Universidade de Aveiro - Departamento de Eletrónica, Telecomunicações e Informática
Grupo 1 Class P2
Davide Cruz - 71776, João Aniceto 72255

Segurança

Blockchain-based auction management

Final Report

February, 2019

By:
Davide Cruz 71776,
João Aniceto 72255

Table of contents

Introduction

In the last decades the use of computers and digital technology has grown exponentially, they have become a pillar of our society, fundamental for almost all industries, unfortunately these systems aren't perfect, either by design or not, they are riddled with security and integrity vulnerabilities that an ill-intent user can exploit.

In this class it was asked to design and develop a blockchain-based auction management program, with a server-client architecture, and to apply security mechanisms to ensure its integrity and confidentiality.

This report illustrates one implementation of this project, and explains the choices that were taken to arrive to this solution.

Auction

Auction states

An auction can have three states:

- Open
- Claiming Mode (with time out defined by who created the auction)
- Closed

An auction only enters claiming mode if it is a blind auction. English auction only have open or closed states.

An auction begins open upon creation by the client. If the time limit defined by user upon creation is reached the auction is closed if english type or enters claiming mode if blind type. The blind auction stays in claiming mode until the time for claiming mode defined by the client upon creation is passed.

The user can also preemptively terminate/close the auction. If english it immediately goes into closed mode. If blind it goes to claiming mode and follows its normal course there, giving the defined window for the users to send their keys.

Auction type

An auction can be one of three types:

- Open ascending price auction, a.k.a English auction - each bid must overcome the value of the previous one. Bids for this kind of auction have a cleartext value and an encrypted identity, which can only be revealed by the Action Manager (upon the end of the auction).
- Sealed first-price auction or blind auction - Bid amounts are encrypted. At the end of the auction all bids are decrypted by the Auction Manager, yielding, upon claiming, the auction winner.
- Full blind - A subset of the blind auction, it works exactly the same except the identities of the bidders are also hidden.

Bid

An bid is composed by an identity, a value, and a type (English, blind or full blind).

Encryption

Throughout the project we will utilize both asymmetric and symmetric algorithms, in some cases their were used together in what is called a hybrid encryption.

Asymmetric encryption

During the development of this project it was made the assumption that each server has a non-certified, asymmetric key pair with a well-known public component, for this reason no key exchange was needed (e.g Diffie-Hellman).

The asymmetric encryption algorithm chosen was Rivest-Shamir-Adleman (RSA), due to its popularity, and the fact that the encryption and decryption uses different keys, one public and another private, its implementation was a safe and easy alternative.

Symmetric encryption

In order to implement some of the asked features, symmetric encryption had to be implemented.

This project supports 5 different symmetric algorithms:

- Advanced Encryption Standard (AES) - a block cipher that is both fast, and cryptographically strong.
- Camellia - block cipher that is considered to have comparable security and performance to AES.
- Data Encryption Standard also known as Triple DES - is a block cipher known by having crypto-analytic flaws, however none of them currently enable a practical attack, it is also slow compared to available alternatives.
- CAST5 (also known as CAST-128) - is a block cipher. It is a variable key length cipher and supports keys from 40-128 bits in length.
- SEED - is a block cipher developed by the Korea Information Security Agency (KISA).

The client is also given a option to choose between 5 different symmetric encryption modes:

- Cipher Block Chaining (CBC) - is a mode of operation for block ciphers. It is considered cryptographically strong.
- Counter (CTR) - is a mode of operation for block ciphers. It is considered cryptographically strong. It transforms a block cipher into a stream cipher.
- Output Feedback (OFB) - is a mode of operation for block ciphers. It transforms a block cipher into a stream cipher.
- Cipher Feedback (CFB) - is a mode of operation for block ciphers. It transforms a block cipher into a stream cipher.
- CFB8 - CFB variant that uses an 8-bit shift register.

Most of the combinations of the algorithms and modes are supported, however there are the following exceptions:

- camellia & CTR
- camellia & CFB8
- triple DES & CTR
- CAST5 & CTR
- CAST5 & CFB8
- SEED & CTR
- SEED & CFB8

Blockchain

A blockchain originally block chain is a growing list of records, called blocks, which are linked using cryptography Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. By design, a blockchain is resistant to modification of the data.

