Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Asynchronous Consensus-Free Transaction Systems

Semester Thesis

Shoma Mori

shmori@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

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

December 19, 2019

Acknowledgements

I thank Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Abstract

The abstract should be short, stating what you did and what the most important result is. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Contents

Chapter 1

Introduction

In the existing blockchain systems, it is assumed that the consensus problems should be solved to achieve correctness. For instance, Bitcoin uses proof-of-work to get consensus among agents in distributed systems.

However, proof-of-work algorithm takes much cost and therefore prevents scalability. The transaction systems can get popularity if they get scalability improvement.

We introduce Asynchronous Consensus-Free Transaction Systems (ACFTS) aiming to get scalability by removing consensus protocol. ACFTS verifies transactions using agents call servers, which dedicate to manage transactions and keep correctness in the system. The servers put their signatures to prove the correctness of transactions. Importantly, the servers work asynchronously meaning each server does not have to communicate with other servers. In other words, ACFTS does not need consensus.

We implemented ACFTS as a payment system and evaluated the throughput. From the results, we found that the part of the verification of signatures becomes a bottleneck.

Chapter 2

Asynchronous Consensus-Free Transaction Systems

2.1 Model

ACFTS consists of two kinds of agents called servers and clients. The clients can send cryptocurrency to the other clients or themselves. In the system, a transaction represents a transfer of cryptocurrency between clients. Each client holds private and public key pairs and the public keys are also called addresses, which are used to show owners of transactions. The servers verify transactions when they receive them from clients and record the history of valid transactions in their local storage.

Every client can send messages to all servers and all other clients. In the same way, every server can send messages to all clients. However, the servers do not send messages to the other servers. Messages are delivered asynchronously, that is, messages reach to receivers eventually, however, there is no guarantee that the messages arrive within finite time.

2.2 Protocol

2.2.1 Transaciton

A transaction represents a transfer of cryptocurrency from one client to other clients or itself. Each transaction consists of one or more inputs and one or more outputs.

An input includes one or more outputs that will be spent. The outputs which can be used as elements of inputs is called UTXO (Unspent Transaction Output). Also, the input has a signature that is generated by a private key corresponding to the public key (i.e. address) of the UTXOs. In other words, each UTXO can be spent by only the client who knows the private key which is linked to the

public key. If a client has a private key corresponding to an address of a UTXO, we say that the client has ownership of the UTXO.

An output includes an address of its owner, amount of cryptocurrency, signatures from servers, and a hash of outputs of the previous transaction. In other words, transactions form a directed acyclic graph (DAG). The sum of the amounts of outputs must be the same as one of the amounts of inputs.

In the case where a transaction has more than one output, the outputs have "siblings." Each output is assigned an index in the siblings. Normally, outputs can be identified with an address and the previous hash. However, even if one transaction has multiple outputs that belong to the same addresses and have the same previous transaction, they are identified by the indexes.

Genesis

All outputs are created from any inputs, but the only genesis is different. The genesis is an initial output and all outputs refer to the genesis as an ancestor. The genesis is created by the system, its address represents the first owner and the amount equals to the sum of the cryptocurrency.

2.2.2 Server

A server is a validator who has the role of verifying transactions from clients. Every server records all transactions they verified in their memories. When a server receives a transaction, firstly, the server checks whether the outputs of the received transaction have been used in the past. If even one of them has been used, the transaction is regarded to be invalid and the server sends an error to the client. If it is valid, nextly, the server verifies a client's signature to confirm the ownership. Then, the server verifies signatures of servers which are also contained in the inputs. We assume that each server knows the public keys of all the other servers. Importantly, the number of valid signatures must be more than two-thirds of all servers (the details will be described in) to use the UTXOs. A UTXO that has signatures from more than two-thirds of all servers is called a valid UTXO. Finally, the server checks if the sum of the amounts of the outputs is the same as one of the amounts of the inputs. When the server completes the verification process without any errors, it approves the transaction by making an own signature from the hash of the outputs using a private key of the server and attaches it to the response to the client. The number of signatures of one server for each transaction is only one, which means the signature is created from the entire outputs, not from each output. This reduces the number of necessary signatures and saves the cost as a result. Finally, the server adds the outputs into their storage and updates the status of the inputs to record that they cannot be used anymore.

2.2.3 Client

Clients can create new transactions. They send requests for getting signatures from servers to make the transaction valid. The client manages not only their own outputs but also their sibling outputs because servers make a signature from the entire outputs in each transaction. Therefore, the client needs to send the UTXOs with the siblings in order to make it possible for servers to verify them. The client attaches a signature to claim the ownership of the UTXOs when sending the request.

