École Polytechnique Fédérale de Lausanne

Improving Byzcoin

DEDIS LAB
Semester Project

Students : Louis Merlin and Hugo Roussel

Supervisor : Linus Gasser

Professor : Bryan Ford

June 5, 2020

Contents

1 Introduction

Bitcoin's original paper promised a new kind of decentralised network allowing value exchange in a peer to peer way. While this promise was successful, the original design is showing signs of age with many issues having creeped out in the last few years. To name a few : voracious energy consumption (as of today the equivalent of Switzerland energy consumption), high transactions fees and long confirmation times. In 2016 an alternative protocol called Byzcoin was proposed with the goal of improving both security and performance of the original paper. Shortly after an implementation was developed by the DEDIS lab at EPFL. The goal of this semester project was to study, measure and improve the state of the current implementation.

2 The Byzcoin protocol

Before dwelling into our work, it is good to remind to the reader some aspects of the Byzcoin [1] protocol.

2.1 PBFT Consensus

The original Byzcoin paper starts by imagining the bitcoin protocol using a different consensus protocol : the Practical Byzantine Fault Tolerant protocol (PBFT for short) [2]. Using such a protocol provides many different advantages compared to the original Proof-of-Work consensus (PoW).

PBFT Advantages

The first advantage is energy efficiency. PBFT can achieve distributed consensus without having to carry out the repeated hashing of a full block before appending it to the chain. The second is deterministic transactions : the transactions of the clients do not require multiple confirmations like the 6 block wait for Bitcoin (around 60 minutes) once the transaction have been finalized and agreed upon.

PBFT Drawbacks

One of the drawbacks of PBFT is that it does not supports open networks and that the communication complexity does not scale out. Indeed we need $O(n^2)$ messages for reaching consensus since each member of the consensus group authenticates to the other members using Message Authentication Codes.

2.2 Scaling

To circumvent these issues that prevents the system to scale out to a large system of nodes, the paper suggest multiple solutions.

Replacing MACs by Digital Signatures

Digital signatures reduces the communication complexity to $O(n)$ from $O(n^2)$ as the leader can collect and distribute the digital signatures to other participants.

Collective Signing

Eventhought the protocol is already more scalable, the original paper presents another improvement. Indeed it will be costly to the leader to distribute 1000 digital signatures and wait for everyone to verify them. To fix this the protocol builds upon the CoSi protocol in a two round fashion to recreate the two phases PBFT, prepare and commit. This allows for for each participant to receive only a $O(1)$-size rather than $O(n)$-size message and to expend only $O(1)$ rather than $O(n)$ computation effort by verifying a single collective signature instead of n individual ones. This reduces both computation and bandwidth costs associated with each round of consensus.

Two chains

Finally a last improvment comes from the realisation that the Proof of Work from the Nakamoto consensus serves two purposes : transaction verification and leader election. Here those two uses are decoupled through the creation of two types of blocks, microblocks which contain transactions and keyblocks that represent leader election. This idea is taken from the Bitcoin-NG paper [3].

The use of such a design allows for the microblocks to be irreversibly committed whatever the behavior of the leader might be. This is an improvement compared to Bitcoin-NG protocol where a malicious leader could rewrite history in between blocks [3].

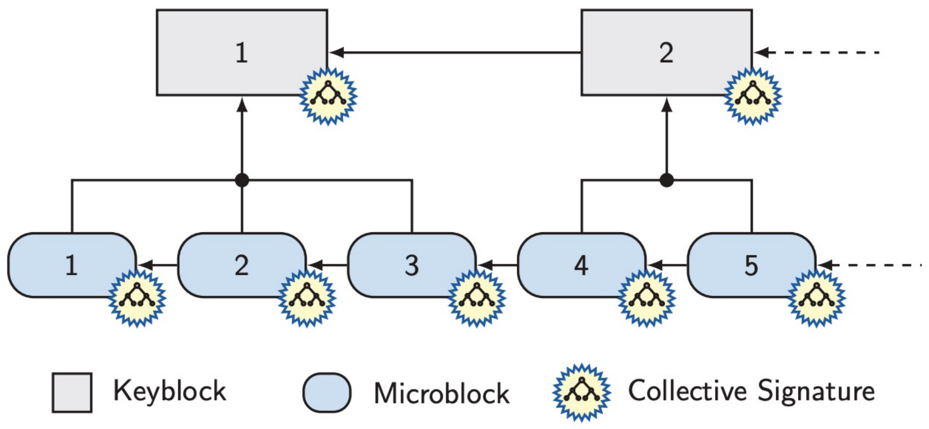


Figure 1: Figure from the original Byzcoin paper that showcases the two paralled chains storing information abouth the transactions (microblocks) and about the leaders (keyblocks)

2.3 Use of the Byzcoin protocol

Smart-Contracts

The Byzcoin data is organized in Smart Contracts, where a smart contract can be seen as a class and an instance of this class. In ByzCoin, one can Spawn a new instance of a contract, Invoke a command (or method) on an existing instance, or Delete an instance. Spawn, Invoke, and Delete represent all the possible actions a user can do to update the ledger. Every request to update the ledger is a transaction made up of one or more instructions. The transaction is then sent to one of the nodes. All instructions in the transaction must be approved by a quorum of the nodes, or else the entire transaction is refused. Current smart-contracts implemented on the Byzcoin protocol can serve different functions :

- A simple coin contract for value transfer
- Proof of Personhood contracts [4]
- Emulation the Ethereum Virtual Machine to support solidity contracts [5, 6]

3 Measuring the current implementation

After having studied and understood the current protocol, we start on trying to improve the current Golang implementation [7]. Luckily for us, a simulation of the protocol was already defined in the implementation which let us monitor the system by altering multiple parameters such as the block time, the number of hosts, servers, the number of transactions and the number of instructions per transactions. This simulation creates a ”coin” contract, registers multiple account and then transfers coins between those accounts.

