Real-time Systems - Mini Project

Gonçalo Carvalho e Diogo Carrasco

November 2018

1 Number of threads and why

For the second part of the project in addition to simulating a dynamic system, the server will also sends out signals that must be responded to, while the simulation keeps running. In order to do this we divided the program in 4 fundamental parts (four threads) that will run alongside each other and the main function. The approach used (four threads) is related to what the system needs to do. From the server we can receive two messages:

- GET_ACK:123.456
- SIGNAL

The first message is obtained sending to the server: "GET". The second one can be received at any time.

In order to know what we received, since the SIGNAL can be sent from the server at any time and can overwrite the buffer (that can have a response to the GET command) we have one thread that periodically will ask for the value of y sending the string GET to the server. And other that will check if what we received from the server is a SIGNAL or GET_ACK.

If we receive a SIGNAL command this must be replied with a SIGNAL_ACK command wich leads to a new thread that will be in charge of sending the the acknowledge to the server when we receive the SIGNAL command. In case we receive the GET_ACK we will need another thread to handle the control of the simulation, calculating the value of u based on the value of y in order the system to follow the reference.

2 Communication

We choose to implement 2 semaphores (*sem_y and sem_get*) in order to have communications between our threads.

The implementation is done in a way that both threats that send signals to the system (*pthread_signal_ack and pthread_controller*) are constantly waiting for the correspondent semaphore to be unlocked (*sem_wait()*), and only after that signal both threads preform their operations. The signal to unlock the semaphores (*sem_post()*) are send by a thread whose function is to unlock one of the semaphores depending on the answer from the system (stored in the recvBuf).

With this implementation we can be sure that the threads that deal with the responses to the server only operate when the system ask for the corresponding operation, consequently we can assure that just one signal is sent to the system at the time.

We rather chose semaphores to mutexes because mutexes can only be lock and unlock by the same thread, and for us that's not useful since we want one thread to control the execution of other two threads.

