HDS Ledger - Group 08

Our system implements a Token Exchange System application of the currency BTE.

Processes

There are two types of processes present in our system:

- Client processes
- Istanbul Member processes

Client processes

The mode of operation of the client processes is the following. First, the client waits for the user to input the next operation. Then, the client broadcasts the operation request to all the consensus members in the network, using an authenticated perfect link. The client then waits for a reply from a quorum of the processes $(2f + 1)$, after which it shows the result to the user. The only exception to this is that on a weak read, the client reads the result of the first reply and verifies a quorum of valid signatures.

Client commands utilize a system of PIDs for transfers, which is actually an alias for their public key, translated through the EncryptionUnit class.

Server processes

Comunication with client

Each server is listening for requests from either another server or a client.

When a server receives a request from a client, it will process the request (which may or may not require it to run a consensus instance), and it will send back the appropriate reply.

Token Exchange System

Currently, our TES builds a blockchain where blocks can hold up to 4 transactions of either Transactions or Accounts. All transactions and account creation requests have a fee incurred of 1 BTE, which are always paid to the miner process: the process with pid=1 which is also the consensus leader. New accounts start off with a balance of 100, but also have to pay the mining fee upon creation. Snapshots are created every 4 blocks appended, upon a successful consensus instance.

Upon receiving a request from the client, the server will do as follows, according to the operation requested:

- Transfer and Create: first, we verify if the request is signed by the client. After that, we create a second transaction, corresponding to the mining fee, and we insert both in our current block.
- Strong read: we read the client's balance in the local state, and reply with the value.
- Weak read: we reply to the client with the latest snapshot available (more on that later).

The TES always has an incomplete block where it appends new transactions as they arrive. When the block is full, we check if the block is valid (unknown account, insufficient balance, invalid amount, etc.), and if it is, we start a new consensus instance with that block as the input.

Members of the consensus also check the block received upon receiving a PRE_PREPARE. In case of an invalid block, they discard it and don't begin a new consensus instance. When the consensus decides a block, the block operations are finally applied to the local state.

Transactions have the client signature checked upon receival. Block integrity and valid blockchain state checks are done before consensus.

Consensus

The mode of operation of the consensus members is the following. First, the members are waiting for a request from the client. When such request is received, each process sets up a new consensus instance. Then, the current leader starts by broadcasting a PRE_PREPARE message to all other members, and starts a timer t. Upon receiving a PRE_PREPARE message from the recognized leader, every correct process will broadcast a PREPARE message to every process.

Each correct process, after receiving a quorum of PREPARE messages, will broadcast a COMMIT message. When a correct process receives a quorum of commit messages, it will locally save the decided value. This marks the end of the consensus instance.

Should the timer t expire before a consensus is reached, a simplified round change protocol will be triggered.

Snapshots

To support weak reads, where the client only reads from a single server, not a quorum, we needed to guarantee that even if the client contacts a byzantine server, the server won't be able to send an invalid balance. This works using snapshots. Every 4 blocks, the members of the network will create a snapshot of their current state, and all correct processes will sign it and exchange signatures during a snapshot consensus process. This way, when the client receives a snapshot, it can verify if the received snapshot is signed by a quorum of processes, meaning that it is a guaranteed valid state. Given this, the balances in the snapshot may be stale, but were always correct at some point in time.

Comunication Links

Fair Loss Link

All of the Fair Loss Link guarantees are assured by the UDP implementation present in Java. This layer will be used as a base link for other layers.

Authenticated Perfect Link

Built upon the FLL, our Authenticated Perfect Link also ensures the behaviour of a Stubborn Link. This was included in our APL design for optimization reasons. This means our Perfect Link guarantees that:

1. If the recipient doesn't acknowledge the message, the APL will resend the message.
2. If the APL receives the same message multiple times, it will only deliver it once.
3. Every message is signed, so it can't be forged by another process.
4. Every message has a nonce, so replay attacks are not possible.

For communication, we utilize a BEB.

Dependability Guarantees

The TES receives signed and timestamped Transactions from client applications, meaning that the system ensures non-repudiation. This signature system also prevents unauthorized account access to bad actors. Blockchain integrity is guaranteed through the verification of sequentiality (through the use of hashes) of the soon-to-be appended block at the start of a consensus instance. Invalid account states and other illegal transactions being present in blocks also cause them to be discarded, meaning that client requests could be discarded in the process. For this reason, clients can only have the guarantee that their requests were successfully appended to the blockchain through reads.

This means that the blockchain will never have or return an invalid internal state.

Implementation Threats

For this implementation of a TES, we have considered the following possible **Byzantine** behaviour threats. For each one, we also make explicit which design choice, security mechanism or dependability guarantee assures that this threat doesn't affect the expected behaviour:

Non-Authorized Process Intrusion:

Processes with no registered public and private keys won't be able to get through the APL's message signature checks.

Replay Attacks:

The APL layer implements nonces, preventing this kind of attacks.

f Byzantine Processes:

As per protocol design, our implementation maintains service functionality in cases of up to

$$f = \frac{n-1}{3}$$

non-responsive or incorrect processes, given a total of n processes. Even if f processes collude to send erroneous values to other processes, since they don't have a quorum they won't be able to prevent the other processes from making progress, even if they refuse to sign, send wrong values, etc.

Byzantine reply on Weak Read:

As explained before, the weak reads work with a system of snapshots. Since the snapshots collect a byzantine quorum of signatures (from all the processes that agree on the current state), then byzantine processes can never forge a wrong snapshot without it being detected by the client.

Incorrect Protocol Messages:

The protocols specifications handle byzantine values for each operation, due to the quorum based behaviour.

As the client also reads from a quorum of replies, a byzantine server's incorrect blockchain will not affect its reading, since incorrect values will always be a minority.

Single Process Quorums:

The protocol is implemented in a way that a byzantine process isn't able to reach a quorum by repeating its proposal in each phase of the algorithm (e.g, sending 4 PREPARE messages in an n=4 run), due to the fact the extra messages will be discarded by the other blockchain members.

Client Sending Server Protocol Messages:

Client process keys are identified and client messages are handled in a way that they won't be able to interfere with the protocol's progression.

Byzantine Client Behaviour

All client commands send serialized objects which include a timestamp and are completely signed with the client's private key. Upon receival, client transactions or account requests have their signature verified with their expected public key, meaning that servers impede unauthorized access from malicious clients.