Design and Implementation of a GameCenter for opportunistic networks

Group Thesis

Renzo Andri
Sebastian Flückiger
Patrick Leu

October 17, 2013

Advisors: **Sacha Trifunovic**
Supervisor: **Prof. Dr. Bernhard Plattner**

Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

This thesis focus lies on the conceptual design and implementation of a Game Center for Android™ for opportunistic networks.
The Game Center will provide a basic framework for developers to create multiplayer games that make use of a distributed database, highscores and stable communication channels between players within opportunistic networks. To manage and access neighbors the Game Center will rely on the WLANOpp library[1].

Acknowledgments

First of all we want to thank our advisor Sacha Trifunovic who provided us with ideas and shared his comprehensive knowledge in Android™ programming and Professor Bernhard Plattner for giving us the opportunity to write our thesis in his group.
Also we would also like to express our gratitude to the Communication Systems Group which provided us with two Android devices during the course of this thesis and allowed us to use WLANOpp in the thesis.

Contents

Chapter 1

Introduction

Games and other forms of competitions have always been a popular pastime.[2] With the arrival of smartphones a new platform for playful interaction has appeared. Looking at statistics from app market places such as the Google Play Store it becomes obvious that games are the most prominent category of mobile applications.[1] Building multiplayer games for smartphones is therefore the logical step to play on the competitive nature of people.

Creating a multiplayer game requires a lot of effort and infrastructure. Programming stable communication channels over networks is a difficult task and to manage any information such as highscores, servers are needed. This is where opportunistic networks come to play. Allowing for networks without any infrastructure except the devices themselves, they are the ideal platform for simple multiplayer games. Having solved the infrastructure issue with opportunistic networks, the next part is to develop stable communication channels and highscore systems for a game. The focus of this thesis lies in providing such functionality for future game developers.

In this thesis we will create a framework to build multiplayer games without worrying about data transmission, highscores or databases by providing them with a powerful yet easy-to-use API to take care of those issues. To set an environment in which future games will take place we base our own framework on WLANOpp. WLANOpp is a library that can create and manage opportunistic WLAN networks on Android™. Within the WLANOpp environment our Game Center will make automatically synchronized highscores available to all devices. Simple API calls will allow developers to store and retrieve highscores and encounters for their game and to transmit data between different players.

[1] On October 12, 2013 the top 65 grossing apps on the Google Play Store were games.[3]

Chapter 2

Related Work

2.1 Opportunistic Networks

As opposed to nodes in a conventional network, participants located in an opportunistic network are not necessarily static. If a communication path can not be provided at a certain moment in time, a node intending to transmit information may store it and wait for future forwarding opportunities. Any adjacent node can be seen as a possible next hop. As routing paths are chosen dynamically, individual nodes do not need any knowledge of the network topology.[4] This concept is especially of use in delay-tolerant applications when facing an outage or lack of networking infrastructure.[5] There is a variety of reasons conventional network communication may be unreliable, be it due to natural disasters or government censorship.

The thesis at hand aims at using this promising, yet challenging approach for an application most smartphone users can relate to: Playing games against networking peers. The Game Center enables the user to challenge other people in the same WiFi and exploits the dynamic nature of opportunistic networks by disseminating user highscores every time a new network is entered.

Further, this project is also an attempt to popularize applications of opportunistic networking in order to direct the attention of smartphone manufacturers to this area. Once the necessary APIs enabling WiFi Ad-Hoc connections are provided on popular platforms, many fruitful applications thereof will be within reach of application developers.

2.2 WLANOpp

WLANOpp is a project of the Communication Systems Group which handles (ad-hoc) opportunistic networks for Android™ devices. Starting from the problem that most smartphones do not support WiFi Ad-Hoc and this fact probably won't change in the near future, WLANOpp builds up ad-hoc networks using the tethering mode to build up a mobile access point AP instead of a stationary one. [1]

The main use in this project is the observation of Neighbors and their IP in the same network. But if there is a lack of network connection, WLANOpp would be able to build up its own network.

Chapter 3

Design

The Game Center project can be grouped into five different parts, namely the WLANOpp-Service, GameCenterApp, GameCenterService, GameCenter-API and related Games. These parts and their interplay can be seen in fig. 3.1 and are described in the following sections.

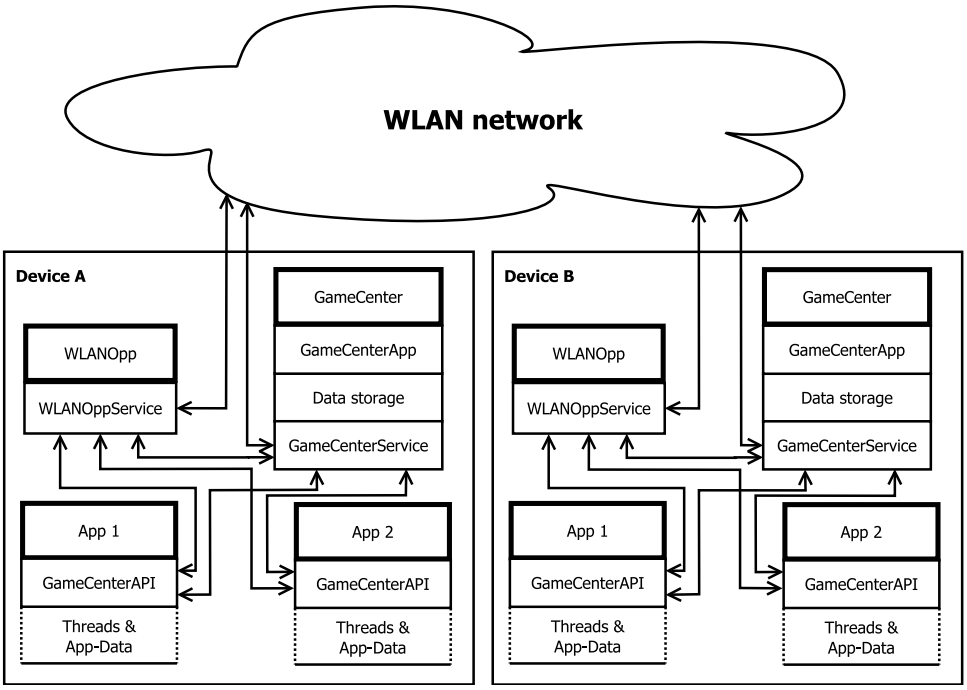

Figure 3.1: Overall design view

3.1 GameCenterApp

The GameCenterApp has two main tasks. First it allows the user to get his statistics of every installed GameCenter Game. These statistics contain the user's overall highscores, his encounters, a comparison to his past adversaries and the highscore of all other users met. (Met in the sense of seen at least once in the same network)

Secondly it is responsible for the overall data storage inlcuding the mentioned statistics, game data (game identifier, game name, highscore type), Nicname (mapping such that an abstraction from IP address, MAC address or some Hash is possible) and timestamps (allowing for resource saving during synchronization).

3.2 GameCenterService

The GameCenterService builds up the core of this thesis. This service is part of the GameCenterApp, but runs independently from it. Games bind themselves and connect to the GameCenterService. The GameCenterService organizes all the data processing over the WLAN network and provides the connected games with networking services like data sending and data receiving. Additionally it takes care of the synchronization of results and highscores.

3.3 GameCenterAPI

The main goal of the GameCenterAPI is to provide an abstraction of network communication to prospective game developers. This abstraction contains simple detection of available co-players, easy data processing without the need to care about IP addresses, sockets and so on. The API also suppllies the developer with simple methods to save and get higscores in the GameCenterApp.

3.4 Related Games

The GameCenter framework provides the game developer an easy way to create multi-player games without investing time in network programming and data storage. Realizable are all two-player games which are based on a Win-(Draw)-Loss or highscore system or a combination of them. Also general multiplayer games are possible, but the GameCenter doesn't provide encounter statistics for more than two players.

