In this research we will refer to such chain as the beacon chain.

Figure 4.1b visualises the concept of cross-linking. As mentioned in chapter 3.3.2, a cross-link is a set of signatures from a shard committee attesting to a block in the shard chain that is stored in a block on the beacon chain. Once the beacon chain block has been finalised, the corresponding shard block is finalised. The validators of other shards rely on this finalisation in order to include cross-shard transactions.

4.1.2 State partitioning

Assume we have a sharded ledger with adequate security. The second challenge of a sharded ledger is to know what to 'partition' and then how to 'partition' it. In order to understand both, we first look at the primary tasks of a validator. A validator in a DLT performs the following three important tasks:

- **Transaction processing:** A validator processes transactions to create valid blocks.
- **Transaction and block relaying:** A validator relays blocks and transactions to other nodes.
- **State storage:** A validator maintains and stores the state and state history of the ledger.

Each of these tasks poses a growing requirement on a validators' resources with an increased number of transactions being processed. First, a validator requires more compute power. Second, a validator requires more bandwidth to not only relay the transactions but also the increased number of blocks. Finally, a validator requires more storage to store the transactions and transaction history. Unlike the other resources, the required storage requirement grows even as the number of transactions being processed maintains constant.

Earlier work that partitioned the transaction processing only solved the scalability problem partially. The increased transaction processing rate would eventually put a burden on the storage requirement. For example, the Zilliqa test network with 1800 full nodes processes roughly 1200 transactions per second [56]. That is an increase of almost 8000% compared to the Ethereum network. If the Ethereum network would processed the transactions as fast as the Zilliqa network, the size of the ledger after 4 years would not have been 100GB, but roughly 8TB.

Sharding protocols that shard all the three above mentioned tasks are referred to as full sharding solutions. Nowadays, there are many projects working on full sharding solutions. However, most of these projects are still in its infancy.

4.1.3 Cross-shard transactions

The third challenge of a sharded solution is to deal with transaction that change the state of multiple shards, i.e. cross-shard transactions. The changes of these transactions need to atomic to ensure that the ledger remains in a consistent state. An example of such transaction is a token transfer. Token transfers consist of two operations; withdrawing tokens from one account and saving it on another account. If one of these operations failed, the ledger would remain in an inconsistent state. That is, tokens would have been 'magically' created or deleted.

It is important to understand how common cross-shard transactions will be. Let us assume a sharded ledger with 1000 shards and a payment transaction that transfers tokens from one account to another. The probability that both accounts are on the same shard is 0.1%, which means that the probability that the transaction becomes a cross-shard operation is 99.9%. The same problem, as we will explain in 4.4.1, arises with smart contract invocation transactions.

Synchronous and asynchronous approaches

There are in general two approaches to enable cross-shard transactions; (i) *synchronous*, and (ii) *asynchronous* [45, 55].

- With **synchronous** cross-shard transactions, the blocks in the shards that contain the state transitions related to a transaction are all produced at the same time. Producing blocks simultaneously however, requires a high degree of coordination and cooperation across shards. Such an approach may involve high message complexity which also increases the time to create new blocks.

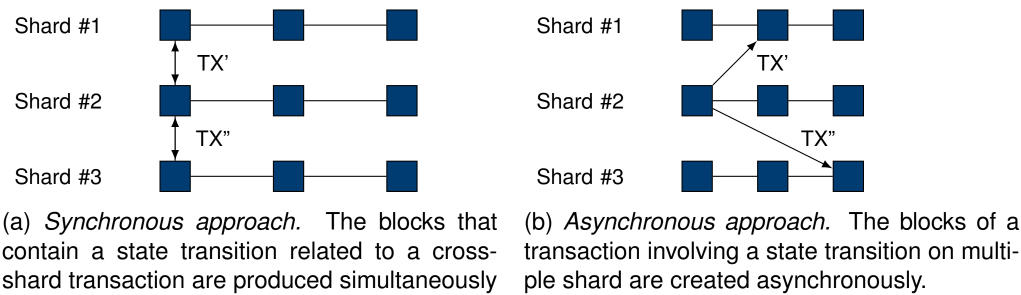
- With **asynchronous** cross-shard transactions, the state transitions in the distinct shards related to the transaction are executed asynchronously. That is, shards communicate asynchronously and non-blocking such that the transition will be executed in its entirety at some future time. This approach eliminates synchronization times and the expensive coordination of producing blocks simultaneously. However, a single cross-shard transaction may incur many block generation rounds which not only results in high transaction latencies, but could also result in long 'locked transition states' to guarantee atomicity.

The choice between *synchronous* and *asynchronous* approaches determines at which level shards will synchronise. Either synchronisation happens at a 'block' level or at a 'smart contract' level.

Synchronous approaches synchronise at a 'block level'; blocks are produced simultaneously and all state transitions related to each transaction occur at the same block height. As a result, every produced block is consistent with the blocks in the sibling shards and every transaction included in the produced blocks are executed in its entirety. However, producing blocks simultaneously comes with a costs; blocks are produced only as fast as it 'slowest' shard and transactions which only incur state transitions on a single shard therefore suffer from the communication overhead of cross-shard transactions.

In asynchronous approaches, on the other hand, shards do not communicate in order to create consistent blocks simultaneously. Each shard creates blocks independently and at their own speed. In order to guarantee atomicity of cross-shard transactions, a shard may lock some state transition until it is certain that the transition on the other shard is executed. Communication between shards happens asynchronously which means that the state of some transaction may be blocked in several consecutive created blocks. In order to overcome these limitations, one could use some optimistic locking mechanism which assumes that the transaction most likely will be applied in its entirety. The trade-off however, is that optimistic locking could result in a cascading effect of 'revert' operations.

Figure 4.2 visualises both approaches. The asynchronous approach requires shards to communicate in order to apply all state transitions of TX simultaneously resulting in an increased block time generation interval. With the asynchronous approach, blocks are produced more frequently but the processing time of a cross-shard transaction increases.



(a) *Synchronous approach.* The blocks that contain a state transition related to a cross-shard transaction are produced simultaneously

(b) *Asynchronous approach.* The blocks of a transaction involving a state transition on multiple shard are created asynchronously.

Figure 4.2: Synchronous versus asynchronous cross-shard transaction approaches

Although there are some interesting synchronous cross-shard discussions worth sharing [10, 20], to the best of our knowledge, there is no sharding project adopting a synchronous cross-sharding model. At this point, we further assume an asynchronous cross-sharding approach in this thesis.

Transaction receipts

A commonly used technique to facilitate cross-shard communication asynchronously is with transaction receipts. Receipts are objects that represent an effect of a transaction which is not directly stored in the state, but where the fact that the receipt was generated can be verified. This way, a shard can send a cross-shard operation via a receipt to another shard which subsequently can verify that the receipt is valid [45, 55, 60].

Continuing our example from earlier, in which tokens are transferred cross-shard, we can illustrate how receipts work;

1. A token transfer transaction is sent to shard A, which maintains the state of the sender.
2. Shard A 'withdraws' the tokens from the sender's account and creates a receipt to add these token on the addressee's account, which is stored on shard B.
3. Shard A sends the receipt to shard B.
4. Shard B waits for the transaction to be finalised and subsequently processes the receipt by adding the tokens on the account of the addressee.

The above approach assumes that transactions and thus receipts are eventually finalised. Without finalisation, the above approach needs to be extended to remain consistent if generated receipts may get reverted.

Hot-shard problem

One problem that a sharded solution could encounter is that the shards eventually become unbalanced. This problem is referred to as the *hot-shard* problem [43] or *hotspot* / *hot-partitioning* problem in traditional databases [24]. In an unbalanced system, some shards receive much higher volume of transactions such that their transaction pool pile up and lead to high latencies for a significant portion of transactions.

As we mentioned previously, Ethereum addresses this problem through independent gas markets. The basic idea behind independent gas markets is that new smart contracts will be deployed on the 'cheapest' shard such that the shards will automatically be balanced. Alternatively, one could rebalance shards by repartitioning the smart contracts and user accounts across the shards.

Train and hotel problem

The *train and hotel problem* is an example given by A. Miller to illustrate the difficulty with cross-shard communication [55]. In this problem, we want to book a train ticket and make a hotel reservation. However, if one of them fails, we want neither. Let us suppose that the train ticket and hotel booking applications are two independent smart contracts. If both smart contracts are assigned to the same

shard, then we can create a transaction that attempts to make both reservation. The booking applications are able to throw an error that will revert all changes made if no ticket is available. However, if the contracts are located on different shards, the shards need to coordinate the reservation in order to make sure that the operation occurs atomic.

The mechanisms designed to guarantee atomicity can be categorised into two types; (i) *locking*, and (ii) *yanking*.

Locking A *lock* is a synchronisation mechanism designed to enforce mutual exclusion concurrency control policies. With locks, one can limit access to a resource in a concurrent environment. This way, one can prevent lost updates, dirty reads and ensure atomicity in distributed transactions.

In the above example, there are three approaches two implement locking:

- **User locks both smart contracts:** A user requires a lock on both smart contracts. The user tries to reserve both tickets. If both tickets are available, then the user finalises the reservation by committing the transaction, otherwise he reverts the state and unlocks both smart contacts.
- **Shard locks both smart contracts:** A user sends a transaction to a new 'hotel-and-train' booking smart contact. Subsequently, the shard which stores the new smart contract and processes the transaction tries to book the tickets in a similar approach as the user would do.
- **Shard applies application specific locks:** A user sends a transaction to a new 'hotel-and-train' booking smart contact. But instead of locking the entire state of the smart contract, individual tickets can be reserved, i.e. locked. If both tickets are reserved, then the tickets can be booked.

The first approach is coordinated by the user itself. To prevent malicious behaviour, i.e. the locking of smart contracts without the intention to unlock it ever again, could be addressed by pricing the lock operation or require some deposit which would be decreased if the duration of time far exceeds the time allowed to lock the smart contract. The problem however, is that the user never knows how long it will take before his transaction would be processed at a specific shard. Both Atomic [35] and S-BAC [4] are distributed atomic commit protocols using this approach.

The second approaches is coordinated by the shard which require to execute state-transitions which are located on another shard. In contrast to a user, a shard is assumed to be honest because it is operated by a majority of honest validators. However, without a guaranteed execution protocol, there is no guarantee that operations of a shard are eventually executed on another shard. Shards require some mechanism to unlock the state after some period of time.

A drawback of those approaches is that the entire smart contract state becomes locked. This means, that it stops all other users from booking any tickets. One particular problem is that external smart contract invoking some smart contract A could prevent other users to access smart contract A for some fixed period of time. This could lead to grief attacks which result in smart contract denial-of-service attacks.

The third approach is rather an application specific approach, in which the smart contract allows to lock, i.e. reserve, and unlock, i.e. book or revert, specific tickets. This way, the smart contract it self does not become locked and other users are able to use the smart contract. However, the smart contract stills needs some mechanism to unlock tickets if they are not locked for some time.

Yanking Another mechanism to ensure transaction atomicity was proposed by P. Merriam and referred to as *yanking* [40, 55]. The basic idea is that the state of some smart contract can be frozen and pulled to another shard, i.e. 'yanked' to another shard. This way, a cross-shard transaction that is processed on shard A will yank all related smart contract to A such that the transaction can be executed in one execution.

Although this solution may simplify smart contract execution, it still blocks other users to access 'yanked' smart contracts during the time the smart contract is moved. At the same time, yanking is more probably 'costly' compared to transaction request, as the entire state needs to be moved.

For yanking to be efficient, the internal state of the smart contract that is to be yankeed needs to be small. Rather than yanking the entire ticketing smart contract, a ticketing contract could create tickets as child smart contracts which subsequently could be yanked. Requiring smart contract to create child smart contract however, is again an application-specific solution.

Types of transactions

Like in traditional distributed databases, transactions in DLT can be categorised in (i) *flat*, and (ii) *nested* transactions.

- A **flat** transaction is a transaction with a sequence of operations.
- A **nested** transaction is a transaction that is started by some operation within the scope of an already started transaction.

Every transaction in a non-shard ledger is a *flat* transaction. The transaction results in a set of sequential operations and if all operations are successful, the transaction it self is successful. In a distributed ledger, the transaction is marked and processed and valid, but not necessarily finalised. An example of a *flat* transaction is a token transfer transaction which result in two operations; a decrease and a increase operation.

If the the account of the addressee is stored on another shard, the flat transaction would result result in a receipt which is subsequently packed in a transaction and send to required shard. This new transaction is created within the scope of an already started transaction, the parent, and thus classified as a *nested* transaction. It is important to understand the difference. *Nested* transactions, in contrast to flat transactions, are successful if the operations within the transaction succeed with the additional requirement that the parent transaction needs to be successful as well.

It is helpful to illustrate this with an example; let's assume a flat transaction F that results in two sequential nested transactions N_1 and N_2. First, nested transaction N_1 is processed and successfully executes its inner operations. Sequentially, the second nested transaction is executed. However, this transaction fails. Then, depending on the logic within the flat transaction F, this transaction may fail as well. In that case the nested transaction N_1 must be reverted despite that it was successful. Reverting N_1 is required in order to guarantee atomicity of F.

The problems of blocking cross-shard transactions

Up to know, we mainly focussed on payment transactions. Payment transactions can be divided into a 'debit' and 'credit' operation. In case of a cross-shard transaction, the 'credit' operation can asynchronously be send to the target shard in the form of a receipt. As long as the receipt is processed at the target shard, the entire ledger remains consistent. None of the operations are blocking, i.e. need to wait for an event to occur.

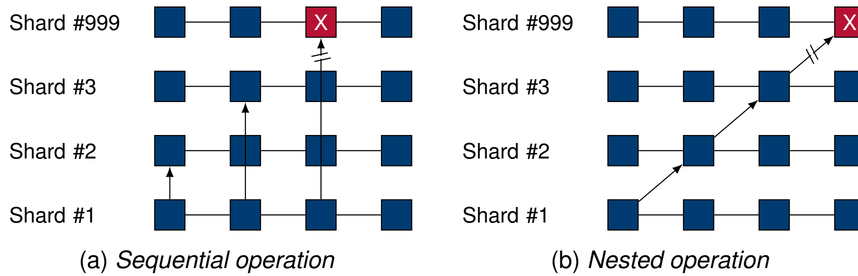
A smart contract transaction, on the other hand, is transaction that invokes a method that carries out a task defined by a sequence of statements. Such a statement may be an method invocation to an external contract maintained by another shard. If the invocation is blocking, e.g. the caller requires the return value to execute the successive statement, then the execution needs to be 'paused' such that the shard maintaining the external contract may process the receipt with the invoking call.

In an ideal situation, the receipt is directly processed by the shard of the callee. In that case, the caller would have been locked for a period of time equal to the time to generate a new block. Then, the time needed to process the transaction in its entirety would at least take 3 block periods; in the first block the transaction was included which created a receipt. The second block to process the receipt on the sibling shard and the third block to continue with the result on the receipt in the original shard.

If the method only invoked one cross-shard transaction, then the overhead of cross-shard transaction may be acceptable. However, it becomes problematic in there are many cross-shard transaction. We can distinguish between two types of operations which may become problematic:

- **Sequential operations:** The sequential execution of operations leading to cross-shard transactions.
- **Nested operations:** The execution of a cross-shard transaction leading to another cross-shard transactions.

Figure 4.3 visualises both operations. We assume a transaction invoked a smart contract method on shard $\#1$. In figure 4.3a, the invokes method executes a set of sequential statements, which all lead to a cross-shard operation. Every operation needs to be processed on the other shard



(a) *Sequential operation* (b) *Nested operation*

Figure 4.3: Nested and sequential operations with cross-shard transactions

before invoking the next one. In figure 4.3b, the invoked method on shard $\#1$ invoked another smart contract method on shard $\#2$, which again invoked a method on shard $\#3$, eventually forming a chain of invocations.

In both cases, the time needed to process the original transaction increased significantly for every additional cross-shard call. That is, in this exaggerated example with 998 cross-shard operations, at least the period to create 1000 successive blocks. Furthermore, remember that every cross-shard operation is a nested transaction which only succeeds if the parent transaction succeeds as well. The state of every invoked smart contract must therefore be lock locked till the parent operation is finished. Alternatively, one could apply an optimistic locking strategy, which means that state of the smart contract could be reverted with a maxim of 1000 blocks. In figure 4.3, we highlighted a failed operation with a red block.

One optimization for the sequential operations is to enable parallel cross-shard invocations. Either the compiler could automatically optimise cross-shard invocations by reorganising unrelated statements or the smart contract language itself could provide parallel invocation constructs. In that case, the sequential operation example could be optimised as visualised in figure 4.4. Note that, this only work if the statements do not depend on each other.

