Wireless and Mobile Networks

Università degli studi di Milano

Emanuele Bosetti[1]

June 21, 2016

[1]emanuele.bosetti@studenti.unimi.it

II

Contents

1

Goals and architecture of the project

The project for the Wireless and Mobile network's exam is an application that communicates with a short range wireless technology. In my project,I've decided to use the bluetooth technology to develop a game called "Bluetooth Battleship".

1.1 Goals

- The application must run on Android devices with version greater than or equal of 4.5.
- Using a embedded Bluetooth module inside a device.
- The application must respect the rules of the traditional battleship game.

1.2 Why Bluetooth

Bluetooth is a technology born in 1999 used to create a short range, low battery consuming and low speed Wireless Personal Area Network (WPAN); the protocol fits exactly my requirements for the Battleship Application.

I've used a Bluetooth in a RFCOMM mode: it provides a simple reliable data stream to the user, used directly by many telephony related profiles as a carrier for AT commands, as well as being a transport layer for OBEX over Bluetooth. Many Bluetooth applications use RFCOMM because of its widespread support and publicly available API on most operating systems.

The data is serialized from Java object to bit array and sent using a emulated RS-232 interface.

The Java object used for communication is a custom data structure called BluetoothMessage. The class contains a flag that defines the type of message sent and other information for shoot and shoot response message type.

1.3 Architecture

The application was developed to do a peer-to-peer network. It can be run in two ways: the server mode and the client mode. The player in the first menu of the application chooses if run the application in server or client mode.

If the application runs in server mode it's started a thread that listens the incoming bluetooth connection on the android device, otherwise if the application starts in client mode, the user can choose the enemy from the list of paired devices provided by operating system. The pairing process can be performed using a operating system method in the device's settings tab, or using a function developed in the application; during the peering process the user must confirm the code send by the other device and the operating system does the authentication process.

1.3.1 Software Architecture

The software was developed using a normal pattern in the Android environment:

- Model-view-controller pattern: this patter is used to spit the control logic (business logic) from the view and model logic. In the controller classes are managed the functionalities of the application: the view is used for manage the user data representation and the model is used for storage the data model used by the application.

- Observer pattern: this pattern is used to notify at the view that the model are changed; the view, when the notification is received, calls a method in the controller object to get the update data. In this manner is possible to do a operation directly on the data without the worry of having to upgrade the view.

- Factory Pattern: this pattern is used to create a standard interface for object generation. In the application it is used for boat creation and for putting them in battle board. The class Factory manages all creation phases, included the error recovery routine and the Board management.

- Singleton Pattern: this patter is used for object instantiated just one time during the application life cycle. This pattern allows to access the object within each part of the application; it is useful for objects like BluetoothWrapper and GameManager.

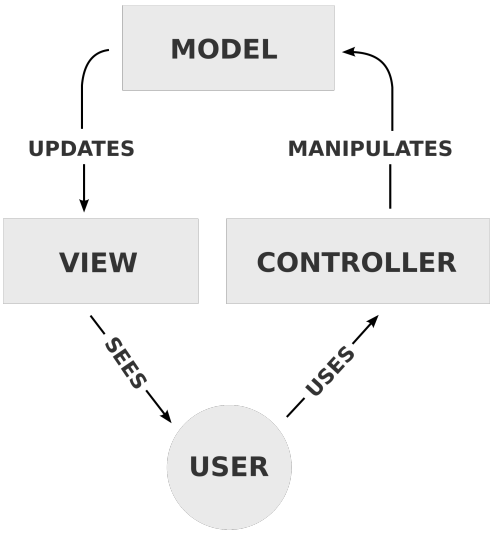

Figure 1.1: MVC Pattern

1.3.2 Communication Message Architecture

The communication of the device is managed by the class Bluetooth Message: it provides a functionality for all message exchanged by the application. The class provides an internal flag that indicate the type of message exchanged.

