

Dynamic Load Controller

And its uses in First Person camera applications

Contents

Introduction	3
Planning	3
Problem.....	4
Proposition to solve problem.....	4
Process	4
How to measure a slower stream/network?	5
Measuring network statistics	5
Latency	5
Packet Loss	6
Throughput	6
Bandwidth	6
Conclusion.....	7
How to change the streams quality dynamically	Error! Bookmark not defined.
Changing encoder settings.....	7
Relevancy of sacrificing quality for data compression	9
Control loop	12
Final product demo	15
Notice about the project.....	15
Future Work	15
Feasibility of planning	15

Introduction

The controls of robots have been an ever-evolving phenomenon over the years. While at first, we were tethering them to a bunch of wires, with only a joystick to control; we now can let them do the most advanced tasks on their own.

Vision still is one of the hardest hurdles to overcome when it comes to robotics. Many robots are past the point of the need for human eyes. Though a niche group of robots still rely heavily on them. This group can be described best as a group of “competitive” robots. Like sports robots, or racing robots.

Our sports robot still has a need for human eyes. Commonly, these eyes are from the side of the field. However, we would like to upgrade our eyes, to a first-person perspective. Essentially being a football player on the field, through the eyes of the robot.

Planning

To keep track of work done, I made a planning which I can adhere my progress to.

6 dec:

Assigned new problem in group.

13 dec:

Form idea of which metrics are important and which aren't.

20 dec:

Present current findings and progress to Bryce and Edwin.

10 jan:

Have prototype working to present to Edwin

17 jan:

Have fine tuned product to show Bryce and Edwin.

Problem

To fetch our camera livestream to utilise it for first person driving, we use the internet. Internet has many benefits, though downsides also are abundant. One of the downsides to internet is, that it can become unstable very quickly.

This poses a risk to our camera stream, since it could likely be negatively impacted by an unstable network.

Luckily, there are systems in place to lessen the load a camera stream puts on a network; though this is not yet in place in our system.

Proposition to solve problem

Create a system which reduces the load the camera stream puts on a network. This can be done in various ways, though the underlying outcome is that the amount of data sent over a set period needs to decrease in contrast to the amount of data sent previously.

Both the and the adjustment of the amount of data sent should be in a feedback loop. This ensures that the system is always online and can work without any manual input.

Process

When orienting how to solve this problem, it is determined that we can best split the problem into two sub-problems.

1. How to measure a slower stream/network?
2. How to change the streams load on the network dynamically?

How to measure a slower stream/network?

The quality of the stream and its speed are roughly determined by three major factors:

1. The speed of the stream algorithm
2. The speed of the hardware
3. The speed of the internet/bandwidth capabilities

The only factor which can influence the speed of a stream unpredicted, is #3.

Measuring network statistics to determine if a stream is slowing down or speeding up is therefore a smart choice when it comes to predicting stream speed.

However, there is another way to measure the speed of a stream. FPS.

The client of a stream receives the frames of said stream in quick succession; how quick this succession is, determines the framerate of a stream.

We can measure the amount of frames received per second on a specific client, this way; we can measure exactly how fast the stream is received on the client. Especially since we know our target FPS.

To summarize, there are two main ways to predict/measure a stream's performance:

1. Predicting how a stream would behave in the future by measuring network performance
2. Measuring the amount of frames received on the client side to determine the streams performance

We utilized both these solutions to make the final product as diverse as possible. Below is the timeline of progress on both these solutions.

Measuring network statistics

There are various network metrics which are relevant to our stream, all of which have influence on the performance of the stream. These are listed below shortly.

Latency

Network latency is a term used to describe the time it takes one data-packet to go to its destination, and back. The time it takes this packet is described in milliseconds.

This information is not always useful but can negatively impact communication protocols which wait for an ACK-signal to progress in their control loop.

Latency cannot be measured over the entire network. We cannot say "The networks latency is x". This is because latency is always between two points, the sender and receiver.

We measure latency by sending a pre-determined amount of bytes to a certain destination. When we start sending the first packages, we start a timer. We stop this same timer when the receiver of the bytes lets us know they have received said bytes.

The time measured is called, latency.

Packet Loss

Previously we already mentioned packets, in those contexts; we assume that all packets make the target destination. Though in some cases, they do not. This is called packet loss.

While previously, those packets would always more or less arrive, though just with a delay; in this case they do not at all. Which means it has more drastic effects on the application in need of that data.

