NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

MINI PROJECT

Group 27

by
Petter H. Wiik and Torbjørn Opheim

TTK4147
Real-Time Systems
11. november 2018

NTNU – Trondheim
Norwegian University of
Science and Technology

1 Intro

In this report we were tasked with using what we've learned to control an unknown system. The server was running on the desktop computer, while the regulation script was running on a NGW100 Xplained developer board. The two units was connected by serial port as well as a crossed ethernet cable ensuring a stable connection. The file system on the NGW100 was accessed through /export/nfs/. on the desktop computer.

We created the program that runs on the NGW100. The board communicates with a server over UDP on port 9999 and therefore receives the current y value and regulates it towards a value of 1 in the interval [0,1], towards 0 in the interval [1,2] seconds and stopping the simulation at the 2 second mark.

2 Considerations, implementation and discussion

We used four threads. One to read from the network, one to respond to signals, one regulator thread and the "mainthread used for starting and stopping the simulation, as well as changing the reference after 1 second. We created multiple threads to better distribute the load of the task. The threads were implemented in code using pthreads".

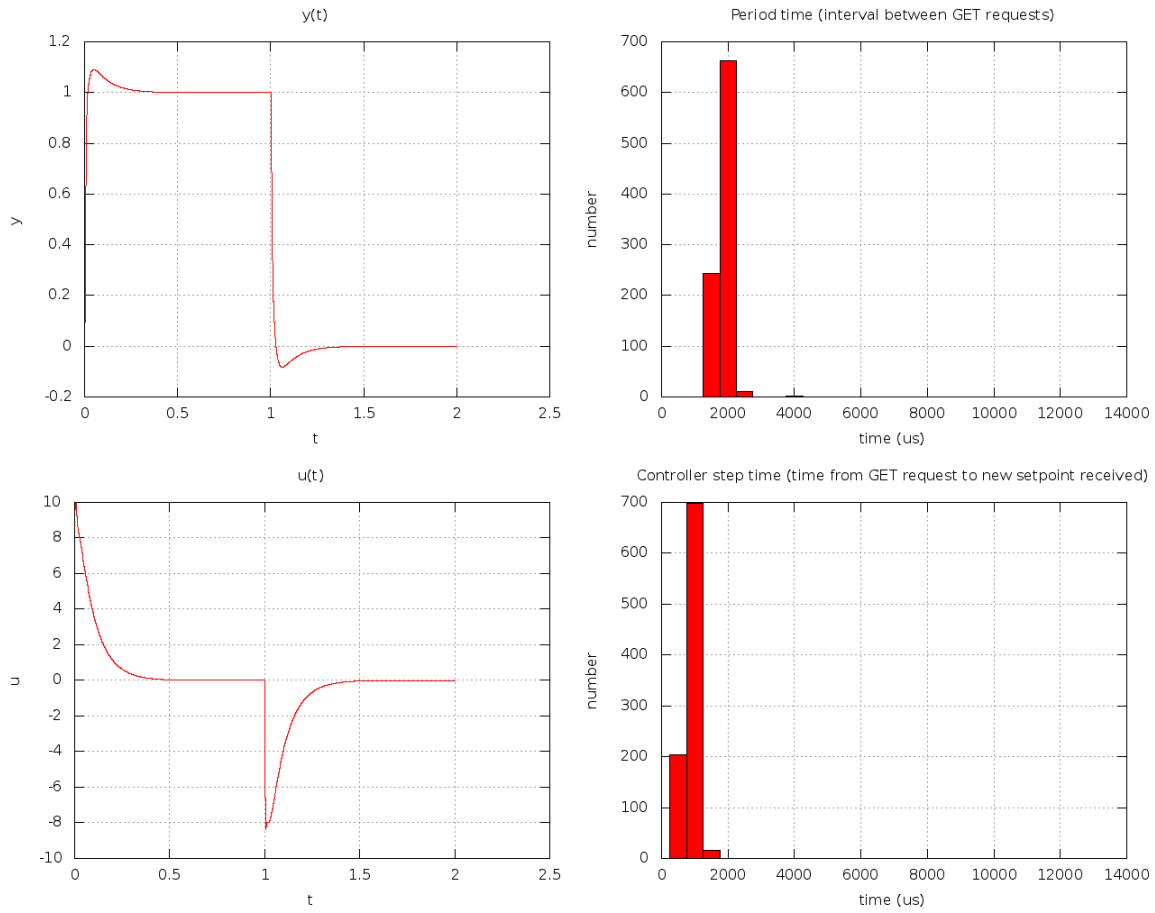
The threads interact using shared memory, with semaphores for synchronization. The regulator thread will request a new y-value from the server, and then wait on the semaphore associated with y. The signal response thread will similarly wait on a separate semaphore associated with a received signal. The network thread will continuously read the network messages, and post to the semaphore associated with the message read (i.e. if a signal is received, post to the signal semaphore). This way, the shared resource y is never accessed at the same time, and responding to a signal does not take priority over stabilizing the system (assuming that posting to a semaphore is cheaper than sending a message on the network).

