Real-Time Systems project group 6

Real-Time Control in the Cloud Using 4G/LTE

December 19, 2014

FRTN01 - Real-Time Systems

Department of Automatic Control
Faculty of Engineering
Lund University

Authors:	Johan Malmberg	(*kurs07jm14@student.lu.se*)
	Maid Delic	(*adi10mde@student.lu.se*)
	John C. Stranne	(*ael10jda@student.lu.se*)
Supervisors:	Victor Millnert	(*victor@control.lth.se*)

Contents

1 Introduction

1.1 Aim

The aim of this project was to create and to evaluate the performance of a network based control, see Fig. 1. The introduced delay would be handled the best way so that it affected the process as little as possible. The process that was controlled was the ball and beam process, see Sec. 1.2.1 for more information on the process. The process would then be connected via a Raspberry Pi that would read and write data to the controller. The controller itself was located in the cloud and via a Huawei dongle connected the Raspberry Pi to the controller over the GSM network.

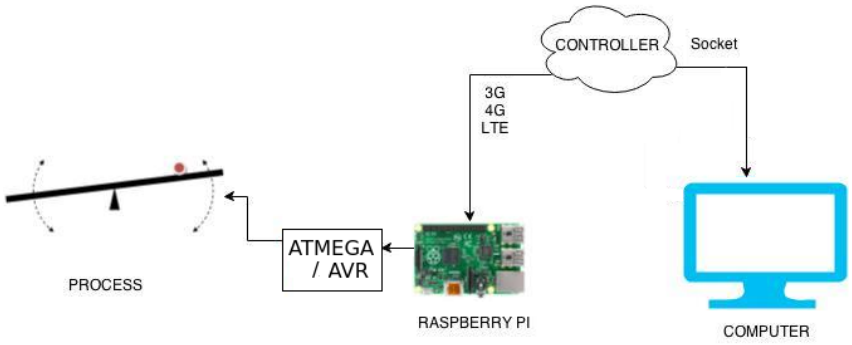


Figure 1: Illustration of the project structure.

1.2 Background

1.2.1 Ball and Beam process

The ball and beam process used in this project consisted of a horizontal beam where the angle of the beam was controlled by a motor see Fig. 2 through a voltage. On the beam a metal ball was placed and its position was measured from the centre through two conductive wires; these were located on the side of the beam. Through simple electronics and classic mechanics the dynamics of the ball position was determined by the following equations:

$$m\ddot{y} = -mg + Ncos\phi + Fsin\phi \tag{1}$$

$$m\ddot{z} = -Nsin\phi + Fcos\phi \tag{2}$$

This gives the transfer function for the process

$$G_x(s) = -\frac{5g}{7s^2} \tag{3}$$

The Ball and Beam process is described fully in:
`http://www.control.lth.se/previouscourse/FRTN01/Exercise3_14/ballandbeammodel.pdf` (**BallAndBeam**)

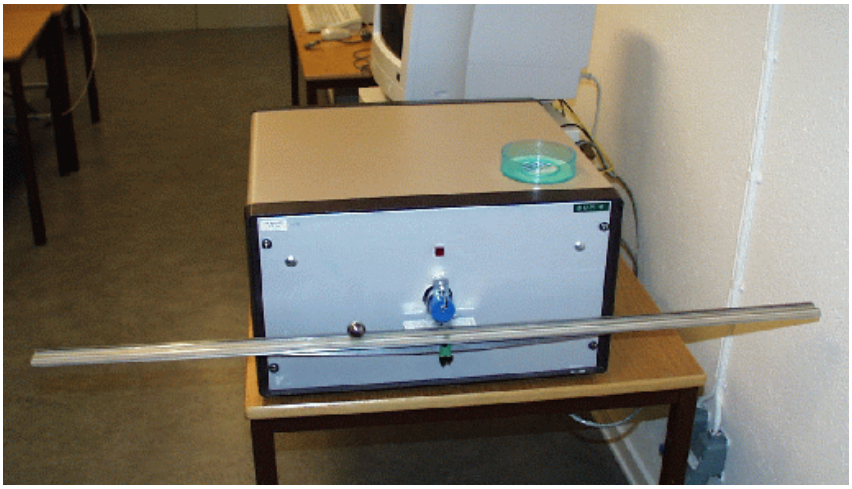


Figure 2: The ball and beam process.

1.2.2 Raspberry Pi

The single-board computer which was used for this project was the Raspberry Pi Model A. The Raspberry Pi's main function in this project was to relay the analogue signal from the ball and beam to the controller via a socket connection and then take the control signal and send an analogue signal to the process. To establish this socket connection, via the GSM network, a Huawei E3276 LTE modem was used.