The possible exchanged messages are the following:

- DECIDE_FIRST_SHOOT: In this phase all boats are positioned in the game board: for deciding what player starts to shoot, two number between 0 a 65536, one for device, are randomly chosen. The numbers are exchanged by the two instances of application and the device with the biggest number is the first to shoot.

- SHOOT: In this message one player sends the shooting coordinate to other player. This message indicates the row and the column of the enemy board.
- SHOOT_RESPONSE: In this message the player responds of a SHOOT message, indicating if the target has been hit or not. This message includes the row and column and the shooting information response.
- NOTIFY_WIN: The device loser notifies to the other device that the game is over.

```
public static final int DECIDE_FIRST_SHOOT = 0;
public static final int SHOOT = 1;
public static final int SHOOT_RESPONSE = 2;

private int type;
private int column;
private int row;
private int shootStatus;
private Object Payload;
```

The MessageBluethoot classes provide an additional generic Object called payload, for generic information transport.

2

Android Application

2.1 Workflow

When the application starts, it presents at the player a simple menu which can be carried out three choice:

- Play: for starting a session in client mode. The application returns a list of paired devices to the user.
- for starting a play server. The application starts in a server mode and it waits a connection to the other device.
- for pairing new device. The application discovers a new device, set up in visible mode and it starts a operating system pairing process.

2.1.1 Start a Play Server

When the application starts, it first checks that the Bluetooth is turned on. If it is off, turns it on. To do this, you need the permission of Bluetooth admin and so it was necessary to modify the Android manifest by adding the following two lines:

```
<uses-permission android:name="android.permission.BLUETOOTH"
   />
<uses-permission android:name="android.permission.
   BLUETOOTH_ADMIN" />
```

If the bluetooth is turned on the application using a BluetoothWrapper instance calls a BluetoothService object. This object has a method called start(); this method set up all necessary data structures for bluetooth server and it starts a specific thread for listening a new connection from the other devices.

```
public void run() {
    setName("AcceptThread" + mSocketType);
    BluetoothSocket socket;
    // Listen to the server socket if we're not connected
    while (mState != STATE_CONNECTED) {
        try {
            // This is a blocking call and will only return
               on a
            // successful connection or an exception
            socket = mmServerSocket.accept();
        } catch (IOException e) {
            break;
        }
        // If a connection was accepted
        if (socket != null) {
            synchronized (BluetoothService.this) {
            switch (mState) {
                case STATE_LISTEN:
                case STATE_CONNECTING:
                // Situation normal. Start the connected
                   thread.
                    connected(socket, socket.getRemoteDevice
                       (), mSocketType);
                    break;
                case STATE_NONE:
                case STATE_CONNECTED:
                    // Either not ready or already connected.
                        Terminate new socket.
                    try {
                            socket.close();
                        }catch (IOException e) {
                            Log.e(TAG, "Could not close
                               unwanted socket", e);
                        }
                    break;
                }
            }
        }
    }
}
```

After the first phase of connection has successfully completed, the server starts a new thread called mmServerSocket. This thread manages all connection phases and sends back to the Bluetooth wrapper the connection status information and the raw message with a Handler object. The Handler object could operate in an asynchronous environment, like the socket communication. The BluetoothWrapper class manages the row data and update

the Battleship model structure.

```
private  final Handler mHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case Constants.MESSAGE_STATE_CHANGE:
                switch (msg.arg1) {
                    case BluetoothService.STATE_CONNECTED:
                        Log.d(TAG,"Connected");
                        setChanged();
                        notifyObservers(Constants.
                            CONNECTION_SUCCESFUL);
                        break;
                    case BluetoothService.STATE_CONNECTING:
                        //BT Service state connect
                        break;
                    case BluetoothService.STATE_LISTEN:
                    case BluetoothService.STATE_NONE:
                        //BT Service just initialized
                        break;
                }
                break;
            case Constants.MESSAGE_WRITE:
                byte[] writeBuf = (byte[]) msg.obj;
                // Message sent
                break;
            case Constants.MESSAGE_READ:
                //Message received
                byte[] readBuf = (byte[]) msg.obj;
                reciveInfo(readBuf);
                break;
        }
    }
};
```

The Handler object manages the asynchronous phases of the service: when an event occurs, the thread that manages the connection sends a notification to the Handler, running in the main loop of the application (main thread) The possible internal application messages are the following:

- MESSAGE_STATE_CHANGE: The connection thread notifies that a state has changed, and the possible state are:
  - STATE_CONNECTED: The connection is established and the main socket is created; the application notifies with the Observer pattern to the GameManager to start a new game.

