Technical Report

April 28, 2014

Anders Strand
Emil Taylor Bye
Petter S. Storvik
Odd M. Trondrud
Ole Bauck

Eksperter i Team
Norges Teknisk-Naturvitenskapelige Universitet

1 Abstract

The abstract is a tl;dr-type thing that *very* briefly summarizes the introduction, method, result and conclusion.

This paper presents our technical project for the subject TMM4850, “Instrumentation and control over the Internet”. The subject is a part of the NTNU course “Experts in Team”, where students from different academic backgrounds join together in a semester-long techical project. We developed a secure, efficient and scaleable system for sending data and commands between various nodes. To demonstrate the system, we built a robot which we controlled trough a server using a web user interface. The system proved to be effective and reliable, and a good platform for further expansion and developement.

1 Contents

2 Introduction

Our group was assigned to the "Instrumentering og Styring over nett" village, which has a focus on remote control over the internet. The tackled problem can be summarized as follows:

> Construct a platform that facilitates instrumentation and remote control over the internet. Demonstrate its feasability by remote controlling a car-like robot with sensors, a camera and actuators. The platform should be modular and security should be focused on during the development process.

We decided to build a platform to facilitate the communication between operators and the devices they wish to remote control over the internet. Devices connected to the platform should have their functionality, available commands and sensory readings made available to authorized users through the platform. The platform should be modular in order to more easily allow the adding of functionality to meet domain specific requirements. It should also meet typical security requirements: Eavesdropping on the communication should not be possible and some secure authorization scheme should be supported in order to restrict access to devices connected to the platform.

Various rescue services often encounter situations where assertaining the risk of entering a location is difficult. A burning house is an example of this, especially if the only information available is what the eye can see and one does not know if there are people trapped inside or not. Another example of such a situation is cave exploration, both above and under water. Rescuing divers is both difficult and time-consuming. A remote controlled device (e.g. such as a wheeled robot) could be sent instead of a live human, reducing the risk of loss of life. While various systems exist that are designed to serve the purpose of rescue operators' "eyes and ears", these are typically domain or application specific. If the same communication platform could be used for any given scenario, one forces the developers to modularize their systems, potentially resulting in an increase of re-usability.

Communication will be done over HTTP, with the possible messages specified by an API. The platform will function as a web-service with a RESTful API. This means users will be able to access the system through a regular web browser when connected to a WLAN or 3G network. In theory the platform will allow any internet capable device connected to the internet to be remote controlled by anyone with an internet connection anywhere in the world, given that the necessary software to communicate with the platform has been written.

We also showcase the feasibility of such a system by using it to remote control a robot with four wheels, a three-jointed grabber, sensors and a camera. The development of this robot and the software required to control it through the system is also detailed.

3 Concept Description

The project's goal is the construction of a platform that facilitates the communication and authorization aspects of remote controlling a device over the internet. Agents connect to the platform through which their sensory information, available commands and functions are made available to authorized users. Use of the platform increases the possible simultaneous audience of a connected agent, as users do not have to query the agent directly for the information. Agents actively query the platform to obtain commands from users. In the same manner, users must actively query the platform to obtain new information from or about the agents. The frequency of such queries are decided by the specific client. See Figure 3.1 for an illustration of the flow of information between the platform and connected entities.

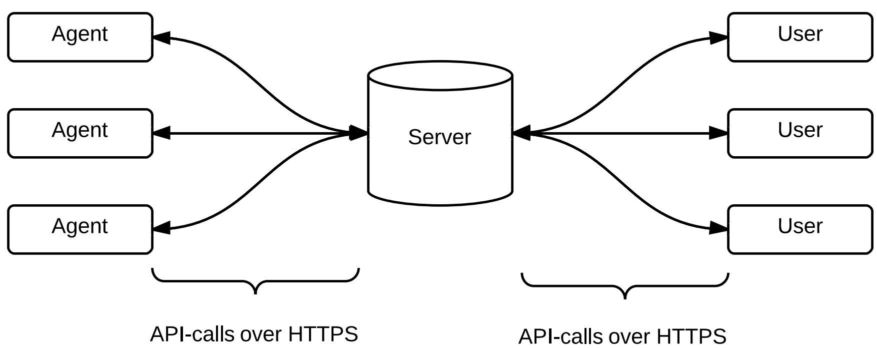

Figure 3.1: An overview of the concept showing three users and three agents communicating through the platform.