3 Period and other parameters

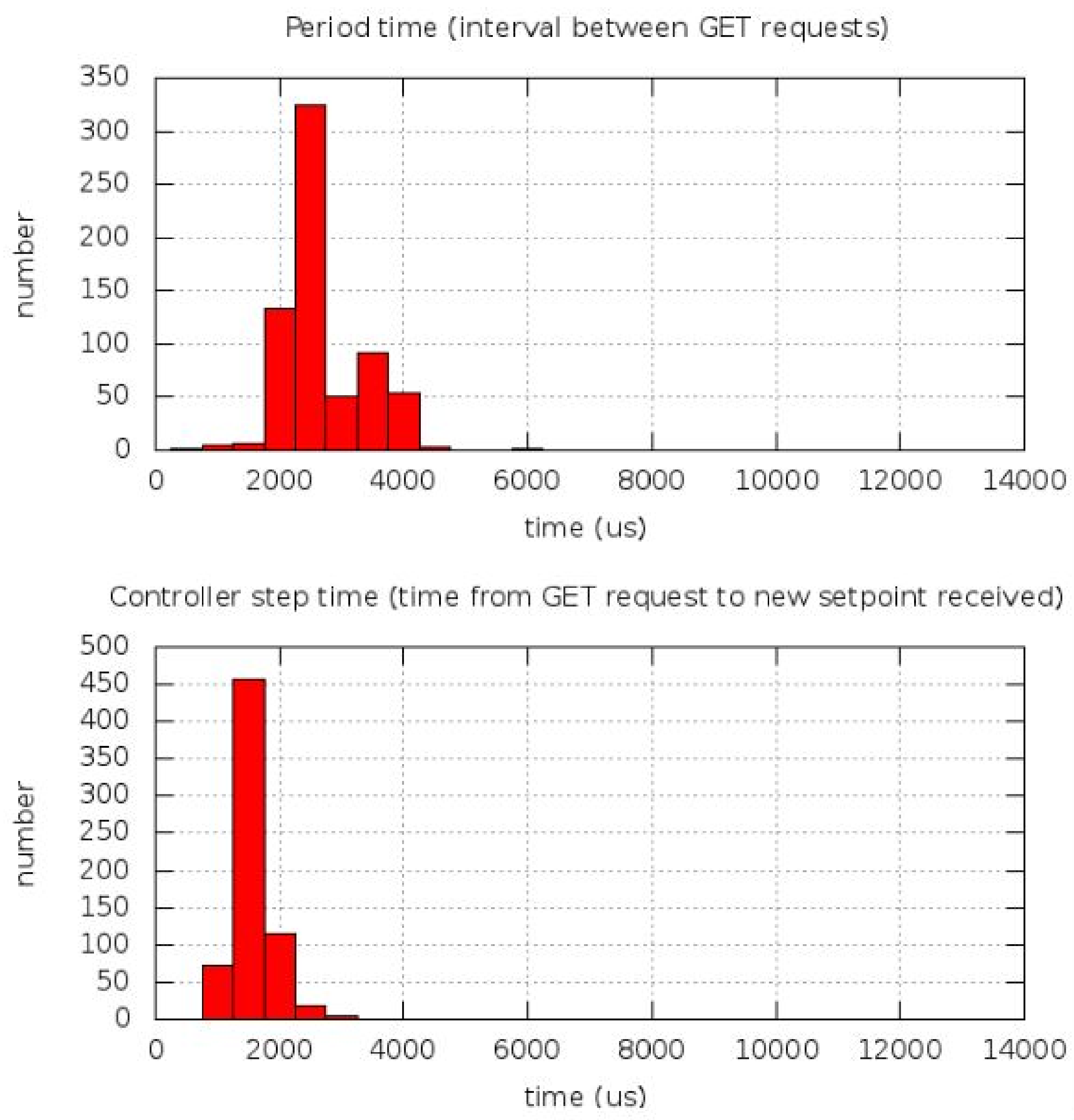


Figure 1: Task B

Delta time (dt) is the period of time over which the system measurements are sampled, integrated over, differentiated against, and it sets the frequency of adjustment. Our dt should be compatible with the rates of change and adjustment in the system under control. Having this in mind and making use of the figure 1. We know that the controller step time is the time that the controller takes to perform the calculation of u after receiving the value of y (which stays always almost the same). Being so we need to have a Period time (interval between GET requests) to be higher than the time that it takes to the controller to perform the operation. Taking a look on the figure 1 we see that the time that takes to perform the calculations is spread around 1800 us although some times the controller takes a little more time, going close to 3 ms. In order to prevent the system to require a value without finish the calculation, and since we want the smallest period time possible, we sat the value of the period to match 3ms.

In order to find the other parameters, K_p and K_i, we proceed with the trial and error method, until we found a set of values that stabilize the system around 0.3 seconds, which among others were $K_p = 1.0$ and $K_i = 805$

4 Analysis of the plots

4.1 Quality of the timing of the period and response time of the controller

In order to answer this question we are going to make a comparison between two implementations, one with the usleep function and other with the clock_nanosleep function. First of all, and how it was talked in the last question, we want the period time to be as close as possible to 3 ms (reasonable value chosen by us). So we try to request from the server the value of y every 3 ms (more time than the time to perform the arithmetic operation to acquire the value of u).

The usleep() function shall cause the calling thread to be suspended from execution until either the number of realtime microseconds specified by the argument useconds has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. **The suspension time may be longer than requested** due to the scheduling of other activity by the system.

While in the other hand the clock_nanosleep() function allows the calling thread (*pthread_get_sender*) to sleep until either at least the time specified has elapsed (with nanosecond precision), or a signal is delivered that causes a signal handler to be called or that terminates the process. Using an absolute timer is useful for preventing timer drift problems (the call is repeatedly restarted after being interrupt by signals leading to drifts in the time when it completes). The suspension time caused by clock_nanosleep() may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time for the relative clock_nanosleep() function (that is, with the TIMER_ABSTIME flag set) shall be in effect at least until the value of the corresponding clock reaches the absolute time specified by rqtp, except for the case of being interrupted by a signal.