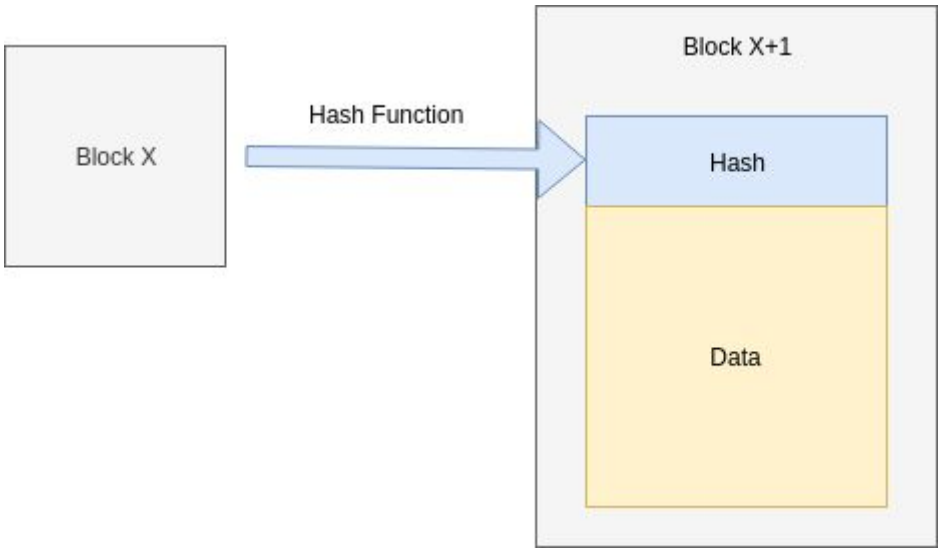


Fig1. Introduction of the new block in a blockchain

One of the requirements of the project was that each auction must be implemented by a blockchain, with a bid per block.

In a auction blockchain each block contains an hash of all the previous blocks, a timestamp of when the block was added, and a data field, this field contains the auction data for in the first block, and the corresponding bid information for each subsequent block.

When an auction closes the blockchain also closes, this is done by adding a new block with the data field empty.

Client Functionality

When the client app is first booted it shows the menu with the following functionality:

- Create an auction
  - The user can create an auction with personalized name, description, type (english, blind, and full blind), time limit, and claim time (if blind or full blind).
- Terminate an auction
  - Terminates an auction created by the user.
- List auctions
  - The user can see a list of all the auctions an their current state (open, closed, claiming).
- List bids of a auction
  - List all the bids of a auction, except if its a full blind auction, in which case it will only know the number of bids.
- List my bids
  - List all the bids a user has made
- Bid on auction
  - Gives the user the opportunity to bid in any auction, of course that if that auction is closed a error message will be returned.
- Check outcome of a auction/bid
  - Checks the outcome of a auction
- Validate receipt
  - Validates a receipt chosen by the user
- Verify and see all info about a closed auction
  - Lists all the bids of a closed auction, checks its validity and decrypts its bids.

The client app validates the Citizen Card (CC) card of the user when necessary.

Message protocol

Message structure

All messages follow a similar general structure composed with three fields that can be empty or not:

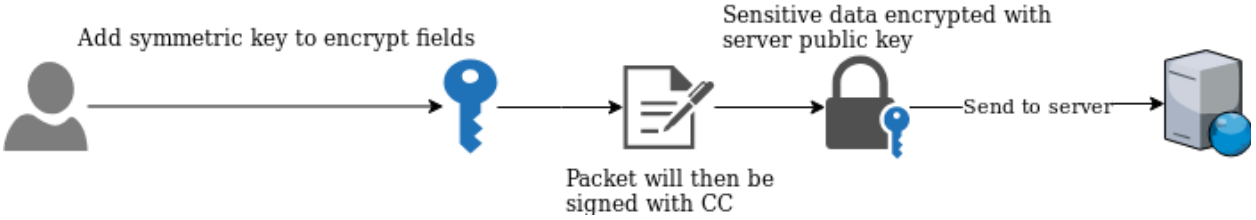
- ***data*** - a dictionary that contains all the message information excluding the certificate and signature, some of its fields can be encrypted. There are only two mandatory fields, the message id and the message type (we also refer to them as packet id and packet type respectively).
- ***signature*** - contains the sender's signature (when necessary) over the data, either using certificates or the manager/repository public key.
- ***certificate*** - although its always there most times will be empty since it's only used by some packets from clients.

```
{
      "data" : {
                  "id" : <packet id> ,
                  "type" : <packet type> ,
                  <other message information>,
                },
      "signature" : <sender's signature>,
      "Certificate" : <sender certificate>
}
```