Chapter 4

Implementation

4.1 GameCenterApp

4.1.1 Data Management

The data collected by games is stored within five different SQLite databases which are all accessed through a content provider. All databases and the relations between entries are depicted in fig. 4.3. The keys most commonly used for queries are the Game ID entry in the table *Games* and the Player ID entry in the table *Players*. With these two identifiers, it is possible to get data associated with a certain game and a certain opponent, e.g. the result of your last match in *TicTacToe* against player *John* or your overall performance in *Hangman* against player *Carl*.

Since data needs to be accessible through different apps, a content provider is used. Access to databases is provided through the ContentResolver interface as depicted in fig. 4.1. The respective database is specified by the according URI in each query. The necessary IPC-mechanisms to access databases from different applications are handled by different ContentResolver instances and the ContentProvider in the background without any programmer involvement. Future application programmers are free to use the basic methods of the ContentResolver interface - *insert()*, *update()*, *query()* and *delete()* - or the methods contained in the *ProviderHelper* class which allow for easy adding, updating and retrieval of player statistics.

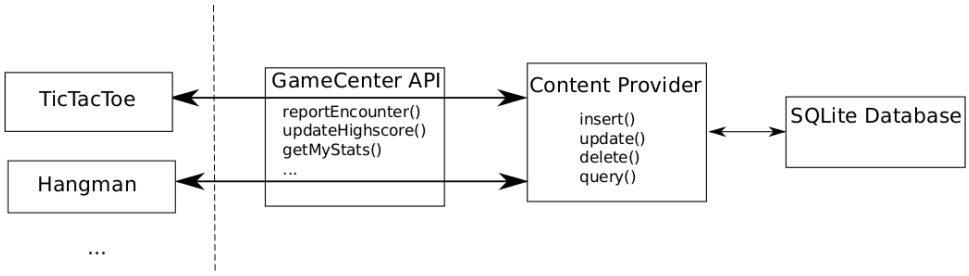

Figure 4.1: Content Model

4.1.2 GameCenter

Besides data management, a central job of the GameCenter is to provide the user with the latest data. Regardless of highscores being continously updated, the entries always have to be up-to-date.

Program Flow

The program flow used in the *GameCenterApp* is depicted in fig 4.2. Building blocks are the two activities drawn as black rectangles and the four fragments used as parts of the *GameStatsActivity* represented by red rectangles. The arrows show the behaviour of the application on interaction with list elements or buttons located in the user interface. However, not explicitly shown is the possibility to use the back button to return from *GameStats* to *GameOverview*. Further, on first startup a dialog will appear, requesting the user to enter his nicname. Once set, this nicname can not be reset and the dialog will not appear anymore. *GameOverview* shows a list of all the games registered on this device. The user can tap on any of the columns to see more information about a game. *GameStats* shows information about the user's score in its first fragment *MyStats*. Swiping sidewards on the screen allows for switching between different fragments showing statistical information to recent encounters *(Encounters)*, comparisons with other players *(Adversaries)* and highscores of all known players *(Highscores)*. All this information is provided subject to the game chosen in *GameOverview*. Visible in each of the four fragments, a button located in the title bar will start the respective game on interaction. This specific behaviour is depicted on the right hand side of the dashed line in fig. 4.2.

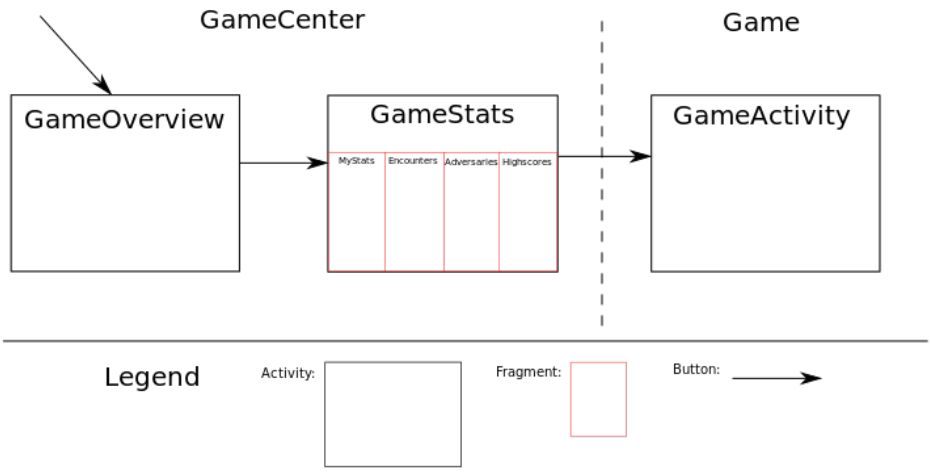

Figure 4.2: Program Flow

Keeping the UI up-to-date - Loaders
In order to guarantee the freshness of lists contained in the user interface, loaders are used to perform the asynchronous loading of data. Each fragment containing lists implements a LoaderManager interface which manages loaders. The Loaders are fed with a description of the columns in the database to observe. Whenever changes occur in these columns a callback will map changed database entries to pre-specified parts of the user interface.

Different GameTypes
Since there are not only games where you can either win, lose or play a draw, but also ones where you achieve a certain score and compare it to your adversary in order to define a winner, different *GameTypes* are allowed. In order to tell them apart and choose the appropriate user interface, a constant tells the Fragment which ListView to choose. This constant is sent via intent from the *GameOverviewActivity* to the *GameStatsActivity*. Subject to this constant the UI-mappings for the loaders are set to the appropriate ListView.

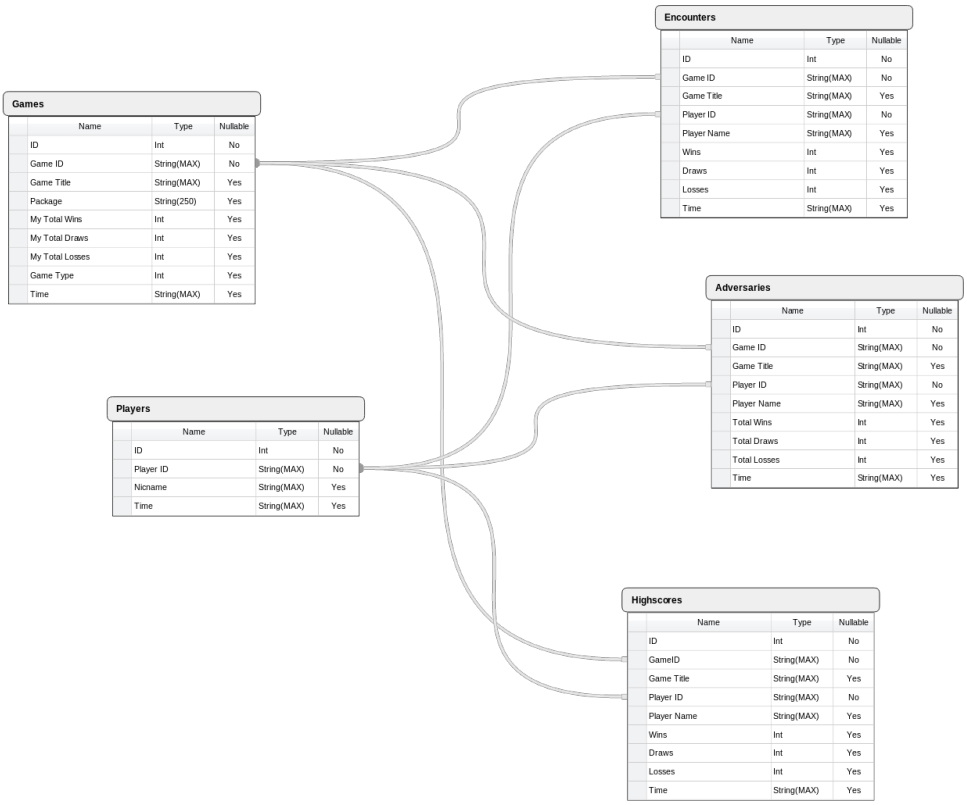

Figure 4.3: Database Overview

4.2 GameCenterService

For the GameCenterService we need the Android™component *Service*. This component allows us to do all the networking in one external and common process. All apps bind to the GameCenterService so that they can transmit data to the service. To allow for an external starting and binding, the *Service* has to be exported and need an intent-filter. This intent-filter is needed to identify the service by external applications. Both have to be defined in the Manifest[1].