3.1 Improving the simulation

We improved the current simulation by making new functions that makes it direct to :

- Create accounts
- Mint coins
- Transfer coins

We then created a new variable in the simulation : ”accounts” that creates the given number of accounts and then transfers coins in a non deterministic way to simulate a real world scenario.
Once we had a satisfactory simulation we could start on measuring the ground truth of the protocol, which will serves us as a comparative point for the end of the project.

Improvements Areas

Byzcoin is a complex machine with many moving parts, as such we had multiples areas to check where improvements could be made.

- Database storage. Byzcoin uses a trie internally to store data from the ledger. Therefore we should monitor the trie usage (creation, insertion, update of values)
- Network usage : which heavy structures are sent around the network, are they all necessary?
- Is the state change application made more than once a block?

3.2 Throughput versus Latency

We define here the difference between throughput and latency in the case of a distributed system.
Throughput is the number of actions executed produced per unit of time. For a transaction based distributed system this would correspond to the number of transactions per second. Latency is the time to perform some action or finish a result. In our case this would be the time a transaction would take to be sent through the network and appended to a block.
Latency is measured in units of time, throughput in number of actions per unit of time.

3.3 Measuring the throughput and the latency

To measure the throughput we can use the simulation and compute the overall time. If the overall time is lower with the same number of transactions we know that we will have increased the throughput.
To measure the latency we can create a simulation with a single transaction and the same parameters as above. If the time to process a single transaction is lower, we will know that we have decreased the latency.

3.4 Measuring the trie usage

Golang Benchmarking

The Golang benchmarking is an integrated tool of the native go test framework. Our goal was to examine a specific part of the stack : the use of the trie structure as the main data storage system. You will find the final trie benchmarks in Figure 2. As we can observe the benchmarks are split between Memory and Disk; indeed, some operations are stored in the ram, and others are put on disk. The goal of this benchmarking was to both measure the current performance, see if there was a point where the system would break under the load and see potential improvements.

Overall the trie structure was quite resilient under pressure. What's more, we felt (rightly, as you can see in the profiling graph), that it was already quite efficient. We moved on to see if there were other improvements possible in other parts of the stack.

3.5 Monitoring the current simulation

We then decided to monitor more closely the actual protocol code by using the onet monitor tool. [8]

BenchmarkOverwriteOne/Memory-8	19731 ns/op
BenchmarkOverwriteOne/Disk-8	4788971 ns/op
BenchmarkOverwriteMany/Memory-8	2446634 ns/op
BenchmarkOverwriteMany/Disk-8	383921557 ns/op
BenchmarkCopy/1-8	1611 ns/op
BenchmarkCopy/50-8	22628 ns/op
BenchmarkCopy/500-8	100455 ns/op
BenchmarkCopy/2000-8	102197 ns/op
BenchmarkBatchSetDel/1000/Memory-8	65863324 ns/op
BenchmarkBatchSetDel/1000/Disk-8	89311748 ns/op
BenchmarkBatchSetDel/10000/Memory-8	767646219 ns/op
BenchmarkBatchSetDel/10000/Disk-8	816768896 ns/op
BenchmarkBatchSetDel/100000/Memory-8	6440816948 ns/op
BenchmarkBatchSetDel/100000/Disk-8	6770227394 ns/op

Figure 2: Benchmark of trie operations

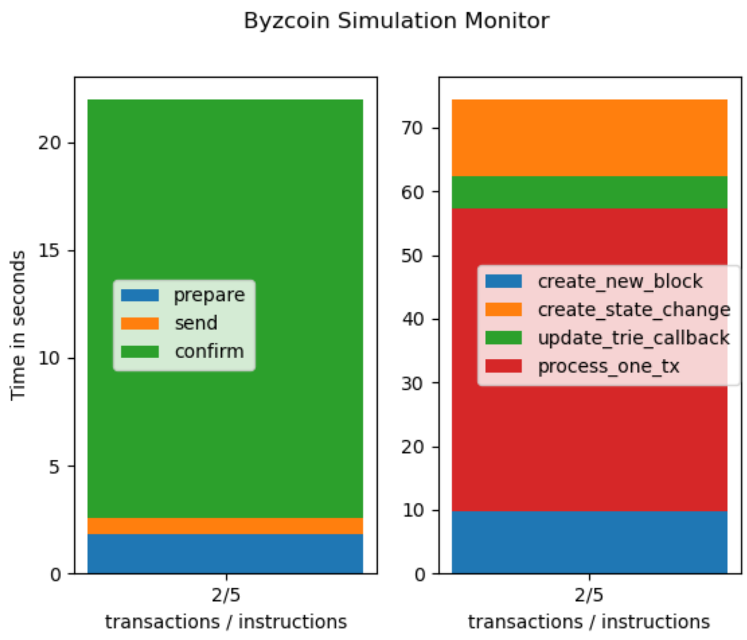


Figure 3: High level overview of function wall time and finer grained observations

As we can see in this graph, on the left part we have the number of seconds taken by the overall simulation, split between three parts : the "preparation" phase, the "send" phase and the "confirm" phase. As we can see, it is the "confirm" phase that takes up the biggest chunk of our time. The right part is a zoomed-in view of the "confirm" phase of the left part. It takes a greater total number of seconds to complete because it was computed in a multiprocessor fashion. We can clearly see from this graph that we need to continue to zoom in on the "process one tx" part (which corresponds to the call to the ProcessOneTx method), which is what we did.

For the second graph we decided to dig in more on why the processing of one transaction was so long. We suspected that the time was mostly spent on cryptographic operations, which we considered out of the scope of this project since the dedicated libraries are already optimized. To verify our intuition we recursively graphed each function. We obtain the following graphs, and indeed we can see that on the last graph that most of the time is spent on checking the configuration of the user and checking the cryptographic signatures.

