

**FHVrobot**

Documentation

**Lecture**

S1: Kopplung und Integration von heterogenen Systemen

**Supervisors**

Prof. (FH) DI Wolfgang Auer

Prof. (FH) DI Patrick Ritschel

**Project Team**

Nicolaj Höss, BSc

Marko Petrovic, BSc

Kevin Wallis, BSc

# Abstract

The project *FHVrobot* describes and documents the implemented components for a game which allows a user to control a robot, using a *Beaglebone Black* as embedded computer, via an *Android* application. The software architecture consists of several heterogeneous components which are connected through a wireless communication channel. The communication is based on the User Datagram Protocol (UDP).

The work describes in detail the defined protocol which supports the remote controlling of a robot, equipped with several sensors, two motors and a flashlight. Furthermore, the document describes how each component is designed and the protocol stack of the defined protocol is implemented for each component, in the programming languages *Java*, *C++* and *C#*.

A major goal of the work also was, to provide a low-latency camera stream from a resource-limited system to several clients. Various third-party programs have been evaluated according their ability to provide a reasonably low latency of the stream in the context of controlling a robot. The final approach to fulfil this requirement was to directly access the compressed camera image using *Video4Linux* and to send the compressed data over UDP. Additionally a server component was used as the only receiver of the camera stream. The server component is then responsible for re-distributing the stream to an arbitrary amount of clients. This noticeably reduces the used bandwidth and processing power of the resource-limited system.

Additionally, the defined protocol and implementation of the protocol stack was analysed with respect to possible security issues and with respect to its performance. The results of the analysis show that the defined protocol misses encryption and incrementing packet number, thus. However, the implementation shows very good performance on local networks and over the internet.

Table of Contents

[Introduction 4](#_Toc443340874)

[Game Concept 5](#_Toc443340875)

[Implementation 6](#_Toc443340876)

[*Server* 6](#_Toc443340877)

[*NetworkLibrary* 9](#_Toc443340878)

[Robot 11](#_Toc443340879)

[*Android* App 15](#_Toc443340880)

[Game Server 17](#_Toc443340881)

[Protocol Definition 20](#_Toc443340882)

[Session Layer 20](#_Toc443340883)

[Presentation Layer 20](#_Toc443340884)

[Application Layer 21](#_Toc443340885)

[Communication Analysis 24](#_Toc443340886)

[Security Aspects 24](#_Toc443340887)

[Performance Analysis 24](#_Toc443340888)

[Quick Start 25](#_Toc443340889)

[Installation and Setup 26](#_Toc443340890)

[Robot Prerequisites 26](#_Toc443340891)

[*Server* Prerequisites 26](#_Toc443340892)

[*Game Server* Prerequisites 26](#_Toc443340893)

[App Prerequisites 26](#_Toc443340894)

[Configuration 27](#_Toc443340895)

[*Server* 27](#_Toc443340896)

[*Game Server* 27](#_Toc443340897)

[*Android* App 27](#_Toc443340898)

[Robot 27](#_Toc443340899)

[Troubleshooting 28](#_Toc443340900)

# Introduction

This document provides technical information and implementation details of the project *FHVrobot*, which was particularly designed for the course *S1: Kopplung und Integration heterogener Systeme* at the *University of Applied Sciences Vorarlberg.* The course addresses the topic of coupling heterogeneous systems, this includes communication between different hardware and software components over a wired or wireless communication channel.  
The project’s target group are visitors of the *FHoffen*, which is the open-door day of the *University of Applied Sciences Vorarlberg.*

The project *FHVrobot* can be considered as a game involving a robot, which can be controlled by a smartphone app. The robot has to be driven through a parkour, wherein the goal of the game is to beat the opponent’s driving time.

To meet the requirements of the project *FHVrobot* various components were implemented. The *Server* component, which is responsible for managing an arbitrary amount of connected app clients and an arbitrary amount of connected robots, was written in *Java* using *JavaFX* as Graphical User Interface (GUI) framework. The *Server* component is also responsible for delegating commands, which are received from the app client, to the controlled robot. Furthermore, various sensor values and media streaming is forwarded by the *Server* from the robot to the app client.  
Additionally, an app client was implemented for *Android* devices. The app client can connect to the above described Server component to control the robot. The app client also shows the camera image which is streamed in realtime from the robot.  
As third component, the robot itself, was implemented. The robot uses a *Beaglebone Black* as embedded computer. The robot can be controlled through two independent Direct Current (DC) motors. Moreover, the robot is equipped with a webcam, a Lidar sensor and a 9-axis Inertial Measurement Unit (IMU). The robot’s software was written in *C++*.

Lastly, a *Game Server* was implemented, which is responsible for managing the game logic. This includes time measurement, using a self-implemented camera trigger system, and managing the high-scores.

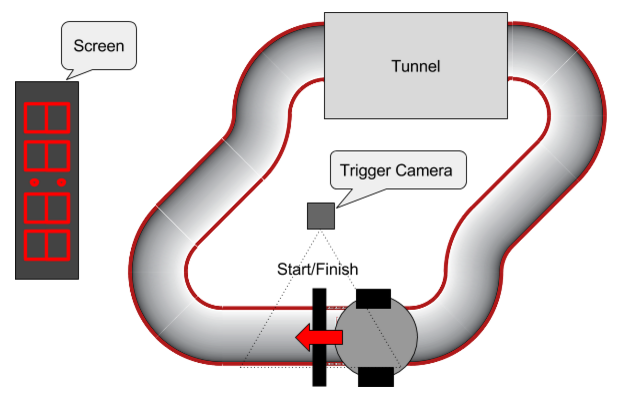
The communication between all components is based on the User Datagram Protocol (UDP), however, a full protocol stack is built upon it. This includes the implementation of a session layer, a presentation layer and an application layer for each component.

The following chapters describe in detail the game concept itself, the software design and implementation details of the components. Furthermore, the defined protocol is documented in detail. Additionally, evaluations concerning possible security issues and several tests to determine the communication performance were done.

# Game Concept

The general game concept is to control a robot via a smartphone application through a parkour. The goal of the game is to beat the opponent’s best time. The parkour consists of curves, tunnels and other obstacles. The players cannot directly compete against each other, instead they play one after another, i.e. there is only one robot on the parkour at a time. To make the game more exciting, the robot is equipped with various sensors and a camera, which is streamed to the user’s smartphone. Additionally, the robot is equipped with a flashlight, which can be used in the tunnel to enlighten it.

Figure 1 shows an exemplary setup of the game. The shown screen displays the current time of the run and the current high-scores. The current time of the run is also propagated to the user’s smartphone. The camera shown in the setup is responsible for triggering the time measurement. The time measurement is started when the robot passes the starting line. The user can then control the robot through the parkour. The time measurement stops when the robot passes the finish line. The exemplarily shown tunnel can be designed in a way, that the ambient light does not lighten it, which forces the user to use the equipped flashlight of the robot.



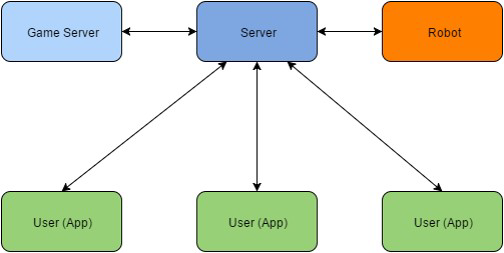
*Figure1: Examplary setup of the parkour and environment.*

However, the game is not limited to competing against each other. A user can choose to do

a free run, which allows him to train controlling the robot or simply explore the environment using the equipped webcam, flashlight and orientation sensors.

