HDS Serenity Ledger
Highly Dependable Systems Project
Stage 2 Report

Jaime Gosai 99239
Instituto Superior Técnico

Edson da Veiga 100731
Instituto Superior Técnico

Miguel Rocha 110916
Instituto Superior Técnico

Abstract

This report details the implementation of a simplified permissioned (closed membership) blockchain system with high dependability guarantees, called HDS Serenity (HDS2). In short, the system allows clients to perform crypto exchanges, as well as consulting their current account balance. Mostly of this report is not about the code that implements the system, but rather about the challenges and proposed solutions for them, when guaranteeing robustness against any possible unforeseen events, such as Byzantine behaviour. In the final section, this report describes implemented tests to show how robust the system is.

1 Introduction

This report describes a Byzantine Fault Tolerant (BFT) system designed to be resilient against multiple Byzantine node and client attacks. The system leverages the Istanbul BFT consensus protocol [1] to achieve consensus on transactions among a set of nodes, even in the presence of Byzantine faults. The report details the system's architecture, including the client, node, and link layer functionalities. It also discusses how the system handles various Byzantine attacks, including those launched by Byzantine nodes and clients. Finally, the report highlights an existing limitation concerning balance account confidentiality and proposes a potential solution.

2 Overview

2.1 Client

The client application runs in a loop, accepting user commands... If the client is correct, there are only two operations: Get Balance and Transfer. Yet, if the client is initialized as Byzantine (set in Client JSON config), the Get Balance operation will accept any process ID, and the transfer operation will also accept any process ID as the source account, both accepting Node identifiers. This allows to test possible client Byzantine behaviours.

It is possible that the client wishes to make two consecutive, identical requests (Get Balance or Transfer). Therefore, it's important to distinguish the replies of this two identical, but separate requests, since each client has to wait for a Byzantine quorum of replies, for each request. Therefore, each client, for each new request, generates a UUID identifier, which is included in the same request. Then, it waits for a Byzantine quorum of replies for the request with this UUID.

When a client issues a transfer request, it includes in the request the signature of the message given by the combination of the sourceId, destinationId and amount.

2.2 Node

The node has two purposes:

- Receive and process the requests of the clients (API);
- Runs the IBFT protocol, to achieve consensus of what Block to append to Ledger.

2.2.1 Processing requests

All requests are processed in a separate thread, avoiding bottlenecks of many clients. Whenever a "Get Balance" operation is requested, the request is validated, and, an appropriate response is sent.

On the other side, the transfer request is also validated in the beginning, but no response is sent, unless the validation fails, indicating that the transaction can not be executed (insufficient amount, not authorized, etc). If valid, the node stores the transfer request, until a given number of requests are cached. Then, it initializes a consensus instance, with all cached requests, but only the leader will propose. After the consensus is over, meaning that at least a majority of correct nodes achieved the same decision, the transactions are again re-validated, and the ones not valid, are discarded, before executing them and the Block is appended to the blockchain.

2.3 Link Layer

The provided base code already included the Link layer, which is used as a lower abstraction to send and receive messages between processes. This layer is used both in the Client and Node processes.

The base Link layer, already guaranteed that **duplicated** messages were discarded, and that messages are guaranteed to, eventually, arrive to its destination, with the use of **ACKs**. Yet, the IBFT protocol relies on an authentication mechanism. Therefore, this Link layer was adapted to include one more guarantee: **authenticity** and **integrity**, by using **Digital signatures**.

Whenever a message is to be sent, the Link layer fetches the process **Private key**, and uses it to sign the message. The signature is appended to the **tail** of the original message. In the recipient, the message is split into the original message, and the signature, which is used to confirm the **authenticity** of the message. The correct Public key is chosen based on the **senderId** field, included in all messages. **This field is always guaranteed to be correct if the signature is verified**, since it guarantees that the sender used the **Private key** of the sender (which should be known only be the sender) to sign the message **(non repudiation)**.

3 The most relevant implementation aspects

3.1 Signature extraction in the Link layer

Because the algorithm behind generating signatures is always the same (SHA1withRSA), the signature length is assumed to be always **512 bytes**. Therefore, this split process is trivial.

3.2 Node replies

Whenever a Client issues a request, it doesn't wait for the reply, since the request could take a long time to be processed, depending on the load of the node. **The replies are received asynchronously**.

A client, always waits for a **Byzantine quorum** of replies, before displaying them to the user. This is because, a **Byzantine node** could, on purpose, reply in a non-correct way, and only makes sense if we assume that the replicas follow the rule of **N >= 3p + 1**, where p is the number of Byzantine nodes.

3.3 Leader change per instance

If a Byzantine node, **which is currently the leader**, decides to **ignore most** requests, and only propose the requests of Client X, the other (correct) nodes will receive all proposes, but, since they are not the leader, they are going to wait for the leader to start a **consensus instance**. The leader will only **propose client X requests**. Since the client X requests **are all authenticated** (signed by client X), they are going to be **validated**, and, eventually, **decided**. Therefore, the solution adopted, is to **change the leader for each new instance**.