  - STATE_LISTEN: The ConnectionService is waiting a new connection.
  - STATE_NONE: The ConnectionService is just created.
  - STATE_CONNECTING: The connection is established but the socket is about to be created.

- MESSAGE_WRITE: A new message has been written in the socket.
- MESSAGE_READ: A new message has been received and has to be read, the reciveInfo() method interprets the raw message, reassembly it into the BluetoothMessage object and passes it into the method doActivityIncomingMessage(), that routes it into the correct controller depending on the type of data received.

```
private void doActivityIncomingMessage(BluetoothMessage bm){
    if (bm.getType()==BluetoothMessage.DECIDE_FIRST_SHOOT){
        Game.getGameBoard().receiveNonce((Integer)(bm.
            getPayload()));
    }else if (bm.getType()==BluetoothMessage.SHOOT){
        Log.d(TAG,"Receive Shoot");
        Game.getGameBoard().receiveEnemyShoot(bm.getRow(),bm.
            getColumn());
    }else if (bm.getType()==BluetoothMessage.SHOOT_RESPONSE){
        Log.d(TAG,"Receive Shoot Repsonse");
        Game.getGameBoard().receiveEnemyShootResponse(new
            ShootResponse(bm.getRow(),bm.getColumn(),bm.
            getShootStatus(),(Integer)bm.getPayload()));
    }
}
```

2.1.2 Start the game as a client

The application could start in the client mode: in this case the application tries to connect to another instance started as a server. The role of the application could be decided by the user in the first menu.

When the "Play" button is pressed, the application starts an Activity, which makes the user select a paired device. When the user selects a device, the application gets the MAC Address of chosen device from the operating system, and it tries to start a connection.

```
//Add observer for notification of connection established
Game.getBluetoothWrapper().addObserver(this);
//Get a paired device and add it to the View
```