# Implementation

The system consists of four components, in particular the *Server*, the robot, the app clients and the *Game Server*. Figure 2 shows these components in their respective context.

  
*Figure 2: Communication between the components of FHVrobot*

The communication between the *Server* and the robot, respectively between the *Server* and the app clients is a persistent or stateful connection, which uses heartbeats to ensure that the connection is still established. The connection between the *Game Server* and the *Server* is a non-persistent or stateless connection, i.e. the *Server* provides a service interface for the *Game Server*. This service interface is used to fetch information from the *Server*, e.g. requesting the current operator’s name and to provide a possibility to delegate the current time of a parkour run and high-scores to the app clients.

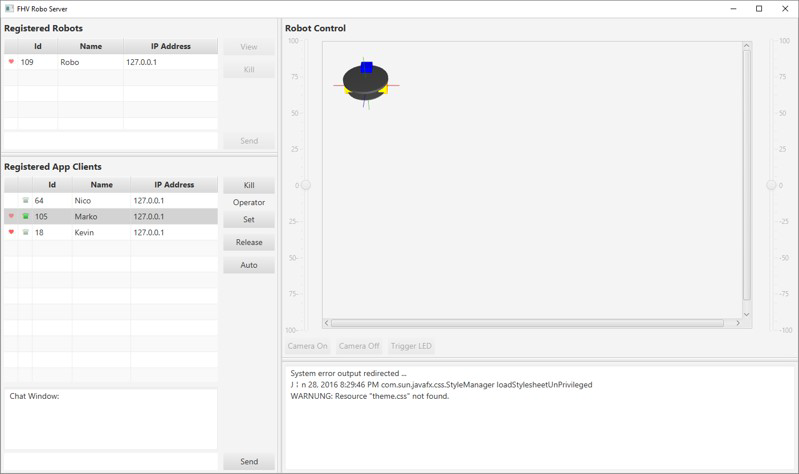
As one can see, the *Server* is the central component of the implemented architecture, therefore, it has to be accessible from all other components, and i.e. it needs a public accessible IP address. Alternatively, if all the components are connected to the same private network, it does not necessarily need a public IP.

The following chapters describes each component’s implementation and software design in detail.

## *Server*

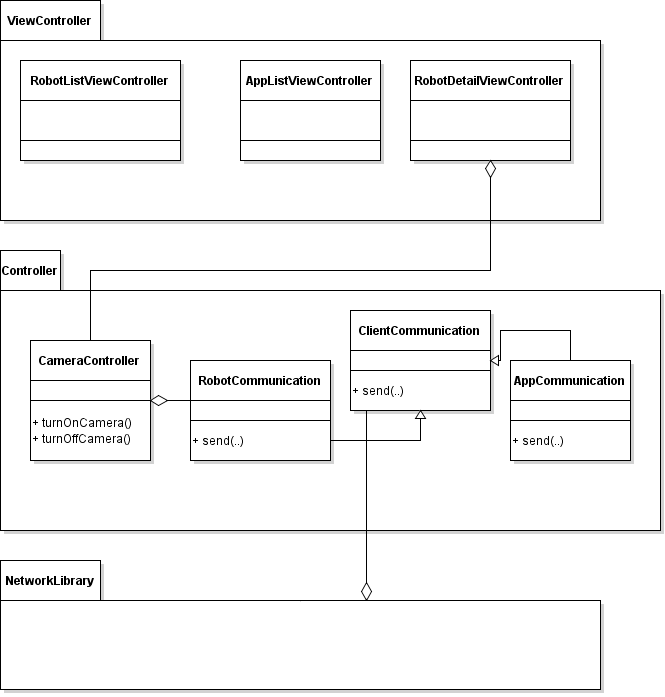
The *Server* is implemented in *Java* using *JavaFX* as GUI framework. For the implemented protocol stack, a library, hereafter called *NetworkLibrary*, was implemented. The *NetworkLibrary* was also re-used in the implementation of the *Android* application. The implementation of the *NetworkLibrary* is described in the section [NetworkLibrary](#h.9wu8jynupa6).

Figure 3 shows a screenshot of the implemented *Server* application. The GUI of the server shows a list of all currently connected robots and a list of all currently connected app clients. From the server all connected robots and clients can be managed, i.e. the connection can be aborted, messages can be sent and app clients can be set as operator. Furthermore, the *Server* supports an *Auto Mode* which automatically sets and releases the operator state from the connected app clients, either in a specific time interval or if a app client disconnects from the *Server*. Additionally, the camera stream of a robot can be shown and the robot can be operated directly, i.e. the motors can be controlled, the flashlight can be turned on and off and the camera stream can be started and stopped. The *Server* also shows the robot’s orientation in 3D-space, represented by a virtual 3D representation of the physical robot.

  
*Figure 3: Screenshot of the implemented Server*

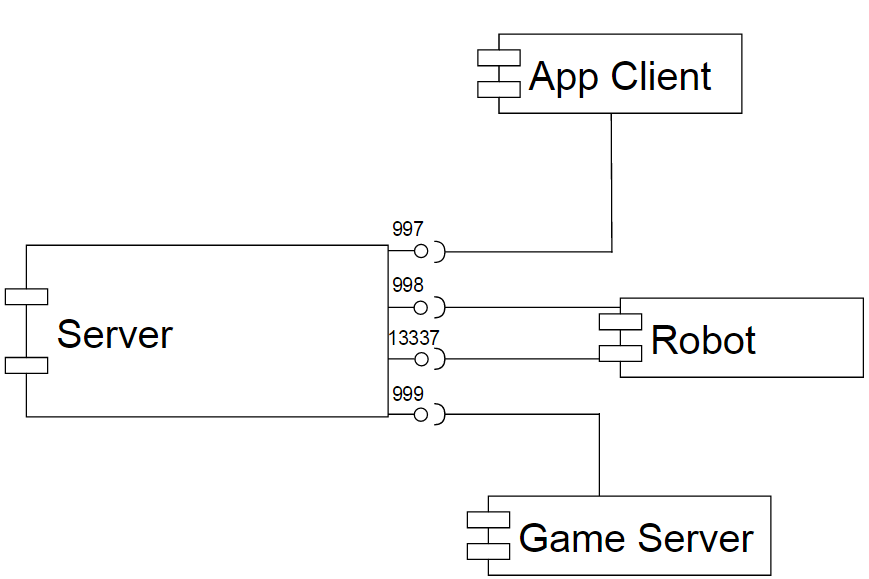
The *Server* is implemented as layer architecture. Figure 4 shows an excerpt of the class diagram of the implemented design. Each part of the view has its own ViewController which is responsible for providing the *JavaFX* property which can be bound to the actual GUI element. The ViewController also controls the view’s behavior. Implementing separate ViewControllers decouples the GUI completely from the lower layers, which offers the opportunity to easily change the look-and-feel of the *Server*’s GUI.  
For each functionality a Controller was implemented. Controllers which need to communicate with a client get a ClientCommunication object injected, which provides methods to communicate with the client using the defined protocol. The ClientCommuncation uses the *NetworkLibrary* to eventually transmit messages.

The implemented design allows easy and straightforward extensibility of new functionality.