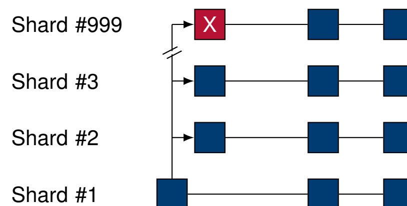


Figure 4.4: Parallel execution of cross-shard transactions

4.1.4 Sharding limitations

Scalability is the ability of a system to handle a growing amount of work by adding resources to the system. With sharding, the increase in transaction throughput is linear in the number of shards. In that sense, the overall performance increases by increasing the size of the computational resources. However, the overall transaction throughput is increased through parallelism. Each shard processes a set of distinct transactions in parallel and the overall transaction rate is calculated by multiplying the number of shards with the shard transaction throughput. Computational work which is inherently serial, is still limited to the maximum transaction throughput of a single shard.

With respect to smart contracts, the transaction throughput is still limited to a single shard. That said, sharding enables smart contracts to invoke smart contracts methods on other shards in parallel, but this comes with a price in terms of the transaction processing time, as explained in 4.1.3.

4.2 State validity

The concept of sharding is to partition a large system into smaller and more manageable parts called shards. In context of DLT, each shard would hold a unique set of accounts, i.e. smart contracts and

external account balances. Validators assigned to a shard only process and validate the transaction related to that shard, instead of validating every transaction.

The partitioning of the state into smaller shard however, raises a significant challenge: how to remain state validity if each shard is only maintained by a small number of validators?

4.2.1 The adaptive adversary

The adversary model defines the capabilities and assumptions of the attacker under which safety and liveness properties of the system are evaluated. In distributed ledgers, the adversary is typically allowed to control or corrupt a set of validators. How these validators are corrupted, depends on the type of adversary. One standard model assumes that the set of malicious validators is chosen in advance. This is the model of the *non-adaptive* adversaries. Alternatively, the adversary may corrupt validators over time after learning new information. These are the *adaptive* adversaries.

In the *non-adaptive adversary* model, random sampling of validator into shard committees protects against single-shard takeover attacks. It may seem obvious that random sampling does not protect against an adaptive adversary; the adversary can simply corrupt the validators after the have been assigned to a shard committee. However, assuming such *fully adaptive adversary*, which can corrupt any validator at any time, seems to be unrealistic. A more realistic assumption would be to assume that the adversary may only corrupt validators at bounded rate [46]. Such adversary is sometimes refereed to as *slowly adaptive*, *weekly adaptive* or *semi-adaptive*.

4.2.2 Sample size and validator rotation

A *slowly adaptive* adversary may corrupt validators in a shard committee at a constant speed. In order to protect against such adversary, one could frequently rotate all validators such that the corrupted shard validators never become the majority of a committee. The validators should at least be rotated as often as the adversary can corrupt new nodes. However, validator rotation comes with a cost. New validators have to bootstrap the shard state, i.e. download a package copy of the ledger, before they can start validating. Even with fast sync, the Ethereum ledger still takes around 6 hours to download. Validators are therefore typically rotated one-by-one. Not only to spread the work of resynching each validator, but also to minimise the probability on any data loss.

The security of the shards not only depends on the rotation of validator, but also on the validator committee size. Each validator committee has a minimum number of validators. This number is typically determined by the probability of having more than $2/3$ of malicious validators in a single shard after random sampling all validators. In Ethereum 2.0, the minimum number of validators is set to 111, which means that the probability of having $2/3$ of malicious nodes in a single shard is $2^{-}40$. However, this calculation assumes that the random function may not be biased by an attacker. If the attacker may bias the the random number generator, it is more secure to choose a larger validator committee.

Remember from section 3, that existing traditional BFT consensus models may efficiently scale up to +-100 validators. Most sharding solutions with traditional BFT consensus therefore face a greater risk on a single-shard takeover attack.

4.2.3 Fisherman

An alternative approach to ensure data validity is known as *the fisherman approach*. Instead of requiring the majority of validators in each shard committee to be honest, the fisherman approach just requires to have at least one honest validator. The approach as follows, whenever a new bock is proposed, there is a time in which any honest validator can provide proof that the block is in fact malicious. This period is called the *challenge period*. Any honest validator can challenge the block which needs to be verified by the other shards. If the challenge is indeed correct, the state invalid state transition may be reverted and the validators who attested for that specific block may be slashed. Figure 4.5 visualises this approach.

Although this approach seems promising, there several aspects that one need to get right. First of all, this model requires at least one 'fishermen' to validate a specific shard to ensure state validity. However, without incentives, there is not only no guarantee that there will be a fishermen at all.

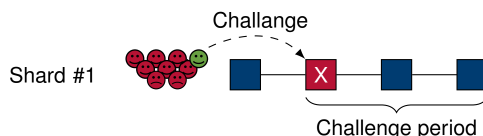


Figure 4.5: The Fisherman approach

Motivating fishermans with incentives is tricky, as a node could just sign up for the fisherman role without actually validating, or alternatively, copy-catting other fishermans. A good fisherman scheme should therefore insure that honest fisherman are rewarded and malicious fishermans are punished. For example by requiring a fisherman to deposit some amount of tokens.

From this perspective, it seems that there are many similarities with a threshold majority approach with certain slashing conditions. Perhaps, with shifted threshold parameters. In a threshold consensus model, a new block is under the condition that the majority of nodes are honest. With the fisherman model, there is not such guarantee. In order to get such guarantee, we require a number of fisherman to provide some us with some proof that they indeed verified the new block. That is, some majority of fisherman.

4.2.4 Verifiable computation

Another approach to ensure data validity is through verifiable computation. Instead of requiring each validator to do the same 'expensive' computation, the block proposer could create some proof that the computation was correct. Then, the verifying validators only need to verify the proof. If the cost of the verification is smaller compared to the original computation, then more validator could participate in the same shard without additional overhead. However, as mentioned in 2, existing verifiable computation schemes are still highly impractical.

4.3 Data availability

The partitioning of data across shards also poses a challenge on the data availability guarantee. While the partitioning increases the computational resources, i.e. yield a higher transaction rate, fewer nodes are maintaining a copy of each part of data. As a result, the number of light clients have increased while being depended on a smaller number of validators for distinct parts of data. This poses the question; how can one ensure data availability?

Data availability is a necessary property in DLT. If a majority of validator collude to produce blocks without disclosing the entire content, other validators may not be able to determine whether the block is valid. This may lead to undesired behaviour. On the hand, in the fisherman approach, nodes operating as a fisherman may not be able to propose a challenge within the challenging period, leading to invalid but finalised blocks. On the other hand, with data availability requirements, not publishing data may stall block production or even lead to situations in which some blocks may get reverted after a while.

Light-clients are in particular vulnerable to data availability attacks. Light-clients only keep track of block headers and only require proof that some transaction is part of a block in order to convince them self that the transaction is valid. For that, light-clients rely on full-nodes and download parts of the block contents. Then, a merchant acting a light client may be convinced that he received some money and send his merchandise. However, because some other transaction was not published, the block may be reverted.

The problem with *data availability* is that publishing data is not a uniquely attributable fault. That is, in any scheme where a node has the ability to make a claim that the publisher was withholding data, one cannot distinguish between the case that a malicious publisher did not publish some data and a node making a malicious claim. If the claim was indeed correct, the publisher could yet release the remaining content and state it was some network issue. This problem is referred to as the 'fisherman's dilemma' and shows that fraud proofs are unusable for data availability schemes.

The current state of literature provides two direction to address data availability; (i) *proof-of-custody* protocols, and (ii) *erasure coding* schemes.

4.3.1 Proof-of-Custody

The idea behind Proof-Of-Custody protocol is to guarantee data availability through some 'proof' of actual custody. A Proof-of-Custody is a cryptographic guarantee that some identity had full access to some pieces of data during the creation of the proof. However, the problem with these 'proofs' is that identities could attest for data pieces without actually having seen them. Similar to *fisherman* schemes, one could copycat attestations of others. It is therefore important that attestations are unique addressable to the registered data availability 'provers'. In that case, an incentive scheme could ensure that these identities indeed follow the protocol.

Most Proof-Of-Custody scheme proposals use some similar approach to the following:

1. An prover creates a secret s and publishes only the hash of that secret, $h = hash(s)$, on-chain.
2. The identity publishes attestations through hashing the data together with the secret.
3. After a fixed number of rounds, the identity releases the secret.
4. Other 'provers' verify that the attestations were indeed correct.

These approaches however introduce quite some additional overhead. Research on Proof-Of-Custody protocol focusses on improving the efficiency of:

- **Computational overhead:** In order to proof correctness of the attestation of a single prove, one not only needs to be in possession of the data, but also redo every calculation.
- **Attestation storage overhead:** Every attestation of every block of every prover needs to be stored on-chain.

One promising idea proposed for Ethereum 2.0 is to add a 1-bit proof-of-custody field to every block attestation of every validators. By only including the 2 first bits of every attention, there is a really small probability that a prover created attestations at random [23].

4.3.2 Erasure coding

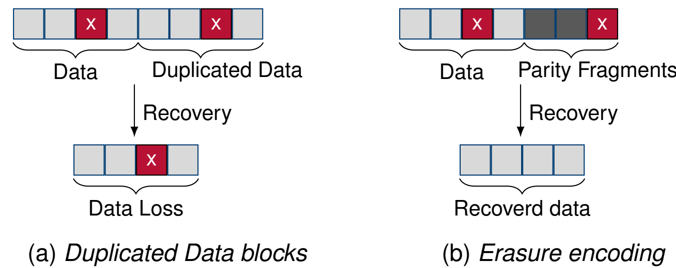
Erasure encoding is a technique in which data is fractioned into segments and encoded with redundant data pieces in such a way that only a subset of pieces is needed to recreate the original data piece. In stead of replicating data over multiple nodes, erasure code have to additional benefits:

- **Storage efficiency:** Erasure codes require less storage compared to the naive replication paradigm.
- **Greater recoverability:** Erasure codes enable recoverability in case of data loss of the same fragments.

Figure 4.6 visualises the difference between a naive replication technique and erasure encoding. In both cases, two blocks are lost. However, with the erasure coding technique the data can be recovered while with the data duplication technique some data is lost forever. Erasure coding provide storage efficiency and recoverability at a cost of additional computation.

4.4 Smart contracts on a sharded architecture

The deployment of smart contract on a sharded architecture is still in early stages of research and poses many challenges [51]. A major issue is that the invocation of cross-shard operations involve significant delays. Highly correlated smart contract may arise in many nested or sequential cross-shard operations. The result is that cross-shard transactions involve high latencies and the state of all involved smart contracts may be locked for other users during this period. Besides, allowing



(a) *Duplicated Data blocks* (b) *Erasure encoding*

Figure 4.6: Recovering data from duplication versus erasure encoded blocks

smart contract to lock the state of other smart contracts could lead to denial-of-service attacks in which attackers prevent users from accessing particular smart contracts.

To minimise the number of blocking cross-shard operations one could dispatch correlated smart contract to the same shard. That is, allow user to select to which shard a smart contract must be deployed. Ethereum 2.0 is adapting this approach [55]. While this increases smart contract performance and reduces latencies, it requires other measures, among others to balance shards. Ethereum addresses the former via independent Gas markets, i.e. new smart contracts will typically be deployed on 'the cheapest' shard and thus automatically balance the shards. However, the assumption is that new new smart contracts are not related to other smart contracts, otherwise it would probably be deployed in the same shard. It seems not unreasonable that new deployed smart contracts will have other dependencies. Moreover, assigning smart contracts statically to shards may be problematic for *adaptive* sharding, in which the number may be reorganised based on the number of active validators.

On the other hand, one could dynamically dispatch smart contracts in shards. This approach is used by Elrond [51]. Elrond partitions the address space deterministically according to a binary tree structure. That is, a node in the binary represents a address range and splitting the range by halve will result into two children. The leafs of the tree represent the actual shards. By dividing the address space deterministically, it also provides a mechanism for automatic transaction routing in the corresponding shards. However, this approach assumes that the contract-state and transaction is equally balanced. If some sets of account addresses are almost never used while others are used all the time, the shards may not be balanced after all. Alternatively, one could use dynamic account address ranges, by storing the start address of each shard as extra overhead instead of using predefined split structure. This way, shards could dynamically be adapted based on the number of transaction being processed during some previous Epochs. However, this will add extra split overhead, as shard need to resynchronise state after adapting.

A key observation is that the design heavily depends on the assumed use case. How correlated will the deployed smart contracts be? Can we split correlated smart contracts in disconnected sets? How frequent will shard be reorganised? Unfortunately, we could not find data regarding smart contract correlation. Moreover, it is questionable whether empirical data reflects future use cases. It is argued that the lack of scalability is the major bottleneck for widely adoption.

Nonetheless, every sharding model is a trade-off between (i) *performance* - the number of transactions being processed, (ii) *latencies* - the time needed for transaction to be finalised, (iii) *security* - the exploitability of possible vulnerabilities , and (iv) *overhead* the additional cost of storage or computation with regards to shard reorganisation and transaction processing.

4.4.1 Smart Contract invocation

While smart contract methods may be invoked by other smart contracts, the initial invocation is always an transaction. Such a transaction typically includes an 'address' field including the address of the smart contract, a 'value' field to add some inherent token that may be send to the smart contract and some 'transaction fee' field to pay for the transaction cost. The execution then includes the following steps:

1. Withdraw the transaction fee and value field from the user account.

2. Execute the specified message call
3. Refund unconsumed transaction fee (if applicable)

In a full-sharding model, the account of the user may not be stored on the same shard as the account of the smart contract. Then, this operation will involve at least 2 cross-shard transactions and thus at least 3 block periods before the transaction is processed in its entirety. Similar to payment transactions, one could assume that the probability of having a user account and a random smart contract on the same shard is $1/n$, with n shards. Then, there will be point in at which adding an extra shard will lead to more cross-shard transaction than actual user transactions. Using a transaction rate to compare sharding designs or improvement with regard to non-sharded ledgers does not seems to make sense.

Alternatively, one could require users to have an account in the shards as the smart contracts. Quarkchain uses this approach [44]. They assume that only a small set of users would want to communicate with multiple smart contracts. However, if a user will use a huge number of dApps, it effectively needs to create multiple accounts in multiple shards. Some argue, that this is not state-sharding [52].

4.5 Related work

Sharding is seen as an approach to scalability by many DLT projects. That having said, the technology is still heavily being researched and under developments. Most projects still have to move through several milestones before the launch of their 'main net'. The typical approach is to create secure shards at first and subsequently focus on improving cross-shard operations. But as we mentioned before, sharding has its limitations and there is a need for fast-cross shard transactions. Most state-of-the-art architectures fail to mention this aspect [50]. In addition, the design goals of these projects are most impressive. But as we have seen before, the launch of several projects have been delayed again and again. Moreover, some project even switched their initial design focus. For example, NEAR Protocol moved from the idea of a mobile-first platform [42] to a general purpose 'community-operated cloud' platform [57].

4.5.1 Sharded-based smart contract architectures

While there are many projects working on sharding, we only studied the results that focused on sharding-based smart contract architectures. Zilliqa [56] was the first sharding-based project that addressed the limited transaction throughput through transaction and network sharding. It introduced a design in which shards could process transactions in parallel. Their protocol processes transactions which only involve state-transitions in one shard in parallel and transactions which involve transitions in multiple shards sequentially. That is, creating a two-step protocol without the notion of cross-shard transactions. Individual shards use a BFT consensus model, while PoW is still used for the verification of validators.

However, one quickly shared the opinion that sharding transaction processing alone would only solve one problems, limiting its potential. At this point, researches started focussing on full sharding designs which lead to two general approaches. The first approach focussed on improving cross-chain *interoperability* while the second approaches focussed on *performance* improvements.

Chain Interoperability

Cosmos [38] is an example of the primer approach. In Cosmos, each shard, referred to as a zone, has its own validators. These validators work independently without any sampling or rotating. The validators of one zone has to trust the validators of the other zone if they want to communicate with each other. The idea of Cosmos is to improve chain interoperability by providing efficient asset transfers protocols. Cosmos shards use Tendermint for consensus and their beacon chain is referred to as Cosmos Hub.

