The system is composed by a notary server (notary), a library which provides an API to communicate with notary (notary-client), the client servers (user), an interface for the users to perform their transactions and queries (user-client) and a library with the cryptographic methods required for the system, which includes methods to read and write asymmetric keys and to implement digital signatures. The servers are implemented using tomcat and the communication are implemented using REST.

In this system, one of the possible attacks is the replay attack, where the messages previous sent are fraudulently repeated. Another possible attack is an attack to the message integrity, i.e. tampering maliciously with the message. It can also be carried out an attack to the message authenticity, i.e. forge the message sender. In this context, there may be attempts of Sybil Attacks. However, as all public keys are known by everyone, it isn’t possible to exist this kind of attack, since the users can’t forge another user identity since all messages are signed.

To ensure **integrity** and **authenticity**, it is performed digital signatures. In order to implement digital signatures, a string consisting of the method, its parameters and the timestamp or the 2 timestamps (sent and answered) in case of being an answer is constructed. The digital signature is made with the private key and receives as input the constructed string and it is sent as an additional parameter. The receiver of the message constructs the same string with the parameters received and checks with the sender’s public key, if the signed string is the same as the one constructed. Since the message is signed with the private key of each user or notary, the authenticity of the message is ensured since only the user or notary can access their private key and therefore only they can sign with it. It also ensures the integrity of the message since the method, their parameters and the timestamps are signed, so if there is any kind of attempt to tamper the message it will be detected in the signature verification and the message will be rejected. The signatures are built and verified with the Signature object of Java.

To prevent **replay attacks**, unix timestamps are added to the messages. The timestamps corresponding to the last message received from each user are stored in the notary. Thus, when the notary receives a message, after verifying that the signature is valid, it verifies if the timestamp is later than the last received timestamp, therefore ensuring freshness. Since the users don’t save a state, they receive the responses with the timestamp of the notary or the server of another user along with the timestamp corresponding to the request, avoiding the replay of previous responses. This also allows the user to verify if the answer corresponds to the request. The replay attacks are mitigated with this mechanism because the timestamps are also signed by the sender so, they can’t be forged. In addition, it prevents an attack window. In case of error, the timestamp is also sent. The timestamps are obtained using the method System.currentTimeMillis().

In the case of transferGood, the notary receives the Buyer and the Seller signature to ensure that the Buyer was the one who wanted to obtain the good and that the Seller is the one who wants to sell it (authenticity).

Digital signatures are performed with the sender’s private key. However, to avoid cryptanalytic attacks due to excessive use of the private key and because asymmetric encryption is more expensive, session keys should have been generated whenever a user wants to communicate with the notary and MAC should have been used instead. However, for the sake of simplicity of implementation only private and public keys are used.

The private keys are encrypted with a password and a salt and are stored in a file. The keys are encrypted using Java Crypto API.

To ensure **dependability** of the notary in the presence of crash faults, the current state of the notary is stored in a serialized file and whenever there is a change in the notary state it is stored in the file notary.ser. Thus, when there is a crash fault, the server recovers its state by reading the serialized file, which acts as a “backup”, ensuring reliability and integrity.