TODO TITLE

TODO AUTHOR

TODO TITLE

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

TODO AUTHOR

2nd June 2017

Author
TODO AUTHOR

Title
TODO TITLE

MSc presentation
TODO GRADUATION DATE

Graduation Committee
TODO GRADUATION COMMITTEE Delft University of Technology

Abstract

TODO ABSTRACT

Preface

TODO MOTIVATION FOR RESEARCH TOPIC

TODO ACKNOWLEDGEMENTS

TODO AUTHOR

Delft, The Netherlands
2nd June 2017

Contents

Chapter 1

Introduction

TODO some positive intro about blockchain

One of the key issues in many blockchain systems today is that they are not scalable. Bitcoin [**?**], the largest permissionless[1] blockchain system in terms of market capitalisation [**?**] has a maximum transaction rate of merely 7 transaction per second (TX/s). This is due to the consensus mechanism in Bitcoin, namely proof-of-work (PoW), miners can only create new blocks every 10 minutes and every block cannot be larger than 1 megabyte. Payment processors in use today such as Visa can handle transaction rates in the order of thousands [1]. While Bitcoin may be a revolutionary phenomenon, it clearly cannot be ubiquitous in its current state.

An different approach is to not reach global consensus at all. For instance in TrustChain [**?**] and Tangle [**?**], nodes in the network only store their personal ledger. Since consensus is left out, nodes can perform transactions as fast as their machine and network allows. The downside of this approach is that it cannot prevent fraud (it is possible to detect fraud). To examplify, a malicious node Mallory may claim she has 3 units of currency to Alice, but in reality Mallory already spent all of it on Bob. If there is no global consensus and Bob and Alice never communicate, then the 3 units that Alice is about to receive is nonexistent.

The scalability property of TrustChain and Tangle are exceptionally desirable. The global consensus mechanism of Bitcoin and many other blockchain systems are also worthwhile for detecting or preventing fraud. These two properties may seem mutually exclusive, but in this work, we demonstrate the opposite. Specifically, we answer the following research question in the affirmative. *Is it possible to design a blockchain fabric that can reach global consensus on the state of the system and also scalable?* We define scalability as a property where if more nodes join the system, then the transaction rate should increases.

Our primary insight came from observing the differences between how

[1] Explain permissionless

transactional systems work in the real world and how they work in blockchain systems like Bitcoin. Take a restaurant owner for example, most of the time the customer is honest and pays the bill. There is no need for the customer or the restaurant owner to report the transaction to any central authority because both parties are happy with the transaction. On the other hand, if the customer leaves without paying the bill, then the restaurant owner would report the incident to some central authority, e.g. the police. On the contrary, in blockchain systems, every transaction is effectively sent to the miners, which can be seen as a collective authority. This consequently lead to limited scalability because every transaction must be validated by the authority even when most of the transaction are legitimate.

Using the aforementioned insight, we explore an alternative consensus model for blockchain systems where transactions themself do not reach consensus, but nevertheless verifiable at a later stage by any node in the network. Informally, out model works as follows. Every node stores its own blockchain and every block is one transaction, same as the TrustChain construction. We randomly selected nodes in every round, the selected ones are called facilitators. They reach consensus not on the individual transactions, but on the state of every chain represented by a single digest, we call this state the checkpoint. If a checkpoint of some node is in consensus, then that node can proof to any other node that it holds a set of transactions that computes (form a chain) to the checkpoint. This immediately show that those transactions are tamper-proof.

We begin the detailed discussion by formulating the model **??**. Next, we analyse the correctness, security and performance of our design in **??**, this is where we present our theorems. Implementation and experimental results are discussed in **??**. Finally, we compare our system with other state-of-the-art blockchain systems and conclude in Sections **??** and **??** respectively.

Chapter 2

System Architecture

The primary goal guiding our design is scalability. As mentioned in the Introduction, having a scalable blockchain system while still keeping global consensus allows the system to be ubiquitous and realise the full potential of blockchain.

The secondary goal is to design an application neutral system. In particular, it should act as a framework that provides the building blocks of blockchain based applications. Application developers using the framework should be able to create any application they wish. Further, we do not impose on a consensus algorithm, the application developer is free to choose between proof-of-work, proof-of-stake, Byzantine consensus as long as it satisfies the properties in **??**TODO.

Due to the nature of our system, we do not explicitly address the Sybil attack [5]. Sybil defence mechanism always require some form of reputation score from the application. For example, social network based Sybil defence mechanims use graph structure of real-world relationships [8]. Online marketplaces such as Amazon use the rating of buyer and sellers. Thus it is not possible to design a Sybil defence mechanism with a a application neutral framework. On the other hand, our system also has no restrictions on the Sybil defence technique and application designers can pick the best mechanism for their application.

The third and final goal is security. Our system should be unaffected in the presence of powerful adversaries. Security is often difficult to verify, especially when it is not formalised, therefore we require our design to be provably secure. To summarise, our system design is designed with the following goals in mind.

- Application neutrality,
- scalability and
- security.

We begin the chapter with an informal overview of our design in Section 2.1. Next, we introduce the Universally Composable (UC) Framework and its explain its importance in Section 2.2. The UC framework is used in Section 2.3 to formally model our design. Finally, we give some possible extensions of our design in Section 2.4.

2.1 Protocol Overview

The protocol can be described in four parts, the extended TrustChain datastructure, the consensus protocol, the transaction protocol and finally the validation protocol. We first describe how the protocols work individually and then explain how they fit together.

2.1.1 Extended TrustChain

Extended TrustChain naturally builds on top of TrustChain, thus we first describe the standard TrustChain. Our description has minor differences compared to the description in [**?**]. This is to help with the description of the extended TrustChain. However, the two descriptions are functionally the same.

Standard TrustChain

In TrustChain, every node has their "personal" chain. Initially, the chain only contains a genesis block. When a node wishes to add a new transaction (TX), a new TX block is generated and is appended to the chain. A TX block must have a valid hash pointer pointing to the previous block and a reference[1] to its *pair*. Suppose Alice made a transaction with Bob, then both parties must create a TX block to acknowledge that the transaction took place. The pair of Alice's TX block is the corresponding TX on Bob's chain and vice versa. An example of 3 nodes is shown in Figure 2.1.