Polkadot [22] also follows the primer approach. But rather than having independent shards, Polkadot validators rotate between shards. Paradot consists of heterogeneous 'shards' known as

parachains which blocks are cross-linked in a beacon chain known as the relay chain. This means that parachains of varying degrees share security and can trust other parachains because of the finalisations on the relay chain. The relay chain uses a special consensus model GRANDPA, which is most similar to Casper FFG, while parachains may choose their own consensus model. The focus of Polkadot is to improve interoperability between chains and therefore offers special 'bridge contracts' to enable contract invocations from external chains. It also states that it will support cross-parachain smart contract invocation, however it is it not implemented yet.

Chain performance

Although the first approach seems promising, many more projects are working on the second approach. The goal of the second approach is to improve performance by processing transaction in parallel on homogeneous shards. A key observation here is that the high-level designs are quite similar, but the differences typically arise due to the chosen balance between security, scalability and decentralization. We will present three projects all using a 'beacon' or 'main' but with subtle differences; (i) *Elrond*, (ii) *Ethereum 2.0*, and (iii) *NEAR Protocol*.

Elrond [51] uses a BFT consensus model in which nodes deposit a fixed amount stake to become a validator. However, Elrond refines the consensus mechanism by adding an additional weight factor called rating. The probability of being selected as a validator for a shard committee is weighted by the rate of the validator which slowly increases after successfully proposing new blocks. They named this model Secure PoS. The validators for the committee are chosen every round from a pool of validators assigned to that shard. This way, the committee rotate more frequently but without having to synchronise state data every time. The other validators can still operate as a fisherman, however, there protocol does not state how data availability is guaranteed. Then, at the end of very epoch, a number of validators (less then $1/3$) assigned to a shard validator pool is rotated. Elrond furthermore uses a hierarchical shard model and introduced 'shard redundancy'. The dividing of the address space between shards is predetermined by hierarchy based on a binary tree. In order to create another shard, the address space of a leaf in the binary will be split into two equals parts, creating two new leafs which represent two new shards. However, the validators of each shard will both maintain the state data of each other. This is referred to shard redundancy. A major benefit of this approach is that two sibling shards can easily be merged without extra ing overhead. Another major benefit is that it is easy to determine to which shard a transaction needs to be addressed. A major limitation of this approach is that shards may not become balanced. Moreover, smart contracts do not have any influence on which address they will be published. A cross-shard smart contract call needs at least 5 rounds because every produced block in a shard needs to be finalised before other shard may process the related transactions, which limits its potential. Cross-shard transaction produced in one block addressed to one other shard are grouped into a 'mini-block', which either need to be processed by the target shard atomically. This subsequently minimises the communication overhead. A final note, it is not yet specified how atomic operations,e.g. hotel-and-train problem, is going to be addressed, neither how the halting problem is going to be solved.

Ethereum 2.0 [55], extensively discussed in section 3.3.3, uses a Nakamoto consensus model in which nodes deposit a minimum amount of stake. The amount of deposited stake is proportional to the number of validator seats. However, each seat is assigned to distinct shards. Validators are slowly rotated every epoch, but the shard committee remains the same for every epoch in every round. Compared to Elrond, (i) the Epoch time is set to 6.4 minutes while Elronds has an Epoch time of 24 hours, (ii) blocks are only finalised every Epoch while Elrond blocks are finalised mostly directly after k blocks, i.e. the fisherman challenging period, and (iii) Ethereum chooses 'availability' over 'consistency'.. Unfortunately, because Ethereum yet has to release the first phase, many cross-sharding details are unknown. The basic idea is to allow smart contracts to choose to which shard they will belong, but details are missing.

In NightShade [45], the sharding design of NEAR Protocol, there are two roles: (i) *block producers*, and (ii) *validators*.. In order to become a validator, a node has to deposit some stake. The validators with the largest stake at the beginning of an particular Epoch are assigned to be block producers for that Epoch. This model is referred to as Threshold PoS. However, in NightShade there are no shard chains, instead the validators are working on a single chain, the 'main chain'. The state of this chain is split into shards. Each block producer is assigned to multiple shards but there are only few block producers per shard. For every block, produced on the main chain, there is one

block producer assigned to produce the part of the block which is related to the shard. This part is called a ‘chunk’ and the producer of that chunk in a specific shard is called a ‘chunk producer’. All block producers rotate to create new blocks uses Nakamoto consensus, and the idea is to use some finalisation gadget on top of it. Another major difference is that a chunk does not include the state execution result of the included transactions, instead, the validators receive a block and process the chunk to which they are assigned to and attest to the execution by sending attestations to the block producers. Then, the next chunk included the state execution result of the previous chunk. Moreover, any of the validators can pose a challenge and operate as a fisherman if one chunk in valid. In that case, the entire block and the blocks on top of it will be reverted. A major benefit of this approach is that allows chunk producers to directly include cross-shard transactions even if the previous block is not finalised.

Part II

Solution

Chapter 5

System overview

In the previous chapter we presented the current state-of-the-art sharding protocols. We identified the major challenges that sharding protocols are facing today and the problems of cross-shard transactions. In the next chapters we will propose and discuss Guaranteed-TX, a guaranteed cross-shard execution protocol for Ethereum 2.0. Although we already discussed many aspects of Ethereum 2.0 in detail, the current specification is not set in stone and may change over-time. This section briefly discusses the design goals and system model of Ethereum 2.0 to state the assumptions our solution is build upon.

5.1 Design Goals

Ethereum 2.0 has stated the following design goals: [54]

- **Simplicity:** To minimise complexity, even at the cost of some losses in efficiency.
- **Resilience:** To remain live through a major network partition and when very large portions of nodes go off-line.
- **Longevity:** To select components such that they are either quantum secure or can be easily swapped out for counterparts when available.
- **Security:** To utilize crypto and design techniques that allow for a large participation of validators in total and per unit time.
- **Decentralisation:** To allow for a typical consumer laptop to process/validate $O(1)$ shards.

5.2 System Model

5.2.1 Network model

We consider a peer-to-peer system consisting of $n = 3f + 1$ validators with established identities, i.e. all having a public-private key pair. The messages sent in the network are all authenticated and the public keys of each validator is known. The messages are propagated through a gossip protocol and the validators are well connected. We further assume partial synchrony, i.e. messages arrive within an upper bound but it is unknown as a priori.

5.2.2 Threat model

We consider a *slowly adaptive adversary*, i.e. the adversary can decide to corrupt $f < n/3$ validators over time based on what it learns, but corrupting validators cost time. Although the specification does not yet specify how quickly validators may become corrupted, we assume they may not corrupt a single shard. That is, the validators are rotated more frequently then the adversary can corrupt the validators in a single shard. Moreover, validators may disconnect from the network at any time due

to any reason. However, we assume the validators to be rational. A validator may deviate from the protocol if it benefits him more than following the protocol as normal.

In addition, we assume that the inherent token, i.e. Eth2.0, is stable and do not consider attackers to have early monopoly power.

5.2.3 Sharding model

We consider a sharding model with a beacon chain and shard chains. The beacon chain maintains a registry of active and inactive validators. The active validators are randomly sampled to shard chain committees of minimal 128 validators and maximum 4,096 validators. Equivocating validators are punished through a slashing mechanism and shard cross-links are made at epoch checkpoints.

Shard chains independently maintain state data, i.e. a set of accounts and smart contracts, and execute state transitions. Erasure codes are used to guarantee data availability in shards. The targeted number of shards is 1024.

Chapter 6

Guaranteed transaction execution

In this chapter, we take a closer look at cross-shard transactions. We identify the need for cross-shard transactions and the problems related to guaranteed cross-shard transaction execution.

6.1 Transactions in a non-sharded ledger

Let us briefly discuss the processing flow of a transaction in a non-sharded ledger for some token transfer. In order to transfer some tokens from account A to account B, a transaction is created which is signed by the owner of account A. The transaction is disseminated through the network and eventually processed given that the transaction fee is sufficient to incent a validator to include the transaction in a block. Transaction fees not only ensure liveness, but also protects the network against denial-of-service attacks.

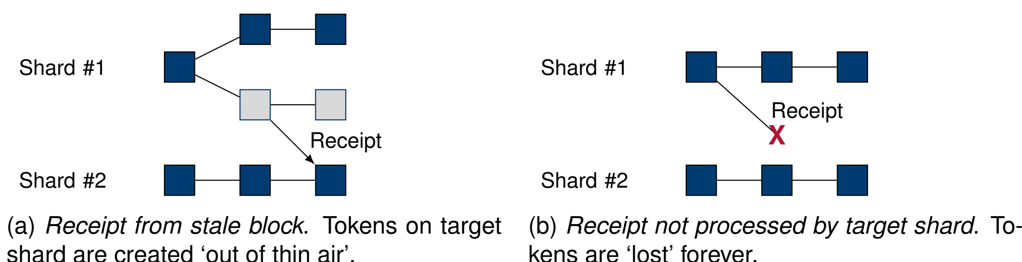
Once the transaction is included in a block, it results in a state transition from the state before to the transaction to a new state after the transaction. The transactions in the block are serialisable, i.e. they happen one after the other in isolation.

6.2 Transactions in a sharded ledger

Now assume the same transaction but in a sharded ledger in which account A is stored on shard S_a and account B is stored on S_b. Then, the transaction is processed on shard S_a which creates a receipt that needs to be processed on shard S_b. With token transfers, the common approach is to 'burn' the tokens on S_a and 'create' new tokens on S_b.

However, this is not as simple as it seems. There are two problems that need to be addressed. First, the block in which the receipt was created that is processed by the target shard could become a stale block, as visualised in figure 6.1a. The receipt will create new tokens out of thin air. Note that, this is only possible with forkful ledgers in which receipts may get processed before being finalised.

Secondly, the generated receipt may not get processed on the target shard. The receipt, a transaction addressed to shard S_b, has no guarantee that it eventually will be processed. The transaction



(a) *Receipt from stale block*. Tokens on target shard are created 'out of thin air'.

(b) *Receipt not processed by target shard*. Tokens are 'lost' forever.

Figure 6.1: Inconsistent cross-shard operations

fee determined by the source shard may not be sufficient to be included by the target shard. Moreover, the transaction may get lost due to network issues or be censored by malicious validators. Figure 6.1b visualises this problem. The 'burned' tokens on the source shard will be lost forever.

6.2.1 The need for guaranteed cross-shard transaction execution

In the previous chapter we identified that the probability of a transaction being cross-shard is $\frac{(N-1)}{N}\%$, given N shards and assuming accounts are uniformly distributed. With a guaranteed cross-shard transaction execution protocol, all cross-shard transaction will eventually be processed. Such a protocol provides the following benefits:

- Ensure state consistency across shards.
- Provide upper bound latencies needed for mission critical applications.
- Prevent performance degradation due to slow cross-shard transactions and locks for smart-contract state being held for long periods of time.

6.3 Separating transactions

The example of a cross-shard transaction in a sharded ledger in section 6.2 gives an intuitive feeling that there are different types of transactions. It is helpful to make a distinction between these transactions, even though the underlying data structures may be the same. We classify every transaction into (i) *application transactions*, and (ii) *cross-shard transactions*.

6.3.1 Application transactions

Users interact with distributed ledgers through commands wrapped in transactions. These commands can be of the following types; (i) token transfers, (ii) smart contract publishments, and (iii) smart contract method invocations. A *user*, which is referred to as an external actor in the yellow page of Ethereum [61], can be a real person or an external entity which goal is to interact with *the services provided by the ledger*. We define such transaction as follows:

Definition 6.1 (Application transaction)**.** An application transaction is a transaction created by a user to alter the state of the ledger from an end-user or business perspective.

In Ethereum 1.0, every transaction is an application transaction. We will further use the term *application transaction* or TX_{app} to refer to this type of transaction.

6.3.2 Cross-shard transactions

Each application transaction is processed by a single shard. However, the command in such transaction can lead to state transitions on multiple shards. That is, the state transition function of the shard in which the application transaction is processed creates one or more transaction receipts with state transitions that needs to be executed in other shards.

These transitions are wrapped into a transaction and sent to the right shard. However, as discussed in 4.1.3, these transactions are only valid if the scope in which the transactions were created is valid. If the application transaction is reverted, these transaction receipts should also be reverted. At the same time, all transactions receipts need to be processed in order to say that the application transaction is executed in its entirety and did not leave the overall ledger in an inconsistent state.

To summarise; (i) the owner of a receipt is the *source shard*, (ii) the validity is assessed by its creation being part of the canonical chain, and (iii) each transaction receipt must be processed in the target shard in order for the application transaction to be valid.

Based on these properties, we define *cross-shard transactions* as follows:

Definition 6.2 (Cross-shard transaction)**.** A cross-shard transaction is a transaction created as part of the state transition function of a TX_{app} in a block B in a specific shard addressed to a sibling shard or the beacon chain. A cross-shard transaction is valid if block B is part of the canonical chain.

Note that, a transaction receipt itself could lead to another set of transaction receipts.

6.4 Rethinking Ethereum's gas mechanism

The Ethereum's gas system is a novel mechanism for metering the amount of computational work and to calculate the proportional fee of a transaction. The amount of work is measured in gas which is multiplied by the gas price to get the total transaction costs. This mechanism serves, among others, the following purposes; (i) *incent new validators*, (ii) *protect against denial-of-service attacks*, and (iii) *guarantee smart contract termination*. However, this mechanism does not address the needs of *cross-shard transactions*.

6.4.1 Guaranteed transaction execution

Ethereum's gas mechanism incentivises validators to include transactions with high gas prices. If the gas price of a cross-shard transaction is too low, it may take a long time before it will be processed, i.e. weeks or days. The current design allows gas prices across shards to be volatile which makes it hard to predict the right price for an application transaction involving cross-shard operations. Moreover, even supposing that the gas price is sufficient, the transaction may be vulnerable to censorship attacks or griefing atttacks.

With the current gas mechanism, there is no guarantee that cross-shard transactions eventually will be processed.

6.4.2 Shard overload protection

Ethereum's gas mechanism protects against denial-of-service attacks. The gas mechanism ensures that in times of heavy load, i.e. many transactions that needs to be processed, the gas price will rise. This impedes attackers with a limited money supply from conducting an effective denial-of-service attack.

However, with a guaranteed transaction execution mechanism of cross-shard transactions validators could, whether intentional or not, execute an flooding attack on a single shard. Block producers in different shards could in particularly include transactions which lead to cross-shard transactions addressed to a particular shard. Random sampling does not protect against this attack. Therefore, the protocol should limit the generation of corss-shard transactions addressed to the same shard to protect against the hot shard problem. Then, validators who equivocate from the protocol could be punished through slashing conditions.

6.4.3 The halting problem

Ethereum's gas mechanism guarantees the termination of Turing complete smart contracts. There is an upper limit to the maximum transaction gas consumption specified by the system. In addition, every application transaction contains a field with the maximum amount of gas the user is willing to spend. The consumed gas is collected by the block producer whether or not the the transaction exceeds the allowed gas limit. However, the execution of the transaction command is stopped and reverted once the execution exceeds the allowed gas limit.

The allowed gas limit for application transactions leading to one or more cross-shard transactions is problematic. The provided gas limit needs to be split over a number of cross-shard transactions which all need to be sufficient in order to get executed in its entirety. If one fails, the others may need to get reverted as well.

Chapter 7

Guaranteed-TX

In this chapter we propose Guaranteed-TX, a guaranteed cross-shard transaction execution protocol for Ethereum 2.0. Guaranteed-TX is a protocol that not only guarantees the execution of cross-shard transactions but also significantly improves the cross-shard transaction processing time by allowing cross-shard transaction to be processed before being finalised - a property we refer to as optimistic execution.

The first section provides an overview of Guaranteed-TX and describes it novelties. The second section describes the overall design while the third sections describes its components in more detail. Finally, in the last section, we provide some modifications that trades-off the high message complexity by processing cross-shard transactions in batches.

7.1 Overview of Guaranteed-TX

Guaranteed-TX improves the interoperability between shards by decreasing cross-shard transaction latencies and economically guaranteeing that cross-shard transactions eventually will be processed. Recall from chapter 3 that the block in which a cross-shard transaction is created must be final before the cross-shard transaction is allowed to be processed by the shard it was addressed to. However, the finalisation mechanism only happens at epoch boundaries every 6.4 minutes causing high cross-shard transaction latencies. Guaranteed-TX reduces the transaction latencies by allowing shards to process cross-shard transactions before being finalised. Consequently, a shard must revert a processed cross-shard transaction if the block in which it was created is reverted. This means that a shard must keep track of the blocks of its sibling shards in which the transactions were created.