4.2.1 Networking

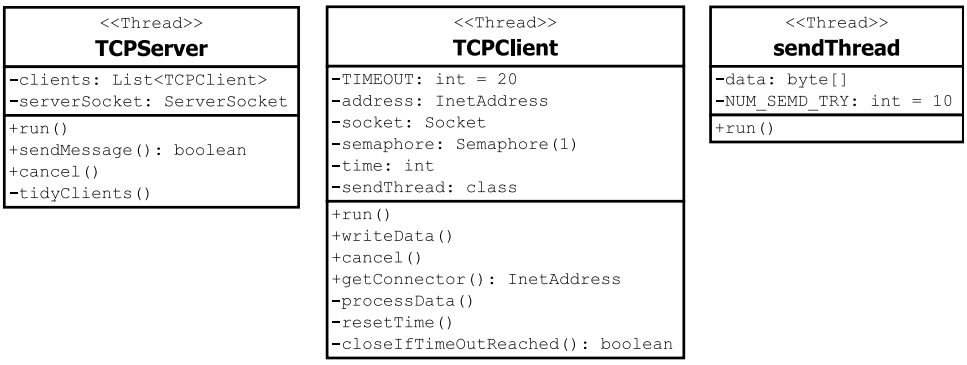

Figure 4.4: Networking Threads

The system is built up on the Transmission Control Protocol TCP. The main advantages of this protocol are: that it is connection-oriented versus the User Datagram Protocol UDP. Data transmission is guaranteed to happen in right order and lost packets are retransmitted. Additionally the TCP's Congestion Control avoids the device from overloading the network. [6]

Networking is done using sockets[2]. Android™ differentiates three types of Sockets: [7]

1. DatagramSocket: Socket based on UDP. Main functions are *receive* (blocking) and *send*.

2. ServerSocket: Socket based on TCP. Aims to wait for connecting clients (*accept*) and establish connections. ServerSocket is not able to connect or send data to other clients directly.

3. Socket: Client-side TCP socket. Its main functions are *connect* to establish connection to a ServerSocket, *getInputStream* and *getOutputStream*.

[1]Android™applications need a Manifest file. In this file all relevant information such as package name, versions, permissions, and properties of application, services and content provider (data storage) are defined.

[2]Sockets is an API (Application Programming Interfaces) provided by the operating system to allow programmers to use the network interface. Detailled informations on socket programming can be found in [6, Ch. 1.4].

The last two functions return streams which allow writing and reading to and from a socket.

Every socket must be implemented in an own thread since IO functions waiting on clients block the thread. For this reason there is always an open *ServerSocket* in one unique TCPServer thread listening to new connecting clients and one TCPClient thread containing an open *Socket* for each established connection. fig. 4.4 illustrates the class declarations.

TCPServer Thread
The main class for networking in the GameCenterService is the TCPServer Thread. The main task of TCPServer is to allow external clients to set up a connection to the device using the ServerSocket and to simplify packet sending from the device to other ones.
The ServerSocket listens on a unique port[3] for connecting clients. If there is an established connection the TCPServer starts a TCPClient thread to handle data exchange to the connected client and adds this TCPClient to the list of *clients*.
TCPServer also provides the *sendMessage(receiver, byte_array)* function which delivers (byte)[4] data to the corresponding TCPClient. If there is no corresponding TCPClient, a new connection is established.

TCPClient Thread
The TCPClient Thread handles the data exchange of an established TCP connection. The UML state diagram in fig. 4.5 a) shows an overview on the functionality of the TCPClient Thread. After the socket is instantiated a DataInputStream is created using socket's InputStream. Packets are then read from DataInputStream and passed to the function *processData* which transmit them to the destination application (Details in Ch. 4.2.2). Data transmission occurs in two steps. First the package length is received so the buffer size can be set to the according data size. Secondly the received data is saved in the buffer.

Data can be sent to the client using the function *writeData* which starts an instance of *sendThread*. A new thread is needed since the function automatically runs in the caller's thread and it would therefore block the caller until transmission was successful.[5]

sendThread
sendThread organizes synchronous data transmission. Before transmission is started, it waits until the *TCPClient* has opened a connection to the receiver's device. (See figure 4.5 b) for a more detailled view.) Then it enters a criticial section acquiring TCPClient's *semaphore*.[6] From the socket a *DataOutputStream*

[3]Ports are used in TCP/UDP to identify the according server/client. The operating system can forward data to the addressed program. Ports between 49152 and 65535 can be freely used.[6]

[4]All data is transmitted in binary form. Therefore all data has to be serialized prior sending. Note that objects can only be serialized if the corresponding class *implements Serializable*.

[5]Android™(3.0 and higher) throws a *NetworkOnMainThreadException* to keep the developer from writing network deadlocks.[7]

[6]Semaphores are used to synchronize threads accessing the same memory or stream concurrently.

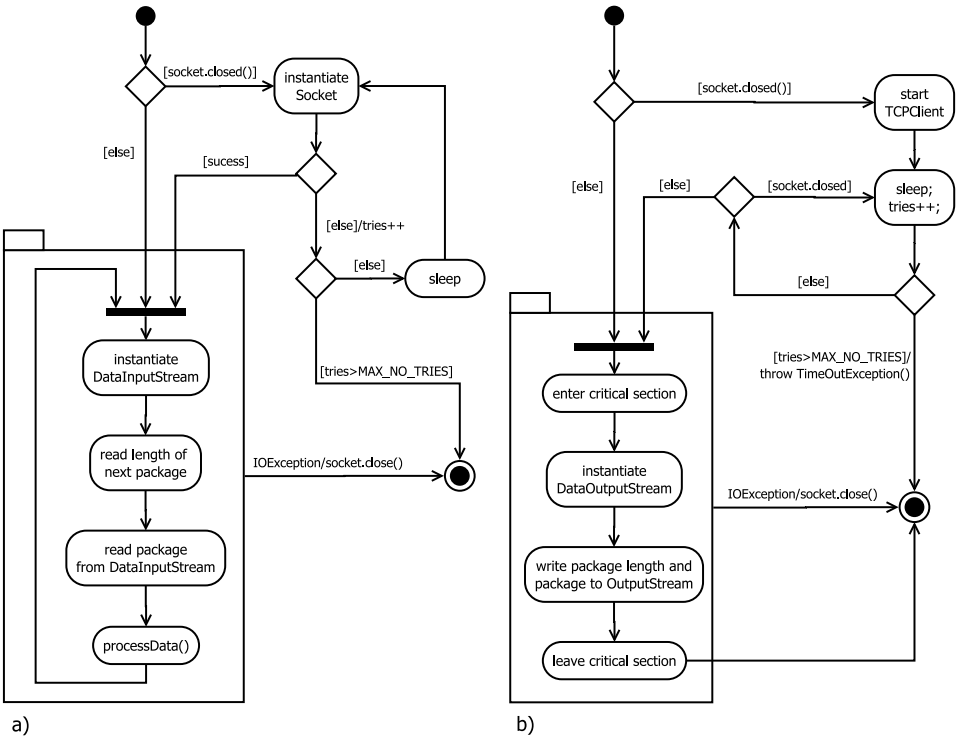

Figure 4.5: UML state diagram of TCPClient Thread a) and sendThread b)

is generated, which isn't synchronized[7], therefore synchronization (e.g. with semaphores) is needed. Otherwise concurrent *sendThread*'s could corrupt data. As described, sending occurs obviously also in two steps. Package size integer followed by the package data in binary form are written on the *DataOutputStream*. Finally *sendThread* leaves critical section and finishes.

