Android practical training

Winter term 2011

Baris Oztop, Stefan Matl

**Documentation: 3D Game App**

Mentoring:

Nils Kannengießer

TUM, Fakultät für Informatik

Lehrstuhl 13, Prof. Dr. Uwe Baumgarten

Boltzmannstr. 3

85748 Garching bei München

Contents

[Abstract 2](#_Toc315799464)

[Project Team 2](#_Toc315799465)

[Baris Oztop 2](#_Toc315799466)

[Stefan Matl 3](#_Toc315799467)

[Project Plan 3](#_Toc315799468)

[Idea 3](#_Toc315799469)

[Product 3](#_Toc315799470)

[How to play 3](#_Toc315799471)

[Differences 3](#_Toc315799472)

[Quality requirements 4](#_Toc315799473)

[Must-have features 4](#_Toc315799474)

[Optional features 4](#_Toc315799475)

[Implementation 6](#_Toc315799476)

[Resources 6](#_Toc315799477)

[GUI 6](#_Toc315799478)

[Graphics 7](#_Toc315799479)

[AI 7](#_Toc315799480)

[Basic interfaces and classes 7](#_Toc315799481)

[Project schedule 8](#_Toc315799482)

[Project Flow 8](#_Toc315799483)

[Detailed Description 9](#_Toc315799484)

[Bluetooth Communication 9](#_Toc315799485)

[Classes 9](#_Toc315799486)

[MultiplayerActivity 9](#_Toc315799487)

[ModeSelectionActivity 9](#_Toc315799488)

[ClientNumberPicker 10](#_Toc315799489)

[DeviceListActivity 10](#_Toc315799490)

[BluetoothMPService 10](#_Toc315799491)

[DataServer / DataClient 10](#_Toc315799492)

[Server and Client Side Bluetooth Communication 12](#_Toc315799493)

[Server Side 12](#_Toc315799494)

[Client Side 15](#_Toc315799495)

[Communication with the Main Activity 16](#_Toc315799496)

[Actions According to the Activity Results 18](#_Toc315799497)

[Detailed Test Plan 19](#_Toc315799498)

[Devices that we tested on 19](#_Toc315799499)

[Testing different screen resolution and size 20](#_Toc315799500)

[Project Conclusion 20](#_Toc315799501)

# Abstract

# Project Team

## Baris Oztop

Baris is currently Informatics Master student at TU München, and this is his second semester. He is graduated from Middle East Technical University, Turkey with Computer Engineering major. His favorite object-oriented programming language is Java. He has done several projects in his bachelor and first semester of his master degree. Some of these projects are Database Implementation, Information Visualization, 3D User Interfaces, Rapid Application Development, and Peer-to-Peer Desktop Search Agent. He is currently working as student trainee in Software Development and Consultant Company. His hobbies are Movie and Music Collecting, Photography, Travelling, World Cuisines, and Cooking.

## Stefan Matl

# Project Plan

## Idea

‘Mensch ärgere dich nicht’ is a strategy board game for four players in the classic version. The name of the game means ‘Do not get angry’. Our idea is to design the game in 3D for android platforms using the sensor capabilities of the devices, and enable multi-player game mode so that up to four players can connect their phones and play against each other. The game is one of the famous classical games for all the age groups. It is a good alternative to turn the people’s waiting time e.g. in subway into joyful moments with their peers.

## Product

The product is provided in English, meaning all options, buttons, labels and if necessary annotations or explanations are written in English. The application is designed for Android systems with Android 2.1 or higher. It requires an API 7 or higher. The game will be tested at least on a virtual device with Android 2.1 and two higher versions as well as on real devices.

### How to play

Each player (possible number of players: two, three or four players) has four pegs and aims at getting them from their first position, out-region, into their final home fields. First, the player has to throw a dice. In the beginning all the pegs of the player are blocked and the player is allowed to throw the dice three times until throwing a six. With every six, a player can unblock one peg from its out-region, and in each round the peg is moved according to the number the dice shows. A player is not allowed to move the peg on a field already occupied by one of his other pegs. If the field is occupied with another player’s peg, he can kick out the other player and force him to restart with that peg.