If every node follows the rules of TrustChain and we only consider hash pointers, then the chain effectively forms a singly linked list. However, if a node violates the rules, then a *fork* may happen. That is, there may be more than one TX block with a hash pointer pointing back to the same block. In Figure 2.1, node b (in the middle chain) created two TX blocks that both point to $t_{b,5}$. If this is a ledger system it can be seen as a double spend, where the currency accumulated up until $t_{b,5}$ are spent twice.

[1] This is different from the original TrustChain definition found in [**?**]. In there, a TX block has two outgoing edges which are hash pointers to the two parties involved in the transaction. This work uses one outgoing edge and a reference.

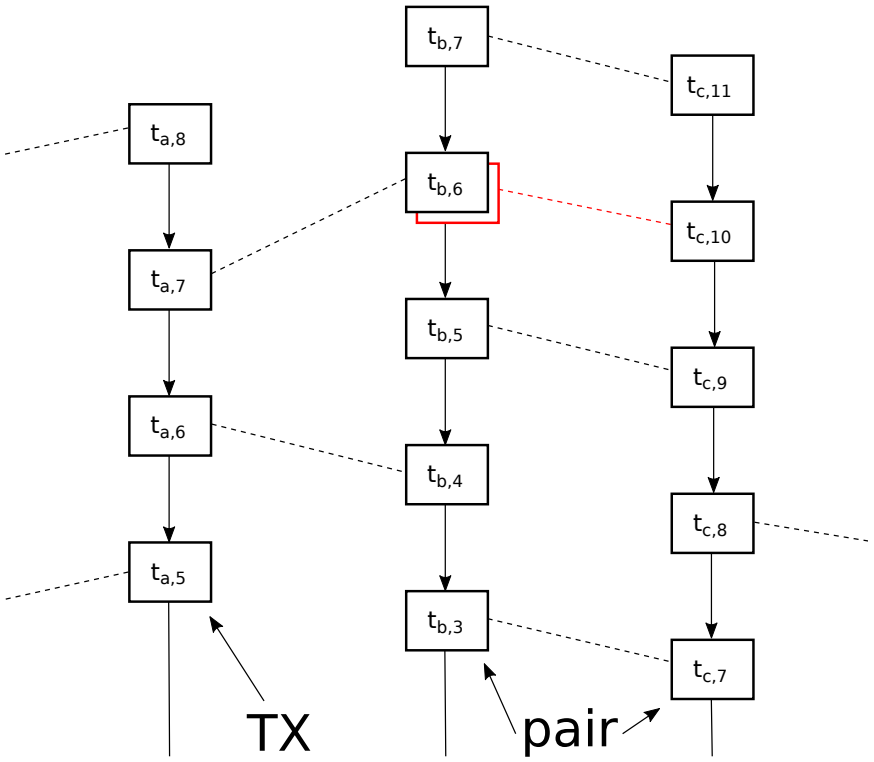

Figure 2.1: Every block is denoted by $t_{i,j}$, where i is the node ID and j is the sequence number of the block. Thus we have three nodes and three corresponding chains in this example. The arrows represent hash pointers and the dotted lines represent references. The blocks at the ends of one dotted line are pairs of each other. The red block after $t_{b,5}$ indicate a fork.

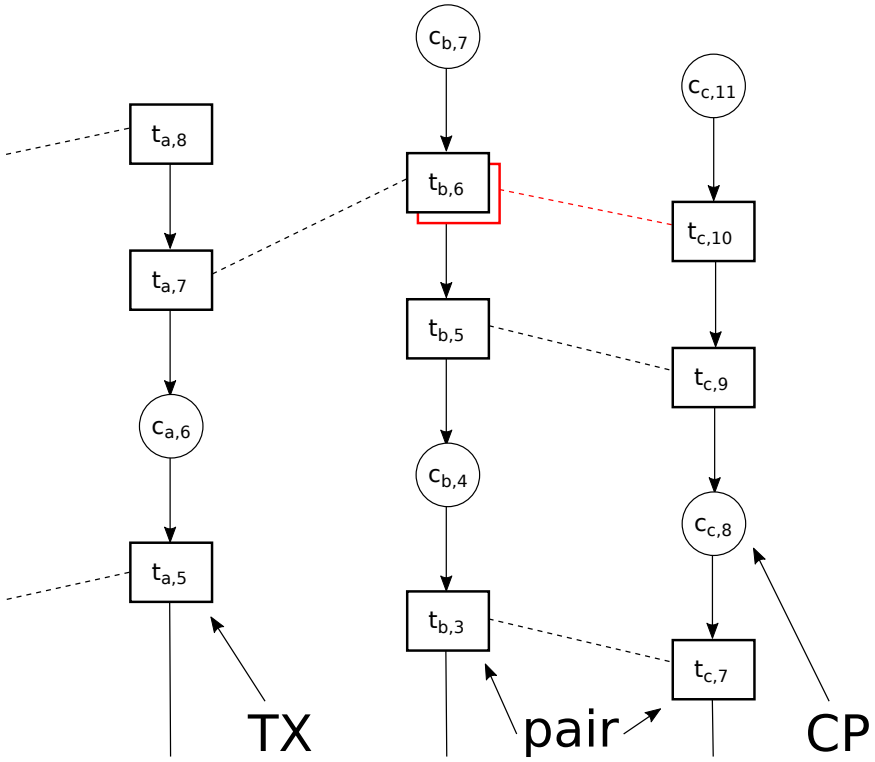

Figure 2.2: The circles represent CP blocks, they also have hash pointers (arrow) but do not have references (dotted line). Note that the sequence number counter do not change, it is shared with TX blocks.

Extended TrustChain

We are now ready to explain the extended TrustChain, which we abbreviate to ETC. In ETC, we introduce a new type of block, namely checkpoint (CP) block. In contract to TX blocks, CP blocks do not store transactions or contain references. They capture the state of the chain and the state of the whole system. In particular, the state of the chain is captured with a hash pointer. The state of the whole system is captured in the content of the CP block, namely as a digest of the latest *consensus result* which we explain in Section 2.1.2. A visual representation is shown in Figure 2.2.

2.1.2 Consensus Protocol

Before describing our consensus protocol, we take a brief detour to explain Byzantine consensus, which is a fundamental building block of our consensus protocol.

Byzantine consensus

Byzantine consensus is also known as *atomic broadcast*. Roughly speaking, atomic broadcast need to satisfy the following properties.