1.2.3 Atmel Atmega 16

Since the Raspberry Pi was not capable to send analogue signals strong enough to be able to control the ball and beam; another method had to be used to step up the control signals from the Raspberrys low interval of $0.0 - 3.3$ Volts to the higher interval needed for the process of $-10 - 10$ Volts. The solution was to introduce an Atmega 16 that could receive digital signals and rewrite those to analogue that could be sent to the process. The Atmega used in the project was the 16 bit Atmega microprocessor and had a serial port to send and pass information over.

1.2.4 Controller

The controller was a simple cascade controller that consisted of an inner and an outer feedback loop. The inner loop was controlled by an PI controller and the outer by a PID controller. The server was implemented using java and the NIO library using the TCP protocol over streaming channels were used.

1.2.5 Server

Through Amazon, servers are available throughout the world. In this project we're focusing on a server located in Frankfurt. A server located in Oregon will also be tested. These two

serveres will have different latency, however, our goal is only to be able to control the system with the server located in Frankfurt.

2 Method

2.1 Configuration of Raspberry Pi

Before using the Raspberry Pi, software configurations had to be made. An operating system had to be chosen and the OS Rasbian was selected. Since Java was to be used on the Raspberry Pi, Java SE Embedded had to be downloaded and installed. This made it possible to program and run Java code on the Raspberry Pi. To be able to connect to the GSM network through a Huawei E3276 LTE modem, the Raspberry Pi had to be configured. A script named wvdial and usb-modeswitch made it possible to setup the LTE modem on the Raspberry Pi. The configuration of this part was made with help of a tutorial found online. Other configurations where made, such as installing the Samba protocol. This enabled an easier transfer between a PC/Mac to the Raspberry Pi.

2.2 Control Design

The controller consisted of a PI and a PID controller which controls the angle and the position of the ball seen in lab 1 (**BallAndBeam**). The controllers are connected as such, there's a inner (angle) and an outer (ball position) controller which controls different aspects of the system, see Fig. 3. The outer (ball position controller) needed the derivative part to be able to stop the ball at the reference point. Without the derivative part, the controller would not be able to fix the ball at the reference point. Instead the ball would continue past the reference.

$$u(t) = K(e(t) + \frac{1}{T_i}\int^t e(\tau) \times d\tau + T_d\frac{de(t)}{dt}) \tag{4}$$

Thus the PID regulator (4)

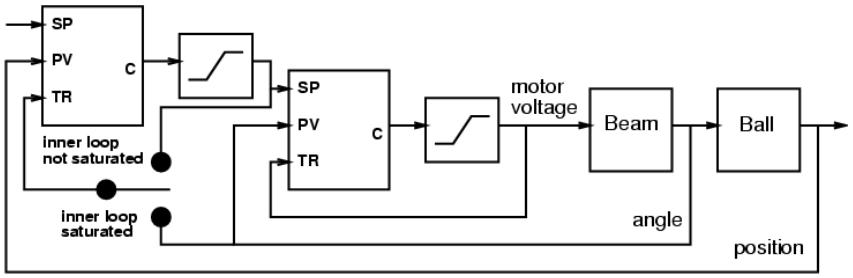


Figure 3: Illustration of the control design.

The motor that controls the beam angle was controlled with the voltage input ranging from -10 V to +10 V. The current voltage value was represented into a string and charac-

ters ranged from 0000 to 1023 where the value 0504 was located at the center (after some calibration). A reference generator was implemented to shift the ball position between two stationary points on the beam and test the controller.

2.3 Code Structure

Since this is a big project, the code had to be divided and tests had to be made before proceeding to the next step. The project was divided in five sections. The description of all the classes and the UML-diagrams can be found in Appendix A.1

2.3.1 Atmega C-code

The first approach was to program the Atmega, this was done using C-code. The program would read the beam angle and position of the ball. Some commands were implemented, start, stop and sending a control signal to the process. The values of the process were printed in the terminal as: *a:0512p:0512*. These values indicated that the position of the beam was horizontal and the ball was placed in the middle of the beam. Pressing 's' in the terminal started the reading of the values and pressing 't' terminated the reading of the values as well as sending a control signal to the process. This control signal was a 0 volt input which meant that the beam would stop moving. While the reading of the values were running, the user could type in a control signal in the terminal. Typing *0540* gave the motor a 0 volt input. When the data was read and written to and from the Atmega a flag had to be used to specify when the value started. For writing data to the output the letters a and p were used for angle and position, and for the data read w was used.

