Prism Voting Chains

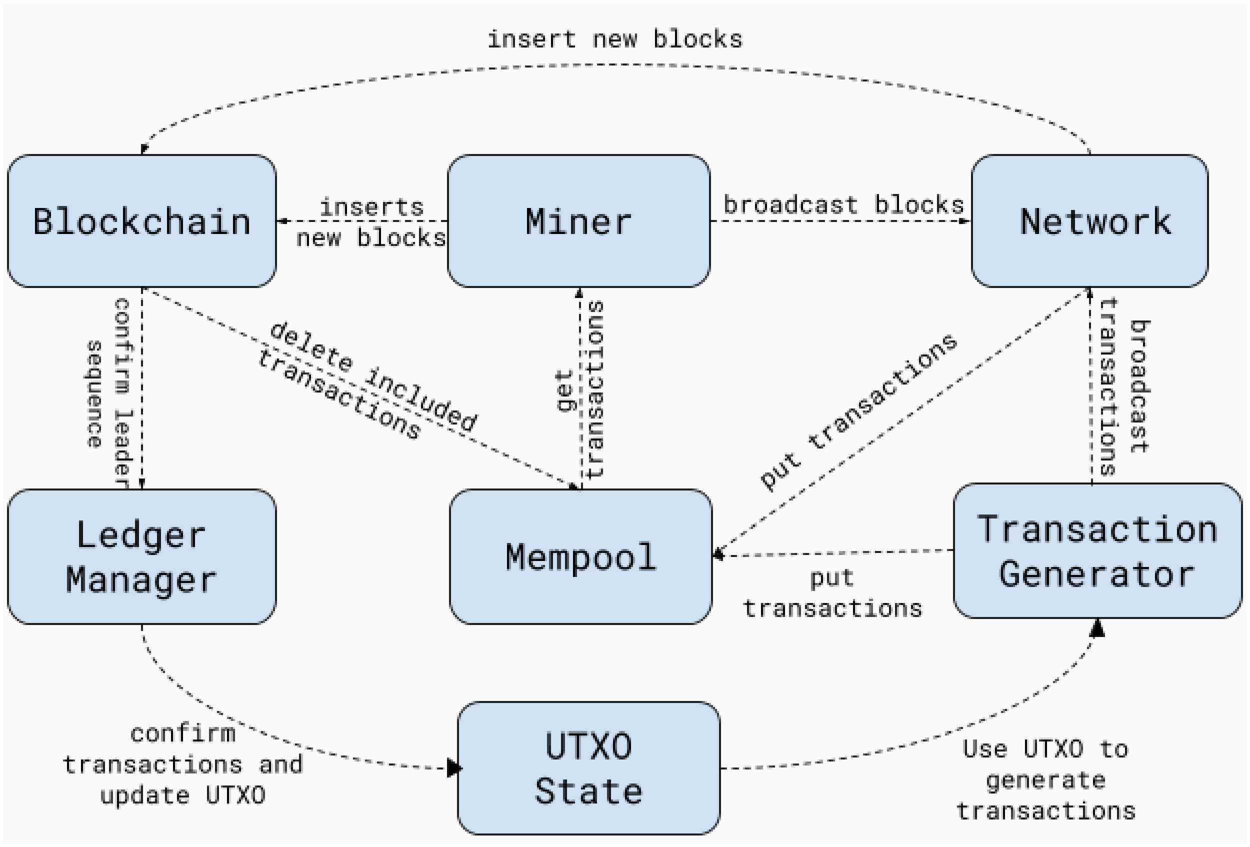
Final Report

bmjain2, yuquanw2, vinithk2, ss77

Code repository: https://gitlab.engr.illinois.edu/vinithk2/prism_voting_chains

Overview

We have implemented Prism voting chains on top of our existing bitcoin client to reduce the confirmation latency of transactions. To achieve this, we have decoupled the roles of a single block into proposer and voter blocks. Instead of having the longest chain confirmation rule in a single chain, we have multiple voter chains that vote for a proposer on each level. The proposer block with maximum votes becomes the leader for that level. We have added a new module called ledger manager which polls blockchain for proposers and their votes to decide a leader sequence and confirm transactions based on it. We have modified the miner to generate a super block (contains proposer block and voter blocks for all voter chains). Once the superblock satisfies proof of work, it is sortitioned into a proposer or one of voter chain blocks. We have added multiple data structures to blockchain which the ledger manager can use to decide leader sequence efficiently. The following block diagram lists the different components of our prism client and shows their interactions:



Difficulties

One of the most tricky part of this project was the confirmation rule used by the ledger manager. Initially, we had converted the description in Appendix A of Prism systems paper into code with the help of the open-source implementation of Prism. When we conducted experiments, we observed that Prism's confirmation latency was much higher than Bitcoin's confirmation latency. Following this, we tried to add some optimisations to our client but it didn't work either. Later we realised that the bug was in the confirmation rule. Gerui suggested that instead of using the complicated confirmation rule in Appendix A, use the following relatively simpler confirmation rule:

- Ledger manager confirms a proposer block when it has more than 50% votes that are v-deep (we use 2-deep) in their respective voter chains.

Now to compare Prism with Bitcoin client, we need to calculate the reversal probability that Prism can guarantee. Suppose the Prism implementation has 100 voter chains, then the ledger manager will confirm a proposer when it has atleast 50 votes that are 2-deep. Hence, to overturn this confirmed proposer, the adversary will have to overtake atleast 50 chains. The probability of overturning a single chain can be obtained from Nakamoto's private attack calculations (we used code from Assignment 3). For example, for beta (adversarial hash power) = 0.3 and v-deep=2-deep, the probability of successfully overtaking a single chain is p=`0.2355`. To calculate overall reversal probability ϵ that Prism guarantees, we need to calculate the probability that adversary will be able to overtake atleast 50 chains:

$$\epsilon = \binom{100}{50} p^{50}(1-p)^{50} + \binom{100}{51} p^{51}(1-p)^{49} + \binom{100}{52} p^{52}(1-p)^{48} + \ldots$$

Now again use the private attack code to figure out the k for Bitcoin at which reversal probability is less than ϵ. This means if the bitcoin client confirms transactions that are k-deep inside the longest chain, it can guarantee ϵ reversal probability.

```
for beta in hashpowers:
    for prism_k in voter_depth_k:
        # p is rev prob for a single chain
        p = success_prob[beta][prism_k]
        for m in num_voter_chains:
            # calculate epsilon overall rev prob guaranteed by prism
            epsilon = 0
            for i in range((m // 2), m + 1):
                epsilon += (comb(m, i) * pow(p, i) * pow((1-p), (m - i)))
            # find k for which bitcoin can guarantee epsilon rev prob
            bitcoin_k = None
            for k in success_prob[beta]:
                if success_prob[beta][k] <= rev_prob:
                    bitcoin_k = k
                    break
```

Another mistake that we were making was using the same overall mining rate for Prism and Bitcoin. We corrected this to ensure that the **mining rate of each chain in Prism** was equal to the mining rate of Bitcoin chain.

```rust
pub fn get_difficulty(num_voter_chains: u32) -> H256 {
   let base_difficulty: H256 =
(hex!("0000ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff")).
into();
   let difficulty = U256::from_big_endian(base_difficulty.as_ref());
   let adjusted_difficulty = difficulty * (num_voter_chains + 1).into();
   let mut buffer: [u8; 32] = [0; 32];
   adjusted_difficulty.to_big_endian(&mut buffer);
   buffer.into()
}
```

Making the above two modifications -- switching to a simpler confirmation rule and scaling the mining rate of Prism -- helped greatly and our Prism client was able to beat Bitcoin’s confirmation latency numbers.

Optimizations

1. **Proactive removal of double-spend transactions from Mempool:** Our transaction generator polls the latest UTXO state to generate transactions. Since it takes time for the ledger manager to confirm a leader, the generator can possibly generate transactions with the same UTXO entry (double spends). To remedy this, we maintain a HashSet in Mempool which records UTXO inputs used in stored transactions. Now, if an incoming transaction tries to use a UTXO saved in the HashSet it is automatically discarded.
2. **Notification system between Blockchain and Miner:** Miner needs to fetch transactions from Mempool to put into superblock. But doing this in every iteration is wasteful, because transactions are removed from Mempool only when a new proposer block is added. Hence we added a channel that notifies the miner when the blockchain inserts a new proposer block. Now, the miner needs to update its set of transactions only when it receives the notification of a new proposer.