1. TODO

TODO downsides (can't run with too many nodes) (high message complexity) We stress that Byzantine consensus is not the same as Byzantine agreement or the Byzantine general's problem. Byzantine agreement is TODO

The literature on Byzantine consensus and atomic broadcast is rich, some noteable ones include TODO. Thus in the ETC consensus protocol, we assume there exist an "off-the-shelf" Byzantine consensus algorithm which we can use (in our implementation we use HoneyBadgerBFT and motivate our choice in TODO).

In our case, every node do not propose some arbitrary data, but a set of CP blocks. Thus the result of the consensus is the set union of all the legitimate proposals.

ETC consensus protocol

The consensus protocol runs continuously in rounds. That is because a blockchain systems always need to reach consensus on new values, or CP blocks in our case. This can be seen as running infinitly many rounds of some Byzantine consensus algorithm, starting a new execution immediately after the previous one is completed.

As we mentioned earlier, the high message complexity prohibits us from running a Byzantine consensus algorithm on a large network. Thus, for every round, we randomly select some node—called facilitators—to collect CP blocks, and then use them as the proposal of the Byzantine consensus algorithm. The facilitators are elected using a *luck value*, which is computed using $H(\mathcal{C}_r||pk_i)$, where $\mathcal{C}_r$ is the consensus result in round r and pk_i is the public key of i. Intuitively, the election is guaranteed to be random because the output of a cryptographically secure hash function is unpredictable and $\mathcal{C}_r$ cannot be determined in advance.

A visual explaination can be found in Appendix A, it walks through the steps needed for a node to be selected as a facilitator.

2.1.3 Transaction Protocol

The TX protocol [2] consist of two messages, `TX_REQ` and `TX_RESP`. When a node wishes to create a transaction, it generates a TX block with the corresponding hash pointer, sequence number, counterparty and the transaction

[2] It is also known as True Halves, devised and implemented by Ewout Bongers. See `https://github.com/Tribler/tribler/pull/2135` for more information.

message itself. It then sends a `TX_REQ` message containing the TX block to the counterparty. Upon receiving a TX block at the counterparty, it creates its own version of the TX block (the pair of the sender's TX block), and sends it back to the sender in a `TX_RESP` message. At the end of the protocol, both parties should have both versions of the TX block.

2.1.4 Validation Protocol

The consensus and transaction protocol by themselves do not provide a mechanism to detect forks or other forms of tamperaing. This is the goal of the validation protocol.

It also has two message—`VD_REQ` and `VD_RESP`. When a node wishes to validate a transaction, it sends a `VD_REQ` with the sequence number s to the party that created the TX block of the transaction. Upon receiving a VD_REQ, the node finds two *agreed* CP blocks that surrounds the TX block and computes the *fragment*. An agreed CP block means that it has reached consensus. A fragment of two CP blocks is a section of the chain where the beginning and the end are defined by those two CP blocks. With the fragment, the node replies with a `VD_RESP` message. Upon receiving a `VD_RESP`, the node checks that the fragment is created correctly. Loosely, that means the agreed CP blocks have actually reached consensus, the fragment has the correct hash pointers, and TX block containing s exists and is valid.

2.1.5 Putting The Protocols Together

The final protocol (Π_{real}) is essentially the concurrent composition of the three aforementioned protocols. Our subprotocol design gives us the highly desireable non-blocking property. In particular, we do not need to "freeze" the state of the chain for some communication to complete in order to create a block. For instance, a node may start the consensus protocol, and while it is running, the node may still perform transactions. By the time the consensus protocol is done, the new CP block is added to whatever the state that the chain is in. It is not necessary to keep the chain immutable while the consensus protocol is running.

2.2 Universally Composable Framework

In order to analyse the security of a system, a formal notion of security is required. For instance, what does it mean if an encryption algorithm is secure? One may say it is secure if the adversay cannot learn any information about the plaintext from the ciphertext. But what if the adversay has some background information, for instance she may know it is English. Do we then still say the encryption algorithm is secure? Goldwasser and Micali introduced the notion of semantic security [6]. That is, imagine two worlds,

a real world and an ideal world. In the real world, the adversay is given the ciphertext, and in the ideal world the adversay has nothing. Then the encryption algorithm is semantically secure if the amount of information that the adversay can learn in the real world is just as much as the ideal world. While our description is informal, the notion of security can be formally captured in this way.

Security sensitive distributed systems such as secure multi-party computation and blockchain systems also require a formal notion of security so that they can be analysed. Fortunately, the idea from semantic security can also be applied in a distributed setting. In an ideal world, we create an ideal functionality $\mathcal{F}_{\text{ideal}}$ that performs all the computation on behalf of every node in the network. The nodes act as dummies and only relay messages between $\mathcal{F}_{\text{ideal}}$ and the environment. Thus $\mathcal{F}_{\text{ideal}}$ essentially becomes the specification of the distributed protocol. If the execution of the real world protocol is indistinguishable from the ideal emulation in the presence of some adversay, then the real world protocol is secure as per the ideal specification. This is in essence the Universally Composable (UC) framework.

We use the UC framework not only because it suits our needs, it is also the only framework used in modelling blockchain systems to the best of our knowledge [**?**]TODO. In the remainder of this section we give an overview of the UC framework. A detailed treatment can be found in [3].

2.2.1 Model of Computation

The model of computation considered in the UC framework is the interactive Turing machine (ITM). Specifically, an ITM is an extension of the Turing machine with externally writable tapes which are the following.

1. input tape—TODO
2. incoming communication tape—TODO
3. subroutine output tape—TODO

ITM can be seen as a specification or an algorithm, a machine running an ITM is an ITM instance (ITI). To communicate, an ITI can write on the externally writable tapes of other ITIs. The writing ITI then pauses exeuction, the receiving ITI begins execution. Consequently, only one ITI is running at any point in time.

2.2.2 Simulation-based Security

Simulation-based security, also known as ideal/real paradigm is a model for defining security. The model consist of a set of , the environment $\mathcal{E}$, the adversay $\mathcal{A}$, the protocol ITM Π_{real} and zero or more ideal functionalities

$\mathcal{F}_0, \mathcal{F}_1, \ldots$. The Other than the machines running the protocol under consideration, the model contains two extra entities, the environment $\mathcal{E}$ and the adversary $\mathcal{A}$. $\mathcal{E}$ can be seen as users of the protocol, it can only provide input and receive output from $\mathcal{A}$ and the protocol.