 *Figure 4: UML diagram of the layer architecture of the Server component*

On startup and at runtime the *Server* application opens and listens on various communication channels. On startup the port 997 is opened as UDP port for communicating with app clients. Additionally, the port 998 is opened as UDP port for communicating with robots. Physically separating these communication channels provides higher security against attacks. Security aspects of the communication is further described in chapter [Security Analysis](#h.puubtfrh2n3a).  
Beside the ports 997 and 998 the *Server* also opens the port 999, which provides the service interface of the *Server*. Connections established on the port 999 are not considered stateful and therefore do not implement heartbeats. Port 999 is used by the *Game Server* which requests and sends data on demand.

On runtime the *Server* opens UDP ports between 13337 and 13592 for receiving a robot’s camera stream. The specific port depends on the robot’s session id and is opened and closed on demand. Figure 5 illustrates the communication channels between the components in a UML diagram.



*Figure 5: Communication ports of the Server component*

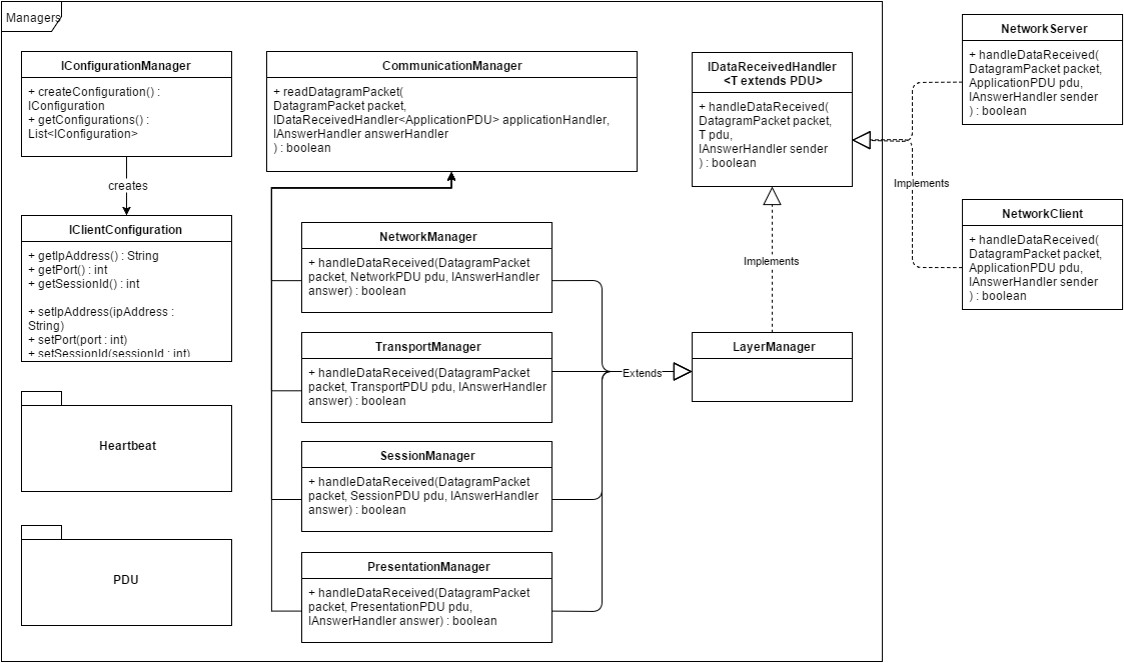
## *NetworkLibrary*

The *NetworkLibrary* is a library written in *Java* and is used by the Server and the *Android* application project. It implements the defined protocol stack described in chapter [Protocol Definition](#h.9d5e8gslm0yo).

Figure 6 shows a class diagram of the implemented *NetworkLibrary* of those classes which are particularly responsible for the communication. It consists of LayerManager classes which implement the Open System Interconnection (OSI) layers above the transport layer. However, the transport layer is also implemented as LayerManager class and wraps the actual UDP connection. This concept allows to easy switch from a UDP connection to a Transport Control Protocol (TCP) connection, without changing the implementations of any other layer.  
As it can be seen, the ApplicationLayer is not implemented in the *NetworkLibrary* project, instead this layer has to be implemented by the *NetworkLibrary*’s using component, i.e. the *Android* application and the *Server* project.

The CommunicationManager class combines the single layer implementations and is responsible for sending received packets through the layers. The received packet is wrapped into a PDU class which can be enhanced and reduced by the specific layer implementations.

Furthermore, the *NetworkLibrary* takes responsibility of sending and receiving heartbeats. Missing heartbeats are propagated to the using component of the *NetworkLibrary*, which can be handled individually, however, most likely ending in aborting the connection.

 *Figure 6: Class diagram of the NetworkLibrary*

The following code snippet shows the handling of received data in the CommunicationManager. First, it reduces the input data length to the actual length of the received data, i.e. it removes unused bytes from the receive buffer. Second, it creates the specific PDU object for each layer, e.g. the NetworkPDU for the NetworkManager. Afterwards it calls the specific LayerManager and let it handle the created PDU object. If the specific layer manager can handle the PDU the handleDataReceived(..) from

the LayerManager returns true. If a LayerManager returns true, the PDU is not handled by the layers above. If a LayerManager return false, which means that the specific LayerManager cannot completely fulfil the request, the PDU is then handed over to the upper layer.

This concept allows that specific requests are directly handled by a specific layer, e.g. session requests are handled by the SessionManager only and upper layers are not affected by received session requests.

|  |
| --- |
| public void readDatagramPacket(DatagramPacket packet, IDataReceivedHandler<ApplicationPDU> applicationHandler, …){  byte[] data = Arrays.copyOfRange(packet.getData(), 0, packet.getLength());  // Network stack  NetworkPDU n = PDUFactory.createNetworkPDU(data);  if (n == null || n.handleDataReceived(packet,network,answerHandler)) {  return;  }  TransportPDU t = PDUFactory.createTransportPDU(network.getInnerData());  if (t == null || t.handleDataReceived(packet,transport,answerHandler)) {  return;  }  …  ApplicationPDU a = PDUFactory.createApplicationPDU(pres.getInnerData());  if (a == null) {  return;  }  applicationHandler.handleDataReceived(packet, application, answerHandler);  } |

*Listing 1: Code snippet of the CommunicationManager*

As described above, the actual application layer is not implemented by the *NetworkLibrary* to allow the using component fully control of the application layer, i.e. implementing its own logic for the ApplicationPDU. The following code snippet shows how the *NetworkLibrary* can be used for implementing application logic. The snippet shows how the command CHANGE\_NAME is implemented in the *Server* application. According to the defined protocol this command requires to send a confirmation to the client, which is done by using the ANSWER\_FLAG.

|  |
| --- |
| public boolean handleDataReceived(DatagramPacket packet,  ApplicationPDU pdu,  IAnswerHandler sender) {  Client client = (Client) manager.getCurrentConfiguration();  byte[] payload = pdu.getPayload();  int command = pdu.getCommand();  int flags = pdu.getFlags();  if (command == Commands.CHANGE\_NAME) {  String name = new String(payload);  client.setName(name);  DatagramPacket dg = DatagramFactory.createDatagramPacket(  client,  Flags.ANSWER\_FLAG,  DatagramFactory.NO\_PAYLOAD);  Commands.sender.answer(dg);  }  …  return true;  } |

