einz

Distributed Systems – Project Proposal

Clemens Bachmann	Josua Cantieni	Fabian Gessler
13-932-488	15-919-038	15-939-341
baclemen@student.ethz.ch	josuac@student.ethz.ch	fgessler@student.ethz.ch
Christian Knieling	Eric Mink	Silvia Siegrist
14-923-809	15-917-057	15-935-893
knielinc@student.ethz.ch	minker@student.ethz.ch	sisilvia@student.ethz.ch

ABSTRACT

We chose to create an android application which allows to play the game "einz" which is very similar to the popular UNO cardgame. The goal is to be able to play this game with friends wherever you are and with the rules you are used to, as long as you have an android smartphone and access to the same local area network.

For this purpose we will create an android application which is able to take the role of server and client at the same time. The device of one of the players is used as the server for the game which saves the state of the game and is responsible for synchronization. In this way, there is no extra server needed.

1. INTRODUCTION

We build a distributed game similar to the known card game "UNO" by Mattel [7]. Because you might often find yourself wanting to play a game with friends - e.g. while you are waiting for the next train - but without a set of cards to play it, therefore it would be useful to always carry the cards on you. We make this easy by implementing a similar game on the phone as a native application.

Obvious difficulties awaiting us include the coordination of a team consisting of six people, each with different skill-sets and time available. Also, we intend to create an easily extensible codebase so that we can first build the base game and in a second phase add further rules without much effort. This poses difficulties on its own as we have to learn coding patterns such as using factories to make the program code more modular.

Technical problems will probably be the smooth and clean communication between the clients and the server and implementing concurrency within the server itself. For the networking difficulties like message ordering and making sure the peers actually get the messages, we will rely on TCP using the `Socket` class [1]. There remains the problem of noticing if a client loses connection unexpectedly, especially because we might have times where we don't need to send messages at all for multiple seconds for example if it's not a players turn. We have to experiment with that problem and probably add a ping message that checks that the connection is still alive.

2. SYSTEM OVERVIEW

We propose a modular approach, building first the baseline functionality of the game, followed by further improvements like additional rules for the game. The baseline functionality of the game is a playable version of UNO where only the most basic cards are featured. Further cards will be implemented as rules, e.g. the rule "We feature a +2 card". The additional rules will be available to the user of the device the server runs on before the game starts, such that we can dynamically change the way the game works. This does not mean that we intend to allow the user to define their own rules - just that they can choose which rules should be applied.

2.1 Server Client Messaging

```
{
  "header":{
    "messagegroup":"registration",
    "messagetype":"Register"
  },
  "body":{
    "username":"roger",
    "role":"player"
  }
}
```

Figure 1: Example: Request to register user

First of all, we implement a simple server-client setup and define the format and the messages that should be provided as an interface between the client and the server.(See our first [2] and second [3] draft) At the same time, we can start implementing the user interface and writing this proposal.

See Figure 1 for an example of how our messages will be structured. We encode all our messages in JSON and split them into a header, which is uniform over all messages, and a body, which is message-specific in content. To parse these messages, we implement factories to make the code modular.

That is, the `ParserFactory` use the `messagegroup` to create a `Parser` object specific to the kind of message. Which `Parser` to choose is decided via a dynamically registered mapping, which means that we can reuse the code on both the server and the client side, and that we have minimal effort if we need to change or add some type of message.

For example, say that we need to add a timestamp to the `Register` message for some reason. Chances are that we also need to update the format of the response to this request, which for this reason has the same `messagegroup`. All of this can be done by changing only the `Parser` for this specific messagegroup.

Once the message is parsed by the `Parser` we use an `ActionFactory` and the `messagetype` to actually turn the specific message into an executable method.

If you're interested in the most current specifications of the messages, see our github page [4]. There will at some point also be specified how we handle loss of connection, probably using our own implementation of keep-alive packets.

In a second step, once the basic communication between server and client is completed and the exact behaviour of the two parts has been defined, we finish implementing the server-side game logic and start using the previously defined messaging functionality.

We plan that the server does most of the computations and only sends the client its state, containing the players hand, the number of cards other players are holding and the most recently played cards. The client receives also a list of possible actions that he is able to do. For example the server informs the client which cards are playable. The client can now choose an action from the list and send a request to the server afterwards. The server will handle the request and check if it is a valid request. If it is he will inform all clients about the updated state.

Sending the actions to the client allows us later on to add game-modes and rules with ease and grants us high flexibility with using the rules. We only have to change a few files to add new content - ideally just one.

We are also trying to keep the classes themselves very modular to keep updating them - e.g. by adding further rules - just as simple.