Experiments

We have run several experiments to sanity check our client and measure it’s performance under different parameter settings. In this section, we describe our experimental setup and present results and analysis from our experiments.

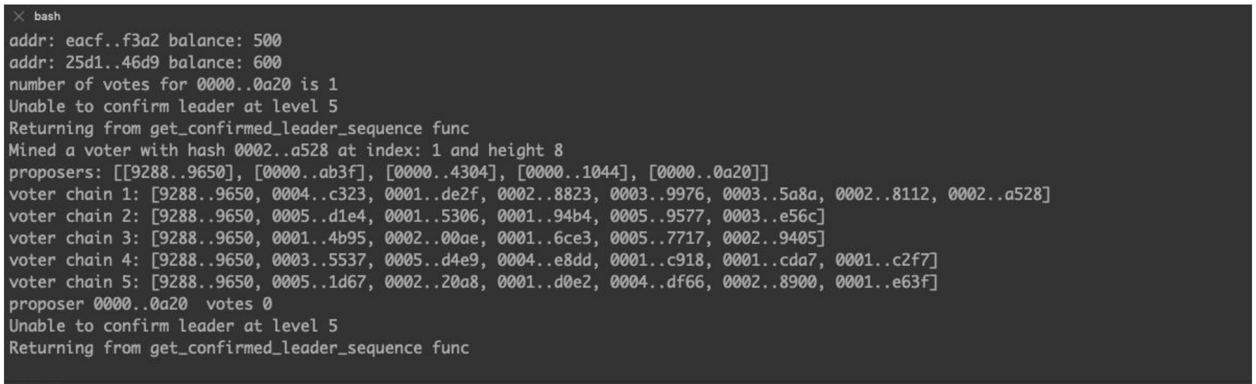
Experiment setup

We run three clients p1, p2 and p3 such that p2 is connected to p1 and p3 is connected to p2. To measure the confirmation latency, we have added logs that note transaction arrival time when it inserted in mempool and confirmation time when ledger manager processes it. Until specified otherwise, we use 2-deep votes for confirming a leader. Each block contains exactly 5 transactions. We run our experiments until at least 10 leaders are confirmed which means we have at least 50 confirmed transactions. Note: since we run three clients, we have 3 data points for each transaction. The python script measure.py calculates latency for each transaction from all three logs and outputs the overall average and standard deviation.

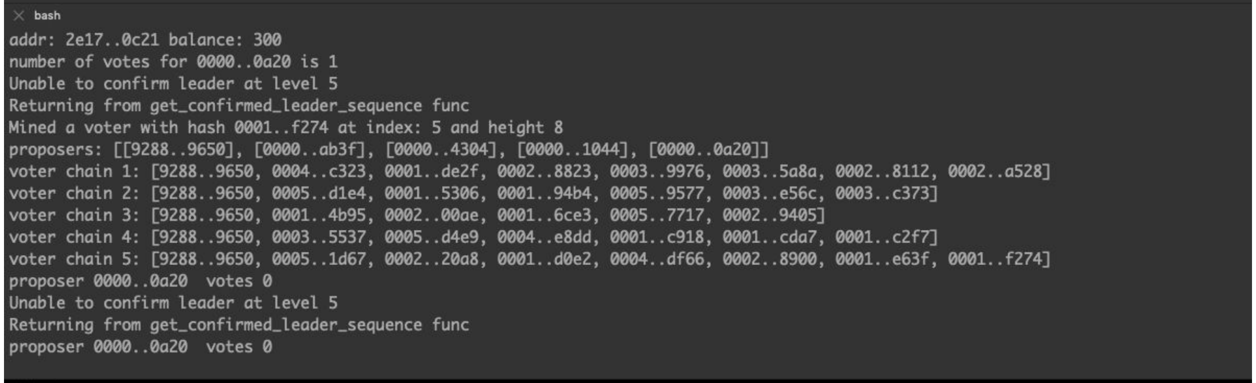
Sanity checks

All Prism clients should have same chains and a consistent UTXO state. Here we are showing results from running with 5 voter chains for better readability.