Consider a transfer of cryptocurrency from client c_1 to client c_2. First, c_1 sends a request for the transaction to all servers and waits for the responses. When c_1 gets signatures from more than two-thirds of all servers, c_1 can "spend" the transaction. In order to show the use of the UTXOs and make it possible for c_2 to use the new outputs, c_1 sends c_2 the outputs with signatures of servers. c_2 can confirm the transaction by verifying the signatures.

2.2.4 The flow of a payment

The following is a flow of creating a transaction that represents a transfer of the cryptocurrency from client c_1 to c_2.

1. c_1 finds valid UTXOs which c_1 owns in their local storage.
2. c_1 creates a request for a transaction whose output address represents c_2 using the UTXOs including a c_1's signature.
3. c_1 sends the request to all servers and waits for the responses.
4. When servers receive the request, they verify the transaction.
5. If the transaction has no errors, servers create their signatures and send them back as responses.
6. When c_1 gets signatures from more than two-thirds of all servers, c_1 sends the output of the new transaction to c_2.
7. When c_2 receives the output, c_2 verifies the signatures of servers.
8. If c_2 confirms that the number of valid signatures is more than two-thirds of the number of all servers, the transaction is regarded to be approved.

2.2.5 Double-spending

In general transaction systems, using the same outputs more than twice, namely, double-spending, is one of the critical problems. In our system, it is impossible

to make transactions that use the same UTXO as the inputs. We call those transactions conflicting transactions.

Consider a situation where c_1 tries to make two conflicting transactions which are payments to c_2 and c_3. To make a transaction valid, c_1 has to get signatures from more than two-thirds of all servers. However, if a server receives two conflicting transactions, it creates a signature for only one transaction which comes first. In other words, it is impossible for the clientto get signatures of both transactions from the same server. Furthermore, the client is incapable of sending each transaction to more than two-thirds of all servers without overlapping. In short, the two conflicting transactions are never be approved at the same time. In this case, only one or neither transaction is approved.

Chapter 3

Implementation

In this chapter, we describe the system in terms of implementation.

3.1 Structure

Servers and clients can communicate through the HTTP protocol. We adopted JSON as the format of messages. Servers keep waiting for HTTP requests from clients. Clients send the servers requests when they want to get signatures of new transactions.

Clients also can send messages to the other clients through the HTTP protocol to notice new UTXOs that are owned by them. Therefore, clients also keep waiting for HTTP requests as well as servers.

Both servers and clients have their database to record transaction outputs. The genesis is initially recorded in the client who has it and servers approve the transaction whose input is the genesis without conditions.

3.2 Documentations

3.2.1 Signature

ACFTS uses public-key cryptography to show ownership of outputs and prove that transactions are approved by servers. The implementation adopts the Elliptic Curve Digital Signature Algorithm (ECDSA) for key pairs of servers and clients and the verification processes.

A client creates a hash of UTXOs with SHA256 and signs it with a private key that is generated with ECDSA when creating a new transaction. A server verifies the signature with the public key of the client and signs the UTXOs with a private key which is also generated with ECDSA by the server. The receiver of the transaction can verify the signatures of servers with the public keys of the servers.

UTXO hash

A UTXO has an address, a hash of outputs of the previous transaction and an index in siblings. We show that UTXOs can be identified with these keys.

Initially, the genesis is the only transaction and there are no identical transactions. Hash values become the same deterministically if the inputs are identical. Without collisions of the hash function, if the input is different, the output becomes different. Therefore, if the contents of the previous transactions are different, two transactions are distinguished. Every output becomes different because even if addresses and the hash of the previous transactions are the same, they have different indexes. In short, every transaction has a different set of an address, a hash of outputs of the previous transaction and an index.

However, if the hash values conflict, two outputs can be the same although it is usually not the problem probabilistically.

3.2.2 Change output

When a client tries to create a request for a transaction, the client collects UTXOs from their database until the sum of the amounts becomes larger than or equal to the amount the client wants to send. If the sum exceeds the necessary amount, the clients make *a change output* to make both ends meet.

3.2.3 Cluster

In the implementation, multiple client addresses can be managed in one database. We call this set of addresses cluster. When creating transactions in one cluster, it is not necessary to send UTXOs because they can refer to through the shared memory. In some sense, a cluster is a wallet and transactions in one cluster represent sorting out UTXOs. When creating transactions among different clusters, it is required to send the UTXOs to the related cluster.

Clients can use different addresses depending on transactions for protecting privacy.

In the initialization process, each cluster exchanges their addresses and therefore they can decide which cluster they should send UTXOs when creating new transactions.

Chapter 4

Experiment

We evaluated the throughputs of our systems by experiments using some scenarios of transactions. The system is implemented in Go and was benchmarked in a local environment. The benchmarking has been performed on a laptop with Intel Core i5 3.1GHz CPU and 16GB RAM. We do not assume network delay.