### Differences

Basically the application is intended to work and behave the same way a real board game would react. But in contrast to a real game, where people are sitting around a table, it is not possible to manipulate the dice or influence the number the dice shows. Once the player has thrown the dice, he has to move his selected peg by exactly that given amount and nothing else. It is not possible to cheat or move the pegs of another player. A move cannot be undone. If the GUI allows it, the player will be able to turn the board to get a different point of view.

## Quality requirements

The app will neither crash during an appropriate usage nor while a time-out or not-responding exception is sent. If there is a connection problem during the game, the user can decide whether to abort or reconnect.

### Must-have features

The following features are non-optional and therefore cannot be removed or skipped:

* *Resources like pictures, models and sound effects:*

In order to make the game more interesting for the player, the graphics and models have to be provided in high detail. Resources like textures are needed for the rendering.

* *Graphical user interface for controlling the game:*

The GUI allows the player to control the game. With the GUI the player can select and move his pegs on the board or change game settings.

* *3D game rendering:*

With a 3D rendering the game should look and feel like 3D. With OpenGL ES we can define properties like perspective or depth for the 3D rendering. Therefore all the game objects, like pegs, board or dice should be rendered in 3D.

* *Enabling connection with up to three other mobile android devices:*

In order to allow multi-player game mode, the app has to connect to multiple different devices where the same app is already installed. After setting up a Bluetooth connection, two or more players can join the same game and play against each other.

* *At least one artificial intelligence in order to enable a two-player mode:*

If there is only one single player and no one else available, the app has to start an artificial intelligence, meaning a computer-based non-human player acting as an enemy for the player. In that mode no connection is needed.

* Hurry-up mode:

If a player doesn’t move within a given amount of time, the app will make a move for him, which is either randomly selected or a calculated worst-case move. In case a player takes too long many times, his best peg is sent back to the start position. The user can set this maximum waiting time in the settings or he can turn this mode off.

### Optional features

Optional features will be implemented if there is time left after the implementation of the must-have features:

* *Real-time rendering:*

This defines a special type of rendering where all the objects are rendered fast enough, so that the player doesn’t notice any time delay or other kinds of interruption. The rendering itself should be separated from the game calculation.

* *Connection for up to eight players:*

There are different versions of the game, e.g. one with an eight-player mode. In order to realize that, the app requires an option to connect with seven other phones.

* *Multiple artificial behaviors in order to increase game fun:*

Because only one AI is easy to beat, there have to be multiple different kinds of computer-based players. Each of them needs another algorithm for calculating the next move. It might even be possible to implement them in a way to provide them on different levels like ‘easy’, ‘medium’ and ‘hard’.

* *Animations like zooming the camera or moving pegs:*

In order to increase game fun, it would be a nice feature to play some animations and have some special effects, e.g. when a peg is moving or forced to return home or a player wins the game.

* *Shaking feature for supporting a realistic dice throwing:*

Because random methods are sometimes not very efficient and don't really return a random number, it would be a great feature and of course providing additional fun, if the player could shake the phone in order to throw the dice. Based on the movement of the player a number is calculated.

* *Dynamic moves:*

Whenever a player makes a move, an animation shows his peg sliding from one field to another. By providing a high frame rate, the single steps will look like one dynamic move.

* *Support of different languages:*

In order to increase the amount of possible customers, the layout of the app is given in multiple different languages, e.g. German, Spanish, French or Chinese.

## Implementation

The app will be implemented with Eclipse and NetBeans as well as with the associated Android emulators. The use of Java 1.6 or higher is intended.

### Resources

* *Pictures:*

In order to reduce calculations, some basic layouts can be drawn with programs like GIMP or other drawing software. Also the app needs an icon, so that everyone can identify it, if it is available in the Android market and for a better marketing. Parts of the models or the board can be textured with images.