To measure packet loss, we send a predetermined amount of packets (bytes) to a receiver. On the receiver, we check how many packets we received. Since we know the start amount, we also know how many we should receive.

For example, we send 100 packets to receiver X. X measures how many packets it receives, in this instance; it receives 78. This means that the packet loss is $100 - 78 = 12\%$.

Throughput

Throughput refers to the amount of data flowing **through** a pipeline (both in and out!), in our case this pipeline is our network. This means you can measure throughput at many different parts of the network.

Measuring throughput is simple, but often hidden in hardware registers. It relies on counting the RX and TX bytes of a network adapter/router/network pipeline etc. Usually this is done in seconds. Observe all the traffic coming and going, and that is the throughput.

For example, a pipeline sends 16mb of data and receives 25mb of data per second. The throughput of the TX is 16mb/s, the throughput of the RX is 25mb/s.

Bandwidth

Bandwidth is a very common term. It refers to the total amount of data a network can transmit in a set amount of time. Usually, the bigger the bandwidth, the more data can be pushed through the network.

Bandwidth cannot be measured, since it is a parameter which exists outside of the control of the users. Bandwidth is “given” to a network by either its internet provider or the hardware which the network is set up with. Available bandwidth CAN be measured, though the bandwidth must be known prematurely to processing network data; which is not always the case.

Conclusion

All that becomes apparent is that most of the network statistics need certain data to be sent in order to recognize and calculate a significant number.

Ideally, we do not want to risk measuring a metric which might endanger our stream; by endangering our stream. That is why we want to focus our attention on the *Throughput* metric. Sadly, this does have one downside, which is that throughput can only be measured at very specific points in a network. And that its output to the first eye seems very insignificant.

To measure throughput, we have made a network measuring tool in CPP. It takes a look at the RX and TX buffers of our network adapter. After this, it measures the total amount sent; the average and the current amount sent. This can be found under the Code folder.

How to change the streams load on the network dynamically?

Like mentioned before, all code written and all terminology assume that the used camera package 'Motion' is installed. No changes have been made to the logic and code of Motion; except our added code.

To effectively control how much bandwidth the stream is using, we must find a way to limit or delimit this amount. We can influence various parameters of the stream, which are listed below.

1. FPS
2. Encoder Quality
3. Resolution

We made the choice to change the load based on Encoder Quality, which has a few reasons. The application for this product is self driving, in self driving; high fps and latency is desired. If we throttle on FPS, we might get stable internet but we are left with an unusable stream.

Since we utilize this stream as a source of data when controlling the robot either manually or, in the future, via self-driving interfaces; changing the resolution constantly is not viable. A constantly changing image size when wearing vr-glasses for example, is a self-explanatory example as to why changing the resolution is not preferred.

This leaves us with the only remaining option, which is changing the encoder quality.

Changing encoder settings

Most encoders allow for tuning of their algorithm parameters. Though we found that the best encoder, does not.

The best encoders are usually found not in software, but in the hardware alongside the camera. Many usb cameras have their own dedicated hardware encoder which is predesigned to output a specific compression rate. This means that if we want to change the image, we need to either have a

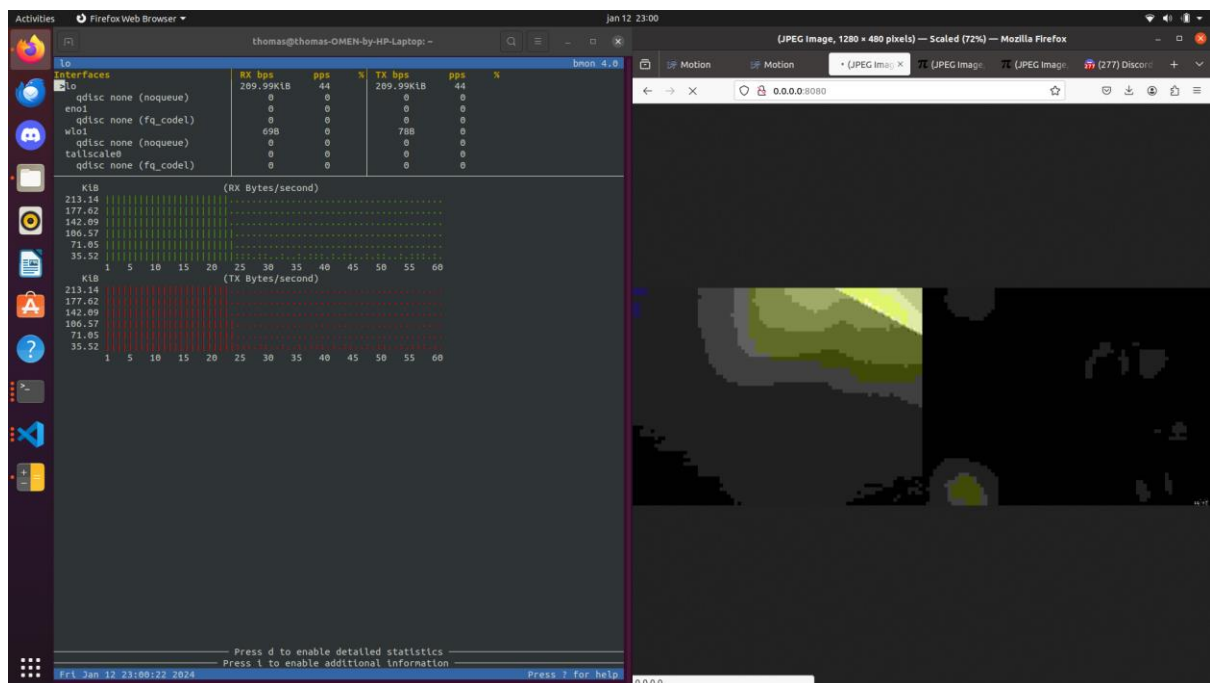
camera which allows for reprogramming of their hardware encoder; or we need to post process the signal with a different encoder.