We also profiled the system to find bottlenecks aiming for improving the throughputs.

4.1 Environments

- The number of servers: 4
- The number of clusters: 2 ($cs0$ and $cs1$)
- The number of clients in each cluseter: 4
  ($\{ct0, ct1, ct2, ct3\} \in cs0$ and $\{ct4, ct5, ct6, ct7\} \in cs1$)
- The genesis: $amount = 1000000$, $owner = cs0$

We executed the following five diffrent scenarios. Every arrow indicates transfer of one amount of cryptocurrency.

- Scenario1: $ct0 \to ct1$
- Scenario2: $ct0 \to ct1$, $ct1 \to ct0$
- Scenario3: $ct0 \to ct1$, $ct1 \to ct0$, $ct2 \to ct3$, $ct3 \to ct2$
- Scenario4: $random \to random$ ($rondom \in cs0$)
- Scenario5: $ct0 \to ct4$

Note that sender and receiver are chosen from $cs0$ with equal probability in scenario4.

4.2 Results

4.2.1 Processing speed

Trials in tables means that execution of a set of transctions in one scenario counts one trial. For example, in scenario1, when trials is 10, transfer from $ct0$ to $ct1$ is executed 10 times. On the other hand, in scenario2, when when trials is 10, transfers from $ct0$ to $ct1$ and $ct1$ to $ct0$ are executed 10 times respectively. Speed is the number of approved transactions per second.

Table 4.1: Scenario1

trials [tx]	speed [tx/s]
10	4.30
100	5.25
1000	4.37
10000	3.10

Table 4.2: Scenario2

trials [tx]	speed [tx/s]
10	5.11
100	5.56
1000	4.82
10000	1.52

Table 4.3: Scenario3

trials [tx]	speed [tx/s]
10	5.35
100	5.51
1000	5.06
10000	1.98

Table 4.4: Scenario4

trials [tx]	speed [tx/s]
10	5.13
100	4.40
1000	5.13
10000	1.97

Table 4.5: Scenario5

trials [tx]	speed [tx/s]
10	3.10
100	3.13
1000	4.54
10000	2.52

There are some points that can observe from each table. Although there is an exemption, in the most scenarios, the throughput improves when trials goes from 10tx to 100tx. After that, although there is an exemption too, the throughputs go down as incresing the number of transacitons. Especially, the throughputs become less than half when trials go from 1000tx to 10000tx in some scenarios.

Next, we can get some tendencies when comparing diffrent scenarios. From scenario1 to 4 are sets of transactions in one cluster, but scenario5 is a set of

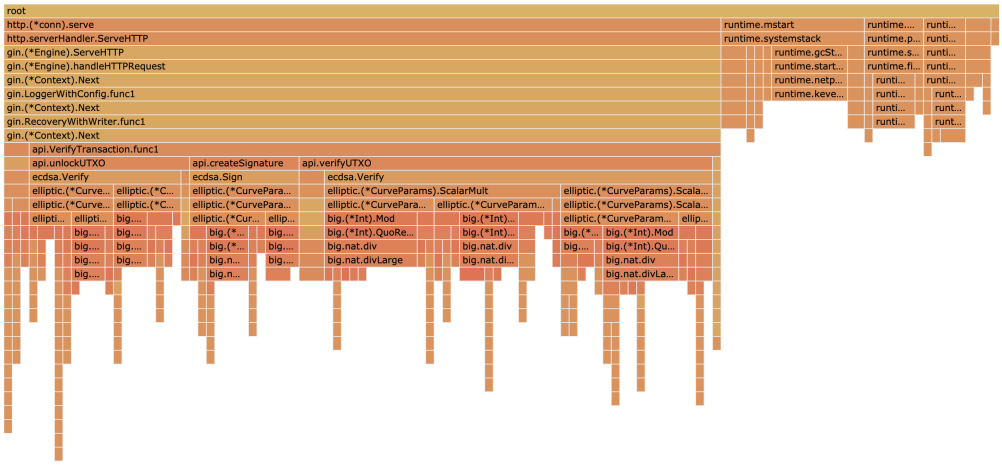

Figure 4.1: Flame graph in a server

ones between two clusters. The overall throughput of scenario5 is less than the others.

4.2.2 Bottlenecks

We employed flame graphs to find bottlenecks of the system.

Flame graph

Flame graph is a visualization tool which allows identify the most frequent code-paths. The y-axis shows the stack depth, ordered from root at the top to leaf at the bottom. The x-axis spans the stack trace collection. The width of each function box shows the frequency at which that function was present in the stack traces, or part of a stack trace ancestry. Bottleneck functions have wide width boxes in the flame graphs.

