Introduction

All communications between **Users** and **Notary** were implemented using **Java RMI** protocol. The users and the Notary implement different interfaces in order to handle requests.

Transaction

Every time a user sends a request, the Notary creates a **transaction** in order to execute (or not) that request.

Every transaction has a **state**. When a transaction is created the Notary checks if the request is legit (check digital signature and freshness) and if the Good is not currently on another pending transaction.

If a Good is on a pending transaction the newer one is **cancelled**, and the user is informed.

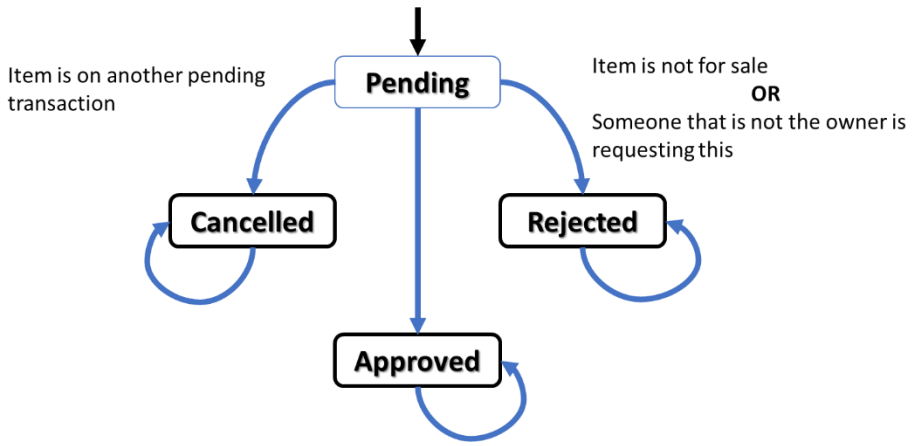

Figure 1: State machine of a transaction

When all checks are finished and the transaction is **approved** the Notary context is altered, which is written to disk (persistent) and only then the response is sent to the user.

Security

Messages Authenticity and Integrity Control

All Notary's reply to user requests are certified by the Notary as described in the following process:

- Notary creates an `Interaction` which is a response to be sent to the client.
- Notary computes a message digest using `SHA-256` algorithm with respect to all fields, namely:
  - Buyer/Seller ID

- Buyer/Seller clocks
- Response to the request

All *user to user* and *user to notary* requests and responses are certified by the emissary in the same way as described above. Let's check all the procedure for the `buyGood` operation.

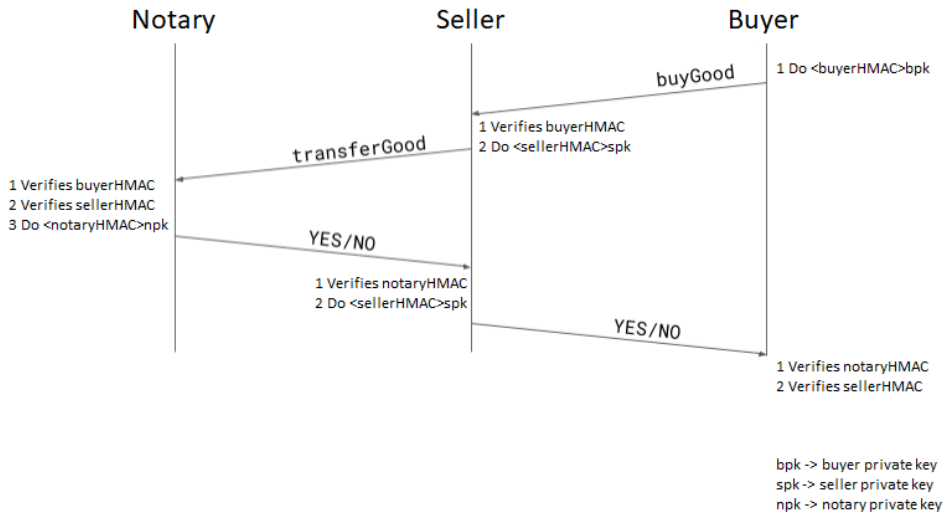

Figure 2: Message Authenticity procedure

This way we guarantee both integrity and non-repudiation of exchanged messages.

Replay Attacks

In order to prevent replay attacks the following procedure is adopted (See Figure 4):

1. User knows their clock (to the Notary).
2. User send the request with their clock to the Notary.
3. The Notary checks and executes the request and sends back the response to the client and **includes the same** clock value. Notary stores the current clock value of the user after sending the response to the client.
4. The client checks the authenticity of the message and the freshness, which is a comparison of the clocks.