With processing speed as a primary scope, processing two times (first with hardware, secondly with software); almost always introduces redundancy and delays onto the system. Furthermore, most encoders rely on a raw signal; rather than an already processed image.

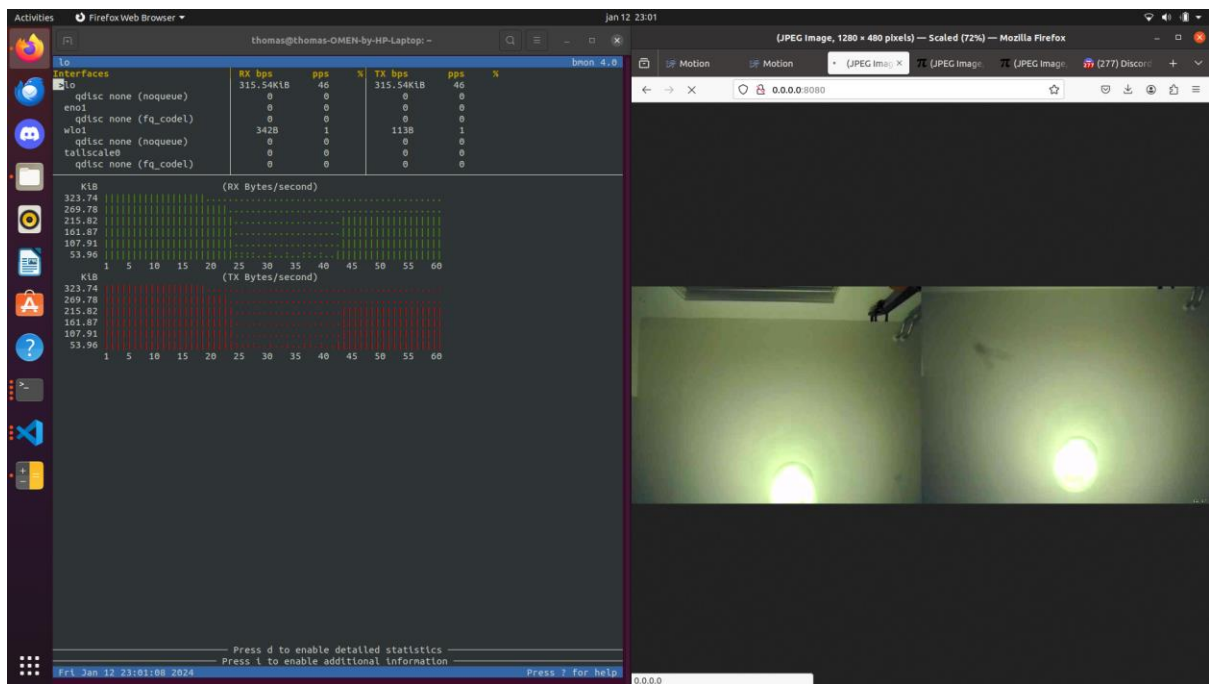
This means that when the hardware encoder does not allow for configuration; it is almost always better to switch to a software encoder.