In addition, Guaranteed-TX requires shards to keep track of the blocks in which the created cross-shard transactions were processed. As a result, shards are able to determine whether a cross-shard transaction was processed and thus able to proof whether a cross-shard transaction was processed within some time period. By adding a slashing condition that a cross-shard transaction must be processed within some time period, Guaranteed-TX economically incentivises validators to process cross-shard transactions.

Guaranteed-TX achieves the above by keeping track of the created and processed cross-shard transactions through a new layer, the messaging layer, in which shards share which cross-shard transactions they created and which they have processed. For every processed cross-shard transaction, there is a ‘created’ and ‘processed’ record. These records are stored from the perspective of a shard. That is, a shard which created a cross-shard transaction in a block stores the hash of the transaction as a TX_{out} record while a shard that processed a cross-shard transaction stores the hash of transaction as a TX_{in} record.

The hash of a *valid* and *processed* cross-shard transaction is thus stored as a TX_{out} record in a block in its source shard and as a TX_{in} record in a block in its target shard. Moreover, both blocks are canonical in the shard to which it belongs to. A key observation is that the hash of a created cross-shard transaction is already stored in trie structure of the receiptsRoot in the source shard, i.e. the TX_{out} record, and in the trie structure of the of transactionRoot in the target shard, i.e. TX_{in} record. Guaranteed-TX separates this information from the original block structure to share this with the other shards. Hence, the introduction of the messaging layer.

Guaranteed-TX novelties

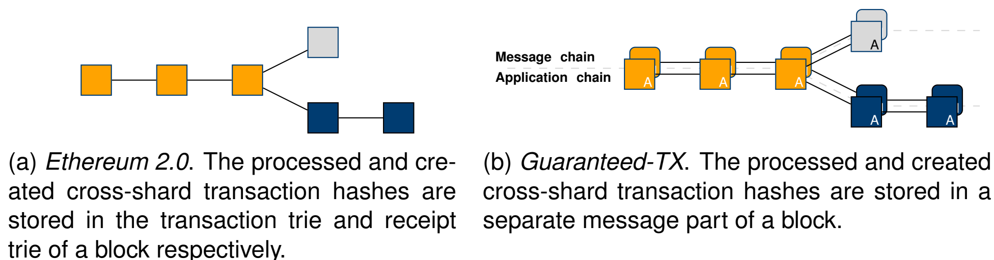
Guaranteed-TX provides the following novelties:

- **Assured delivery:** Once a cross-shard transaction is created and the block in which it was created in the source shard is part of the canonical chain, it is guaranteed to be delivered to the target shard.
- **Optimistic execution:** Once a cross-shard transaction is created and the block in which it was created in the source shard is part of the canonical chain, the target shard may process the transaction before it is finalised in the source shard. Either the created cross-shard transaction remains part of the canonical chain, or the execution of the operation in the target shard is reverted.
- **Economically guaranteed execution:** Once a cross-shard transaction is created and the block in which it was created in the source shard is part of the canonical chain, it is economically guaranteed that the transaction is executed on the target shard. That is, either the transaction is eventually processed, or all the stake of the validators in the shard committee of the addressed shard is slashed.

7.2 High-level design

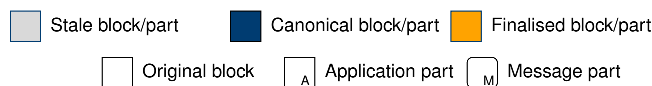
Guaranteed-TX splits the original block structure of Ethereum 2.0 into two parts; an (i) *application* part, and a (ii) cross-shard *message* part. The parts together are equivalent to the original block structure and one part cannot exist without the other. The message part contains the information related to the cross-shard transactions, i.e. the hashes of the created cross-shard transactions and the hashes of the processed cross-shard transactions, together with information concerning the shard, the slot, the block producer and the previous block. The application part contains all other information. Both parts individually contain a hash to the identical part type of the previous created block. That is, both parts can individually be chained together forming a tightly coupled message chain and application chain. Hence, the message chain and application chain are two tightly coupled chains which together correspond to the original blockchain structure. Figure 7.1 visualises the new ledger design.

Throughout the rest of this document we will refer to the matching application and message part together as a *full block*. Once a validator creates a full block, it creates both the application part and the message part. The application part is only sent to the shard-committee, while the message part is disseminated to all other shards. With the message blocks of the other shard, a validator is able to (i) determine the canonical chain of the other shards, (ii) keep record of the created cross-shard transactions in the other shards, and (iii) keep record of the processed cross-shard transactions in the other shards.



(a) *Ethereum 2.0*. The processed and created cross-shard transaction hashes are stored in the transaction trie and receipt trie of a block respectively.

(b) *Guaranteed-TX*. The processed and created cross-shard transaction hashes are stored in a separate message part of a block.



Legend

Figure 7.1: Ledger designs

Guaranteeing safety The validators in a shard committee include cross-shard transactions which are addressed to their shard and registered in a message part by the source shard. However, the message part may become stale if another block at the same height became canonical. In that case, the block in which the cross-shard transactions was processed must be reverted. Guaranteed-TX uses a modified version of the fork choice rule which filters blocks out that are inconsistent with other shards. That is, the fork choice rule filters out blocks that include cross-shard which were created by blocks that are not canonical. The idea of filtering blocks was earlier suggested by V. Zamflir [47]. Note that Guaranteed-TX reverts entire blocks rather than 'correcting' inconsistent transactions. We believe that the PoS consensus model of Ethereum 2.0 will drastically reduce the probability of forking which means that blocks are rarely filtered. Correcting a single inconsistent transaction could cause a domino effect of other reverting events, adding additional complexity. Choosing simplicity over the additional complexity aligns with the *simplicity* design goal principle presented in section 5.1.

Minimising overhead The dissemination of the message part together with the modified fork choice rule allows the validator to include cross-shard transactions before they are actually finalised while satisfying safety. However, it requires a validator to process and store all message parts of the other shards. Storing message parts of other shards which are finalised is unnecessary because these block will never get reverted. Guaranteed-TX modifies the cross-link mechanism that allows a shard to prune all message parts of its siblings shards once they are finalised.

Cross-linking shard blocks Guaranteed-TX uses a cross-linking mechanism in which a block of every shard is cross-linked at the same time through one cross-link record. This modification prevents a block which processed a cross-shard transaction to be finalised while the block in the source shard which created the transaction is not. The block producer of the cross-link record uses the message parts of all shards to include blocks which either processed cross-shard transactions that are finalised or will be finalised by this proposal. Consequently, every shard needs to create an aggregated signature attesting for this proposal.

The validators of a shard committee not only attest that the block in the cross-link is correct, but also that they received all cross-linked transactions addressed to their shard which blocks are finalised by this cross-link proposal. This prevents an attacker from withholding cross-shard transactions from a target shard at a later time to slowly drain their stake.

Guaranteeing cross-shard transaction execution. Guaranteed-TX utilises the finalisation mechanism, i.e. cross-linking mechanism, as a heartbeat system. Every cross-link is considered a heartbeat and shards have to process cross-shard transactions within a number of heartbeats. Once a cross-shard transaction is created, the first cross-link which finalises the block in which it was created proofs that the target shard received the transaction. Then, after every successive heartbeat, a validator is able to pose a challenge that a shard did not process some cross-shard transaction. A correct challenge will drain their stake. For every additional heartbeat, a new challenge can be posed until the transaction is finally processed. Note that once the stake of a validator is below a minimum threshold, it will be removed from the validator pool. Figure 7.2 visualises the heartbeat mechanism.

A major benefit of this approach is that the mechanism is adaptive. In times of network failures or network partitions, blocks may not get cross-linked and validators who did not receive the cross-shard transaction are not punished for not processing these transaction. However, in times of network synchrony, blocks frequently get finalised providing economic guarantees the transaction will be processed.

7.2.1 Cross-shard transactions

Figure 7.3 visualises the transaction flow of a cross-shard transaction. A validator of shard i processes an application transaction TX_{app} which produces a cross-shard transaction TX_{cross} addressed to shard $i+1$. The hash of the created cross-shard transaction is stored in the TX_{out}

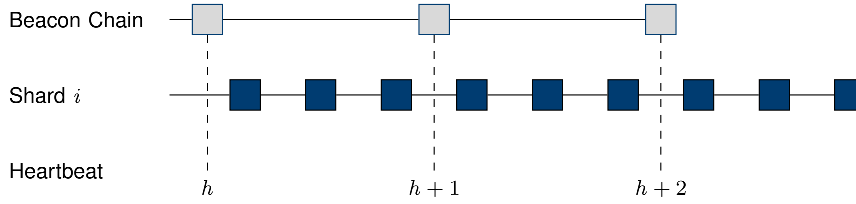


Figure 7.2: Cross-linking heartbeat mechanism

field of the message part. Subsequently, the message part and the created cross-shard transactions are disseminated. Once a validator of shard $i+1$ receives the message part, it processes the transaction which stores the hash of the transaction in the TX_{in} list.

Note that, cross-shard transactions are only sent to the shard to which it is addressed, while message blocks are disseminated to all shards.

Hash uniqueness

A cross-shard transaction is the result of the state transition function of an application transaction TX_{app} on a state S. In order to guarantee uniqueness of a cross-shard transaction, the hash of the cross-shard transaction needs to incorporate the hashes of the state and application transaction. The hash uniqueness of an application transaction is guaranteed by the nonce which is incremented after every transaction. The hash of two valid similar operations, i.e. a user that sends the same amount of tokens to the same person, differs because of the incremented nonce. Similarly, the hash of the state differs if another application transaction modified the state. Thus, the combination of the state and application transaction in the canonical chain is always unique.

Note that, the combination is only unique in the canonical chain and not for all stale blocks.

Multiple receipts

The state transition function of an application transaction may lead to several transaction receipts that need to be processed by distinct shards. Each receipt is packed in a cross-shard transaction addressed to a particular shard. If the block in which the application transaction was created is reverted, i.e. became a stale block, then all created cross-shard transactions related to the application transaction are reverted as well. Consequently, the blocks that processed these cross-shard transactions are filtered out in its siblings shards by the fork choice rule. It makes no difference whether one or more cross-shard transactions are created with regard to safety or correctness.

Processing 'future created' cross-shard transactions

There is one exceptional case which needs extra attention. Let us consider a block b^1_{source} at height h that created a cross-shard transaction TX_{cross} which was processed by a block b^1_{target} at height $h+1$. Subsequently, block b^1_{source} is reverted and block b^1_{target} is filtered out. Then, two slots later, a block b^2_{source} was produced which processed the same application transaction with the same state creating the same transaction TX_{cross}.

If b^2_{source} becomes canonical, then b^1_{target} is not filtered out any more. This leads to situation in which a cross-shard transaction was processed before it was 'created'. While this situation is still safe, it could lead to liveness problems. That is, in order to finalise blocks at slot h, one need to process future blocks at slots height $h+x$.

Guaranteed-TX therefore adds an additional condition that blocks which processes cross-shard transactions at height h while the cross-shard transaction was created at height $h+i$, given $i \geq 0$, are invalid.

7.2.2 Layered architecture

Guaranteed-TX partitions a full block into a message part and an application part. Both parts contain a link to the same part type of the previous block forming a chain. Each chain is considered

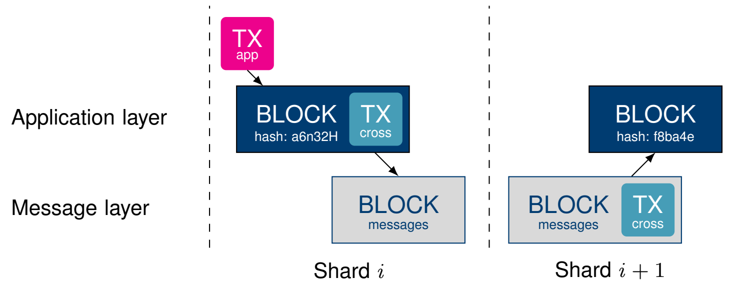


Figure 7.3: Transaction flow of a cross-shard transaction

a separate layer. The (i) *message layer* facilitates which cross-shard messages are created and processed, while the (ii) *application layer* stores all information regarding state, state changes and the world state view..

Figure 7.4 visualises Guaranteed-TX's design from the perspective of a validator in the validator committee of shard 1. The validator processes and stores all application parts and message parts produced in shard 1. In addition, it processes and stores the not finalised message parts of the other shards. With those messages, the validator can determine the canonical chain of all other shards and use that information to determine which cross-shard transactions are valid. Once a cross-link is made, the validator prunes the message blocks of the other shards up till the last finalised cross-linked blocks.

Guaranteed-TX neatly fits Ethereum 2.0 forkful design in which block producers create new blocks upon a previous block without having to wait for a threshold number of confirmations. Every additional block build upon some previous block provides some additional *certainty* that the block will not get reverted. With PoW every additional block represents the amount of computational work put in the chain, while with PoS every additional block represents the increased amount of deposit that is put at stake. Economically finalised blocks will not get reverted and thus have a finality level of 100%, while a block without additional confirmations have high probability of being reverted and thus a low certainty level. Consequently, it is advised to wait for several 'confirmation' parts before a validator will include cross-shard transactions to prevent blocks from being reverted.

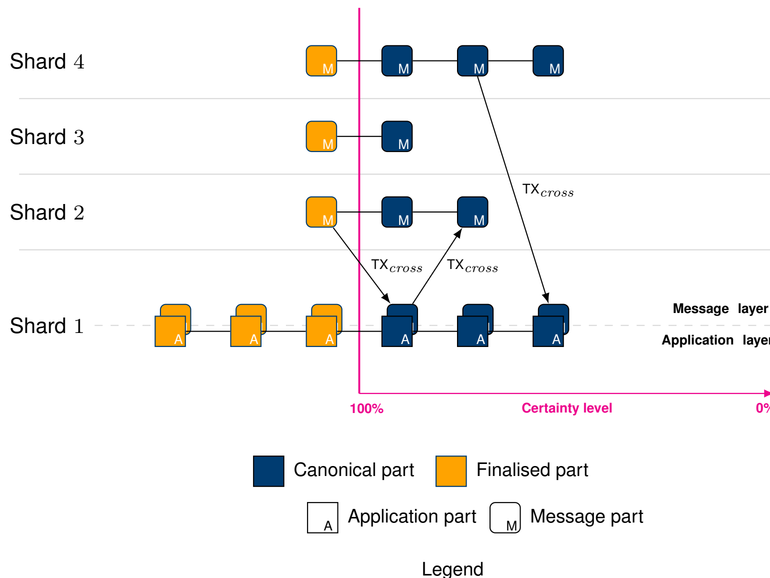


Figure 7.4: *Layered architecture.* The layered design from the perspective of a validator in the shard committee of shard 1. The validator stores all message and application parts of its own shard while it only processes the message parts of it siblings shards. Cross-shard transactions are visualised by an arrow which starts in the message part of the shard in which it was created and ends in the message part of the shard in which it was processed.

7.2.3 Notation and terminology

Let us consider a cross-shard transaction TX_{cross} created in full block B_{full}^{source} in shard i. Then, we say TX_{cross} is;

- *consistent* if B_{full}^{source} is canonical, TX_{cross} is processed in a block B_{full}^{target} on the addressee shard and B_{full}^{target} is canonical as well. Consistent transactions are processed transactions.

- *inconsistent* if B_{full}^{source} is canonical, but TX_{cross} is not processed in a block B_{full}^{target} on the addressee shard or B_{full}^{target} is not canonical, and

- *invalid* if B_{full}^{source} is not canonical.

Moreover, the hash of TX_{cross} in the message part is referred to as:

- TX_{cross}^{out} on the source shard, and

- TX_{cross}^{in} on the target shard.

Hence, TX_{cross} is *consistent* if both TX_{cross}^{out} and TX_{cross}^{in} exists and the blocks in which these records exists are canonical.

We furthermore say that a TX_{cross} is finalised if there exist a finalised heartbeat h cross-linking the B_{full}^{source} or a child block of B_{full}^{source} . That is, the creation of the cross-shard transaction can not be reverted.

Finally, a cross-link proposal h finalises all blocks since the last cross-link proposal $h-1$. Then, the depth of every cross-shard between h and $h-1$ is 1 and the depth of the cross-shard transaction $h-1$ and $h-2$ is 2. Hence, a cross-shard transaction which is not finalised has depth 0.

7.3 Design components

In the following, we describe the core components of Guaranteed-TX in more detail.

7.3.1 Message part

The message part of a full block contains all information related to the cross-shard transactions. It comprises the following fields (i) TX_{out}, which contains the list of created cross-shard transactions in that block, and (ii) TX_{in}, which contains the list of processed cross-shard transactions in that block. Both fields are a list of tuples that include the (i) *shard id* the transaction is addressed to or was created in, and (ii) the *hash* of the cross-shard transaction it self. That is, each shard committee can easily filter cross-shard transactions addressed to their shard.