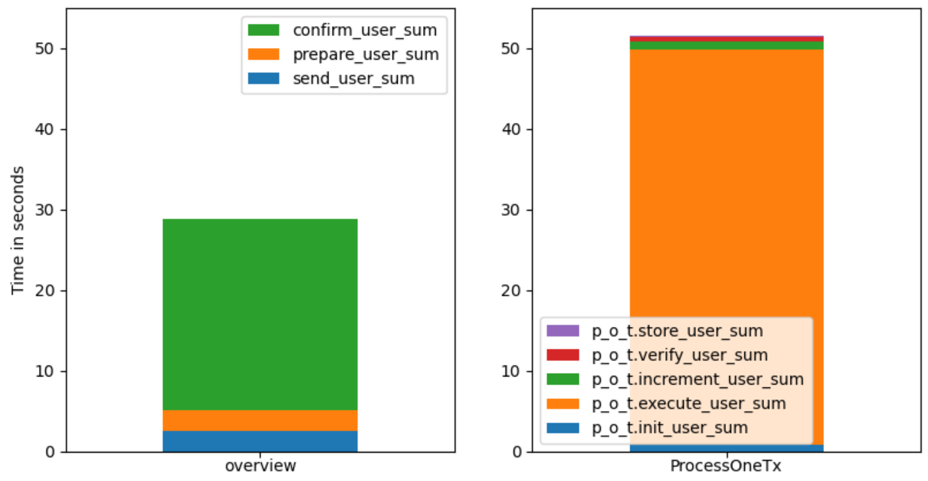


Figure 4: Recursively graphing costly functions

We can analyse the graphs from left to right :

- This is the exact same left part of the first graph. We see that we need to zoom into "confirm"
- This is the zoomed in part of ProcessOneTx. We see that "execute"

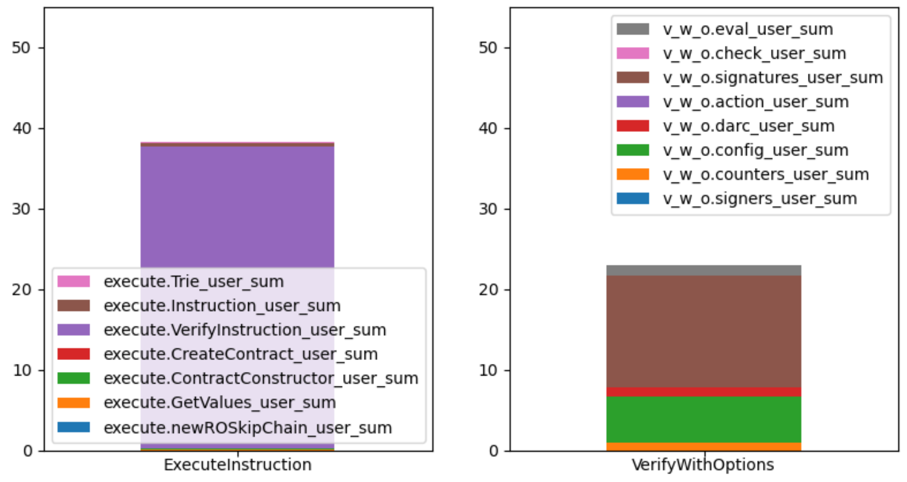


Figure 5: Recursively graphing costly functions II

(which is the ExecuteInstruction method) takes up most of the time.

- We now zoom into ExecuteInstruction. Here too, most of the time is spent doing one thing : VerifyInstruction. If we go see what that method does, we find out that it calls another method, called Verify-WithOptions.

- Inside VerifyWithOptions we find that the "signatures" part (cryptographic signature computation" takes up more than half of the compute time, and "config" takes a quarter of the time. This "config" part is actually the code that loads up the configuration from the trie data structure.

3.6 Profiling the simulation

To finally confirm our observations we used a go profiler to understand better the flow of execution and memory allocation.

Pprof

We used the pprof tool to generate a graph of functions calls of the simulation. Pprof [9] is a tool created by Google for visualization and analysis of profiling data. The tools reads profiling samples and creates visual reports that helps analyze the data. By running this tool on the simulation we obtain the following graph : 1. The graph shows the different function calls of the program during the simulation. Each box is a function and each arrow represents a call from this function. A dashed line represents that some function calls were skipped. We can see quickly what we saw earlier, that the ProcessOneTx is very costly and leads to functions which are equally costly : cryptographic functions related to the Edwards25519 Elliptic Curve (also used in the Monero key pair generation [10]).

This confirmed our intuition on two parts : first the time taken by the trie is not a significant one and that most of the CPU time is spent on cryptographic operations.
We then decided to try a different approach and actually modify the protocol instead of improving the current code of the implementation.

4 Improving Byzcoin

After the observations from the measurement parts, we had to chose the best way to improve the Byzcoin protocol. We decided to work on the latency part and particularly change the way transactions are sent through the network.

4.1 The Collect Transaction Protocol

In the Byzcoin protocol transactions are not transferred between nodes using a gossiping protocol as it is the case in the Ethereum or Bitcoin blockchain. Here the sharing is made using communication tree through the use of the collect transaction protocol. The collect transaction protocol works as follows : the leader of the current round periodically sends empty transaction "requests" and the children will respond with the transaction in their transaction pool.

This scheme is useful to have a simple way to get the transactions at the right place, and insures that only the leader gets the transactions that he needs before creating a new block. However the drawback is that it generates a lot of network traffic only for those empty request transactions. Under the advice of our supervisor we started designing a new type of protocol for the transaction propagation.