One second into the simulation, nothing is done to synchronize the change in reference from 1.0 to 0.0 between the "main"and the regulator thread, even though there is a possible race condition. This is done because the change in reference naturally makes the system unstable in this single iteration, so the impact is minuscule compared to the performance overhead of using e.g. a mutex.

To control the system a PI controller was used. No tuning was needed to obtain a sufficient regulation, however we observed some oscillations which was fixed with adjusting down Kp and therefore landed on:

$$Kp = 5, \quad Ki = 800 \tag{1}$$

The regulation when performing on one thread took around 1000 μs, so we used 2000 μs as the period. This also worked for the multiple threads, as we saw that the majority of these responses was able to conform to this period as well.

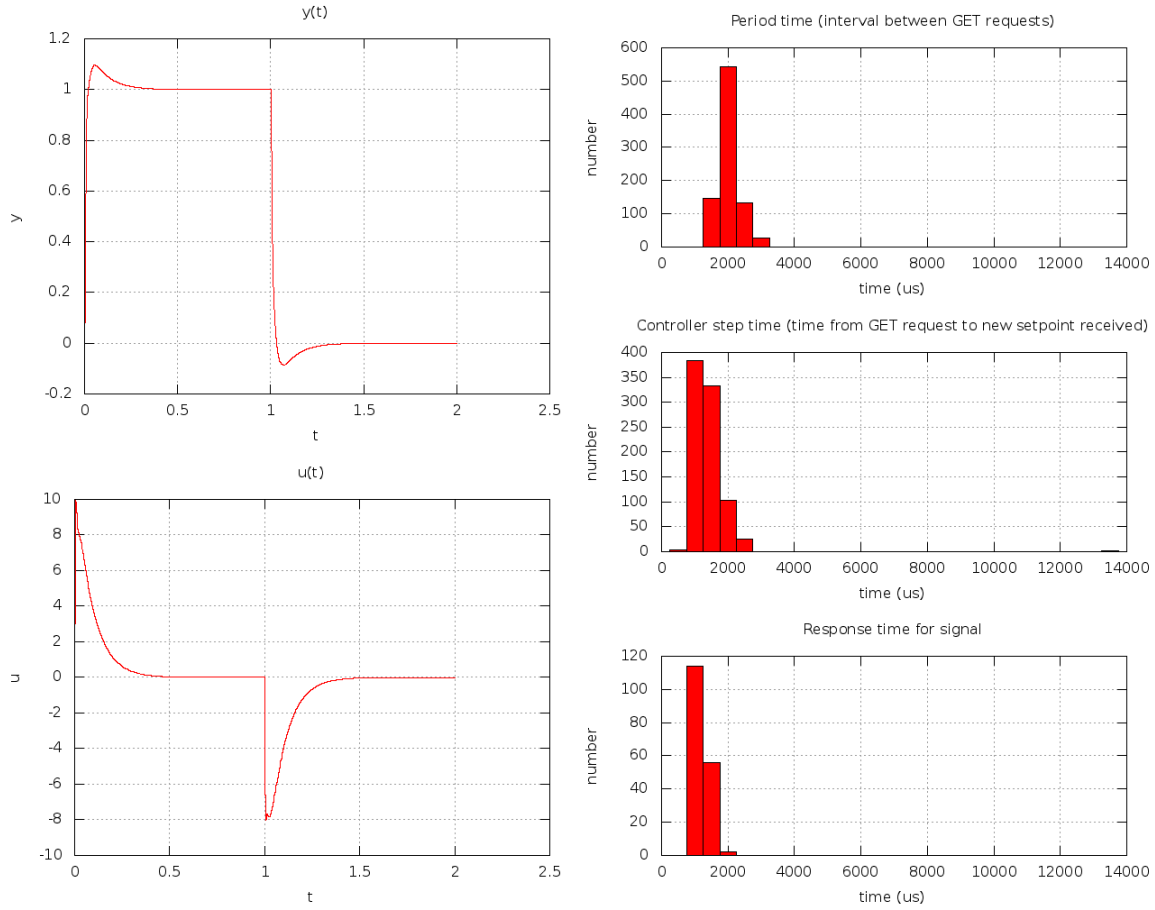


Figur 1: plot4 results with single thread

From Figure 2, we conclude that the response time of the controller is sufficient, reaching the set-point with zero oscillations after ≈ 0.3 seconds. The time between GET requests is also represented by a tight Gaussian approximation around 2000 μs which is also satisfactory.

The response times are consistently less than 2000 μs, with some variation, possibly caused by Linux interruptions. The expected response time is less than 2000 μs, as the PI controller calculations and network response take $\approx$ 1000 μs. (see plot 1 controller step time), allowing other threads to run within the 2000 μs period.

Comparing Figure 1 and 2 we see that introducing the signals results in more variance in



Figur 2: plot5 results with multiple threads

the controller period (more GET requests exceeding the time period), as well as a longer controller step time. This is expected, and does not noticeably affect the stability of the system.