In our case, the hardware encoder did not support configuration; thus we switched to a software encoder. With this software encoder, we can tune various parameters; one of which includes image quality.

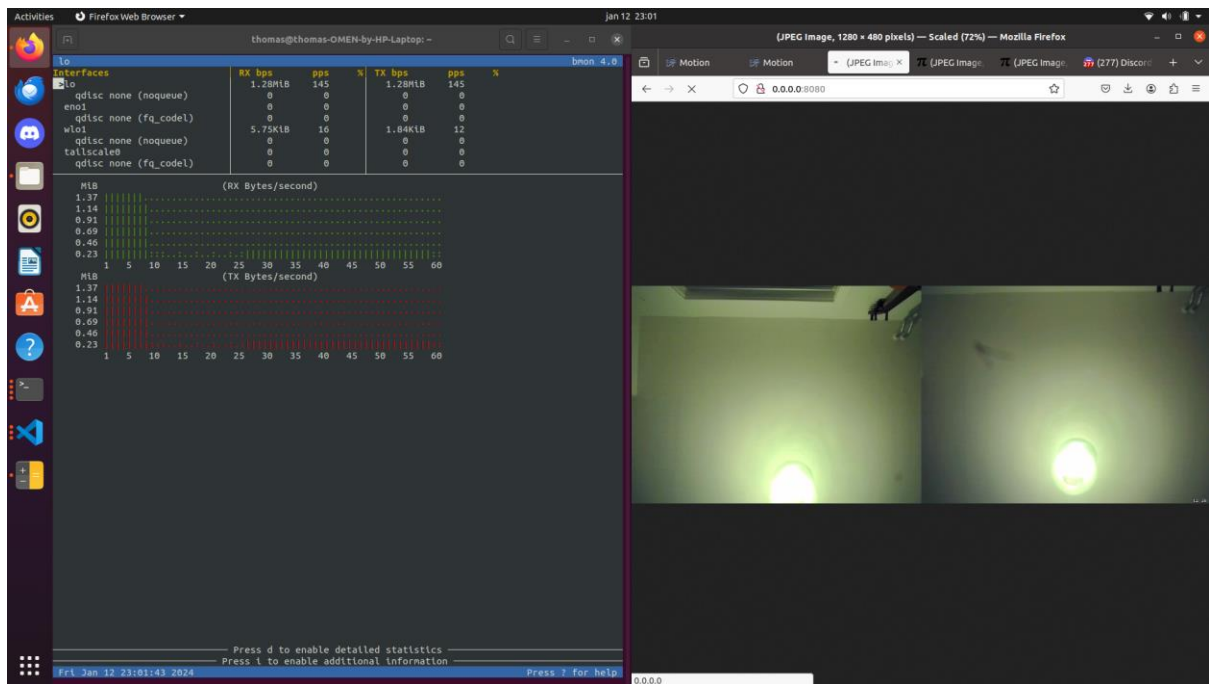
Different quality settings are shown below for visualization purposes, the resolution is 1280x480.