Connection properties

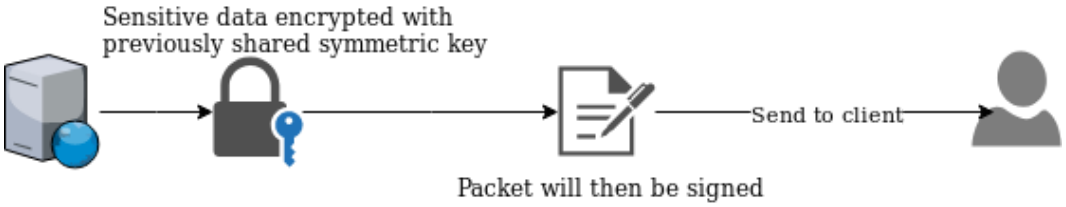
UDP will be used for all communication.

The communication for clients-servers:



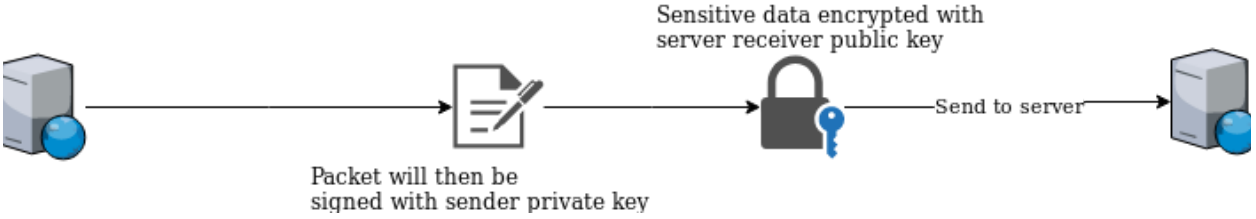
By signing with the citizen card the server guarantees the sender authenticity and integrity. If needed the client also sends a symmetric key to encrypt possible sensitive returning data. This symmetric key will be sent encrypted with the servers public key.

The communication for servers-clients:



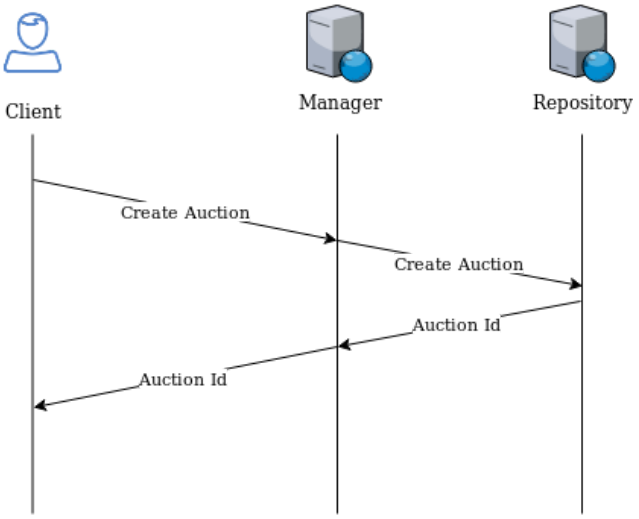
Fields that are sensitive will be encrypted with a symmetric key sent in the request. Packets will be signed with the servers keys to assure the source and integrity.

The communication for manager <-> repository:



Create auction

This is the simple process for creating an auction. The client needs to communicate through the manager for it and the manager with the repository for the effective creation.



Process for creating a auction

Request - Client -> Manager

```
{
  "data" : {
    "id" : <packet id>,
    "Type" :  <packet type>,
    "Auction_type" : <auction type> ,
    "claim_time" : <if the auction is blind or full blind it will be a
number of minutes, if is english it will contain 'None'>,
    "time_limit" : <time limit to bet>,
    "name" : <name of the auction>,
    "description" : <auctions description>
  },
  "signature": <data signed by client citizen card>,
  "certificate" : <client citizen card certificate>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" :  <packet type>,
    "status" : <error or success>,
    "auction_id" : <auction id>,
  },
  "signature" : <data signed by manager private key>,
  "certificate" : <empty>
}
```

Request - Manager -> Repository

The manager's job is to tell the repository to create the auction with the client's parameters.

```
{
  "data" : {
            ...
  },
  "signature": <data signed by manager private key>,
  "certificate" : <empty>
}
```

Response

```
{
  "data" : {
           ...
  },
  "signature": <data signed by repository private key>,
  "certificate" : <empty>
}
```

Terminate auction

Follows the same process as create Auction but instead terminates given an auction type. The end result may have different consequences depending on the auction type.

