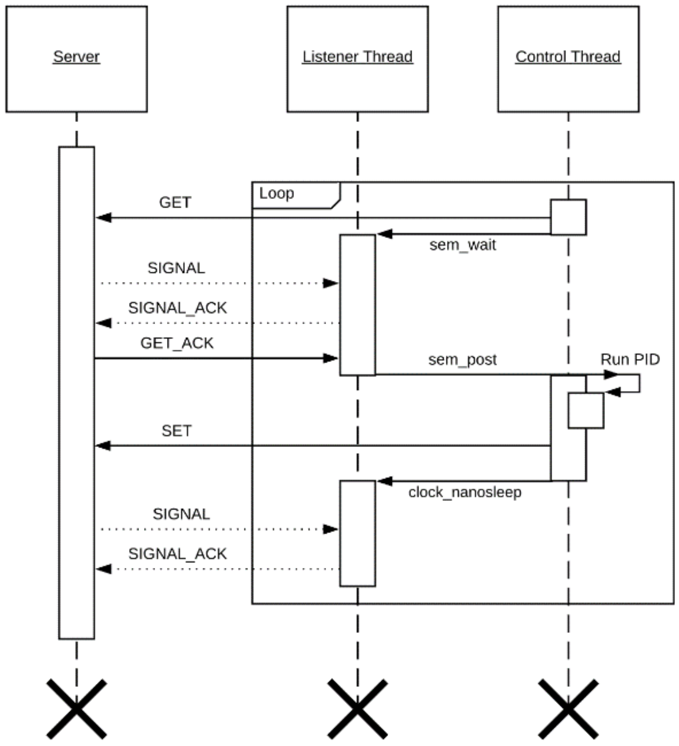
MINIPROJECT REPORT

TTK4147 – Lars Henrik Bolstad – Sondre Ninive Andersen

Thread structure



We chose to use two threads, one control thread, and one listener thread. The control thread runs periodically, at a period of 3 milliseconds, but yields to the listener thread by waiting on a semaphore after sending the “GET” message. When the listener thread receives the “GET_ACK” message, it updates the last received measurement, and posts the semaphore, allowing the control thread to run the controller, and send the “SET” message. After this is done, the control thread will go to sleep until the next period start. If the listener thread receives a “SIGNAL” message at any point, it immediately responds with a “SIGNAL_ACK” message.

This structure should give fast response time, as the control thread only runs to send the “GET” message, run the controller, and send the “SET” message. The rest of the time, the listener thread is listening, and will immediately respond. An advanced scheduler could also pick up on the fact that the control thread only runs rarely, and only for a very short time, and could therefore allow it to run exactly when it requests.

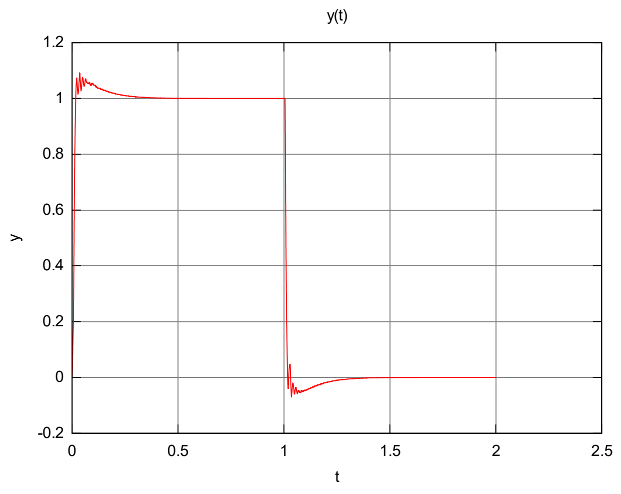
One possible flaw with this setup is that if several UDP messages are delayed, so that the `sem_timedwait` function times out, but arrives after the timeout, the semaphore might be incremented more than one time, and since the wait and post calls are run in pairs, the semaphore will never reach zero again, basically removing the semaphores functionality. We remedied this by only sending the “GET” message if the semaphores value is zero. While this does not guarantee perfect recovery, it does remove the persistent fault condition.