Quality: 1



Quality: 30



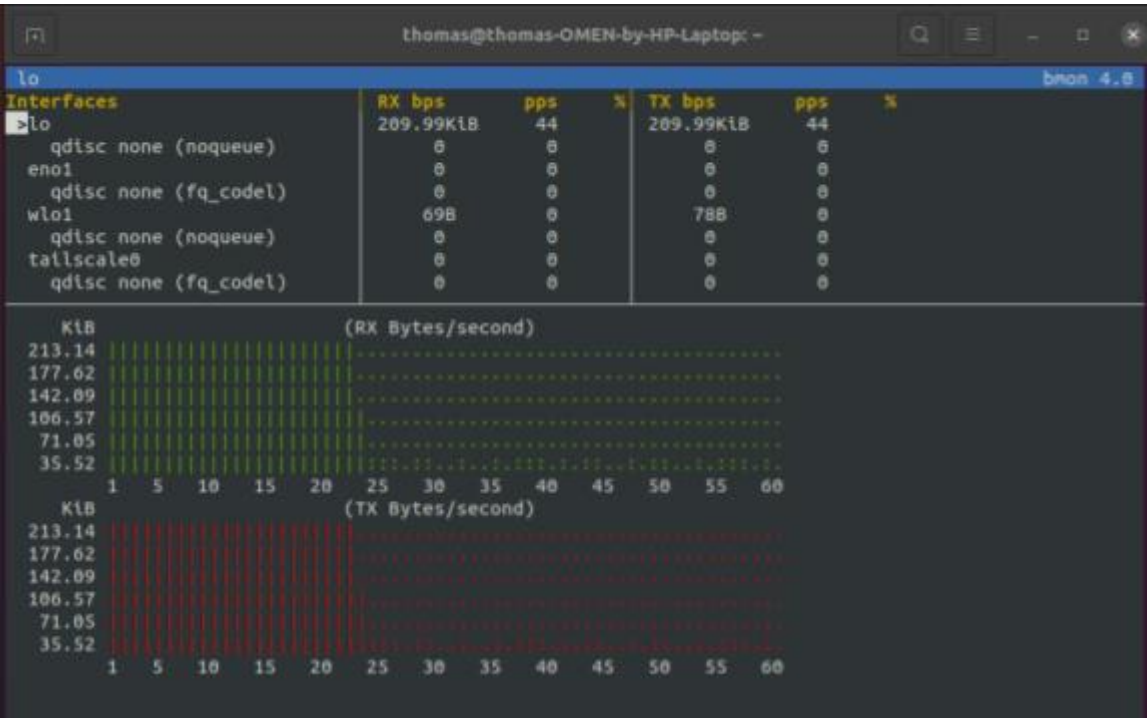
Quality: 90

Relevancy of sacrificing quality for data compression

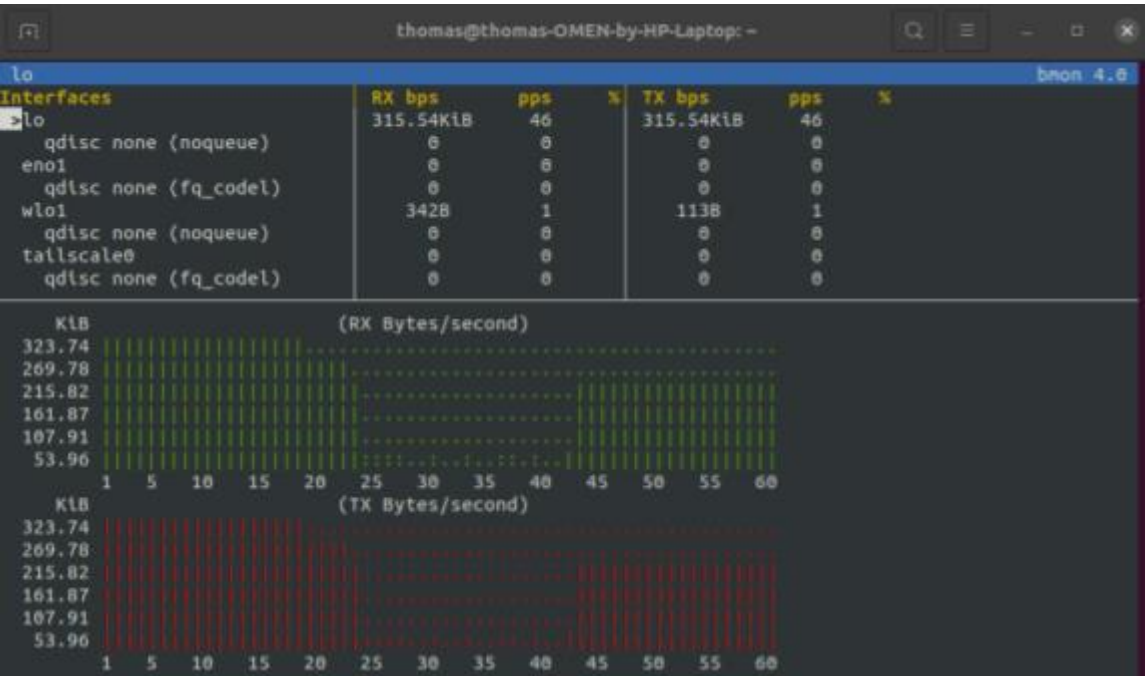
Since up until now, it was assumed that reducing the quality of a stream reduces its data output.

Though this was never proven. We can use a tool called "bmon" to measure the used bandwidth on certain network adapters. Using this tool, we can prove that impeding the quality of a stream means it transmits less data.