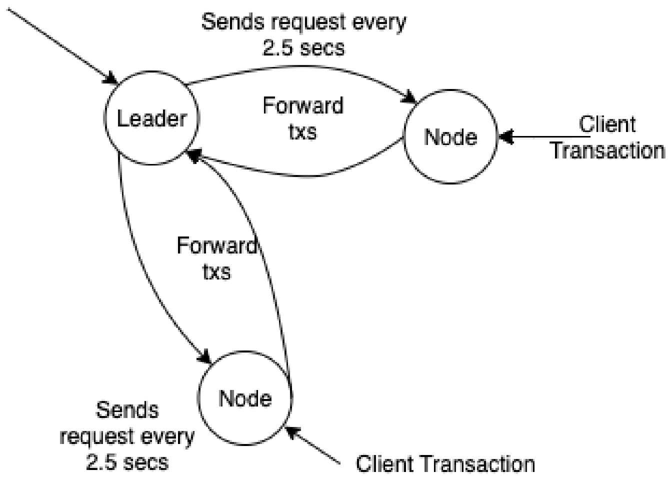


Figure 6: The way the transaction are propagated between the nodes using the collect transaction protocol

4.2 The Rollup Transaction Protocol

We came up with a different approach to replace the collect transaction protocol. The new protocol would be very simple, getting away with the request transactions and working as follows :

- If you are the current leader, you have nothing to do and can process the transaction in a new block directly
- If a you are not the leader, forward the transaction to your parent in the communication tree which will then transmit its own parent until reaching the leader

Such a simple scheme hides many caveats that we will cover after.

4.3 Implementation timeline

We detail here the implementation timeline as well as the challenges encountered when creating this new protocol.
We first had to understand the way the collect transaction protocol was implemented and integrated in the rest of the stack.

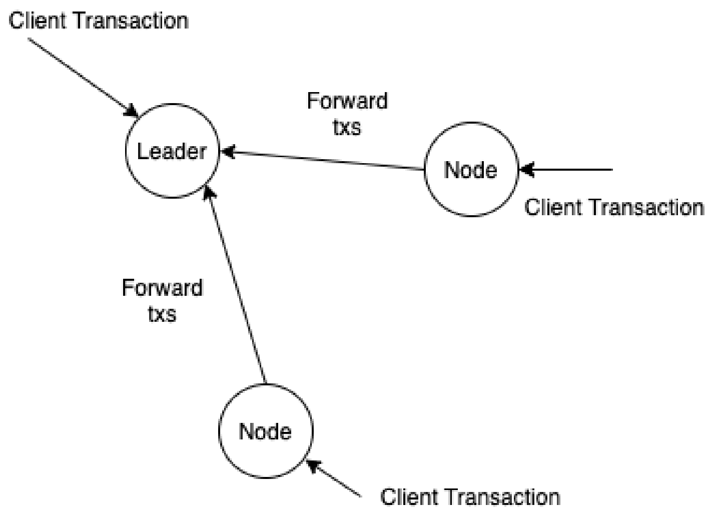


Figure 7: The way the transaction are propagated between the nodes using the roll up transaction protocol

Then we used the cothority template repository to create a toy protocol which was added alongside to the old one. After that we implemented the actual protocol. We then removed all the references from the old one, removing for example the structures corresponding to the old request packets and creating new types of replies. After a while we realized we had registered the protocol in a non functional way and had to use a different way, like the protocols of the skipchain.
The integration was tricky since we had not only to change the protocol but also change the actual integration to the system. Indeed we had to modify functions related to the service part of the byzcoin protocol which serves the client API. For example registering the object `txPipeline` used by the leader to propose new block in the Byzcoin service file instead of doing it in the protocol directly. We were finally able to make the simulation run again and we had to make sure that the tests would all pass to be able to make a pull request on the main repository at the end of the project.
The first tests we had to fix were related to transactions not being sent correctly, after debugging for a while we found that there was a race condition and we were able to fix it using the `---race` option of the go test suite.
The other tests that then started failing were related to the update of the

version those were fixed quickly.
We then encountered issues with the tests of the transaction pipeline `txPipeline` and we realized that those tests were broken by the new implementation. We had to skip them and we did not rewrite them since it would have been too time consuming at this point of the project.
Eventually we last tests failing were related to the view change requests that were triggered uselessly by the children. In fact this happened because the `requests` packets used in fact two purposes : one for polling and the other one as a "heart beats" monitor to insure that the leader of the round was not down. Since those packets were removed, the children always thought that the leader was failing and triggered view change requests. The fix was to replace this function by instead using the response from the request to add a transaction which usually contains an acknowledgment that the transaction was indeed included as well as the hash of the block etc.

Implementing this new protocol was very challenging and we had to modify a lot of the parts of the Byzcoin protocol while insuring that the development tests would still run correctly.
We had a lot of trouble understanding the code, and it was difficult to apprehend on how to register correctly a protocol in the system, since there was two ways to do it and the one appropriate was not the same as the one used by the collect transaction protocol. Once we had something that made the old simulation run we had to insure every test from the local test suit would pass, as well as making the online Jarvis tool happy with both the formatting and the linting of the code to keep the standard at the level of the implementation.
All in all we spent a considerable time working on the understanding of the old implementation, the creation of the protocol, its inclusion in the implementation and the debugging afterwards. Hopefully with the help of our supervisor as well as a code review, we were able to create a pull request that could be added on the current repository.

Challenges and risks

The new protocol comes with some drawback as discussed earlier. For example some future work should think about the following points :

- Transaction flooding to ddos the current leader
- Edge cases that might not be covered by tests
- Optimality of the protocol

- Measuring using a cluster of nodes/not in localhost

Results

It was now time to verify if our work had any influence on the latency of the protocol. We created this graph which measures the difference of the latency of transactions locally on both of our computers.