4.2.2 Inter Process Communication IPC

Inter Process Communication IPC allows processes to communicate with each other. Android™ provides three methods for IPC[8]:

- Extending the Binder class: Not applicable because service is running in an external thread.
- Android Interface Definition Language AIDL: Allows for multiple calls. Not really needed and leading to more complicated implementations.
- Using Messenger: Concurrent calls are queued and processed sequentially.

Therefore the GameCenterService uses Messenger.
Messenger IPC generally consist of:

Message: Data object which can be transmitted by Messenger.
Handler: Extendable class. The function *handleMessage(Message)* can be overridden, to define how to handle incoming messages.

Messenger: Main messenger class. A handler can be passed while instantiating. Provides function *getBinder* which returns an IBinder which can be used for binding to this messenger.

The service and all application generate their own *Messenger*. When a client binds to the GameCenterService, the service returns an *IBinder* to the client. With this binder the client can generate a *Messenger* to send messages to the service. Using this messenger, all the clients send a message to the GameCenterService containing a registration request. The service then gets a *Messenger* from the received message and save them. Details in Ch. 4.3.2.

4.2.3 Synchronization

Highscore data has to be synchronized automatically. This is done by the following scheme:
If a new neighbor is detected by the GameCenterService it sends a small package containing a timestamp. This timestamp is the last received timestamp by the new neighbor. If this neighbor detects that he has newer highscores, it sends them back with a new timestamp.
If a new highscore is achieved it is sent directly to all neighbors except the adversary.

4.3 GameCenterAPI

4.3.1 Overview

The GameCenterAPI places the GameCenterController at the game developer's disposal. The GameCenterController organizes binding and data transmission to the GameCenterService and the access to GameCenterApp's database (e.g. highscores) and has to be instantiated by every GameCenter game.

GameCenterAPI contains the following classes:

Player: A *Player* is an abstraction of the WLANOpp Neighbor[7]. The *Player* class contains a *Neighbor* and a corresponding Nickname. The *Player* provides the following member functions:

Function	**(return type) explanation**
getNic	(String) get Nickname of player.
getID	(String) get WLANOpp ID of player
getNeighborObject	(Neighbor) get the corresponding WLANOpp Neighbor Object.
equals	(boolean) are player and argument equal
compareTo	(int) returns the lexicographical comparison between *Player*'s and argument's nickname

[7]Neighbor consists of an unique ID determined by device's MAC address and an IP address of IPv4 or/and IPv6 (class Inet{ε,4,6}Address in Android^{TM}).

PlayerObserverTask: Implementable interface which can be passed to GameCenterController. The function *run(List<Player>neighbors)* is called whenever the List of available *Player*s has changed.

DataListener: Extendable class which can be added to the GameCenterController using the *addListener* function. The overridable function *onDataReceived* is called by the GameCenterController if data was received from the network. The constructor allows a Player as argument so only data from this player invokes the *onDataReceived* function.

DataListenerClass: This internal class is instantiated once and contains all registered *DataListener*. GameCenterController calls DataListener's function *fireEvent* which passes data received from the GameCenterService to the according DataListeners.

DataReceivedEvent: Event which is passed to the *onDataReceived* function containing the sender, package data and data type.

Function	**(return type) explanation**
getSender	(Player) get the player package is coming from.
getData	(Serializable) get received data package.
getDataType	(int) get the data type (e.g. an invitation)

GameData: Helper class which contains game name and game ID.

GameCenterController: The GameCenterController is the main class of the GameCenterAPI. Using an instance of it, developer has access to the full functionality of the GameCenter. The following table describes all available functions:

Function	**(return type) explanation**
getInstance	(GameCenterController) returns an instantiated GameCenterController. Arguments are game name, game ID, application context, some handler (needed for binding to WLANOpp) and a facultative PlayerObserverTask.
isGameCenterInstalled	(boolean) whether GameCenterApp including GameCenterService is installed.
isRegistered	(boolean) whether game is registered in the GameCenter.
addGame	() adds the game to the GameCenter.
start	() starts WLANOpp and connects to the GameCenterService.
isConnected	(boolean) returns whether GameCenterService is bound.
stop	Stops WLANOpp and disconnects from GameCenterService.
addDataListener	() adds a *DataListener*.
removeDataListener	() removes a *DataListener*.
getPlayers	(ArrayList<Player>) returns available players.
isPlayerOnline	(boolean) returns whether player is reachable.
send	() sends a data package to a player.
isMyNicSet	(boolean) returns whether the player has already set a nickname in the GameCenterApp.
getMyNic	(String) returns own nickname.
setMyNic	(boolean) sets the nickname of the player and returns false if nickname has already been set.
getOwnStats	(ArrayList<int>) returns the own results. (win or score/draw/losses)
getMyHighscore	(ArrayList<int>) gets the own highscore.
getHighscore	(ArrayList<int>) gets highscore of a player.
getEncounter	(ArrayList<int>) gets encounter against player.
getVersus	(ArrayList<int>) gets sum of all encountera against player.
reportEncounter	(ArrayList<int>) adds a new encounter against player (win or score/(draw)/losses).

4.3.2 Data processing

The main goal of the GameCenterAPI is the easy processing of data from one device to another. The following subsections will shortly describe how this task is solved.

GameCenterController sender side

As described before, the GameCenterController provides the function *send* with the arguments receiver as *Player*, data package as some serializable Object and a facultative data type as integer. fig. 4.6 illustrates this. In a first step the data object is serialized (put in byte array form). Then the relevant data (adversary, byte data, sender's ID and GameData) is saved in a *MessengerData* object which itself is serialized. This byte array is attached to a *Message* object, to which type, replyTo messenger and data type are passed as well. Type is used to inform the receiving side what type of data is sent: e.g. normal data processing (GCS_SEND_DATA) or register requests (GCS_REGISTER_CLIENT). ReplyTo allows the sender's GameCenterService to know to which messenger to pass answer messages. This is mainly used on the register request where the GameCenterService saves active clients. Clients as already mentioned in Ch. 4.2.2 achieve the messenger depending to the GameCenterService during binding.

GameCenterService sender side

The message from GameCenterController is received at the GameCenterService and causes the call of *handleMessage*. Depending on*Message's* type (specified in it's *what* attribute), the according routine is invoked. In this case the *send_data* routine. In a next step the MessengerData is deserialized. With its content the SocketData (Fig 4.7) is instantiated and sent to the active TCPServer by calling sendMessage. The SocketData contains relevant receiver data and the receiver Neighbor object.

[8]These diagrams are inspired by the UML activity diagram, UML class diagram and data dependy graph DDG. UML diagrams are usually split up in behavioural and structural diagrams, so that these diagrams are not completetly conform to UML standard. Although UML2 does allow mixing diagrams.[9]