* *Models:*

For a 3D rendering, 3D models are needed. Models can be used in order to render a peg, the game board and the cube. This 3D data can be created with software like SolidEdge, SolidWorks, Catia V5 or any other modeling software. Even small Java programs could be used for calculating 3D models.

* *Sound:*

During special effects e.g. when a peg is sent back or a player wins, it is recommended to play some sound. If there is enough time to implement this, a sound will be played all the time during the game. Therefore the player should have an option to turn it off.

* *Path on the board:*

Every single peg is only allowed to move along special paths on the board. Those paths can be implemented as a graph or an array.

### GUI

A graphical user interface is required in order to control the game. Therefore touch events have to be registered as well as a change in gravity or acceleration if the player shakes the phone. With the use of an OnTouchListener and a SensorListener it is possible to catch those events and process them. The GUI will provide an option menu to change some game settings or enable the user to start and quit the game. Most of the screen will be used to show the game and has to be updated very fast, meaning a very high frame rate. By touching the screen, the player can e.g. select his current peg which he wants to move. The game has to match the touch coordinates with the 3D coordinates in order to select the right peg. Also the player can move the camera and zoom close to his pegs. Of course the GUI shows the player the current situation, meaning that after every move all the players (devices) have to receive an update.

### 

### Graphics

The game can be rendered with OpenGL ES, which can be used as a package, which provides high-level methods in order to paint the single models. Those need to be read and converted from a 3D resource into a triangle-based model. Once it is written to a byte buffer, OpenGL will add it to an intern 3D matrix. Additional parameters like luminosity and depth can be set.

### AI

In order to implement an artificial intelligence it is recommended to analyse the game as well as different strategies and choose the best ones for the behavior of the AI. For an algorithm that calculates the next move it is important to have certain criteria to find and rate a move. Usually the AI can calculate moves during the player’s turn, meaning that the calculation can be done parallel to the rest of the game. If different AIs are implemented (see 4.2 Optional Features), the user can set the level of difficulty of the current one(s).

### Basic interfaces and classes

classes: PLAYER, DICE, PEG, BOARD, GLSURFACEVIEW

interfaces: RENDERABLE

PLAYER:

properties: name, color

methods: dice()

PEG extends RENDERABLE:

properties: number, color, position

methods: move(), render(), return\_home()

BOARD extends RENDERABLE:

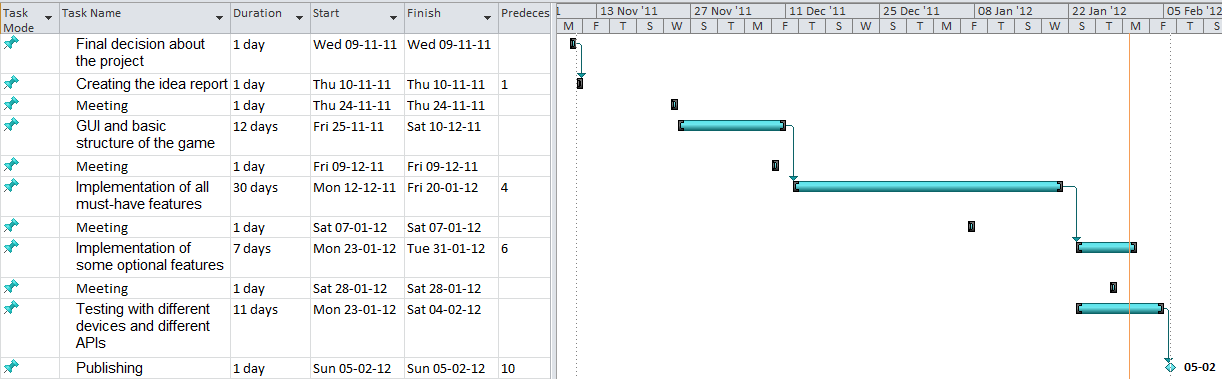
properties: size, paths, color, texture

methods: render()

RENDERABLE:

methods: render()