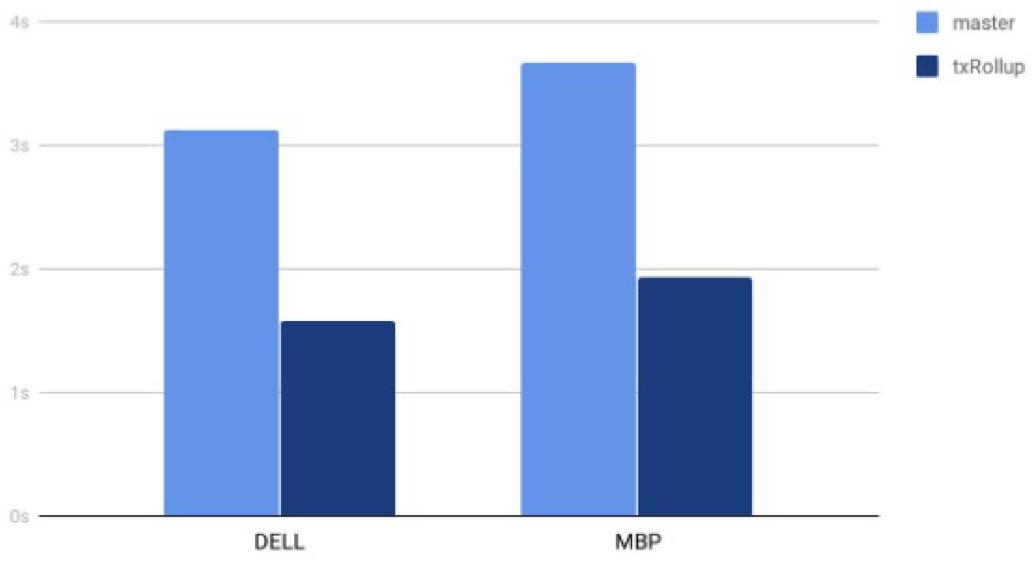


Figure 8: Comparing the latency difference of the two protocols

We generated this graph using a simple simulation of the protocol, that is creating one block after a child node submits a transaction.

We can clearly see from the graph that the new protocol actually decreased the latency with an improvement of around 1.5. It would now be interesting to perform such an analysis on a large cluster with many more nodes. Also it would be interesting to measure the total number of packets used for a sample run and compare them with the new protocol to see if there was a decrease in the network usage. We we rerunned pprof to check for possible changes in the data and there seem to be some changes but nothing substantial.

5 Conclusion

To conclude we first had to learn about Byzcoin and its protocol. We then had to measure the current implementation and get used to the codebase which revealed to be a significant challenge. We then designed, implemented and debugged a new protocol to be added to the current code and measured its performance compared to the old one. We were happy to achieve a decrease in the latency of Byzcoin by the use of our new Rollup Transaction protocol. Despite the context of this semester which made our work harder, we would have liked spending more time on writing code and we were often blocked by bugs which slowed us significantly all along the project. Still we learned a lot both in the field of distributed systems but also in development best practices, on how to apprehend a large codebase and on how to maintain its quality over time.

Bibliography

[1] Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing,
`https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_kokoris`
Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford, 2016.

[2] Practical Byzantine Fault Tolerance,
`http://pmg.csail.mit.edu/papers/osdi99.pdf` Miguel Castro and Barbara Liskov, 2016

[3] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse *Bitcoin-NG: A Scalable Blockchain Protocol*
`https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-eyal.pdf`

[4] Maria Borge, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Bryan Ford
Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies
`https://ieeexplore.ieee.org/document/7966966`

[5] *Byzcoin Ethereum Virtual Machine*
`https://github.com/dedis/cothority/tree/master/bevm`

[6] *Ethereum Whitepaper*
`https://ethereum.org/whitepaper/`

[7] *Cothority implementation*
`https://github.com/dedis/cothority`

[8] *Onet Monitoring tool*
`https://github.com/dedis/onet/blob/master/simul/monitor/monitor.go`

[9] *Pprof*
`https://github.com/google/pprof`

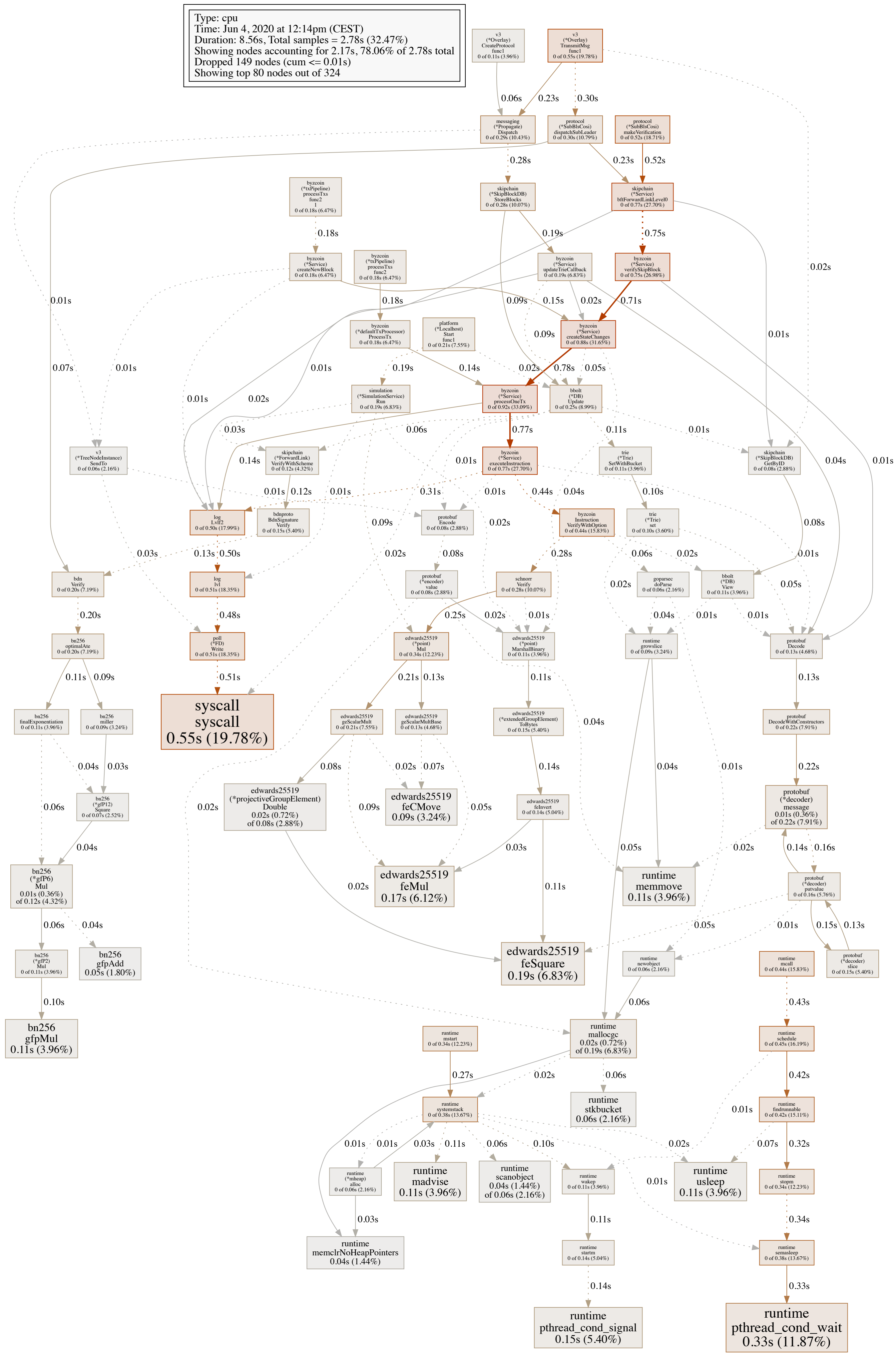
[10] *Monero Whitepaper*
`https://cryptonote.org/whitepaper.pdf`