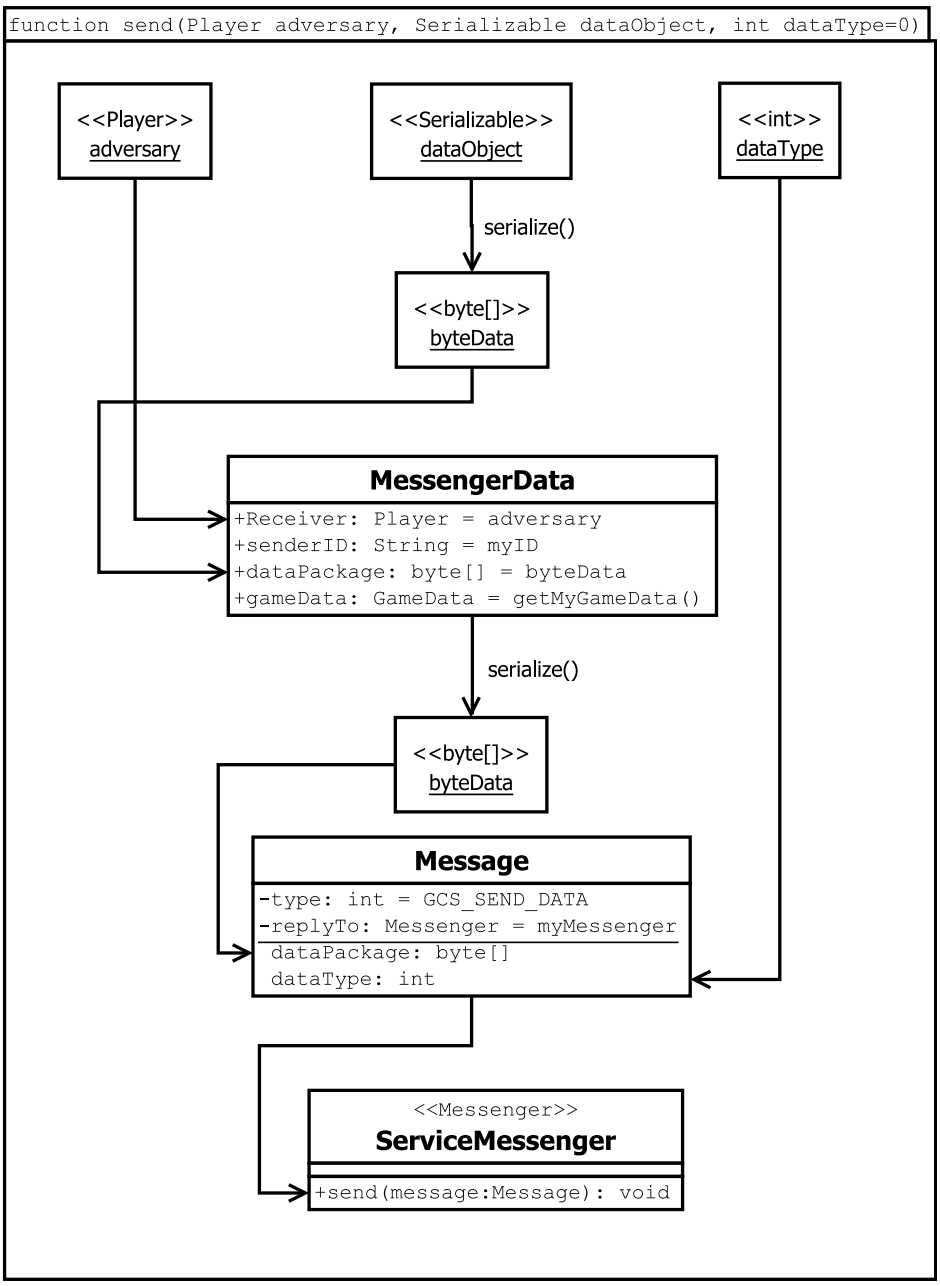

Figure 4.6: Activity diagram: GameCenterController sender side.[8]

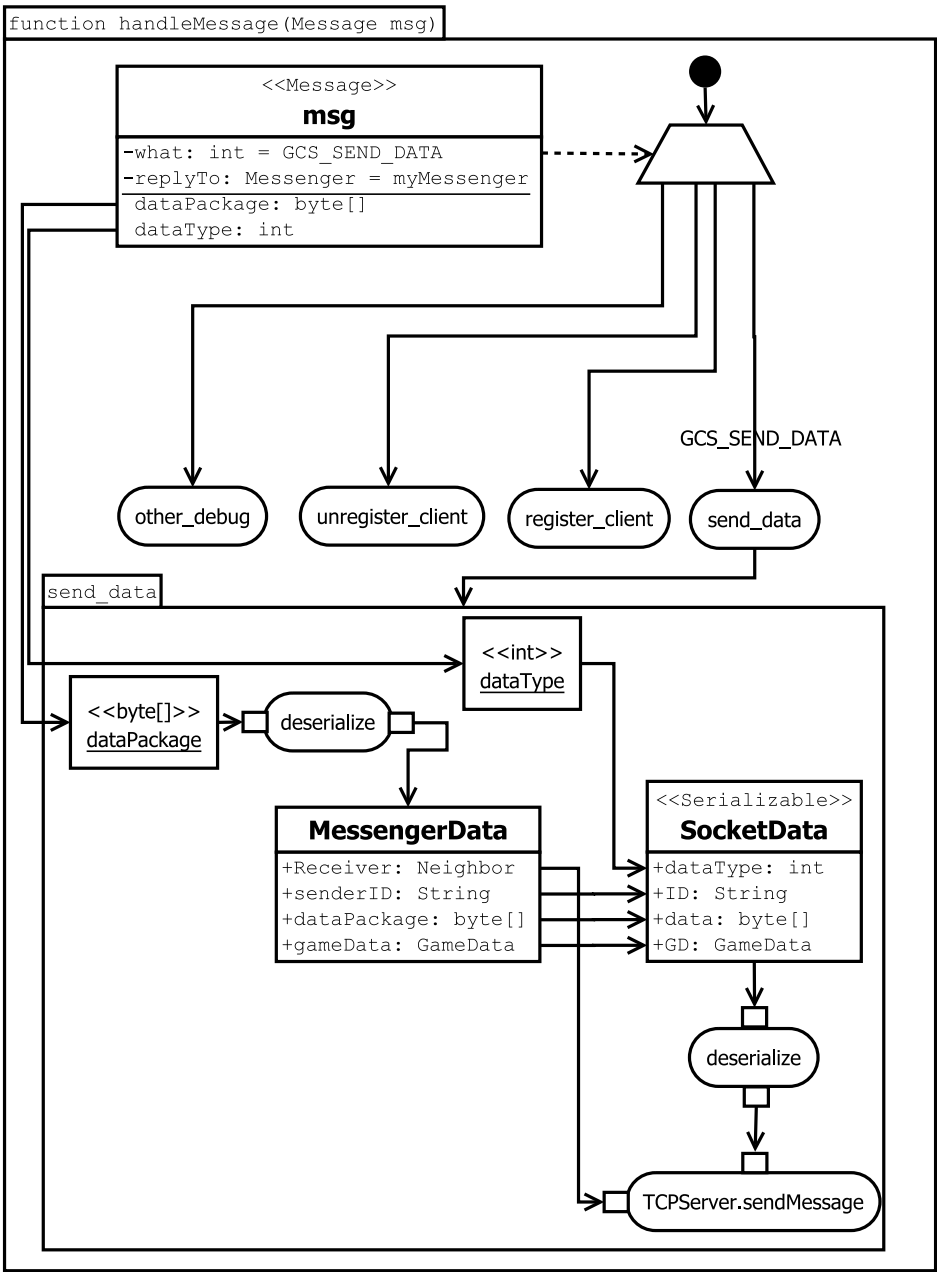

Figure 4.7: Activity diagram: GameCenterService sender side

GameCenterService receiver side

TCPClient is mainly described in Ch. 4.2.1. After having read data from input stream the function processData is invoked which sends the received data to the according game. In fig. 4.8 and in the following lines, a short overview of its functionality is presented:

SocketData is then recovered by serialization of the passed byte array data (data received by TCPClient). After this the data type is checked whether it contains a propriertary GameCenterService value. If this is not the case data has to be transmitted to the right client. From the *Client*s' list we get the right client by passing the GameData to clients' *getClientByGameData* function. If there is no such client, it means this game is (currently) not bound to the GameCenterService. If the data type is equal to zero, it contains an invitation to a game. In this case a Notification[9] is built. Clicking on this notification the user is able to get into the game and may accept the invitation. The data from the *SocketData* are then put into a message and transmitted to the messenger belonging to the *Client*.