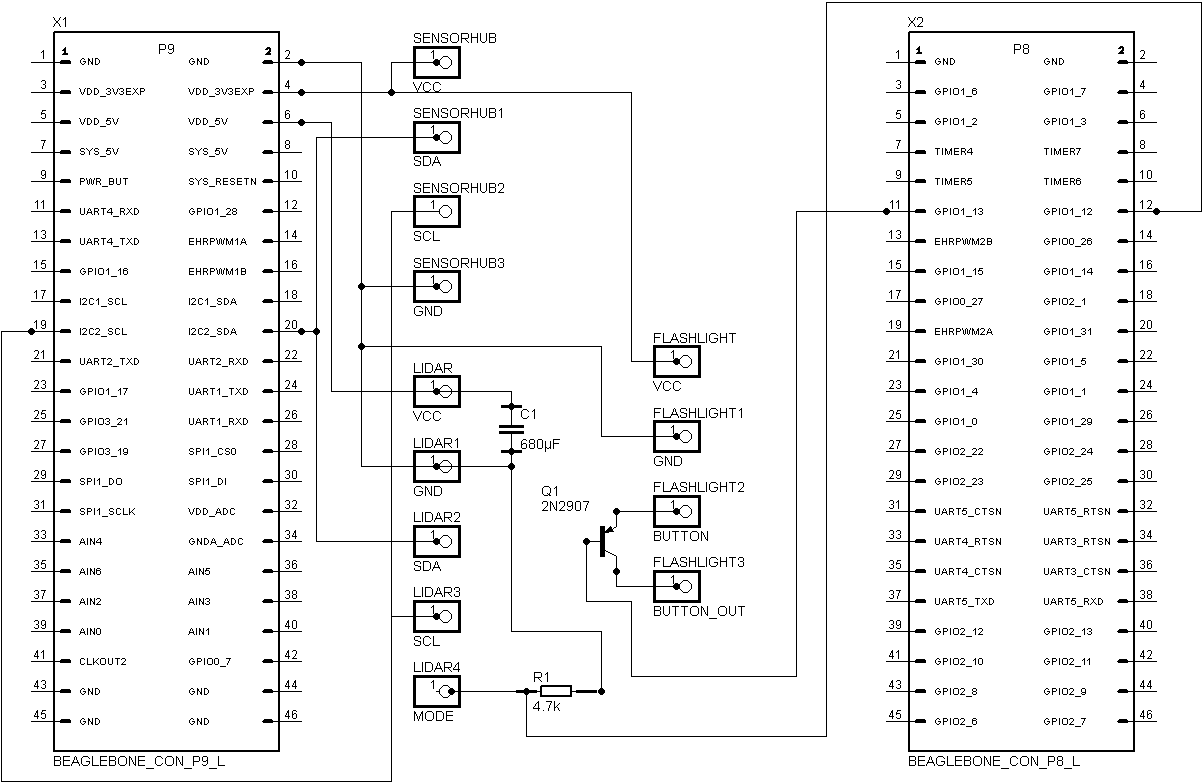
*Listing 2: Exemplary implementation of the application layer*

## Robot

The robot’s hardware is built upon the *2-wheel DFRobot Turtle mobile platform[[1]](#footnote-1)*. As embedded computer a *Beaglebone Black*, running with a *Debian Linux*, is used. The Beaglebone Black’s functionality was extended using the following extension capes:

1. *Power Supply Cape Revision 1[[2]](#footnote-2)* with a rated input voltage of 6 to 28 Volts
2. *DMCC Dual Motor Control Cape[[3]](#footnote-3)* which allows to control two DC motors independently in forward and reverse
3. Self-made extension cape with an integrated adapter for a *Texas Instruments Sensor Hub BoosterPack[[4]](#footnote-4)*, for accessing a *Lidar Lite v2[[5]](#footnote-5)* and for triggering a flashlight.

Figure 7 shows the schematics of the extension cape described above. It can be seen, that both, the values of the sensor module and the Lidar module, are accessed through Inter-Integrated Circuit (I2C). To control the equipped flashlight its button is bridged using a Field-effect Transistor (FET) which is triggered by a General Purpose Input/Output (GPIO) pin.

  
*Figure 7: Schematics of the extension cape*

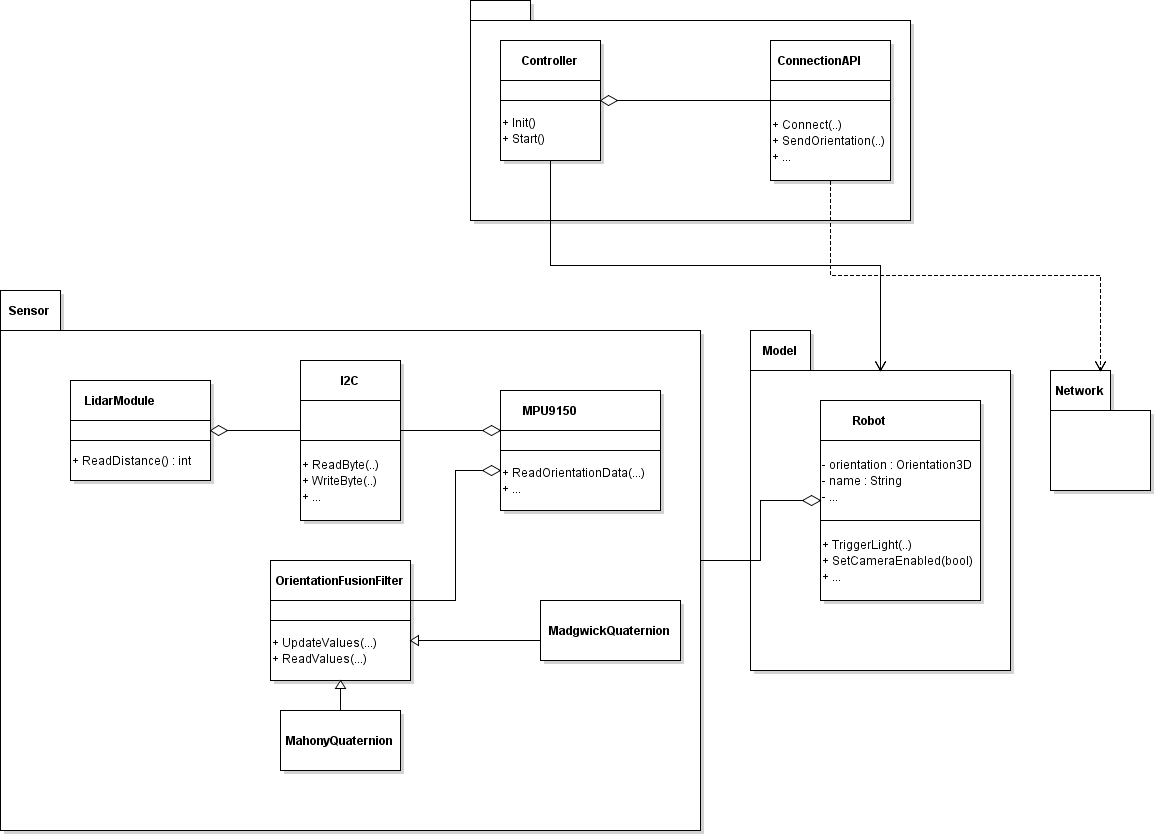
Additionally, the *Beaglebone Black*’s single USB port is connected to an externally powered USB hub, which is also powered by the power cape. The USB hub has the following devices connected:

1. *Microsoft LifeCam HD-2000* which supports hardware accelerated MJPEG compression
2. WIFI dongle with a Realtek RTL8188EU chipset

Having both above described devices connected exceeds the maximum rated power consumption of the *Beaglebone Black*’s USB port, thus an externally powered USB hub is mandatory.

