Blockchain Project: An auction system to (finally) have a chance to book a room at HSG

Advanced Numerical Methods and Data Analysis
University of St. Gallen

Course professor:
Dr. Peter Gruber

St. Gallen, 29.05.2018

Damien Baldy - 17-620-626

Danilo Zocco - 11-510-013

Heang Mao Heng - 13-760-905

Sabine Wilke 17-608-522

Thomas Mendelin 13-051-156

Twinkel Van Impe 17-601-311

I. Introduction

A. Problem

The room booking system at HSG is primitive and not very efficient. Firstly, the number of rooms is limited, considering the increasing number of students and the focus of the HSG on group projects that require regular meetings. Secondly, the rooms often end up overbooked by people that abuse the system. Despite existing limitations, they can still book rooms but never use them, or book a room for the maximum period while needing only one or two hours. As many people don't plan weeks ahead to book rooms, the co-working rooms end up all booked and it becomes hard to find a place to meet, building up frustration as the students don't have time to waste to check if every room allocated is occupied. The result is that only a few students use the booking system, while most meet up in the library building and wander around looking for a not too noisy spot where they can discuss and concentrate. The problem persists, and the university hasn't come up with a solution to allocate rooms efficiently and ensure a balanced booking system.

When a resource is abundant, it is not so critical to let people take it freely, without much supervision. But when a resource is scarce, there are several mitigation possibilities, such as supervision by a central entity or regulation by the market economy where money settles the problem of affectation and the highest bidder gets the resource. In the case of room allocation, supervision by the university is required: the HSG does not want students to abuse the system, but allocation of the rooms should still be flexible and efficient.

B. Why use the blockchain

As a solution to this problem, we propose the creation of a token on the Ethereum blockchain that we call H$GCOIN. Each student will be assigned a fixed and equal amount of H$G at the beginning of each semester. These tokens would serve for any internal transaction that requires allocation of scarce resources: course allocation, room booking, participation at events,... Since bidding on a room (and getting it) costs H$G which can therefore no longer be used for future room bookings, students will not bid on rooms unless they are sure they actually need one.

The blockchain is used as an open-source database so anyone can consult and place his or her bids. The smart contract is run by the organisation, but the whole process of bidding and booking happens on the blockchain. The idea is to provide full transparency and fairness to the process of booking. People would have to be careful about the way they allocate their tokens and the limited amount available each semester would make it more balanced to allocate the scarce resources.

II. Solution

We provide two different smart contracts which implement an auction system to efficiently allocate rooms to students who need them the most. Thus, we prevent overbooking rooms and time slots. These contracts address the room booking part of the problem. The first one uses ether to bid and the second one our own H$G token.

A. Bidding System I

The administrator would have to be programmed as an Ethereum node outside the blockchain, for example on a server and would pass the order to run the auction and allocate the rooms. The process happens as follows:

1. The administrator deploys the contract on the blockchain (needs to be done only once)
2. He sets the time span for the auction.
3. He triggers the reset function that lets the user bid for a given time span (for instance 24 hours)
4. The users transparently place their bids on the rooms associated with time slots, the bidding gets stored in the contract.
5. If someone gets out bidded, they are given back their bid automatically
6. The auction ends, and the administrator allocates the rooms to the winning users
7. The contract transfers the winning bid to the administrator

Only the admin can use the functions which start the bidding and allocates the rooms to students. Once the auction has ended, the rooms are assigned to the winning students temporarily. When the administrator starts a new round of bidding (e.g for the next day) all rooms get assigned again to him. The file “auction.sol” contains a working version of this bidding system, where participants bid with integer amounts of ether. In reality, the bidding system should not work by bidding with ethereum but with a fixed amount of a token issued by HSG at the beginning of every semester. In this script, one can bid for two rooms with two time slots. In order to place a bid, one has to type in the room and slot combination as an integer. 101 stands for the first room and first slot, 201 stands for the second room and first slot, 102 for the first room and second slot, 202 for the second room and second slot. The bidding system keeps track of the biddings. The admin is not allowed to bid.

