TODO TITLE

TODO AUTHOR

TU Delft
Delft
University of
Technology

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

7th 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
7th 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 [4]. 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 [7]. 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 intuitive overview of the architecture in Section 2.1. Next, we give the formal description, starting with the model and assumptions in Section 2.2. Then, the three protocols which make up the complete system, namely consensus protocol (Section 2.3), transaction protocol (Section 2.4) and validation protocol (Section 2.5). Finally, the possible extensions are described in Section 2.6.

2.1 System Overview

The system consist of one data structure—Extended TrustChain, and three protocols—consensus protocol, transaction protocol and validation protocol. We first describe each component individually and then explain how they fit together in Section 2.1.4.

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 a "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*. As a result, a single transaction generates two TX blocks, one on each party's chain. An example of 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.

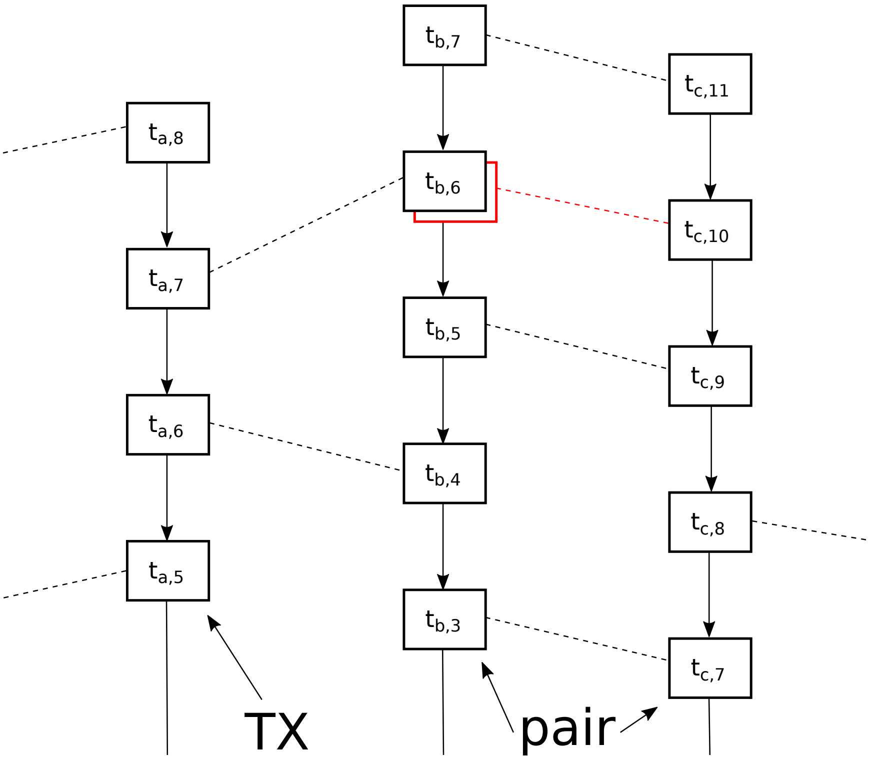

Figure 2.1: Every block is denoted by $t_{i,j}$, where i is the node ID and j is the sequence number of the block. Thus we have three nodes and three corresponding chains in this example. The arrows represent hash pointers and the dotted lines represent references. The blocks at the ends of one dotted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