2.2 User Interface

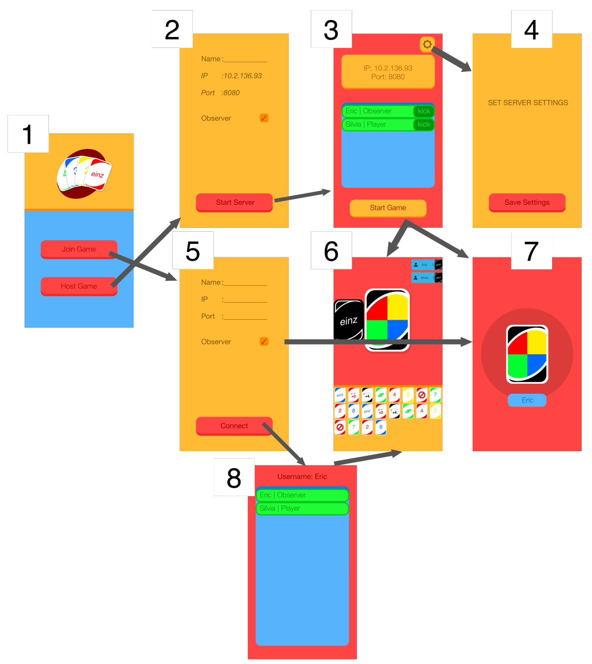

Figure 2: User Interface

The user-interface for the standard scenario from starting the app to start playing looks like the following: The first screen (1) is for the player to choose, whether he wants to start a new game or join an already existing game. If he decides to start a new one, he will start the server in a thread in the background. He will be prompted to enter his username and decide whether he wants to play or observe/spectate the game. If he presses the "Start Server" Button in (2) after filling out the missing information, he will connect himself as an administrator to the previously started server running in the background of his device. After that he will be forwarded to an activity (3), with the necessary information for others to connect and a list of already connected players/observers. He can kick them if he decides to do so. By pressing the settings button he will be taken to (4) and there he can specify the rules for the game.

A client that joins a game will go through a similar process. He has to start by filling out his name, the host IP and port and whether he wants to act as a player or a spectator (5). Once the player connects to the server he will see a list of other players/observers, but not be able to kick them or change any settings, as this is left for the host/admin (8).

When the host starts the game, everyone will see the activity in which the game itself will take place - either a standard view with his hand, the topmost card, all connected players and the amount of cards in each hand (6) or the view of the spectator (7). Players will be able to play any card that is declared as playable by the server or draw cards from the heap. A player arbitrarily leaving the game by pressing the back button is also an action that we take into consideration.

Spectators will see the public game-state, but not any players hand cards. Also they will not be able to interact with the game in any way. The use-case for this option is that you could place a - possibly larger - android device (tabled, phablet) in the middle of a table to get the "classic cardgame feel" with a physical representation of the stack of discarded cards while still playing on mobile devices.

2.3 Dynamic Rules

Since every group plays UNO with different rules we want to provide them with the possibility to play by the rules they are used to. We will feature a setting-screen for the admin where he can specify the rules for the next game. These will mostly be checkboxes or multiple-choice dropdown menus. This means that we will not feature user-designed rules. However, we intend to make use of the efforts we put into the extensibility of our codebase by adding the ideas we had during our first meeting [6] if we have enough time to do so.

2.4 Organizing

Working in a larger team and managing it is new to us. We try to help ourselves by using git with github [4] and trello [5] to manage the tasks. We have also appointed a Team Lead whose task it is to distribute the work so that everybody can contribute meaningfully to the Project yet also learn something while doing so.

2.5 Additional Possible Goals

Once the game is in a working state with a few rules such as an additional card like the one where the player can choose a color or the well-known "+2" card from UNO, we will decide whether we need to focus on bug-fixing, cleaning up code, implementing a lookup-server (LUS) for NAT-traversal (See Figure 3) or adding more rule options.

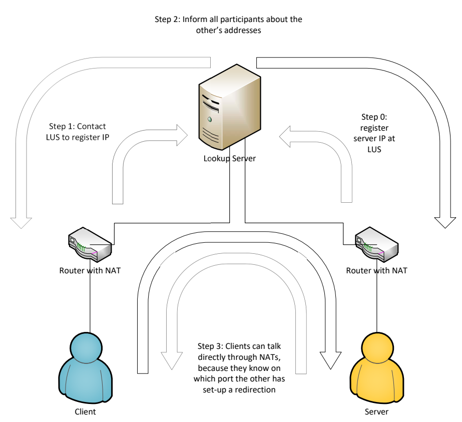

Figure 3: NAT-traversal

If we find enough time after adding enough rules to make the game interesting - we probably will not - and decide to do NAT traversal, we would set up a Lookup-server that is reachable for all clients and thus allow the routers to setup forwarding on ports that we know. The LUS can then inform every client about the servers's IP address and ports, through which they can communicate the same way as we implemented in the first phase (And vice-versa. See Figure 3). The gameserver on one phone will have the option to choose whether to use the LUS or only accept players from within the same LAN.