3.4 Preventing attacks against the Ledger

One possible **attack** is when a Byzantine node, that is currently the leader, decides to propose a consensus instance with a number **much greater** than the previously existent consensus instance, to trick the system into waste a lot of space/blocks. One **apparent** possible solution, is to append the value after the **previous appended next block** of the blockchain. But it is impossible to assume that **different nodes** will decide the **same consensus instance** in the **same order**. Therefore, to ensure that **all nodes** insert the **same block** in the **same index of the Ledger**, the solution adopted in this system, is to **wait for the previous consensus instance to be decided**, and, in this way, ensuring **correctness**.

3.5 Checking transactions in the end of consensus

Lest's imagine the following scenario: a client 1 issues transfer request 1. Then, Client 2 issues transfer request 2. Let's imagine that Node 1 receives both requests and proposes a block with both requests. Before the block is decided, Node 2, also, receives these two requests, and proposes them in consensus instance 2. Both transfer requests are being proposed in two different consensus instance's blocks. As a solution, in the **end of each consensus instance**, each node checks if the given transaction **was already appended to any block** of the blockchain, using the UUID request ID. In short, the problem arises because **different nodes** could receive the requests in **different order**, or **delayed**, and **different consensus instances** having **different leaders**, leading to the possibility of the same transaction to be proposed in multiple consensus instances.

It is, also, possible that **two distinct transaction requests** are issued almost at the same time, and both **valid**. For instance, if the Client has 10 units, and he issues two transfer requests of 9 units, both are validated in the node, (accounting for the fee, ex: 1 unit). Yet, after the first transaction is decided and executed, the second one will not be validated. A **possible solution**, but not implemented duo to **time constraints**, is to store this requests in the client, and make the validation there, to avoid sending transaction request to the nodes, which are obviously invalid.

3.6 Fee value

For simplicity, the value chosen for the fee is 1 unit, per transaction. This encourages **Byzantine clients** not to slow down the system, by having them co-operate and constantly be transferring money between themselves. Also, by being a fixed fee, and not a percentage of the amount of the transaction, it encourages clients to make **one big transaction** instead of **several smaller ones**.

3.7 Detecting transactions made by Byzantine nodes

It is possible that a Byzantine node could try to **propose** a block with a transaction that was not issued by any client. The solution adopted is to, whenever a Pre Prepare/Prepare message is received, **verify the authenticity of every transaction in the block**. This verification is done by verifying the signature (included in each transaction of the Block), with the Public key of the client that issued the transaction. This way, the only way the transaction is validated is if the owner of the account uses **his Private key** to generate the signature.

3.8 Round Change timeout

For simplification, this system starts with a round-change timeout trigger of 3 seconds. Whenever there is a round change, the timeout is **duplicated**, in order to ensure **liveness**. If the conditions of the network deteriorate, the consensus should be able to make progress anyway, even if it's slower.

3.9 Client request broadcast

In this system, the clients always broadcast the request to **all nodes**. The reason is that, they need a **Byzantine quorum** of responses. Therefore, it's not a good idea to broadcast the request to only a quorum of nodes, since one of them could be Byzantine and **decide not to answer**. Also, in the case of the transfer, it's a much better idea to broadcast the request for all nodes, since, even if the leader is the only one that is going to broadcast the proposal, this way, all nodes initiate the consensus instance for that transaction, meaning, they also initiate the **round-change timer**, which is important to **detect non progression**. If the leader is Byzantine, the other (correct) nodes will **trigger a round change**, leading to a **change of leader**. On the other side, if the leader is the only one that receives the request from the client, he could be **Byzantine or slow**, and the transfer might take much longer to be processed.

4 Behaviour under attack

4.1 Byzantine Nodes

To prove that this system is resilient to several Byzantine node attacks, the system was tested with multiple configurations, each one corresponding to a different set of processes and respective behaviours:

- CORRECT_CONFIG: all nodes are **correct**, and the system don't face any attacks;
- IGNORE_REQUESTS: the initial leader (id = 1) ignores all the requests coming towards him. The test shows that, after all nodes start consensus, the other nodes will get their round-change **timers expired**, and they will change their view, so the node with id = 2 will become the new leader, and, since it is **correct**, it will lead to a consensus;
- BAD_LEADER_PROPOSE_WITH_GENERATED_SIGNATURE: What if the **current leader** ignores the client requests, and, instead, proposes one of its own? In this test, the leader (Byzantine) **generates a random block** (with random transactions, each one with a **signature generated by the Byzantine node Private key**), and proposes it. All the other nodes will receive the message, but, when checking the block transaction signatures, **the block will be rejected**. A ROUND-CHANGE will follow, leading to a change of leader to a **correct node**, which will **propose the correct Block**, leading to an eventual decision;
- BAD_LEADER_PROPOSE_WITH_ORIGINAL_SIGNATURE: similar to the test above, but the signatures used with the random generated block, are the **same as the original block**. Yet, once again, the **signature check will fail**, since the transactions are randomly generated, and **don't correspond to the used signatures**.
- BYZANTINE_UPON_PREPARE_QUORUM: shows that if a Byzantine process, **after receiving a quorum of PREPARE requests**, decides to broadcast a COMMIT message, with a **random generated block**, instead of the one extracted from the quorum of PREPARE messages, the consensus will **still decide the right Block**;
- BYZANTINE_UPON_ROUND_CHANGE_QUORUM: the **first node** is **silent**, meaning it doesn't answer requests. The **second node** takes the lead, **after a round change**, and sets a random generated input value/block, on purpose, after receiving a quorum of ROUND-CHANGE messages, instead of computing the HighestPrepared(Qrc)) in the IBFT protocol. It, then, broadcasts PREPARE messages **with the new wrong input value**, and, the **other nodes** won't be able to verify the **authenticity** of this wrong value, leading to a round-change, until a correct node proposes the original block;
- FAKE_LEADER_WITH_FORGED_PRE_PREPARE_MESSAGE: a Byzantine node **is not the leader** but sends a broadcast pretending to be the leader, proposing a random generated block. The other nodes try to verify the **authenticity of the request**, which **will fail** since the Byzantine node **signed** the message with its **own Private key**;
- FAKE_CONSENSUS_INSTANCE: A **correct** client broadcasts a transfer request, and a Byzantine node that

is currently the leader, tries to trick the blockchain by proposing a value for a consensus instance with a **number much greater than the previous one** (ex: 900). The tests show that this consensus instance **will eventually be decided**, but the value **won't be appended to the blockchain until an instance number 899 has been concluded**. Since the other (correct) nodes also received the transfer request, and triggered a timer for the consensus with a valid consensus instance number (ex: 3), eventually, **this timer will expire**, and a leader change will follow, and eventually, this consensus instance will be concluded, and the transaction appended in the blockchain. This will **invalidate** the transaction of instance 900, which will not be appended in the blockchain, since that transaction request **was already** appended in consensus instance 3;

- FORCE_ROUND_CHANGE: a Byzantine node that is not the leader, tries to force a round-change, by broadcasting a round-change message to all other nodes, after waiting a little bit of time for all nodes to initialize the respective consensus instance. The other (correct) nodes will receive the round change, and, since they **only receive one**, instead of a Byzantine quorum, this will have no effect on them. Of course, the Byzantine node would have to have a way to forge the round-change messages to make them look like they came from correct nodes. Yet, it was already proved that it's not possible, since the Byzantine node doesn't have other correct node's Private keys, in order to forge the signature;

4.2 Existent problem with balance account confidentiality

There is a problem that is worth mentioning, which we can't solve it... Since the Byzantine nodes **have access to the client accounts**, they **can reply** to Byzantine clients, asking for another client account balance. One possible way to fix this is if the clients would encrypt the balance value of the account, with a symmetric key, which only they have. Yet, this solution would prevent nodes from validating transactions, since they don't have the symmetric key to access the balance of the client accounts.

4.3 Byzantine Clients

To prove that this system is resilient to several Byzantine clients, each client is parameterized by a ByzantineBehavior type, which, for simplicity, is always the same: IS_BYZANTINE. The possible attacks are the following;

- To simulate unregistered clients in this system, these have ID equal to "null". The nodes ignore these in setup (and also their Public keys). When an unregistered client sends a request to a node, the Link layer cannot authenticate the request, since their ID is not known (in this system is null for simplicity). Therefore, they **cannot do anything** against the system.

- A known client tries to request the balance of another client, by sending the correspondent Public key. Yet, the authenticated Link of each node **already ensures authenticity**. To be authenticated by this layer, the Byzantine node has to use its own Private key, along with its own sender ID, and **only then**, the request proceeds to be handled by the above layer. The node **uses the sender ID**, field to retrieve the account balance. Therefore, the Byzantine client, even using another client's Public Key, will **always receive its own account balance**.

- A known client tries to request an **invalid transaction**, such as transferring an amount that he doesn't have, transferring from another account, that it's not his own, or even transferring units involving node client accounts. The node will first check the validity of the transaction. Yet, what if the Byzantine client has the help of a Byzantine node, which is the leader, and it allows this invalid request? In that case, the Byzantine node tries to start a consensus, for that "invalid" transaction, and, when decided, the other **correct** nodes will **reject** the transaction, since they **validate it again**, after the consensus instance is decided. The Byzantine node will eventually apply the transaction and send a success response, but the other **correct nodes won't**.

References

[1] H. Moniz, "The istanbul bft consensus algorithm," *ArXiv*, vol. abs/2002.03613, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211069668