This behavior can be observed in the figures below:

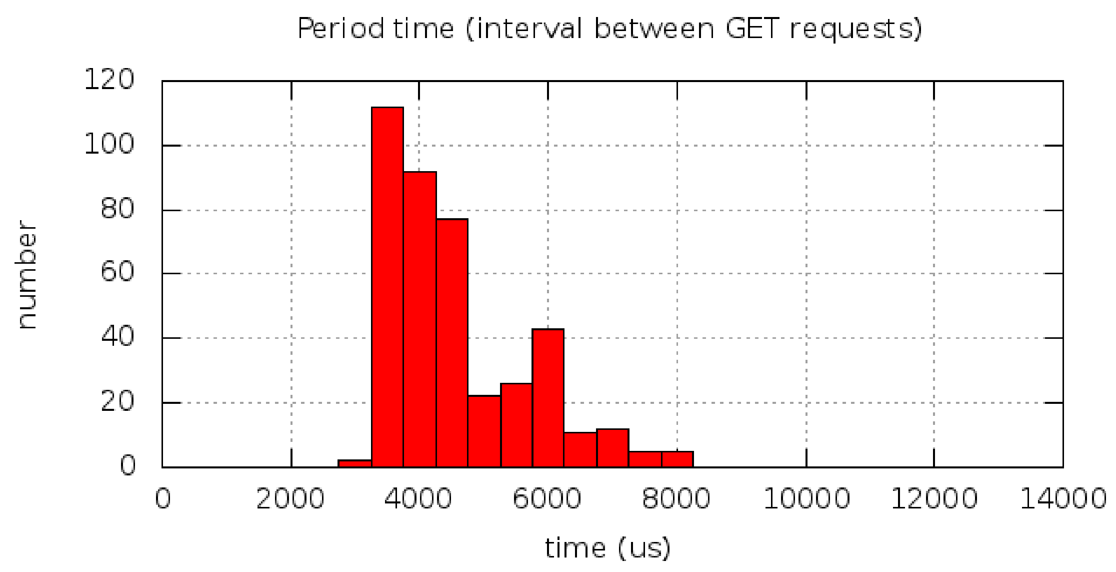


Figure 2: Using *usleep()* function

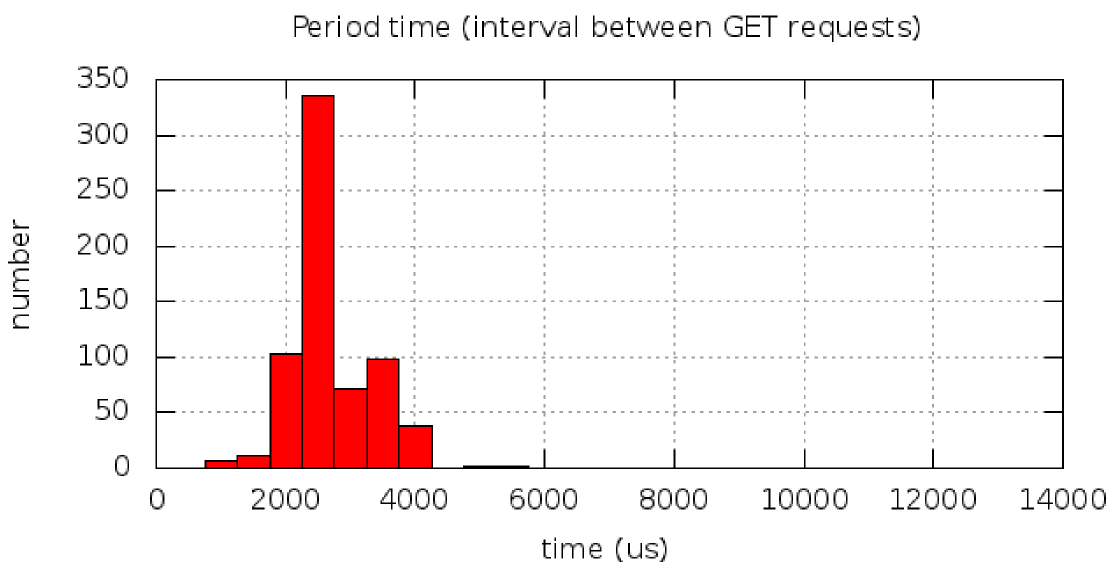


Figure 3: Using *clock_nanosleep()* function

As it can be seen in figure 2 the value of the period time is more spread than close to the value that we established, reaching 8ms of period time(inadmissible for what we want). This happens due scheduling of other activity by the system which delayed the suspension time. However in figure 3 when using clock_nanosleep(), most of the period time is gathered around the 3 ms (what we really wanted). The reason why the values deviate from the requested can be explained based on what we talked in the beginning. When putted face to face, is easy to see that the using the clock_nanosleep maintain the period time much closer to 3ms than when using the usleep(). By that we can say that the quality of the timing of the period was good when implemented with the clock_nanosleep()

funtion.