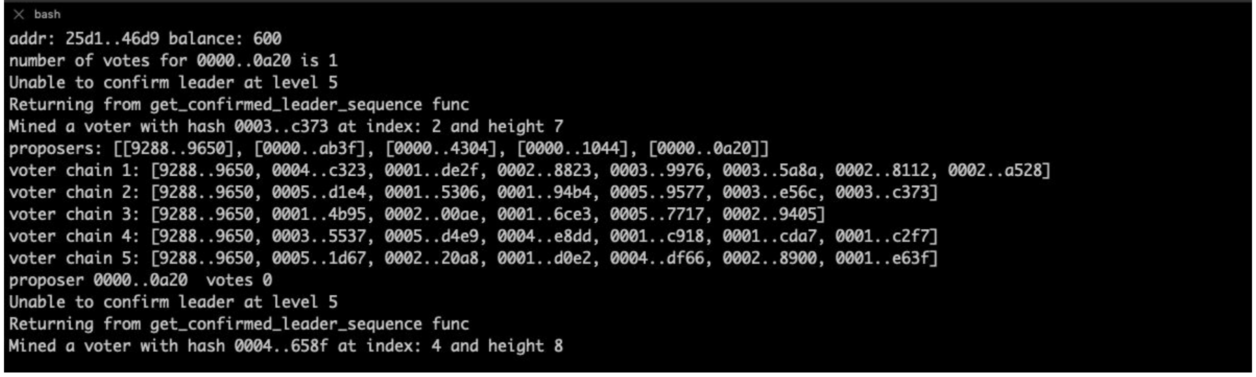
1. All clients must have the same proposer chain, leader sequence and voter chains



```
bash
addr: eacf..f3a2 balance: 500
addr: 25d1..46d9 balance: 600
number of votes for 0000..0a20 is 1
Unable to confirm leader at level 5
Returning from get_confirmed_leader_sequence func
Mined a voter with hash 0002..a528 at index: 1 and height 8
proposers: [[9288..9650], [0000..ab3f], [0000..4304], [0000..1044], [0000..0a20]]
voter chain 1: [9288..9650, 0004..c323, 0001..de2f, 0002..8823, 0003..9976, 0003..5a8a, 0002..8112, 0002..a528]
voter chain 2: [9288..9650, 0005..d1e4, 0001..5306, 0001..94b4, 0005..9577, 0003..e56c]
voter chain 3: [9288..9650, 0001..4b95, 0002..00ae, 0001..6ce3, 0005..7717, 0002..9405]
voter chain 4: [9288..9650, 0003..5537, 0005..d4e9, 0004..e8dd, 0001..c918, 0001..cda7, 0001..c2f7]
voter chain 5: [9288..9650, 0005..1d67, 0002..20a8, 0001..d0e2, 0004..df66, 0002..8900, 0001..e63f]
proposer 0000..0a20  votes 0
Unable to confirm leader at level 5
Returning from get_confirmed_leader_sequence func
```