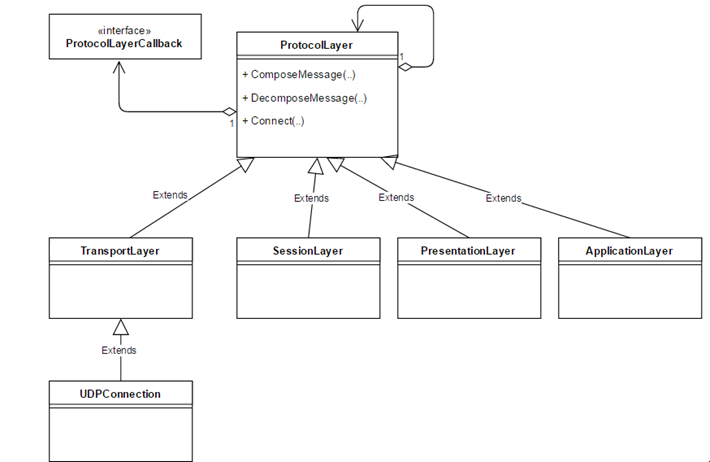
The robot’s controller application is written in *C++*. Figure 8 shows a simplified package diagram of the implemented software design. The main component is the implemented Controller, which starts a closed control-loop after a connection has been established to the *Server* successfully. If at any time the connection cannot be established or the connection is aborted because of some reason, the Controller implements an auto-reconnect functionality. The control-loop is responsible for fetching and sending the sensor values to the *Server* at a frequency of 20Hz. Additionally the control-loop uses the readings from the Lidar distance values to either act autonomously in some special cases and/or implements safety aspects, i.e. preventing the robot to hit a wall at full speed. The actual implemented behavior can be configured via the robot’s configuration file. The format and supported values of the configuration file is fully documented in chapter [Configuration](#h.f5vke9mgedpv).

The controller performs all actions using the classes ConnectionAPI, which is a facade implementation of the protocol stack. The ConnectionAPI triggers events in the Controller class which is registered in the ConnectionAPI class through an interface. The controller performs all operations in the model class Robot which has its own control thread to ensure consistent setting of the motor values and consistent reading and filtering of sensor values. To access the sensor values a re-usable library was written which consists of the classes MPU9150, which provides accessing methods to the MPU9150 IMU device, and the class LidarModule to access the Lidar module.



*Figure 8: Class diagram of the robot’s architecture*

The robot’s implementation of the protocol stack, described in chapter [Protocol Definition](#h.9d5e8gslm0yo), which is used by the facade class ConnectionAPI described above, was implemented as shown in the class diagram below. Similar to the implemented *NetworkLibrary* described in chapter [NetworkLibrary](#h.9wu8jynupa6), the session layer, the presentation layer and the application layer are implemented by extending from an abstract base class. Additionally, the transport layer is wrapped in the class UDPConnection, which allows to change the used transport protocol without affecting the upper layer’s implementation. The layers can be connected in the specific order at instantiation time. Listing 3 shows an example of connecting four layers. The code snippet specifically shows a unit test method to test the send method, using a mock implementation for the transport layer. It can be seen how each layer is instantiated by injecting each lower layer in the constructor.



*Figure 9: Design of the protocol stack implementation*

|  |
| --- |
| bool test\_classes\_layers() {  bool result = true;    TransportMock mock;  result = mock.Send("Test", 4) && result;  SessionLayer session(&mock);  result = session.Send("Test", 4) && result;  PresentationLayer presentation(&session);  result = presentation.Send("Test", 4) && result;  ApplicationLayer application(&presentation);  result = application.Send("Test", 4) && result;  return result;  } |

*Listing 3: Usage of the implemented network classes*

An important functionality of the robot is to provide a low-latency stream to the *Server* component which is then distributed to all connected clients.

Several approaches have been tested to provide the functionality of low-latency media streaming to the *Server*. These tests included the following:

1. Streaming using third-party programs, especially
   1. VLC[[6]](#footnote-6), the command cvlc respectively
   2. GStreamer[[7]](#footnote-7)
2. Directly accessing the webcam driver through *Video4Linux* and sending over a UDP connection

The tests have shown that the tested third-party programs do not have the ability to provide low-latency streaming, using the above described hardware. The best configuration for both, *VLC* and *GStreamer*, had a noticeably delay of approximately one second.  
Further tests have shown, that directly accessing the webcam driver and fetching MJPEG compressed images is reasonably fast and more promising in order to provide a low-latency stream. Furthermore, this approach also offers the possibility to fully control the behaviour of the stream, e.g. decreasing the frame rate and/or frame size according to the available bandwidth in a very flexible way.

The implementation of the media streaming was done according to the *MJPEG-4 Video Codec* standard, i.e. sending a fixed Huffman table at the beginning of each MJPEG compressed frame. Having a compliant implementation of this standard allows any streaming client to receive and interpret the stream.  
To minimize the latency of the implemented media streaming, several drawbacks have to be taken. The following enumeration summarizes the key characteristics of the implementation:

1. The minimal frame size which is supported by the hardware is used, i.e. 160x120 pixels at 16 Bit colour depth.
2. The fetched frame is compressed as much as possible, resulting in a compression level of approximately 70%
3. The media streaming fetches and sends 25 frames per second (fps)

Furthermore, the functionality of media streaming was implemented as a separate application, thus running as separate process. The main reason for this was to ensure that the control application is not affected by the media streaming and vice versa.  
The latency of the implemented media streaming was tested and analysed. The test results are shown in chapter [Performance Analysis](#h.jesj8whojsj5).

## *Android* App

The implemented *Android* application allows a user to control the robot. As this is one of the components which used by an end-user, the main focus was put on a user-friendly and good-looking user interface. Figure 10 shows some screenshots of the implemented *Android* application.

The start screen of the app consists of a paged introduction to the project *FHVrobot*. The introduction gives details on the project itself, the developer team and shows an animated tutorial for how to use the app. The user is then prompted to enter her or his name which is mandatory to connect to the *Server*.  
After the application has connected to the *Server*, the camera stream is shown to the user and several controls give the user the possibility to chat and control the robot. The robot’s orientation is shown in a compass view.

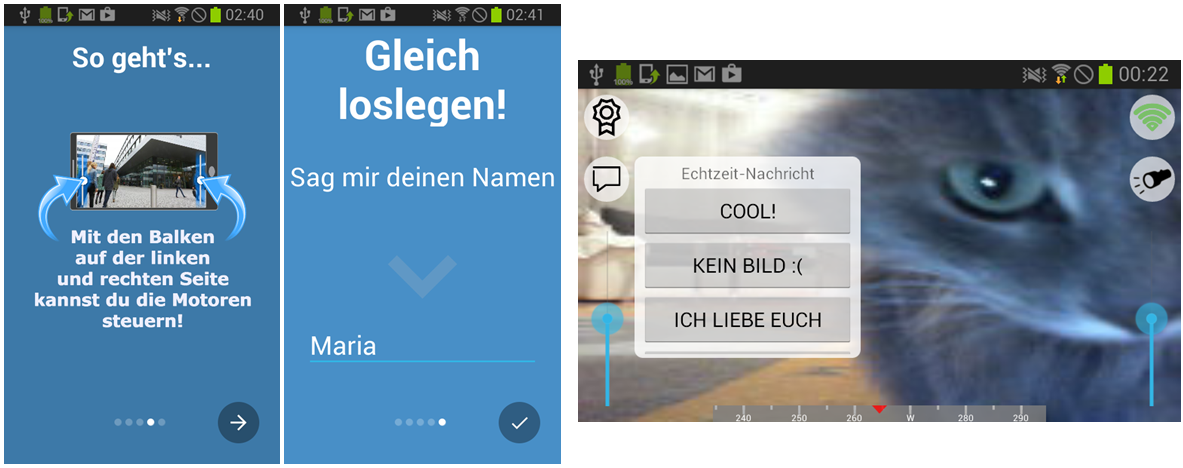
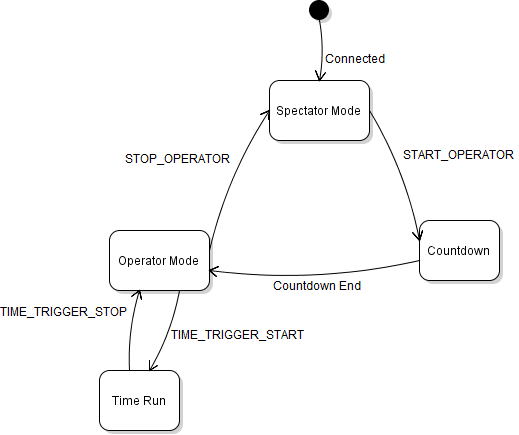


