Final Project Report

Yongzhou Chen(yc28), Emerson Sie(sie2), Junhao Pan (jpan22), Ruihao Yao (ruihaoy2)

1 Algorithm and Protocol

We want to implement the Prism1.0 protocol, especially the transaction block part so as to improve the throughput of the overall system. Here we'll list the important algorithms.

1.1 Mining

In our system, every miner will mine transaction blocks and proposer blocks simultaneously. The miner achieves it by mining a super block, which is a merkle tree containing one transaction block and one proposer block. The difficulty to mine a proposer block is higher than that for a transaction block.

Algorithm 1 Mining

```
1: function miningLoop()
2:     while True do
3:         $txContent \leftarrow getTxContent(txPool)$
4:         $prpContent \leftarrow getPrpContent(unRefTxBlkPool, unRefPrpBlkPool)$
5:         $parentMT \leftarrow MerkleTree(txParent, prpParent)$
6:         $contentMT \leftarrow MerkleTree(txContent, prpContent)$
7:         $nonce \leftarrow randString()$
8:         $header \leftarrow < parentMT.root, contentMT.root, nonce >$
9:         if $Hash(header) \leq f_t$ then                         ▷ Mined a transaction block
10:            $i \leftarrow 0$ and break
11:        else if $f_t \leq Hash(header) \leq f_t + f_p$ then     ▷ Mined a proposer block
12:            $i \leftarrow 1$ and break
13:        end if
14:    end while
15: return $< header, parentMT.proof(i), contentMT.proof(i) >$
16: end function
```

When a node receive a block, it will verify the PoW and abandon it if it's invalid. If it's a transaction block, the node will append it to the *unRefTxBlkPool*, and remove the redundant transactions in the transaction pool. If it's a proposer block, the node will insert it to the proposer chain and request its parent block if it's an orphan block.

Algorithm 2 Receiving a block

```
procedure RECEIVEBLOCK(B)
    if $verify(B, type_t)$ then
        $txPool.removeTxFrom(B)$
        $unRefTxBlkPool.append(B)$
    else if $verify(B, type_p)$ then
        if $B.level == prpParent.level + 1$ then
            $prpParent \leftarrow B$
        else if $B.level > prpParent.level + 1$ then
            $RequestNetwork(B.parent)$
        end if
        $prpTree[B.level].append(B), unRefPrpBlkPool.append(B)$
        $unRefTxBlkPool.removeTxBlkRefsFrom(B)$
        $unRefPrpBlkPool.removePrpBlkRefsFrom(B)$
    end if
end procedure
```

1.2 TxConfirmation

In Prism1.0, the proposer block tree serves the same purpose as bitcoin block tree. We want to check if a specific given tx is confirmed in all possible ledger sequence so we store blocks in level order and take outer product to get all possible sequences of ledgers. Then we could confirm the tx.

```
procedure TXCONFIRMED(tx)
    $ProposerBlkSetList = []$
    for $i$ from $1$ to $ProposerTree.maxlevel$ do
        $ProposerBlkSetList.append(\{\})$
    end for
    Store proposer blocks in level order by going through each level of the tree
    $ProposerBlkSeqs = ProposerBlkSetList[1] \times ProposerBlkSetList[2] \times ... \times ProposerBlkSetList[l]$
                                                                          ▷ outer product
    for $OneProposerBlkSequence$ in $ProposerBlkSeqs$ do
        $ledger = BuildLedger(OneProposerBlkSequence)$
        if $tx$ not in $ledger$ then return False
        end if
    end for
    return True
end procedure
```

1.3 Ledger Building

When building a ledger, we need a sequence of proposer blocks, one from each block tree level. Then we perform the necessary ledger sanitization to build a ledger of txs.