The platform itself only responds to queries from connected entities and performs user identification and authentication if needed. When a user sends a command to an agent, the command is received by the server and is stored away until the agent queries the platform for commands directed at it. When an agent uploads its sensor data or whatever other information this is stored on the platform and users must actively query the platform for it.

Agents and users access the platform through a client, which is a piece of software that communicates with the platform. On the user side, the client communicates with the platform through API calls and presents the information it receives to the user. On the agent side, the client communicates with the platform through API calls, and translates (if required) and passes this information on to the device that is to be remote controlled. Clients are not a part of the platform itself, and the only requirement is that they are able to communicate with the platform through the API.

3.1 Terminology

Below follows an explanation of the various words that refer to different parts of the system.

Server the physical computer or cluster of computers on which the service is hosted.

Service the program that facilitates the communication between clients and manages user authentication.

Platform see Service.

Client any software that sends requests to the service and receives the responses.

Agent any entity that is connected to the service that can receive commands through the service. The term "agent" refers to the entire entity, which is considered to begin where the requests to the service are generated and end wherever the commands are acted out. A robot, its actuators and sensors, the software that communicates with the service, and the computer that is connected to the robot which the software runs on, and any other parts or components are considered part of the agent.

Agent client a client that takes care of the communication with the service as a part of an Agent.

User any non-agent entity that interacts with the service. This includes actual people and automated services – anything that interacts with the service but cannot receive commands through it.

User client any client through which users access the service.

4 SWOT

SWOT analysis is one of the most commonly used marketing strategies. This analysis identifies both strengths and opportunities, and more important weaknesses and threats.[1] It's also important to use the information gathered in the SWOT analys to form project objectives and long term goals. The following SWOT analysis will be brief and will not go into detail.

Figure 4.1: SWOT

Strengths

- Everything is built on already existing technologies
- Module based, which means easy to expand
- Relatively cheap harware and free software

Weaknesses

- Latency caused by http limitations
- Limited to the dynamixel servos (possible to support more manufacturers)
- No team to take the project further

[1] http://www.businessnewsdaily.com/4245-swot-analysis.html

Opportunities

- No universal platform (that we could find) exists
- Many different solutions have been tried, but no module based and easy to develope

Threats

- Existing technologies doesnt allow easy expansion, but there are many good soluitions for different specific tasks
- Marketing problems
- End of EiT

As described above the project has a relatively high degree of innovation. This means that there are no similar systems which are based on the same simple principles as this project. This may be a result of manufacturers often tends to overcomplicate things. Its weaknesses are not significant as they can be fixed with a small amount of invested time. The one thing really standing in the way of this project is the end of EiT, and the fact that the group is being dissolved.

5 Method

5.1 Robot

To demonstrate the platform it was decided to make a vehicle with a manipulator controlled remotely over internet. In addition the robot should have sensors that send data (if available) continuously to the server and be able to send camera feed to the operator. Due to the possibility to expand this later with more complex functionality, the program was written in C++. The following was used to make this vehicle:

- Raspberry Pi
- Raspberry Pi Camera
- Dynamixel AX-12 Servomotors
- Dynamixel AX-S1 Integrated Sensor
- USB2Dynamixel
- Dynamixel SDK

Raspberry Pi (Pi) is a single board credit-card-sized computer. It runs on an 700 MHz ARM processor. It has two USB inputs, ethernet, HDMI and gpio (general purpose input output) headers. This makes it the perfect prototyping computer for this kind of project. The Pi is running its operating system from a SD card, and the OS is RaspBian which is a debian based linux distro.