2.3.2 Java over C

After reading the correct values and being able to send a control signal to the process was achieved, a Java program had to be implemented so that it could read the values from the Atmega register and send back a control signal. The Java Simple Serial Connector (*jssc*) had to be downloaded and used. A class called *SerialConnect.java*, see Fig. 7, was written where the serial port to the Atmega was read. Two thread classes were written, one for reading from the Atmega register, *Read.java*, and one for writing to the Atmega, *Write.java*. These two classes were nested classes in the *SerialConnect.java* class and initialized in a separate method. The two thread classes had a sleep method, where each thread had to sleep in 10 ms since the Atmega has a sampling frequency of 10 ms. The reading was done in such a way that the run method read one byte at a time until it came across an a and then it read 9 bytes for the rest of the value pair. This way it could be assured that the correct angle and position values were read. The *jssc*-connection object was always synchronized before accessed so the threads would not read and write at the same time and then notify was called before exiting the synchronized section.

The UML-diagram of this part can be seen in Fig. 6. A small test was done where the Java program printed the angle and position of the ball, a control signal from the Java code was generated and sent to the Atmega which in its turn sent the analogue signal to the process.

2.3.3 Controller implementation

After a successful implementation of the *SerialConnect* was achieved, the controller could be tested. The two controllers had to be synchronized when used in the code. The class *Regulator.java* was introduced so that the process values could be used together with a reference value, generated by *ReferenceGenerator.java*, to calculate the control output to the Atmega which then sent it to the process. The classes *PID.java*, *PI.java*, *PIDParameters.java* and *PIParameters.java* were used for the calculation of the control.

2.3.4 Client-Server implementation

Two Java packages were now created. One for the client where the Raspberry Pi would be implemented and one for the server (the cloud server). The client side had the SerialConnect class which read and wrote to the Atmega. The control of the process was now migrated to the server and the SerialConnect was at the client side. The server and client communicated over a socket for a given port and selectors were establishing the connection.

2.3.4.1 Client-side

SerialConnect.java was not changed in this part but a new class called *Client.java* was written. The Client class had two nested thread classes, *Read.java* and *Write.java*. These two thread classes worked in the same way as the read and write classes in SerialConnect. *Read.java* read from the server and sent the input to the SerialConnect which in its case sent it to the Atmega. *Write.java* took the value from the Atmega and sent it to the server side. All of this had to be synchronized, sleep methods were introduced in all thread classes so that they read and wrote the correct values. As before; the period was 10 ms. A main class, *ClientMain.java* started all the classes which in their case initialized the thread classes. The UML-diagram of the client side can be found in Fig. 7

2.3.4.2 Server-side

The server sides assignment was to take the values of the process and use them together with a reference to calculate a output. The server opened a client-socket-channel and listened to the incoming packets. It took the values from the client, which were a ByteBuffer, and used the *PI.java* and *PID.java* classes in the *Regulator.java* to calculate the correct control signal for the process. It then sent the value of the control signal to the client in a CharBuffer. The main class on the server side, *Server.java*, see Fig. 8 established the connection to the client and started reading the incoming ByteBuffers. When a value was read on the incoming socket it was immediately passed to regulator class, which wasn't threaded and the control signal was calculated. The control value was converted to a string and a *w* and a *0* was appended at the start of the value before the control signal over the socket in a CharBuffer. The reason for appending this zero was that values were three digit and a fourth needed to be added to the start to make up for the missing digit. A limit was set on the numbers so that the control signal never could go below 100 and above 900, thus skipping the issues of numbers above 999 and below 100.The UML-diagram of the server side can be seen in Fig. 8.

The implementation of the server was done with the use of the java.nio library. This library is a non blocking I/O wherein a thread does not have to wait for read or write events. Some of the classes from the nio library that was used were: *Selector*, this class handled the combination of multiple channels wherein a channel is registered to a selector. The class *SocketChannel* represents a channel for a socket, this channel can then be selected by the selector. The class is non blocking and thread safe.

2.3.4.3 The Server

The actual server that the hosting was done, was located at different computers. It started out with first only being the lab computer connecting to *localhost* before migrating it to another computer in the lab. When the project went into the final stages of the project the server code had to be migrated to a computer which could be accessed from the 4G network. The server code was first put on one of the group members own computer, but the internet supplier had a firewall in the way so the connection could not be reached. Then it was decided to move the server into the factual cloud and Amazon EC2 servers were chosen. At first the server was uploaded to a server in Oregon, USA, but as soon as the mistake was realized it was moved to Frankfurt. The TCP connection was configured for the new IP addresses and the default interface was changed on the Raspberry Pi to be the *ppp0*; for the LTE dongle.