Algorithm 3 Build tx ledger according to given proposer block sequence

```
procedure BuildLedger($OneProposerBlkSequence$)
    $ledger = []$
    for $ProposerBlk\ in\ OneProposerBlkSequence$ do
        $refProposerBlks = ProposerBlk.getReferencedProposerBlks()$
        $TxBlks = GetOrderedTxBlks(ProposerBlk, refProposerBlks)$
        for $TxBlk\ in\ TxBlks$ do
            $Txs = TxBlk.retrieveTxs()$
            for $Tx\ in\ Txs$ do
                if $Tx\ is\ not\ duplicate\ nor\ double\ spent$ then
                    $ledger.append(Tx)$
                end if
            end for
        end for
    end for
end procedure
```

In this algorithm, we retrieve the transactions from the longest chain of proposer block tree and this ledger is the final ledger that contains confirmed txs.

Algorithm 4 Get confirmed Txs from proposer tree longest chain

```
procedure GetOrderedConfirmedTxs()
    return $BuildLedger(ProposerBlockTree.longestchain)$
end procedure
```

2 System Diagram

The system diagram is virtually the same for the baseline client and the Prism client, since the only significant changes to the code base are self contained within each module.

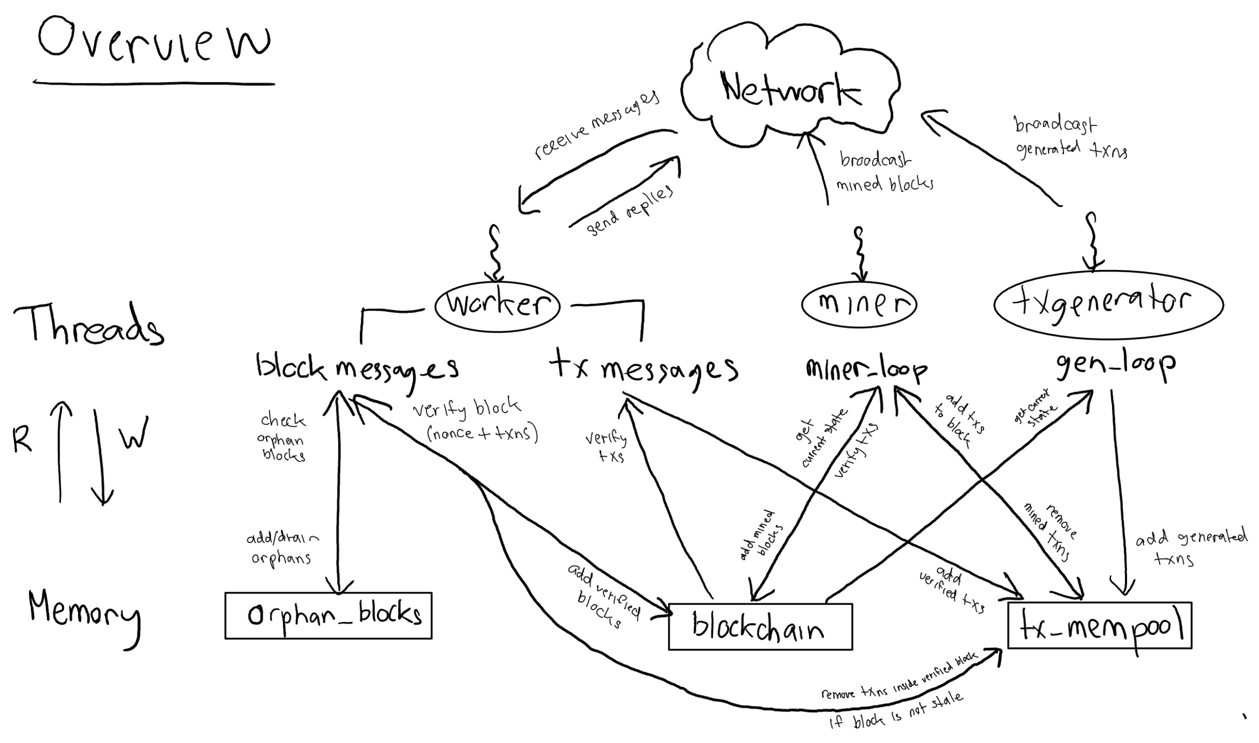
3 Experiment

3.1 Experiment Setup

We plan to run 100 processes in the rented AWS or the laptop to emulate 100 miners and 100 accounts. The difficulty to mine a block should be tuned to a constant value.
In the baseline, the mining rate of blocks is k block per second. In our system, the mining rate of proposer blocks is the same, while the mining rate of transaction blocks is pk blocks per second. Besides, every proposer block can have at most M reference links to transaction blocks.