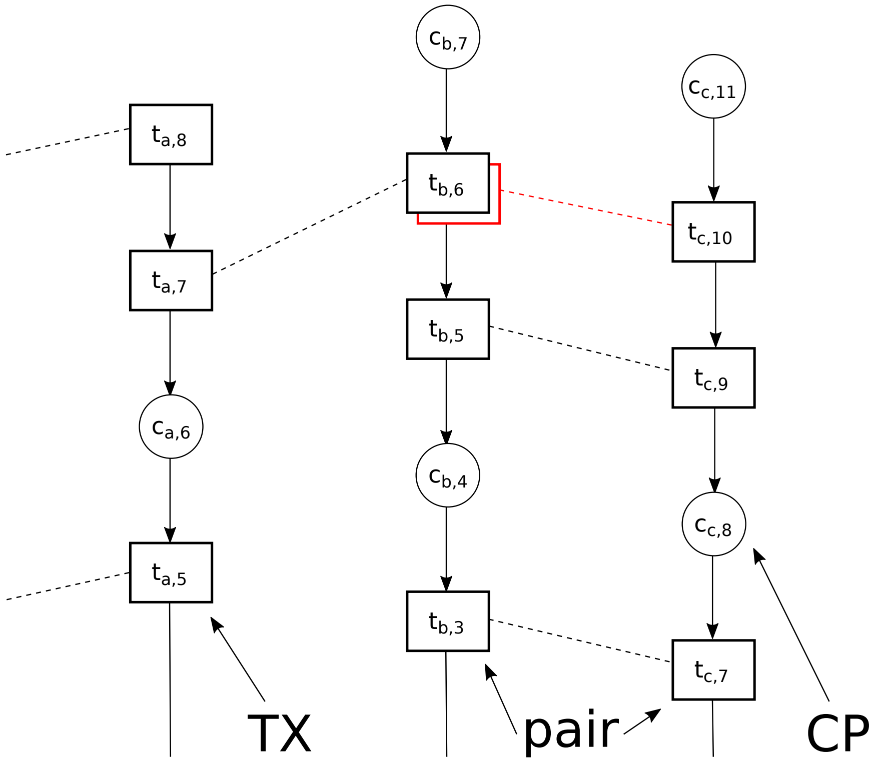

Figure 2.2: The circles represent CP blocks, they also have hash pointers (arrow) but do not have references (dotted line). Note that the sequence number counter do not change, it is shared with TX blocks.

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—checkpoint (CP) block. In contract to TX blocks, CP blocks do not store transactions or contain references. Their purpose is to 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.

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 the CP blocks as proposals 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 and Validation

The TX protocol is a simple request and response protocol. The nodes exchange one round of messages and create new TX blocks on their respective chains. Thus, as we mentioned before, one transaction should result in two TX blocks.

The consensus and transaction protocol by themselves do not provide a mechanism to detect forks or other forms of tamperaing. Thus we need a validation protocol to counteract malicious behaviour. When a node wish to validate one of its TX, it asks the counterparty for the *fragment* of the TX. A fragment of a TX is a section of the chain beginning and ending with CP blocks that contains the TX. Upon the counterparty's response, the node checks that the CP blocks are in consensus, the hash pointers are valid and his TX is actually in the fragment. The TX is valid if these conditions are satisfied. Intuitively, this works because it is hard (because hash collision is hard) to create a different chain that begins and ends with the same two CP blocks but with a different middle section.

2.1.4 Combined Protocol

The final protocol is essentially the concurrent composition of the three aforementioned protocols, all making use the Extended TrustChain data structure.

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 Model and Assumptions

For notational clarify, we use the following convention (adapted from [5]) throughout this work.

- Lower case (e.g. x) denotes a scalar object.
- Upper case (e.g. X) denotes a set or a constant.
- Sans serif (e.g. $\mathsf{fn}(\cdot)$) denotes a function.
- Typewrite (e.g. `ack`) denotes message type.

We assume a distributed network where nodes are fully connected, the channels are reliable[2], but messages may be re-ordered and delayed by at most some time Δ. Nodes are identified by their public key thus we assume there exist a Public Key Infrastructure (PKI). TODO this is weak synchrony? In addition, we assume the existance of cryptographically secure

[2] Reliability can be achieved in unreliable networks by resending messages or using some error correction code.

hash function $\mathsf{H}(\cdot)$, that has preimage resistance, second preimage resistance and collision resistance.

In our model we consider N nodes, which is the population size. n of them are facilitators, t out of n are malicious and the inequality $n \geq 3t + 1$ must hold.