Request - Client -> Manager

```
{
  "data" : {
    "id" : <packet id>,
    "type"  :  <packet type>,
    "auction_id" : <auction id>,
  },
  "signature": <data signed by client citizen card>,
  "certificate" : <client citizen card certificate>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "Status" : <error or success>,
    "auction_id" : <auction id>,
  },
  "signature": <data signed by manager private key>,
  "certificate" : <empty>
}
```

Request - Manager -> Repository

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "auction_id" : <auction id>,
  },
```

```
   "signature": <data signed by manager private key>,
   "certificate" : <empty>
 }
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "Type" : <packet type>,
    "status" : <error or success>,
    "auction_id" : <auction id>,
  },
  "signature": <data signed by repository private key>,
  "certificate" : <empty>
}
```

List auctions (closed/open)

With a simple request from the client to the repository we can get all the auctions. It is also possible to filter by open or closed auctions.

Request - Client -> Repository

```
 {
   "data" : {
     "id" : <packet id>,
     "type" : <packet type>,
     "auctions_state" : <OPEN/CLOSE/ALL>,
   },
   "signature": <Empty>,
   "certificate" : <Empty>
 }
```

Response

```
 {
   "data" : {
     "id" : <packet id>,
     "type"  :  <packet type>,
     "status" : <error or success>,
     "auctions"  : [
       {
         "id" : <auction id>,
         "type" : <English/Blind/Full_Blind>,
         "name" : <auction name>,
         "description" : <auction description>,
         "creation_date" : <auction creation date>,
```

```
        "time_limit" : <time limit to bet>,
        “claim_time" : <claim time if exists>
      },
      ...
    ],
  },
  "signature" : <data signed by repository private Key>,
  "certificate" : <empty>
}
```

List bids

Gets all the bids given an auction id. In case of a english gives the bid value and timestamp. For the blind auction how many bids where submitted.

Request - Client -> Repository

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "auctions_id" : <auction id>,
  },
  "signature": <Empty>,
  "certificate" : <Empty>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type"  :  <packet type>,
    "status" : <error or success>,
    "bids"  : [
      // If english
      {
        "date" : <timestamp of the instant the bid was added to auction>,
        "value" : <value of bid>,
      },
      ...
      // If blind
     {
        "identity" : <identity of bidder>,
      },
      ...
      // If full blind  we return not a list but only the number of bid
```

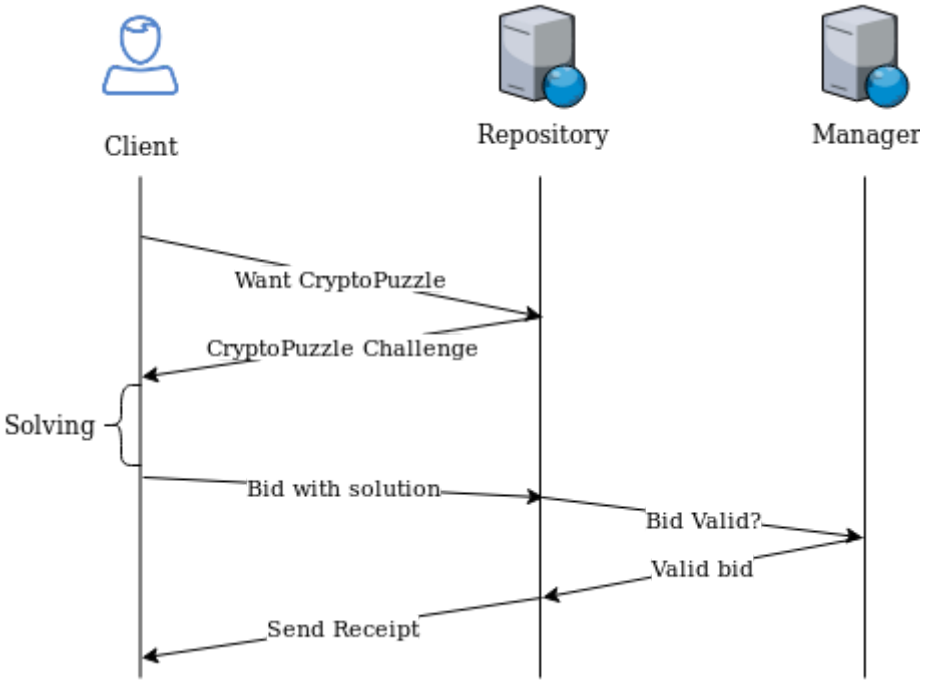
```
    ],
  },
  "signature" : <data signed by repository private Key>,
  "certificate" : <empty>
}
```

Bid on auction

The bidding process starts with the crypto-puzzle phase. Only with a solved challenge (proof of work) can the user submit a bid. So first of all it must ask for a challenge to solve.