As we can see in the flame graph of a server-side, it is found that api.verifyUTXO, api.unlockUTXO and api.createSignature cost much. If you trace the stacks, ecdsa.Verify and ecdsa.Sign are the causes of the bottlenecks. ecdsa is a cryptographic library of the Elliptic Curve Digital Signature Algorithm in Go. In api.verifyUTXO, a server verifies signatures of the receiving UTXO used in the inputs. In api.unlockUTXO, a server verifies a signature of the client who sent the request. In api.createSignature, a server issue a signature to prove the correctness of the transaction.

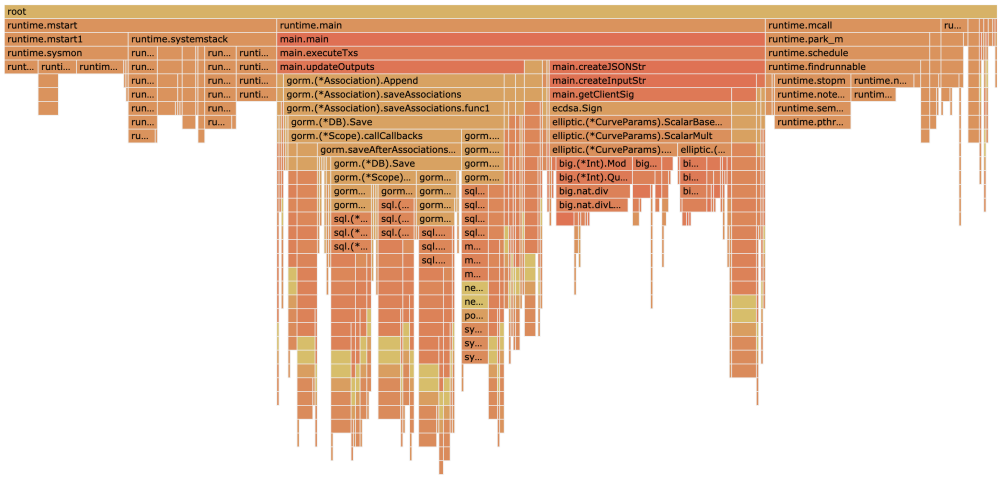

Figure 4.2: Flame graph in a client

4.3 Discussion

4.3.1 Transaction between different clusters

Throughput of scenario5 is smaller than the other scenarios. Clients have to send UTXOs to different clusters when they make outputs that include ones belonging to different clusters. Now the sender waits for a response from the receiver, which can cost time.

One of the ways to improve throughputs of transactions among clusters is that not to wait for the responses. Actually, in the real use of this system such as payment in cafe, a payer sends signatures of servers in some way or just show something (e.g. QR code) which has the information. Then, it is not necessary for a payee to respond in some digital way.

4.3.2 Waiting for signatures from servers

Currently, clients wait for responses from all servers after they sent requests of new transactions. However, it is enought if they have only two-thirds of all signatures to show the validity of the new transactions. Suppose that a client sends valid transaction and all servers will send signatures back, if the client do not wait after getting more than two-thirds of all signatures, the speed will 1.5 times faster.

4.3.3 Algorithm for cryptographic processes

The most of bottlenecks is attributed to crytographic processes such as make signatures and verification of the signatures. There are no specific reasons why we must adopt ECDSA as the algorithm for the cryptographic processes. We can take into account other algorithms for our system.

4.3.4 Changes of servers

Now the number of servers and their addresses are fixed. However, if the servers are fixed permanetly, they can be a vulnerability when some of them stop or break. To avoid that situation, the system should adapt to changes of servers.

4.3.5 Run in a real environment

Chapter 5
Conclusion

foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo

Todo: This is a TODO annotation.

Theorem 5.1 (First Theorem)**.** *This is our first theorem.*

Proof. And this is the proof of the first theorem with a complicated formula and a reference to Theorem 5.1. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

$$\frac{\mathrm{d}}{\mathrm{d}x}\arctan(\sin(x^2)) = -2 \cdot \frac{\cos(x^2)x}{-2 + (\cos(x^2))^2} \tag{5.1}$$

□

And here we cite some external documents [1, 2]. An example of an included graphic can be found in Figure 5.1. Note that in LaTeX, “quotes” do not use the usual double quote characters.

Figure 5.1: This is an example graphic.

Bibliography

[1] A. One and A. Two, "A theoretical work on computer science," in *30th Symposium on Comparative Irrelevance, Somewhere, Some Country*, Jun. 1999.

[2] A. One and A. Two, "A theoretical work on computer science," in *30th Symposium on Comparative Irrelevance, Somewhere, Some Country*, Jun. 1999.

Appendix A

First Appendix Chapter Title