Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Asynchronous Consensus-Free Transaction Systems

Bachelor's Thesis

Max Mathys

mmathys@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Roland Schmid, Jakub Sliwinski
Prof. Dr. Roger Wattenhofer

February 18, 2021

Acknowledgements

I wish to express my sincere gratitude to my supervisors Roland and Jakub who not only always provided their valuable feedback and expertise during the weekly meetings, but also supported me in the development process of this thesis. Also I would like to thank Prof. Dr. Roger Wattenhofer and the distributed computing group for this great opportunity in the first place; contributing to building a faster form of a permissioned blockchain system was challenging and rewarding at the same time.

Abstract

Permissioned blockchain-based transaction systems assume that it is required to solve the consensus problem, which prevents scalability. This makes blockchain-based transaction systems unsuitable for everyday use cases, such as for example retail payments.

We present a scalable transaction system that is asynchronous and does not require the consensus problem to be solved. We show that the system is scalable and has the necessary properties to be used in everyday use cases.

Contents

Chapter 1

Introduction

1.1 General

Permissioned blockchain-based transaction systems are popular because they provide a reliable, resilient and fault-tolerant way of transferring funds between participants. Blockchain-based transaction systems have however several drawbacks: (i) their throughput is limited; (ii) transactions are not final; (iii) confirmation time can be prohibitively high. These drawbacks make blockchain-based transaction systems not suitable for use cases such as retail payments where high throughput, final transactions and low confirmation time are required.

We present a protocol and an implementation that overcomes these drawbacks. The protocol is byzantine fault-tolerant, has instant confirmation time, very high throughput and transaction finality. We show that the protocol and implementation are efficient and suitable for use cases such as retail payments or real-time gross settlement systems (inter-bank payments).

Blockchain-based transaction systems aim to solve the *consensus problem* in a permissioned or unpermissioned setting. This yields the previously mentioned drawbacks; the protocol of this thesis does not solve the consensus problem while still guaranteeing a correct payment system.

1.2 Related Works

Gupta. Gupta [1] proposes an approach of constructing a decentralized financial transaction processing model in which a static set of validators verify transactions. A majority signature based replicated storage protocol is used for transaction authorization. Gupta [1] also provides a way to create verifiable audit trails.

Apart from the validators being in a permissioned setting, the model is distributed and permissionless. The model works without full consensus and is thus faster than previous models for distributed transaction models such as

blockchains. The validators are byzantine fault-tolerant.

This thesis proposes a model, a specific protocol, and an implementation of a consensus-free transaction system that satisfies the same properties as Gupta [1]: distributed, consensus-free and final. Moreover, sharding is introduced in 4.6, allowing our model to scale horizontally without loss of efficiency.

ABC. ABC [2] is a permissionless blockchain architecture that can be used as a transaction system. It validates transactions using proof of stake that is much more efficient than Bitcoin's proof of work. The architecture does not need to solve full consensus, but rather relies on a relaxed form of consensus called *ABC Consensus*.

ABC is completely permissionless, making it an attractive solution for a cryptocurrency protocol. ABC does not use proof of work, it is very efficient and economical to use. ABC transactions are final, asynchronous, deterministic, and parallelizable.

This thesis applies the concepts of ABC and improves them. An actual implementation is provided and evaluated. Further, the thesis determines whether such a protocol is fast enough for usage in the real world.

FastPay. FastPay is concurrent work.

Given a set of fixed authorities, FastPay [3] provides a protocol and an implementation of a settlement system. The authorities are in a permissioned setting, like Gupta [1]. The protocol is byzantine fault-tolerant for the authorities. It also does not need to solve full consensus, making the protocol fast. The authors present both an implementation and benchmarks.

On a reference instance, the FastPay implementation can process up to 80,000 transactions per second with 20 authorities while satisfying properties such as asynchrony and finality.

FastPay is more complicated than the protocol is this thesis. Special attention has to be given to the transaction and confirmation order. The FastPay paper mentions that sharding is process-based and comes with limitations. The performance is not reported clearly: benchmarks are divided into *transaction* and *confirmation*, whereas one is demonstrated not to scale. Finally, the evaluation is performed with an unjustifiably small load and few authorities. The protocol in this thesis scales perfectly. That is, when the number of shards is increased k-fold, the maximum throughput increases k-fold as well.

FastPay does not augment itself with any specific efficiency improvements. The implementation in this thesis makes use of Merkle signatures and BLS threshold signatures, as explained in 4.2 and 4.3: Merkle signatures offer a significant speedup while providing the very same guarantees. BLS threshold signatures

enable the system to have a constant verification time no matter how many authorities/validators there are in the system.

Chapter 2

Model

Participants. There are two types of participants: *clients* and *validators*.