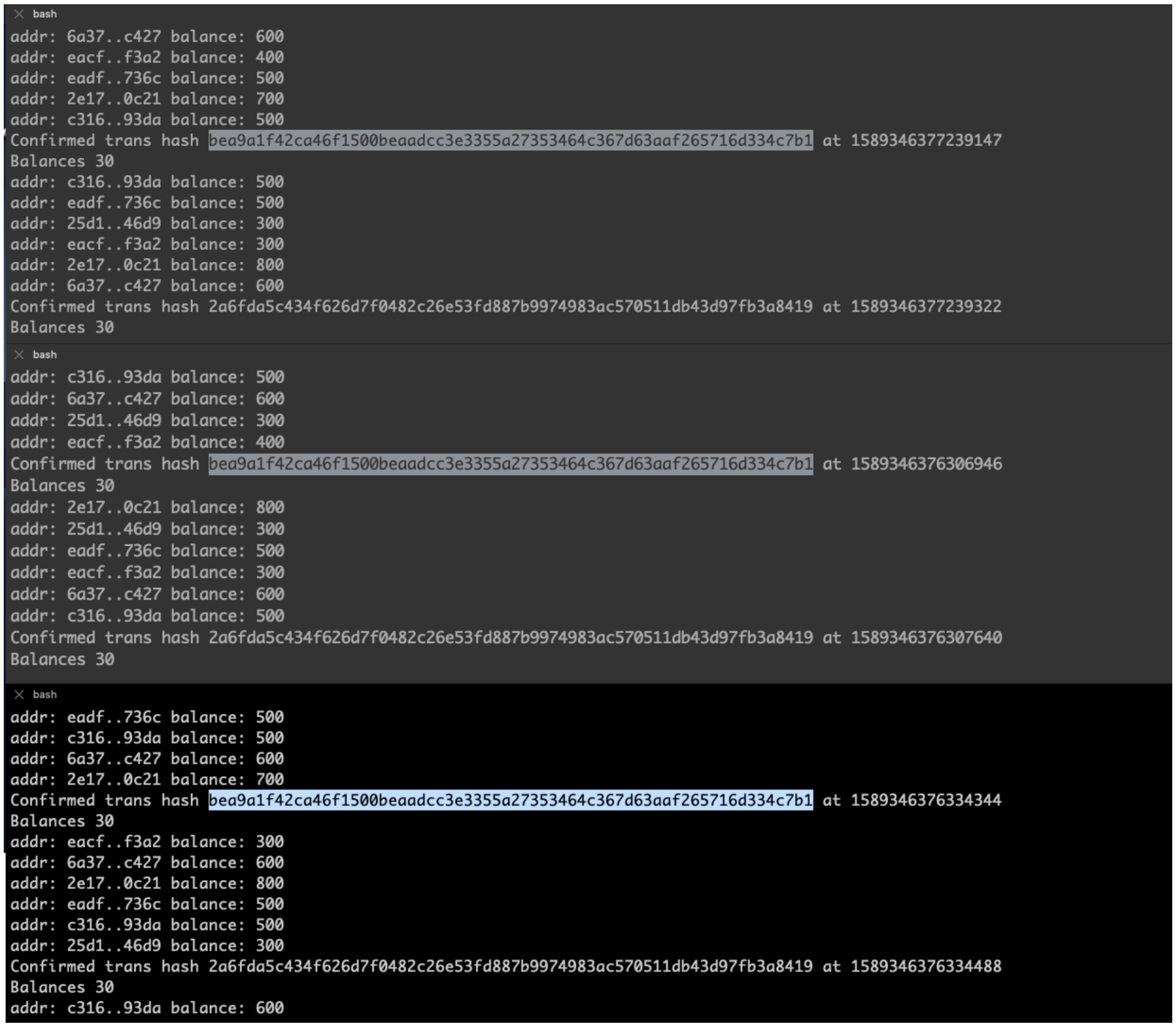


```
bash
addr: 2e17..0c21 balance: 300
number of votes for 0000..0a20 is 1
Unable to confirm leader at level 5
Returning from get_confirmed_leader_sequence func
Mined a voter with hash 0001..f274 at index: 5 and height 8
proposers: [[9288..9650], [0000..ab3f], [0000..4304], [0000..1044], [0000..0a20]]
voter chain 1: [9288..9650, 0004..c323, 0001..de2f, 0002..8823, 0003..9976, 0003..5a8a, 0002..8112, 0002..a528]
voter chain 2: [9288..9650, 0005..d1e4, 0001..5306, 0001..94b4, 0005..9577, 0003..e56c, 0003..c373]
voter chain 3: [9288..9650, 0001..4b95, 0002..00ae, 0001..6ce3, 0005..7717, 0002..9405]
voter chain 4: [9288..9650, 0003..5537, 0005..d4e9, 0004..e8dd, 0001..c918, 0001..cda7, 0001..c2f7]
voter chain 5: [9288..9650, 0005..1d67, 0002..20a8, 0001..d0e2, 0004..df66, 0002..8900, 0001..e63f, 0001..f274]
proposer 0000..0a20  votes 0
Unable to confirm leader at level 5
Returning from get_confirmed_leader_sequence func
proposer 0000..0a20  votes 0
```



```
bash
addr: 25d1..46d9 balance: 600
number of votes for 0000..0a20 is 1
Unable to confirm leader at level 5
Returning from get_confirmed_leader_sequence func
Mined a voter with hash 0003..c373 at index: 2 and height 7
proposers: [[9288..9650], [0000..ab3f], [0000..4304], [0000..1044], [0000..0a20]]
voter chain 1: [9288..9650, 0004..c323, 0001..de2f, 0002..8823, 0003..9976, 0003..5a8a, 0002..8112, 0002..a528]
voter chain 2: [9288..9650, 0005..d1e4, 0001..5306, 0001..94b4, 0005..9577, 0003..e56c, 0003..c373]
voter chain 3: [9288..9650, 0001..4b95, 0002..00ae, 0001..6ce3, 0005..7717, 0002..9405]
voter chain 4: [9288..9650, 0003..5537, 0005..d4e9, 0004..e8dd, 0001..c918, 0001..cda7, 0001..c2f7]
voter chain 5: [9288..9650, 0005..1d67, 0002..20a8, 0001..d0e2, 0004..df66, 0002..8900, 0001..e63f]
proposer 0000..0a20  votes 0
Unable to confirm leader at level 5
Returning from get_confirmed_leader_sequence func
Mined a voter with hash 0004..658f at index: 4 and height 8
```