## Project schedule



**Kick-off**

09.11.2011 Final decision about the project

**Analysis**

10.11.2011 Creating the idea report

24.11.2011 Meeting

**Draft**

25.11.2011 - 10.12.2011 GUI and basic game structure

**Implementation**

11.12.2011 - 19.01.2012 Implementation of all must-have features

11.01.2011 - 19.01.2012 Implementation of some optional features

**Testing**

20.01.2012 - 27.01.2012 Testing with different devices and different APIs

**Time buffer**

28.01.2012 - 04.02.2012

**Publishing**

05.02.2012 Early project submission

# Project Flow

Below you can find the problems that we have encountered during the development phase of the game application:

* We wanted to develop an application to run on the earlier versions of Android OS to make our game accessible by more people. Therefore we used the API level 10 (Android 2.1). However, there was a case where standard methods, which is also written in the Android developers’ page, didn’t work. One of these cases was showing a progress bar while client devices are connection to server device. Hence, server device user will know how many users are connected via the completion level of the progress bar. We initialized the progress bar object with the corresponding activity’s context by getting it with getApplicationContext() method, however this didn’t work. This was a hard to spot failure that we spent nearly one and a half day to figure what the problem is. After our research, we figured out that Android 2.1’s progress bar initialization doesn’t get the Activity’s context with getApplicationContext()method but simply calling the *ActivityName.this* technique. However, getApplicationContext() works only newer versions of Android. This is the case just for initializaing the progress bar, the other calls for getApplicationContext()still works in Android 2.1.
* We wanted our game application able to handle 7 client devices connected to a server device. Therefore, while 4 devices’ users are playing the game, the other 4 devices’ users might be able to watch the game with their smart phones. However, during our tests we realized that the third and more devices are not able to connect to server device whose Android version is 2.3.4 or below. This was the most time consuming problem that we encountered during the development of the Bluetooth communication.

# Detailed Description

## Bluetooth Communication

### Classes

Multiplayer version of the game uses Bluetooth for the communication between devices. Following graphs show the class diagrams for the package de.tum.multiplayer and de.tum.multiplayer.bluetooth, which are responsible for the multiplayer version of the game.

### MultiplayerActivity

After user selects the Multiplayer version of the game from the WelcomeActivity screen, MultiplayerActivity is created. Upon creation of the activity, board and dice object are drawn to allow the upcoming dialog activities use it as background view for better user experience. Other related classes and threads communicate with MultiplayerActivity via onActivityResult() and the Handler object, respectively. Under the titles named “Communication with the Main Activity” and “Actions According to the Activity Results”, you can find the details of this communication written on the code snippet.

### ModeSelectionActivity

Right after the MultiplayerActivity is created, another activity named ModeSelectionActivity comes as a dialog screen. This activity allows the user to select the device mode, and make the current device discoverable for other devices to allow pairing if they haven’t been paired before. One device might be server or client.

### ClientNumberPicker

If user selected to make his device as server device at the ModeSelectionActivity, next activity will be ClientNumberPicker activity. This allows the server device user to select how many clients will connect it to play the multiplayer mode. Game application supports 7 more devices to connect to server device. However, only 3 of them can play the game while the rest can follow/watch the game on their devices’ screen.

|  |
| --- |
| **Note:** During our tests with different devices and different Android OS versions, we have seen that only the devices whose Android version is 2.3.5 or above can accommodate up to 7 devices as client, while earlier versions of Android only able to accept 2 client devices. Therefore, it’s suggested to make the device whose Android version is 2.3.5 or above be as client if the users are planning to play with more than 3 devices total. |