Clients can hold money and can send money to any other client. Each client generates a public/private key pair. Clients are identified by their public key ("address").

A set of validators *validate* transactions. Each validator holds a public/private key pair. The public key of each validator is known to each client.

Byzantine Fault tolerance. The system is byzantine fault-tolerant. Up to $\frac{1}{3}$ of the validators can be crashing or be byzantine. It is assumed that a system has $n = 3f + 1$ validators where at most f validators are byzantine.

Adversary. A completely asynchronous network model is assumed. An adversary knows the protocol and can control the behavior of the entire network, including blocking, dropping, delaying, replaying and reordering messages. However, an adversary does not know the private key of any honest participant.

Moreover, it is assumed that the cryptographic operations used in the protocol are secure. Cryptographic operations include hashing, signing, verification and BLS signature aggregation.

Chapter 3

Protocol

3.1 General

Validators *verify* and *sign* transactions. In a system, there is a fixed number of validators. All validators are known to each client. Validators do not have to exchange messages with each other. The keys for all validators are generated beforehand and are securely distributed.

A *validator quorum system* S is a quorum system where *validator quorum* Q is any subset of the validators and $|Q| \geq \frac{2}{3}n$, n being the number of validators; $f < \frac{1}{3}n$ where f is the number of byzantine validators.

Definition 3.1 (f-disseminating)**.** A quorum system S is f-disseminating if (1) the intersection of two different quorums always contains $f + 1$ nodes, and (2) for any set of f byzantine nodes, there is at least one quorum without byzantine nodes. [4]

The validator quorum system is f-disseminating. This means that the intersection of any two validator quorums contains at least one honest validator.

Any validator quorum can produce *validator quorum signatures* by using a threshold signature scheme. A valid validator quorum signature can be produced by aggregating $> \frac{2}{3}n$ of validator signatures from distinct validators where n is the total number of validators.

An **output** is the base form of information. An output contains the following fields:

- an *owner* (client), denoted by its public key (address);
- a *value* (amount of currency);
- an *identifier* that uniquely identifies the output; and
- a *signature* from a validator quorum. This is a threshold signature that is constructed by aggregating signatures from all validators in any validator quorum.

A client holds currency in the form of *unspent transaction outputs* (UTXO). These are outputs that have not been marked as "spent" by the validators. A UTXO can only be spent by its *owner*.

3.2 Transactions

If a client wishes to transfer funds to another client, the client *spends* a UTXO by contacting all validators in a validator quorum and finally forwards a new set of valid UTXOs to the receiving client.

Example transaction. An example transaction is visualized in figure 3.1. Let A, B be clients, $\{O, v, id\}_S$ be an UTXO with owner O, amount v, output identifier id, signed by a validator quorum that yields the threshold signature S. Suppose that A owns a valid UTXO $\{A, 100, id_1\}_S$ and wants to send B 40 units.

A uses $\{A, 100, id_1\}_S$ as an input for his transaction. A specifies a new set of outputs $\{B, 40, id_2\}$ and $\{A, 60, id_3\}$ and obtains signatures for the outputs from every validator of a validator quorum.

For each output, A combines the received signatures $\{s_i\}$ from each validator into a single threshold signature S. The client is only able to combine the signatures into a threshold signature S if there are $> \frac{2}{3}n$ validator signatures.

Finally, the client A forwards the signed UTXO outputs to the client B. By checking the signature S of the output, B can verify that the received UTXO is valid.

3.3 Correctness

Double-spending. Double-spending is prevented in this system even when validators are byzantine. To spend a UTXO twice, a client has to get a signature for the same output from any two validator quorums. Because the validator quorum system is f-disseminating, the intersection of the two quorums contains at least one honest validator. The honest validator rejects the attempt to spend an output more than once. The client is not able to create a valid threshold signature.

Finality When a client receives a payment in the form of a UTXO signed by a validator quorum, it can consider the transaction as final: because our quorum system is f-disseminating, there is at least one quorum with only honest nodes. This quorum will accept the UTXO as valid input.