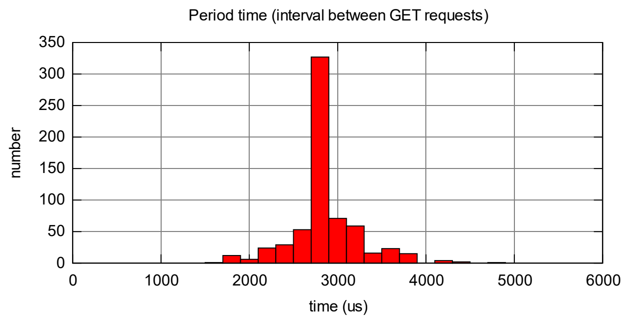
Controller

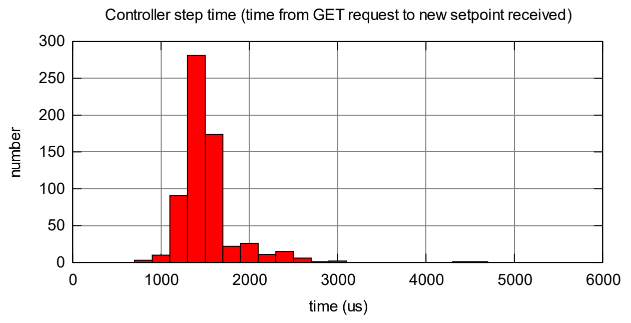
We implemented a very basic PID controller, and tuned it manually to $K_p = 20$, $K_i = 1000\text{s}^{-1}$, and $K_d = \frac{1}{100}$s. This gave adequate regulation, and stabilized the system within 0.3 seconds, with minimal oscillation. From early tests, we noted that the system oscillated with a period around 15 milliseconds. Based on this we should run the regulator at least a few times faster, so we chose 3 milliseconds. We were however initially unable to stabilize the system, but after testing with earlier server-executables found on github, and talking to Anders, we realized there was a bug in the server. Using the new server yielded much better results.

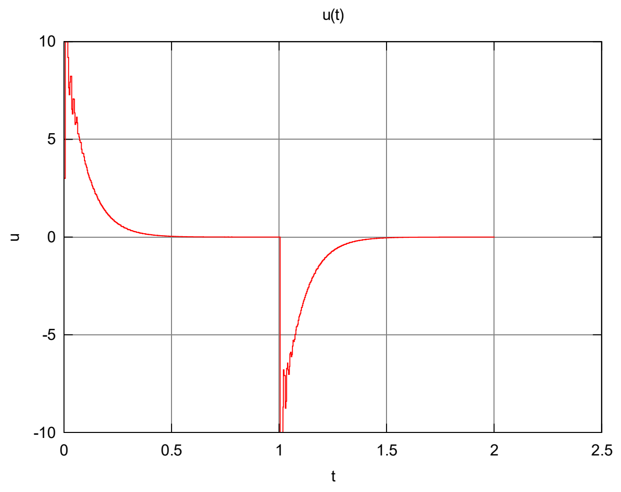
The period was also a tradeoff, as we observed that shorter target-periods generally gave a larger variation in the measured period, and with anything lower than 2 milliseconds the regulator was limited by the network delays.

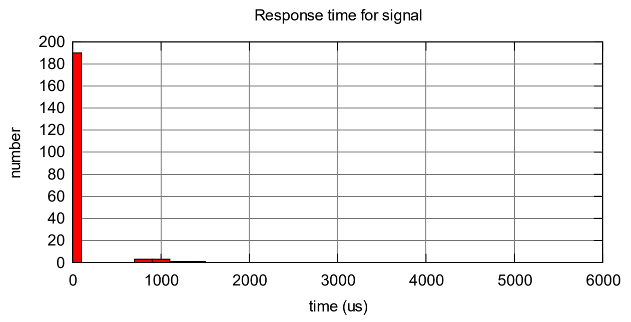
Results











The results show that the regulation is indeed successful, with the system being stabilized within ~3 milliseconds, and with low oscillation and overshoot.

The period is centered around 2.9 – 3.0 milliseconds. As we implemented the periodicity with `clock_nanosleep`, and not a pure delay, a longer period will be followed by a shorter period, which explains why approximately half of the histogram shows too short periods.

The controller step time should be mostly limited by the network delay, and is centered around 1.5 milliseconds. This seems plausible as we were unable to run the regulator faster than approximately 2 milliseconds.

The response time for signal is almost zero (examining the raw data file shows times of 2-12 µs). However, the first few datapoints show response times of around 1000 µs. We are not completely sure why this is, but it could be related to initialization of the UDP sockets, or other factors outside of the applications control.