### DeviceListActivity

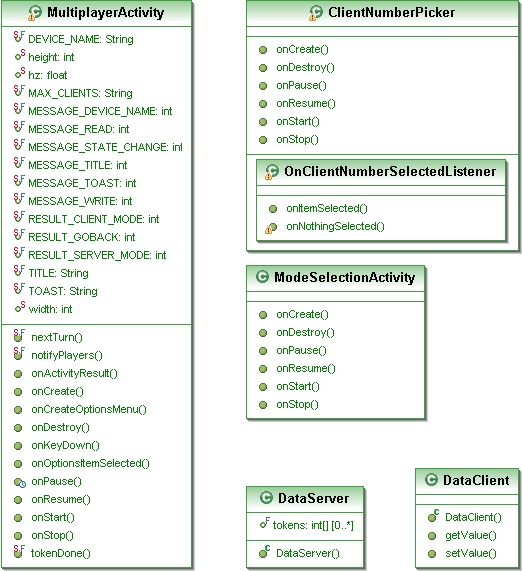
If user selected to make his device as client device at the ModeSelectionActivity, next activity will be DeviceListActivity displaying the already paired devices and allows scan more devices which are discoverable. Therefore, user won’t need to switch from game to his device’s Bluetooth settings to pair his device with others. Upon selection of device name form the list. MAC address will be passed to already initialized BluetoothMPService, and client’s connection attempt to server device will start.

### BluetoothMPService

This is the class where all the connection is handled via Threads. You can find the detail explanation at the following section with its sequence diagrams.

### DataServer / DataClient

These classes’ objcets are represents the data sending to other devices. It has fields to accommodate game’s state.



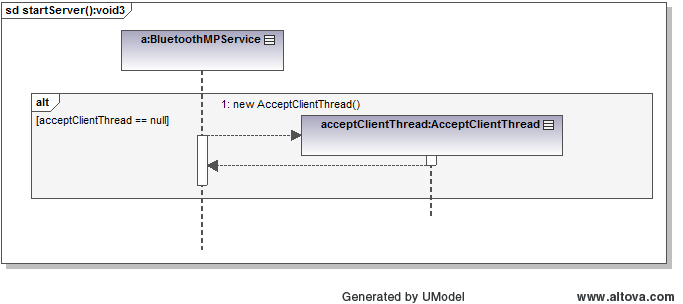
Class Diagram for the package de.tum.multiplayer



Class Diagram for the package de.tum.multiplayer.bluetooth

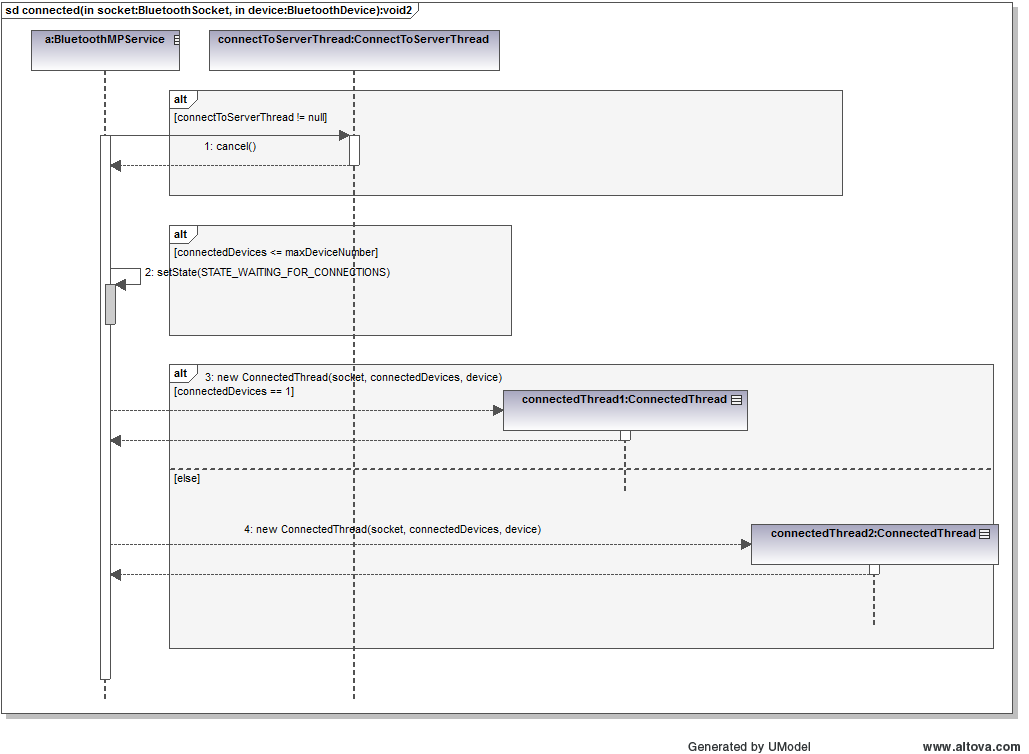
## Server and Client Side Bluetooth Communication