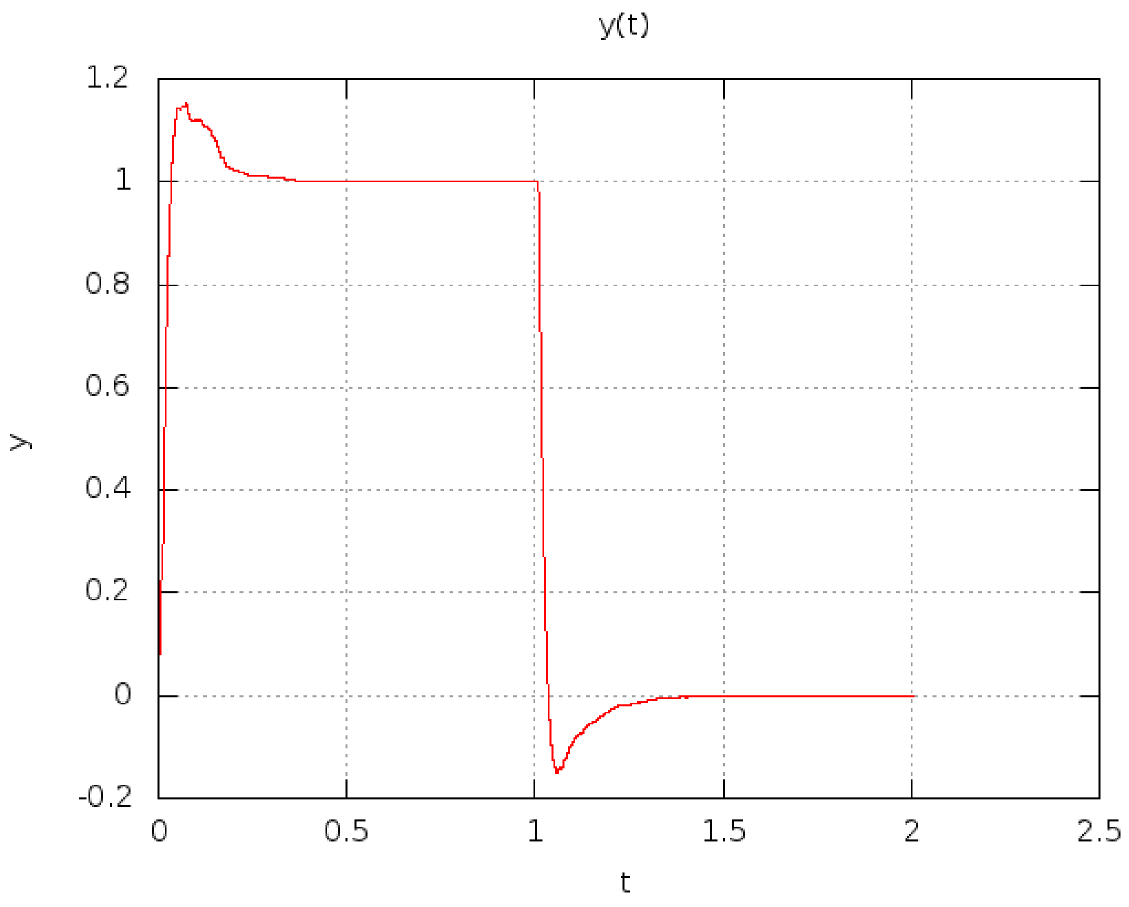


Figure 4: Using *usleep()* function

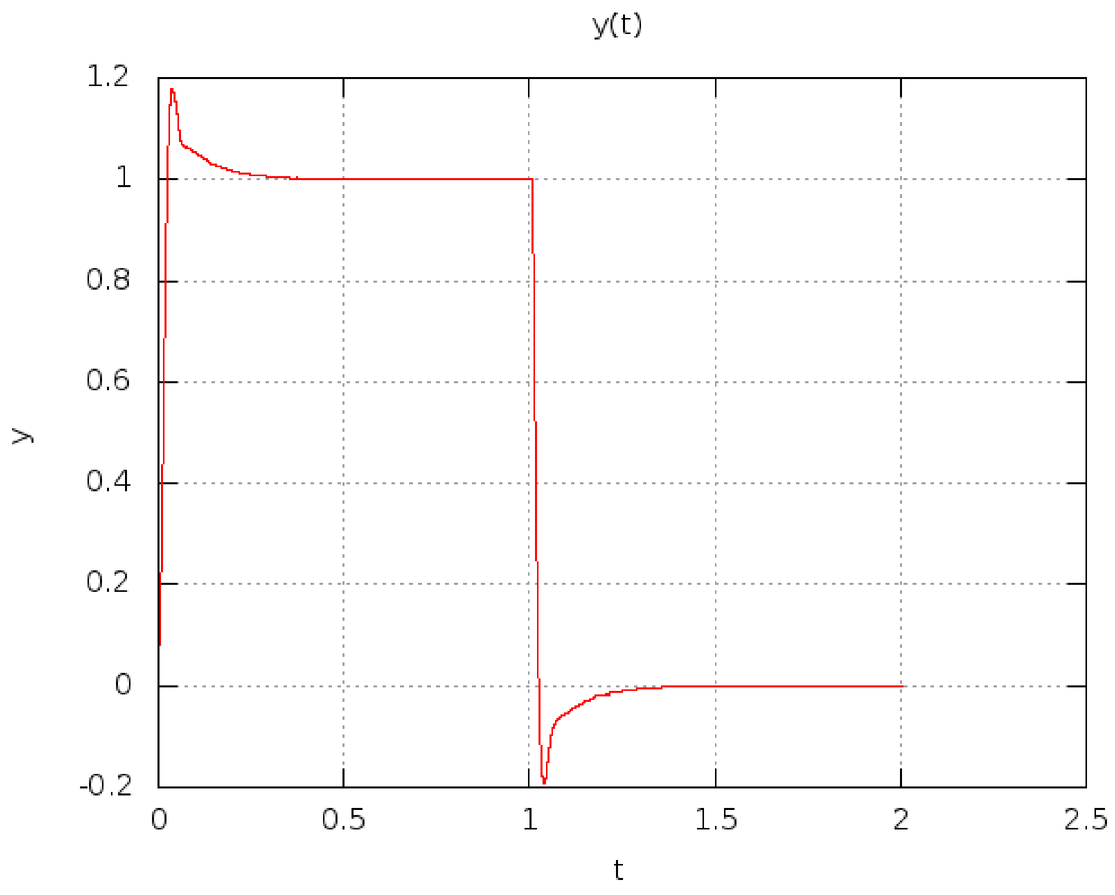


Figure 5: Using *clock_nanosleep()* function

Analyzing now the response time of the controller, for both implementations they were quite similar. They both follow the reference in a correct way and the system stabilizes before 0.3 seconds (good quality of the response). There are no obvious changes in both figure 4 and 5.

We can assume that the use of clock_nanosleep() or usleep() for the parameters chosen doesn't affect too much the response time of the controller, since they both had similar behaviours.