Several functions are included to increase the user-friendliness of the application, such as the possibility to look up the address of the administrator, to see the highest current bid and to retrieve the room assigned to one’s account.

We choose to implement an open auction for full transparency, meaning everybody sees the leading bid. A blind auction in the case of room allocation is problematic since then a student only knows at the very end if he gets the room or not. However, in a classic auction he is able to exactly bid according to his preferences. If he gets overbid and he is not willing to bid even higher. then the room is allocated to another student which is willing to pay an even higher price. The address of the smart contract on Rinkeby is:

```
0x33bF9176d2614a403c3aFAFA003ec68386173d85
```

B. Bidding system with tokens

We adapted the bidding system in order to run it with the custom H$G token. This boils down to using the smart contract of the token in order to make the money transfers. We need to add the prototype of the ERC20 token at the top of our *RoomAuction* contract, so that it knows what the ERC20 contract looks like. *RoomAuction* also needs to inherit the function to receive tokens from the interface *tokenRecipient*.

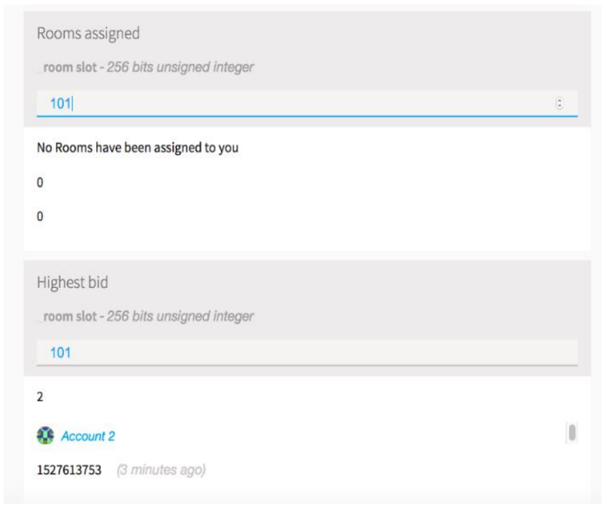
```
contract ERC20 {
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint balance);
    function allowance(address tokenOwner, address spender) public constant
returns (uint remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns
(bool success);
    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint
tokens);
}

interface tokenRecipient { function receiveApproval(address _sender, uint256
_value, ERC20 _tokenContract, uint _roomSlot) external; }

contract RoomAuction is tokenRecipient {
…
}
```

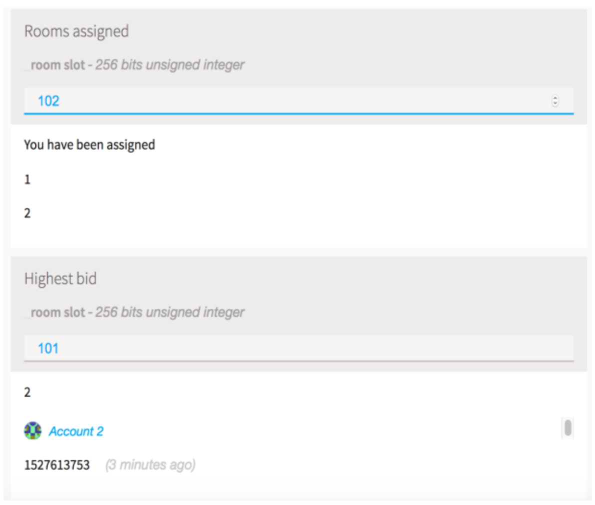
Then the bidding works as follow: the user calls the function *approveAndCall* from the token's smart contract, that we have adapted to pass a uint to the bidding function (data types conversion is still too rudimentary in solidity to support bytes to uint transformations). This function gives the right to an address to transfer up to a given amount of tokens to another: the user gives the right to the *RoomAuction* address to transfer up to the amounts he wants to bid. The token smart contract then triggers the *RoomAuction* smart contract that then makes the token transfers and takes the bidding into account.
In the token smart contract, we have:

```
function approveAndCall(address _spender, uint256 _value, uint _extraData)
        public
        returns (bool success) {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, this, _extraData);
            return true;
        }
    }
```