Control function TODO

2.2.3 Universal Composability

2.3 Formal Specification

The formal model follows the same structure as **??**TODO.

Protocol Π_{real}
The ETC protocol. On initialisation do the following.

- Generate public and private key pair, pk_i and sk_i.
- Set personal chain $C := \{genesis\}$.
- Set checkpoint buffer $\mathcal{C}' := \varnothing$.
- Set the facilitators F to the bootstrap nodes provided by $\mathcal{E}$.

Run the protocol as specified below after initialisation.

- Upon (`tx_init`, m, p) from $\mathcal{E}$, $C := C \cup \{new\_tx(m)\}$, send (`tx_req`, latext TX in C) to p.
- Upon (`tx_req`, m) from $p \in \mathcal{P}$, $C := C \cup \{new\_tx(m)\}$, let m' be the latest TX in C and set m to be the other half of m', send (`tx_resp`, m') to p. Output the complete TX to $\mathcal{E}$.
- Upon (`tx_resp`, m') from $p \in \mathcal{P}$, find the corresponding pair of m' and name it m, add m' as the other half of m. Output the complete TX to $\mathcal{E}$.
- Upon (`vd_init`, s) from $\mathcal{E}$, if the sequence number s does not exist in C or the block with s does not have the other half then do nothing. Otherwise send (`vd_req`, s') to p' where s' is the sequence number of the other half and p' is the counterparty.
- Upon (`vd_req`, s') from $p \in \mathcal{P}$, if the sequence number s' does not exist then do nothing. Otherwise send (`vd_resp`, $agreed\_fragment(s')$) to p.
- Upon (`vd_resp`, x) from $p \in \mathcal{P}$, run $validate(x)$ and output result to $\mathcal{E}$.

- Upon (`consensus`, r, D) from $\mathcal{F}_{\text{BFT}}$, TODO, send (`propose`)
- Upon (`checkpoint`, c) from $p \in \mathcal{P}$,

Functionality $\mathcal{F}_{\text{BFT}}$
The ideal Byzantine consensus protocol, parameterised by...

- Upon (`propose`, r, C),
- Upon (`fetch`),

2.3.1 Discussion

Global clock for synchronisation

Where there is no Byzantine corruption, we conjecture that our protocol runs in the asynchronous setting. However, security of many Byzantine consensus algorithms, especially the one we adopted—HoneyBadgerBFT, fall apart when there is dynamic corruption. In order to enforce that the corrupted machines do not change when running an instance of some Byzantine consensus algorithm, we introduce synchrony.

We make use of a global clock... TODO

Dynamic corruption only between different clock ticks is enforced by the control function.

Static corruption versus dynamic corruption

A subtlety exists when modelling the type of corruption. In dynamic corruption[3], $\mathcal{A}$ take full control of a number of machines and also learns all of their states. The states include private keys. Thus, using the dynamic corruption model from [3] we cannot guarantee security as $\mathcal{A}$ can corrupt different machines over time and eventually learn all the private keys..

First is to use static corruption, where the corrupted machines are fixed. While this circumvents our problem, it is a much weaker model. Alternatively, we modify the aforementioned dynamic corruption model and weaken the adversary's abilities. In particular, we introduce that forgetful adversary that only remembers the state of the currently corrupted nodes and forget the state of the recovered nodes.

[3] In [3], dynamic corruption is termed Byzantine corruption.

2.4 Protocol Extensions

Chapter 3

Checkpoint Consensus

3.1 Preliminaries

3.1.1 Requirements

- Permissionless
- Byzantine fault tolerant
- No PoW
- Works under churn
- Underlying data structure is TrustChain
- Detects forks or double-spends
- No step in the protocol blocks transactions
- Application independent

3.1.2 Assumptions

- Asynchronous network
- Private and authenticated channel
- We elect N consensus promoters in every round, we assume the number of faulty promoters is f and $N = 3f + 1$.
- Promoters have the complete history of the previously agreed set of transactions (TX).

3.1.3 Notation, definition and properties

- $y = \mathtt{h}(x)$ is a cryptographically secure hash function (random oracle), the domain x is infinite and the range is $y \in \{0, 2^{256} - 1\}$.

- We model our system in the permissionless setting, where each participating party has a unique identity i, and a chain B_i.

- A chain is a collection of blocks $B_i = \{b_{i,j} : j \in \{1 \dots h\}\}$, the blocks are linked together using hash pointers, similar to bitcoin. All blocks contain a reference to the previous block, the very first block with no references is the genesis block.

  The sequence number of the block begins at 0 on the genesis block and is incremented for every new block. The height of the chain is $h = |B_i|$.

- There are two sets of blocks T_i and C_i, where $T_i \cup C_i = B_i$ and $T_i \cap C_i = \emptyset$. This can be seen as the block type, where $t_{i,j}$ and $c_{i,j}$ to represent a *transaction block (TX block)* and *checkpoint block (CP block)* respectively.

- $\mathtt{typeof} : B_i \to \{\tau, \gamma\}$ returns the corresponding type of the block.

- A block of type τ is a six-tuple, i.e.

$$t_{i,j} = (\mathtt{h}(b_{i,j-1}), h_s, h_r, s_s, s_r, m).$$

  h_s and h_r denote the height (the sequence number for when the TX is made) of the sender and receiver respectively. s_s and s_r are the signatures of the sender and the receiver respectively. $i = \{s, r\}$ and $j = \{h_s, h_r\}$ depending on whether i is the sender or the receiver.

- Given two TX blocks $t_{i,j}$ and $t_{i',j'}$, if $i \neq i'$, $i = s$, $i' = r$, $h_s = h'_s$, $h_r = h'_r$, $m = m'$ and the signatures are valid, then we call them a *pair*. Note that given one TX block, the pair can be determined directly.

- If there exist two TX blocks $t_{i,j}$ and $t_{i',j'}$, where s_s and s'_s is created by the same public key, $h_s = h'_s$, but $i \neq i'$, then we call this a *fork*.

- A block of type γ is a six-tuple, i.e.

$$c_{i,j} = (\mathtt{h}(b_{i,j-1}), \mathtt{h}(\mathcal{C}_r), h, r, p, s)$$

  where $\mathcal{C}_r$ is the consensus result in round r and $p \in 0, 1$ which indicates whether i wish to become a promoter in the following consensus round, finally s is a signature of the block.