Figure 10: Screenshots of the implemented Android application

The *Android* app uses the *NetworkLibrary*, as described in chapter [NetworkLibrary](#h.9wu8jynupa6), to communicate with the *Server*. To allow full control of the app’s behaviour from the *Server*, the app’s implementation of the application layer follows the state chart shown in figure 11. After the app is connected successfully to the *Server* the GUI switches to the spectator mode, i.e. all controls for controlling the robot are hidden, only the camera stream and the chat menu is shown. If the app receives the command START\_OPERATOR the GUI switches to operator mode, i.e. all controls are shown and enabled. Between the transition of the spectator mode and the operator mode, a countdown is shown to the user to allow him or her to prepare properly.

Additionally, the app will show the timer of a parkour run if the command TIME\_TRIGGER\_START is received, respectively it stops the timer if the command TIME\_TRIGGER\_STOP is received. To ensure consistent timer values between the *Game Server* and the app, no time synchronisation in any way is done. However, it is assumed that the latency between the commands TIME\_TRIGGER\_START and TIME\_TRIGGER\_STOP is approximately the same, therefore it can be assumed that in most scenarios the difference between the measured times is reasonably low. Additionally, as payload of a command TIME\_TRIGGER\_STOP the actual time of the parkour run is sent, which will then the app show as final time. System tests have shown, that a user is not able to spot any noticeable difference between the *Game Server*’s timer value and the app’s timer value while the timer is running. Furthermore the user is not able to see the transition from replacing the app’s timer value with the actual timer value received with the TIME\_TRIGGER\_STOP command.



*Figure 11: State chart of the app client*

In addition to the state chart above, in any state unexpected connection losses can occur. If the connection to the *Server* is lost, the app shows a reconnect view, which tries to reconnect after a delay of ten seconds. This dialog can also be dismissed by the user. If the reconnect is successful the app again switches to the spectator mode.

## *Game Server*

The *Game Server* is responsible for implementing the game logic which is described in [Game Concept](#h.43hwr04tt0y6). The *Game Server* component is written in *C#* using Windows Presentation Foundation (WPF) as GUI framework. Figure 12 shows a screenshot of the main view of the *Game Server*. The following views are shown:

1. In the upper part of the view, the current operator’s name and best time of her or his parkour run is shown
2. In the center area the time of the current parkour run can be seen
3. In the lower part of the view the current high-score list is shown.

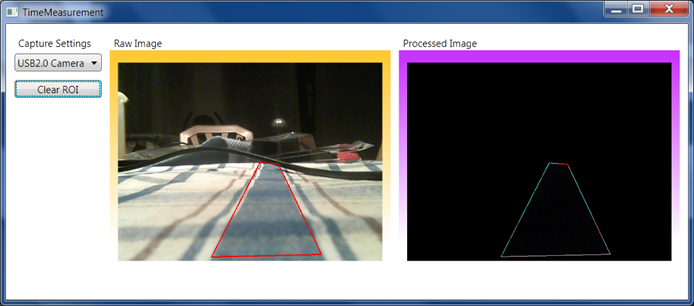


*Figure 12: Screenshot of the Game Server’s main view*

Since the *Game Server* is written in *C#* the implemented *NetworkLibrary* cannot be reused. However, as the *Game Server* does not have a persistent connection to the *Server*, instead only using the service interface of the *Server*, the implementation of the network stack is not that extensive. Non-persistent connections do not need to request a session ID or send and handle heartbeats. The implemented service interface can be considered to be similar to a Hyper Text Transfer Protocol (HTTP) 1.0 request, i.e. opening a connection, sending a request, getting a response and closing the connection. However, since we do not use a TCP connection, service calls could possibly fail. However, this case is not handled explicitly, the *Game Server* rather repeats all its requests in frequent service calls.  
The concept of frequently repeated service calls is necessary anyway, since the *Server* could change its state unexpectedly, e.g. the operator can change at any time, without the *Game Server* noticing it. This would lead to inconsistent states between the *Server* and the *Game Server*, which has to be addresses using the approach described above.

The *Game Server*’s main responsibility is to do the time measuring of a parkour run. To trigger the start and finish event, a camera-based trigger system was implemented. The basic idea behind this system is to fetch camera frames with a high frequency, i.e. 30 fps, and calculate the difference between two consecutive frames. If the difference between two frames is reasonably big, a trigger is raised.

To ensure suitable precision of the time trigger event, a specific Region of Interest (ROI) can be defined. Figure 13 shows a screenshot of the configuration window for the time measurement system. On the left hand side, the original image is shown. The defined ROI is highlighted in red. On the right hand side, the difference between the current and previous image is shown, whereby a completely black image means that no relevant difference between the last two frames was observed.



*Figure 13: Screenshot of the configuration window of the camera trigger system*

Besides the above described technique, the time measurement system’s behaviour was improved by various additional approaches:

1. Calculating the difference not only between two consecutive images, but four. This leads to less false positive triggers if the ambient light flickers.
2. Taking the average of the last 100 images, which is roughly equivalent to the last three to four seconds, as reference for comparison. This ensures less false positives if one tries to trigger the system by quickly moving the hand in front of the camera.

Tests have shown that the implemented camera trigger system is very stable in almost all environments. In all tests which were performed, no false positives were observed and only in some cases the system failed to detect a valid event.

# Protocol Definition

For the communication between the *Server* and clients, either app clients or robot clients, a protocol was defined. The protocol is defined for the session layer, the presentation layer and the application layer. The following sections describe each layer separately and in detail.

## Session Layer

Figure 14 shows the defined protocol for the session layer. The session layer’s header consists of two bytes. The first byte consists of several flags which can be set. The second byte represents a unique session ID for the connection. The flags which can be used in the first byte are shown in figure 15. The two version bits are not yet used, therefore, they are set to 0 in the current implementations. The defined request bit is used to indicate a session request by the client. If the session request bit is set the payload is ignored and is not being delegated to the upper layer. In any other packets, beside session requests, this bit has to be set to 0.