2.3.5 OpCom

The final part of the implementation was to create a new client where a user opens a Graphic User Interface (GUI). This GUI was supposed to show the control signal, position of the ball, angle of the beam and the reference signal. Setting the parameters of the inner and outer controllers to calibrate the control signal. This package work as a client that connects to the server and fetches the values from the server. This meant that two threads had to be implemented. One that reads the values from the server and plots the values on the GUI. The second thread class has to take the updated PI and PID values and send them to the server where the controller parameters in *PIDParameters.java* and *PIParameters* should change.

2.4 User Interface

The Graphical User Interface (GUI) showed the reference value (green), the ball position (black) and the control signal (red), see Fig. 4. To edit the parameters of the two controllers, the respective GUI's for the controller parameters were used, see Fig. 5. To edit these parameters, the fields had to be emptied after entering the new value. After the new value were written, the user had to press 'ENTER' to make the change, see Fig. 5a and 5b.

3 Results

3.1 The 4G connection

The first part of the project was to connect the Raspberry Pi to a server via the GSM network. For this the *wvdial* script was used. The *wvdial* runs on old code from the time

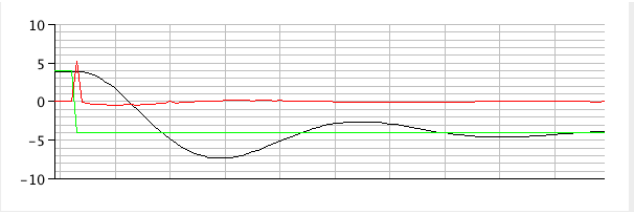
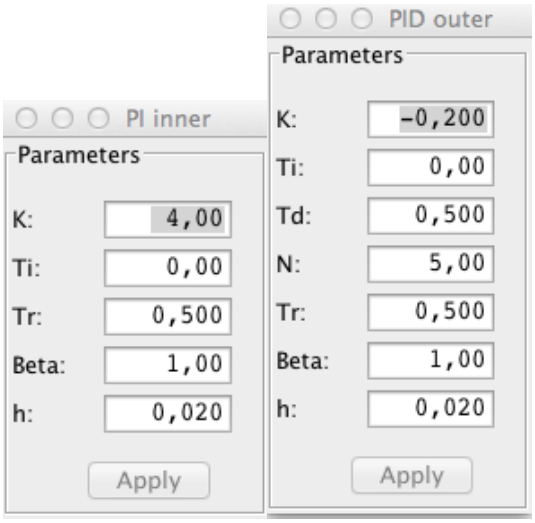


Figure 4: Opcom



(a) PI (b) PID

Figure 5: GUI for regulator parameters

of the modems. This meant that the correct flags in the protocol had to be set before the modem could dial, see Table 1 for parameters used, the problem was that the documentation of these flags were non-existing on the internet and a trial and error commenced to see which flags would establish a connection. It was later found by running the bash command *sudo wvdialconf > init* would save the complete configuration file to a text file.

Once the configuration of the modem was done it had to be automated to connect at start up. This was done by editing the */etc/rc.local*-file in the Raspberry Pi. This file was also modified to choose the newly established connection, i.e.: via the LTE modem, as default interface. The connection was tested using a previously written, for another course, chat program.

3.2 Connecting the ball and beam

Connecting the serial port from the Atmega16 was another problem. The Raspberry Pi, to where all the code would migrate later, did not have the 9 pin serial port but only USB ports. This meant that an conversion had be created. Luckily the department had a cable that did just this.

[Dialer 4gconnect]

Modem	=	/dev/ttyUSB0
Baud	=	460800
Init	=	ATZ
Init2	=	ATQ0 V1 E1 S0=0
Init3	=	AT+CGDOCONT=2, "IP", "online.telia.se"
ISDN	=	0
Modem Type	=	USB Modem
Phone	=	*99#
Stupid Mode	=	1

Table 1: The parameters used in the wvdial program to establish a 4G connection via the Huawei LTE-Dongle to the Telia network.

Writing code for the Atmega16 was not easy, as it meant reading datasheets, trying to understand how different registers worked, reading and writing from these registers and handling the interrupts that occurred every 10 ms. This was a slow process putting the project a few weeks back.

3.3 The Client

3.3.1 A Direct connection

The program was developed in the IDK Eclipse on a lab computer in school with a direct connection the program ran fine. It wrote and read from the Atmega16 at given time and it read the correct values and sent the correct values back. This setup connected to the process could control the ball and beam effortless. Parameters for the PI and the PID controllers were tweaked to gain a better control. One problem that occurred during the implementation was that the values that were read from the Atmega were 16 bits Integer values. The first implementation of this part used Java doubles to calculate the control signal to the Atmega. A good thing to keep in mind when using a C program and Java program together is the size of the data types. A double in Java is a 64-bit IEEE floating point. This resulted in an overflow in the Atmega. Converting the double values into strings in the Java code was necessary so that the Atmega could read the full representation of the control value.