3.2 Experiment Evaluation

To prove that the introduction of transaction blocks can improve the throughput, we'll set the same network environment, user payment behavior, and measure the throughput of transactions confirmation in the longest chain.
We'll also evaluate the influence of parameters in our system. The parameters include: block size B, mining



(a) Overall system diagram for baseline and Prism client.

rate of proposer blocks divided by that of transaction blocks p, maximum reference links number M, etc. On midterm evaluation, we'll try to finish the whole Prism1.0 blockchain client and make sure it's running bug-free. On final evaluation, we'll move everything either to AWS or a 192.168.XXX.XXX private network and plot the throughput increases against that of baseline bitcoin client.

3.3 Throughput Experiment

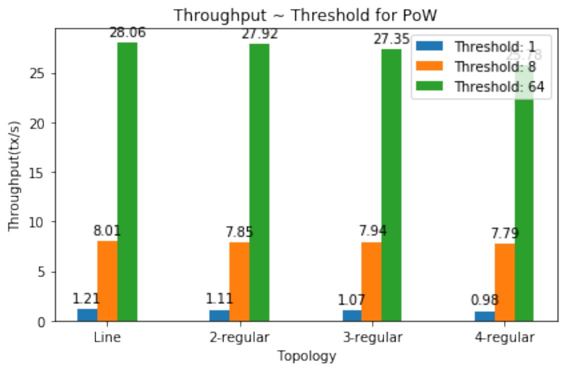
We have implemented the Prism protocol to achieve enhanced throughput. The result of this experiment has been demonstrated in the video which was presented in the Apr.30th lecture. We also carry out a more thorough set of experiments to see the advantage of prism over the baseline.

d (base/pr)	d (tx)	baseline	prism
2	**32**	17.6	662.6
2	**64**		831.4
4	**32**	47.2	652.8
4	**64**		833.8
8	**32**	89.8	645.6
8	**64**		841.8
16	**32**	163.8	657
16	**64**		835.8

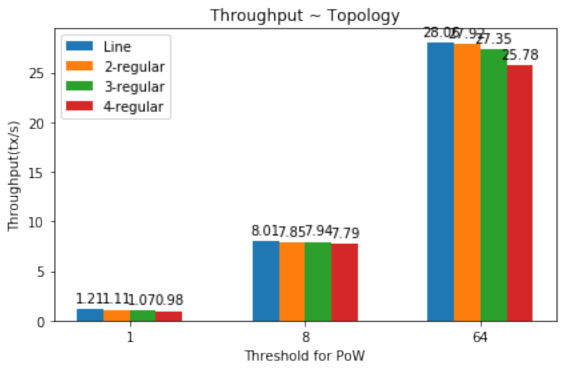
Table 1: Throughput of Baseline vs. Prism

We evaluated Prism against baseline implementation and set the difficulty of proposer blocks equal to that of blocks in baseline. The results show clear evidence that the mining throughput of Prism protocol is depended on the the difficulty of transaction blocks. We can set the threshold of difficulty much bigger (easier) for transaction blocks to obtain higher throughput.

We have also evaluated Prism's throughput according to the topology of the network. The setting is similar to what we have done in midterm report for the baseline evaluation. We have tested the line topology against 2-regular, 3-regular, and 4-regular graphs with various threshold of PoW. As shown in figure ??, the results meet our expectations.