- We define
$$\texttt{newtx} : B_i \times B_j \times M \to B_i \times B_j$$
as a function that creates new TX blocks. Its functionality is to extend the given chains using the following rule. If the input is (B_s, B_r, m) then the output is (B'_s, B'_r) where $B'_s = \{(\texttt{h}(b_{s,h_s}), h_s{+}1, h_r{+}1, s_s, s_r, m)\} \cup B_s$, $B'_r = \{(\texttt{h}(b_{r,h_r}), h_s + 1, h_r + 1, s_s, s_r, m)\} \cup B_r$, $h_s = |B_s|$ and $h_r = |B_r|$.

- We define
$$\texttt{newcp} : B_i \times \mathbb{R}_{\geq 1} \times \{0, 1\} \to B_i$$
as a function that creates new CP blocks. Concretely, given (B_i, r, p), it results in $\{(\texttt{h}(b_{i,h}), H(\mathcal{C}_r), p)\} \cup B_i$ where $h = |B_i|$, and $\mathcal{C}_r$ is the latest consensus result at round r.

- Note that in the actual system, `newtx` and `newcp` perform a state transition.

- We define
$$\texttt{round} : C_i \to \mathbb{R}_{\geq 1}$$
as a function that gets the consensus round number used to create the given CP block.

- The CP blocks that follows a pair of TX blocks must be created using the same $\mathcal{C}_r$, otherwise the transaction is invalid. Concretely, given a pair $t_{i,j}$ and $t_{i',j'}$, then there exist $c_{i,k}$ and $c_{i',k'}$ where $j < k$, $j' < k'$, $\{\texttt{typeof}(b_{i,x}) : x \in \{j, \dots k-1\}\}$ are all τ, $\{\texttt{typeof}(b_{i',x}) : x \in \{j', \dots k'{-}1\}\}$ are all τ and $\texttt{round}(c_{i,k}) = \texttt{round}(c_{i',k'})$. This constraint is not the result of the consensus protocol, but the validation protocol.

- The result of a consensus in round r is a set of CP blocks. Namely,
$$\mathcal{C}_r = (r, \{c_{i,j} : \text{agreed by the promoters}\}).$$

- A TX block can be valid, invalid or unknown. All TX blocks begin as unknown, they can be validated using our validation protocol.

- A TX block has two *ancestor* blocks. Let a pair be $t_{i,j}$ and $t_{i',j'}$, then $(b_{i,j-1}, b_{i',j'-1})$ is the input block of $t_{i,j}$.

- A CP block has one *ancestor* block, that is simply the block with the previous sequence number, i.e. the ancestor of $c_{i,j}$ is $b_{i,j-1}$.

- Every TX block $t_{i,j}$ is enclosed by two CP blocks $(c_{i,a}, c_{i,b})$, where
$$a = \underset{k, k<j, \texttt{typeof}(b_{i,k})=\gamma}{\arg\min} (j - k),$$

$$b = \underset{k,k>j,\texttt{typeof}(b_{i,k})=\gamma}{\arg\min} (k-j).$$

We call this the *enclosure* of $t_{i,j}$.

- The *piece* of $t_{i,j}$ defined by the enclosure $(c_{i,a}, c_{i,b})$ is $\{b_{i,j} : a \leq j \leq b\}$. We define
$$\texttt{pieces} : B_i \to P(B_i)$$
as the function that returns the pieces (P denotes power set).

- Every TX block $t_{i,j}$ is enclosed by two *agreed* CP blocks $(c_{i,a}, c_{i,b})$ (CP blocks that are in some consensus result), where
$$a = \underset{k,k<j,\texttt{typeof}(b_{i,k})=\gamma}{\arg\min} (j-k),$$
$$b = \underset{k,k>j,\texttt{typeof}(b_{i,k})=\gamma}{\arg\min} (k-j).$$
We call this the *agreed enclosure* of $t_{i,j}$.

- The *agreed piece* of $t_{i,j}$ defined by the agreed enclosure $(c_{i,a}, c_{i,b})$ is $\{b_{i,j} : a \leq j \leq b\}$. Note that *piece* $\subseteq$ *agreed piece* for some TX block. We define
$$\texttt{a-pieces} : B_i \to P(B_i)$$
as the function that returns the agreed pieces.

3.2 Checkpoint consensus

3.2.1 Luck value

First we define the luck value $l_{i,j} = \texttt{h}(k_i||c_{i,j})$, where k is the public key of i. A lower luck value equates to higher luck. We assume an application agnostic system and do not attempt to defend against the sybil attack.

An alternative is to use *proof of work*. This defends the sybil attack. But an incentive is needed for the nodes that expend their CPU resources.

3.2.2 Promoter registration

Node i can register as a promoter when the latest consensus result is announced (suppose after the completion of round $r-1$), then it generates a new block $b_{i,j} = \texttt{newcp}(B_i, r-1, 1)$. The current promoters (in round r) may decide to include $b_{i,j}$ in the new consensus result. If b is in it, then i becomes one of the promoter of round $r+1$.

We can fix the number of promoters to N by sorting the promoters by their "luck value"' and taking the first N.

3.2.3 Promoter invitation

The output of promoter registration is a random set of N promoters. If $1/3$ of the population is malicious, then we cannot guarantee that the chosen promoters satisfy the $< n/3$ requirement.

Promoter invitation is an attempt to involve human in the protocol. A naive method method is to use N tickets, and then distribute them to trusted nodes. Nodes with the ticket can forward it to others. We have to rely on the humans to always forward the tickets to other honest humans. Finally, the nodes that hold a ticket are promoters. The result is that we have a permissioned system.

3.2.4 Setup phase

The setup phase should satisfy the BFT conditions regarding the promoter selection, that is:

1. *Agreement*: If any correct node outputs a promoter p, then every correct node outputs p.

2. *Total Order*: If one correct node outputs the sequence of promoters $\{p_1, p_i, \ldots, p_n\}$ and another has output $\{p'_0, p'_1, \ldots, p'_{n'}\}$, then $p_i = p'_i$ for $i \leq \min(n, n')$.

3. *Liveness*: All $N - f$ correct nodes terminate eventually.

