HDS Serenity Ledger

Paulo Bolinhas, *110976, Nuno Palma*, *86903 Rui Martins*, *110890*

Abstract

Serenity Ledger is a simplified permissioned blockchain system with high dependability guarantees, that uses the Istanbul Byzantine Fault Tolerance Consensus Algorithm protocol as its core. In this report, we will present our implementation of this application, a codebase that handles several threats, and arbitrary behaviour and guarantees to the client.

The blockchain works as a distributed database that stores transactions of the users, what is known as a ledger. The clients have the possibility to send funds to other clients by sending transactions to the blockchain. These transactions should pay a fee to the leader, pass through the validation system and the consensus protocol. The nodes process and validate these transactions by keeping a local version of the blockchain and an account model similar to Ethereum, mapping the address to the balance.

Despite the guarantees given by the Bizantine Fault Tolerance, our system is prepared to handle other threats coming from both clients and users. As long as the byzantine node number is kept below or equal to one-third of the total, any node should not be able to fake transactions or try to change the balance of unauthorised users. Also, even if the users try to send illegitimate transactions, these are going to be ignored and the balance of the user slashed.

In this paper, we approach how we prevent several attacks or arbitrary behaviours and how we tested those situations.

1. Design

1.1. Entities

As an assumption, every public key is pre-distributed (published) to every Client and Node using config files that are globally known. The Client accesses a library, Client Library, that handles the delivery of requests to the blockchain and gets the responses. The Node interactions in the blockchain are defined in the NodeService class.

1.2. Links, Signatures and Keys

The connection, is secured with authenticated perfect links by implementing digital signatures on top of perfect links. To achieve that, each entity signs on the sending and the receiver checks it. The signatures are created with keys generated with OpenSSL, and after loaded on Java. We did not implement confidentiality since was out of our scope.

It is important to state that the nodes communicate through the class Link but the clients send their messages through their library, Client Library. Despite this, the code of the Link class is similar to the one in the Client Library but adapted to the type of messages that a client can receive and also guarantees the delivery of messages.

2. Implementation

2.1 Client

The client accesses a menu interface where he can choose from the following options: ‘list available clients’, ‘get balance’ and ‘transfer’.

To begin with, the first one simply prints out the identifier of each client that is known by the library.

The ‘get balance’ broadcasts a message ‘GET_BALANCE’ and each node replies with the local balance of the user. As long as less than one-third of the nodes are byzantine, the user is going to get at least f+1 messages with the same value. With this information, the client sets his local balance equal to the received in case of any desynchronization had happened.

The ‘transfer’ option is the more complex one. Here is described only the simple case of success, but further analyzed the possible attacks and arbitrary behaviours. This option asks the user for each data of the transaction, including the fee. The current fee of the network is displayed and the user should put at least that amount to make the transaction work. If he puts more, then his transaction is going to be prioritised. If he puts less, he is going to pay the current fee anyway and the transaction is discarded.

If everything goes well, the user is going to receive at least f+1 ‘TRANSFER’ messages from the nodes which indicate that the transaction he sent was decided. At this moment the client knows that each node executed the transaction so he does the same in his local environment to synchronize with the blockchain.

2.2 Nodes

The blockchain implemented is a closed membership one, which means that the validators are a fixed number.

One of these nodes is the leader that is defined by ‘round mod N’, thus at the beginning of each consensus, the leader is node 1. The leader changes when the round changes.

The leader gathers the transactions until a block can be proposed (the maximum value was set to 2, to facilitate testing) and starts a consensus. Here the Istanbul BFT begins, so to be concise, each node is prepared to receive four types of messages, ‘PRE_PREPARE’, ‘PREPARE’, ‘COMMIT’ and ‘ROUND_CHANGE’.

If the consensus is decided in the first round no round change message is exchanged, otherwise a process to justify the round changes begins.

In order to do this, it was implemented the piggybacking referred to in the IBFT. Thus, if the ‘ROUND_CHANGE’ message comes claiming that the node x prepared a value in some round, then the quorum of signed ‘PREPARE’ messages should come with the ‘ROUND_CHANGE’. This proves that the node did not forge the round change message.

Finally, the amplification phase was implemented too to enhance the performance of the consensus.

3. Behavior Under Attack

3.1. Byzantine Clients

There are several problems that the blockchain can come across related to Byzantine clients. These clients can act either maliciously or arbitrarily (for instance, the clients that unintentionally behave wrongly).

