Formatting Submissions for a USENIX Conference: An (Incomplete) Example

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Your abstract text goes here. Just a few facts. Whet our appetites. Not more than 200 words, if possible, and preferably closer to 150.

1 Introduction

2 Overview

In summary, this system is composed by Client and Node processes.

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 Byazantine, the Get Balance operation will accept any process ID, and the transfer operation will also accept any process ID as the source process. This allows the testing of possible Byzantine behaviours by clients.

It is possible that the client makes two consequitive identical requests (Get Balance/transfer). Yet, it is important to distinguish the replies of this two identical but diferent requests, since the clients wait for a Byzantine quorum of replies, for each request. Therefore, each client, for each request, generates a UUID, which is included in the request. Then, it waits for a Byzantine quorum of replies for this UUID request.

2.2 Node

Each node stores the state of the blockchain and serves two purposes:

- Receives and processes the requests of the clients;
- Runs the IBFT protocol, in order to append transactions to the Ledger.

2.2.1 Processing requests

All requests are processed in a separate thread. 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. If validated, the node initializes a consensus instance, but only the leader will broadcast the proposal. After the consensus is concluded, meaning that at least a majority of correct nodes achieved the same decision, the transaction is again re-validated, and, if valid, is appended to the blockchain, and an appropriate response is sent to the client that requested the transaction.

2.3 Link Layer

The provided base code includes 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 replica processes.

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

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, plus the signature, which is used to confirm the authenticity of the message. The correct public key is chosen based on the sender ID field, included in all messages. This field is always guaranteed to be correct if the signature is verified, since it guarantees that the emiter 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 Extracting the signature from the message coming from the network

Since the algorithm is for generating signatures is always the same, the signature length is assumed to be always 512 bytes. Therefore, this split process is trivial.

3.2 Handling replies from the server (nodes)

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 server. The reply is received asynchronously.

To ensure authenticity, since there could be Byzantine replicas that reply in an erroneous way. Therefore, the clients wait for a Byzantine quorum of replies, before displaying them to the user. This 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 full-fill the requests of Client X, the other (correct) nodes will receive all requests, 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, they are going to be validated, and, eventually, decided. Therefore, the solution adopted, is to change the leader for each 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 while trying to append the value/block into the blockchain, since it could mean that it exists a lot of consensus instances still pending. One apparent possible solution is to append the value after the next block of the blockchain. But this leads to other problems where a node could receive a quorum of commits for Transaction1, and then, for Transaction2, and another node could receive those in the reverse order. Therefore, to ensure all nodes insert the same block in the same index of the Ledger, the only solution, and the one adopted in this system, is to wait for the previous consensus instance to be decided. If it's decided for one node, it's decided for every node.

3.5 What if a transaction is duplicated?

Lest's imagine the following scenario: a client 1 broadcasts a transfer request 1. Then, Client 2 also broadcasts transfer request 2. Node 1 and 2 receive these two requests in the opposite order. So, Node 1 (leader for instance 1) proposes request 1 in consensus instance 1, and request 2 in consensus instance 2. Node 2 (leader for instance 2) does it in the opposite order. Therefore, they are both proposing the same transaction for both instances. As a solution, in the end of the consensus, each node check if the given transaction was already appended. To check this, each transaction has a unique UUID, since the client can make multiple similar transactions (transferring the same amount to the same destination). In short, the problem arises because different nodes could receive the requests in different order, and different consensus instances having different leaders, leading to the same transaction to be proposed in multiple consensus instances.

3.6 Consensus Instance Ending

Whenever a consensus instance is over, the transaction is re-validated by all correct nodes. This is crucial since it is possible that two transaction requests were issued almost at the same type, and both valid during the request. For instance, if the Client has 10 units, and both requests ask for a transfer of 10 units, both are validated, (accounting for the fee, ex: 1 unit). Yet, after the first transaction is decided and executed, the second one should not be validated. What happens, in this case, is that the transaction won't be validated, but the consensus instance will still be considered decided/over This is important because there could be a third request, issued in the meantime, with a valid transaction, and this, will only be appended after the previous instance is over/decided. This means that the number of the consensus instance is not necessarily equal to the index of the block where the transaction is appended.