2. UTXO state is consistent across all clients



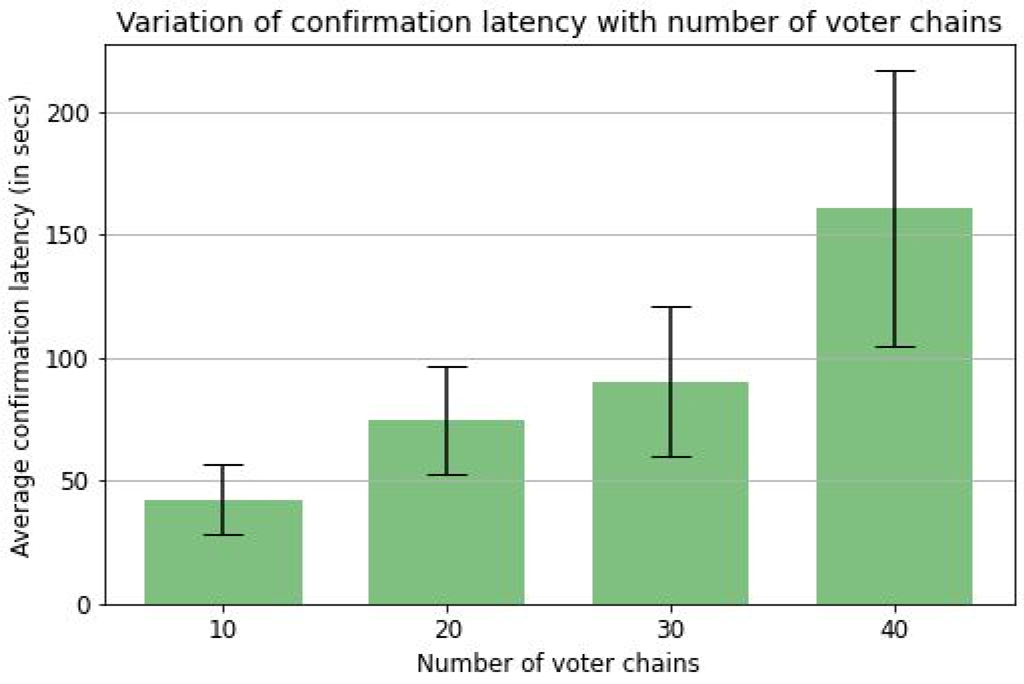
```
addr: 6a37..c427 balance: 600
addr: eacf..f3a2 balance: 400
addr: eadf..736c balance: 500
addr: 2e17..0c21 balance: 700
addr: c316..93da balance: 500
Confirmed trans hash bea9a1f42ca46f1500beaadcc3e3355a27353464c367d63aaf265716d334c7b1 at 1589346377239147
Balances 30
addr: c316..93da balance: 500
addr: eadf..736c balance: 500
addr: 25d1..46d9 balance: 300
addr: eacf..f3a2 balance: 300
addr: 2e17..0c21 balance: 800
addr: 6a37..c427 balance: 600
Confirmed trans hash 2a6fda5c434f626d7f0482c26e53fd887b9974983ac570511db43d97fb3a8419 at 1589346377239322
Balances 30
```

```
addr: c316..93da balance: 500
addr: 6a37..c427 balance: 600
addr: 25d1..46d9 balance: 300
addr: eacf..f3a2 balance: 400
Confirmed trans hash bea9a1f42ca46f1500beaadcc3e3355a27353464c367d63aaf265716d334c7b1 at 1589346376306946
Balances 30
addr: 2e17..0c21 balance: 800
addr: 25d1..46d9 balance: 300
addr: eadf..736c balance: 500
addr: eacf..f3a2 balance: 300
addr: 6a37..c427 balance: 600
addr: c316..93da balance: 500
Confirmed trans hash 2a6fda5c434f626d7f0482c26e53fd887b9974983ac570511db43d97fb3a8419 at 1589346376307640
Balances 30
```

```
addr: eadf..736c balance: 500
addr: c316..93da balance: 500
addr: 6a37..c427 balance: 600
addr: 2e17..0c21 balance: 700
Confirmed trans hash bea9a1f42ca46f1500beaadcc3e3355a27353464c367d63aaf265716d334c7b1 at 1589346376334344
Balances 30
addr: eacf..f3a2 balance: 300
addr: 6a37..c427 balance: 600
addr: 2e17..0c21 balance: 800
addr: eadf..736c balance: 500
addr: c316..93da balance: 500
addr: 25d1..46d9 balance: 300
Confirmed trans hash 2a6fda5c434f626d7f0482c26e53fd887b9974983ac570511db43d97fb3a8419 at 1589346376334488
Balances 30
addr: c316..93da balance: 600
```