To demonstrate some of the behaviors we created a set of tests that aim to prove the correctness of the system and performance under attack.

3.1.1 Not Enough Balance and Fee

In order to prevent wrong messages from the clients we designed a validation system that prevents wrong transactions from occurring. When a transaction is sent (the client sends a ‘TRANSFER’ message), the node that receives it validates the signature of the message and if it is a repeated message. If it is not, there are two main things analyzed (about incorrect clients), the balance of the user and the fee provided.

The node queries his account model to get the balance of the user and checks if it has enough to support the amount to send plus the fee to be paid. If it does not have, then a message with the error ‘NOT_ENOUGH_BALANCE’ is sent to the client.

Following that, the node checks if the fee provided in the transaction is equal to or bigger than the current fee. If it is not, then a message with the error ‘NOT_ENOUGH_FEE’ is sent to the client.

Even if the transaction does not pass these validations, the current fee is always applied (or the balance is set to zero if the balance is not enough).

It is important to state that these verifications are also done in each transaction when the block is decided. Because there could exist transactions that become invalidated after another is executed.

If the client is a legitimate one with our software, waits for f+1 messages of replying to the ‘TRANSFER’ with an error, otherwise, a malicious node could just send an error, which would enable nodes to oppress clients.

These verifications ensure the correctness of the transactions sent to the network and the consequent blockchain state, finally enhancing the performance of the network by rapidly discarding invalid messages.

3.1.2 Client Trying to Fake Transactions

A client can try to make transactions on behalf of others, to avoid this, we implemented the following.

The network assumes a set of known clients, so their public keys are known by each node (in a real-world situation, new clients could connect by broadcasting their public key). Thus, every time a transaction is sent to the blockchain it needs to be signed by the sender, therefore, the node that receives the message, after checking the signature, uses the public to verify if it corresponds to the source of the transaction. If it is not the current fee is paid by the user to discourage malicious nodes from repeating the message.

This also ensures the correctness of the transactions sent to the network and the consequent blockchain state, finally enhancing the performance of the network by rapidly discarding invalid messages.

3.1.3 Client Trying to Make DoS or Slow Down the Network

The Denial-of-Service attacks are mitigated by our fee system. The fees have a really important job to incentivise the validators to be correct, but also, in this context, more importantly, prevent malicious clients from flooding the network with unintended messages.

If the clients did not pay fees not only the nodes-related network security would be compromised but they could send infinite transactions to slow down or even stop the network. Therefore, when a client sends a transaction to the system, regarding the transaction validity, a fee is paid immediately (as mentioned before) before being added to

the buffer or being validated. This happens in the local account of a client with our software and in the nodes.

This system makes attacking the network economically impractical for the client so that the gains of the attack never compensate for the cost. This makes the performance under attack of the network only a little bit compromised at the beginning because the attacker is going to get out of funds eventually. This could be enhanced even more with a dynamic fee that increases along the network congestion levels which would end the attack even faster.

To test this out we implemented a test that creates a malicious client that floods the network with transactions with fees equal to 1. Regarding this value, the network slashes the value of the current fee on each message. As the user has only 200 tokens and the fee is 20, after 10 transactions he gets out of funds.

3.2. Byzantine Nodes

In this context, akin to clients, the nodes can act maliciously or be arbitrary (for instance, the clients that unintentionally behave wrongly). A node can be incorrect if it crashes down, or if its messages take too long or are lost in the network (more of a network problem that is seen further).

Our system does not have a similar approach to Ethereum where the validators' staked tokens would be slashed if they behaved wrongly, instead, we created a consensus mechanism based on the Instabul BFT consensus algorithm that creates several dependability guarantees and fee system that encourages the leader to be correct.

As long as the Byzantine nodes do not represent more than a third of the total network, these dependability guarantees hold. We will refer to their definitions later, here we are going to explain how nodes behave to avoid inconsistencies by explaining our tests.

3.2.1 Bad Node Signature

Similar to clients, the public key of each node is known globally, so each message that is sent by a node should be signed with its private key and the receiver validates it. This makes the network inaccessible to unknown outsiders and rapidly discards incorrect messages. Which does not mean that a known node can not be Byzantine.

Therefore, we created a simple test that corrupts the signature and, when validates it, it fails.

3.2.2 Replay Attacks

To avoid replay attacks, for instance, the leader keeps a message from the client to broadcast it later, we created a validation on the receiver that checks the id of the message and the timestamp.