The primary data structure used in our system is Extended TrustChain. Each node u has a public and private key pair—pk_u and sk_u, and a chain B_u. The chain consist of blocks $B_u = \{b_{u,i} : k \in \{0, \dots, h-1\}\}$, where $b_{u,i}$ is the ith block of u, and $h = |B_u|$. We often use $b_{u,h}$ to denote the latest block. There are two types of blocks, TX blocks and CP blocks. If T_u is the set of TX blocks of u and C_u is the set of CP blocks of u, then it must be the case that $T_u \cup C_u = B_u$ and $T_u \cap C_u = \varnothing$. The notation $b_{u,i}$ is generic over the block type. We assume there exist a function $\mathsf{typeof} : B_u \to \{\tau, \gamma\}$ that returns the type of the block, where τ represents the TX type and γ represents the CP type.

2.2.1 TX Block

The TX block is a six-tuple, i.e $t_{u,i} = \langle \mathsf{H}(b_{u,i-1}), seq_u, txid, pk_v, m, sig_u \rangle$. We describe each item in turn.

1. $\mathsf{H}(b_{u,i-1})$ is the hash pointer to the previous block.

2. seq_u is the sequence which should equal i.

3. $txid$ is a cryptographically secure random number representing the transaction identifier.

4. pk_v is the public key of the counterparty.

5. m is the transaction message.

6. sig_u is the signature created using sk_u on the concatination of the binary representation of the five items above.

The fact that we have no constraint on the content of m is in alignment with our design goal—application neutrality.

TX blocks come in pairs. In particular, for every block $t_{u,i} = \langle \mathsf{H}(b_{u,i-1}), seq_u, txid, pk_v, m, sig_u \rangle$ there exist one and only one *pair* $t_{v,j} = \langle \mathsf{H}(b_{v,j-1}), seq_v, txid, pk_u, m, sig_v \rangle$. Note that the $txid$ and m are the same, and the public keys refer to each other. Thus, given a TX block, these properties allow us to identify its pair.

2.2.2 CP Block

The CP block is a five-tuple, i.e. $c_{u,i} = \langle \mathsf{H}(b_{u,i-1}), seq_u, \mathsf{H}(\mathcal{C}_r), r, sig_u \rangle$, where $\mathcal{C}_r$ is the consensus result in round r, the other items are the same as the

TX block definition. Note that unlike in our prior work [6], CP blocks and TX blocks do not have independent sequence numbers.

The genesis block in the chain must be a CP block in the form of $c_{u,0} = \langle \mathsf{H}(\perp), 0, \mathsf{H}(\perp), 0, sig_u \rangle$ where $\mathsf{H}(\perp)$ can be interpreted as applying the hash function on an empty string. The genesis block is unique due to every node due to sig_u.

2.2.3 Consensus Result

Our consensus protocol runs in rounds as discussed in Section 2.1. Every round is identified by a round number r, which is incremented on every new round. The consensus result is a tuple, i.e. $\mathcal{C}_r = \langle r, C \rangle$, where C is a set of CP blocks agreed by the facilitators of round r.

2.2.4 Chain Properties

Here we define a few important properties which results from the interleaving nature of CP and TX blocks.

If there exist a tuple $\langle c_{u,a}, c_{u,b} \rangle$ for a TX block $t_{u,i}$, where

$$a = \underset{k, k<i, \texttt{typeof}(b_{u,k})=\gamma}{\arg\min} (i - k)$$

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

then $\langle c_{u,a}, c_{u,b} \rangle$ is the *enclosure* of $t_{u,i}$. Some TX blocks may not have any subsequent CP blocks, then its enclosure is $\perp$.

If the enclosure of some TX block is $\langle c_{u,a}, c_{u,b} \rangle$, then its *fragment* is computed as $\{b_{u,i} : a \leq i \leq b\}$. For convenience, the function $\mathsf{fragment}(\cdot)$ represents the fragment of some TX block if it exists, otherwise $\perp$.

Agreed enclosure is the same as enclosure with an extra constraint where the CP blocks must be in some consensus result $\mathcal{C}_r$. Similarly, *agreed fragment* is computed using agreed enclosure. We define its function to be $\mathsf{agreed\_fragment}(\cdot)$

The length of the fragment is constrained by L, namely $\forall t |\mathsf{fragment}(t)| \leq L$. The purpose to prevent spam and encourage nodes to create more CP blocks. L should be sufficiently high so that busy nodes are not hindered by it.