[9] A notification is a message you can display to the user outside of your application's UI. When you tell the system to issue a notification, it first appears as an icon in the notification area (on top of the screen). To see the details of the notification, the user opens the notification drawer. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time.[7]

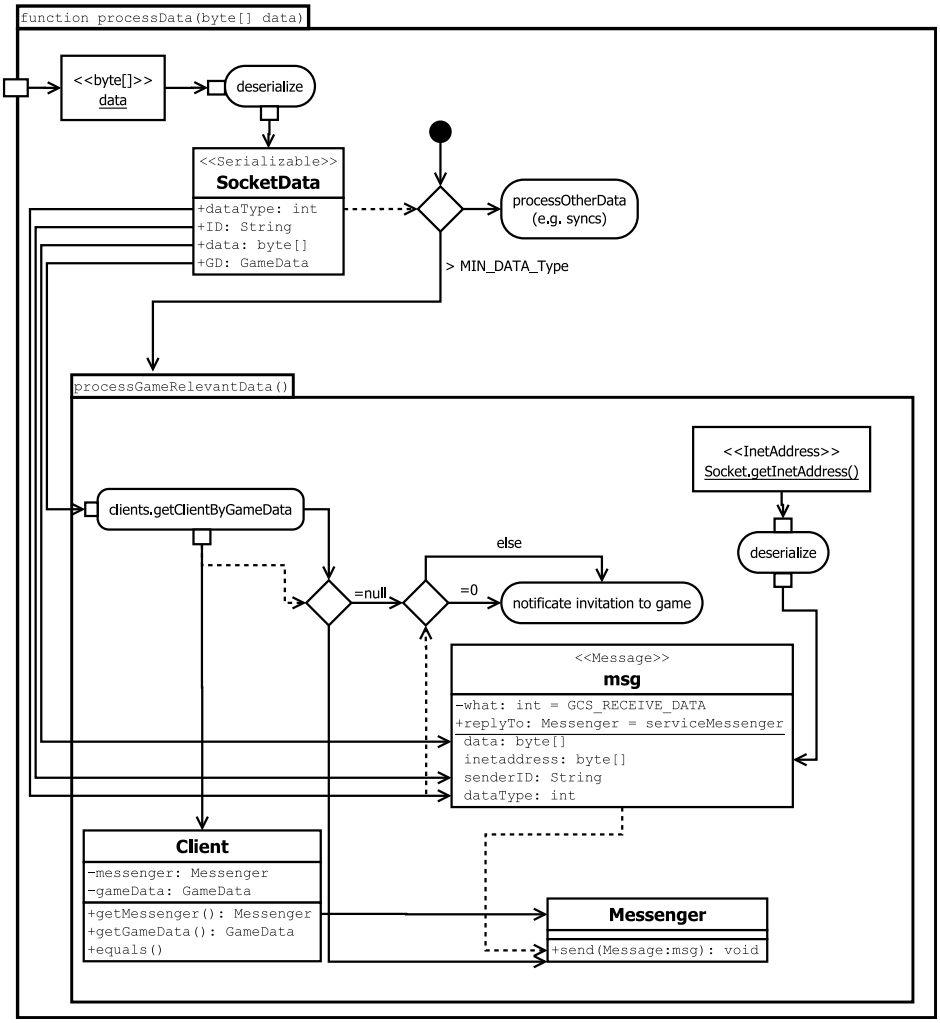

Figure 4.8: Activity diagram: processData function (GameCenterService receiver side)

GameCenterController receiver side

Finally from the Message GameCenterController gets the dataPackage, data type and an according Neighbor using InetAddres and senderID. This is passed to the DataListenerClass' *fireEvent* function. This last step is illustrated in fig. 4.9. Note: The abstraction from Neighbor to Player happens while calling *onDataReceived*.

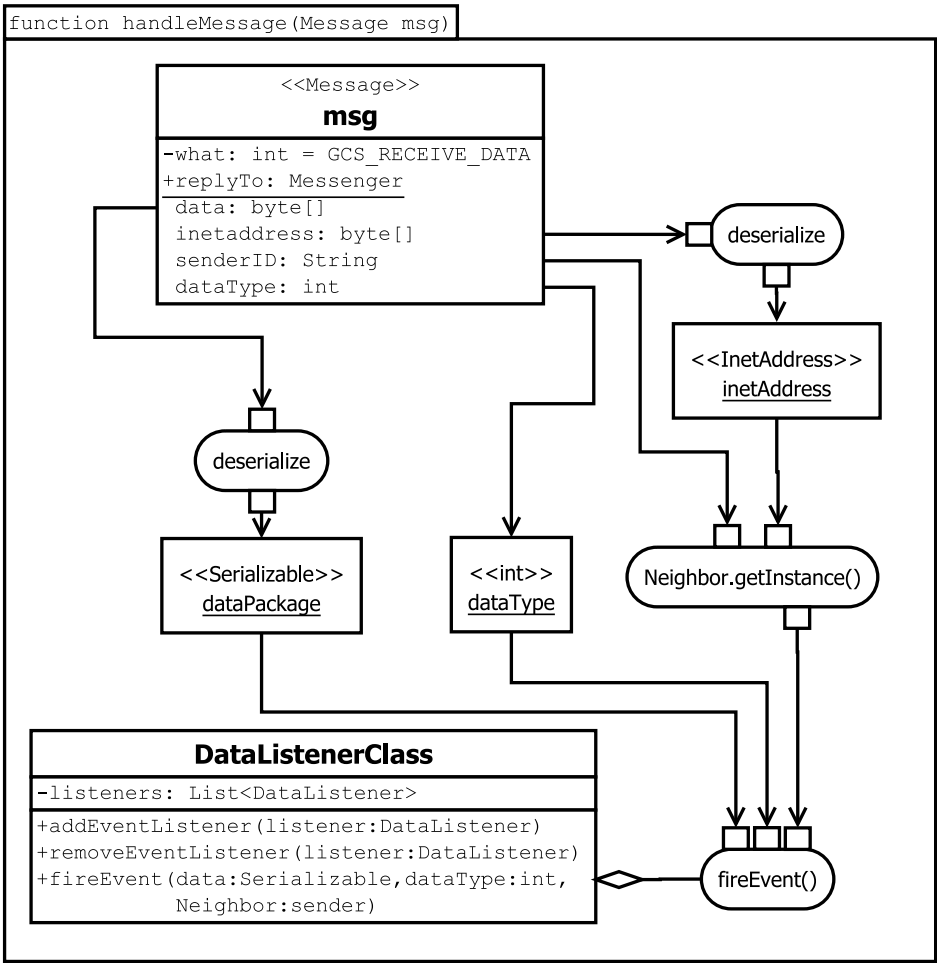

Figure 4.9: Activity diagram: GameCenterController receiver side

Chapter 5

Case Study TicTacToe

In order to test the result of this thesis and to provide reference for future developers, we decided to create a sample game using the GameCenter Library in its full extent. We choose to implement TicTacToe. The reason to go with such a simple game was mainly a high transparency of testing, and easy to follow source code as reference for game developers.

The creation process of the TicTacToe implementation was the benchmark to measure how easy it is to implement GameCenter functionality. We managed to use the core functionalities of the GameCenter with very little effort. The binding and registration of the game TicTacToe is done with one function call. Two more were necessary to create a *PlayerObserverTask* and a *DataListener* to manage players in the network and to handle incoming messages from neighbors.

All together the core functionality of the GameCenter were available after less than 10 lines of source code.[1]

For interested developers we recommend to read through Appendix B which contains source code from the *GCTicTacToe* game and a guide on how to create GameCenter games.

[1] Additional code is necessary to handle interaction with opponents, such as interpreting messages received by the DataListener.

Chapter 6

Future Work

6.1 Extension

The *GameCenterAPI* and the *GameCenterApp* though closed and stable have a lot of potential for extension. The *GameCenterApp* at the moment is optimized for two-player games. Although multi-player games are highly possible, a lot of work to make up for the lack of ready-to-use multiplayer support would be necessary by the developer. Additionally the *GameCenterAPI* could be broaden to support player groups/teams and broadcast between multiple players. Another way of expanding the capabilities of the framework could be the adding of support for more complex game types. The currently available game types for standard win/loss/draw games could be expaned to further highscore types.

6.2 More Games

The main reason of this group thesis - to provide a stable framework for simple game development - lays the foundation for future work. After providing a basic TicTacToe to demonstrate the capabilities of the GameCenter framework, a stable base is put in place for others to develop their own games.
Further efforts could be made in exploiting the dynamic properties of opportunistic networks for a unique gaming experience. For instance, gameplay could consist of spreading and collecting unique identifiers tied to some location. This concept would allow to get a feeling for the way information propagates in opportunistic networks. The game center at this point is restricted to spreading gaming statistics consisting of a nickname and a few integers. Though a user does not necessarily know where some other player's information originated this way, she gets a better idea of how good her stats are the more other players she meets. This mechanism serves as an incentive for joining new networks, a very desirable behaviour regarding applications of opportunistic networks in general.