Flags

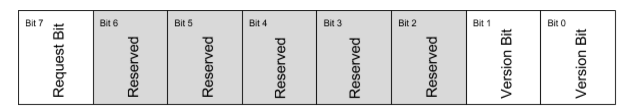
Session ID

Payload

1 Byte

1 Byte

*Figure 14: Defined protocol for the session layer*

Figure 15: Defined flags for the session layer

## Presentation Layer

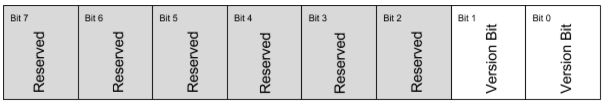
Figure 16 shows the defined protocol for the presentation layer. The header of the presentation layer consists of one byte which contains various flags. The flags are defined in figure 17. The defined version bits are not yet defined and are set to 0 in the current implementations of the presentation layer. At the moment the presentation layer has not implemented any checksum or encryption.

Flags

Payload

1 Byte

*Figure 16: Defined protocol for the session layer*



*Figure 17: Defined flags for the presentation layer*

## Application Layer

Figure 18 shows the defined protocol for the application layer. The header of the packet contains one byte for various flags, one byte for a command ID and one byte specifying the length of the following payload. If the packet does not contain a payload this byte has to be set to 0.

Flags

Command ID

Payload

1 Byte

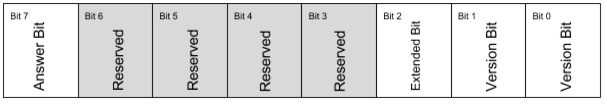
1 Byte

Length

1 Byte

*Figure 18: Defined protocol for the application layer*

The following figure shows the defined flags for the application layer. The defined *Version Bits* are not yet used, therefore, they are set to 0 in all implementations. The defined *Answer* *Bit* can be used to indicate that the sent packet contains an answer to a preceding request.



*Figure 19: Defined flags for the application layer*

If the *Extended Bit* is set in the header of a packet, the packet structure differs from the defined packet structure shown in figure 18. Figure 19 shows the packet structure if the *Extended Bit* is set. The *Extended* *Bit* allows to send packets with a longer payload than 255 bytes, i.e. an additional byte is used to define the length of the packet’s payload, thus allowing a theoretical maximum payload length of 65535 bytes. However, in practice the actual payload of a packet is limited by transport layer.

The *Extended* *Bit* is used to send the high-score data from the *Game Server* to the *Server*, which is then delegated to app clients. The high-score data is transmitted using the command IDs 60 (PERSIST\_DATA) and 61 (REQ\_PERSIST\_DATA).

Flags

Command ID

Payload

1 Byte

1 Byte

Length  
MSB

1 Byte

Length  
LSB

1 Byte

*Figure 18: Packet structure if Extended Bit is set*

The above described command ID defines the specific action which should be executed on the receiving machine. The following table gives a complete overview over the implemented commands and further describes the respective command:

|  |  |  |
| --- | --- | --- |
| **ID** | **Human readable name** | **Description** |
| 1 | CHANGE\_NAME | This command is sent from the app clients and the robot to the Server. The payload of this command contains the actual name. The Server responds with the *Answer Flag* set to 1 and an empty payload, if the name was accepted. The payload of the response contains an alternative name, if the name is already in use. |
| 2 | GENERAL\_MESSAGE | This command can be sent from either component and is used to send human readable messages between two components. |
| 3 | FORWARD\_MESSAGE | This command can be sent from a client to the *Server*. The *Server* will forward the given message over a bridged connection to another client. |
| 11 | DRIVE\_LEFT | This command can be sent to a robot. The robot will set the motor power to the given value in the payload. The payload is a single signed byte value. |
| 12 | DRIVE\_RIGHT | This command is equivalent to the above described command, but setting the motor power of the right motor. |
| 13 | DRIVE\_BOTH | This command sets both motors of the robot to the same value. |
| 18 | TRIGGER\_LED | This command is sent to a robot, which will trigger the flashlight. |
| 20 | CAMERA\_ON | This command is sent to a robot, which will then turn the camera stream on. The payload of this command specifies the receiver’s ip address and port. |
| 21 | CAMERA\_OFF | This command will inform the robot to stop the camera stream. |
| 30 | REQUEST\_OPERATOR | This command can be sent to the *Server*, which will respond with an answer packet with the *Answer Flag* set and the payload containing the name of the current operating app client. |
| 40 | TIME\_STARTED | This command can be sent to an app client, which will then show a stopwatch in the GUI. |
| 41 | TIME\_STOP | This command can be sent to an app client, which will then stop the shown stopwatch. The payload contains the measured time by the trigger system. |
| 41 | TIME\_DISMISSED | This command can be sent to an app client, which will then dismiss the stopwatch silently. |
| 50 | ROBO\_STEERING | This command is sent to an app client. The command will change the app’s state to operator mode. |
| 51 | ROBO\_NOT\_STEERING | This command reverts the state of an app client to spectator mode. |
| 60 | PERSIST\_DATA | This command can be sent to the *Server*, which will then persist the given payload in a data container. |
| 61 | REQ\_PERSIST\_DATA | This command can be sent to the *Server*, which then responds with an answer packet having the *Answer Flag* set and the payload containing the previously persisted data. |
| 72 | ORIENTATION\_DATA | This command is sent from the robot to the *Server* which contains three pairs of 2 Byte values, whereby the MSB precedes the LSB, containing the orientation data for 3-axis. |
| 77 | REQ\_DISCONNECT | This command informs the receiver that the sender is going to disconnect. |
| 78 | DISCONNECTED | This command informs the receiver that it should abort the connection. |

# Communication Analysis

The following chapter addresses the analysis of the communication between the components. First, the communication is analysed regarding its security aspects, i.e. possible security issues in the defined protocol and implementation.

Second, the performance aspects of the communication is analysed. This includes analysis of packet sizes, respectively used bandwidth and analysis of round trip times.

## Security Aspects

In chapter [Protocol Definition](#h.9d5e8gslm0yo) the implemented protocol is described in detail. Analysing the characteristics of the protocol shows, that several attacks can be performed on the communication between a client, either a robot or an app client, and the *Server*.

A major security issue is the low number of available session IDs. According to the protocol definition only one byte is used for the session ID, thus allowing simple Denial of Service (DoS) attacks on the system. An attacker simply could send several session request packets to the *Server*. The *Server* will respond with valid session IDs until all 255 session IDs are assigned. It will then reject any succeeding session request. This issue could be addressed in two ways:

1. Increasing the size of the session ID to at least four bytes. This will increase the number of available session IDs to .
2. Rejecting session requests from the same IP address in a short time interval.

Another exploit on the system could be to hijack a session and take over the control of the robot. This could also be a critical safety issue, as the robot could be either intentionally be damaged by the attacker or it could even be stolen. Although session hijacking is secured in a trivial way, i.e. preventing session hijacking from a different IP address, spoofing the IP address of the other client, will bypass this. A better approach to prevent session hijacking using IP spoofing would be to asymmetrically encrypt the messages. However, implementing this approach would presumably lead to a lower overall performance of the communication, which is not satisfactory in the context of controlling a robot.

An additional security issue is the possibility of performing a replay attack on the system, including all OSI layers. Although, this will not work in any case, as the replayed session request will probably result in a different session ID as in the recorded traffic, this fact can easily bypassed by the attacker. One way to prevent replay attacks on the system would be to extend the protocol by an incrementing packet number.