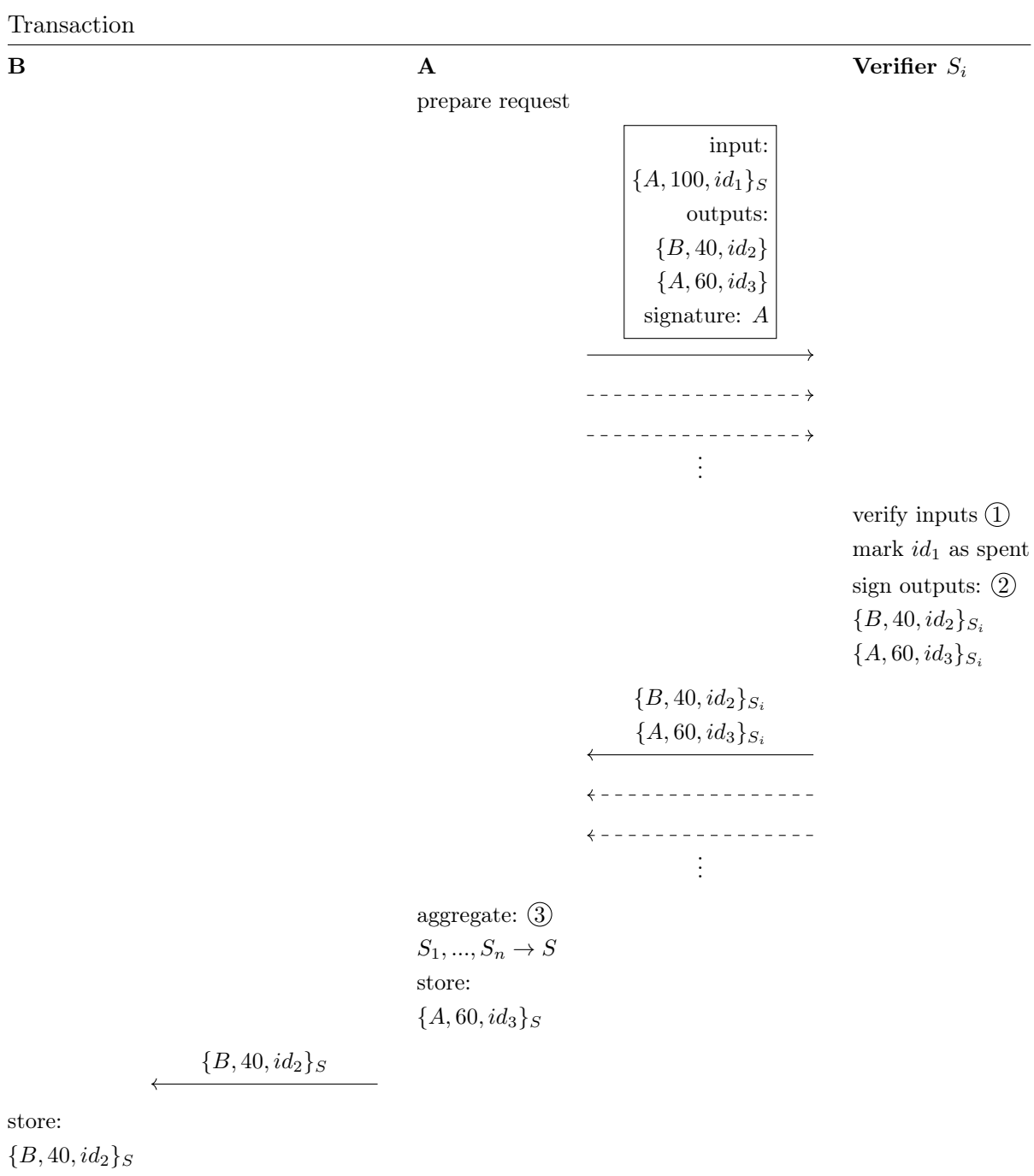


Figure 3.1: Transaction of 40 from A to B

Chapter 4

Implementation

The validator and client are built using Go. Go suits the system's purpose because the implementation uses a lot of coroutines that are natively supported within Go.

Having optimal signing and verification algorithms is crucial because cryptographic operations dominate the total processing time of the system.

The implementation contains three *cryptographic signature schemes*:

- Naive scheme;
- Merkle signatures scheme; and
- BLS threshold signature scheme.

One can switch between the three cryptographic schemes by setting an option when running the executable. A cryptographic scheme can only be chosen once; they are not interoperable and cannot be changed during runtime.

In the implementation, three essential cryptographic operations are used:

- Signing;
- Threshold signature aggregation $a : s_1, ..., s_k \to S$; and
- Threshold signature verification.

A verifier *signs* outputs ("signing", see ② in the figure 3.1). The client then *aggregates* multiple signed outputs from a quorum to a full signature ("threshold signature aggregation", see ③ in the figure 3.1). Before signing outputs, a verifier additionally *verifies* the client's inputs ("threshold signature verification", see ① in the figure 3.1). For each scheme, the implementation of every operation is described.

4.1 Naive Scheme

For signing and verification, the naive scheme uses EdDSA with Curve25519 (Ed25519). A Go library [5] provides bindings to ed25519-donna [6]. Ed25519-donna is written in C++ and provides a fast implementation of the Ed25519 public-key signature system [7]. Batch verification can be used for greater throughput.

Signing. The validator S_i signs the hash of the output.