Chapter 7

Conclusion

The thesis at hand presents a way to simplify development of multiplayer games that use a wireless network. In order to convince users as well as manufacturers of the possibilities of opportunistic networks, it is achieved by making use thereof via the WLANOpp system.
During the course of this project, a platform for playing games over a wireless network using WLANOpp was successfully implemented in Android™. Additionally, an example game was created, serving as reference for future developers. The API leaves space for different scoring systems, allowing developers to be creative on their part.
Power consumption is an important factor in opportunistic networks as nodes are mobile and have only limited energy at their disposal. However, developing the game center and the example game it was made sure only to exchange data whenever the user actively uses these apps in order to conserve energy.

Appendix A

Java programming for Android™

A.1 Serialization

Serialization is a process to convert abstract objects to binary form. This is useful in many cases. In this thesis mainly used to transmit data via WLAN.

Altough the synchronization is done by the GameCenterAPI and therefore game developer may send arbitrary (serializable) data objects without handling synchronization, you can find basic java code for synchronization and desynchronization in Listings A.1 and A.2. [10]

```
private static byte[] Object2Bytes(Serializable input) throws
    IOException {
 ByteArrayOutputStream bos = new ByteArrayOutputStream();
 ObjectOutputStream oos = new ObjectOutputStream(bos);
 oos.writeObject(input);
 oos.flush();
 oos.close();
 bos.close();
 return bos.toByteArray();
}
```

Listing A.1: Serialization code

```
private static Object Bytes2Object(byte[] bytearray) throws
    StreamCorruptedException, IOException {
 ByteArrayInputStream in = new ByteArrayInputStream(bytearray);
 ObjectInputStream is = new ObjectInputStream(in);
 try {
  return is.readObject();
 } catch (ClassNotFoundException e) {
  Log.e("Bytes2Object", "Fatal error while deserializing: Class not
      found: " + e.getMessage());
  return null;
 }
}
```

Listing A.2: Deserialization code

A.2 Missing or not being regenerated R.java problem

If you use *Eclipse* which is currently standard due to the fact that it depends to the official Android Developer Tools ADT, you will someday have the problem, that you can't compile anymore. If you check console for compile errors you will propably notice that the R file is missing. (This file is used to map memories)

If this is the case the following checklist proved as helpful in all cases: [11]

- Check terminal for errors.
- Make sure that anything the R.java links to is not broken. Fix all errors in your XML files. If anything in the ADKs are broken, R will not regenerate.
- If you somehow hit something and created import android.R in your activity, remove it.
- Run Project ->Clean. This will delete and regenerate R and BuildConfig.
- Make sure Project ->Build Automatically is ticked. If not, build it manually via Menu ->Project ->Build Project .
- Wait a few seconds for the errors to disappear.
- If it doesn't work, delete everything inside the /gen/ folder
- If it still doesn't work, try right-clicking your project ->Android Tools ->Fix Project Properties.
- Check your *.properties files (in the root folder of your app folder) and make sure that the links in there are not broken.
- Right-click your project >properties >Android. Look at the Project Build Target and Library sections on the right side of the page. Your Build Target should match the target in your AndroidManifest.xml. So if it's set to target 17 in AndroidManifest, make sure that the Target Name is Android 4.2. If your Library has an X under the reference, remove and re-add the library until there's a green tick. This might happen if you've moved a few files and folders around.

Appendix B

Tutorial for game developer

This tutorial will explain how the GameCenterAPI and App can be used to create a custom two player game with working highscores.

B.1 Prerequisites

In order to create a WLANOpp GameCenterGame the following software and libraries are required.

- Android SDK [12]
- WLANOpp which includes the *BurundiLib* Library. Contact Sacha Trifunovic for current version.
- WLANOpp GameCenter Library.
- WLANOpp GameCenter App.

B.2 Initial Setup

Follow the next steps to set up a new WLANOpp game to take advantage of the GameCenterAPI.

1. Fire up your Eclipse / Android SDK.
2. Select 'New >Android Application Project'.
3. Make sure to use the following naming convention for the 'Package Name' *ch.ethz.csg.gamecenter.yourgamename*.
4. Set the minimum required SDK at least to API 14 (IceCream Sandwich).
5. The rest of the project setup (Basic Activity layout, etc..) is up to you.

B.3 Import of and linking of Libraries

This section will explain how to import the GameCenter libraries into your workspace.

B.3.1 Import Libraries

Repeat the following steps for the WLANOpp *BurundiLib* Library and the Game Center *GameCenterAPI* API.

To install the *GameCenterApp* and *WLANOpp* drop the according *.apk* files into your Android™device.

1. Select 'File >Import >Existing Android Code Into Workspace'
2. Choose the folder in which you downloaded the newest versions of the libraries.
3. Import the libraries.
4. Repeat until *BurundiLib* and *GameCenterAPI* are visible as projects in your Eclipse Workspace.
5. Include all above libraries into your new project.

B.3.2 Link Libraries

Next, we need to include the two new libraries in the game.

1. Right click on your games project in the package explorer.
2. Select 'Android'
3. In the lower half of the window showing *Libraries* press 'Add'.
4. Select the *GameCenterAPI* and then repeat for *BurundiLib*.

B.3.3 Code Setup

In this part you will learn how to instantiate a *GameCenterController*. Also the use of the *PlayerObserverTask* and the *myDataListener* are introduced.

Before you can start on the Game Center implementation you will need to define a few constants that will help identify your game. For this purpose create a new class called *Constants*. The following code shows the minimum required constants for Game Center to work with your game. You can use this *Constants* class for all other constants you might need for your game to work as well.

The *GameID* must be unique. Recommended lenght for it is 20 characters. You can generate your own GameID by using a random string generator such as *http://random.org*.

```
package ch.ethz.csg.gamecenter.yourgamename;
public class Constants {
 // Definition of vital Game Identifiers
 public static final String GameID = "0QYhjyy7L0T9Jde8Ag5o";
 public static final String GameName = "YourGameName";
 public static final String PacketName = "ch.ethz.csg.gamecenter.
    yourgamename";

 // Other Identifiers you might need
 public static final int MessageTypeChallenge = 0;
 public static final int MessageTypeResponse = 1;
}
```

Listing B.1: Constants declaration

Note: The *MessageType 0* should be used for Invitations. The GameCenter will issue an Android™notification to the opponents device for messages with type 0 if the app is closed.

You can do the following steps anywhere in your application - the recommended place is the *MainActivity* of your game.

For the *GameCenter Service* to work, your *MainActivity* will need three private variables. The names of the variables in this tutorial are merely suggestions. After completing the next three sections of this tutorial you will be able to start developing your own game.

PlayerObserverTask

The *PlayerObserverTask* is derived from the *NeighborObserverTask* that is responsible for updating and managing visible Neighbors in the current WLANOpp. The *PlayerObserverTask* is automatically called each time a change in the observed Players in the current WLANOpp is witnessed.

First you need to declare your *PlayerObserverTask* as a private variable to your *MainActivity* as follows:

```
private PlayerObserverTask updatePlayerTask;
```

Listing B.2: PlayerObserverTask declaration

Create a new *PlayerObserverTask* with the next few lines:

```
updatePlayerTask = new PlayerObserverTask() {
 @Override
 public void run(List<Player> newPlayerList) {
  //TODO: Update Screen
 }
};
```

Listing B.3: PlayerObserverTask instantiation

Every time a change in neighboring players is observed, the function *run(List<Player>newPlayerList)* will be called. The passed argument *newPlayerList* will contain the currently visible Players you can reach.