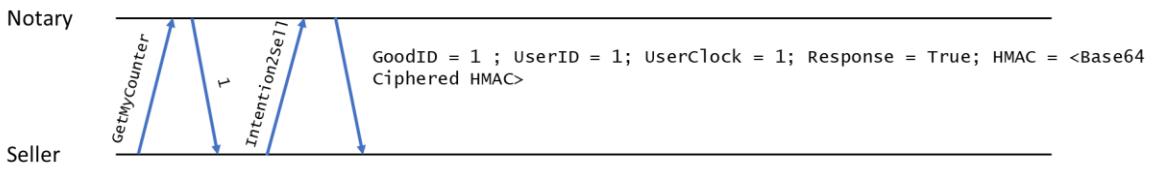

Figure 3: Replay attacks prevention procedure

Let's suppose that Eve is eavesdropping the network and capturing packages. Suppose that one seller is contacting the Notary to inform that some item is now for sale (1) and that Eve was able to capture that message.

Later Eve tries to send that exactly same message to the Notary. That instruction even though is legitimate (it is properly signed by the Seller) it's not fresh (because the clock of the seller cannot be lower or equal to 1) and therefore is detected and rejected.

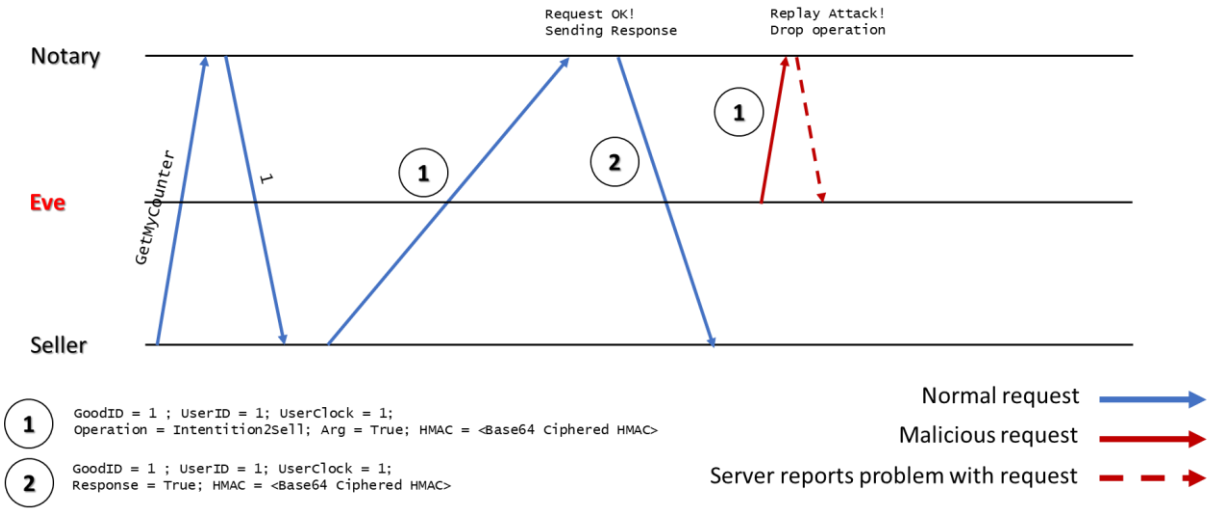

Figure 4: Replay attack dropped by the Notary

Sybil attacks

Since we are properly using digital signatures this kind of attack is not an issue.

Considerations regarding security

It is important to have in mind that the Notary **always use its private key to sign data**. This is a big issue, because the **probability of exposing the private key increases**. In order to prevent this, some secret should be shared between Notary and each of the users, this way the notary private key is only used as a handshake, after this step, a secret key should be exchanged, and all further data should be signed using that secret. This secret must be renewed to prevent its excessive use.

Dependability

Requirements

Notary must be crash tolerant (without replication), preventing it from losing its state.

Solutions

We have two files:

- `UserGoods.bin:`

  This file keeps updated information about the users and goods.

  Modifications considered:

  - Logical clocks (with these clocks we avoid clock synchronization's issues);
  - Good's owner;
  - Good marked for sell;
  - Good marked as in transaction.

  We only write to disk when we finish `intentionToSell` or `buyGood`, this way we never get an inconsistent disk state, either the operation is fully performed and then written in disk or if it fails somewhere in the middle, the state modifications are ignored.

  When Notary crashes and then goes up, all state changes are recovered.

- `Transaction.bin`

  This file is updated when a transaction is started or terminated.

  If a transaction is started and the Notary crashes, when it goes up again the transaction is performed and deleted from the file when finished.

We are aware of some **problems** with this solution:

- There is a **probability of corruption** if the Notary crashes while the file is being written. We discussed it with Professor Paolo Romano. In a real system an **atomic database** would be used in order to guarantee **atomic writes**;
- The **time gap** between the Notary internal state modification and the write operation on disk to make it persistent. This is not a big issue because if an operation finishes and is not written to a file, no user will be able to see the operation as finished. This issue could be fully addressed in the future stage of the project, where we would have **multiple Notaries** providing **replication**.