3.3.2 A Client-Server connection at localhost

When the program was rewritten for a client-server it still worked as expected, and the server part was moved to another computer at another part of the lab and the program could still regulate the process without any difficulties.

3.3.3 Other units as clients

When the Raspberry Pi received the client code, still in Java, and a laptop received the server code the problems started. At first the Raspberry was connected to the Atmega via

USB to serial and to the server via a TCP-connection. The Raspberry was passing correct data to and, in the terminal, supposedly wrote correct data to the Atmega. The Atmega however responded by sending completely different signals to the ball and beam. Often sending extreme voltages, with a bang-bang controller-like pulses. There was no control of the ball position at all. After double checking the Raspberry Pis clock frequency the sampling period of the system was changed from 10 ms to 200 ms. A drastic change just to down sample the system. This resulted in a no better control at all. The sampling frequency was once again changed now to 50 ms. It was now possible to send correct control signals from the Atmega and the ball position was controlled; however, the controller was too slow which lead to dropping the ball on the ground. The sampling period was once again changed, now to 30 ms which lead to the ball almost being controlled, but still not good enough, and the system would definitely not tolerate any disturbance. This lead to a shorter sampling period at 20 ms, but it only took the system back to sending either +10 V or -10 V.

The raspberry Pi was then exchanged for a standard laptop, the same as the one used as sever, i.e.: the same experiment was conducted as with the lab computer using *localhost* but now with different hardware. The result was no better than it had been before completely independent of the sampling period. As the code had been debugged and checked several times it was returned to the lab computer just to make sure that it could run the code. The result was that the lab computer could still control the ball and beam without any problems.

Some tampering was made on the server side, and by adding not a *0* but rather *w0* the program all of a sudden was able to read the correct signal from the server. The main problem had been that the Atmega could start reading anywhere in the signal, i.e.: if the signals *0540* and *0560* were sent the Atmega could read this as *xxx0,5400,560x* creating completely wrong signals. This was solved by adding the flag at the start of every signal, turning *0540* into *w0540* and forcing the Atmega to wait for a *w* before allowing it to read four digits to convert to a volt signal. This turned out very well and the code was returned to the Raspberry Pi to test if the Raspberry was able steer the process as well. The Raspberry Pi did not have any problems with this. The next step was to test the process over a 4G connection, see Sec. 3.4.2.

3.4 The Server

3.4.1 The local server

There was one problem that occurred during the implementation of the Client-Server. At first the connection used I/O-sockets these were however blocking threads. The Server, and the client alike, had one thread reading data and one thread writing data. This lead to the reading thread blocked the socket until information arrived not making it possible to write data to client. This could be solved by having one thread waiting for incoming information reading and once it had information it would execute all code and then sending the code to client. That method would have disrupted the sampling period and a different approach had to made. The solution was to use the *java.nio* library which use non-blocking threads and a selector that sends to sockets when data is available.

3.4.2 The remote server

Once the problem with the server-client problem was solved, see Sec. 3.3.3, the server side had to be migrated to the cloud. The server side was set up in the Amazon EC2 servers. An online service for virtual servers. After reading many different manuals and tips one was set up. As the Raspberry Pi got a dynamic IP-address for every time it connected the servers security group, where the allowed IP-addresses, ports and protocols are specified, had to be changed for that specific IP. Then to make the server run, the server socket had to bind to the virtual servers specific IP-address. The virtual server, however, had two IPs: one public and one private address to connect to. The public one was the one to use for external connection but the code could only access the private IP, which resulted in no connection. This was solved by adding instead of a given IP the complete range of IP address, *0.0.0.0/24*, which Amazon did not like. The Raspberry Pi could however connect. When running the server, initially, it was set up in Oregon, USA, which meant crossing the Atlantic Cable and adding big latency making it impossible to control the ball and beam. The virtual server was then moved to the Amazon EC2 in Frankfurt for short distance and thus delays. When running the server from Frankfurt for the first time, 10 pm CET, meaning presumably less traffic on the internet which lead to us being able to control the ball on the beam from a stationary point of 0, i.e.: in the middle of the beam, but not at the edges. But when controlled during daytime the delay was up above 300 ms making it impossible to control the ball due to the delays.