### Server Side

After user, who is using the server device, decides how many clients devices will be available for the game and decides the properties of teams, startServer() method is called from the BluetoothMPService class. This method creates a thread (AcceptClientThread) for listening upcoming client connections to the server device. At this moment, server device user is informed with a progress dialog box for each connected devices.

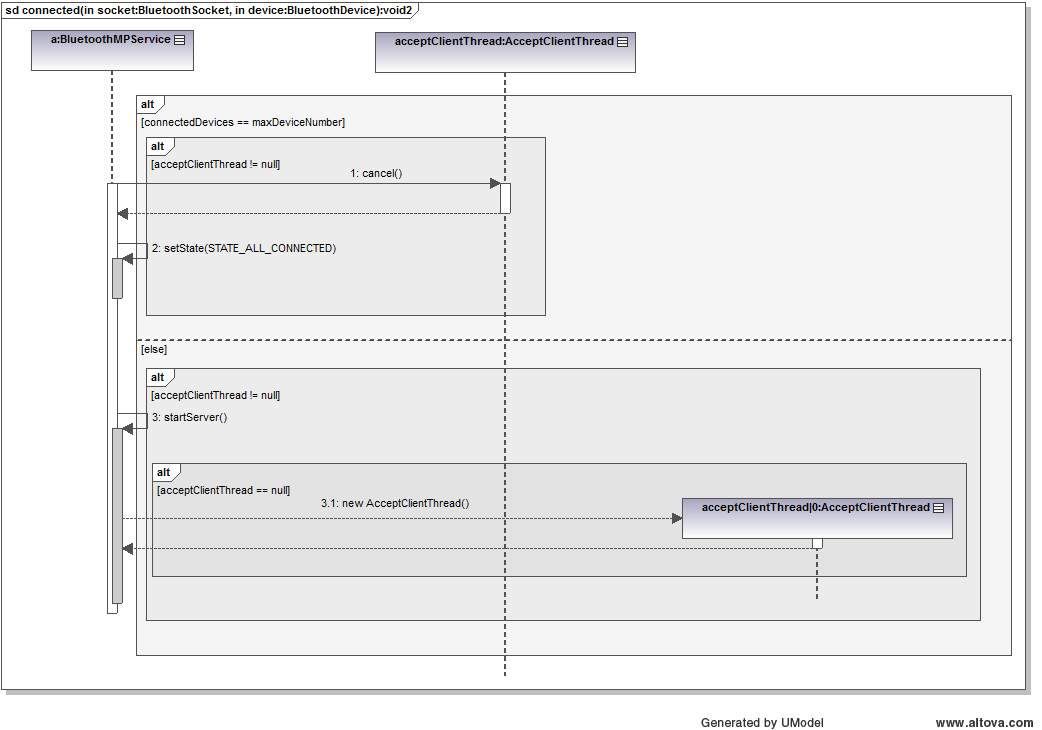
Creating a thread for listening upcoming client connections to the server device.

After a client requests a connection to the server device, another thread (ConnectedThread) is created for dealing with the communication between devices. If the maximum number for the connected devices is not reached, state of the service is set to “STATE\_WAITING\_FOR\_CONNECTIONS”, and the thread for accepting new connections (AcceptClientThread) is recreated. Following sequence diagram is the simplified version for server accepting two client devices.



