Cryptocurrency implementation built on IBFT consensus

Simen Myrrusten IST1108924
Tilde Eine IST108444
Goncalo Fernandes IST93260

Abstract

The following report shows the implementation of a simple cryptocurrency based on IBFT consensus. There are two main sides, the clients and the service. For the client, the service and consensus are abstract and they only interact with its application. The service is a fixed set of nodes which agree upon blocks of transactions. Blocks are added to a blockchain after the nodes run consensus on a hash of the block they believe to be next. The implemented system provides the basic functions of a cryptocurrency being able to transfer money and check the balance while being dependable under Byzantine behaviour and attacks.

1 Introduction

This project is an implementation of a simple cryptocurrency using hard consensus through the Istanbul Byzantine fault-tolerant algorithm. The algorithm is used to implement state machine replication in the Quorum blockchain. IBFT is leader-based and allows f faulty processes out of n processes where $n \geq 3f+1$. It assumes a partially synchronous communication model, where it only needs periods of synchrony to maintain the liveness property and security does not depend on timing assumptions. [1] A shell of the IBFT algorithm was already provided to the group upon starting the project. This included communication channels and provided normal behaviour of the IBFT algorithm.

For a complete cryptocurrency, multiple features needed to be completed. First of all, there was a need for a client interface such that clients could send and receive money. Furthermore, round changes needed to be implemented to handle byzantine leader nodes and have fair payment of transfer fees. On the application side, both the nodes and clients needed an application which handled the transfer and worked as an interface to the consensus protocol. Finally, tests were developed to prove the dependability property and ensure that the application worked as intended.

2 System design

The system contains four main folders: Utilities, Service, Communication and Client. The Utilities consists mainly in functions from the initial code provided, but also the Authenticate class, used for the verification and signing of digital signatures. The Communication contains all the messages that are sent. Most relevant is the added RoundChangeMessage and ClientMessage, which are respectably used for changing round and sending and responding to client requests. ClientMessage contains the data class ClientData which is used for handling transactions.

The Client library was created from scratch and leverages the Communication and Utility libraries for the setup of clients and creating messages. The Client.java file is the main file which builds the clients, while all the application logic can be found in ClientService.java. This includes transferring money, checking the account balance and verifying the collective response from the service application.

The Service library is kept at each node. It has two main features; running the IBFT consensus algorithm and handling transactions/balances. Transactions are received and verified before being put into

blocks. A hash of the next block will then be agreed upon using the consensus. More on this in 3.1.

3 Implementation details

This section will go through the main features in the code and the reasoning behind their design.

3.1 Blockchain

As most cryptocurrencies implemented blocks, so did we on this project. It comes with the advantage of not running consensus on every single transaction and therefore leads to fewer messages in general. Our blocks are simple and contain the block ID, which increments by one for each block, the previous block hash and the transactions to be executed in this block.

The transactions are sorted on the clients and the client's request ID, such that each node's block is equal. This is important as the consensus is not done on the blocks themselves but on a hash for the blocks. The reason for making consensus on a hash of the block is to prevent the packages from becoming too large. Even though our application will not be stress-tested, a real-world network would be less congested if every consensus message contained the full block.

Every node starts consensus every 15 seconds. At this time a block will be created and filled with either incoming transaction requests or waiting transactions from a queue. Before the block is stored in a local map, any invalid transactions are removed from the block. Invalid transactions include overspending, double-spending and invalid receivers. Then a hash of the block is calculated, and the leader starts consensus.

If consensus is achieved, the nodes will grab the agreed-upon hash and check against a calculated hash of the locally saved block. Every correct node will have a match and append the block to the blockchain. The nodes do not need to do another validation of the transaction as a change in the transactions would have changed the hash itself.

3.2 Round change

The IBFT algorithm dependability guarantees depend on its round change mechanism, which we implemented in the Service library, which all the server nodes use to reach consensus. The round change mechanism is used in case of a faulty leader or untimely communication, ensuring the liveness of the consensus process.

We also implement a maximum number of rounds of 4, corresponding to the number of nodes. If the number of rounds exceeds 4, then all the nodes had the opportunity to lead for a round, and if there was no decision in those rounds, the instance is undecided and we move on to the next instance. This helps ensure liveness in case of a problem with the instance.

We use a timer of 5 seconds, which is the time allocated to reach a consensus on a given round. If consensus is not reached in that time, each node initiates a round change process, broadcasting RoundChangeMessages. This new round will have a new leader, in an alternating fashion, responsible for the round. This process is repeated until consensus is reached.

3.3 Authenticated links

Our system intends to simulate a real network. It is an industry-standard to require confidentiality and integrity of communication. Being a project about dependability, we do not provide confidentiality as that is out of our scope. However, we would recommend that a real-world application would use HTTPS for all messages, such that confidentiality is upheld.

To make sure that both clients' and nodes' integrity was upheld, we implemented authenticated links with digital signatures. When any message is sent, it is signed. The signature is attached to the message and verified on the receiving end. To sign the message, it is serialized into json. On the receiving end, the message will only be considered if the signature is valid. This guarantees that the contents of the message were not changed, assuring integrity and authenticity.