2.3 Consensus Protocol

Our consensus protocol runs on top of the model describe before, it makes heavy use of CP blocks. We describe the protocol step by step. Starting with the bootstrap phase and then moving on to the actual consensus phase.

2.3.1 Bootstrap Phase

Recall that facilitators are computed from the consensus result, but the consensus result is agreed by the facilitators. Thus we have a dependency cycle. The goal of the bootstrap phase is to give us a starting point for the cycle.

Our bootstrap phase runs as follows. First we assume all N nodes start simultaneously. The facilitators for round 0 are hard coded in the program, in practice this could be the machines controlled by the developer. If some node u not a facilitator, it sends the message $\langle \texttt{cp\_msg}, c_{u,0} \rangle$ to all the facilitators. If u is a facilitator, it does the same, but also waits for some time $\Delta + 1$ to collect messages of type `cp_msg`. After $\Delta + 1$ elapses, it begins the atomic broadcast protocol using the collected CP blocks as the proposal. $\mathcal{C}_0$ is agreed at the end of the protocol. This concludes the bootstrap phase.

The bootstrap phase can be seen as an idealised version of the setup phase combined with the consensus phase, which we describe next.

2.3.2 Setup phase

The setup phase begins immediately after some consensus result is agreed, but not yet disseminate. This could be right after the bootstrap phase. The goal is to reach agreement on a list of new facilitators between every node. Concretely, for every consensus round r, the following is required.

1. *Agreement*: If any correct node receives $\mathcal{C}_r$, then every correct node receives $\mathcal{C}_r$.

2. *Liveness*: All $N - t$ correct nodes receives a value eventually.

The setup phase works as follows. Assume that facilitators of round $r - 1$ have just agreed on $\mathcal{C}_{r-1}$. They then immediately broadcast two messages to all the nodes— first the consensus message $\langle \texttt{cons\_msg}, \mathcal{C}_{r-1} \rangle$, and second the signature message $\langle \texttt{cons\_sig}, r, sig \rangle$.

We make two remarks regarding those message. First, broadcasting is inefficient in terms of message complexity. Gossiping is often better in practice but it introduces additional complexity due to its probabilistic nature. Thus, to keep the analysis clear in **??**, we use broadcasting. Second, recall that channels are not authenticated, and there are no signatures in $\mathcal{C}_{r-1}$. If a non-facilitator sees some $\mathcal{C}_{r-1}$, it cannot immediately trust it because it may have been forged. Thus, To guarantee authenticity, every facilitator sends an additional message that is the signature of $\mathcal{C}_r$.

Continuing with the setup phase. Upon receiving $\mathcal{C}_{r-1}$ and at least $t + 1$ valid signatures by u, u performs two asks. First, it computes the new facilitators using $\mathsf{derive\_facilitator}(C, n)$ (Algorithm 1) and updates its facilitator list to the result. Second, it creates a new CP block using $\mathsf{new\_cp}(\mathcal{C}_{r-1}, r-1)$ (Algorithm 2). This concludes the setup phase.

Algorithm 1 Function $\mathsf{derive\_facilitator}(C, n)$ takes a list of CP blocks C and an integer n, sort evey element in C by its luck value (the λ-expression), and outputs the smallest n elements.

```
$\mathbf{take}(n, \mathbf{sort}(\mathbf{map}(\lambda x.\mathsf{H}(x||pk \text{ of x}), C)))$
```

Algorithm 2 Function $\mathsf{new\_cp}(\mathcal{C}_r, r)$ runs in the context of the caller u. TODO

```
$h \leftarrow |B_u|$
$c_{u,h} \leftarrow \langle \mathsf{H}(b_{u,h-1}), h, \mathsf{H}(\mathcal{C}_r), r, sig_u \rangle$
$B_u \leftarrow B_u \cup c_{u,h}$
```

2.3.3 Consensus Phase

The consensus phase begins after the new facilitators are selected. Assuming the facilitators are of round r, the goal is to agree on some $\mathcal{C}_r$ between the facilitators. There are two scenarios in the consensus phase. First, if node u is not the facilitator, it sends $\langle \texttt{cp\_msg}, c_{u,h} \rangle$ to all the facilitators. Second if the node is a facilitator, it waits for some duration δ and collect messages of type cp_msg. After δ is elapsed, it begins the atomic broadcast algorithm. In particular, the facilitators must satisfy the following conditions.