We begin in the state where $\mathcal{C}_{r-1}$ has just been agreed but has not been disseminated yet. The exact technique to disseminate $\mathcal{C}_{r-1}$ is irrelevant, broadcasting or gossiping are both sufficient. In fact, dissemination is not necessary, nodes interested in the result can simply query the promoters that created $\mathcal{C}_{r-1}$.

The consensus result $\mathcal{C}_{r-1}$ does not contain all the signatures of the promoters. Thus, the signatures must also be disseminated by the promoters. This can be done in the same way as consensus result dissemination. Nodes must collect $f + 1$ signatures in order to trust $\mathcal{C}_{r-1}$ because there is at least one honest node in $f + 1$ nodes. If there is a sufficient number of signatures on the consensus result, we say it is *valid*.

Lemma 1. *If a node sees a valid $\mathcal{C}_{r-1}$ and another node sees a valid $\mathcal{C}'_{r-1}$, then $\mathcal{C}_{r-1} = \mathcal{C}'_{r-1}$.*

Proof. $\mathcal{C}_{r-1}$ and $\mathcal{C}'_{r-1}$ are signed by at two groups of $f + 1$ promoters (does not have to overlap) from round $r - 1$. In both groups, at least one node is honest. Thus the consensus result is what is actually agreed. □

Lemma 2. *Eventually all node sees a valid $\mathcal{C}_{r-1}$.*

Proof. (sketch) Liveness is satisfied because $\mathcal{C}_{r-1}$ is eventually propagated to all node by gossiping. □

The potential promoters now need to first discover whether they are the first N lucky promoters.

Lemma 3. *The new set of promoter for the next consensus round is consistent with respect to all the nodes in the network.*

Proof. (sketch) All nodes use the same deterministic function to compute the luck value. □

Nodes should wait for some time to collect the CP blocks, so they wait for some time Δ before moving on to the next phase. Note that promoters waiting for a some time Δ to collect transactions does not violate the asynchronous assumption because this behaviour can be seen as a long delay in the asynchronous system.

Corollary 1. *Setup phase satisfies agreement and total order because the protocol is run deterministicly on the same input (lemma 1, lemma 3). It also satisfies liveness due to lemma 2.*

3.2.5 Consensus phase

The consensus phase should satisfy the BFT conditions regarding the CP blocks, that is:

1. *Agreement*: If any correct node outputs a CP block c, then every correct node outputs c.

2. *Total Order*: If one correct node outputs the sequence of CP blocks $\{c_1, c_i, \ldots, c_n\}$ and another has output $\{c'_0, c'_1, \ldots, c'_{n'}\}$, then $c_i = c'_i$ for $i \leq \min(n, n')$.

3. *Liveness*: All $N - f$ correct nodes terminate eventually.

We need an atomic broadcast algorithm for the consensus phase. We use a similar but simplified construction as [7], where atomic broadcast is constructed from the reliable broadcast[1] [2] and asynchronous common subset (ACS). The ACS protocol requires a binary Byzantine agreement protocol, and for that it needs a trusted dealer to distributed the secret shares. Promoters can check whether the secret shares are valid, but they cannot prevent the dealer from disclosing the secrets.

There are techniques that uses no dealers. First is to use PBFT [4], but we must change our asynchronous assumption into the weak synchrony assumption. Most likely this is not possible because of the "When to start?"

[1] Reliable broadcast solves the Byzantine generals problem.

problem. It's difficult to give a bounded delay for propagating the consensus result.

Second is to use an inefficient binary Byzantine agreement protocol where its message complexity is $O(N^3)$ rather than $O(N^2)$ and becomes a bottleneck. Or a suboptimal one, e.g. $n/5$ instead of $n/3$.

Suppose we use a dealer, what is the effect to the algorithm if the dealer is malicious?

Lemma 4. *Consensus phase satisfies agreement, total order and liveness.*

Proof. Defer proof? Refer to the papers. □

Theorem 1. *Checkpoint consensus satisfies agreement, total order and liveness.*

Proof. (sketch) Both phases are asynchronous so we do not need to make assumptions on when begin phase begins. Both phases also satisfy the agreement, total order and liveness. □

3.3 Validation

Since we reach consensus on checkpoints rather than all the transactions, we need to detect fraud, such as forks. Further, we need to ensure the system is secure in a sense that valid transactions cannot be forged into invalid transactions once it has reached consensus, and vice versa.

First, we define the requirements of a valid transaction $t_{i,j}$ as follows, much of it is derived from the TrustChain model.

1. There exist a pair $t_{i',j'}$ that satisfies the pair definition.
2. $t_{i,j}$ and $t_{i',j'}$ is created using `newtx`, i.e. valid signatures and hash pointers, etc..
3. There exist an agreed piece that contains $t_{i,j}$ and another agreed piece that contains $t_{i',j'}$. All the blocks in the agreed pieces have valid hash pointers, the blocks in the pieces do not need to be valid.
4. The CP blocks $c_{i,k}, c_{i',k'}$ that immediately follow $t_{i,j}$ and $t_{i',j'}$ are created using `newcp` using the same consensus result $\mathcal{C}_r$ as input and are in the agreed pieces.

Blocks that do not satisfy the definition above are not necessarily invalid. It may be the case that the validity cannot be determined due to incomplete information. For such cases we say its validity is unknown. Now we define an invalid transaction.

1. TODO

Now we define the properties that are desired by the validation protocol.

1. *Correctness*: The validation protocol outputs the correct result according to the aforementioned requirements.

2. *Agreement*: If any correct node decides on the validity of a transaction, then all other correct nodes are able to reach the same conclusion.

3. *Liveness*: Any valid transactions can be validated eventually.

4. *Unforgeability*: If some transaction is determined to be valid, then it cannot be changed to an invalid transaction, the opposite also applies.

Note that the input of the validation protocol is a TX block, so these properties hold with respect to TX blocks. In practice, if a node has a set of TX blocks that are in the unknown state, then it must run the validation protocol to determine whether they are valid. Further, these conditions do not imply all invalid transactions (forks) can be found globally, only the validity of the TX blocks that the honest nodes are interested in can be determined. The advantage of this scheme is that it saves a lot of computational and bandwidth costs because nodes only run the validation protocol on the TX blocks of their own interest.

3.3.1 Validation protocol

Assume node u is aware of all the past consensus results $\mathcal{C}_r$. Suppose u wish to validate $t_{i,j}$. It performs the following.