Which triggers the following function in the RoomAuction:

```
function receiveApproval(address _sender, uint256 _value, ERC20 _tokenContract,
      uint _roomSlot) external {
        // Check if right token
        require(_tokenContract == ERC20(tracker_0x_address));
        // transfers the token to the smart contract
        require(ERC20(tracker_0x_address).transferFrom(_sender, address(this),
_value));
        balance += _value;
        inputBid(_sender, _roomSlot, _value);
      }
```


Rooms assigned
room slot - 256 bits unsigned integer
101
No Rooms have been assigned to you
0
0
Highest bid
room slot - 256 bits unsigned integer
101
2
Account 2
1527613753 (3 minutes ago)


Rooms assigned
room slot - 256 bits unsigned integer
102
You have been assigned
1
2
Highest bid
room slot - 256 bits unsigned integer
101
2
Account 2
1527613753 (3 minutes ago)

Rooms assigned takes the address of the person asking and a room number and returns the room number and the time slot if he was affected the room in question. *whichRoom* works on Remix, but not on Rinkeby, for some reason. So, the relation becomes a triangle relation: the user interacts with the token smart contract, which then interacts with the *RoomAuction*, which then draws the tokens from the user. The address of working Smart Contract is:

```
0xA15A7AFE0A1875Ff06d384382d20c6f7deB49Bc1
```

III. Further developments

For now, it is quite complicated to organize, and the bidding process requires the user to understand and know correctly how the resource is identified: for instance, the user has to know that booking room 1 at the third time slot requires him to type “103” and the amount he wants to bid on this room.

A good way to improve the system would be to provide a web interface for users to see in more details the bidding process and how much they should put on the room at a given timeslot to get the room. If they run an Ethereum node on their computer or use Metamask, they could even place the bids using the web interface instead of the Ethereum wallet. An interface could also make the process of bidding on several slots in a row easier. It could be implemented such that one can select a longer time span for a certain room and the total amount of H$G one wants to spend and then propose how to split up the amount of tokens per time slot, i.e. by splitting the total amount evenly or based on previous experience on how many tokens are necessary for each slot.

Also, one could adapt the system such that left-over slots are allocated in a second round, e.g. by having auctions for the left-over slots that run until one hour before the beginning of the respective time slot. Another possibility would be to automatically allocate students that did not get their preferred combinations to different rooms during the same time that have not been allocated in the auction.

After several rounds of the auction, one could adapt the total amount of H$G in the system such that the room allocation system works efficiently. The goal is that there are neither too many nor too few H$G in the system: Having too many of them does not incentivize students to book rooms only if they need them. Having too few H$G would imply that at the end of the semester, most slots would remain empty since nobody has H$G left to bid on them.

IV. Custom Token

The most prominent token standard on the Ethereum blockchain is the ERC20 Token standard. Creating custom tokens is fairly easy and standardized. However, certain functions need to be implemented inside the token contract for it to interact with different smart contracts. One can change the parameters in the first few lines. In our case the Token is called H$GCOIN with H$G as the symbol and a total supply of 1 Mio. Tokens. For the full token contract see *HSGToken.sol*. The code follows to a large extend the ERC20 standard (Etherium Foundation, n.d). The token's address is as follows:

```
0xaca6f10e07526a128003b310c4b2c5187c9d8d35
```

V. Airdrop Tokens

An airdrop is a distribution of Tokens to designated addresses. Our Airdrop is executed by means of a smart contract. Since Ethereum Wallet is still not fully developed and has issues compiling and deploying more sophisticated contracts, we use "truffle" to deploy our AirDrop contract. For this purpose we follow the instructions on https://github.com/IoTChainCode/TokenAirdrop (IoTChain Project(ITC), n.d.) This solution allows us to efficiently compile the Airdrop contract, deploy it and execute it. We can also easily add addresses into an Excel file and distribute a fixed amount of the Token to each of them.