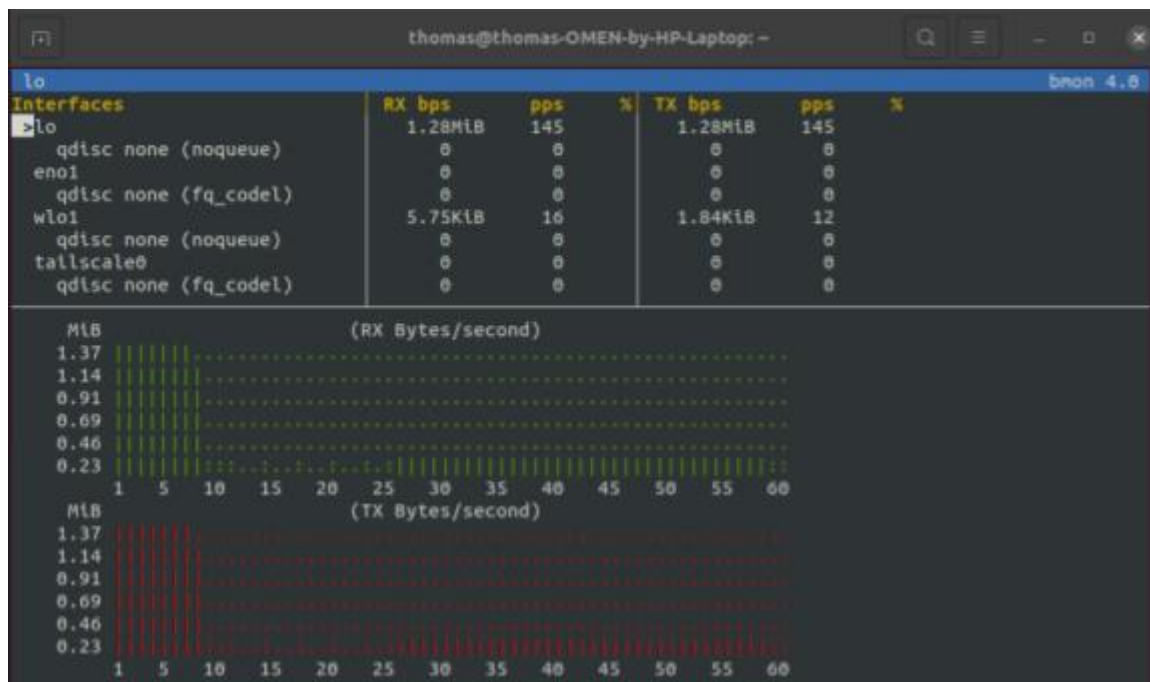
We can see it in the left window of the provided screenshots above. Most specifically, we want to look at the “lo” adapter, the systems local network adapter.



We can see that for the 1 quality preset, we barely transmit 300 kilobytes. Though as seen, we cannot use that output stream.



If we update to a 30 quality preset, we can see that the transmitted bytes barely differ. Though our output becomes quite usable.



If we use almost max quality, we can see that the data quickly begins to ramp up. Remember, we are shooting with 30 fps. That means 30 times the data of one very high quality frame.

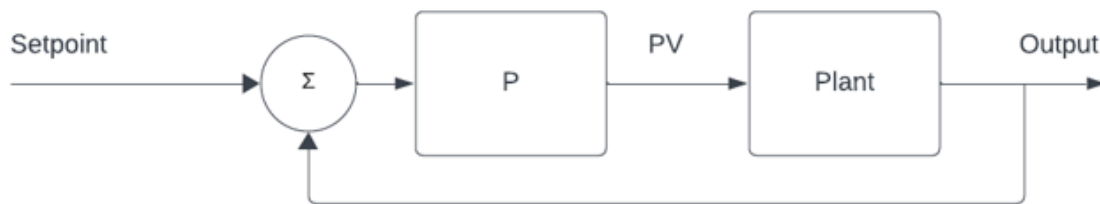
As we can see, even with a slight decrease in quality; we can save over 60% of bandwidth previously taken up. Naturally, there are sweet spots in between the three measurements provided.

Control loop

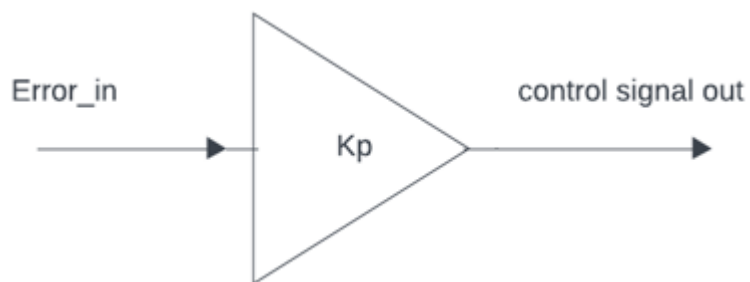
Now that we can influence the systems parameters on the go, a control loop is necessary to keep the system in a desirable state. We did extensive research about this, more information about the topics discussed can be found in the research file.