1. Determine the pair $t_{i',j'}$.

2. Find the agreed enclosure for $t_{i,j}$ and $t_{i',j'}$ from $\mathcal{C}_r$, otherwise return "unknown".

3. Query i and i' for the agreed pieces and ensure hash pointers are correct. Otherwise return "unknown".

4. Check that $t_{i,j}$ and $t_{i',j'}$ are in the agreed pieces and are created correctly using `newtx`. Otherwise return "invalid".

5. Check the checkpoints $c_{i,k}$ and $c_{i',k'}$ that immediately follow $t_{i,j}$ and $t_{i',j'}$ are in the agreed pieces and are created in the same round, i.e. $\texttt{round}(c_{i,k}) = \texttt{round}(c_{i',k'})$. Otherwise return "invalid".

6. Return "valid".

Most likely $u = i$ or $u = i'$, because they are incentivised to check the validity of their TX blocks that are of unknown validity.

3.3.2 Analysis

In this section we analyse the validation protocol with respect to the four properties in section 3.3.

Lemma 5. *The validation protocol is correct.*

Proof. Correctness follows directly from the protocol specification, namely it directly implements the validation requirements. □

Lemma 6. *The validation protocol satisfies the agreement property.*

Proof. We proof by contradiction. Suppose two nodes i and j where $i \neq j$ runs the protocol on the same TX block $t_{k,l}$, i outputs "valid" and j outputs "invalid". i and j would have both picked the same enclosure $c_{k,a}$ and $c_{k,b}$, as these are determined from the consensus result. Then k must produce two agreed pieces that start with $c_{k,a}$ and end with $c_{k,b}$ with valid hash pointers, but one satisfies the validation protocol and the other one doesn't (e.g. missing $t_{k,l}$). Producing these two pieces is equivalent to producing a collision. Due to the properties of cryptographic hash functions this is not possible, thus a contradiction. □

Lemma 7. *The validation protocol does not satisfy the liveness property.*

Proof. (sketch) Nodes may be offline. □

There are many ways around the liveness. As long as the TX is validated once, then the agreed pieces can be gossiped.

Lemma 8. *A valid TX block cannot be made into an invalid TX block and vice versa.*

Proof. We proof by contradiction. Suppose $t_{k,l}$ is determined to be valid using the enclosure $c_{k,a}$ and $c_{k,b}$ and the agreed piece p. Then $t_{k,l}$ is forged into an invalid transaction. To forge it into an invalid transaction the attacker must either tamper with the data within the block or remove $t_{k,l}$ from p. In other words create p' where $p \neq p'$, enclosed by $c_{k,a}$ and $c_{k,b}$, and has valid hash pointers. Producing p' is equivalent to finding a collision, thus a contradiction.

TODO proof "vice versa". □

3.3.3 Time and message complexity

For a single transaction, the message complexity is linear with respect to the size of the agreed pieces for both part of the TX pair. Time complexity is constant since we only perfrom two queries (to i and i').

3.4 Fraud detection

Existence testing detects fraud when a single party of the TX is malicious. To detect fraud when both parties are malicious, every node performs existence testing on all TX blocks between CP blocks as if they are some other node. The assignment is determined using CP block digest where every node is assigned to the nearest node that has a higher digest, cycle back if there is no higher digest.

If the number of malicious parties is low, then it's more probable that a fork is detected. If where there are a large number of malicious parties, we repeat the random sampling process.

3.5 Privacy

Currently the validation protocol requires TX blocks to be sent in full. We can do better by modifying how the chain is computed to enhance user's privacy. First, recall that hash pointers are calculated on the whole block (TX or CP), i.e. $\mathtt{h}(b_{s,h_s})$. In the validation stage, b_{s,h_s} must be provided to recompuate the digest and check that it matches the block at $h_s - 1$. To improve this algorithm, we introduce a new property *digest* for each block, which simply is $d_{s,h_s} = \mathtt{h}(b_{s,h_s})$, d is computed dynamically. Then the chain of hash pointers is formed using $\mathtt{h}(d_{s,h_s})$. During the validation phase, there is no need to provide the complete block, only the *digest* is sufficient. Of course, the pair of the TX of interest must be provided in full.

Chapter 4

Implementation and Experimental Results

4.1 Implementation

Python. Twisted.

4.2 Evaluation

- How fast is the consensus algorithm? Possibly plot graph of time versus the number of nodes.
- Does the promoter registration phase add a lot of extra overhead?
- What's the rate of transaction such that they can be verified "on time", i.e. without a growing backlog?
- Our global validation rate is somewhat equivalent to the transaction rate in other systems. Does the validation rate scale with respect to the number of nodes? In theory it should. Plot validation rate vs number of nodes, we expect it to be almost linear.

Bibliography

[1] Visa inc. at a glance.

[2] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162. ACM, 1984.

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

[4] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[5] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[6] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377. ACM, 1982.

[7] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[8] Haifeng Yu, Michael Kaminsky, Phillip B Gibbons, and Abraham Flaxman. Sybilguard: defending against sybil attacks via social networks. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 267–278. ACM, 2006.