Threshold signature aggregation. The threshold signature scheme is implemented in a naive way: aggregating multiple *validator signatures* to a validator quorum signature $a : s_1, ..., s_k \to S$ is performed by simple concatenation of all signatures $a(s_1, ..., s_k) = \langle s_1, ..., s_k \rangle$.

Threshold signature verification. When verifying a threshold signature $\langle s_1, ..., s_k \rangle$, the signatures $s_1, ..., s_k$ of a single output are verified using Ed25519 batch verification. There must be $> \frac{2}{3}n$ signatures signed by different validators.

The verification time increases linearly with the quorum size.

4.2 Merkle Scheme

In the Merkle signature scheme, Merkle signatures (see A.1) are used. The number of expensive EdDSA operations needed to sign an output is decreased.

A validator collects ("pools") a large number of outputs and combines them into a Merkle tree that then can be used to efficiently generated signatures for each of the outputs.

Signing. By *pooling* multiple signing requests, the validator collects a large number p of unsigned outputs. For each output o_j, a verifier i creates a Merkle signature $\{path_{o_j}, m_{s_i}\}$. For more information on Merkle signatures, see the appendix A.1. When a verifier signs p outputs, only one EdDSA signing operation is executed (compared to p operations with the naive scheme).

Threshold signature aggregation. The same applies as in the naive scheme: the aggregation $a : s_1, ..., s_k \to S$ is performed by concatenation $a(s_1, ..., s_k) = \langle s_1, ..., s_k \rangle$.

Threshold signature verification. As in the naive scheme, when verifying a threshold signature $\langle s_1, ..., s_k \rangle$, there must be $> \frac{2}{3}n$ signatures signed by different validators. We verify each Merkle signature $s_j = \{path_{o_j}, m_{s_i}\}$: we reconstruct the authentication path and verify the signature of the Merkle root. There are p signatures with the same Merkle root m_{s_i}; we can cache the verification result. The verifier only has to verify m_{s_i} for the first time the verifier has encountered the signature. For all $p-1$ subsequent encounters of m_{s_i}, the verifier can look up the cached result of the verification. This leads to a significant speedup because there is only one EdDSA verification needed for verification of p outputs.

4.2.1 Optimal Merkle Tree Size

The Merkle tree size has implications on the signing and verification time. If the Merkle tree is very large, the hashing time dominates the verification and signing process. However, if the Merkle tree size is small, the EdDSA cryptographic operations dominate.

Depending on the number of validators in the system, there exists a Merkle tree size that minimizes the work for a validator. Given the number of validators in the system, we can derive the optimal size of the Merkle tree. That is the size of the Merkle tree that minimizes the expected time a validator requires when verifying and signing a UTXO. We create cost variables $\mathbb{E}[C_{naive}]$ and $\mathbb{E}[C_{merkle}]$ that denote the expected cost (time) per UTXO for a single validator for the naive and Merkle scheme. The cost consists of verification cost and signing cost. Finally, we create a minimizer n^* that calculates the optimal number of leaves in a Merkle tree given the quorum size of the system.

Let q be the quorum size of the system, n the number of leaves in the Merkle tree, c_h, c_s, c_v the costs of hashing, EdDSA signing and verification. The expected cost for the naive scheme C_{naive} is

$$\mathbb{E}[C_{naive}] = \underbrace{q \cdot c_v}_{\text{Verification}} + \underbrace{c_s}_{\text{Signing}}$$

The expected cost of the Merkle scheme $\mathbb{E}[C_{merkle}]$ is

$$\mathbb{E}[C_{merkle}] = \underbrace{q \cdot \left(\log_2(n) \cdot c_h + \frac{c_v}{n}\right)}_{\text{Verification}} + \underbrace{\frac{2n \cdot c_h + c_s}{n}}_{\text{Signing}}$$

We measure the relative costs of the operations: $c_h = 1$, $c_s = 63$, $c_v = 107$. Given the quorum size q' calculating the optimal number of leaves in a Merkle tree can be done by minimizing $\mathbb{E}[C_{merkle} \mid c_h = 1, c_s = 63, c_v = 107]$ for n:

$$n^* = \arg\min_n \ \mathbb{E}\left[C_{merkle} \ \middle| \ c_h = 1, c_s = 63, c_v = 107, q = q'\right]$$

As an example: for a system with 10 validators, the quorum size is $q' = 7$. With this value, we get

$$n_{10}^* \approx 80.41$$

This means that for a system with 10 validators, either a Merkle tree with 64 or 128 leaves is optimal.

Benchmarking on the reference instance. The signing and verification times for the Merkle signature scheme have been measured on the AWS reference instance for different Merkle tree sizes. The results have been plotted in figure 4.1.