Diagrams

According to the research done in control loops, it was found that a closed-loop with P controller approach was the best. If we implement this in the system, we can diagram it like this.



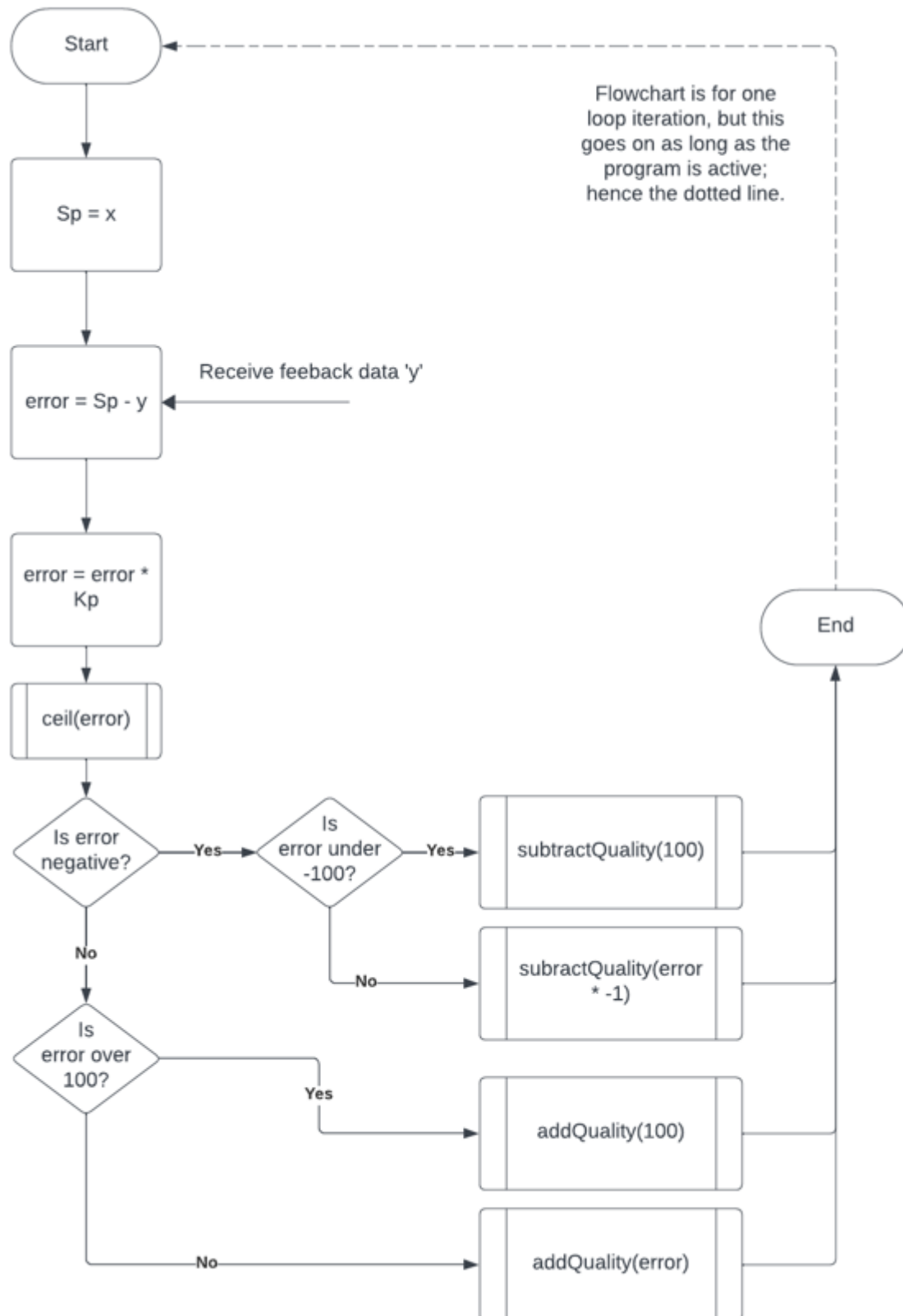
The P block is where our decision logic resides and is what controls the new input to the system. If we zoom in on the P block, we can diagram it like this.



Where K_p is a specified number. The symbol used is for 'Gain', in other words: multiplication.

Flowchart

To show how logic flows in the system, a flowchart is made. This flowchart assumes one revolution through the program loop. While in reality, this process goes on until termination of the program.