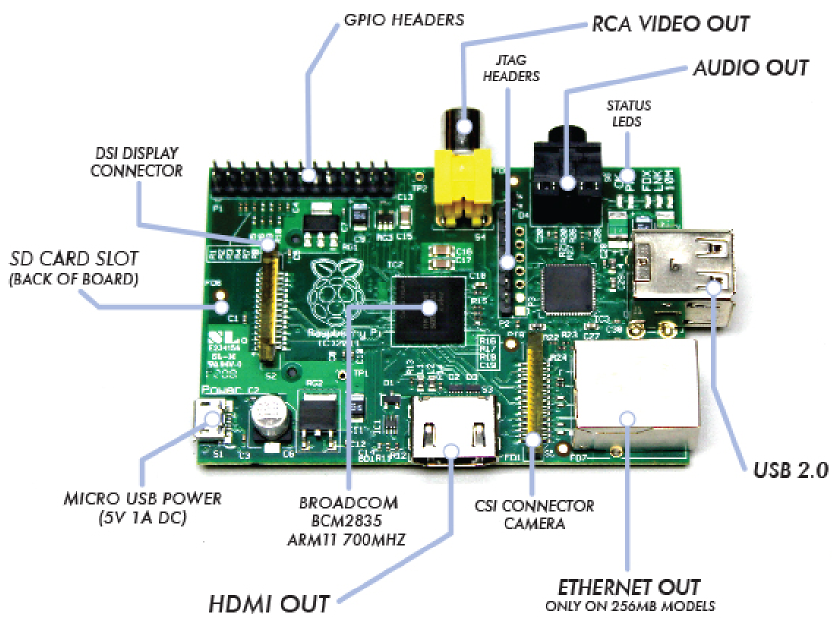

Figure 5.1: Raspberry Pi overview over peripherals

To control the Pi you can hook up a keyboard and a monitor directly to the board, or you can control it from another computer. This is achieved by using a SSH server on the Pi and a SSH client on the computer. SSH (Secure SHell) is a protocol that allows one computer to remotely controll another via command line. A widely known program for doing this is PuTTY. To login over SSH, you need to know the IP-address of the host. This can be done by setting the IP-address static or to scan the network to find out which IP the Pi would have. Since both of these approaches are difficult on a big network like NTNU, we had to find another approach. The solution was to connect the Pi directly to the computer via an ethernet cable. To do this we had to do some modifications, the full tutorial on how to do this see [1]. To get internet access we first shared the Wi-Fi connection on the computer described here [2]. Later we used a Wi-Fi usb dongle. The driver for this dongle installed automatically and all we had to do was to configure */etc/network/interfaces*, and add the following:

```
auto wlan0
iface wlan0 inet dhcp
wpa-ssid "<nameOfNetwork>"
```

[1] http://pihw.wordpress.com/guides/direct-network-connection/

[2] http://anwaarullah.wordpress.com/2013/08/12/sharing-wifi-internet-connection-with-raspberry-pi-through-lanethe

```
wpa-psk "<networkPassword>"
```

We had to set up a own network to do this, because the eduroam network uses different setup.

Raspberry Pi camera

Sven: si noe om kamera, og hvorfor det er så bra at det ikke går på usb, nevn litt om seriell kommunikasjon i samme slengen.

Dynamixel AX-12 servomotors are motors that allows for precise controll of angle and velocity. These motors are controlled over a half duplex UART, which is a byte oriented asynchronous serial communication protocol. You can control the motors and receive feedback by sending commands to it corresponding to the control table in the datasheet [PUT REFZ H3R3]. The most important features is the control of position and velocity. The servos can work like normal servos where you can put in the desired position, and a controller inside the servo will make the servo go to that position. This position is limited to between 0-300 degrees (see datasheet). The servo can also work in so called "Endless Turn" mode, where the servos can spin infinite. Here you can only control the velocity which the motors run. "Endless Turn" mode is activated by setting the angle limits (CW Angle Limit and CCW Angle Limit) to zero.

Dynamixel AX-S1 Integrated Sensor is a sensor device capable of measuring sound, brightness, heat and distance to objects. It is also capable of making sound. The communication is the same as the servomotors and the sensor is connected to the same bus.

USB2Dynamixel [3] is a USB device that allows the computer to create a virtual serial port (UART) and with some other circuitry, communicate with the servos over the USB port. This USB requires no driver installation when running linux, and since RaspBian is a linux distro this simplifies things. The USB2Dynamixel is inserted into one of the USB ports on the Pi and the servor motors are connected to the device.