Accepting client a connection and waiting for more devices.

Accepting client connection thread runs until all the devices are connected to the server device (When connectedDevices is equal to maxDeviceNumber), and after each connected device it is reinitialized to make the BluetoothSocket free. For each of the 7 clients there is a unique UUID, which is incremented after each connected device. When all the client devices are conencted, the state is set to “STATE\_ALL\_CONNECTED”, and letting the MultiplayerActivity know with handling a Message, and game starts. Following sequience diagram shows those method calls in the BluetoothMPService class.



Checking if all devices are connected or not.

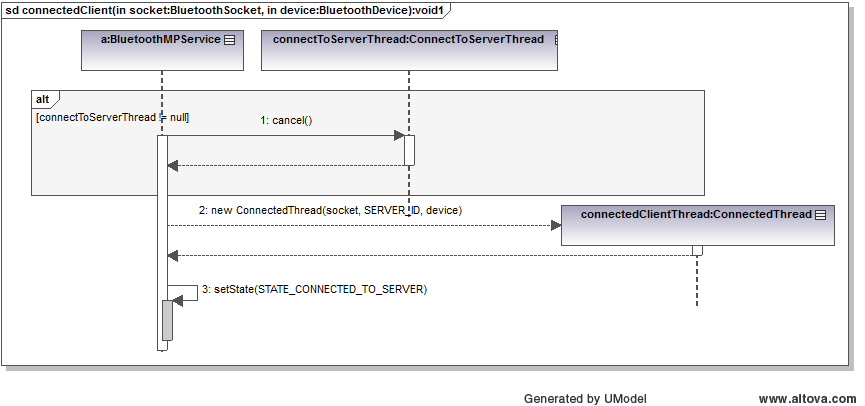
Following table shows the states while communicating with the main activity (MultiplayerActivity). Messages are sent via Handler initiated in the MultiplayerActivity, and handled to the BluetoothMPService when it is initialized.

|  |  |
| --- | --- |
| **STATE\_NONE** | When BluetoothMPService class is initiated |
| **STATE\_LISTEN** | When server is listening for incoming client connections |
| **STATE\_WAITING\_FOR\_CONNECTIONS** | When one or more clients are already connected to server, but waiting for more devices |
| **STATE\_ALL\_CONNECTED** | When all the devices are connected to server, and ready to start the game. |

### Client Side

When the user of the device selects the client mode, the next activity will be the screen where the paired devices list and unpaired devices list appear. User can also select to scan for more devices by clicking the button named “scan for devices”. If both devices are not paired before, the server machine user can select the option “Make This Device Discoverable” before switching to server mode. Hence server machine will be discoverable for 300 seconds, afterwards client device user can pick the server name from the list to be paired and complete the device selection activity (DeviceListActivity).

Following sequence graph reflects the activities done after the user decides his machine to be client. Selected server device’s MAC address is passed to the thread which is establishing connection by probing the socket with corresponding UUID. If one of the UUID is failed due to already reserved for another device on the server side, clients continue with probing the next UUID on the Socket. After the connection is established, another thread is created (connectedClientThread:ConnectedThread) to start the data transfer between the server and itself.



Client device connects to server device

Following table shows the states to communicating with the main activity which is MultiplayerActivity. Messages are sent via Handler created in the MultiplayerActivity, and handled to the BluetoothMPService when it is initialized.

|  |  |
| --- | --- |
| **STATE\_NONE** | When BluetoothMPService class is initiated |
| **STATE\_CONNECTING\_TO\_SERVER** | When probing the UUID on the opened Socket |
| **STATE\_CONNECTED\_TO\_SERVER** | When Socket is accepted and connected to the Server device |

## Communication with the Main Activity

Following table describes the messages arriving from BluetoothMPService, and the actions according to message.