1 Appendix: Profiling Appendix

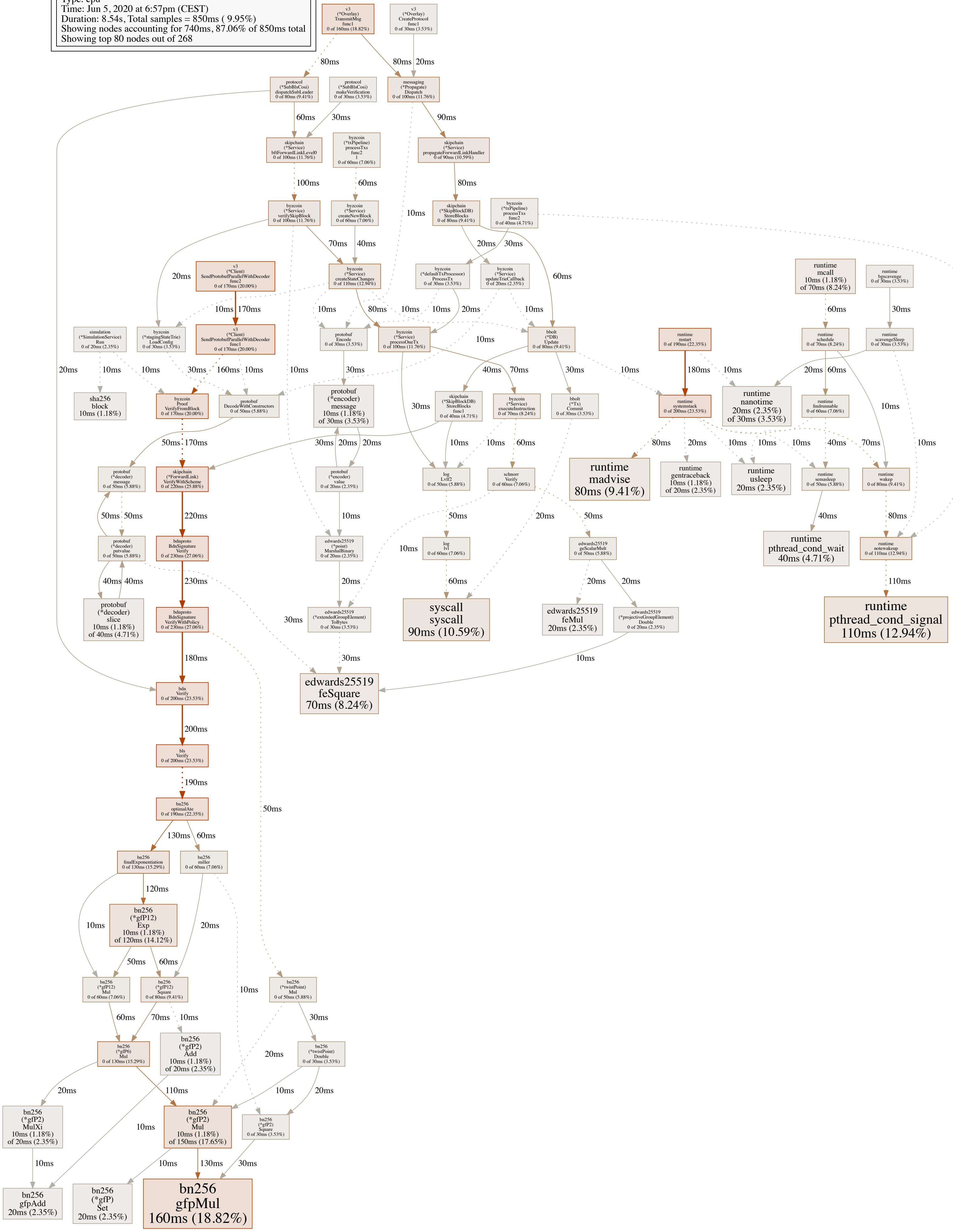
The first graph profiles the simulation with the same parameters as in histograms, i.e : 200 transactions, 5 instructions by transaction, 5 hosts and a one second block interval.

The second graph profiles the simulation with the following parameters : 1 transaction, 1 instruction by transaction, 5 hosts and a one second block interval.

We put them in full page otherwise they are too small to read.