3.4 ClientData, transfers and fees

To make sure that the transfers from clients are valid and immutable, digital signatures were leveraged. Even though we have authenticated links, it would be possible for the application nodes to change the values themselves, when they are considered by the application. To prevent the leader or someone else from changing the values we created an object called ClientData. This contains the ClientID, RequestID, the value and a signature. The value is a string containing the amount to be transferred and the receiver, and will be parsed upon validating and executing the transfer in the nodes. When a node receives a transfer message from a client it will verify the signature of the value before being put in the queue of transfer. Furthermore, another verification will be done right before the hashing of the block, which is mentioned in 3.1.

The transfer itself will be executed after consensus has been achieved. This will update a local map clientBalances, that keeps track of the balances of the clients. This mapping is used to avoid going through all the blocks each time we need to find out what the balances are and thereby improve response time.

During the execution of the transfers, a fee of 10% of the transfers will be given to the leader. This should be further improved before deploying the blockchain as the same node will always be paid if no round change is forced. The fee is also quite large and would need to be justified or lowered. Justifications could be the burning of currency during transfers or a highly inflationary cryptocurrency. Neither is implemented in this project and 10% is kept for demonstration purposes.

After execution and updating of the local clientBalances, each node will send back a CLIENT CONFIRMATION to the client. The clients need to be sure that they get at least one response from a correct node. Therefore, the client application will wait for f+1 CLIENT CONFIRMATION before confirming that the transfer was executed.

3.5 Check balance

A similar approach to the CLIENT CONFIRMATIONs is done when we check the balance of a user. The client can ask for its or any other client's balance. The client application will then broadcast this request to all nodes and ask for the balance in focus. They will then wait for a QUORUM of Q = 2f+1 equal responses. The reason for this is that now we need to have a QUORUM on the returned balance instead of just a confirmation. In the case of an executed transfer, all the client needs to know is whether or not it has been processed. In this case, we want the exact value and therefore a QUORUM is required to ensure that we get the correct value.

4 Behavior under attack

This section will detail some of the tests we have implemented, and how they demonstrate the system's functionalities, as well as dependability under Byzantine behaviour. Our tests are divided into three main test classes: *NodeServiceConsensusTest.java*, *NodeServiceTransactionTest.java*, and *NodeServiceByzantineClientsTest.java*. These extend our *NodeServiceBaseTest.java* class, which contains helping functions for testing. For simplicity, we focus on the nodeService, which is where the main mechanisms of our system work.

4.1 Security

- *testRejectReplayedMessage*: Ensures that the system can reject replayed messages. We use the request ID as a nonce that protects against a node replaying a previous transfer, which would otherwise pass verification since the message will have a valid signature.

4.2 Validity

- *testNoBlockIfInvalidHash*: Ensures blocks with incorrect hashes are not added, upholding the blockchain's integrity.

- *testRejectUnsignedClientData*: Ensures unsigned client data is not propagated, protecting against Byzantine nodes altering transaction values during consensus and maintaining transaction authenticity.
- *testClientBalancesNegativeMoney*: Ensure we prevent the addition of transactions that could corrupt the ledger state by introducing negative values.
- *testClientBalancesNonExistentClient*: Ensures we prevent invalid transactions directed to nonexistent entities, ensuring all transactions have valid participants.
- *testClientOverspend*: Ensures a user cannot spend more than their balance, a critical check for transaction validity.
- *testInvalidData*: Ensures that any transactions or data that do not conform to expected formats or values is rejected, crucial for maintaining system integrity. Also checks that the system does not crash from this, just ignores the message.
- *testDoubleSpendingRejected*: Demonstrates the system's ability to reject attempts at double spending, which would violate the integrity and agreed-upon state of the blockchain, and let a client "use money twice".
- *testClientImpersonation*: Ensures that one client can't impersonate another one and make transfers on his behalf.
- *testNoCommitNoTransactionToBlock*: Shows that the system does not commit blocks that do not have a quorum, which contributes to the integrity of the blockchain.

4.3 Liveness

- *testCommitAddsTransactionToBlock*: Shows that the system correctly adds valid transactions to the blockchain, contributing to liveness.
- *testCommitOnPrepareQuorum*: Shows that the consensus mechanism functions under correct conditions, when we get a quorum of prepare messages, ensuring the system remains live and reaches agreement.

4.4 End to end

- *testCompleteRun*: Illustrates the system's end-to-end transaction handling capability.

4.5 Byzantine Leader Election and Behavior

- *testUpdateLeader*: Directly tests the system's capability to switch leaders, an essential feature for recovering from Byzantine leader behaviors.
- *testConflictingLeaderPrePrepareMessages*: Checks the system's response to conflicting messages from the current leader, ensuring the system performs round change.

5 Conclusion

We achieved an implementation of the IBFT consensus algorithm similar to the one described in [1]. The project implements blocks, which are executed every 15 seconds and are similar to the industry standard. Furthermore, dependability guarantees such as security, validity, liveness and protection against Byzantine behavior can be guaranteed, as shown by the tests provided.

Further work could be done to benchmark the application to find out how well the application performs. In addition, configurations such as forced round-change, block size and time between blocks could be tested to figure out the best configuration. This would likely also need a context of use, as client demand would very much determine the optimal configurations.

References

[1] Henrique Moniz. *The Istanbul BFT Consensus Algorithm.* May 2020.