Since the Pi also has a hardware UART driver on two of its gpio headers, we thought about if we could communicate with the servos through these pins. To do this we had to make the circuitry described on page 8 in the datasheet [PUT REFZ H3R3]. We would also have to implement our own code for the lower part of the communication (where we used a library from the manufacturer, more on that later). UART is also widely supported by many lower end microcontrollers which don't have the support for an OS. Therefore using UART would mean that the code would be even more platform independent.

Dynamixel SDK is a programming library for controlling dynamixel servo motors. This library is available for Windows, Mac and Linux and easy to run on the Pi. Since the library is written in C it is easy to use it in this C++ project. There is a great API reference [4] with the library. The

[3] http://support.robotis.com/en/product/auxdevice/interface/usb2dxl_manual.htm

[4] http://support.robotis.com/en/software/dynamixelsdk.htm

functions treat the lower part of the communication over USB2Dynamixel. There are functions for initializing the communication, terminate the communication, sending byte and words (16 bit), receiving byte and words, and for ping. Ping is for checking if a device is connected. There is also a page describing platform porting [5]

5.1.1 The overall system

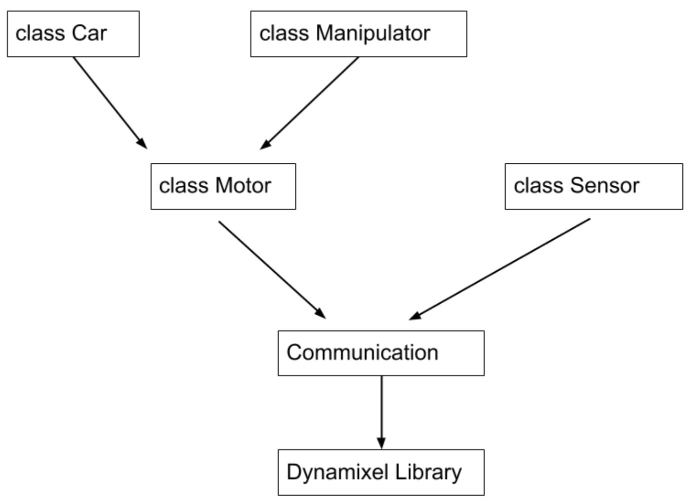

Figure 5.2: Robot overall system

This is how the overall system was made. It was our intention to make the system module based, such that it is easy to develop more things using the same modules. To make the modules we implemented classes in c++. That way you can make several motor, sensor, car or manipulator objects.

The communication module was added because we used threads in our program. Since there are only one data-bus and therefore it can only be accessed by one thread, we had to implement *Mutexes* such that only one thread can access the communication functions available in the dynamixel library at the time. *Mutexes* are "locks" that protect such code blocks that can only be accessed by one at the time.

[5] http://support.robotis.com/en/software/dynamixel_sdk/sourcestructure.htm

5.1.2 Wiggle It

The first step is to make a servo motor move.

Figure 5.3: Servo motor test setup

To make the servo motor move the example-program for the motor class could be used [6]. Alternative the example-file readWrite form the dynamixel library could be used [7]. For this to work there are some important parameters which should be known:

- ID - Each motor has its own ID. This way its possible to control which motor that should get the command. The ID need to be change in the code to the ID of the motor. If the motor ID is unknown, you could use the *pingAll()* function. This will search for IDs on the bus, and print out those that are active
- Port - Which port the USB2Dynamixel is connected to. This is used to set up the communication to the motors. USB devices under linux can be found under */dev/usb*. When testing we used deviceIndex = 0 (*/dev/usb0*).
- Baudnumber - used to set the speed of the bus. This should always be one, which is 1Mbit/s, unless lower speeds are needed.

By replacing these parameters in the example file the servos should move back and forth when enter is pressed.