In figure 4.2 the expected cost $\mathbb{E}[C_{merkle}]$ has been plotted for a system with 10 validators using the measurements from figure 4.1. The global minimum seen in figure 4.2 matches the result derived by n_{10}^*.

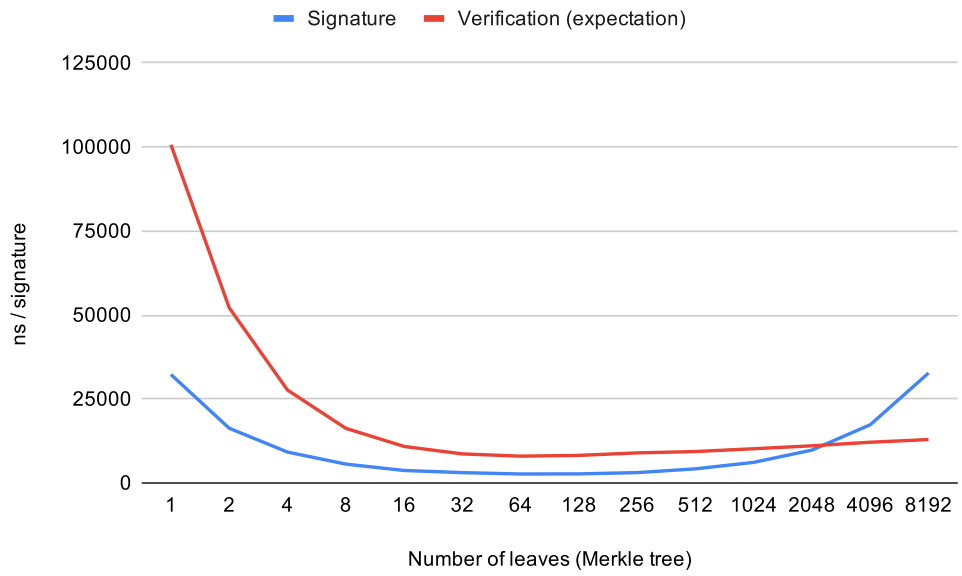


Figure 4.1: Mean signing and verification time given the number of leaves in the Merkle tree.

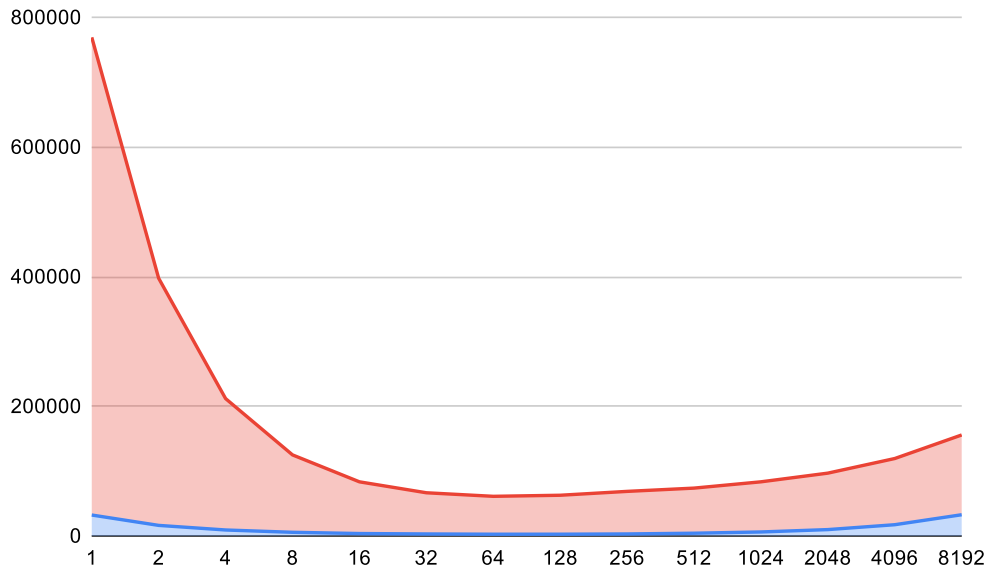


Figure 4.2: Objective function for a system where the number of validators is 10.

4.3 BLS Signature Scheme

The great advantage is that aggregated BLS threshold signatures have a constant verification time. `herumi/bls` [8] with Go-bindings [9] is used. This signature scheme is not interactive.

Each validator receives a BLS key share for signing. Each client receives a copy of the BLS master public key.

Signing. The verifier signs the hash of the output with its BLS key share.

Threshold signature aggregation. The aggregation $a : s_1, ..., s_k \rightarrow S$ is performed by native BLS signature aggregation $a(s_1, ..., s_k) = S$. Signature S is a valid signature when verifying using the master public key that is known by all clients. BLS signature can only be performed with $> \frac{2}{3}$ distinct signatures.