3.7 Fee value

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

3.8 Detecting transactions made by Byzantine nodes

It is possible that a Byzantine node could try to propose a transaction/value that was not issued by any client. The solution adopted is to, whenever a PRE-PREPARE/PREPARE message comes in, verify the authenticity of the value. This verification is done by authenticating the value signature (included in the consensus messages), with the Public key of the client that issued the transaction (the owner of the account where the money will get out).

3.9 One transaction per block

Ideally, in a real blockchain, the system waits for multiple transactions to be requested, before proposing them to the blockchain. Yet, duo to time constraints, we couldn't make the necessary changes to support multiple transactions in one block, while ensuring a robust system, and, so, this system supports only one transaction per block.

This might seam like a bottleneck, but, the system also supports multiple consensus instances running at the same time. If the node receives 10 transaction requests, and he starts 10 consensus instances, one after the other, they will all run at the same time, in parallel. Therefore, it is possible they end all at the same time. The only constraint is that, the transaction, in each block, is validated only after the previous one have been.

3.10 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. The duplication of the timeout guarantees that.

3.11

In this system, the clients always broadcast the request to all nodes. The main 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 issue the request for all nodes, since, even if the leader is the only one that is going to broadcast the request, this way, all nodes initiate the consensus instance for that transaction, meaning, they also initiate the round-change timer, which is important to detect problems with the current leader. 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, this could be Byzantine or slow, and the transfer will take much longer to be processed.

4 Behaviour under attack

4.1 Byzantine Nodes

To prove that this system is resilient to several Byzantine nodes, 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: a single node will ignore the requests coming towards him. In particular, this one will start as leader (id = 1). The tests will show that, after all nodes start consensus, the other nodes will get their 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 the consensus to a decision;
- BAD_LEADER_PROPOSE_WITH_GENERATED_SIGNATURE: What if the leader ignores the client requests, and, instead, proposes one of its own? In this test, the leader (Byzantine) generates a random value, and signature with its own Private key (the only one it knows) and proposes it. All the other nodes will receive the message, but the signature check will fail. A ROUND-CHANGE will follow, leading to a change of leader to a correct node, which will propose the correct value, leading to an eventual decision;
- BAD_LEADER_PROPOSE_WITH_ORIGINAL_SIGNATURE: similar to the above test, but the signature used with the random generated value is the original one, received with the client transaction request;
- BYZANTINE_UPON_PREPARE_QUORUM: shows that if a Byzantine process, after receiving a quorum of PREPARE requests, and decides to broadcast a COMMIT message, with an incoherent value, the consensus will still decide the right value;
- BYZANTINE_UPON_ROUND_CHANGE_QUORUM: the first node is silent, and it doesn't answer requests. The second node takes the lead, after a round change, and will set a wrong value after receiving a quorum of ROUND-CHANGE messages. It, then, broadcasts PREPARE messages with the new input value, and, the other nodes won't be able to verify the authenticity of this new value, leading to a round-change, until a correct value proposes the original value;
- BYZANTINE_UPON_ROUND_CHANGE_QUORUM: a Byzantine node is not the leader but sends a broadcast pretending to be the leader. The other nodes try to verify the authenticity of the request, which will fail since the Byzantine node has to sign the message with its own Private key;

- FAKE_CONSENSUS_INSTANCE: A correct client broadcasts a transfer requests, and a Byzantine node that is currently the leader, tries to trick the blockchain into 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 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 to the blockchain. This will invalidate the transaction of instance 900, which will not be appended to 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 messages to all other nodes, after waiting a bit for all nodes to initialize the respective consensus instance. The other (correct) nodes will receive the round change, since they only receive one, and not a Byzantine quorum, this will have no effect on them. Of course, the Byzantine node would have to get a way to forge the round-change messages to make them look like they came from correct nodes. Yet, we already proved that it's not possible because the Byzantine node doesn't have other correct node's Public keys, 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, it is possible that they 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 is processed. The node uses the sender ID, field to retrieve the account balance. Therefore, the Byzantine client, even using a different Public Key, will always receive its own balance account.
- A known client tries to request an invalid transaction, such has transferring an amount that he doesn't have, transferring from another account, that it's not his own, or even transferring units involving none 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, which will not check it? 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