Dynamic Load Scaling

Contents

Introduction

To implement a Dynamic Load Scaling feature into our software, we made and integrated two new functionalities. A feature which scales the load according to input parameters, and a network monitoring tool. This monitoring tool provides the previously mentioned input parameters for the load scaling.

This document provides a more detailed description of the code written, to aid the use of these features in the future.

Dependencies

The proposed classes in this document do not directly rely on any other official libraries other than native C libraries for some math. Though, it does rely on both Linux systems; which use the camera package “Motion” to take care of the image handling. Without this, and its including into the files, the *dynamicLoad* files will not compile. The *networkScanner* files are built in such a way that they compile on most Linux systems.

All the math in these functions **is** however usable for any camera package if it boasts a powerful software encoder and scales its quality using percentages.

Dynamic Load Scaler

The responsibility for this class is to take threshold information and input different values into the encoding configuration. The package supports multiple active streams at one time, which is why all configurations are stored in arrays. It is very important to apply changes to all the instances of these configurations.

The *dynamicLoad* functions rely heavily on the output values from the *networkScanner*.

File structure

dynamicLoad.h

↳*dynamicLoad.c*

Available functions

```
int GetUsagePercentage(int threshold, int amountOfBytes);
```

Purpose:
Calculates the percentage of bandwidth usage of the stream according to a threshold the user determined.

Parameters:

- *threshold*

The allowed bandwidth usage in percentages.

Min-max value:
1- 100.

- *amountOfBytes*

The amount of bytes per second.

Min-max value:
0 – INT_MAX

Returns:
Either a -1, when given a false threshold; or the current usage percentage in percentages.

Return min-max:
(Error = -1) 0 – 100.

.

```
int SetQuality(int quality);
```

Purpose:

Sets a different quality preset in the encoder configuration to down or upscale the current quality of the stream.

Parameters:

- *quality*

The given (new) quality to update in the encoder configuration.

Min-max value:
0 - 100.

Returns:
Either a -1, when given a false quality; or a 1 upon successfully setting the encoder configuration for stream quality.

Return min-max:
Error = -1, Succes = 1. No other values allowed.

```
int AddQuality(int quality);
```

Purpose:
Adds the specified amount of percentages to the current encoder configuration. If by addition, the quality exceeds 100; the quality will be set to 100.

Parameters:

- *quality*

The amount of percentages which should be added to the current encoder configuration for stream quality.

Min-max value:
0 - 100.

Returns:
Either a -1, when given a false threshold *or* when the addition of the given quality and the current quality sums to exceed 100.
Otherwise, a 1 is returned which indicates success.

Return min-max:
Error = -1, Succes = 1. No other values allowed.

```
int SubtractQuality(int quality);
```

Purpose:
Subtracts the specified amount of percentages to the current encoder configuration. If by subtraction, the quality goes under 0; the quality will be set to 0.

Parameters:

- *quality*

The amount of percentages which should be added to the current encoder configuration for stream quality.

Min-max value:
0 - 100.

Returns:
Either a -1, when given a false threshold *or* when the subtraction of the given quality and the current quality sums to go below 0.
Otherwise, a 1 is returned which indicates success.

Return min-max:
Error = -1, Succes = 1. No other values allowed.

```
int SetAdaptiveScalingModifyer(int usagePercentage, int threshold);
```

Purpose:
Part of the control loop, ensures that the bandwidth usage hovers around the given threshold. We cannot make this exact, since various factors influence bandwidth; some which are out of our control.

Parameters:

- *usagePercentage*

The amount of percentages the current stream draws on the system. This value is retrieved from the network scanner, which calculates this usage percentage agains another threshold.

Min-max value:
0 - 100.

- *threshold*

The threshold which the control loop aims to hover at in percentages. Given 25, the system aims to keep the network usage 25% of the threshold stated by the network scanner.

Min-max value:
0 – 100.

Returns:
Either a -1, when given a false threshold.
Otherwise, a 1 is returned which indicates success.

Return min-max:
Error = -1, Succes = 1. No other values allowed.

```
int SelectScalingModifyerHalf(int usagePercentage);
```

Purpose:
Part of the control loop, ensures that the bandwidth usage hovers around 50%. This 50% is hardcoded. We cannot make this exact, since various factors influence bandwidth; some which are out of our control.

Reason we did this is to have an easy entry point to test the system. Also, the approachment of 50% is done in a different way than the other function.

Parameters:

- *usagePercentage*

The amount of percentages the current stream draws on the system. This value is retrieved from the network scanner, which calculates this usage percentage agains another threshold.

Min-max value:
0 - 100.

Returns:
Upon failure a -1 is returned, otherwise a number ranging from 1 – 11 is returned depending on the decision taken in the decision tree.

Return min-max:
Error = -1 or Succes = 1 - 11. No other values allowed. (unless addition to the decision tree takes place).

```
int PControl(int SP, double Kp, int current);
```

Purpose:
Provides controller logic for a P controller. This is the final and suggested implementation of control logic into the system; even though other options are provided.