3.5 Other Problems Encountered

More problems arose when trying to test the Raspberry Pi with the lab computer. The lab computer, being connected to the school network, had to authenticate itself to the school servers to be able to run. Thus it could not be disconnected and plugged into a router brought to the lab. Therefore the Raspberry Pi had to be connected to the school network. The school DNS-server gave the Raspberry Pi a dynamic IP-address which meant there was no way of knowing the IP of the Raspberry. In an attempt to retrieve the IP a Java script was written to connect with a server on the lab computer when started. However this did not work as the Raspberry Pi could not be made to run the Java code on start up. Thus posed another problem for the project, if a simple Java program wasn't able to run automatically on the Pi how would the complete program run. The program was written to be started by a bash-script on start up. The bash script was then called from */etc/rc.local*, */etc/inti.d/*, */etc/init.d/rc.local* and */etc/init.d/rc.update* but none of these, scripts that ran on start up and managed to start the wvdial script were able to start the Java code. Both static and variable links were used. The Java code did not run. If called from command line the bash script ran the Java code though. The solution to this, after many hours of research, was to run the bash script at start up with user **pi**, the user that created the script.

4 Discussion

The control design that was implemented worked fine in the direct implementation and the Client-Server implementation, this was when the client and server was on the same host

(*localhost*). The same PI-parameters and PID-parameters were used when running local mode and Client-Server Mode. This was not a big surprise since there was no delay in the network. Since the control design was made in a previous project (laboratory 1), there was no effort in making a better design since the one implemented worked.

The java.nio library was an effective way to implement the Client-Server part of the project. It made the implementation easier and it took care of lot of problems that was presented when using the java.io library. One of the setbacks in this project was that the Opcom was not finished. There was a possibility to see how the angle of the beam, position of the ball and the control signal. A GUI could be used on the server side, this GUI read the analog values that the Atmega sent to the process and then plotted it. The setting of the controller parameters implementation was not finished. One could enter the server code and change the parameters in the code but this is not a good way of handling the problem.

After migrating the client-side code to the Raspberry Pi we encountered several problems. What worked on the school computers did not work on the Raspberry Pi.
After several of hours of discussions and research a working solution was found. The client side had to send correct values to the Atmega and the solution was to add a flag to the beginning of the control value. A simple '*w*' was added before the value and when the Atmega read the input value it would only accept values with the flag. This solved all the problems and testing the Client-Server setup with the 4G network was initialized.
The first test that was done was to ping, ICMP-echo, the server in Frankfurt from the Raspberry Pi to see how big delay the system is working with. The first test was made around 10 pm CET, the delay was measured around 120 ms. With this delay the system could control the ball with a stationary reference at the center of the beam with some overshoots. The control was not optimal but it worked. A second test was made the day after around 3 pm CET but then the delay was around 320 ms, this made it impossible to control the ball and beam. A test to just control the beam was made and even this approach was not successful at all times.

When running the server from the remote virtual server there were big spikes in the control signal present which could not be removed nor investigated any further in the time span of this project. At the very writing of this report it was concluded that the baud rate of the Huaweii dongle was 460800 while the baud rate for the Atmega was 38400 this difference of baud rates could be a reason to why spikes were present in the control signal when the 4G network was introduced. The spikes also introduced a control problem if the Atmega sampled in one of those spikes it would send a way too high voltage signal to the beam making the control unstable. The biggest problem with the system was that we did not have any dead time compensation in the controller. An approach would be to introduce a Smith-predictor.

Compared to this project being run previous years, these results were probably worse. There are a few differences in the projects such as this year a majority of the code was written in Java instead of C. As C is a language closer to the machine language it is also faster than Java, however, there was no proof of Java not being fast enough and completed this task satisfactory. The cloud server that was used this year was located in Frankfurt while previous year it was located at one group members home reducing the delay considerably last year compared to this year. This long delay was crucial for the control of the system as a too big delay could make the system, which was quite fast, impossible to control.

5 Conclusion

Although this project did not work out as intended in the end, we still do not see this as a failed project. At the end all parts fell together except for the control of the process with the huge delay. If more time would be given a better controller would have been designed and implemented. It was discussed that a Smith-predictor could be used to deal with the delay.

The project also suffers from other bad real time design choices such as the us of TCP and not UDP, this could slow down the process but guaranteed throughput. The realisation of different baud rates should have been discovered earlier which could have lead to further investigation to remove the spikes. The parameters for the *wvdial* program are also questionable as there is no real documentation of what exactly each flag does, therefore unnecessary flags might be used or not enough. The process chosen for this project might also be questioned as, although not investigated, the system was too fast for such a big delay.

An internal goal before the project was to try to finish the project couple of days earlier and investigate the delay used in the system. A time series would have been made to make a model of the delayed system. Since the project did not finish earlier this was not possible. The GUI was not finished and this part should not be that difficult to implement but since all other parts of the projects were so time-consuming it had to be neglected.

Overall the project was completed and the results were satisfactory more time could have been spent choosing a process and writing a controller that could handle a delay. Time and people in the project was a limiting factor. This project was described as enough for 2 people and writing in C, but we consider this being a project for 4 people and it can most certainly be written in Java.