5.1.3 Driving

The next step is to make four wheels cooperate! This was implemented in the class Car. Four servomotors are mounted under the base of the robot. The two motors on the right turns the same way and the two on the left turns the other way. When creating a car object all you have to do is write the IDs of the wheels. The motors are initialized in endless turn mode when the car object is created. There was functions for setting speed and turning the car both when the car is driving and when it had stopped. Turning the car was accomplished by setting the two right wheels one way, and the two left wheels the other way when the car had stopped. When the car was moving only one side was set to a lower speed to make the car turn while moving.

Figure 5.4: car overview

[6] LOCATION TO MOTOR EXAMPLE PROGRAM
[7] LOCATION TO READWRITE EXAMPLE FILE

In the example directory is an example program which demonstrates the car driving forward, backward and turning to left and right. The only thing needed are the IDs of the wheels. You can still use the pingAll() to check for IDs on the data-bus.

5.1.4 Manipulator

A manipulator was added to be able to pick up things and look around with the camera. The manipulator use three servomotors in the arm, and two in the gripper. The setup was like in the drawing below.

Figure 5.5: manipulator overview

The class Manipulator implements the arm and the gripper.

Arm

The angles of the servomotors in the arm can be set directly by using the function setAngles(θ_1, θ_2, θ_3), or you can set the desired position in x, y and z, by using the function goToPosition(x,y,z). This function make use of inverse kinematic, with a geometric approach. The problem can be split up in to calculations, the calculation of θ_2 and θ_3, and the calculation of θ_1. To calculate θ_2 and θ_3 we use the law of cosines.

Figure 5.6: manipulator from the side

$$\begin{aligned}
-2d_2d_3\cos(\pi-\theta_3) &= z_c^2 + r^2 - d_2^2 - d_3^2 \\
r^2 &= x_c^2 + y_c^2 \\
\cos(\theta_3) &= \frac{z_c^2 + x_c^2 + y_c^2 - d_2^2 - d_3^2}{2d_2d_3} = D \\
\sin(\theta_3) &= \pm\sqrt{1-D^2} \\
\theta_3 &= \arctan\left(\frac{\pm\sqrt{1-D^2}}{D}\right)
\end{aligned}$$

Figure 5.7: manipulator from the side

$$\theta_2 = \frac{\pi}{2} - \alpha - \phi$$
$$\theta_2 = \frac{\pi}{2} - \arctan(\frac{d_3 \sin(\theta_3)}{d_2 + d_3 \cos(\theta_3)}) - \arctan(\frac{z_c}{r})$$

Hence there are two solutions to the equation of θ_3, one where θ_3 is negative and one where θ_3 is positive. The two solutions are called *elbow up* and *elbow down*. We chose the *elbow up* solution because this would lead to less crashing with objects on the ground. The length d_2 and d_3 had to be measured in mm and set in the code.

Figure 5.8: manipulator from the top

The calculation of θ_1 was simple.

$$\theta_1 = \arctan(\frac{x_c}{y_c})$$

Since the servomotors only can move inside 0-300 degrees, the code had to include saturation limits on the angles, and will an error code if these limits are reached.

Gripper

The gripper used to servomotors on the end of the arm. The main problem with the gripper was to make it stop when it noticed it couldn't move further, and therefore not squeezing the object it had to grip. This was done by first setting the zero position where the two parts of the gripper could touch, and setting this as the first goal position. Then it had to read the position continuous until it noticed that it had reach the max number of consecutive reads. When this number was reached, the motors would stop ensuring that the object would not be destroyed.

Figure 5.9: Gripper

Example

The example file creates a manipulator with some given IDs and makes it move in y and z direction using inverse kinematic. It also demonstrates the gripper. The only thing needed are the IDs of the wheels. You can still use the pingAll() to check for IDs on the data-bus.

5.1.5 Testing it all with a local interface

One important feature of all system is to be able to test things at different levels. All of the modules are supplied with example-code testing the most basic functions. We thought that it would be nice to test all of the modules together without going through the internet. Then we wouldn't have problems of latency for example.

The interface example program creates a window on the pi by using the X Window System. The window can detect mouse and keyboard input. To see the window you could hook up a monitor to the pi or you could use a program such as *Xming*[8] which make it possible to open windows on the Raspberry Pi through the SSH connection. Explanation on how to do this is also explained here [9] under step 3.