Parameters:

- *SP*

Abbreviation for setpoint. The threshold which the control loop aims to hover at in percentages. Given 25, the system aims to keep the network usage 25% of the threshold stated by the network scanner.

Min-max value:
0 - 100.

- *Kp*

Value to give P. Otherwise often mentioned as gain. This value should be found by tuning the p controller. We advise 0.2

Min-max value:
INT_MIN – INT_MAX

- *Current*

The amount of percentages the current stream draws on the system. This value is retrieved from the network scanner, which calculates this usage percentage agains another threshold.

Min-max value:
0 - 100.

Returns:
Control signal in an integer form. This integer is then used to directly control the encoder settings.

Return min-max:
INT_MIN – INT_MAX
(so pay attention to how this output is used, it should be scaled after!)

Network Scanner

The responsibility of this class is to provide the rest of the system with accurate network details to determine if a quality change is needed, and if so: how severe.

The functionalities of this class rely on the Linux network adapter registers, these must be hardcoded into the functions to ensure optimal performance.

File structure

networkScanner.h

↳*networkScanner.c*

Available functions

```
int GetRx_Bytes();
```

Purpose:
Reads network adapter register of Rx Bytes for the hardcoded network adapter. In our case, we want to use a local network so we use the “lo” prefix. This needs to be configured in code and built for every machine.

Returns:
Returns amount of total bytes in the rx_bytes register.

Return min-max:
MIN_INT – MAX_INT.

```
int GetTx_Bytes();
```

Purpose:
Reads network adapter register of Tx Bytes for the hardcoded network adapter. In our case, we want to use a local network so we use the “lo” prefix. This needs to be configured in code and built for every machine.

Returns:
Returns amount of total bytes in the tx_bytes register.

Return min-max:
MIN_INT – MAX_INT.

```
int GetRxThroughput();
```

Purpose:
Calculates the difference between two separate readings of the rx_bytes register. Like mentioned above, these registers can be negative; that means this class is responsible for handling this data correctly.

This function is meant to be used in a loop/polling!

Returns:
Returns the difference between the current and previous read values.

Return min-max:
0 – MAX_INT.

```
int GetTxThroughput();
```

Purpose:
Calculates the difference between two separate readings of the tx_bytes register. Like mentioned above, these registers can be negative; that means this class is responsible for handling this data correctly.

This function is meant to be used in a loop/polling!

Returns:
Returns the difference between the current and previous read values.

Return min-max:
0 – MAX_INT.

```
double GetApproxMaxThroughput(double nominalPercentage);
```

Purpose:
According to a nominal percentage value of the bitrate from the network adapter, tries to calculate a suggestive threshold to limit the system to.

This function is not currently used, since we found that there are way to many factors at play which can influence this outcome negatively; and create false values. We kept it in, since the underlying idea is positive and a great addition to our product. Though the implementation is not yet there.

Parameters:

- *nominalPercentage*

The nominal percentage given by a reliable source to approach the optimum threshold with.

Min-max:
0 – 100.

Returns:
Returns a calculated percentage of the bitrate of the used network adapter to approach a good mean threshold to limit the system to.

Return min-max:
0 – 100.

Control loop

The software proposed in this design document is meant to act as a control loop. Continuously polling specific network data to keep a process within a certain goal. We can define these goals and monitoring classes as follows:

Goal:

Threshold set by user.

Decision logic:

```
int SetAdaptiveScalingModifyer(int usagePercentage, int threshold); or
int SelectScalingModifyerHalf(int usagePercentage); or
int PControl(int SP, double Kp, int current);
```

The preferred and most dynamical logic is the first function.

Tweaking of actual parameters:

```
int AddQuality(int quality);
int SubtractQuality(int quality);
int SetQuality(int quality);
```

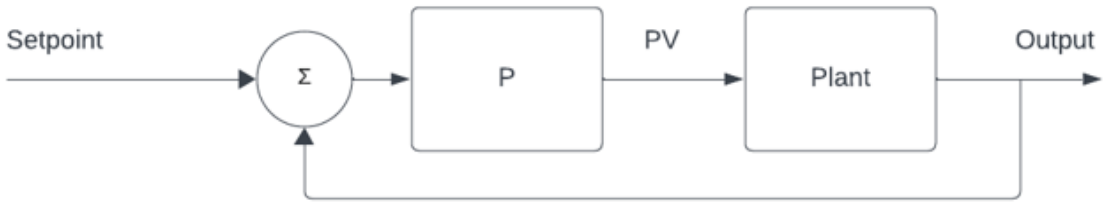
Various functions to tweak the parameters of the system, all modify the same parameter.
These functions are nested in the decision logic.