Threshold signature verification. The threshold signature S can be verified using the master public key.

Scheme	Operation	ns per signature
Naive	Signature	29,967
	Verification, single	100,663
	Verification, batch, 64 signatures	51,247
Merkle	Signature	2709
	Verification, no caching	106,771
	Verification, cached	6473
BLS	Signature (share)	640,205
	Verification of an aggregated signature	1,918,578

Table 4.1: Benchmarks for relevant cryptographic operations for each scheme. Fort the Merkle scheme, the number of leaves is 64.

4.4 Cryptographic Scheme Comparison

Only the cryptographic operations executed on verifiers (② and ③ in figure 3.1) are considered for performance evaluation. The bottleneck of the system are the validators: in the real world, it is assumed that the number of clients is vastly bigger than the number of validators. The benchmark of the cryptographic operations executed on a single core on the reference instance is denoted in table 4.1.

The verification time of the naive scheme is more than three times higher than its signing time. If multiple signatures are verified in one batch, EdDSA can take advantage of x86 SIMD instructions. This gives verification a speedup of up to ≈ 2.

Assuming that the Merkle tree in the Merkle scheme has 64 leaves, compared to the naive scheme, signing using the Merkle scheme is more than a magnitude faster. Thanks to caching, verification is up to 7 to 15 times faster; depending on whether batch verification is used in the naive scheme.

Signing and verification time using the BLS signature scheme are high. In the BLS scheme, clients create threshold signatures: clients aggregate q validator signatures into a single threshold signature, where q is the quorum size. The verification of threshold signatures is constant, making BLS efficient when the number of validators in the system is high. BLS is faster than the naive scheme if the quorum size is > 25 (the number of validators is > 37). BLS is faster than the Merkle scheme if the quorum size is > 317 (the number of validators is > 475). What's more, for a large quorum size, the naive and Merkle scheme produce prohibitively large signatures whereas BLS signatures have a constant size.

In conclusion: let V be the number validators. If $1 \leq V \leq 475$, the Merkle

scheme should be used. If $V > 475$, the BLS scheme should be used.

4.5 Storage

To prevent double-spending, each validator keeps track of which outputs have been spent. A UTXO o is marked as spent, if a verifier signs a new output o' where o has been used as an input.

The identifiers of spent outputs are stored in a hash map (see A.2). The hash map has to be thread-safe and efficient. Golang's built-in thread-safe hash map, sync/map, exhibits excessive lock usage and coroutine blocking, limiting the total throughput of the system. We created an optimized hash map that performs well: running on the AWS reference instance with a coroutine for each processor, Golang's implementation reaches $7.04 \cdot 10^5$ inserts per second. Our implementation is more than a magnitude faster: $3.30 \cdot 10^7$ inserts per second.

4.6 Sharding

To increase verifier performance, verifiers can are sharded. A *verifier shard* consists of multiple *verifier instances*. A client is assigned a verifier instance inside a shard that can be derived from its address: $addr \pmod k$ calculates the index of the assigned instance, where $addr$ is its address and k is the total number of instances in a verifier shard.

Chapter 5

Evaluation

5.1 Benchmarks

To approximate the real-world performance of the system, end to end benchmarks of the whole system are performed. The benchmark is executed with servers rented at AWS. This benchmark should give an intuition on how the system behaves on our reference instance.

We set the number of shards per validator to 1. More shards can easily be added. Each validator is spun up on an `m5ad.16xlarge` EC2 instance that has 64 vCPUs and 256 GiB of memory. As of August 2020, Amazon offers this instance for $4.00 per hour.

The benchmarks are run using the naive scheme. Because the bachelor's thesis time was limited, the Merkle and BLS signature scheme have not been benchmarked yet.

The throughput of the system is practically unbounded: validators can be scaled horizontally with sharding with an *ideal speedup* (linear speedup): doubling the number of instances doubles the system's throughput. For example, if we set the number of total validators in the system to 10 and have 5 shards per validator, the system reaches a throughput of 130,000 transactions per second using the naive scheme. For 50 shards, 1,300,000 transactions per second can be reached.