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 $\{c_1, c_i, \dots, c_n\}$ and another has output $\{c'_0, c'_1, \dots, c'_{n'}\}$, then $c_i = c'_i$ for $i \leq \min(n, n')$.

3. *Liveness*: All $n - t$ correct nodes terminate eventually.

We remark that this procedure is the same as what is described in the bootstrap phase (Section 2.3.1), but the precondition and the waiting time is different. In particular, the bootstrap stage assumed that every node initiated and sent the cp_msg simultaneously, but here we make no such assumption. Further, the bootstrap waits for duration Δ in order to collect all the CP blocks, here we wait for some δ which is a system variable. We make no assumption on the value of δ in relation to Δ, thus it may be the case that not all cp_msg are delivered before the the facilitators begins the atomic broadcast protocol.

At the core of the consensus phase is the atomic broadcast protocol. While using any atomic broadcast protocol will suffice, we use a (minimal) simplification of HoneyBadgerBFT as it is designed for blockchain systems and runs in fully asynchronous networks. The transactions in HoneyBadgerBFT are first queued in a buffer, the main consensus algorithm starts only when the buffer reaches an optimal size. We do not have an infinite stream of CP blocks, thus buffering is unsuitable. The main algorithms are kept the same, namely Asynchronous Common Subset, Reliable Broadcast and Binary Agreement. At the end of the consensus sphase, some $\mathcal{C}_r$ should be agreed upon. This bring us back to the setup phase and the cycle can be started again.

2.4 Transaction Protocol

The TX protocol, shown in Algorithm 4, is run by all nodes. It is also known as True Halves TODO. Node that wish to initiate a transaction calls $\mathsf{new\_tx}(pk_v, m, txid)$ (Algorithm 3) with the intended counterparty v identified by pk_v and message m. $txid$ should be a uniformly distributed random value, i.e. $txid \in_R \{0,1\}^{256}$. Then the initiator sends $\langle \texttt{tx\_req}, t_{u,h} \rangle$ to v.

Algorithm 3 Function $\mathsf{new\_tx}(pk_v, m, txid)$ generates a new TX block and appends it to the caller u's chain. It is executed in the private context of u, i.e. it has access to the sk_u and B_u. The necessary arguments are the public key of the counterparty pk_v, the transaction message m and the transaction identifier $txid$.

```
  $h \leftarrow |B_u|$
  $t_{u,h} \leftarrow \langle \mathsf{H}(b_{u,h-1}), h, txid, pk_v, m, sig_u \rangle$
  $B_u \leftarrow B_u \cup \{t_{u,h}\}$
```

Algorithm 4 The TX protocol which runs in the context of node u.

```
Upon $\langle \texttt{tx\_req}, t_{v,j} \rangle$
  $txid, pk_v, m \leftarrow t_{v,j}$                ▷ Unpack the TX
  $\mathsf{new\_tx}(pk_u, m, txid)$
  store $t_{v,j}$ as the other half of $t_{u,h}$
  send $\langle \texttt{tx\_reqp}, t_{u,h} \rangle$ to $v$
Upon $\langle \texttt{tx\_resp}, t_{v,j} \rangle$
  $txid, pk_v, m \leftarrow t_{v,j}$                ▷ Unpack the TX
  store $t_{v,j}$ as the other half of the TX with identifier $txid$
```

A key feature of the TX protocol is that it is non-blocking. TODO

2.5 Validation Protocol

2.6 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 $j < k$, $j' < k'$, $\{\texttt{typeof}(b_{i,x}) : x \in \{j, \ldots k - 1\}\}$ are all τ, $\{\texttt{typeof}(b_{i',x}) : x \in \{j', \ldots k'{-}1\}\}$ 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
$$a = \operatorname*{arg\,min}_{k, k<j, \texttt{typeof}(b_{i,k})=\gamma} (j - k),$$

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

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

  $$a = \operatorname*{arg\,min}_{k, k<j, \texttt{typeof}(b_{i,k})=\gamma} (j - k),$$

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

  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 $\{c_1, c_i, \dots, c_n\}$ and another has output $\{c'_0, c'_1, \dots, c'_{n'}\}$, 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 [5], 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 [3], 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. $\texttt{h}(b_{s,h_s})$. 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 $d_{s,h_s} = \texttt{h}(b_{s,h_s})$, d is computed dynamically. Then the chain of hash pointers is formed using $\texttt{h}(d_{s,h_s})$. 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] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

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

[5] 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.