4.1.1 Comparison between task A and Task B

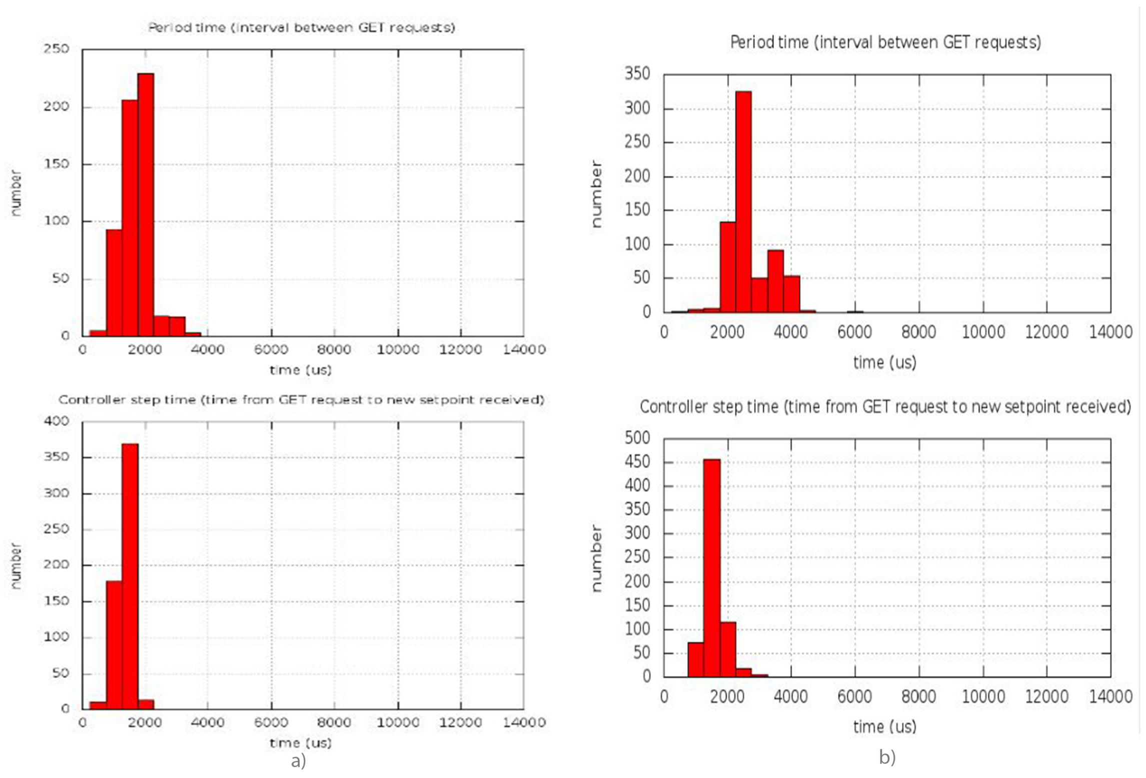
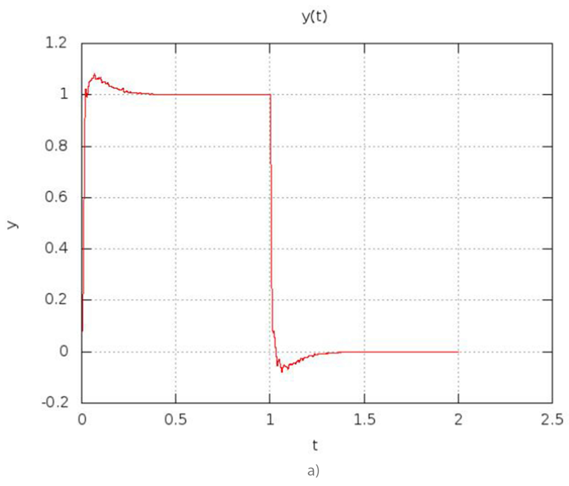


Figure 6: a) Task A - b) Task B

We will now analyze the plots of the Period Time and Controller step time from the task A to task B. In task A all the code was running in the main function, where we perform the controller calculations (using thread in Taks B). In both cases this calculations took the same amount of time in average as can be seen in figure 6 .

The difference between the period time can be explained by a change in the function used to wait before requesting the server for another value, which in task A was the usleep() while in task B was the clock_nanosleep(). As we talked before, usleep has his problems when controlling the exact time that we really want to wait. Even having the averaged period Time bigger than the controller step time, when moving to task B we decided to change the value of the sleep time from 1ms to 3ms in order to prevent unpredictable problems regarding the controller calculations and the request of the y values. which led to a bigger interval between GET requests.