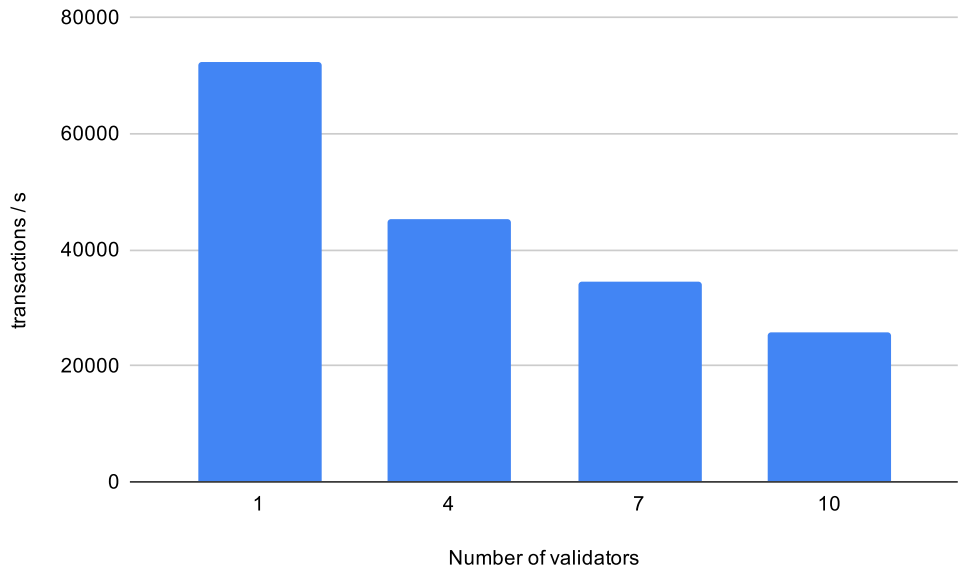


Figure 5.1: Benchmark of non-sharded naive scheme: transactions per second given the number of total validators in the system.

Chapter 6

Conclusion

In this thesis, we presented a protocol and a corresponding implementation of a consensus-free transaction system that has a fixed number of validators. The system is byzantine fault-tolerant: a client does not have to trust a single entity to ensure that the system is correct; up to a third of validators can behave maliciously without influencing the correctness of the system. This makes the system secure and stable. Thanks to numerous optimizations such as exploiting Golang's Goroutine parallelism, efficient cryptographic operations (EdDSA), high-level cryptographic optimizations (Merkle signatures) and BLS signatures, the system demonstrates a high throughput even when there are many validators in the system. All successfully executed transactions are guaranteed to be final; the transactions have instant confirmation time. The system can be perfectly scaled horizontally without losing efficiency.

These features make the protocol design a good basis for commercial applications such as large scale transaction systems.

6.1 Possible Applications

This system allows the implementation of a payment network where users can execute real-time low-latency retail payments. In this use case, each bank would be assigned a validator key and would run its validator. Then, customers could securely and efficiently transfer funds between each other, even if the customer are from another bank. The integrity of the payment system is ensured, even if up to one-third of the banks crash or act maliciously.

The system could also be used as a Real-Time Gross Settlement (RTGS) system between banks. RTGS systems are used to transfer funds between banks. Usually, RTGS systems are centralized and run by a single authority (usually the central bank) — using our system would allow banks to run a high-throughput RTGS without the need of a single authority while still maintaining the same level of integrity.

Benchmarks of Merkle and BLS signature schemes. Due to lack of time, only the naive scheme has been benchmarked. Given the benchmarks of the signing and verification operations (table 4.1), rough estimates of the throughput of Merkle and BLS schemes are possible:

Compared to the naive scheme, the Merkle scheme's signing operation was more than a magnitude faster. Depending on the quorum size, verification time was 7 to 15 times faster. Given that cryptographic operations dominate the execution time of validators, using the Merkle scheme could lead to a speedup of around a magnitude.

BLS signatures are fast when there are a large number of validators (see 4.4), because the verification time is constant.

6.2 Future Work

Benchmarking the Merkle and BLS scheme. Performing benchmarks of the Merkle and BLS scheme would further emphasize the advantages of Merkle and BLS threshold signatures.

Validator joins and leaves. The number of validators in the system is fixed. A good future optimization would be allowing us to add or remove validators. A new validator could join the validator set if all validators agree. However, this would lead to further complications, such as communicating to the client that the set of validators has changed.

UTXO store. In the implementation, the UTXO store is stored in memory. To ensure that validators do not lose information on spent UTXO and that the memory doesn't get full, the information would have to be swapped out to memory.

Bibliography

[1] S. Gupta, "A non-consensus based decentralized financial transaction processing model with support for efficient auditing," Master's thesis, Arizona State University, Jun. 2016.

[2] J. Sliwinksi and R. Wattenhofer, "ABC: Proof-of-stake without consensus," 9 2019.

[3] M. Baudet, G. Danezis, and A. Sonnino, "Fastpay: High-performance byzantine fault tolerant settlement," Mar. 2020.

[4] D. Malkhi and M. K. Reiter, "Byzantine quorum systems," in *Distributed Computing*, vol. 11, Oct. 1998, pp. 203–213.

[5] "ed25519 - optimized ed25519 for go," https://github.com/oasisprotocol/ed25519, 2020.

[6] D. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "ed25519 donna," https://github.com/floodyberry/ed25519-donna, 2020.