Use this function to display a list of available opponents to allow the player to challenge someone. A simple way to handle challenges can be created with

an *AlertDialog*. The following code sequence outlines this approach. First the players are displayed in a list. A tap on a player's name will open a popup to allow to send a challenge - or any other custom Object.

```
List<String> ListArray = new ArrayList<String>();
AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(this);
// Add all neighbors to List
for (Player n : newPlayerList) {
 ListArray.add(n.getNic());
}
ArrayAdapter<String> listAdapter = new ArrayAdapter<String>(context, android.R.layout.simple_list_item_1,ListArray);

ListView listView = new ListView(context);
listView.setAdapter(listAdapter);
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
 @Override
 public void onItemClick(AdapterView<?> arg0, View arg1, int position, long arg3) {
  final Player player = newPlayerList.get(position);
  AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(context);
  // Set Title
  alertDialogBuilder.setTitle(player.getNic());
  // Set message
  alertDialogBuilder
  .setMessage('Do you want to challenge this opponent?')
  .setCancelable(false)
  .setPositiveButton('Challenge',new DialogInterface.OnClickListener() {
   public void onClick(DialogInterface dialog,int id) {
    gcController.send(player, new ChallengeObject(), Constants.MessageTypeChallenge);
   }
  })
  .setNegativeButton(getString(R.string.cancelButton),new DialogInterface.OnClickListener() {
   public void onClick(DialogInterface dialog,int id) {
    dialog.cancel();
   }
  });
  AlertDialog alertDialog = alertDialogBuilder.create();
  alertDialog.show();
 }
});
```

Listing B.4: Player List with AlertDialog challenge

We will describe the lines *gcController.send()* in the next section.

GameCenterController

The *GameCenterController* is the heart of the GameCenter. It provides access to the GameCenter Database and methods to send custom data objects between players.

Declare a new private *GameCenterController* within your *MainActivity*.

```
private GameCenterController gcController;
```

Listing B.5: GameCenterContoller Declaration

In a next step you need to instantiate the *GameCenterController*. The best place to do this is the *onCreate()* method of your *MainActivity*.

```
try {
 gcController = GameCenterController.getInstance(Constants.GameID,
     Constants.GameName, this, handler,updatePlayerTask);
 gcController.addGame(MyGamesContract.GAME_TYPE_WDL, Constants.
     packetName);
} catch (GameCenterNotFoundException e) {
 // TODO Auto-generated catch block
 e.printStackTrace();
}
```

Listing B.6: GameCenterContoller Instantiation

The first line creates a service dedicated to your game. The second registers your game with the GameCenter Application with an appropriate highscore mode (This example uses *GAME_TYPE_WDL* which stands for Win-Draw-Loss).

myDataListener

The last stand of the GameCenterAPI tripod is the *myDataListener* object. Whenever a player sends a custom data object to another player, this method is called. Within your *myDataListener* you will be able to react to any data received, such as invitations or game moves.

```
private myDataListener dataListener;

 dataListener = new myDataListener(null);
dataListener.dlContext = this;
GCController.addDataListener(dataListener);

private class myDataListener extends DataListener{
 private myDataListener(Neighbor neigh) {
  super(neigh);
 }
 @Override
 public void onDataReceived(final DataReceivedEvent e) {
  // DataObject object = (DataObject)e.data;
  int messageType = e.getDataType();

  if(messageType == Constants.MessageTypeChallenge){
   // React to challenge
  }
  else if(messageType == MessageTypeResponse){
   // React to ChallengeResponse
  }
 }
}
```

Listing B.7: MyDataListener Instantiation

In this example you will notice the different messageTypes we have created in your *Constants* file. You can use those (and more) to differentiate between

different Object types you want to send over your *gcController* to your opponents.

In a last step you will need to add the new *myDataListener* object to your *gcController*.

Storing and retrieving Highscores

To conclude this tutorial we will have a quick look on how you can store encounters and retrieve highscores.

To store a new encounter with an opponent, you will need the API call *reportEncounter()*. It will add your played game into all relevant databases.

```
int myScore = 10;
int oppScore = 5;
int drawScore = 2;
gcController.reportEncounter(opponent, myScore, drawScore, oppScore
    );
```

Listing B.8: Reporting a finished encounter

This is all. It is equally easy to retrive highscores or encounters of other players.

```
ArrayList<Integer> highscoreArray = gcController.getHighscore(
    somePlayer);
int wins = highscoreArray.get(0);
int draws = highscoreArray.get(1);
int losses = highscoreArray.get(2);
```

Listing B.9: Retrieve highscores

For more information on how the individual API calls work, read through the Implementation part of this thesis.

B.4 Sending data with the Game API

After you've successfully set up the basic steps up to here, you are ready to develop a game. The most important aspect of your game is the interaction between players. The *GameCenterAPI* allows you to send custom java objects between players. The only requirement for those objects is, that their class must implement *Serializable*. Serializable Classes are explained in Appendix A.

Attached you will see a simple GameChallenge class that contains the sender's unique identifier as *senderID* and a timestamp to check when the invitation was sent.

```
public class GameChallenge implements Serializable{

 private static final long serialVersionUID = 1L;
 private Timestamp tstamp;

 public Timestamp getTimestamp() {
  return tstamp;
 }
```

```
 public String toString() {
  return tstamp.toString();
 }

 public TTTGameInvitation() {
  tstamp = new Timestamp((int)System.currentTimeMillis());
 }

 public TTTGameInvitation(byte[] bytearray) throws
    StreamCorruptedException, IOException, ClassNotFoundException {

     ByteArrayInputStream in = new ByteArrayInputStream(bytearray);
     ObjectInputStream is = new ObjectInputStream(in);
     TTTGameInvitation Puff = (TTTGameInvitation) is.readObject();
     tstamp = Puff.tstamp;
 }
}
```

Listing B.10: Simple invitation object

To send such an invitation you can use the predifined *send()* method of your *GameCenterController*.

```
TTTGameInvitation challenge = new GameChallenge();
GCController.send(nei, challenge, Constants.MessageTypeChallenge);
```

Listing B.11: GameCenterContoller Instantation

The challenged player will receive this message in his *onDataReceived* method and can dismantel it in a few simple steps.

```
public void onDataReceived(final DataReceivedEvent e) {
 // DataObject object = (DataObject)e.data;
 int messageType = e.getDataType();
 Neighbor challenger = e.getSender();
 if(messageType == Constants.MessageTypeChallenge){
  // React to challenge
 }
 else {return;}
}
```

Listing B.12: GameCenterContoller Instantation

This code snippet concludes this tutorial. You can now start developing your game.

Bibliography

[1] Sasha Trifunovic, Bernhard Distl, Dominik Schatzmann, and Frank Legendre. Wifi-opp: Ad-hoc-less opportunistic networking. 2011.

[2] Wikipedia. History of games. http://en.wikipedia.org/wiki/History_of_games. Accessed October 2013.

[3] Appannie. http://www.appannie.com/top/android/united-states/overall/. Accessed October 2013.

[4] Luciana Pelusi, Andrea Passarella, and Marco Conti. Opportunistic networking: Data forwarding in disconnected mobile ad hoc networks. *Communications Magazine, IEEE*, 44:134–141, 2006.

[5] Alex Pentland, Richard Fletcher, and Amir Hasson. Daknet: Rethinking connectivity in developing nations. *Computer*, 37:78–83, 2004.

[6] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 5th edition, 2012.

[7] Android reference. http://developer.android.com/. Accessed August 2013.

[8] Android Developers. Bound services. http://developer.android.com/guide/components/bound-services.html. Accessed September 2013.

[9] Wikipedia. Unified modeling language. http://de.wikipedia.org/wiki/UML. Accessed September 2013.

[10] JavaFAQ.nu. How can i convert any java object into byte array? and byte array to file object. http://www.javafaq.nu/java-article236.html. Accessed June 2013.

[11] StackOverflow. R.java not generating. http://stackoverflow.com/questions/2757107/developing-for-android-in-eclipse-r-java-not-generating. Accessed March 2013.

[12] Android sdk. http://developer.android.com/sdk/index.html. Accessed September 2013.