In the window that the program creates you could move the manipulator in the X-Y-plane by pushing the left mouse button and moving the mouse around. You can move up and down in z-direction by scrolling the wheel, and grab things with the gripper by pushing right mouse button. With w,s,a,d buttons you can drive around.

The interface was very useful when debugging the hole program of the robot without the need of internet. Many problems arose due to the fact that there was much more communication on the bus, and due to other stress on the system. Also it was very pleasing to drive around and pick up things.

5.1.6 Sensor

The use of the AX-S1 Integrated Sensor was realized by the Sensor class. Here are functions for calculating distance to objects using IR, functions for measuring light and functions for playing sound. Distance to objects are done by the sensor sending pulses with a IR-diode and measuring the strength of the IR returning. This way one can predict how close an object is (higher strength = close).

The sensor can also play sound, either from its internal memory or by sending notes and how long it should play that note to it. In the datasheet is a table of notes and corresponding values. There are one function for each of these modes. The latter function takes in a array of notes and length of each note.

5.1.7 Exception handling

Another important feature of the setup of the robot is exception handling. We discovered that we often got error in communication when using the Dynamixel library. So we decided that there should be some kind of exception handling when the robot looses one of its motors or sensors. When one motor is lost it throws an exception with information on what went wrong and which motor it was. The exception can then be caught in the Car or Manipulator class and set the object in *fail-safe* mode. When entering *fail-safe* mode a thread is created that ping all the motors in the object regularly. This way the program will discover automatically when the object has recovered and set it in *idle* mode again. One other way to do this would be to ping all the motors regularly and

[8] `http://www.straightrunning.com/XmingNotes/`

[9] `http://pihw.wordpress.com/guides/direct-network-connection/`

then also automatically discover when motors become unreachable. But this would make the data bus even more busy in *idle* mode which may be undesirable and may cause more communication error.

When in *fail-safe* mode the manipulator or car object disables all its functions. This way you can not for example drive the car if one wheel is lost, which would result in the car not driving straight forward. We thought about having a how we could operate the car when one wheel was gone. For example could we disable only the back wheels if just one of the back wheels was lost. We tried this and discovered that it was to much friction in the two wheels that the car wouldn't move at all, or not move in a straight line. Therefore we didn't choose this approach.

For future work one could also think about how to implement a mode for operating the manipulator with one or more of the servomotors lost.

5.1.8 Future problems/challenges

As said in the last section future problems could be to implement a better *fail-safe* mode where you could operate the car or manipulator even though one of the servomotors is lost.

Another problem with the code that we discovered is the use of threads together with the use of the Dynamixel library. To ensure that multiple threads wasn't speaking on the data bus at the same time, we had to introduce mutexes for the send and receive functions. The problem was with a the timeout in the Dynamixel library functions. If to long time pass before the functions receive something back on the bus, the functions would give an error back with the error command *RXTIMEOUT*. The problem then arose if one thread sends something on the data bus waiting for a response, another thread takes over and when the thread that was sending come back again it has gone to long time, and it issues an error. This could be done by increasing the timeout in the Dynamixel function, which requires some knowledge on how the library works. A more elegant way is to make the communication functions *atomic*, which would mean that no other thread could interrupt that thread when executing this line of code.

5.2 Agent-server communication

The vehicle needs a solid way of communicating with the server and other devices. We chose to use a server which hosts a rest API to achieve this. A rest API uses HTTPS requests to communicate text strings in JSON format. All actions are initiated by the clients, and the server is "resting" otherwise. A JSON object is a normal string, formatted in a certain way known by both end points. This API is described in greater detail in the server section of this report. The vehicle part of the communication was written in C++ to ensure compability with the other modules. The following libraries were used

- Jansson - A library for processing JSON objects in C

- Curl - A library for making HTTP calls in C

5.2.1 Sending and receiving commands

Any client using this C++ framework for server communication can both receive commands from the server, and send commands to other client. Commands are sent using the *json_send_command* function. It takes a command string as input, as well as the receiving client id for the command. The *json_get_commands* function provides a vector of command strings. It contains all the commands that were sent to your client id since the last time this function was called. It is up to the user to iterate trough this vector and extract the command strings, as well as perform action based on the commands received. We use commands like "forward","backward","turn_left" and "stop". It is up to the user what the various commands are, but both endpoints of communication must agree upon which commands are used. The server will be "stupid" in this regard, as it only stores and send information when requested. All agents and users in the system can send commands to each other, unless acess restrictions are implemented.