Type: cpu
Time: Jun 4, 2020 at 12:14pm (CEST)
Duration: 8.56s, Total samples = 2.78s (32.47%)
Showing nodes accounting for 2.17s, 78.06% of 2.78s total
Dropped 149 nodes (cum <= 0.01s)
Showing top 80 nodes out of 324
v3 (*Overlay) CreateProtocol func1 0 of 0.11s (3.96%)
v3 (*Overlay) TransmitMsg func1 0 of 0.55s (19.78%)
messaging (*Propagate) Dispatch 0 of 0.29s (10.43%)
protocol (*SubBlsCosi) dispatchSubLeader 0 of 0.30s (10.79%)
protocol (*SubBlsCosi) makeVerification 0 of 0.52s (18.71%)
skipchain (*SkipBlockDB) StoreBlocks 0 of 0.28s (10.07%)
skipchain (*Service) bftForwardLinkLevel0 0 of 0.77s (27.70%)
byzcoin (*txPipeline) processTxs func2 1 0 of 0.18s (6.47%)
byzcoin (*Service) createNewBlock 0 of 0.18s (6.47%)
byzcoin (*txPipeline) processTxs func2 0 of 0.18s (6.47%)
byzcoin (*Service) updateTrieCallback 0 of 0.19s (6.83%)
byzcoin (*Service) verifySkipBlock 0 of 0.75s (26.98%)
byzcoin (*defaultTxProcessor) ProcessTx 0 of 0.18s (6.47%)
platform (*Localhost) Start func1 0 of 0.21s (7.55%)
byzcoin (*Service) createStateChanges 0 of 0.88s (31.65%)
simulation (*SimulationService) Run 0 of 0.19s (6.83%)
byzcoin (*Service) processOneTx 0 of 0.92s (33.09%)
bbolt (*DB) Update 0 of 0.25s (8.99%)
v3 (*TreeNodeInstance) SendTo 0 of 0.06s (2.16%)
skipchain (*ForwardLink) VerifyWithScheme 0 of 0.12s (4.32%)
byzcoin (*Service) executeInstruction 0 of 0.77s (27.70%)
trie (*Trie) SetWithBucket 0 of 0.11s (3.96%)
skipchain (*SkipBlockDB) GetByID 0 of 0.08s (2.88%)
log Lvlf2 0 of 0.50s (17.99%)
bdnproto BdnSignature Verify 0 of 0.15s (5.40%)
protobuf Encode 0 of 0.08s (2.88%)
byzcoin Instruction VerifyWithOption 0 of 0.44s (15.83%)
trie (*Trie) set 0 of 0.10s (3.60%)
bdn Verify 0 of 0.20s (7.19%)
log lvl 0 of 0.51s (18.35%)
protobuf (*encoder) value 0 of 0.08s (2.88%)
schnorr Verify 0 of 0.28s (10.07%)
goparsec doParse 0 of 0.06s (2.16%)
bbolt (*DB) View 0 of 0.11s (3.96%)
bn256 optimalAte 0 of 0.20s (7.19%)
poll (*FD) Write 0 of 0.51s (18.35%)
edwards25519 (*point) Mul 0 of 0.34s (12.23%)
edwards25519 (*point) MarshalBinary 0 of 0.11s (3.96%)
runtime growslice 0 of 0.09s (3.24%)
protobuf Decode 0 of 0.13s (4.68%)
syscall syscall 0.55s (19.78%)
bn256 finalExponentiation 0 of 0.11s (3.96%)
bn256 miller 0 of 0.09s (3.24%)
edwards25519 geScalarMult 0 of 0.21s (7.55%)
edwards25519 geScalarMultBase 0 of 0.13s (4.68%)
edwards25519 (*extendedGroupElement) ToBytes 0 of 0.15s (5.40%)
protobuf DecodeWithConstructors 0 of 0.22s (7.91%)
bn256 (*gfP12) Square 0 of 0.07s (2.52%)
edwards25519 (*projectiveGroupElement) Double 0.02s (0.72%) of 0.08s (2.88%)
edwards25519 feCMove 0.09s (3.24%)
edwards25519 feInvert 0 of 0.14s (5.04%)
protobuf (*decoder) message 0.01s (0.36%) of 0.22s (7.91%)
bn256 (*gfP6) Mul 0.01s (0.36%) of 0.12s (4.32%)
edwards25519 feMul 0.17s (6.12%)
runtime memmove 0.11s (3.96%)
protobuf (*decoder) putvalue 0 of 0.16s (5.76%)
bn256 (*gfP2) Mul 0 of 0.11s (3.96%)
bn256 gfpAdd 0.05s (1.80%)
edwards25519 feSquare 0.19s (6.83%)
runtime newobject 0 of 0.06s (2.16%)
runtime mcall 0 of 0.44s (15.83%)
protobuf (*decoder) slice 0 of 0.15s (5.40%)
bn256 gfpMul 0.11s (3.96%)
runtime mstart 0 of 0.34s (12.23%)
runtime mallocgc 0.02s (0.72%) of 0.19s (6.83%)
runtime schedule 0 of 0.45s (16.19%)
runtime systemstack 0 of 0.38s (13.67%)
runtime stkbucket 0.06s (2.16%)
runtime findrunnable 0 of 0.42s (15.11%)
runtime (*mheap) alloc 0 of 0.06s (2.16%)
runtime madvise 0.11s (3.96%)
runtime scanobject 0.04s (1.44%) of 0.06s (2.16%)
runtime wakep 0 of 0.11s (3.96%)
runtime usleep 0.11s (3.96%)
runtime stopm 0 of 0.34s (12.23%)
runtime memclrNoHeapPointers 0.04s (1.44%)
runtime startm 0 of 0.14s (5.04%)
runtime semasleep 0 of 0.38s (13.67%)
runtime pthread_cond_signal 0.15s (5.40%)
runtime pthread_cond_wait 0.33s (11.87%)