A Appendices

A.1 Code

A.1.1 Classes

A.1.1.1 Client Package

- **ClientMain.java**: The main class that starts the client side.
- **Client.java**: Initializes the *SerialConnect* and the its nested thread classes.
  - ··· **Read.java**: Nested thread class that reads the control value from the server and send it to the SerialConnector.
  - ··· **Write.java**: Nested thread class that reads the measurement values from the SerialConnector and writes it to the server.
- **SerialConnect.java**: Initializes its nested thread classes.
  - ··· **Read.java**: Nested thread class that reads the measurement values and sends it to the Client.
  - ··· **Write.java**: Nested thread class that reads the control value from the *Client* and sends it to the Atmega.

A.1.1.2 Server Package

- **Server.java**: The main class that establishes connection to the client. Uses the java.nio library. Reads the measurement values from the client and sends the control value back to the client.
- **Regulator.java**: Calculates the control value and returns it as a String.
- **ReferenceGenerator.java**: Generates the reference value for process and for the calculations for the control value.
- **PID.java**: The outer controller for the process.
- **PI.java**: The inner controller for the process.
- **PIDParameters.java**: The parameters used in outer controller.
- **PIParameters.java**: The parameters used in the inner controller.

A.1.1.3 Atmega C-code

- **ballandbeam.c**: Reads the angle of the beam and position of the ball and prints them to the terminal. Writes the control value to the process.

A.2 UML Diagrams

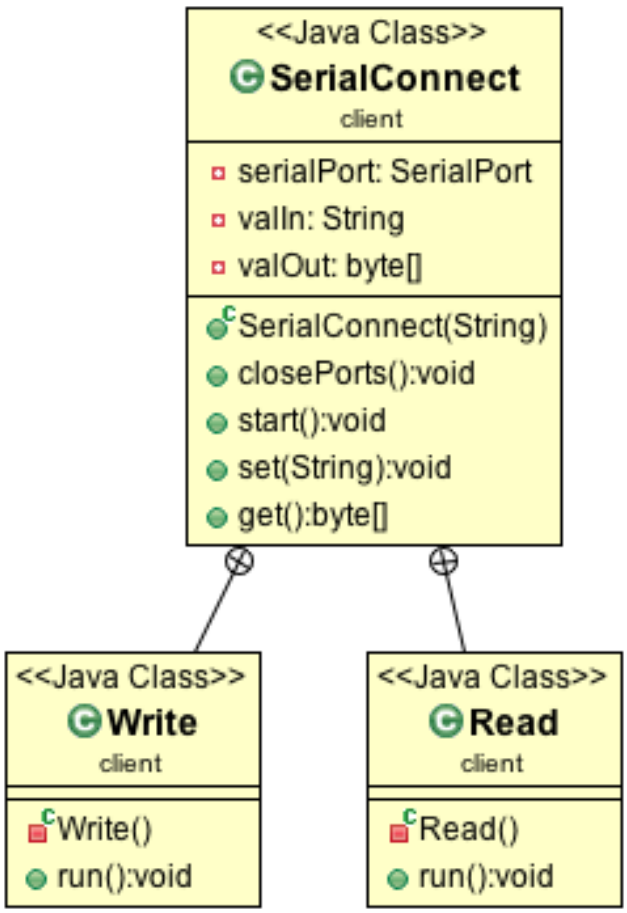


Figure 6: UML Diagram of the Atmega16L.