Firstly, we have to download and install the necessary programs and packages, on a MAC in terminal:

```
npm install -g truffle
truffle version
root npm install -g truffle
npm install -g truffle
git
```

Then, download Xcode from AppStore and install:

```
sudo xcodebuild -license accept
```

Next, we can download the TokenAirdrop folder from https://github.com/IoTChainCode/TokenAirdrop:

```
mkdir airdrop
cd airdrop/
git clone/usr/bin/ruby -e "$(curl –fsSL
https://rawgithubusercontent.com/Homebrew/install/master/install)"
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
cd airdrop/
git clone https://github.com/IoTChainCode/TokenAirdrop.git
```

To access the Rinkeby Test Net, we can either install "geth" or open the "Ethereum Wallet" and keep on working inside Terminal. To install "geth":

```
brew install geth
geth --h
geth --rinkeby
geth --datadir=$HOME/.rinkeby attach ipc:$HOME/Library/Ethereum/rinkeby/geth.ipc
console
geth --rinkeby&
geth --datadir=$HOME/.rinkeby attach ipc:$HOME/Library/Ethereum/rinkeby/geth.ipc
console
```

```
geth --rinkeby&
pgrep geth
```

Moreover, we need to install “npm” in order to let the compiling work:

```
cd airdrop
cd function/
npm install
```

We should have set up everything to compile, deploy and run the AirDrop contract. Now we need to make a few changes to the files to make the AirDrop work the way we want in the lines 50 to 62 of the config.js file:

1. In the lines “userPrivateKey” we insert the private key of the account we want to deploy the contract with and send the tokens from (lines 50, 57, 64).
2. In line 51 “ercAirDropAmount” we can define the amount of tokens we want to drop to each address.
3. By copying our Token Contract Address into the line “tokenContractAddress” we define the Token to be airdropped (lines 53, 60).
4. In line 54 “transferFromAddress” we change the sender of the tokens
5. In line 58 “amount” we define the maximum allowance for the contract to access our tokens from our account.

The only lines missing are 52 and 59 “airdropContractAddress”. However, we first need to compile and deploy the AirDrop contract. We change directory to function and run deploy.js:

```
cd function/
node deploy.js
```

Once successfully deployed, we will receive a contract address. We can also check on https://rinkeby.etherscan.io if the contract was deployed successfully. This address has to be copied into the config.js file in the lines 52 and 59 “airdropContractAddress”. One can check on Etherscan again to see how the contract has been used so far. In our case, the contract address is:

```
0x5A8590e89f6E134aa5bF20d515d3e770A6B0037a
```

Next, we need to tell our account that it should allow the AirDrop contract to access its tokens. We already defined the maximum allowance above and can simply run approve.js:

```
node approve.js
```

As soon as we have received confirmation of the transaction, we can finally execute the AirDrop. Before, we can add addresses to the Excel file itc_airdrop_total.xlsx. We need to make sure that there are enough rows (about 18) and that all columns are filled out accordingly. If there’s not as many addresses, we can simply copy the rows. To execute the AirDrop contract, we need to exit the function/ directory and run normal_airdrop.js:

```
cd ..
node normal_airdrop.js
```

Note that we can execute the airdrop again through normal_airdrop.js without the previous steps, since the AirDrop has already been deployed and can be used for any Tokens with the necessary approveAndCall function integrated. In between we can change the amounts to be sent and add more addresses to the list. However, if we want to airdrop a different Token, we need to approve the contract to the account again.

VII. References

Ethereum Foundation (n.d.). Retrieved on 25.05.2018 from: https://ethereum.org/token

Puppeth (2017). Rinkeby. Retrieved on 25.05.2018 from: https://www.rinkeby.io/#stats

IoTChain Project (ITC) (n.d). Retrieved on 25.05.2018 from: https://github.com/IoTChainCode