3.2.3 Leader Timeout

As mentioned before, a node can have arbitrary behavior by simply crashing. In this test, we reproduced that by simulating the crash of the leader which is going to trigger a timeout in the client because he does not receive the response of his message.

After this timeout, the message is broadcasted by the client and each node starts its timer (in the hope of receiving something from the leader), after this also expires, a round change is triggered and the new leader starts the consensus.

The test is successful because we can observe the inactivity of the leader, the broadcasting, the round changing and a decision being achieved at the end by every node sending their response to the client.

When a leader timeouts like this, it can impact the performance of the network at the beginning because of the timers that wait for the leader to answer, but after that, the consensus proceeds as usual.

3.2.4 Round Change After Prepared Value

The leader timing out is not the only way a round change can happen. When a consensus is started, or a ‘PRE-PREPARED’ message is received, each node starts its timer, if there is any latency in the consensus after a quorum of ‘PREPARE’ messages is received (this means that a value has been prepared), then each timer expires and triggers the round change.

This is also achieved by simulating network delays of the nodes after a value has been prepared. The test is successful because we can observe the value being prepared, the latency after that, the round change and yet a decision being achieved at the end.

3.2.5 Chain Validation

The name blockchain comes from the blocks being linked by having the hash of the previous block. This guarantees tamper-proof, therefore, if a node changes an already decided block a chain validation is going to detect it.

To make the blockchain auditable by an external source we implemented a function that, for each node, computes the hash of the previous block and compares it with the stored previous hash.

If the previous block’s hash is not validated, it will not be linked to the newly created block.
Also, checks if the transactions were made by a known client and if the block was decided in the position that was found.

To test this we created a test that makes some transactions and then runs a function that verifies the entire chain. An additional test was added that allows us to validate the behavior of corrupted blocks, in which the chain validation fails.

3.3 Network Adversities

Before, it was seen that the nodes crashing can impact the consensus speed because certain messages never arrive. This also can happen during moments of network delay or packet dropping.

The problem has different origins but the same impact, thus, the solution is the same. The timers in the nodes and the client aim to detect this and initiate a round change as fast as possible to minimize performance impact.

4. Dependability Guarantees

4.1 Liveness

Ensuring the continuous progress of the consensus protocol is crucial for maintaining system functionality. This is achieved through dynamic round changes and timers, which facilitate the rotation of leader roles among different processes. By periodically initiating new rounds, the protocol guarantees that a leader is eventually selected to drive the consensus process forward, resulting in decisive actions and positive outcomes.

4.2 Safety

The protocol prioritizes safety by employing message justification and quorum-based validations. Upon the arrival of a quorum of 'ROUND-CHANGE' messages with piggybacking, the protocol ensures that the proposal values chosen by the leader of a new round are secure and trustworthy. Through quorum validations and the propagation of prepared values across rounds, the protocol establishes a robust framework for safeguarding against undesirable outcomes and ensuring the integrity of the consensus process.

4.3 Agreement

If a correct process pi decides on a value v, then no other correct process pj can decide on a different value v' where $v' \neq v$. This ensures consistency in decision-making across the network. Moreover, as long as the number of Byzantine nodes remains at or below the threshold f, each node cannot decide on a conflicting value, reinforcing the protocol's resilience to adversarial behavior and maintaining the integrity of the agreement property.

4.5 Validity

Upholding the validity of messages is fundamental to the integrity of the consensus protocol. Correct processes meticulously assess the authenticity and integrity of incoming messages, relying on robust authentication mechanisms such as digital signatures. By requiring evidence of message integrity and sender authentication, the protocol safeguards against malicious actors and ensures the reliability of message exchanges throughout the network.

The external validity predicate ought to be true for the value carried in the 'ROUND-CHANGE' message. If a 'pr' value comes in it, then $pr < r$ should be true.

4.6 Termination

The protocol ensures that every right process will eventually make a decision. By consistently participating in the consensus process, these processes stay strong and committed to reaching clear decisions. This commitment to finishing the process guarantees that the protocol keeps working well, even when facing difficulties or problems

5. Conclusion

In this paper we proposed a solution that mixed concepts of several protocols. Instead of proof-of-work or proof-of-stake the protocol has as the foundation the Istanbul BFT, which guarantees that the network stays correct under byzantine actors. Furthermore, we used the account model of Ethereum, which maps the address of the user to the balance, and the incentives based on higher fees, similar to several blockchains.

This way we built a system that allows clients to transfer funds without third parties, only relying on a blockchain, while ensuring its security, correctness and performance.