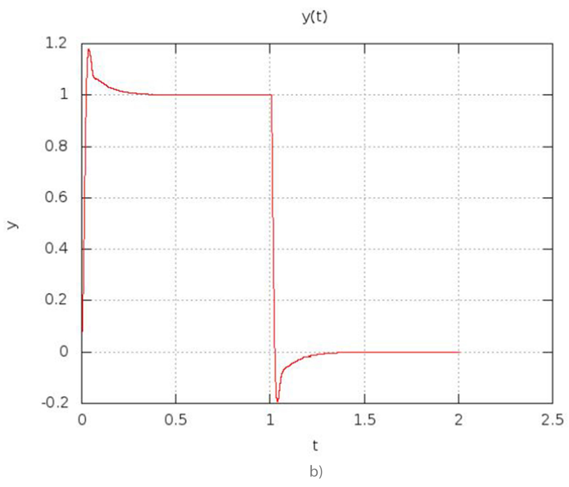


Figure 7: a) u(t) Task A - b) u(t) Task B

Besides this changes from one task to the other the response of the system, in both cases, was good, following the reference and stabilizing before $0.3s$ as can be seen in figure 7.

4.2 Response time signal

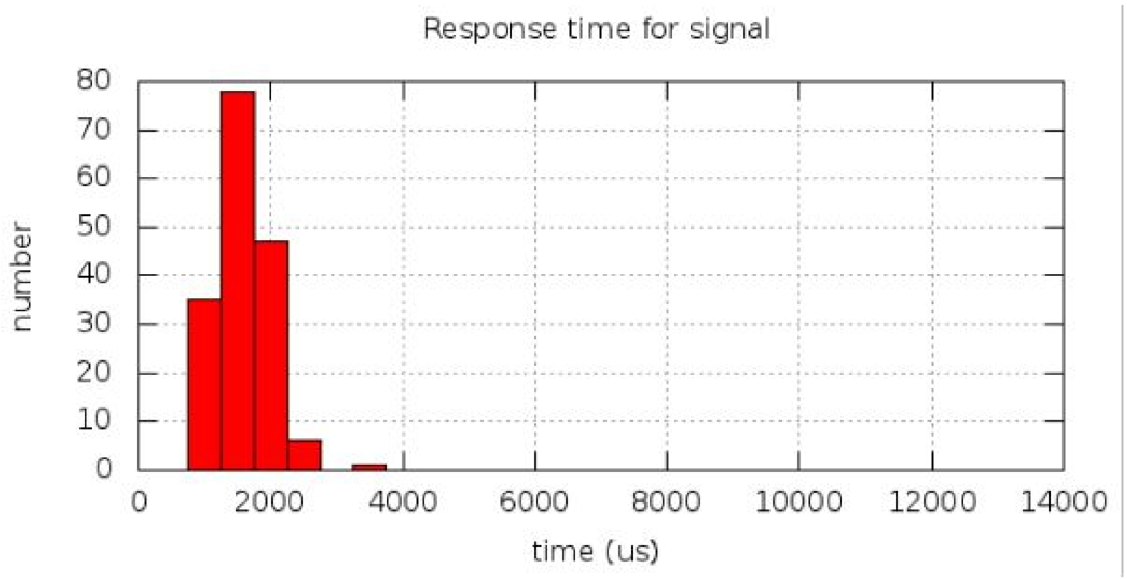


Figure 8: Response time signal

The recvBuf receives signals from the server either from get requests or sporadic signals. The quality of the signal that can be seen in figure 8 is good. The system respond to the signal around almost around the same time. The response time for signal doesn't change the response of the system, y(t), meeting the required time.

4.3 Introduction of signal task

The introduction of the signal task will force the continuous feedback control system to have some stops through its process, because the program with the signal task will need to send 2 messages to the server. Besides the u value, the program will also need to send to the server the response to the signals, so this means that the processing of generating a new value of u will be interrupted whenever the system receives a signal. Obviously this sporadic interruptions will cause a delay in the control time. Comparing the plot of the y(t) from task A (without signal - figure 7) with the one obtain when multiple threads were implemented there are no evident changes in the response of the system (y(t)). This can be explained by the synchronization method that was chosen to this project. With this synchronization method (semaphores) it was possible to respond to the signal without compromising the performance of the system to be controlled.