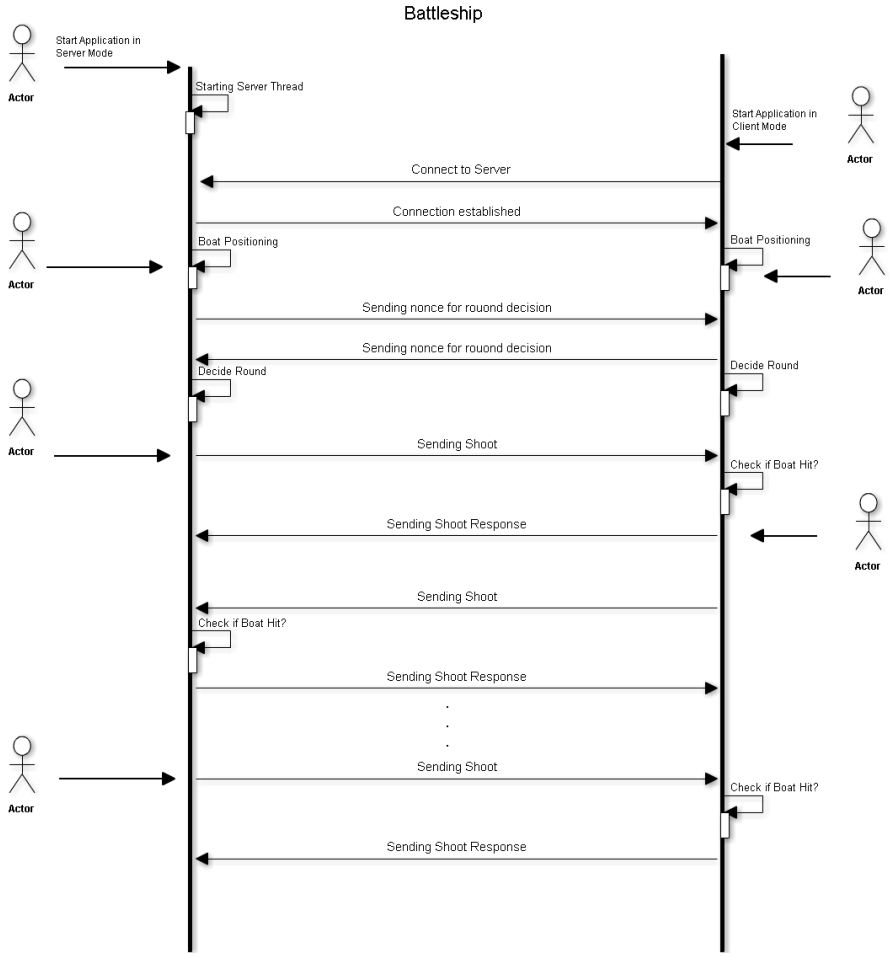

Figure 2.1: Workflow

```
for (BluetoothDevice device : Game.getBluetoothWrapper().
    getPairedDevices()) {
     deviceName.add(device.getName());
     this.bdevices.add(device);
}
ArrayAdapter listAdapter = new ArrayAdapter<>(this, android.R
    .layout.simple_list_item_1, deviceName);
listView.setAdapter(listAdapter);
listView.setOnItemClickListener(new AdapterView.
    OnItemClickListener() {
     @Override
     public void onItemClick(AdapterView<?> parent, View view,
         int position, long id) {

         //Set listener for onPressButton for start a
```

```
            connection
        BluetoothDevice bd = bdevices.get(position);
        Game.getBluetoothWrapper().getBluetoothService().
            connect(bd,false);
        Log.d("ChooseEnemy","Press"+position);
    }
});
```

When the connection is established, the application starts a new thread and it communicates with the BluetoothWrapper with an Handler. (See "Start a Play Server" subsection for more information).

2.2 Application logic and data structure

The logic of Battleship game is managed by the class GameBoard: this class is run in singleton pattern mode and iit can be accesseed in all part of the application.

The class contains the data structures used for managing the game and the status of the Boat. The data structures used are the following:

- Board: Matrix of Field object, used for storaging the status of the single board object.
- EnemyBoard: Matrix of Field object, used for storaging the information of the enemy Board.
- Boat: ArrayList of Boat.
- Boat: ArrayList of Enemy Boat.

During the boat positioning phase, the user chooses a Field inside the Board, and the BoatFactory class manages the entire process of creation. The boats are filled inside the Board and the OutOfBoundException is checked for boats that are placed outside the game field.

The BoatFactory class manages also the dimension of the boat: an array list programmatically set contains the available dimension. In the first version of the application the available dimensions are:

- 3 Boat of dimension 2
- 2 Boat of dimension 3
- 1 Boat of dimension 4

- 1 Boat of dimension 5

The process of shooting, inside the application, is managed by the method receiveEnemyShoot(). This method, when a shoot from enemy arrives, checks on the local board if a ship has been hit and in case if the ship is sunk.

The control for checking if the boat is hitten is performed directly on the board; for checking if the ship sinks, it is performed on the ArrayList of boat.