[7] D. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering 2*, 2012.

[8] S. Mitsunari, "BLS threshold signature," https://github.com/herumi/bls, 2020.

[9] S. Mitsunari, "BLS threshold signature for ETH with compiled static library," https://github.com/herumi/bls-eth-go-binary, 2020.

Appendix A

Concepts

This section will clarify the concepts of Merkle signatures and hash maps. Merkle signatures are used in section 4.2 to improve the efficiency of validators. A custom hash map is developed to improve the performance of the UTXO store, as described in section 4.5.

A.1 Merkle Signatures

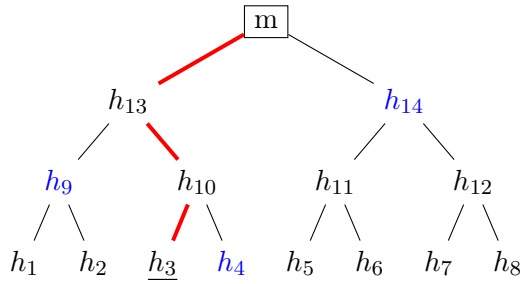
Merkle signatures are used to improve the performance of creating and verifying signatures.

Signing. A validator S_i collects n hashes to sign where $n = 2^k, k \in \mathbb{N}$. The hashes are combined into a Merkle tree. The validator S_i signs the Merkle root m using EdDSA, denoted as m_{S_i}. For each hash $h_i, i \in \{1, .., n\}$, the validator calculates the Merkle path and outputs $path_{h_i}$. $path_{h_i}$ consists of the hashes and side (left or right) of the nodes from h_i to the Merkle root (in blue). The resulting signature for h_i by S_i is the tuple $(path_{h_i}, m_{S_i})$. An example can be seen in figure A.1.

Signing n hashes using Merkle signatures is more efficient than signing n hashes separately: the cost of an EdDSA signing operations c_s is much higher than the cost of a hash operation c_h. The cost of signing n hashes with Merkle signatures is $2n \cdot c_h + c_v$ whereas the cost of signing n hashes separately (without Merkle signatures) is $n \cdot c_v$.

Verifying. First, the Merkle root m' is reconstructed from the path $path_{h_i}$. If $path_{h_i}$ is a valid path, m' matches m. Finally, m_{S_i} is verified using EdDSA.

Verifying n hashes using Merkle signatures is also more efficient than verifying n hashes separately because the cost of an EdDSA verification operation is high, even higher than an EdDSA signing operation (and thus also higher than the cost of a hashing operation). The Merkle root can be reconstructed with a cost



$$path_{h_3} = \{h_4, \text{right}\}, \{h_9, \text{left}\}, \{h_{14}, \text{right}\}$$
$$sig = (path_{h_3}, m_{S_i})$$

Figure A.1: Merkle tree and Merkle signature for hash h_3.

of $\log_2(n) \cdot c_h$ where c_h is the cost of hashing. The Merkle root m only has to be verified for the first encountered Merkle signature, after that, the result of the verification can be cached, making this signature perform very well.

A.2 Hash Maps

Hash maps are used to efficiently calculate set membership in the context of spent UTXO identifiers. Elements $e \in E$ (in our case id_i) are hashed into the hash range $H = \{0, ..., l\}$ and used as an index to an array of linked lists. A linked list at index $i \in \{0, ..., l\}$ contains all items e where $\text{hash}(e) = i$. The linked list is then traversed.

Hashing and indexing are implemented as a lock-free operation; traversing and modifying a linked list are protected by a mutex. If l is large enough, the probability of hash collisions is small, minimizing lock contention and the length of the linked list. Therefore l is initialized to be as large as the memory size permits.

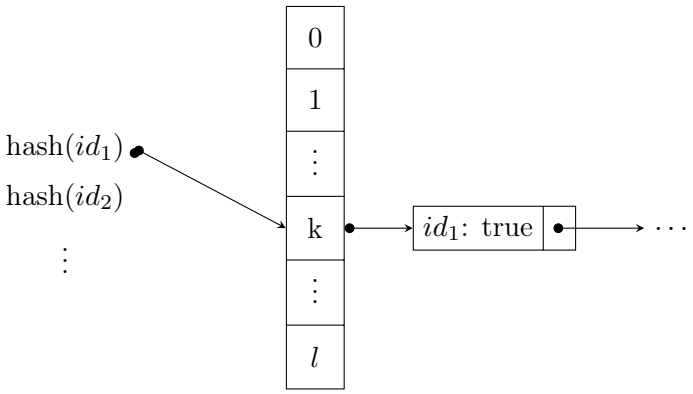


Figure A.2: A hash map, where id_1 indexes into $\text{hash}(id_1) = k$