|  |
| --- |
| **private** **final** Handler mHandler = **new** Handler() {  @Override  **public** **void** handleMessage(Message msg) {  **switch** (msg.what) {  **case** *MESSAGE\_STATE\_CHANGE*:  Connection state is changed  **case** *STATE\_ALL\_CONNECTED*:    All the devices are connected to the server, so start the game, and set the title to “All connected”  **break**;  **case** *STATE\_WAITING\_FOR\_CONNECTIONS*:    Toast the connected device number (Server side)  **break**;  **case** *STATE\_CONNECTED\_TO\_SERVER*:  **break**;  **case** *STATE\_CONNECTING\_TO\_SERVER*:  Set the title on the client side to "Connecting to server..."  **break**;  **case** *STATE\_LISTEN*:  **break**;  **case** *STATE\_NONE*:  **break**;  }  **break**;  **case** *MESSAGE\_WRITE*:  The message that this device has been sent to other(s)  **break**;  **case** *MESSAGE\_READ*:  Message has arrived to this device from other(s). Convert it to do the corresponding object depending on if this is client or server device  **break**;  **case** *MESSAGE\_DEVICE\_NAME*:  **Server Side:** A client connected to the server. Change the value of the progress bar on the server side according to total number of connected devices.  **Client Side:** Toast the name of the server device/  **break**;  **case** *MESSAGE\_TOAST*:  Game state has been changed, and letting the player know about the command that he has to do.  **break**;  **case** *MESSAGE\_TOAST\_WARNING*:    Warn the user about the connection lost or connection failed situation, and bringing options menu to the screen to let the user continue to the game as client or server again.  **break**;  **case** *MESSAGE\_TITLE*:    Change the text on the title bar with message arrived from BluetoohMPService  **break**;  }  }  }; |

## Actions According to the Activity Results

Following table describes the actions after the results come back from the Activities started in the MultiplayerActivity.

|  |
| --- |
| **public** **void** onActivityResult(**int** requestCode, **int** resultCode, Intent data) {  **if** (*D*)  Log.*d*(*TAG*, "onActivityResult " + resultCode);  **switch** (requestCode) {  **case** *REQUEST\_CONNECT\_SERVER*:    Result comes from ModeSelectionActivity. MAC address of the server selected device will be used to probe the connection in the BluetoothMPServer’s connectServer() method.  }  **break**;  **case** *REQUEST\_ENABLE\_BT*:    If Bluetooth is not enabled on the device, application will enable the Bluetooth. If result is not positive, then user will be prompt to continue with single player mode.  **case** *REQUEST\_MODE\_TYPE*:  **if** (resultCode == *RESULT\_CLIENT\_MODE*) {  User selected to be client on his device at the ModeSelectionActivity. Now, another activity named DeviceListActivity will let the user to select the name of the server device to connect.  } **else** **if** (resultCode == *RESULT\_SERVER\_MODE*){  **if** (clientNumberPickerIntent == **null**) {  User selected to be server on his device at the ModeSelectionActivity. Now, another activity named ModeSelectionActivity will let the user to select the number of the client devices to play with.  } **else** {  Server device’s user selected how many clients will be available. Now, this information will be set in the BluetoothMPService as well, and it’s server process will be started. Then a progress bar will be created to show the states of the connections to the server device.  } **else** **if** (resultCode == *RESULT\_CANCELED*) {  } **else** **if** (resultCode == *RESULT\_GOBACK*) {    Result when an Activity is ended with pressing back button before it does its job. So, the MultiplayerActivity will be ended as well.  }  }  } |

# Detailed Test Plan

## Devices that we tested on

* HTC Desire HD (Android 2.3.3)
* HTC Desire HD (Android 2.3.5)
* Samsung Galaxy S2 (Android 2.3.4)
* Samsung Galaxy S (Android 2.3.3)
* Nexus One (Android 2.3.6)
* Nexus One (Android 2.3.7)

## Testing different screen resolution and size

We tested our game application different activity screens on different screen sizes to see how it fits to the screen. We mainly used the different screen size options given in the Android’s graphical layout development tool in Eclipse IDE.

# Project Conclusion