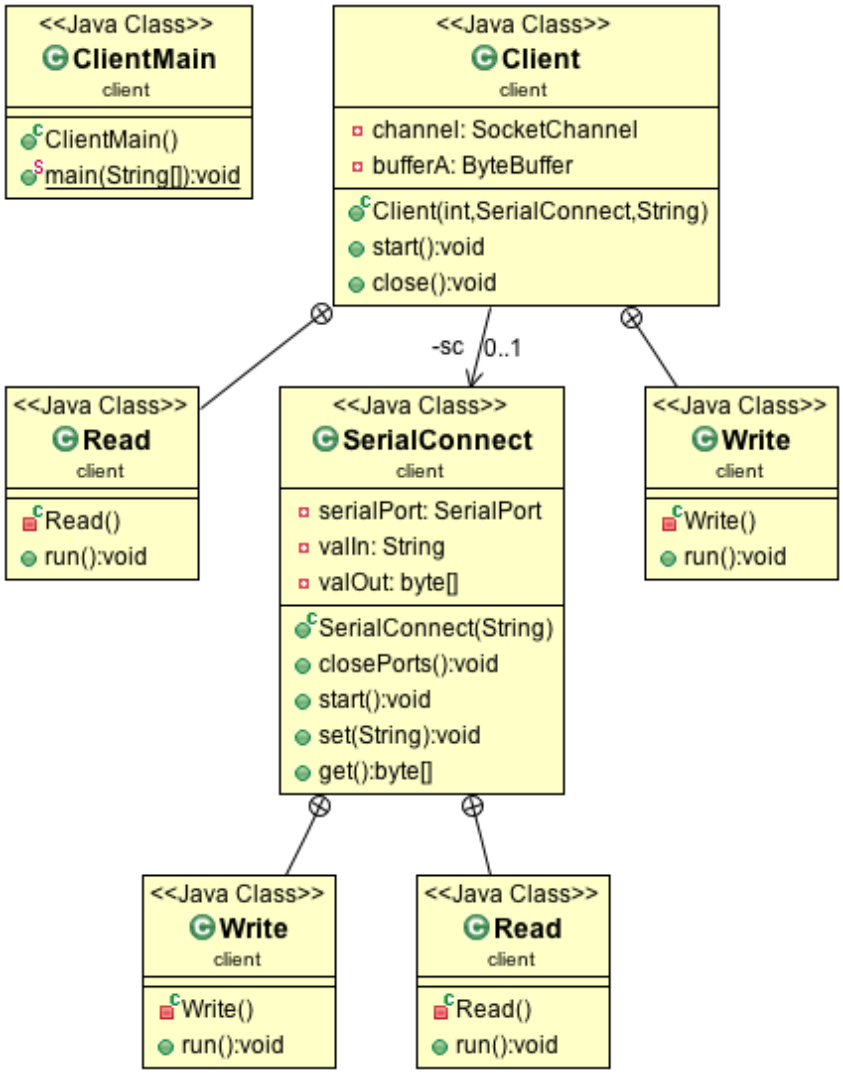


Figure 7: UML Diagram of the Client.

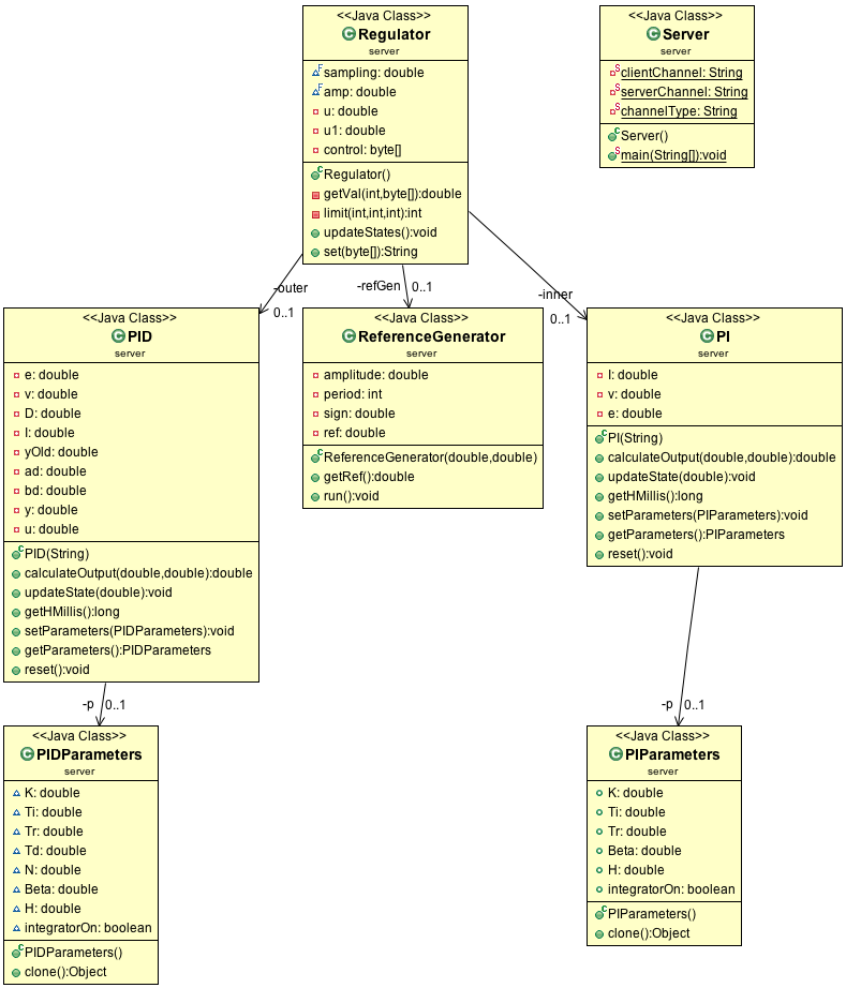


Figure 8: UML Diagram of the Server.