On the next phase the client sends the bid attempt to the repository. The repository forwards the bid attempt to the manager. If the repository receives the ok to insert from the manager it insert the bid in the corresponding blockchain and forwards the receipt to the client.

The third phase is the bid validation by the manager which is performed in the middle of the previous phase. The manager check if the bid is valid. If it is gives it send the ok to the repository.



Bidding process

1º Step Crypto-Puzzle Phase - **Client -> Repository**

In this step, the client requests a crypto-puzzle challenge to solve. The repository generates a random one and sends it back.

Request

```
{
  "data" : {
    "id" : <packet id>,
    "type"  :  <packet type>,
  },
  "signature": <empty>,
  "certificate" : <empty>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "Status" : <error or success>,
    "challenge" : <crypto puzzle challenge>,
  },
  "signature": <data signed by repository private key>,
  "certificate" : <empty>
}
```

2º Step - Bid Phase - **Client -> Repository**

In this step, all the bid parameters are sent, such as the value. The result of the crypto-puzzle challenge is also sent and is checked by the repository.

The packet is different in case the bid is for a blind or a english auction. In case of a blind auction, only the client should know the value until the end of the auction. So it is sent encrypted.

Also to hide the identity from the repository, we need to encrypt the client's certificate with something only the manager know. It would be wise to encrypt with manager's public key but unfortunately the certificate is too big for the RSA key. To work around that we use a hybrid key that is sent also with the packet. The hybrid key is used to encrypt the certificate and it is sent encrypted with manager's public key.

In every step of the interaction, the last packet is encapsulated into the next and signed. This means that every interaction is signed on top of the other. This gives us a secure object and the assurance of the path and process taken. This is used to create/validate the receipt.

The symmetric key is sent by the client to hide sensitive return data. The server will encrypt the response sensitive data with the algorithm and mode of choice algo sent in the request. Obviously this key is sent encrypted with the repository public key to hide it from possible MitM attacks.

The client receives a receipt which is the result of multiple encapsulations of packets signed. The receipt and status of the request is sent encrypted to the client using the symmetric key he sent in the request. This is sent encrypted so that external MitM doesn't have a clue whether a bid was inserted or not.

Request - English type auction

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "symmetric_key" : <key to encrypt receipt> (Encrypted with repository
public key),
    "Hybrid_key" : <key to encrypt certificate> (Encrypted with manager
public),
    "auction_id" : <auction id>,
    "algorithm" : <algorithm to encrypt receipt>,
    "mode" : <mode to encrypt receipt> ,
    "crypto_puzzle_result" : <crypto puzzle challenge result>,
    "bid": <value>
  },
  "signature": <signed by client but with certificate encrypted with
manager public key (hybrid)>,
  "certificate" : <clients citizen card> (Encrypted with Hybrid key),
}
```

Request - Blind type auction

```
{
  "data" : {
    ...
    "bid": {
      ...
      "value" : <value> (Encrypted with a symmetric key generated and kept
by the user)
    },
  },
...
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success> (Encrypted with symmetric key),
    "receipt" : <bid receipt> (Encrypted with symmetric key),
  },
  "signature": <data signed by repository private key>,
  "certificate" : <empty>
}
```

3º Step - Validate bid - **Repository -> Manager**

It is on this step the dynamic code should run, and the bid is validated or not. If is validated the Manager and it encrypts the identity if the auction is english or full blind with a key generated by auction upon its creation.

Request

```
{
  "data" : {
    "id" : <packet id>,
    "type"  :  <packet type>,
    "bid"  :  {<bid from client> (signed by repository and encrypted)},
  },
  "signature": <data sign by repository private key>,
  "certificate" : <empty>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success>(Encrypted with Repository public key),
    "bid" : <bid to be inserted inside blockchain>,
    "packet": <last packets encapsulated>,
  },
  "signature": <data signed by managers private key>,
  "certificate" : <empty>
}
```

Check auction outcome

This is the process the client uses to get information about the winner or to claim its bid in case of a blind auction.

English auction - **Client -> Manager**

The same process with the use of a symmetric key, algorithm and mode is used to hide sensitive returning data. The key is encrypted with the manager public key.

Request

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "symmetric_key" : <symmetric key (encrypted)>,
    "algorithm" : <algorithm>,
    "mode" : <mode>,
    "auction_id" : <auction id>,
  },
  "signature": <data sign by client citizen card>,
  "certificate" : <clients citizen card certificate>
}
```