(a) Throughput Comparisons between Different Topologies



(b) Throughput Comparisons between Different PoW Thresholds

3.4 Security Experiment

For both the Bitcoin client and Prism client security experiment, we've fixed the topology of our Bitcoin network where there are 4 honest nodes and 1 adversarial node. The security graph is plotted against β^*, a number between $[0, 1]$, and $f\Delta$, which has the unit of $f\Delta_0$ (Δ_0 equals 0.2s here). β^* means the beta threshold such that the adversarial node always succeeds given a specific value of $f\Delta$, which equals the product of mining rate and propagation delay.
Here we fix f as a constant value, which is the mining rate when the mining difficulty is $[0, 8, \underbrace{0...0}_{30}]$, and adjust the propagation delay in terms of seconds. For the propagation delay tuning, we just let the miner thread sleep for a number of seconds we want.
To simulate the β adversary hash power, we tune the mining threshold of the single adversarial node to get the expected β, assuming the relationship between mining rate and mining threshold is in positive correlation. Beginning from 0.5, we decrease β gradually to find the point when the adversary cannot always attack the system successfully, and that point is our desired β^*. If the attacker can successfully attack the system 10 times continuously with k-depth longer than 10, we view this attacker always succeeds.

For both Prism client and Bitcoin client security plot, they both exhibit a decreasing pattern. This makes sense because as $f\Delta$ gets bigger, the chance of forking gets bigger so that the private attack has a higher chance to succeed.

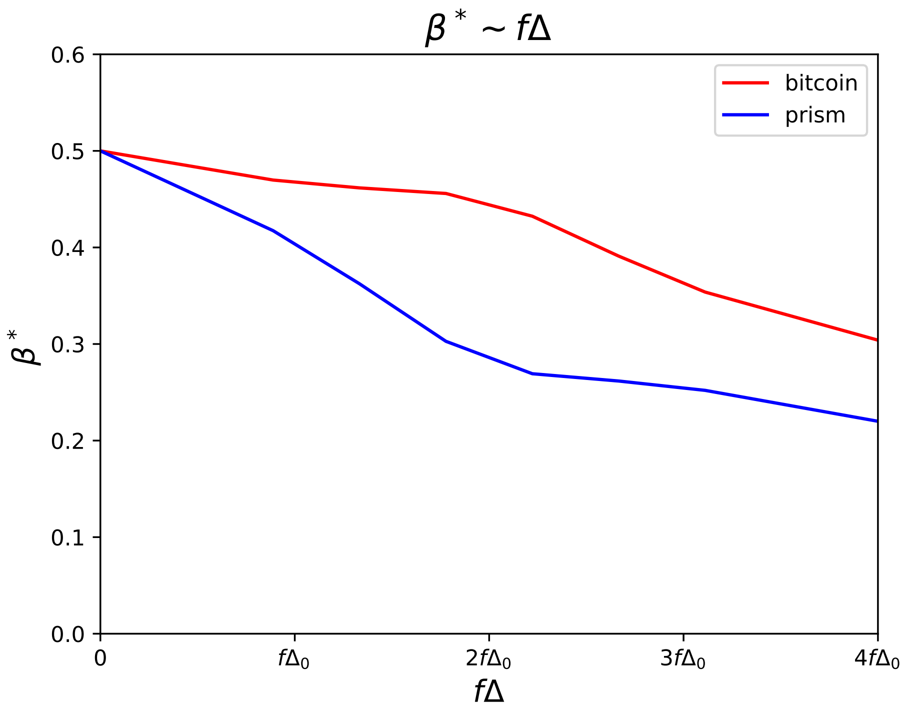


Figure 3: Throughput Comparisons between Different Topologies

However, as we can see from the plot, Bitcoin network could tolerate a higher β^* threshold in general for all possible $f\Delta$. We think the reason is that the traffic in Prism network is much more congested than that in Bitcoin network. The number of transaction blocks is several times more than that of proposer blocks, and the broadcast of such many transaction blocks impedes the propagation of proposer blocks in the air and results in a higher delay. Therefore, it's harder for the proposer blocks in Prism network to reach consensus.