Type: cpu
Time: Jun 5, 2020 at 6:57pm (CEST)
Duration: 8.54s, Total samples = 850ms (9.95%)
Showing nodes accounting for 740ms, 87.06% of 850ms total
Showing top 80 nodes out of 268



v3 (*Overlay) TransmitMsg func1 0 of 160ms (18.82%)
v3 (*Overlay) CreateProtocol func1 0 of 30ms (3.53%)
protocol (*SubBlsCosi) dispatchSubLeader 0 of 80ms (9.41%)
protocol (*SubBlsCosi) makeVerification 0 of 30ms (3.53%)
messaging (*Propagate) Dispatch 0 of 100ms (11.76%)
skipchain (*Service) bftForwardLinkLevel0 0 of 100ms (11.76%)
byzcoin (*txPipeline) processTxs func2 1 0 of 60ms (7.06%)
skipchain (*Service) propagateForwardLinkHandler 0 of 90ms (10.59%)
byzcoin (*Service) verifySkipBlock 0 of 100ms (11.76%)
byzcoin (*Service) createNewBlock 0 of 60ms (7.06%)
skipchain (*SkipBlockDB) StoreBlocks 0 of 80ms (9.41%)
byzcoin (*txPipeline) processTxs func2 0 of 40ms (4.71%)
v3 (*Client) SendProtobufParallelWithDecoder func2 0 of 170ms (20.00%)
byzcoin (*Service) createStateChanges 0 of 110ms (12.94%)
byzcoin (*defaultTxProcessor) ProcessTx 0 of 30ms (3.53%)
byzcoin (*Service) updateTrieCallback 0 of 20ms (2.35%)
runtime mcall 10ms (1.18%) of 70ms (8.24%)
runtime bgscavenge 0 of 30ms (3.53%)
simulation (*SimulationService) Run 0 of 20ms (2.35%)
byzcoin (*stagingStateTrie) LoadConfig 0 of 30ms (3.53%)
v3 (*Client) SendProtobufParallelWithDecoder func1 0 of 170ms (20.00%)
protobuf Encode 0 of 30ms (3.53%)
byzcoin (*Service) processOneTx 0 of 100ms (11.76%)
bbolt (*DB) Update 0 of 80ms (9.41%)
runtime mstart 0 of 190ms (22.35%)
runtime schedule 0 of 70ms (8.24%)
runtime scavengeSleep 0 of 30ms (3.53%)
sha256 block 10ms (1.18%)
byzcoin Proof VerifyFromBlock 0 of 170ms (20.00%)
protobuf DecodeWithConstructors 0 of 50ms (5.88%)
protobuf (*encoder) message 10ms (1.18%) of 30ms (3.53%)
skipchain (*SkipBlockDB) StoreBlocks func1 0 of 40ms (4.71%)
byzcoin (*Service) executeInstruction 0 of 70ms (8.24%)
bbolt (*Tx) Commit 0 of 30ms (3.53%)
runtime systemstack 0 of 200ms (23.53%)
runtime nanotime 20ms (2.35%) of 30ms (3.53%)
runtime findrunnable 0 of 60ms (7.06%)
protobuf (*decoder) message 0 of 50ms (5.88%)
skipchain (*ForwardLink) VerifyWithScheme 0 of 220ms (25.88%)
protobuf (*encoder) value 0 of 20ms (2.35%)
log Lvlf2 0 of 50ms (5.88%)
schnorr Verify 0 of 60ms (7.06%)
runtime madvise 80ms (9.41%)
runtime gentraceback 10ms (1.18%) of 20ms (2.35%)
runtime usleep 20ms (2.35%)
runtime semasleep 0 of 50ms (5.88%)
runtime wakep 0 of 80ms (9.41%)
protobuf (*decoder) putvalue 0 of 50ms (5.88%)
bdnproto BdnSignature Verify 0 of 230ms (27.06%)
edwards25519 (*point) MarshalBinary 0 of 20ms (2.35%)
log lvl 0 of 60ms (7.06%)
edwards25519 geScalarMult 0 of 50ms (5.88%)
runtime pthread_cond_wait 40ms (4.71%)
runtime notewakeup 0 of 110ms (12.94%)
protobuf (*decoder) slice 10ms (1.18%) of 40ms (4.71%)
bdnproto BdnSignature VerifyWithPolicy 0 of 230ms (27.06%)
edwards25519 (*extendedGroupElement) ToBytes 0 of 30ms (3.53%)
syscall syscall 90ms (10.59%)
edwards25519 feMul 20ms (2.35%)
edwards25519 (*projectiveGroupElement) Double 0 of 20ms (2.35%)
runtime pthread_cond_signal 110ms (12.94%)
bdn Verify 0 of 200ms (23.53%)
edwards25519 feSquare 70ms (8.24%)
bls Verify 0 of 200ms (23.53%)
bn256 optimalAte 0 of 190ms (22.35%)
bn256 finalExponentiation 0 of 130ms (15.29%)
bn256 miller 0 of 60ms (7.06%)
bn256 (*gfP12) Exp 10ms (1.18%) of 120ms (14.12%)
bn256 (*gfP12) Mul 0 of 60ms (7.06%)
bn256 (*gfP12) Square 0 of 80ms (9.41%)
bn256 (*twistPoint) Mul 0 of 50ms (5.88%)
bn256 (*gfP6) Mul 0 of 130ms (15.29%)
bn256 (*gfP2) Add 10ms (1.18%) of 20ms (2.35%)
bn256 (*twistPoint) Double 0 of 30ms (3.53%)
bn256 (*gfP2) MulXi 10ms (1.18%) of 20ms (2.35%)
bn256 (*gfP2) Mul 10ms (1.18%) of 150ms (17.65%)
bn256 (*gfP2) Square 0 of 30ms (3.53%)
bn256 gfpAdd 20ms (2.35%)
bn256 (*gfP) Set 20ms (2.35%)
bn256 gfpMul 160ms (18.82%)