Appendix A

Consensus Example

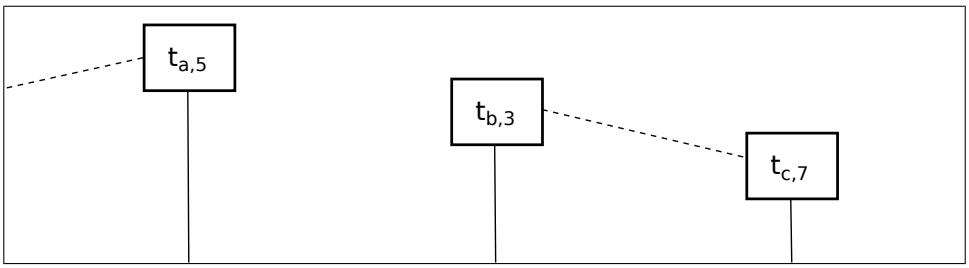

Figure A.1: Initial state

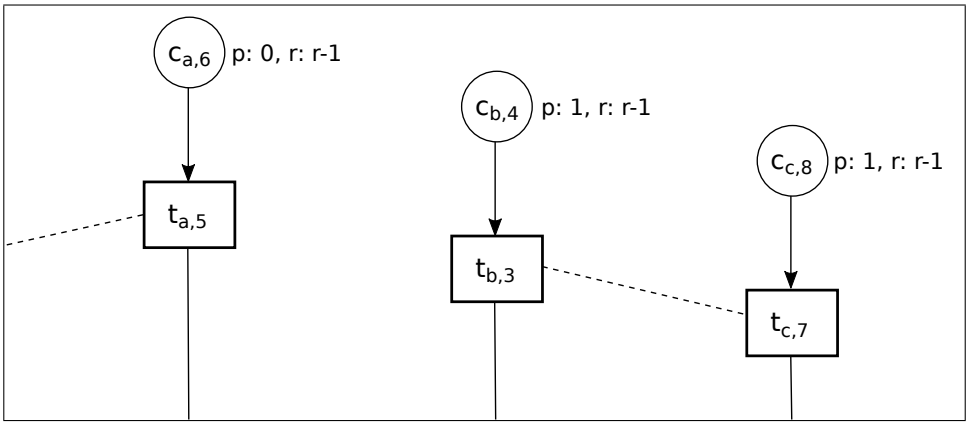

Figure A.2: Initial state

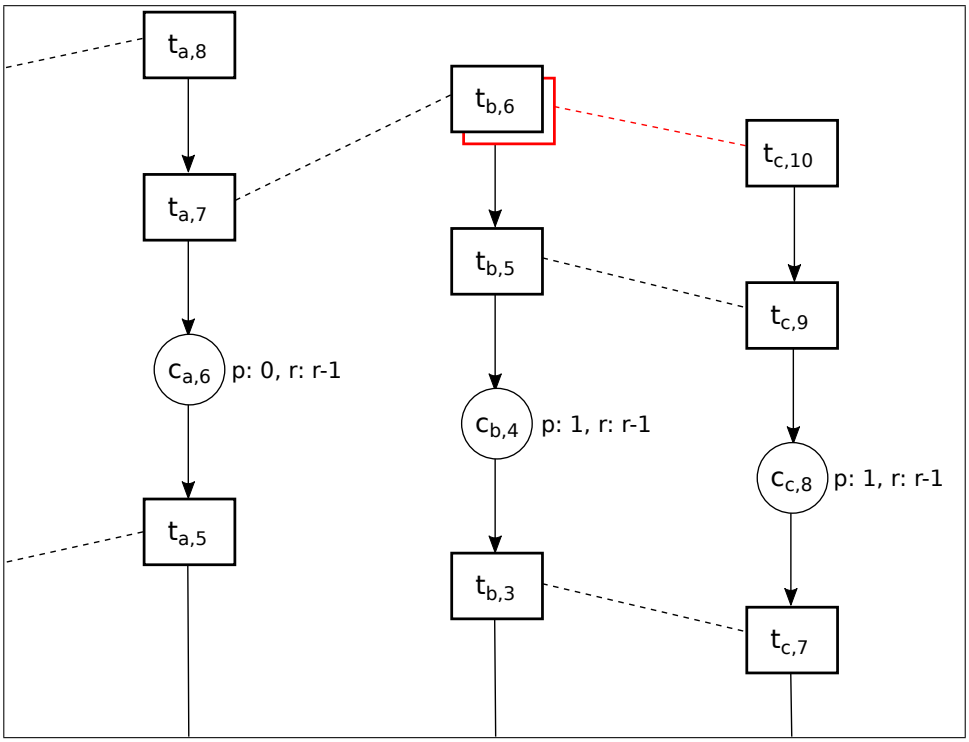

Figure A.3: Initial state

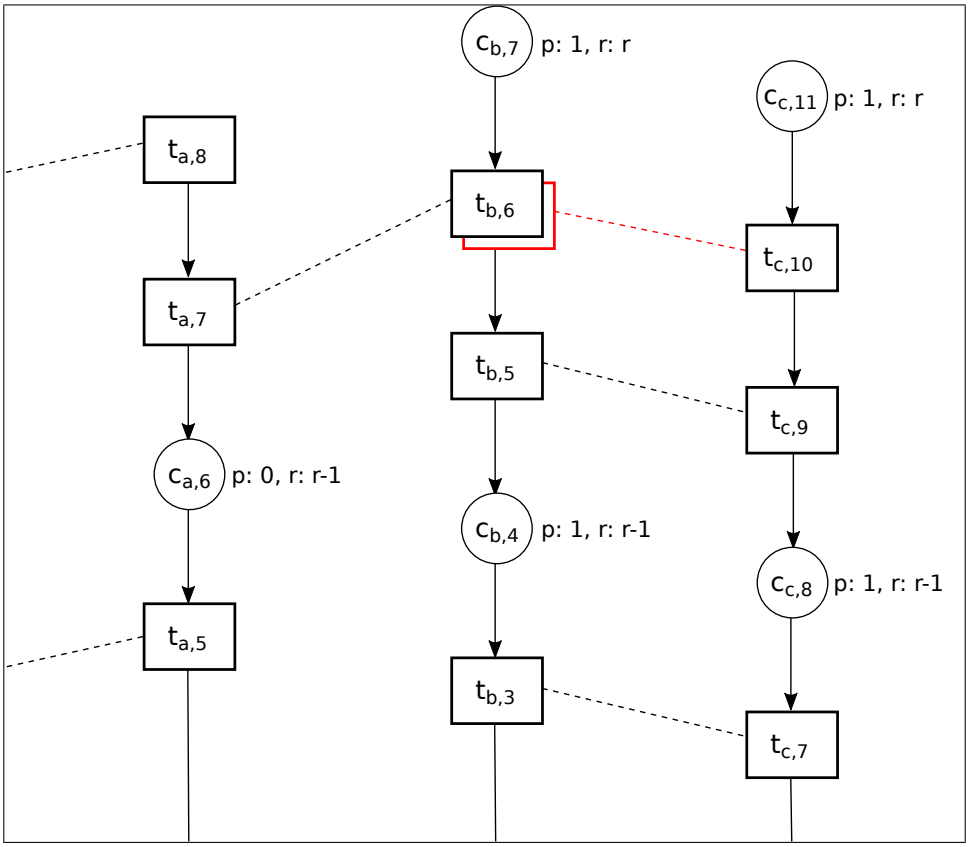

Figure A.4: Initial state