Verify and return new data:

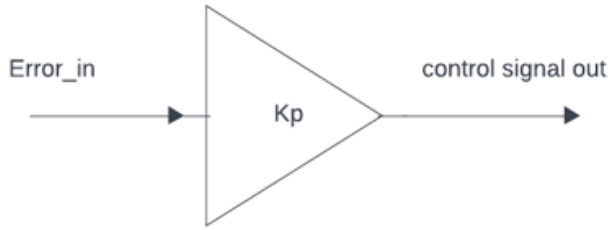
```
int GetRxThroughput();
int GetTxThroughput();
```

Depending on the desired data, either the Rx or Tx function provides a check and new data to adjust goal to.

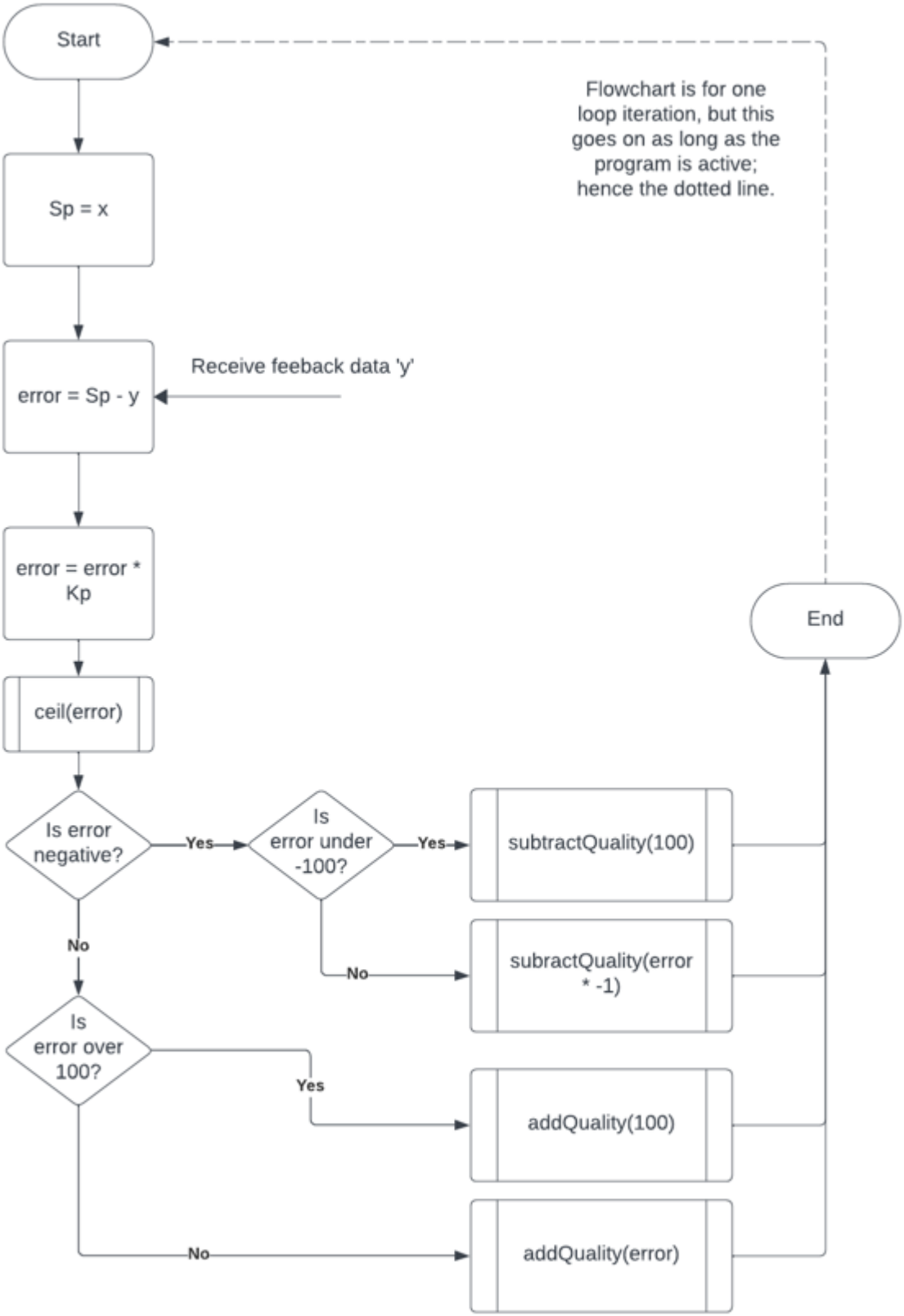
We can visualize this control loop with the following graphic.



The inside of the P Block looks like so.



To illustrate what happens when the system cycles one loop, we made a flowchart to show the logic involved.


Start
Flowchart is for one loop iteration, but this goes on as long as the program is active; hence the dotted line.
Sp = x
Receive feeback data 'y'
error = Sp - y
error = error * Kp
ceil(error)
Is error negative?
Yes
Is error under -100?
Yes
subtractQuality(100)
No
subractQuality(error * -1)
No
Is error over 100?
Yes
addQuality(100)
No
addQuality(error)
End