5.2.2 Sending and receiving sensor data

Sensor data is sent and received in the very similar fashion as commands. The vehicle code must construct a table of sensors and their associated sensor values. In practice this table is a c++ map. The vehicle passes this map into the *json_send_data* function, which then constructs a json object containing all the data arranged in a predermined structure. The json object is sent to the server using its REST API. When a client wishes to get the latest sensor data from the vehicle, it simply calls the *json_get_data* function. It must specify which vehicle it want to receive data from. The function return a map of sensors to the client, identical to the one the vehicle uploaded.

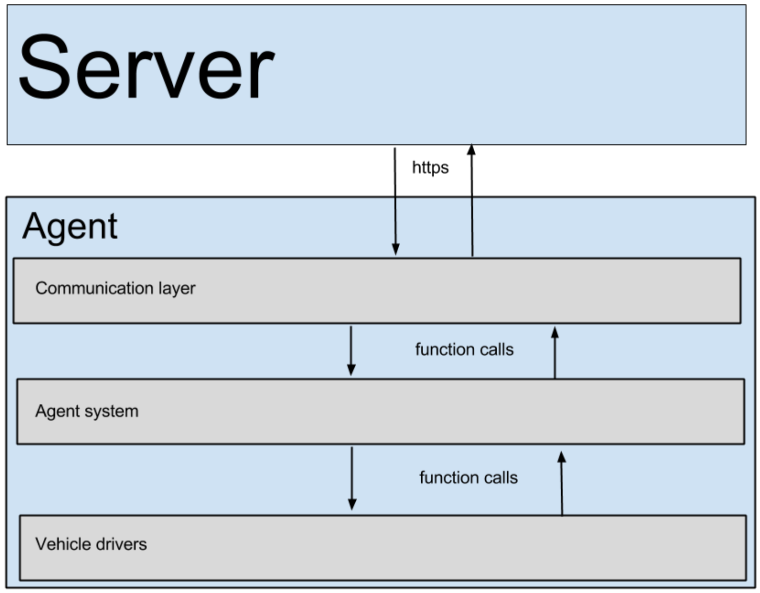

Figure 5.10: Structure of server-agent communication

6 Results

We should have like, a graph or drawing of the finished system here shouldn't with the involved components and actors and what gets transferred between them. Like those drawings Anders have been keeping safe in his lockerbox or whatever.

The project was initially split into three parts. These were "server","agent(robot)" and "server-agent communication". At the concluding phase of the project, these three part were combined to form the whole system. The planning work done early proved to be very good, as the finishing phase went surprisingly smooth. The server part was augmented with a graphical user interface, which can be used from any web browser. It can be accessed by both smartphones and computer, and from anywhere in the world. The robot's wheels and arm was successfully controlled via this interface. The latency was tolerable for controlling the vehicle, but the system can in no way guarantee real time performance. The worst case agent-server latency is too high for this. The robot's error handling worked well, as its behavior was as intended when we introduced various error conditions.

7 Concluding remarks

7.1 Conclusion

There should be some, like, text here. Probably. Summarizing what we got done and what kind of future work exists. Like ”recommended avenues for future research.” Oooh we should mention that the system can be used as the basis for a web service (like github) or on a local network to allow the remote control of devices on it (but not from the outside, like idk maybe a company would want to do this). The great thing is that once you’ve written the agent-client and user-client you can re-use these on other platforms.

This paper has presented a simple approach to making a very general and scaleable system for communication trough the internet. It can be implemented with various degrees of security layers, depending on the use case. The system can just as easily be implemented on a local network, allowing control with total separation from the outside. We have shown a working implementation of the system, where we used it to control a robot.

7.2 Future work

Direct connections for real time communication like video and driving