[6] Zhijie Ren, Kelong Cong, Johan Pouwelse, and Zekeriya Erkin. Implicit consensus: Blockchain with unbounded throughput, 2017.

[7] 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

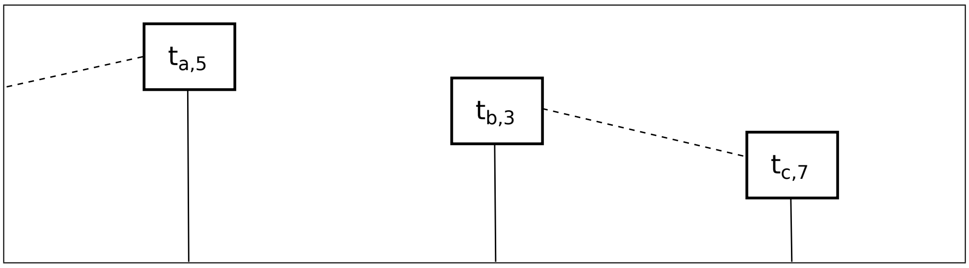

Figure A.1: Initial state

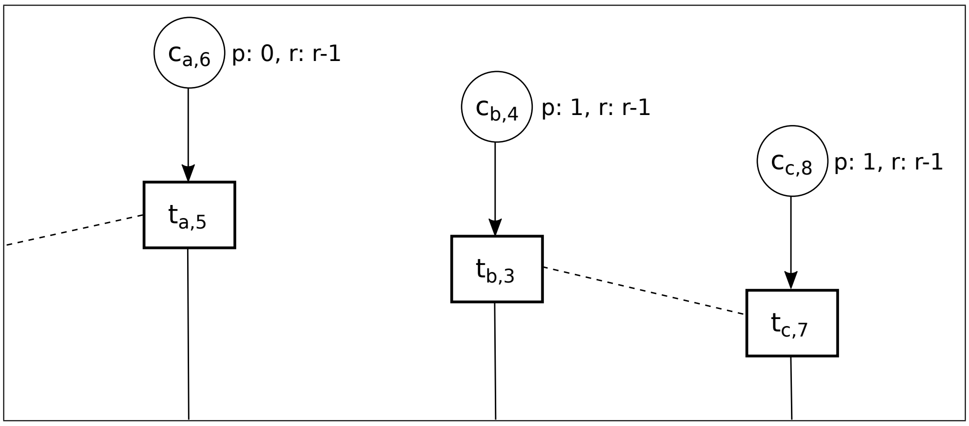

Figure A.2: Initial state

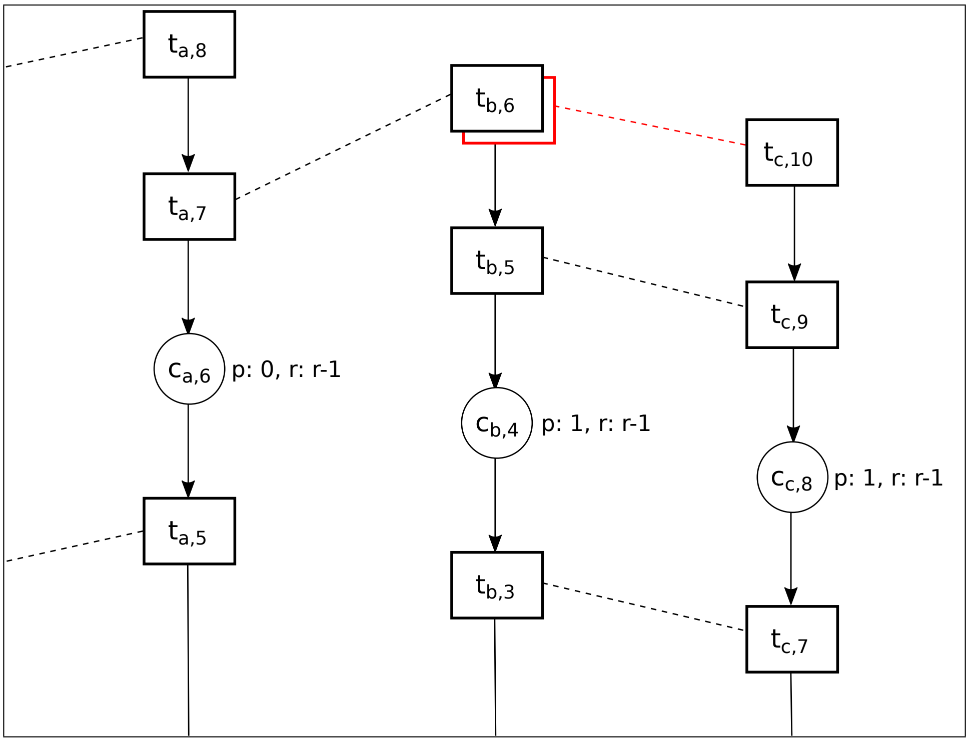

Figure A.3: Initial state

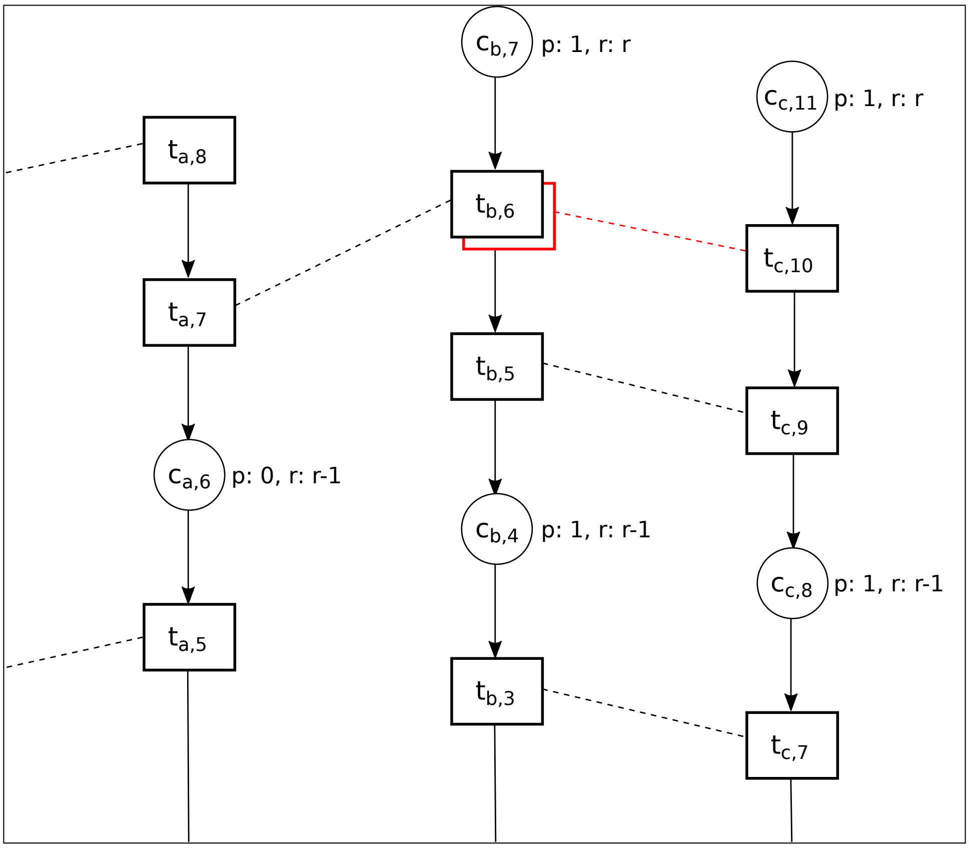

Figure A.4: Initial state