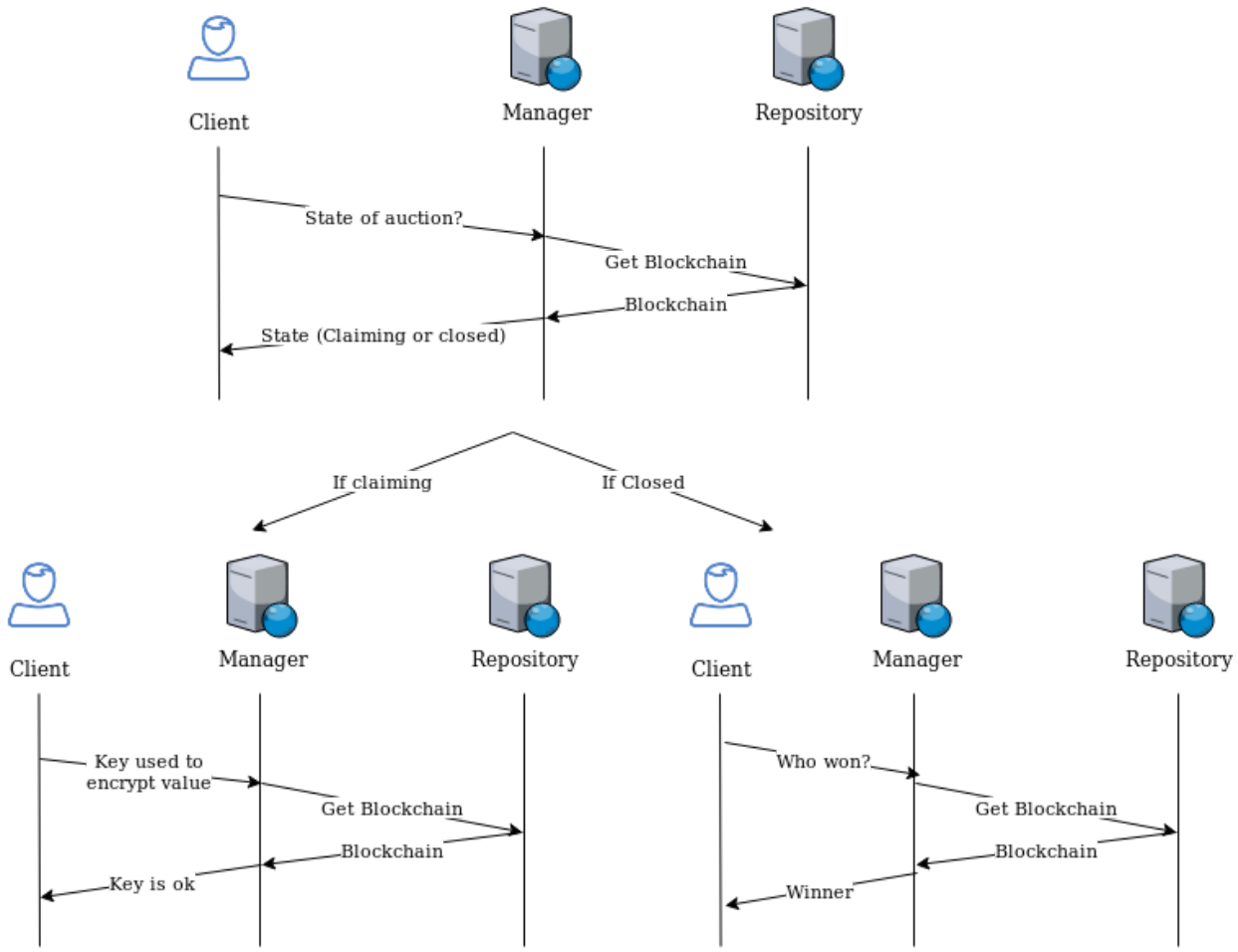
Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success> (Encrypted with symmetric key),
    "winnerID" : <Winner Identity> (Encrypted with symmetric key),
    "winnerValue" : <Winner Value> (Encrypted with symmetric key),
  },
  "signature": <data signed by managers private key>,
  "certificate" : <empty>
}
```

Blind auction - **Client -> Manager**

The process is a little bit different for the case of a blind auction. This is the time the client must send its key used to encrypt the value.

The first packet exchange is used to check whether the auction is in claiming mode (where the users should send the key used to encrypt the value), or to closed (get the winner).



Check outcome of blind bid process

1st Packet - Verify auction state

Request

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "symmetric_key" : <symmetric key (encrypted)>,
    "algorithm" : <algorithm>,
    "mode" : <mode>,
    "auction_id" : <auction id>,
  },
  "signature": <data sign by client citizen card>,
  "certificate" : <clients citizen card certificate>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success>,
    "statusAuction" : <status of auction: closed or in claiming mode>
(Encrypted with symmetric key)
  },
  "signature": <data sign by managers pubic key>,
  "certificate" : <empty>
}
```