Adding a LUS might impose additional difficulties because some mobile carriers might use symmetric NATs. [8]If these difficulties arise, we will probably resort to only implementing NAT-traversal for the other (easier) types of NATs, if any.

3. REQUIREMENTS

The app will run on Android devices that run at least Android 5.0, which corresponds to the API-level 21. We use the local wireless connection to communicate between the devices. Thus we need the permissions `INTERNET` and `ACCESS_WIFI_STATE`.

Since this is a multi-player game we also need more than one device that are able to communicate through the local WiFi with each other for the game to run.

4. WORK PACKAGES

Our project broken down into subtasks:

- **WP1**: Define Client-Server Communication
  We define the protocol that the client and the server use to communicate. Once we agreed on a message exchange protocol, the development of the server and the client can be done separately.
- **WP2**: Design and implement Game Model
  Implement the data-structure and objects used by both client and server. This means classes like `GameState`, `Card`and `Rule`.
- **WP3**: Server Game Logic
  Implement the game logic without thinking about the networking part or multi-threading, apart from clearly denoting which functionalities are thread-safe.
- **WP4**: Dynamic Message Parser
  Implement a reusable parsing system that maps incoming messages to executable methods as described in the System Overview. WP5 and WP6 will reuse this code.
- **WP5**: Client Communication Backend
  Implement message parsing and sending on the client side.
- **WP6**: Server Backend
  Implement message parsing on the server side.
- **WP7**: Implement Client Server Protocol
  Register parsers and actions for every type of message specified. The server could implement this in multiple threads and has thus to call the actions in a safe way.
- **WP8**: Connect Server-logic to Messaging
  Messages should be parsed and the corresponding action should be executed. The logic calls back to the messaging part to inform the clients about updates in the game-state.
- **WP9**: Application UI
  Make a UI like previously described for all the different Activities. Add resources for them to all have the same style.
- **WP10**: Polish the Game and Bugfixes
  Optimize the user experiences and fix all those nasty bugs.
- **WP11**: Implement Additional Rules:
  Rules to make each game individual.
- **Optinal WP**: If there is time implement additional goals
  Only if we really have the time. We can implement the ideas we had in the section "Additional Possible Goals"

5. MILESTONES

5.1 Milestone 1: Specification

Specify the interfaces and get ready for implementation

- WP1 - Eric, Fabian, Josua
- WP2 - Josua

Due: 19.11.2017

5.2 Milestone 2: Fundamental

Laying the foundation for the client-server messaging on both the client and the server

- WP4 - Eric
- WP5 - Christian, Clemens
- WP6 - Eric, Silvia

Due: 26.11.2017

5.3 Milestone 3: Messaging

Implement and test the previous specified messaging protocol

- WP7 - Eric, Silvia

Due: 30.11.2017

5.4 Milestone 4: Base Game

Getting the game to work in a basic fashion

- WP3 - Fabian
- WP8 - Eric , Fabian
- WP9 - Christian

Due: 7.12.2017

5.5 Milestone 5: Final Game

Implement additional rules and test and polish the game.

- WP10 - Everyone
- WP11 - Everyone

Due: 17.12.2017

Before we start to code, we first write a specification for the interfaces to prevent a big mess. For the implementation we see two areas where we assign our team members to: the server and the client. Also someone is designated as tech-lead and keeps an overview about the entire process.

Responsible for the server are Eric, Fabian and Silva.
Responsible for the client are Christian and Clemens.
Josua reigns over order and chaos of the whole project.

6. REFERENCES

[1] Android documentation: Socket. https://developer.android.com/reference/java/net/Socket.html. Accessed on 16 Nov 2017.

[2] documentation_Messages.md, Early Draft of JSON Interface Documentation. https://github.com/lucidBrot/einz/blob/5b142dc53962acd63335dd5f38f38d6bd24a4d74/protocols/documentation_Messages.md. Accessed on 13 Nov 2017.

[3] documentation_Messages.md, Second Draft of JSON Messaging Interface Documentation. https://github.com/lucidBrot/einz/blob/b67a627214d3d903fe54848b5be9bc4188387375/protocols/CommunicationModel/messages.md. Accessed on 16 Nov 2017.

[4] Einz git repository. https://github.com/lucidBrot/einz. Accessed on 13 Nov 2017.

[5] Our trello dashboard. https://trello.com/b/4FOOdNPI/distributed-systems. Accessed on 13 Nov 2017.

[6] Protocol of our meeting on 06.11.2017. https://github.com/lucidBrot/einz/blob/5b142dc53962acd63335dd5f38f38d6bd24a4d74/protocols/protocol20171106.md. Accessed on 16 Nov 2017.

[7] "UNO" seller. http://shop.mattel.com/shop/en-us/ms/uno-game. Accessed on 13 Nov 2017.

[8] Wikipedia: Network Address Translation. https://en.wikipedia.org/wiki/Network_address_translation#Methods_of_translation. Accessed on 17 Nov 2017.