Experiment 1: Confirmation latency vs number of voter chains

We study the variation in Prism's confirmation latency with the increase in the number of voter chains. We use 2-deep votes for confirming the leader. The following figure plots the average latency and standard deviation (in seconds) against the number of voter chains.

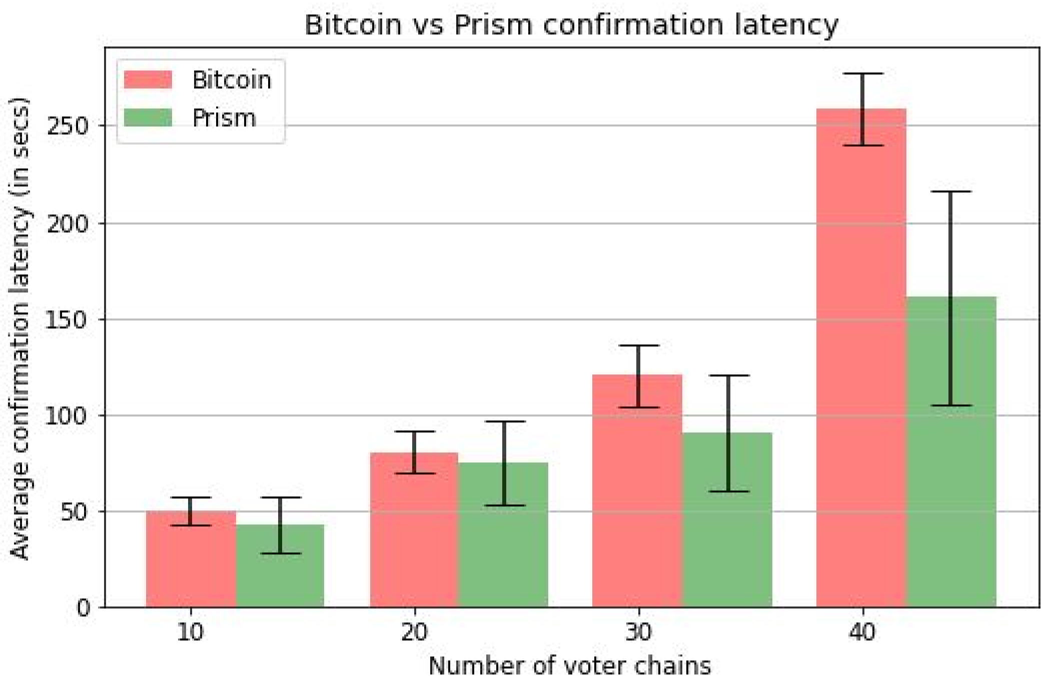


Experiment 2: Comparison between Prism and Bitcoin client

Onto the exciting part, how does our Prism client compare with Bitcoin client? For the following experiment, we vary the number of voter chains from 10 to 40 in increments of 10, and use 2-deep votes for confirming leader. Now to compare with Bitcoin we assume adversarial hash power (beta) = 0.3, calculate reversal probability for a single chain, calculate overall reversal probability that Prism guarantees (call it epsilon) and find out k for which Bitcoin can guarantee epsilon reversal probability if it confirms transactions that are k-deep inside the longest chain. Following table shows our calculations:

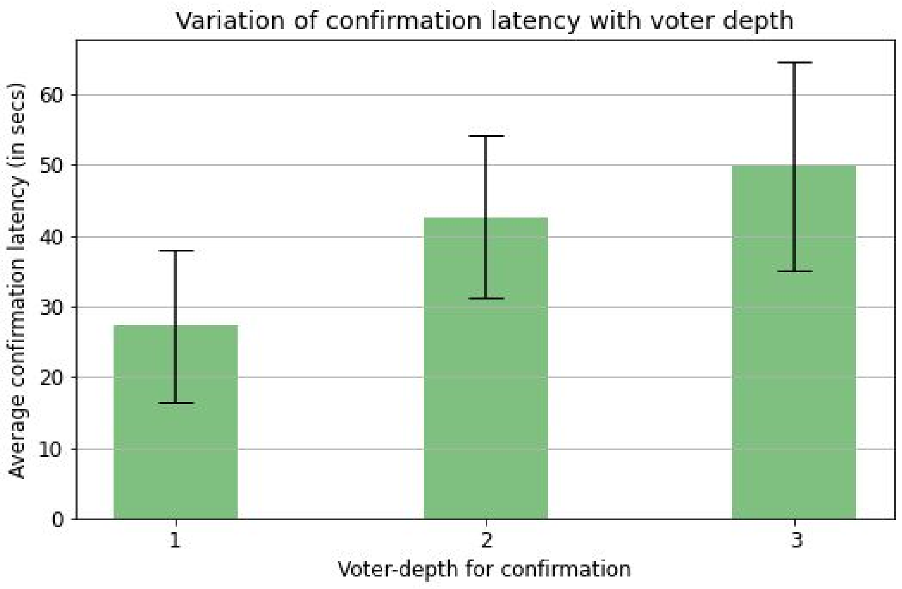
Adversarial hash power	Voter depth for confirmation	Reversal probability of a single chain	Number of voter chains	Prism accumulative reversal probability	Bitcoin k-deep value
0.3	2	0.2355	10	1.46E-02	15
0.3	2	0.2355	20	2.37E-03	24
0.3	2	0.2355	30	4.00E-04	33
0.3	2	0.2355	40	6.95E-05	45

The following bar plot compares the confirmation latency of Bitcoin vs Prism. As expected, Prism achieves a lower latency and this gaps increases as we increase the number of voter chains. We had to limit the number of voter chains used based on our machine's resources (since we use in-memory data structures to store state).



Experiment 3: Confirmation latency vs voter depth used for confirmation

We measure the effect of increasing the voter depth used for confirmation on the latency. When we increase the voter depth (v-deep), we can guarantee a much lower reversal probability and can hence can tolerate higher adversarial hash power (beta). The following figure plots the average latency and standard deviation (in seconds) against the voter depth.



Potential optimisations

Even though our Prism client can achieve lower confirmation latency, we believe there is still a lot of scope for improving performance.

1. Fine-grained locking over components of Blockchain: Modules like miner, network and ledger manager need to lock blockchain to poll some data or modify its state. But each of them needs different components, for example, ledger manager needs information related to votes whereas miner/network need to insert a new block. Implementing a fine-grained lock over these different components can boost performance.

2. Notification system between blockchain and miner: We already have a mini-notification system between blockchain and miner to notify of a new proposer. We can expand it to notify the miner of a new proposer or a new voter block. In our current implementation, we generate a new super block in every iteration. This is wasteful. The miner needs to update the super block only when there is a new block inserted in the blockchain. If a voter block is added to chain i, the super block only needs update the content of voter block corresponding to chain i. If a proposer is added, the super block needs to change proposer and all voter content (to add votes for the new proposer).

3. Caching of votes in Ledger manager: In our current implementation, the ledger manager counts the number of votes that are v-deep for a particular proposer. If the number of v-deep votes is not greater than 50%, it does not confirm the leader. In the next iteration, it again starts counting from scratch. Instead, if we can cache the incomplete number of votes and only update the votes from newly inserted voter blocks. This can lead to faster leader confirmation and hence a lower latency.