2nd Packet - if auction is unclaimed mode

If the first packet tells the client that the auction is in claiming mode, the user must submit the key used to encrypt the value when the bid was sent. If the user doesn't do this during claiming window its bid is invalidated as the the manager couldn't decrypt the value.

The manager gets the respective blockchain from the repository to check the value and bid respectively.

Request

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "key_blind_bid" : <symmetric key to decrypt bid value> (Encrypted with
manager public key),
    "auction_id" : <auction id>,
  },
  "signature": <data sign by client citizen card>,
  "certificate" : <clients citizen card certificate>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success>,
  },
  "signature": <data sign by managers public key>,
```

```
  "certificate" : <empty>
}
```

2nd Packet if auction has ended

If the auction has ended, the winner is calculated and sent. The manager get the blockchain, verifies its integrity and performs the calculations needed to get the winner.

Request

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "symmetric_key" : <symmetric key (encrypted)>,
    "algorithm" : <algorithm>,
    "mode" : <mode>,
    "auction_id" : <auction id>,
  },
  "signature": <data signed by client citizen card>,
  "certificate" : <clients citizen card certificate>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success>,
    "statusAuction" : <status of auction: closed or in claiming mode>
(Encrypted with symmetric key)
  },
  "signature": <data signed by managers private key>,
  "certificate" : <empty>
}
```

Get blockchain

This packet is used when the manager or the client needs to see the blockchain to get data from it or validate something. It returns the blockchain of the auction id sent in the request.

Request - Manager -> Repository

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "auction_id" : <auction id>,
  },
  "signature": <data signed by managers private key>,
  "certificate" : <Empty>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success>,
    "blockchain" : <complete blockchain>
  },
  "signature": <data signed by repository private key>,
  "certificate" : <empty>
}
```

Request - Client -> Repository

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "auction_id" : <auction id>,
  },
  "signature": <Empty>,
  "certificate" : <Empty>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success>,
    "blockchain" : <complete blockchain>
  },
  "signature": <data signed by repository private key>,
  "certificate" : <empty>
}
```

Get auction keys

This method is used by the client to request to the manager the keys used to encrypt the identities in the auction if needed and the key used by the clients to encrypt their bids. Obviously this method only returns the keys if the auction is closed after verification from the blockchain returned from the repository.

It is used by the client in conjunction with get blockchain to validate a auction.

Request - Client -> Manager

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "auction_id" : <auction id>,
  },
  "signature": <data signed by clients citizen card>,
  "certificate" : <clients citizen card certificate>
}
```

Response

```
{
  "data" : {
    "id" : <packet id>,
    "type" : <packet type>,
    "status" : <error or success>,
    "identity_keys" : <keys to decrypt identity of bids (if auction is
english or blind)>
    "blind_keys" : <keys to decrypt values of bids (if auction is blind or
full blind)>
  },
```