## Performance Analysis

The below performance analysis is based on several test runs and was performed with the following test environment:

1. All components were connected to the same local network
2. The used local network was the *eduroam* network of the *University of Applied Sciences Vorarlberg* on a typical week-day
3. All components were set-up in the same room. No walls were between the devices
4. The *Server* and the *Game Server* were running on different machines
5. Two app clients were connected to the *Server*
6. One robot was connected to the *Server*

The following test runs were performed:

1. Sending approximately 1000 packets from the robot’s application layer to the *Server*’s application layer, which immediately sent an answer to the robot
2. Sending approximately 1000 packets from one of the app client’s application layer to the *Server*’s application layer, which immediately sent an answer to the app
3. Starting the stream on the robot and delegating the stream to two different app clients

The time measurement of the round trip time of the test runs 1. and 2. started immediately before sending a packet and stopped immediately after receiving the answer packet. The results were then printed to the console, respectively the debug output of *Android*. The latency of the media streaming, according to test run 3., was measured between the robot and one of the two app clients by capturing a photo of a stopwatch and the phone’s screen which displayed the stream.

The following table shows the results of the test runs:

|  |  |  |  |
| --- | --- | --- | --- |
| Test Run | Replications | Average | Std. Deviation |
| 1 | >1000 | 10ms | 3ms |
| 2 | >1000 | 8ms | 2ms |
| 3 | 2 | ~250ms | - |

Above results show, that the performance of the control packets is reasonably good, however, the measured latency of the media streaming shows a noticeable delay. One reason for the poor performance of the media streaming is the lack of hardware accelerated H.264[[8]](#footnote-8) compression by the used webcam. Compressing using H.264 would result in significantly lower frame sizes and therefore lower latency of media streaming.

# Quick Start

This chapter provides detailed information about how to install, configure and run the applications of *FHVrobot.* This guide should help to set-up and start *FHVrobot* at *FHVoffen* very quickly.

## Installation and Setup

The following sub-sections describe the prerequisites and/or other requirements to run the components needed for *FHVrobot*.

### Robot Prerequisites

In order to run the robot’s application properly some modules have to be installed on the *Beaglebone Black* first. The following enumeration lists those modules:

1. *Video4Linux* driver to access the webcam  
   sudo apt-get install v4l-utils
2. Drivers for the *RTL8188EU* chipset. To install those drivers a script was implemented. To run this script the following command has to be executed:  
   ./build\_realtek\_driver.sh
3. Setup the WIFI connection using the command wpa\_supplicant
4. Copy the application executables fhvrobot and fhvrobot\_streaming onto the *Beaglebone Black*

### *Server* Prerequisites

In order to run the *Server* properly *Java8* has to be installed properly. Additionally, if the *Server* is not in the same local network as the client devices, the *Server* needs a public accessible IP address.

The executable of the *Server* application is RobotServer.jar and has to be copied to a local folder on the machine.

### *Game Server* Prerequisites

The *Game Server*’s application is written in *C#*, exclusively for Windows systems and was tested on Microsoft Windows 7, 8 and 10.

Additionally, the *Game Server* needs an external webcam which can be used as trigger camera for the time measurement. Preferably, the machine on which the *Game Server* runs should have a second screen connected. On one screen the main view can be shown and on the other screen the *Game Server* can be controlled and configured, e.g. specifying the ROI of the trigger system.

### App Prerequisites

The App was written for *Android* phones only. The App only runs on *Android 4.0* and above. The device needs a reasonably fast internet connection or has to be at least in the same local network as the *Server*.

## Configuration

All the components are designed so that the effort to configure each component is as low as possible. The following sections describe how each component can be configured:

### *Server*

The *Server* can not specifically configured. However, it is important to get the IP address of the machine running the *Server* in order to configure the other components.

The IP address can be determined using the following commands:

For Windows systems, open a command shell and type in the command ipconfig. On Linux systems the command ifconfig shows the associated IP addresses of a network interface.

### *Game Server*

On startup the *Game Server* prompts the user to enter the *Server’*s IP address. At runtime the *Game Server* can be configured using the settings window, which opens automatically on startup. If the settings window is closed unintentionally, the keyboard shortcut ALT+S reopens the window. The settings window should is self-explanatory.

### *Android* App

The *Android* app can be configured regarding the *Server*’s IP address. To set the IP address a hidden dialog can be opened by double clicking on the label “Los geht’s” on the last page of the app’s intro.

### Robot

The robot’s control application has several setting options. To configure the robot’s behaviour the control application tries to open a config file config.ini on startup. If this file is not present it can simply be created by entering the command touch config.ini and afterwards in can be edited by vi config.ini.The config.ini’s syntax is described in the following listing:

|  |
| --- |
| [server]  address=127.0.0.1 ; Use your server's ip address    [robot]  name=FHVrobot ; Robot's name    [motor]  factor=100 ; Factor to set max power of motor (0, 100]    [lidar]  enable=false ; Enable/disable usage of LIDAR  christ-kindle-mode=false ; Enable/disable christ kindle mode  block-forward=false ; Block forward movement at 20cm |

*Listing 4: Syntax of the config.ini file*

The configuration file allows to:

1. Specifying the *Server*’s IP address
2. Setting the robot’s name
3. Specifying the maximum motor power in percent
4. Enabling and disabling the Lidar sensor
   1. Using the *christ-kindle mode* which enables the robot to automatically drive backward if it detects an obstacle in front of it
   2. Using the *block-forward mode* which prevents the operator of the robot from further running forward if

## Troubleshooting

In the following, some frequently occurring problems are described and possible solutions are presented:

**The media streaming does not work**

Sometimes the process for media streaming is not started properly. If this is the case just stop and start the media streaming again. If the webcam’s status LED is turning on, the media streaming should work. If the camera image is still not visible on the *Server* or App, it is most likely that the bandwidth of the robot’s WIFI connection is to slow.

**The motors of the robot cannot be controlled**

If the motors cannot be controller from an app client, one should first ensure that the motor can be controller from the *Server* application. If this approach fails, it is most likely that either the batteries of the robot are low or some cables are not plugged in properly.

**The app cannot connect to the *Server***

In order to connect the app to the *Server*, the app has to be configured properly, i.e. the correct IP address has to be used. If the app still fails to connect, it is most likely a network problem at the client or at the *Server*.

**The robot does not automatically connect to the *Server***

If the robot does not connect to the *Server* on startup, one of the following is causing this issue:

1. The robot failed to connect to the WIFI:  
   Use a USB cable to connect to the *Beaglebone Black* and ensure its connectivity.
2. The robot is not properly configured:  
   Set the correct IP address of the *Server* in the config.ini file on the robot.
3. Something failed at startup:  
   Restart the robot and try again.

**Neither the app nor the robot can connect to the *Server***

If this happens, it is most likely that the *Server* was not given the permission to open the needed ports in the firewall of the machine. On Windows systems the firewalls usually asks for permission if the *Server* should be granted the permissions to open the network connections. This dialog has to be accepted.

1. http://www.dfrobot.com/index.php?route=product/product&product\_id=65#.VsDdT\_nhDBQ [↑](#footnote-ref-1)
2. http://boardzoo.com/index.php/beaglebone/power-supply-cape.html#.VsDEl\_nhDBQ [↑](#footnote-ref-2)
3. http://exadler.blogspot.co.at/ [↑](#footnote-ref-3)
4. http://www.ti.com/tool/boostxl-senshub [↑](#footnote-ref-4)
5. http://pulsedlight3d.com/products/lidar-lite-v2-blue-label.html [↑](#footnote-ref-5)
6. http://www.videolan.org/vlc/ [↑](#footnote-ref-6)
7. http://gstreamer.freedesktop.org/ [↑](#footnote-ref-7)
8. https://en.wikipedia.org/wiki/H.264/MPEG-4\_AVC [↑](#footnote-ref-8)