Faculty Electrical Engineering, Mathematics and Computer Science

TODO title

TODO subtitle

Kelong Cong

Supervisors:
Dr. J. Pouwelse
Dr. Unknown
Prof. Unknown

Delft, Month 2017

Abstract

THIS IS MY ABSTRACT

Preface

Please write all your preface text here. If you do so, don't forget to thank your supervisor, other committee members, your family, colleagues etc. etc.

Contents

Chapter 1

Introduction

Nothing here…

Chapter 2

Checkpoint Consensus

2.1 Preliminaries

2.1.1 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

2.1.2 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).

2.1.3 Notations

- $y = H(x)$ is a cryptographically secure hash function (random oracle), the domain x is infinite and the range is $y \in \{0, 2^{256} - 1\}$.
- Every node in the system has an identifier i and a blockchain

  $$B_i = \{b_{i,j} : j \in \{1 \dots h\}\},$$

  where h is the height of the chain. Note that $h = |B_i|$
- Each block $b_{i,j}$ has a type $t \in \{\tau, \gamma\}$, denoted by $b^t_{i,j}$. Blocks without the superscript can be of any type.

- $T(b_{i,j}) = \{\tau, \gamma\}$ is the type function, where its domain is a block and outputs the corresponding type of the block.
- A block of type τ is a *transaction block* or *TX block*. It is a six-tuple, i.e.

$$b^{\tau}_{i,j} = (H(b_{i,j-1}), h_s, h_r, s_s, s_r, m).$$

  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 denote the signature of the sender and the receiver respectively. $i = \{s, r\}$ and $j = \{h_s, h_r\}$ and we call the block a sender block or a receiver block respectively.
- Given two TX blocks $b^{\tau}_{i,j}$ and $b^{\tau}_{i',j'}$, but $i \neq i'$. If $h_s = h'_s$, $h_r = h'_r$, $m = m'$ and the signatures are valid, then we call them a *pair*.
- If there exist two TX blocks $b^{\tau}_{i,j}$ and $b^{\tau}_{i',j'}$, where s_s and s'_s is created by the same public key and $h_s = h'_s$ but $i \neq i'$, then we call this a *fork*.
- A TX block has two *inputs* blocks, to retrieve the input we define the *get-tx-input function* I^{τ}. Let the pair of $b^{\tau}_{i,j}$ be $b^{\tau}_{i',j'}$, then

$$I^{\tau}(b^{\tau}_{i,j}) = (b_{i,j-1}, b_{i',j'-1}).$$

- A block of type γ is a *checkpoint block* or *CP block*. It is a three-tuple, i.e.

$$b^{\gamma}_{i,j} = (H(b_{i,j}), H(\mathcal{C}_r), p)$$

  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.
- A CP block has one input, we define the *get-cp-input function*

$$I^{\gamma}(b^{\gamma}_{i,j}) = b_{i,j-1}.$$

- Given the input $b^{\tau}_{i,j}$, we define the *get-nearest-cp function*

$$C(b^{\tau}_{i,j}) = (b^{\gamma}_{i,a}, b^{\gamma}_{i,b})$$

  where $a = \arg\min_{k,k<j,T(b_{i,k})=\gamma}(j-k)$ and $b = \arg\min_{k,k>j,T(b_{i,k})=\gamma}(k-j)$.
- Given two γ blocks, we define *get-piece function*

$$P(b^{\gamma}_{i,a}, b^{\gamma}_{i,b}) = \{b_{i,j} : b_{i,j} \in B_i, a \leq j \leq b\}.$$

- Given two blockchains and a message, we define the *do-tx-function*

$$X_{\tau}(B_s, B_r, m) = (B'_s, B'_r)$$

  where $B'_s = \{(H(b_{s,h_s}), h_s{+}1, h_r{+}1, s_s, s_r, m)\} \cup B_s$, $B'_r = \{(H(b_{r,h_r}), h_s{+}1, h_r{+}1, s_s, s_r, m)\} \cup B_r$, $h_s = |B_s|$ and $h_r = |B_r|$.
- Given a blockchain, we define the *do-cp function*

$$X_{\gamma}(B_i, r, p) = \{(H(b_{i,h}), H(\mathcal{C}_r), p)\} \cup B_i$$

  where $h = |B_i|$, and $\mathcal{C}_r$ is the latest consensus result.
- Note that X_{τ} and X_{γ} perform a state transition.
- The result of a consensus in round r is a set of two-tuple of CP. Namely, $\mathcal{C}_r = \{(b^{\gamma}_{i,a}, b^{\gamma}_{i,b}) : a < b, \text{agreed by the promoters}\}$.

2.2 Checkpoint consensus

2.2.1 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 using $b = T_\gamma(B_i, r-1, 1)$. The current promoters (in round r) may decide to include b 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 promotors to N by sorting the promotors by their “luck value”’ and taking the first N.

2.2.2 Setup phase

We begin in the state where $\mathcal{C}_{r-1}$ has just been agreed but has not been disseminated yet. The promoters

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

Proof. □

Lemma 2. *The new set of promoter for the next consensus round is consistent with respect to all the nodes in the network.*

Lemma 3. *Promoters waiting for a some time Δ to collect transactions does not violate the asynchronous assumption.*

Corollary 1. *The setup phase satisfies the validity, correctness and termination properties.*

2.2.3 Consensus phase

We need an atomic broadcast algorithm for the consensus phase. We use a similar but simplified construction as [3], where atomic broadcast is constructed from the reliable broadcast[1] [1] 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 techniques that uses no dealers. First is to use PBFT [2], but we must change our asynchronous assumption into the weak synchrony assumption. 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.

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

2.3 Fraud detection

Here we provide two techniques for fraud detection. The first guarantees fraud detection but is not practical. The second is a randomised solution that detects fraud with a high probability.

2.3.1 Depth first search

```
$bs = \emptyset$
**procedure** REVERSE-DFS$(G, b)$
    $bs \leftarrow b \cup bs$
    **if** $t =$ genesis **then**
        **return** 1
```

[1]Reliable broadcast solves the Byzantine generals problem.

```
        end if
        for all $d \in I(b)$ do
            if VALID$(d) = 0$ then
                return 0
            end if
        end for
        return 1
    end procedure
    procedure VALID$(b, bs)$
        $(b_a^\gamma, b_b^\gamma) = C(b)$
        $p = P(b_a^\gamma, b_b^\gamma)$
        if $(b_a^\gamma, b_b^\gamma)$ is in a consensus result then
            if $p$ is a valid chain then
                if $b$ does not have a fork in $bs$ then
                    return 1
                end if
            end if
        end if
        return 0
    end procedure
```

Lemma 4. *The* `reverse-dfs` *algorithm traverses all nodes that may have an effect on block b.*

Proof. Recall that the TrustChain is a DAG, which always has a topological ordering. That is, for every edge (b', b), b' comes before b in the ordering, and nodes before b are its ancestors. If we reverse all the edges, then b' comes after b in the ordering, and nodes before b are its descendents.

`reverse-dfs` is essentially running DFS with all the edges reversed. If the DFS outputs the visited nodes in reverse postordering, then the output is in topological order and contains all of b's original ancestors. Due to the hash pointers, only the ancestors can have an effect on b. □

2.3.2 Random sampling

Bibliography

[1] 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. 5

[2] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999. 5

[3] 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. 5