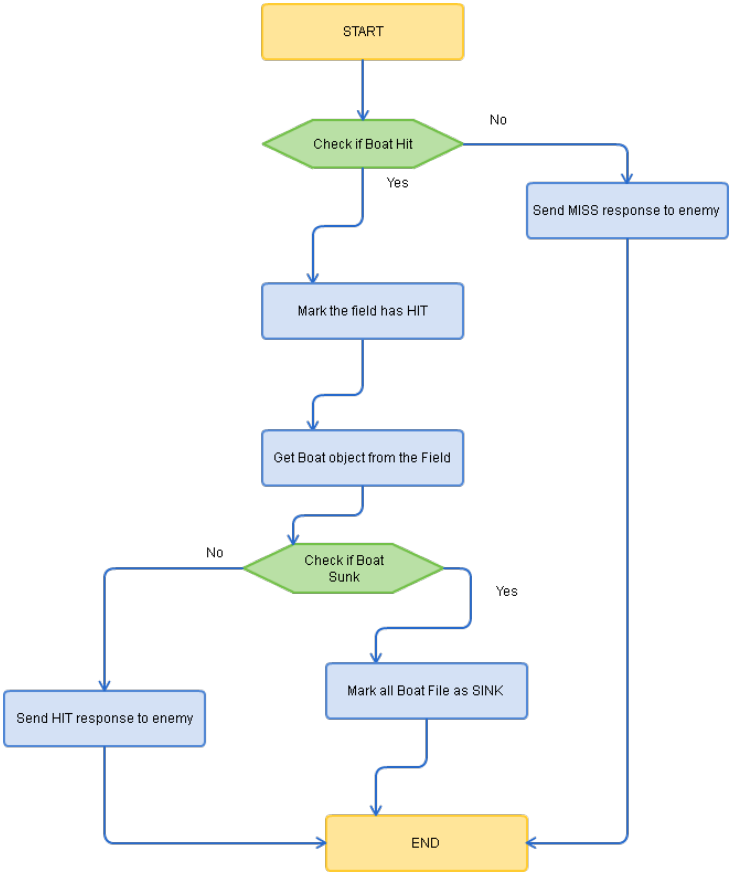

Figure 2.2: The Shoot Process

```
public void receiveEnemyShoot(int r, int c){
    ShootResponse sr;
    try {
        if(this.Board[r][c].getField().equals(Field.BOAT)){
            this.Board[r][c].setField(Field.HITBOAT);
            Boat b = this.Board[r][c].getBoat();
            if(b.checkIfSink()){
                sr = new ShootResponse(r,c,ShootResponse.SINK
                    ,b.getId());
```

```
            }else{
                sr = new ShootResponse(r,c,ShootResponse.HIT,
                    b.getId());
            }
            this.sendShootResponse(sr);
        }else {
            sr = new ShootResponse(r, c, ShootResponse.MISS,
                -1);
        }
        this.sendShootResponse(sr);
    } catch (RequestBoatOfEmptyFieldException e) {
        e.printStackTrace();
    }
}
```

3

Conclusion

The Battleship application perfectly fits the project goal. Now, with this application, it is possible to playing a Battleship game using a bluetooth connection; the application is easily useful and it can be run in Android device with the built in bluetooth module: this method permits to maintain the cost of the product very low.

The application use a RFCOMM bluetooth method, that uses an emulated RS-232 console over a wireless connection, so it's easy using and developing it. The application is built with multi-thread technology, that allows the application to be running without wasting too many resources and it allows the user to use his mobile phone during a heavy or waiting operation.

3.1 Future development

In this project it could be developed more features:

3.1.1 Develop of watchdog for socket connection

Now, if the socket is interrupted during the game (phone locked or standby, high distance between the device etc.) the client application doesn't restart the socket, so the application becomes unusable. It's important, in the next release of application, developing a watch dog for the socket: it can monitor the socket and it tries to restart it in case of disconnection or, in case of failure, it notifies this to user via GUI.

3.1.2 More useful and beautiful GUI

The application in this version use a basic graphic provided by android, for the game is important to have attractive GUI, is important to use a

framework for game that could implement more user-friendly graphics.