In addition, the message part compromises a field with the id of the shard, a hash of the related application part, a hash of the previous message part, the slot number, and the signature of the block producer. The validators of a shard committee use these fields to determine the validity of the block. Table 7.1 contain a list with the message part fields.

The *application part* is similar to the current block design of Ethereum. However, the elements of the *Transaction Trie* are the the list of application transactions merged with the list of processed cross-shard transactions. Consequently, the *Transaction Receipt Trie* is the result of the merged execution. For simplicity, we assume that all cross-shard transactions are processed first and subsequently all application transactions.

The hash of the *application part* is included in the *message part*. Consequently, cross-links comprise the hash of the application part to cross-link shard blocks in the beacon chain such that the separation of the messaging part does not have impact on the block header structure. Light clients only need to download block headers.

Shard	uint64	ID of the shard
Slot	uint64	Current slot number
ApplicationChunkHash	Bytes32	Hash of the related application chunk
ParentHash	Bytes32	Hash of parent message block
Signature	Bytes96	Signature of the validator
TXin	Tuple(int16, Bytes32)[]	List of incoming $<$shard id, TX_{cross} hashes $>$
TXout	Tuple(int16, Bytes32)[]	List of outgoing $<$shard id, TX_{cross} hashes $>$

Table 7.1: Message part structure

Validation of message parts Each block produced in a shard i is validated by the shard committee of shard i. However, the validators in the shard committee of shard j, given $j \neq i$, only receive the message part of the produced block. While these validators can verify that the message part was produced by the correct validator assigned to that particular slot in the beacon chain, they cannot verify that the received *message part* of shard i belongs to a valid application block. A malicious validator of shard i could publish a 'fake' message part to fool the shard committee of shard j to let them process malicious transactions.

Recall that in Guaranteed-TX all validators of each shard-committee attest for the same cross-link. The validators of a shard committee will not attest for a malicious message part and malicious behaviour is punished. Consequently, malicious message parts will not get cross-linked. However, this mechanism leaves room for griefing atttacks and double spend attacks. In both cases, the block in the target shard will eventually be filtered out.

A key observation is that the produced message chain of the honest majority will outgrow the malicious produced message chain given that the majority of validators are honest. Moreover, the assignment of validators to a slot in a block is at random. Thus, the probability of having x malicious validator in consecutive slots decreases for every additional slot. In other words, every additional message part increases the probability of being valid. Therefore, a validator should only include cross-shard transactions which have 'sufficient' consecutive confirmations to prevent blocks from being filtered.

Note that the validators of the other shard are able to detect and report malicious behaviour. That is, report that a validator signed multiple message parts of the same slot or message parts produced by a block producer which was not assigned to a particular slot.

7.3.2 Cross-linking

A cross-link is a set of signatures from a shard committee attesting to a block in the shard chain. In Guaranteed-TX all cross-links of all shards are produced at the same time. This way, a block that processes a cross-shard transaction which was created by a block which is not final can be cross-linked at the same time. Otherwise, the block which created the cross-shard transaction must be finalised before the block which processed the cross-shard transaction, which is inefficient for many blocks. In Guaranteed-TX the cross-link process involves the following stages:

1. A validator in the beacon-chain committee proposes a block with a cross-link for every shard.
2. Each shard committee produces an aggregated BLS signature (off-chain).
3. The combined BLS signatures are included in the next block.

The validators in the beacon chain committee validate that the produced cross-link records are correct.

Inconsistent cross-shard transaction trie

A cross-link proposal in the beacon chain comprises a field that contains the cross-shard transaction trie root of all finalised but inconsistent cross-shard transactions. This allows the validators of a shard committee to prune all finalised message parts of its sibling shards. This significantly reduces the

storage overhead while yet being able to proof that a cross-shard transaction was not inconsistent at epoch boundaries.

Cross-link proposal

The validators in the beacon chain committee use the *message* parts of all shards to determine which blocks will be cross-linked. There are two important conditions that apply to the chosen blocks;

1. All the cross-shard transactions in the blocks that will be finalised are valid (either consistent or inconsistent)

2. The blocks in the proposal are produced during the last epoch.

The first condition ensures that invalid transactions will not get finalised. The second condition ensures that only recent block get finalised. This prevents malicious validator from finalising older blocks in order to create 'heartbeats' retroactively. Creating heartbeats retroactively is problematic for liveness and may lead to situations in which validator committees are wrongly punished for not including cross-shard transactions.

Shard overload protection

Guaranteed-TX limits the number of cross-shard transactions to a number X since the last finalisation in order to protect against the Hot-Shard problem discussed in section 6.4.2. Although this mechanism prevents a shard form being overloaded, it is not very efficient. Shards that are more correlated than others will be limited in their performance while other shards without any cross-shard transactions have overcapacity.

Alternatively, one could use the total number of inconsistent transactions addressed to a particular shard between the last two finalised cross-link records in order to determine the available capacity. That is, if the shard still has to process many inconsistent transaction during the last epoch, the maximum capacity of open TX_{in} records is limited. This solution requires more research to determine the impact.

Slashable 1-bit shard vector

The shard overload protection ensures that there is a limit of cross-shard transaction that is allowed to be sent to a single shard. If a shard is overloaded, one can not punish the shard committee for not including all the cross-shard transactions. We therefore add a 1-bit vector to the cross-link proposal. For 1024 shards, this will only add 128 bytes. The position of the bit in the vector equals the number of the shard, e.g. the value of the first bit relates to shard 1. The value represents if the shard did include enough cross-shard transactions between two cross-link records. That is, if the total processed cross-shard transactions is larger than the number of blocks that is going to be finalised times a minimum number of cross-shard transactions. The slashable bit is set to 0 if the shard committee did include enough validators.

7.3.3 Slashing conditions

Guaranteed-TX introduces two additional slashing conditions:

1. **Malicious part production:** Validators that produce malicious message parts are punished by slashing its stake entirely.

2. **Not processing cross-shard transactions:** Validators in a shard committee that do not process cross-shard transactions with a depth of 2 or higher are punished by draining its stake slowly.

The above slashing rules ensure that malicious behaviour is very costly. Although we provide a mechanism to economically guarantee cross-shard transactions by slashing validators, more research is needed to determine the economics of the slashing conditions. Slashing validators too mild may lead to high upper bound processing boundaries, while slashing validator too hard may possibly introduce new types of attacks.

Economic execution guarantee The heartbeat mechanism of Guaranteed-TX ensures that cross-shard transactions are eventually processed. Every consecutive heartbeat increases the depth of a cross-shard transaction.

We highlight three important cross-shard transaction depth values:

- **Depth 0:** A created cross-shard transaction is not finalised.
- **Depth 1:** A created cross-shard transaction is finalised and the validators of the shard committee to which it was addressed attested that they received the transaction.
- **Depth 2:** The validators of the shard committee can be punished if they did not include the cross-shard transaction and the shard was not overloaded.

Every validator is able to create a challenge that a cross-shard transaction was not processed within two heartbeats. A fault proof of an inconsistent transaction compromises the following fields: (i) TX_{cross}, the hash of the inconsistent transaction. (ii) *cross-link hash*, the hash of the cross-link record in the beacon-chain which increased the inconsistent cross-link transaction depth to 1. (iii) *Merkle Tree hashes*, the set of hashes that are needed to verify that TX_{cross} is part of the inconsistent cross-shard transaction trie in the latest cross-link record. (iv) *Validator signature*, the signature of the validator which will receive a reward or get punished for stating an incorrect challenge.

Dependent slashing condition Guaranteed-TX depends on the following slashing conditions in Ethereum 2.0:

1. **Equivocation:** The stake of validators that send contradicting messages is entirely slashable.
2. **Inactivity** The stake of inactive validators will slowly decrease over time.

These conditions ensure that the validators who do not attest for cross-links or maliciously attest against cross-links will get slashed. This prevents the validators from a shard to stall the cross-linking process.

7.3.4 Fork choice rule

Guaranteed-TX uses an adapted version of the *Latest-Message-Driven Greedy Heaviest-Observed Sub-Tree* fork choice rule that takes the canonical chains of the other shards into account. The modified fork choice rule filters blocks that processes invalid cross-shard transactions. Recall that invalid cross-shard transactions are transaction created in a block in the source shard that became stale. Figure 7.5 visualises the filtering of an invalid block. The modified fork choice rule allows a validator to:

1. Keep track of the canonical chains of the other shards using the message parts and the Greedy Heaviest-Observed Sub-Tree rule.
2. Keep a list of all inconsistent transactions addressed to this shard.
3. Filter out blocks which contains invalid transactions.

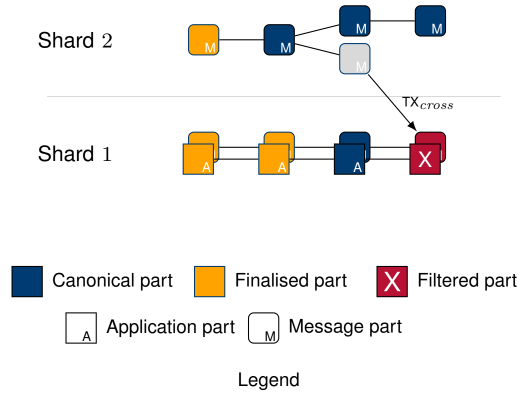


Figure 7.5: *The modified fork choice rule.* Blocks with invalid cross-shard transactions are filtered.

7.3.5 Guaranteed-TX optimisation

In Guaranteed-TX, the hash of every created cross-shard transaction is stored twice; once as a TX_{out} record and once as a TX_{in} record. The message parts in which these records are stored are shared with every other shard and consequently every shard also receives the hashes of the cross-shard transactions which are not created or processed by that shard. In the next chapter, we will show that this results in disproportionate overhead of data storage and data transfer if the distributed ledger comprises many shards and a high cross-shard transaction rate.

Therefore, we propose two modifications which drastically reduces the amount of data shared with each shard.

1. **Batching cross-shard transactions:** Instead of recording and processing the cross-shard transactions created in a block in a message part separately, the created cross-shard transactions addressed to the same shard will be batched. Consequently, the target shard needs to process all cross-shard transactions in one batch, i.e. the created cross-shard transaction in one block, at once. Then, the target shard records the hash of the processed block as a TX_{in} record instead of recording all processed cross-shard transactions separately.

2. **Shard action lists** Instead of sharing a list with the created and processed cross-shard transactions, the message part shares which shards need to take action with regard to the produced block. By using a 1-bit vector of length N, given N shards, Guaranteed-TX maps whether a block created a batch of cross-shard transactions addressed to shard i to the ith item of the bit vector. That is, if the ith item of the vector is set to 1, then the block created a batch of cross-shard transactions addressed to shard i. Consequently, every shard knows whether they need process a batch of cross-shard transactions. This way, the message part is of a constant small size and the shards that need to process a batch of cross-shard transactions can request these from the shard.

Batching cross-shard transactions

Batching cross-shard transactions reduces the amount of data shared with every shard. A batch contains all cross-shard transactions that have been created in a single block that is addressed to the same shard. Then, the hash of the batch, i.e. Merkle trie root hash, is stored in the message part in stead of the hashes of all these transactions.

Batching reduces the storage requirements in the message part if the number of created cross-shard transactions is greater than one. For example, if a block created N cross-shard transactions addressed to the same shard, then it only needs to store the hash of the batch instead of N hashes of all cross-shard transactions.

Consequently, the validators in the target shard to which the batch is addressed needs to verify that they received all created cross-shard transactions by verifying the batch hash. Then, they can process the entire batch at once and store the hash of the block which created the batch to indicate that they processed the batch. Storing the hash of the block rather than the hash of batch has an

additional advantage. A shard can processes multiple batches of a source shard at once and only need to store the latest block hash.

The batching approach is based on the mini block approach of Elrond [50].

Shard action lists

The high overhead costs incurred by Guaranteed-TX are mainly caused by the fact that every cross-shard transaction is shared with every other shard. In a distributed ledger with $N = 1024$ shards, the hash of a cross-shard transaction is unnecessarily shared with $N - 2 = 1022$ shards. The incurred overhead can significantly be reduced if the record keeping of a cross-shard transaction is only shared with the source and related target shard.

Instead of recording which cross-shard transactions were created within one block, one could only signal that some cross-shard transactions were created addressed to some shard. Consequently, each shard is still informed that they have to process some cross-shard transactions, but the precise set with the cross-shard transaction hashes is only shared between the source and target shard. The tuple of outgoing cross-shard transactions in the message part is replaced with a 1-bit vector of length N, given N shards. The ith item in the vector represents whether a batch of cross-shard transactions addresses to shard i was created. In addition, a field with the Merkle Tree root hash of all created cross-shard transaction batches is added to the message part such that a shard can easily verify that a received cross-shard transaction batch was created in a particular block. The size of a 1-bit vector representing 1024 shards and the state root hash of all cross-shard transaction combined requires 160 Bytes, equivalent to the storage size of $\approx$ 5 cross-shard transaction items in the message part list. Another benefit of this approach is that the list of cross-shard transaction in the *message part* does not need to store the shard id to which it was addressed or created.

Obviously, every created cross-shard transaction batch will also be recorded as a processed cross-shard transaction batch. Although the previous optimisation allows to process multiple consecutive batches at once and only requiring to store the hash of the block of the latest processed batch, in a synchronous network many batches will presumably be processed individually. Therefore we reduce the storage overhead of the processed cross-shard transaction records using a similar 1-bit vector. In this case, the ith item in the vector then represents whether a batch of cross-shard created by shard i was processed in the block. The structure op the optimised message part is given in table 7.2.

Note that this solutions adds an extra message round trip between two shards. While the round trip is negligible for the created cross-shard transaction batches, since validators should only include cross-shard transaction batches after some additional confirmation, the extra round trip for processed cross-shard transaction batches could add an extra round trip delay before blocks are filtered. However, as mentioned in section 7.2, we presume that filtering blocks is the exceptional case.

Cross-linking problem

The shard action list optimisation significantly reduces the amount of data shared between shards. However, it makes it impossible for a validator in a shard to propose a cross-link record in which all blocks are safe without downloading all 'processed' cross-shard transaction blocks for every shard. Recall that we can only finalise blocks which processes cross-shard transactions that are or will be finalised in the same cross-link proposal. We then define a 'safe' cross-link proposal in which every finalised block in the proposal only processed finalised or final cross-shard transaction batches.

Although the current design allows to propose a safe cross-link by downloading all processed cross-shard transaction hashes, it involves high costs. We thought of two improvements to reduce this overhead:

- **Epoch boundaries obligations** Once a new epoch starts, each shard first need to process all cross-shard transaction batches of the previous epoch before they are allowed to process cross-shard transaction batches of the new epoch. Consequently, every shards will be synchronised at some block in the next epoch. Every shard could inform the block producer of the cross-link proposal which block it is.

Shard	uint64	ID of the shard
Slot	uint64	Current slot number
ApplicationChunkHash	Bytes32	Hash of the related application chunk
ParentHash	Bytes32	Hash of parent message block
Signature	Bytes96	Signature of the validator
ShardIn	Bytes128	1-bit vector of processed shards
BocksInRoot	Bytes32	Merkle tree root hash of all processed blocks
ShardOut	Bytes128	1-bit vector of cross-shard transaction batches addressed shards
BatchOutRoot	Bytes32	Merkle tree root hash of all created cross-shard transaction batches

Table 7.2: Optimalised message part structure

- **Probabilistic cross-link proposal:** The block producer receives all hashes of the synchronised blocks. After X confirmations, there is some certainty that these blocks will not get reverted. Instead of validating correctness of each block, the block proposer will propose the blocks and let each shard validate if there shard is indeed safe.

These optimisations require further investigation.

Chapter 8

Security and Performance Analysis

This chapter evaluates the proposed solution Guaranteed-TX. We evaluate the *safety* and *liveness* properties and reason about its performance and incurred overhead.

Preliminary note. We will stress once more that Ethereum 2.0 is still under development and many aspects are yet undefined. There are several aspects in Ethereum 2.0 of which the practical impact in the distributed ledger is still unclear. For example:

- *Slashing conditions*. It is unclear how 'slashable' behaviour eventually will be included in the distributed ledger and how likely it is that validators will follow the protocol. However, it seems reasonable to assume that all 'slashable' behaviour eventually will be recorded in the distributed ledger.
- *Fork rate*. It is unclear how the Nakamoto Proof-of-Stake consensus model will perform. The fork rate significantly impacts the performance of Guaranteed-TX. However, it seems reasonable that under normal network conditions the fork rate will be very low. Validators have to attest in a slot for the same block and blocks are not created simultaneously.