```
  "signature": <data signed by manager private key>,
  "certificate" : <empty>
}
```

Certificate Validation

To validate the certificate of the users we must guarantee its certificate chain. The servers have the relevant upper certificates that can validate the Citizen Authentication Certificate.

To validate a certificate we must for each certificate in the chain until the root auto-signed:

- Is signed by the certificate issuer
- Is not on the its CRL's
- The CRL's signature is valid
- Is not expired

We go through the 4 steps in the chain, validating these 4 points every time. To help dealing with the certificates we use the python library cryptography[1] methods and variables.

CryptoPuzzles

We use hashcash[2] which is currently most known by its use in bitcoin.

The process is simple: given a challenge (string) and a number of zeros (difficult) we need to find a token that given the hash of string+token has the number of zeros at the beginning. That's why getting the number of zeros is the difficult. Higher number requires mores tries to calculate a hash.

To solve you simply need to: generate a token, which is all the combinations of string going from lower to higher size, until the hash of the token+challenge has the number of zeros required.

To check: hash the solution(token)+challenge and check the number of zeros

Obviously to check you only need to hash one time, but to solve you need to find the right values, iterating until a solution.

This process is applied to refrain the client from spamming bids. This makes the client require a PoW to bid. The same principle is usually applied to email using hashcash.

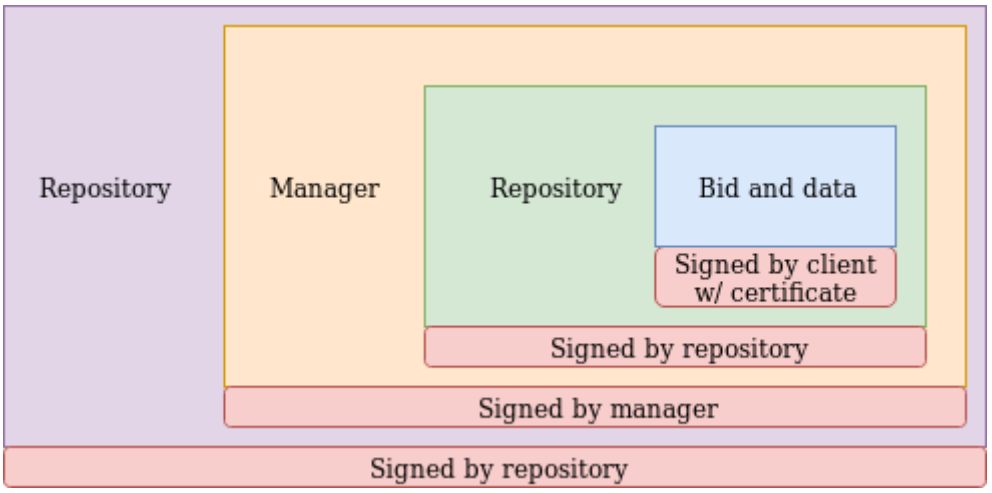
Given that there is multiple implementations of the hashcash for python we decided to take one[3] and adapt it. We transformed it from python 2 to python 3 and simplified it. This makes the code more readable and adapted to our needs.

[1] https://cryptography.io/en/latest/x509/
[2] http://www.hashcash.org/
[3] https://gist.github.com/i3visio/388ef5154052ed8173df4b7b9eda541b

Receipt

Structure

The receipt is a secure object that is representative of the whole path and process of the bid. By having the signature inside signature we can block the information from changing and verify that the server read the information and approved it.



Representation of the structure of the receipt

Validation

By validating the signatures inside the receipt (including our own), we can guarantee the authenticity of the whole process. This is proof that the manager and repository validated and inserted the bid into the blockchain. We can check at the end of auction if there is tampering/cheating involved and have proof that we made this bid.

Additional Considerations

Certificates Citizen card

After checking the certificates provided by the state for the citizen it is to note that even though they have the same extension, they aren't in the same format even though they are from the same "serie"/level. Some are formatted in DER while some are in PEM. To deal with this we need to check which format is the certificate in before loading it into python package cryptography.

Configuration to execute the application

There is also a file which lists all the requirements used in the project so that the user can easily install python dependencies by calling pip -r requirements.txt .

Base requirements:

- Python >= 3.7 (because of a datetime function used 17)
- Pycharm (optional)

Python packages requirements:

- colorama==0.4.1
- cryptography
- Pykcs11
- Citizen card middleware[4]

To run the project:

1. Check if you have all the requirements:
    a. Run "pip install -r requirements.txt"

    OR

    b. Install manually each package
2. Run Project

    > cd src/

    > python clientMain.py

    > python managerMain.py

    > python repositoryMain.py

[4] https://www.autenticacao.gov.pt/cc-aplicacao

Conclusion

The objective of this project was to implement a secure blockchain based auction managed program, that follows a server-client based architecture.

After working on this project and reading this report one can conclude, that this is not a linear and simple task, several challenges are tackled in which choices and compromises have to be made, often between security and integrity and performance.

References

https://en.wikipedia.org/wiki/RSA_(cryptosystem)

https://en.wikipedia.org/wiki/Blockchain

https://cryptography.io/en/latest/

https://pkcs11wrap.sourceforge.io/api/index.html

http://www.hashcash.org/papers/hashcash.pdf