Appendix A

Application Screenshoot

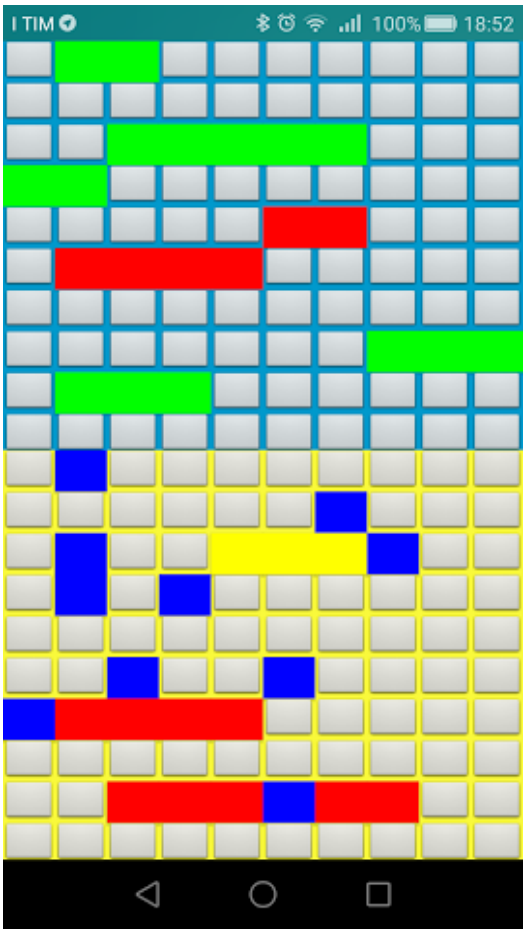

Figure A.1: Game Board

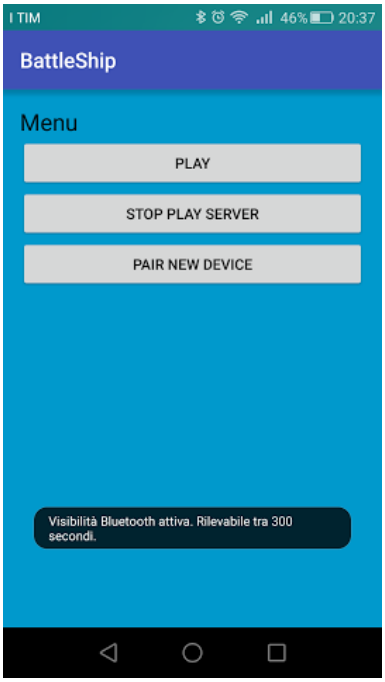

Figure A.2: Screenshoot 1

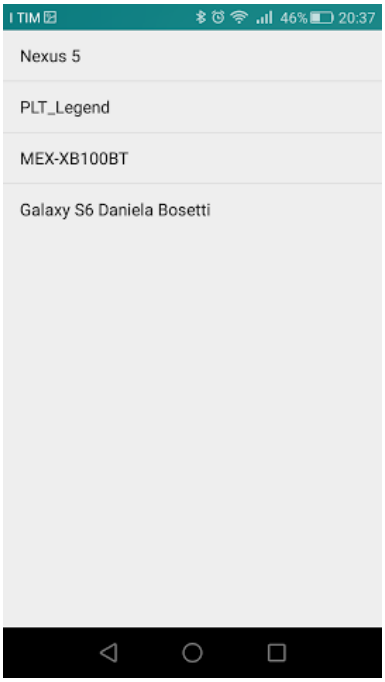

Figure A.3: Screenshoot 2

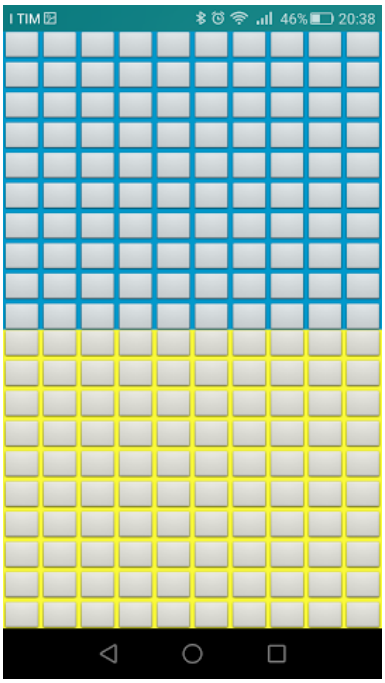

Figure A.4: Screenshoot 3

Figure A.5: Screenshoot 4