8.1 Correctness

In this section we reason about the correctness and liveness properties of Guaranteed-TX. We proof that Guaranteed-TX is safe and that there is *plausible liveness*. In other words, it should not be possible for Guaranteed-TX to get stuck and not be able to finalise anything. Then, under network synchrony and a honest majority, liveness is guaranteed.

Note that Guaranteed-TX does not change the Casper the Friendly Finality Gadget (Casper FFG) consensus protocol. Thus, Casper FFG guarantees that the validators will never commit two conflicting proposals. However, we need proof that finalised proposals never include invalid cross-shard transactions.

8.1.1 Safety

We first define a finalisation f to be *valid* if all N shard committees attested for a cross-link proposal. That is, N times $n_{shard} - f_{shard}$ validators attested in favour of the proposal, given n_{shard} validators in the shard committee of which at most f_{shard} are malicious.

Let us first proof that the modified cross-linking proposal, i.e. one cross-link record that finalises N cross-links at the same time, can not lead to two conflicting proposals at the same height.

Theorem 8.1. *For any valid finalisation f_1 and f_2 in which $f_1.crosslinks[] \neq f_2.crosslinks[]$, then $f_1.height \neq f_2.height$.*

Proof. We proof this theorem by contradiction. Suppose $f_1.crosslinks[] \neq f_2.crosslinks[]$ and $f_1.height = f_2.height$, then there is at least one cross-link for a shard i for which hold $f_1.crosslinks[i] \neq f_2.crosslinks[i]$.

Because a valid finalisation can only be formed if N times $n_{shard} - f_{shard}$ validators attested for the finalisation, then there should be correct validators of shard i that voted in favour for both proposals. This is impossible because a correct validator does not equivocate and attest for two conflicting proposals. □

Let us now proof that an invalid cross-shard transaction can never get finalised.

Lemma 8.2. *For any valid finalisation f and a cross-shard transaction TX_{cross} of which the block which holds the TX^{in}_{cross} record is cross-linked by f. Then, TX^{out}_{cross} is final as well.*

Proof. We proof this lemma by contradiction. Suppose there is a finalisation f that cross-linked a block b which processed TX_{cross} on shard i without the block that holds the TX^{out}_{cross} record being final. Then, $n_{shard} - f_{shard}$ validators of shard i attested for cross-link f. However, this is impossible, because a correct validator would never attest for a cross-link which finalises an invalid transaction, as defined in section 7.3.2. □

Theorem 8.3. *For any cross-shard transaction TX_{cross}, if TX_{cross} is invalid then TX_{cross} is not final.*

Proof. We proof this theorem by contradiction. Suppose TX_{cross} is invalid and final. Then, there should be a finalisation f which finalised the block that stores the TX^{in}_{cross} record but did not not finalise the block that stores the TX^{out}_{cross} record. However, by lemma 8.2, this is impossible. □

8.1.2 Liveness

We have to proof *plausible liveness* of Guaranteed-TX. That is, at any given point in time in which there are valid but not final blocks, there is at least one block that can be finalised. Let us first define a *happen-before* relationship on the creation of blocks, denoted by '$\rightarrow$', as follows; Given two blocks a and b, if block a was created before block b then $a \rightarrow b$ and if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. Obviously, by the properties of physical time, if '$a \rightarrow b$', then block a cannot process cross-shard transactions created by block b.

Let us proof that one can always cross-link new blocks, given that there are non-final blocks. Note that, we consider these blocks to be valid according to the state transition function, slot producer and other inherent rules. We show that there is at least non-final block which is valid with respect to the processed cross-shard transactions.

Theorem 8.4. *Given a set of valid, non-filtered, non-final blocks B and the latest valid finalisation f, then there exist a block $b \in B$ such that all processed cross-shard transactions in b are valid.*

Proof. Consider a set of valid, non-filtered, non-final bocks B, ordered by the *happen-before* relationship, and b_0 being the first block in the set. Then, b_0 processes x cross-shard transactions. If $x = 0$, then b_0 can safely be cross-linked. If $x > 0$, then these transactions are at least valid, otherwise the block would have been filtered and b_0 would not be an item in the set B. Because b_0 is the first block in the set, there is no other block of which b_0 could have include cross-shard transactions, thus these cross-shard transaction are already finalised by f. Consequently, b can safely be finalised. □

A cross-link block producer could repeat the above process until set B is empty, which means that there is no valid non-final block any more that can be finalised. But instead of proposing a cross-link after every block, the block producer would only propose the latest defined cross-link proposal. This proposal, would cross-link every previous block.

8.1.3 Guaranteed-TX simulator

In order to demonstrate the viability of Guaranteed-TX, we implemented a proof-of-concept. The proof-of-concept is designed to verify and visualise the exclusion and inclusion of cross-shard transactions, the validation and invalidation of generated blocks and the cross-linking of shard blocks on the beacon chain. The proof-of-concept provides good insights in the protocol and visualisation how the protocol works in varying circumstances. A detailed description of the visualiser is given in Appendix A.

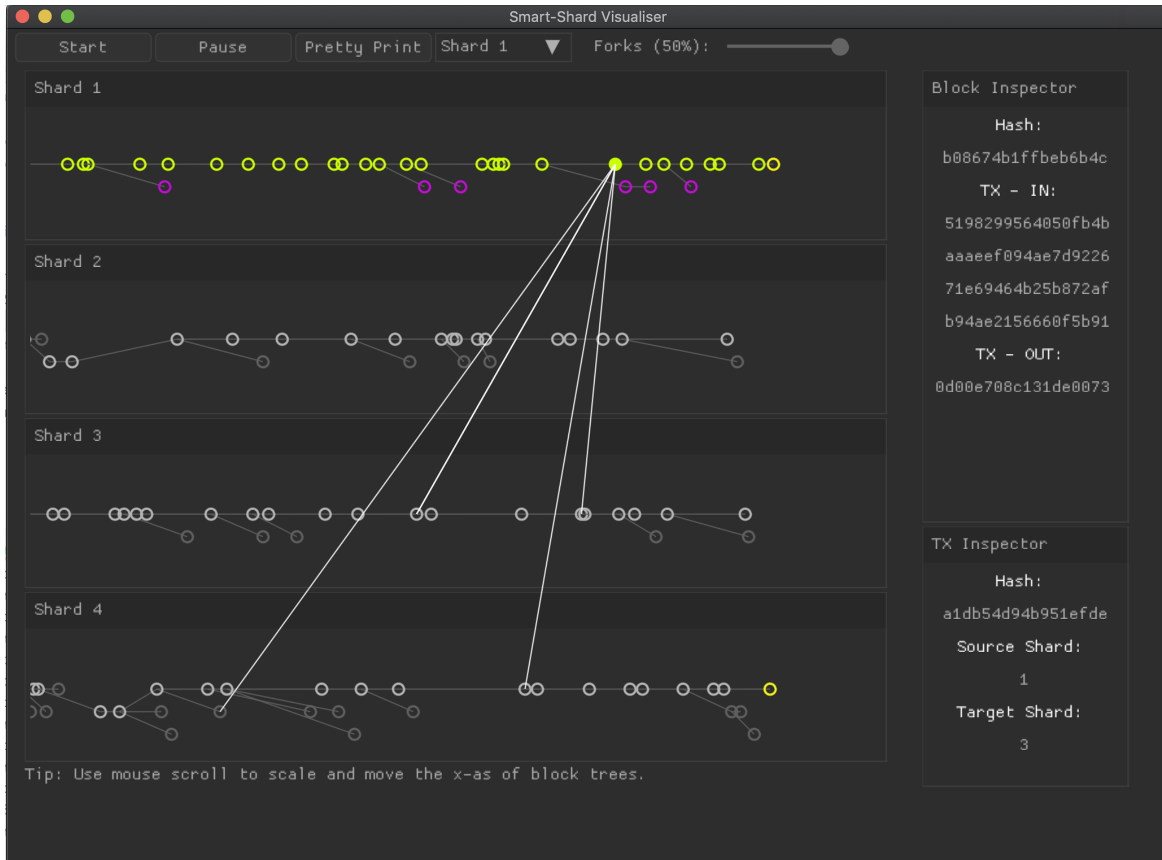


Figure 8.1: Screenshot of the Guaranteed-TX visualiser.

A screenshot of the simulator is shown in figure 8.1. The simulator simulates a number of shards which all produce and process cross-shard transactions. Each circle represents a block and inspecting the block allows one to see which cross-shard transactions the block processed. the *fork* slider at the top of the screen modifies the rate at which forking occurs. Note that there are many more variables that can be modified in the source code. The visualiser helps to better understand how Guaranteed-TX works in changing circumstances.

Findings. One major finding of the simulator was the plausible liveness problem described in section 7.2.1. Once a cross-shard transaction is processed but the block in which it was created is reverted, the block in which it is processed is filtered out. However, a block at a later point in time could recreate the cross-shard transaction, which would re-active the filtered block. Then, in order to finalise the re-activated block, the block producer need to process future blocks. Consequently, the '$\rightarrow$' relation ship does not apply any more and neither does the property that if the fork choice rule favours chain b at time t, then the extend in which it favours b should naturally increase over time.

8.2 Performance

In this section we will reason about the theoretical performance of Guaranteed-TX. Because Ethereum 2.0 is not yet launched and it is unclear what the performance of Ethereum 2.0 will be, we can only reason about Guaranteed-TX's performance. We will first point out which aspects of Ethereum 2.0 have the most impact on the performance of Guaranteed-TX, and subsequently reason about the expected performance.

The performance of Guaranteed-TX heavily depends on the following aspects:

- **Fork rate** A low forking probability within a shard will lead to a low probability of filtered blocks in other shards.

- **Required additional confirmations** Every additional block confirmation before a cross-shard transaction is processed will reduce the probability of a block being reverted, while it increases the average cross-shard transaction processing latencies.
- **Epoch length** A small epoch length results in a small time window between two heartbeats. Consequently, validators can be penalised sooner for not processing cross-shard transactions.
- **Slashing conditions** The more stake is drained for not processing a cross-shard transaction, the more incentive there is for a validator to process cross-shard transactions.

8.2.1 Average transaction rate

The average transaction rate is the number of processed transactions in the canonical chain divided by the time in which these transactions were processed. It is an important metric for distributed ledgers. We will show that the impact of Guaranteed-TX on the average transaction rate is very small.

It is expected that Ethereum 2.0 will have a low fork rate, which means that; (i) the probability of blocks being reverted is low, (ii) the probability of a cross-shard transaction becoming invalid is low, and (iii) the probability of blocks being filtered out is low. Consequently, Guaranteed-TX will not impact the fork rate on other shards given honest validators.

However, as pointed out in 7.3.1, malicious validators could collaborate and create 'fake' message parts to trick other shards for processing malicious transactions. Then, a block in an other shard which processed these malicious transactions could eventually get reverted. Note that we estimate the probability of such attack to be low, because; (i) the malicious message part would never get finalised (ii) the probability of getting caught is high, and (iii) the probability of getting randomly assigned to slots in a row is low. However, it could slightly influence the average transaction rate.

Finally, the additional amount of data, i.e. the message parts, that needs to be send and processed by the other validators in the shard committee is very low. Although the message part includes the hashes of the created and processed cross-shard transactions, these hashes were already shared between the validators in a full block. The only additional data that is shared with the validators in the same shard committee is the information needed to form a chain of all the message parts.

Note that, the *shard action lists* optimisation will add 320 bytes to each block. However, given that the block sizes of Ethereum are 30 KB, this is only an increase of 1%. Thus, we believe that Guaranteed-TX will not influence the time needed to process and attest for a block within one slot.

8.2.2 Transaction latencies

Transaction latency is defined as the time it takes for a transaction to get processed since it was created. Without Guaranteed-TX a cross-shard transaction must first be finalised before a target shard can cross it. Which means that;

1. The average time before a cross-shard transaction can be processed is $6.4/2 \approx 3.2$ minutes
2. The variation between cross-shard transaction latencies is high, i.e. jitter of ± 3.15 minutes
3. The latencies of application transaction with n nested cross-shard transactions is very high, i.e. $n \times 6.4$ minutes

.

Guaranteed-TX significantly improves these latencies;

1. The average cross-shard transaction latency is $m \times slottime$, given m additional required confirmations.
2. The variation between cross-shard transaction latencies caused by Guaranteed-TX is negligible.
3. The latencies of application transaction with n nested cross-shard transactions is proportional to the average cross-shard transaction latencies.

.

8.2.3 Upper bound cross-shard transaction latencies

Guaranteed-TX ensures that cross-shard transactions are eventually processed by penalising validators for not processing cross-shard transactions. However, as we will describe in section 8.4, the tools to define an upper limit on the cross-shard transaction latency are limited. The heartbeat design only allows to drain the validators stake at a slow rate and it takes at least two heartbeats before validators can be punished. Moreover, the upper limit depends on the cross-shard transaction rate addressed to one shard and the possibility to rebalance shards. The hot shard protection mechanism protects the validators for being punished if there are too many cross-shard transactions that the shard can process.

In summary, Guaranteed-TX guarantees the eventual execution of cross-shard transactions and ensures eventual consistency. However, it does not provide any guarantee that a cross-shard transaction is processed within an upper time boundary.

8.3 Overhead

In this section we evaluate the overhead introduced by Guaranteed-TX. We estimate the data transfer costs of a single validator within a time window of one epoch. The estimation measures the data shared between every shard that is used to enable cross-shard transactions. That is, the costs of the message parts and the block headers of every shard shared with every validator.

A key observation is that the sharding model determines whether shards need to share block headers with each other. If a validator in a shard needs to read the result of the state transaction function of some created cross-shard transaction, then it needs to verify that the result if part of the block header. However, as we describe in the next chapter, we believe that 'outer-shard' communication should be limited to messages and shards should only process each other's messages. Consequently, a shard should not have to download the block headers of every other shard.

Therefore, we estimated the incurred data transfer cost for an implementation; (i) with Guaranteed-TX (ii) without Guaranteed-TX, and (iii) Guaranteed-TX with optimisations.

Assumptions

We attempt to make a realistic estimation by assuming a model based on the defined parameters in the Ethereum 2.0 specification and using empirical data of Ethereum 1.0. We assume a model with 1024 shards, 64 slots per epoch and 150 transactions per block. Moreover, we assumes that in every slot in every shard a block is produces that is part of the canonical chain.

Because we do not know how many application transaction will lead to an application transaction, we defined four use cases with different cross-shard transaction rate. We assume that $x\%$ percentage of the application transaction will create one cross-shard transaction. The use cases are given in table 8.1.

Use case	A	B	C	D
Percentage of TX_{app} results in TX_{cross}	0 %	30 %	50 %	80 %
Number of TX_{app} per block	150	115	100	83
Number of TX_{cross} per block	0	35	50	67

Table 8.1: Use cases

8.3.1 Estimations

In this section we estimate the data transfer cost for the three implementation. Note that these are rough estimations.

No Guaranteed-TX We assume a validator in shard i acts as a light client for all other shards. It downloads every block header once the blocks are finalised and subsequently downloads the additional hashes from a full node in order to validate that a cross-shard transaction is indeed part of a particular block. The size of one block header is 508 Bytes and the required the number of hashes to verify that some transaction is part of the transaction Merkle tree in a block header is $\log_2(150) \approx 8$. Thus, the estimated data transfer costs for a validator is the the sum of all the produced block header sizes and in addition the size of 8 hashes times the number of processed cross-shard transactions. The estimated costs are given in table 8.2.

Note that, in this model there is no guarantee that cross-shard transactions eventually will be processed.

Use case	A	B	C	D
Block header cos	32.72 MiB	32.72 MiB	32.72 MiB	32.72 MiB
Transaction costs	0 MiB	0.55 MiB	0.78 MiB	1.04 MiB
Total	31.72 MiB	32.27 MiB	32.50 MiB	32.77 MiB

Table 8.2: Amount of cross-shard transaction data received by one validator during one epoch without Guaranteed-TX

Guaranteed-TX without optimisation We assume a validator in shard i receives all message parts with the created and processed cross-shard transactions of every shard. Because all transactions are already listed in the message part, there is no additional verification required. The size of an empty message part, a message part without any processed or created cross-shard transaction, is 80 Bytes. Then, the costs of the transactions is the sum of all created and processed cross-shard transaction in every shard. The estimated costs are given in table 8.3.