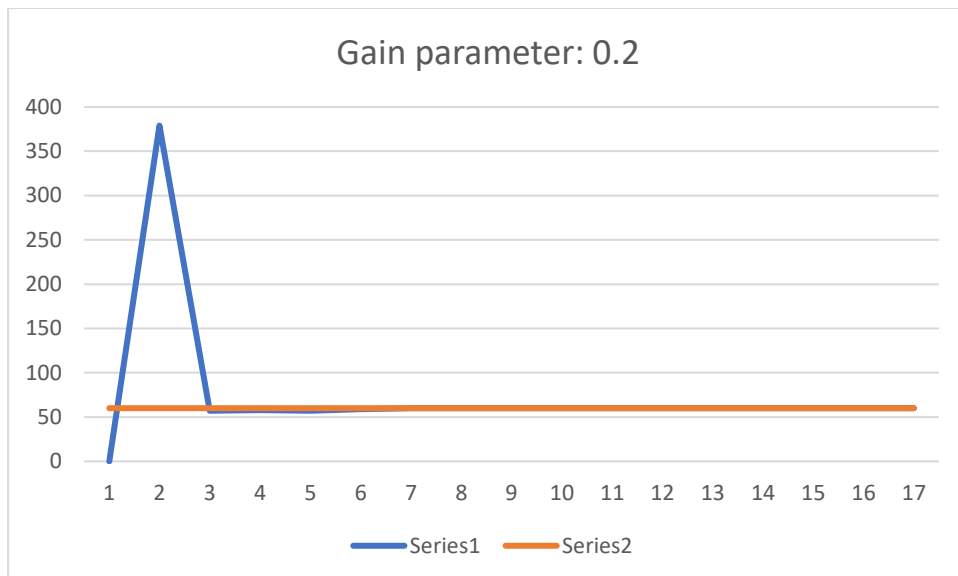
Tuning for Kp

To find a suitable value for Kp, the gain, tests were done to see how the system responds to multiple Kp values.

We are doing this by utilizing a variation of the 'Trial and Error' method; but simultaneously measure the output of our program to a file and graphing it to visually represent its characteristics.

Trial and error is a method of tuning which relies on guessing, and 'being content with' the output. Usually the proportional factor is the main control during tuning. If applicable, the other factors are used for refining.

While tuning, we found that a value of 0.2 behaves the best in this system. A graph of the signal response is shown below.



Final product demo

Below is a short video which shows the product in action. We open a stream and view its contents. To increase the network usage, we open another stream in our browser. We can see the program adjusting its quality levels, both in the console and on the stream this is noticeable.

Video added as dynamic_demo.mp4

Notice about the project

Currently the code is all written in a compilable way, however it cannot easily be implemented in the projects makefile. This is due to it being an automatically generated makefile, which is hard to manipulate. I have spoken to the teacher about this and this problem might be resolved. If not it will be clearly mentioned here.

Currently the code resides as local functions in the motion.c file.

Future Work

The future can bring some new work to this feature, namely extending the way bytes are read. Since the network registers do not only support the rx and tx bytes, but support way more. This way a more extensive network scanner can be built.

Furthermore, the possibility of adjusting other parameters than just quality were discussed. Though while this needs more nuance for our current usecase; this can definitely aid system response in the future.

Feasibility of planning

In order to validate my planning, I want to look at it again and provide insights into what stage I was in at those specific times.

To recall the planning:

6 dec:

Assigned new problem in group.

13 dec:

Form idea of which metrics are important and which aren't.

20 dec:

Present current findings and progress to Bryce and Edwin.

10 jan:

Have prototype working to present to Edwin

17 jan:

Have fine tuned product to show Bryce and Edwin.

6 dec:

No real comment.

13 dec:

Started researching network metrics and phasing out unimportant metrics. Started picking up on throughput here and trying to write a program which measures it.

20 dec:

Spoke with Bryce about my progress a lot, it wasn't easy to explain at first; but after a while Bryce understood where I was and understood the network measuring demos I showed him. I understood what I was working with.

Also beforehand spoke to Edwin and asked some advice, he gave some pointers and approved or denied some of my questions regarding way of working.

10 jan:

Demoed my work to Edwin, he was very exited and surprised I made it work so well. We tried with a different camera aswell, to validate its generic usage. It all worked flawlessly.

Only real improvement was to port it to a raspberry, if I had time we would do it we agreed upon.

17 jan:

Proposed findings to both teachers. Both were enthusiastic about the work done.