Use case	A	B	C	D
Empty message part costs	5 MiB	5 MiB	5 MiB	5 MiB
Transaction costs message part	0 MiB	214.79 MiB	304.71 MiB	406.41 MiB
Total	5 MiB	32.27 MiB	32.50 MiB	32.77 MiB

Table 8.3: Amount of cross-shard transaction data received by one validator during one epoch with Guaranteed-TX

Guaranteed-TX with optimisation With the optimisations discussed in section 7.3.5, the message part has a fixed costs of 400 bytes. However, a validator has to download a number of additional hashes from a full node in order to verify the batch with cross-shard transactions is part of some message part. The estimated costs are given in 8.4.

Note that, we did not consider the extra costs of finalisation.

Use case	A	B	C	D
Optimised message part costs	24.98 MiB	24.98 MiB	24.98 MiB	24.98 MiB
Transaction batch hashes costs	0 MiB	0.42 MiB	0.80 MiB	1.072 MiB
Total	24.98 MiB	32.27 MiB	32.50 MiB	32.77 MiB

Table 8.4: Amount of cross-shard transaction data received by one validator during one epoch with optimised Guaranteed-TX

8.3.2 Model comparison

The results of the estimations show that the unoptimised version of Guaranteed-TX causes high data transfer overhead. As mentioned before, the disproportionate incurred overhead is caused by the

fact that every cross-shard transaction hash is shared with every other shard. Consequently, a shard receives $N-2$ times as many cross-shard transactions then required. Although the unoptimised version is used to proof correctness, it is not usable in practise.

Another observation is that, at first glance, the optimised version of Guaranteed-TX has a similar data transfer overhead as the version without Guaranteed-TX. However, there are two major differences. A key difference is that Guaranteed-TX only guarantees cross-shard transaction execution while it does not allow validators to retrieve the result of a cross-shard transaction. The structure of the message part does not include the stateRoot hash. Consequently, a distributed ledger implementation that guarantees cross-shard transaction execution and allow for the verification of cross-shard transactions, require a combination of both versions.

The second difference is that Guaranteed-TXrequires that every message part is send to every shard such that a validator in a shard can determine the validity of a cross-shard transaction. In contrast, with no Guaranteed-TX, cross-shard transactions are only allowed to be processed if the creation record is finalised. Instead of downloading all block headers, there may be optimisations possible in which shards only need to download the block headers in which a cross-shard transaction was created.

Furthermore, distributed ledgers communicate over a peer-to-peer network. While one can reduce the amount of data being transferred using direct shard to shard communication, there is no direct connection between them. Consequently, the data shared between two shard still flows over several hops which we did not take into account.

Finally, we point out that the average block size of an Ethereum 1.0 block is around 25 KB. Given that the average size of an Ethereum 2.0 full block is equivalent, then the sum of data transferred of all the full blocks within one shard within one epoch is only $\pm 25KB \times 64\ slots \approx 1.5MB$. This amount of data is significantly smaller than the sum of data of all block headers of every shard or the amount of data shared with the optimised version of Guaranteed-TX. We want to point out that the defined number of shards in the Ethereum 2.0 specification is based on the total number of validators in Ethereum 1.0 divided by an ideal number of validators required in a shard. One could argue that the number of shards may not be ideal for the involved cross-shard communication overhead.

8.4 Known limitations

Guaranteed-TX is a first step towards a guaranteed cross-shard transaction execution protocol. Although we have pointed out many aspects that need further investigation, the basic design poses two major limitations:

Epoch length. The epoch length defines the time between two cross-link proposals. That is, the time between two heart beats. Once a cross-shard is created, it requires one heartbeat to finalise the created block and a consecutive heartbeat before the validator of the target shard can be penalised for not processing the cross-shard transaction. In the current design, the epoch length is set to 64 slots which is respectively 6.4 minutes. Consequently, it can take up to 12 minutes before the validators in the target can be punished. This upper bound may not be sufficient for mission critical applications.

Maximum stake penalisation. There is a limit to the amount of stake that can be drained for not processing a cross-shard transaction. If the costs of being an inactive validator is lower than the cost of not processing a cross-shard transaction, then a rational validator would not sign a cross-link proposal. It would benefit a validator to first process the inconsistent cross-shard transactions and subsequently attest for the next cross-link proposal.

Chapter 9

Sharding model modifications

In chapter 4 and 6 we highlighted the cross-sharding problems for smart contract distributed ledgers. We pointed out that; (i) the existing gas mechanism is inconvenient and requires modifications, (ii) the train and hotel problem is not easily solved, and (iii) balancing shards is an important requirement. While Guaranteed-TX ensures overall consistency between shards, it obliges shards to process cross-shard transactions. However, we did not address how cross-shard transactions must be priced and how Ethereum prevents cross-shard transactions from stalling the system, i.e. the halting problem.

In this chapter we address these problems by proposing three major changes to the Ethereum 2.0 architecture. We first zoom in on two sharding model approaches and subsequently describe our own perspective on sharding. Then, we state the modifications which solve the above described problems.

9.1 Different sharding approaches

In this section we explain the line of thought of two distinct sharding approaches. Recall from section 4.1.4 that sharding improves the overall transaction rate by processing transactions in parallel but that cross-shard operations come with high latencies. Consequently, there is a trade-off between;

- **Performance:** How many application transactions can the ledger process? Does the ledger lock or yank smart contract states, restricting other users to use the smart contract?
- **Usability:** Are cross-shard transactions supported? Is a user obliged to have an account on every shard? Are atomic cross-shard operations supported?
- **Latencies:** How long does a cross-shard transaction take? How long does an application transaction take?
- **Overhead:** What is the computational, storage and communication overhead of a cross-shard operation?

Ethereum 2.0. V. Buterin explained sharding once with the following metaphor [21]; imagine Ethereum being split into 1000 islands. Each island is having its own features and everyone belonging to that island, i.e. the accounts and smart contracts, can freely interact with each other. However, if they want to contact another island, they will have to use some special protocol.

The sharding model envisioned for Ethereum 2.0 assumes that (i) correlated smart contracts are assigned to the same shard, and (ii) cross-sharding is slow and exceptional. This way, Ethereum achieves high performance for intra-shard communication. The independent gas markets for every shard ensures that shards are eventually balanced. Note that Ethereum assumes a static number of shards that can not be rearranged.

Elrond. Elrond's primarily focus is scalability. Elrond dynamically splits the total address space into address ranges which are the shards respectively. These address ranges can be rearranged once validators join or leave. A smart contract in Elrond is automatically dispatched to a shard and a cross-shard transaction requires two smart contracts to be yanked to the same shard. If we assume that many smart contracts invoke each other's methods, then many smart contract must be yanked. Consequently, the performance of a single smart contract is decreased and the yanking introduces high latencies for correlated smart contracts.

Ethereum 2.0 and Guaranteed-TX We strongly believe in the envisioned model of Ethereum 2.0. However, we also believe that cross-shard communication must be fast and shards must be able to rearrange them self. The high incurred latencies of cross-shard payment transactions and smart contract invocations could cause a usability barrier. Moreover, we disagree with the reasoning that validators can only leave the validator pool once new validators joins. We strongly believe that the number of shards must be adapted to the number of active validators.

Take for example the following exceptional but plausible use case; at some point in time a significant number of validators is shut down and can not participate any more, i.e. a validator farm burned down or a country suddenly blocked the network from the internet. Then, not only the security has greatly decreased, but the validators will also be punished for being inactive while they may not even have had a hand in the occurred problem. Consequently, this could scare validators away.

We envision sharding as follows; imagine Ethereum being split into islands and these island being split into cities. The companies, i.e. the smart contracts, belonging to a city can freely interact with each other. However, the communication between companies of different cities is restricted and companies are advised to open a new office in another city if they want to operate freely in that city. If some inhabitant wants to contact an inhabitant of another city, they can communicate faster than before because the islands now have carrier pigeons.

The concept of a city is what we refer to as a smart contract cluster. In our design, smart contracts are assigned to clusters and these clusters are assigned to shards. This allows the ledger to adapt the number of shards to the active number of validators and rearrange the clusters across the shards. Moreover, we limit the communication between clusters to non-turing complete functions. In other words, we limit a cross-shard transaction method to a maximum number of operations and prevent these methods to invoke other cross-shard transactions. Consequently, the gas price and minimum amount of gas required to execute a cross-shard transaction can be specified in the source shard once the application transaction is processed.

We believe that most cross-shard transactions will either (i) transfer tokens, or (ii) invoke smart contract methods, of which the account of the sender is assigned to another cluster than the account of the addressee. Without Guaranteed-TX, these transactions will all incur high latencies.

If some application really needs to interact with multiple clusters, then it is best to deploy a smart contract in each cluster. These smart contracts could have their own Ether balance which could be used to pay for more complex cross-shard method invocations. That is, a cross-shard transaction could invoke a more complex methods which is paid for by the smart contract it self.

9.2 Ethereum modifications

In this section we introduce smart contract clusters and explain the modifications to Ethereum's gas mechanism.

9.2.1 Smart contract clusters

We introduce the concept of smart contract clusters. A smart contract cluster is a range of non-overlapping smart contract addresses. Each smart contract is statically assigned to a particular cluster when being published. A published smart contract can never change its cluster. Moreover, the address range of a smart contract cluster is always processed by a single shard.

The design has three major benefits:

- Correlated smart contracts published in the same cluster benefit from being processed in the same shard.

- Cluster placement in shards can be balanced.
- The number of shards can be adapted to the number of active validators.

Intra-cluster communication

Smart contracts assigned to the same cluster will operate similarly to a non-sharded ledger. That is;

- All state transitions related to an application transaction occur in a single block.
- All modification occur atomically.
- All smart contracts can read/update smart contract state in isolation.

Outer-cluster communication

Instead of having cross-shard communication, we will have cross-cluster communication. Note that, a cluster may or may not be located in the same shard at a particular point in time. We limit cluster communication to non-turing complete methods and specify a maximum amount of gas that can be consumed in that function. This way, cross-cluster transactions have a maximum cost that is used to determine if an application transaction's specified gas limit is sufficient to execute each cross-shard transaction.

9.2.2 Gas mechanism modification

In this section we propose the modifications to the gas mechanism of Ethereum. We distinguish four types of cross-shard transactions and their modified gas price and consumption modifications:

- **Payment transactions:** The gas price and gas consumption of the application transaction is fixed and the source shard can determine whether or not to include the transaction. Once the transaction is included, the target shard must execute the cross-shard transaction.
- **Smart contract invocation (user):** The gas price is fixed but the gas consumption is flexible. If the specified gas limit is above a minimum threshold, the source shard chooses whether or not to include the application transaction. Once a validator includes the transaction, it decreases the consumed gas with the cost of executing a cross-shard call and wraps the method invocation in a cross-shard transaction including its decreased gas consumption. Then, either the consumption is sufficient and the invocation is executed or the invocation is reverted and only the gas is consumed in the target shard. If it is executed successfully, the unconsumed gas can be booked back to the user with another cross-shard transaction.
- **Smart contract invocation (account):** A cross-shard smart contract invocation from another smart contract is limited to non-Turing complete methods. Consequently, the maximum gas consumption of the cross-shard transaction is known beforehand. Thus, one assumes that the cross-shard transaction will consume the maximum amount of gas and the total amount of gas consumed can be calculated in the source shard. Consequently, either the application transaction and all cross-shard transactions are executed or none of them. Note that the unconsumed gas of the cross-shard transaction can be booked back to the user.
- **Gas refunds:** After successfully invoking a smart contract method, the unconsumed gas will be refunded. The only requirement is that the refund is higher than the cost of a cross-shard invocation, otherwise it will not be refunded and marked as spent in the transaction.

Gas price

The modifications require to have a minimum fixed gas price for every shard. This way, a validator of shard i can not include a cheap operation which needs to be executed on another shard. The value of the minimum gas price can be adapted with a cross-linking proposal. Consequently, including cross-shard transactions that do not meet the minimum gas price is considered malicious behaviour.

Gas consumption Modifying the gas consumption mechanism introduces two thresholds:

- **Minimum gas threshold** There is a minimum amount of gas that is required for cross-shard smart contract invocations initiated by a user account.
- **Maximum gas threshold** There is a maximum amount of gas that can be consumed on cross-shard non-turing complete functions. This ensures that the maximum gas amount can be calculated at the invocation of the initial smart contract.

Note that the language for implementing smart contracts need to introduce a new type of functions; non-Turing complete cross-shard methods. These methods are the only methods that can be invoked by smart contracts published in another cluster.

9.3 Discussion

A key observation is that one model is not necessarily superior to the other. It depends on the assumptions of the smart contracts being published on the distributed ledger. If we assume that most smart contracts will operate independently, then it is best to choose for Elrond's sharding model. On the other hand, if we assume correlated accounts, i.e. users accounts and smart contracts, that can be grouped in disjoint sets, the Ethereum's model looks most promising.

However, we believe that most cross-shard transactions will be payment transactions and cross-shard smart contract method invocations. Consequently, these transactions must be fast in order to be functional. In addition, we believe that the umber of shards should be adapted to the number of active validators to ensure adequate security. Then, the Ethereum model with Guaranteed-TX, smart contract clusters and non-turing complete cross-shard operations seems most promising.

Part III

In closing

Chapter 10

Conclusion

In this research, we wanted to find a way to facilitate guaranteed cross-shard transaction execution in Ethereum 2.0. We zoomed in on a problem that has not been studied extensively before. We carried out a thorough examination of the state-of-the-art sharded distributed ledgers and pointed out the challenges that arise with cross-shard transactions. In particular, the problems arising with smart contracts.

We presented Guaranteed-TX, the first guaranteed cross-shard transaction execution protocol for Ethereum 2.0. Guaranteed-TX allows shards to process cross-shard transactions before being finalised in the block it was created - a property called optimistic execution - which significantly improves cross-shard transaction latencies. In order to do so, shards have to keep track of the created cross-shard transactions addressed to their shard. By sharing every created cross-shard transaction, shards can determine which cross-shard transactions are valid and which cross-shard transaction they still need to process. Guaranteed-TX combines cryptoeconomic design and consensus aspects to punish validators for not processing cross-shard transactions if and only if these validators were able to do so.

The evaluation shows that Guaranteed-TX has little impact on the overall performance. While the initial version of Guaranteed-TX is proven to be correct, it is unusable in practice due to the disproportionate incurred data transfer overhead. The optimised version reduces the overhead to an equivalent amount of downloading the block headers of every block of every shard. We stress that this amount is still $30\times$ more than the data shared within a single shard regarding the produced blocks.

While Guaranteed-TX guarantees cross-shard transaction execution, the design restricts itself from defining an upper bound on the 'latency' of cross-shard transaction. If the penalty for not processing cross-shard transactions is too severe, validators will rather face the consequence of being inactive. Consequently, cross-links cannot be made.

In addition, we propose two major modifications in the Ethereum sharding design. By introducing smart contract clusters, correlated smart contracts benefit from being deployed in the same shard while these clusters can be rearranged across shards. Consequently, the number of shards can be adapted to the number of validators. Moreover, we restrict cross-shard transaction invocation from another smart contract to non-Turing complete methods to address the halting problem.

To conclude, Guaranteed-TX is a first step towards a guaranteed cross-shard transaction protocol. It provides a theoretical basis for future protocols.

10.1 Suggestions for further research

We list the following suggestions for further research:

- In our design we assumed that the new consensus model will have a low forking rate. The forking rate has a significant impact on the performance evaluation of Guaranteed-TX. By modelling the Ethereum Proof-of-Stake consensus model one could validate our assumption.
- In our research we did not specify how severe validators will be punished for not processing cross-shard transactions. More research is required to define a cryptoeconomic model with

the punishing rates.

- With the performance evaluation of Ethereum's consensus model and the defined punishment rates, a probabilistic model could be defined to determine the minimum number of confirmations before a cross-shard transaction can be included.
- The optimised version of Guaranteed-TX significantly reduces the amount of transferred data. Unfortunately, we were not able to provide optimisation for the cross-linking mechanism. Further research is required to improve the cross-linking mechanism.
- The design of Guaranteed-TX incurs additional overhead to form a chain of message parts. This additional overhead may be reduced by combining information from the block headers. This optimisation could reduce the amount of data transferred.
- The current shard overload protection limits the number of cross-shard transactions based on a quota for every shard. This design limits the performance if some shards are more correlated than others. Further research in shard overload protection mechanisms could lead to a more optimal performance.

References

[1] Daniel J Abadi. Consistency Tradeoffs in Modern Distributed Database System Design. page 6. URL: http://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf.

[2] Ittai Abraham. The threshold adversary. URL: https://ittaiab.github.io/2019-06-17-the-threshold-adversary/.

[3] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR Fault Tolerance for Cooperative Services. page 14. URL: http://www.cs.cornell.edu/lorenzo/papers/sosp05.pdf.

[4] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A Sharded Smart Contracts Platform. page arXiv:1708.03778. arXiv:1708.03778.

[5] Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling byzantine fault-tolerant replication towide area networks. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 105–114. IEEE. URL: https://www.researchgate.net/publication/220958091_Scaling_Byzantine_Fault-Tolerant_Replication_toWide_Area_Networks.

[6] BitFury Group. Block Size Increase. URL: https://bitfury.com/content/downloads/block-size-1.1.1.pdf.

[7] Blockchain.com. Transaction Rate. URL: https://www.blockchain.com/charts/transactions-per-second.

[8] Vitalek Buterin. Immediate message-driven GHOST as FFG fork choice rule. URL: https://ethresear.ch/t/immediate-message-driven-ghost-as-ffg-fork-choice-rule/2561.

[9] Vitalek Buterin. In favor of forkfulness. URL: https://ethresear.ch/t/in-favor-of-forkfulness/1225.

[10] Vitalek Buterin. Merge blocks and synchronous cross-shard state execution. URL: https://ethresear.ch/t/merge-blocks-and-synchronous-cross-shard-state-execution/1240.

[11] Vitalik Buterin. Casper CBC and Ethereum 2.0. URL: https://www.youtube.com/watch?v=EoNQUGsPvuU&feature=youtu.be.

[12] Vitalik Buterin. The Meaning of Decentralization. URL: https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274.

[13] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. page 14. URL: http://pmg.csail.mit.edu/papers/osdi99.pdf.

[14] Cbeci.org. Cambridge Bitcoin Electricity Consumption Index (CBECI). URL: https://www.cbeci.org/.

[15] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. URL: https://www.usenix.org/legacy/events/nsdi09/tech/full_papers/clement/clement.pdf.

[16] Jordan Clifford. Understanding the Block Size Debate. URL: https://medium.com/scalar-capital/understanding-the-block-size-debate-351bdbaaa38.

[17] Tyler Cowen. Bitcoin Is (Probably) Here to Stay. URL: https://www.bloomberg.com/opinion/articles/2019-06-28/bitcoin-s-rise-shows-crypto-is-alive-and-well.

[18] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On Scaling Decentralized Blockchains: (A Position Paper). In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, volume 9604, pages 106–125. Springer Berlin Heidelberg. URL: http://link.springer.com/10.1007/978-3-662-53357-4_8, doi:10.1007/978-3-662-53357-4_8.

[19] Christian Decker and Roger Wattenhofer. Information Propagation in the Bitcoin Network. URL: https://ieeexplore-ieee-org.ezproxy2.utwente.nl/document/6688704.

[20] Casey Detrio. Synchronous cross-shard transactions with consolidated concurrency control and consensus (or how I rediscovered Chain Fibers). URL: https://ethresear.ch/t/synchronous-cross-shard-transactions-with-consolidated-concurrency-control-and-consensus-or-how-i-rediscovered-chain-fibers/2318.

[21] District0x.io. Ethereum Sharding Explained. URL: https://education.district0x.io/general-topics/understanding-ethereum/ethereum-sharding-explained/.

[22] Dr. Gravinwood. Polkadot: A vision for a heterogeneous multi-chain framework. URL: https://polkadot.network/PolkaDotPaper.pdf.

[23] Justin Drake. 1-bit aggregation-friendly custody bonds. URL: https://ethresear.ch/t/1-bit-aggregation-friendly-custody-bonds/2236.

[24] Mark Drake. Understanding Database Sharding. URL: https://www.digitalocean.com/community/tutorials/understanding-database-sharding.

[25] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. 35(2):288–323. URL: http://portal.acm.org/citation.cfm?doid=42282.42283, doi:10.1145/42282.42283.

[26] Etherscan.io. Ethereum Sync (Default) Chart. URL: https://etherscan.io/chartsync/chaindefault?source=post_page.

[27] J Fischer and A Lynch. Impossibility of Distributed Consensuswith One Faulty Process. page 9. URL: https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf.

[28] Daniel Gallas. Politicians suspected in bribery scandal. URL: https://www.bbc.com/news/world-latin-america-41109132.

[29] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in Bitcoin and Ethereum Networks. page 18. URL: https://fc18.ifca.ai/preproceedings/75.pdf.

[30] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A Scalable and Decentralized Trust Infrastructure. URL: https://research.vmware.com/publications/sbft-a-scalable-decentralized-trust-infrastructure-for-blockchains.

[31] Adrianne Jeffries. 'Blockchain' is meaningless. URL: https://www.theverge.com/2018/3/7/17091766/blockchain-bitcoin-ethereum-cryptocurrency-meaning.

[32] Raul Jordan. Ethereum 2.0 Development Update #32 — Prysmatic Labs. URL: https://medium.com/prysmatic-labs/ethereum-2-0-development-update-32-prysmatic-labs-1fce63459403.

[33] P Jovanovic. ByzCoin: Securely Scaling Blockchains. URL: http://hackingdistributed.com/2016/08/04/byzcoin/.

[34] Jeff Kauflin. Why Everyone In Crypto Is Talking About DeFi. URL: https://www.forbes.com/sites/jeffkauflin/2019/04/26/why-everyone-in-crypto-is-talking-about-defi/.

[35] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A secure, Scale-Out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. doi:10.1109/SP.2018.000-5.

[36] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE. URL: https://eprint.iacr.org/2017/406.pdf.

[37] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM. URL: http://www.cs.cornell.edu/lorenzo/papers/kotla07Zyzzyva.pdf.

[38] Jae Kwon and Ethan Buchman. Cosmos - A Network of Distributed Ledgers. URL: https://cosmos.network/cosmos-whitepaper.pdf.

[39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 21(7):558–565. URL: http://portal.acm.org/citation.cfm?doid=359545.359563, doi:10.1145/359545.359563.

[40] Piper Merriam and Vitalek Buterin. Cross-shard contract yanking. URL: https://ethresear.ch/t/cross-shard-contract-yanking/1450.

[41] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. page 9. URL: https://bitcoin.org/bitcoin.pdf.

[42] Illia Polosukhin. NEAR Protocol. URL: https://nearprotocol.com/blog/near-ai-near-protocol.

[43] QuarkChain Foundation. Countdown to QuarkChain Mainnet Singularity V1.0: A New Era Of The Next Generation of Blockchain Universe. URL: https://medium.com/quarkchain-official/countdown-to-quarkchain-mainnet-singularity-v1-0-3b21f2d70177.

[44] QuarkChain Foundation. QuarkChain - A High-Capacity Peer-to-Peer Transactional System. URL: https://quarkchain.io/.

[45] Alex Skidanov and Illia Polosukhin. Nightshade: Near Protocol Sharding Design. page 39. URL: https://nearprotocol.com/downloads/Nightshade.pdf.

[46] Alexander Skidanov. How Unrealistic is Bribing Frequently Rotated Validators? URL: https://medium.com/nearprotocol/how-unrealistic-is-bribing-frequently-rotated-validators-d859adb464c8.

[47] Alexander Skidanov. So what exactly is Vlad's Sharding PoC doing? URL: https://medium.com/nearprotocol/so-what-exactly-is-vlads-sharding-poc-doing-37e538177ed9.

[48] Derek Sorensen. To Clear Things Up: CBC Casper Liveness Issue. URL: https://medium.com/pyrofex/cbc-casper-proof-of-stake-consensus-algorithm-liveness-issue-c965e88d163e.

[49] Zephyr Teachout. The Problem of Monopolies & Corporate Public Corruption. URL: https://www.amacad.org/publication/problem-monopolies-corporate-public-corruption.

[50] The Elrond Team. Architecture overview. URL: https://docs.elrond.com/learn/architecture-overview-1.

[51] The Elrond Team. Elrond - A Highly Scalable Public Blockchain via Adaptive State Sharding and Secure Proof of Stake. URL: https://elrond.com/files/Elrond_Whitepaper_EN.pdf.

[52] The Elrond Team. Elrond vs. Quarkchain. URL: https://docs.elrond.com/detalied-comparative-analysis/elrond-vs.-quarkchain.

[53] The Ethereum Foundation. CBC Casper FAQ. URL: https://github.com/ethereum/cbc-casper.

[54] The Ethereum Foundation. Ethereum 2.0 (Serenity) Phases. URL: https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/eth-2.0-phases/.

[55] The Ethereum Foundation. The Ethereum Wiki. URL: https://github.com/ethereum/wiki/wiki/.

[56] The Zilliqa Team. The ZILLIQA Technical Whitepaper. URL: https://docs.zilliqa.com/whitepaper.pdf.

[57] Erik Trautman. The Beginner's Guide to the NEAR Blockchain. URL: https://nearprotocol.com/blog/the-beginners-guide-to-the-near-blockchain.

[58] Trustnodes.com. Ethereum 2.0 Planned For Launch on the 3rd of January 2020. URL: https://www.trustnodes.com/2019/06/15/ethereum-2-0-planned-for-launch-on-the-3rd-of-january-2020.

[59] Luca Ventura. Global Finance Magazine - Wealth Distribution and Income Inequality by Country 2018. URL: https://www.gfmag.com/global-data/economic-data/wealth-distribution-income-inequality.

[60] Hsiao-Wei Wang. Ethereum Sharding: Overview and Finality. URL: https://medium.com/@icebearhww/ethereum-sharding-and-finality-65248951f649.

[61] Dr Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. page 39. URL: https://gavwood.com/paper.pdf.

[62] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus in the Lens of Blockchain. URL: http://arxiv.org/abs/1803.05069v6.

[63] Linda Yueh. Enriching the rich in the US? URL: https://www.bbc.com/news/business-31377189.

[64] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling Blockchain via Full Sharding. In *Runchao Han's Blog*. URL: http://SebastianElvis.github.io/2018/11/17/CCS-18-RapidChain-Scaling-Blockchain-via-Full-Sharding/index.html.

[65] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert van Renesse. REM: Resource-Efficient Mining for Blockchains. URL: http://eprint.iacr.org/2017/179.

[66] Zilliqa.com. Zilliqa is a scalable, secure public blockchain platform. URL: https://zilliqa.com/.

Glossary

beacon chain A system chain providing basic functionality that is not specific to any particular shard. 29, 30, 36, 52, 63, 64

Byzantine fault tolerance The dependability of a fault-tolerant computer system to Byzantine faults. 14, 89, 91

canonical chain The chain which is agreed to be the main chain. 25, 72, 89

dirty read A dirty read is occurs when a transaction reads data that has not yet been committed. 39

distributed ledger technology A general term used to describe the family of technologies deriving from or built to support distributed ledgers. 7, 91

fork choice rule A rule that determines the canonical chain in forkful distributed ledgers. 25, 26, 30, 59, 60, 65, 66, 71, 89

fork-free A distributed ledger classification in which each block in the chain only has one child block. 26

forkful A distributed ledger classification in which a block in the chain can have multiple child block. 26, 53, 89

full sharding A sharding approach that partitions state storage, transaction partitioning and network communication. 36

gas Unit of computational effort. 38, 55

gas price Unit of computational effort plural. 55

griefing atttack An attack that does not benefit the attacker but causes grief to the victim. 55, 63

latest-message-driven greedy heaviest-observed sub-tree A fork choice rule that ranks blocks by the number of blocks of its subtrees and the heaviest observed subtree is considered part of the canonical chain. 26–28, 65, 91

longest chain rule A fork choice rule that ranks blocks by the chain length and the longest chain is considered the canonical chain. 26, 27

practical Byzantine fault tolerance An consensus algorithm that optimises aspects of Byzantine fault tolerance. 22, 91

proof-of-elapsed-time A lottery election based on a random idle time. 25, 91

proof-of-stake A lottery election based on invested stake. 26, 91

proof-of-work A lottery election based on a cryptographic puzzle. 25, 91

stale block A block that is valid but not included in the canonical chain. 26, 53, 60

two-phase commit A type of atomic commitment protocol. 91

unspent transaction output An output of a transaction that has not been spent and can be used as an input of a new transaction. 91

Acronyms

2PC Two-phase commit. *Glossary:* two-phase commit

BFT Byzantine fault tolerance. 14, 20–23, 26, 30, 32–34, 42, 46, 47, *Glossary:* Byzantine fault tolerance

Casper CBC Casper Correct-by-Construction. 30

Casper FFG Casper the Friendly Finality Gadget. 30, 69

DLT Distributed Ledger Technology. 7, 13–16, 20, 22, 32, 36, 40, 41, 43, 46, *Glossary:* distributed ledger technology

GHOST Greedy Heaviest-Observed Sub-Tree. 26–28, 65, *Glossary:* latest-message-driven greedy heaviest-observed sub-tree

IMD Immediate-Message-Driven. 28

LMD Latest-Message-Driven. 27, 28, 65, *Glossary:* latest-message-driven greedy heaviest-observed sub-tree

pBFT Practical Byzantine Fault Tolerance. 22–24, *Glossary:* practical Byzantine fault tolerance

PoET Proof-of-elapsed-Time. 25, 26, *Glossary:* proof-of-elapsed-time

PoS Proof-of-Stake. 26, 30, 32, 33, 47, 59, 61, *Glossary:* proof-of-stake

PoW Proof-of-Work. 25, 26, 29, 30, 32, 46, 61, *Glossary:* proof-of-work

UXTO Unspent Transaction Output. *Glossary:* unspent transaction output

Appendices

Appendix A

Guaranteed-TX Simulator

The section describes the Guaranteed-TX simulator in more detail. The simulator was designed to get better insights in the guaranteed cross-shard transaction execution protocol and visualises the operation of Guaranteed-TX in varying circumstances. The simulator is build upon several abstractions which among other thing will be discussed. The source-code has been published on GitHub. The repository is located at https://github.com/sjoerdwels/Guaranteed-TX.

A.1 Implementation

The Guaranteed-TX simulator is an application that simulates a sharded ledger that evolves over time. In the simulation, a number of shards independently create blocks that create cross-shard transactions. The simulation has many variables that can be changed, among those are properties such as the block generation period, finalisation period, and probability of creating forks. This simulation is visualised in a graphical user interface.

The simulator is written in Golang because of the efficient concurrency primitives Go provides and the highly concurrent work of the simulator requires. Golang however, has no build-in GUI library. The visualiser therefore uses Go bindings for Nuklear.h, a small ANSI C GUI library, to create graphical windows and render visual elements to visualise the shards, created blocks and relation between those blocks.

A.2 Compiling the repository

In order to compile the source code, the build machine requires Golang (version $> 1.4+$), cloned the Nuklear.h Go bindings library and a gcc compiler to compile Nuklear.h. A more detailed compile instruction can be found in the README.MD of the repository.

A.3 Simulation abstractions

The simulator is build upon several abstractions.

Validators, network latency and malicious behaviour. We have abstracted from validators in a shard by making the shard a self-contained entity that by randomness builds upon a block. In order to simulate delays within shard committees and simulate malicious behaviour in which validators intentionally build upon previous blocks, we randomly build upon one of the last produced blocks in a shard. The *fork* slider determines the probability the shard builds upon the latest block.

Application transactions. We have abstracted from application transactions and created random cross-shard transactions using the current time as data input. In order to prevent that every cross-shard transaction is included in a block, we randomly choose which cross-shard transactions are

included in a block. In addition, the siimulator randomly creates and removes cross-shard transactions from the validator pool. Consequently, some cross-shard transactions are only processed in a canonical or stale block. T

Beacon chain. We abstracted from all functionality in the beacon chain with the exception of cross-linking shard blocks. Cross-links are created at random and finalise previous block. Note that, the shard do not prune the message parts of previous blocks because in order to visualise how cross-shard transactions are included.

A.4 Usage

Table A.1 describes the controls to interact with the visualiser.

Control	Corresponding event
Start button	Start the simulator.
Stop button	Pause the simulator.
Pretty Print button	Pretty Print all created blocks of the selected shard in the terminal.
Shard drop-down	Select the visualised shard.
Fork probability slider	Probability that a generated block extends the canonical chain.
Finalise speed slider	Relative time needed to cross-link shard blocks on the bacon chain.
Vertical scroll	Move all
Horizontal scroll	Scale
Block on-click	Open the block inspector of the selected block.
Transaction on-click	Open the transaction inspector of the selected transaction.

Table A.1: Guaranteed-TX visualiser controls and corresponding events.