# Queuing Theory

**Abstract**

These notes present some findings we made when investigating the problem of optimizing the queue policy to perform fast messages requests and processing. After describing the theory which we learnt in the process we present the analysis setup we implemented and the obtained results.

## Introduction and Motivations

The problem started when we wanted to find an optimal configuration for a messaging system in terms of buffer size, number of threads and queue blocking policy, or at least to get a feeling of which variables come into play and which are the most affecting.
In particular we are exploiting the OSGi Push Stream API, which differs from the Java Stream API because it expects data to be generated and processed asynchronously. The most important difference is that the return value of a Push Stream's terminal operation is a Promise. This allows the Stream of asynchronously arriving data to be processed in a non-blocking way. This constitutes an event-based system, meaning that data events may occur far more rapidly than they can be consumed. In order to manage such data flow, the first thing you can do is using more threads, but sooner or later your system will run out of capacity, and events will have to be queued until they can be processed. Buffering is supported by the Push Stream specification, but it is useful only when dealing with short-term spikes in the event flow. When the long-term arrival rate is higher than the processing rate we can discard some of the events or fail the stream, depending on the buffer queue policy, or apply what is called *back pressure*, meaning tell to

the event source how long it has to wait before sending other events, in such a way the consumer has the time to catch up. All these information (and much more) can be found in the OSGi Push Stream Specification (Compendium Release 7).
In order to solve our problem, the first thing we did was trying to grab some general piece of information about the problem, by simply searching through the Web. We discovered that there exists a whole theory beyond this subject, the so called *Queuing Theory*, which models and derives useful expressions in order to estimate the main quantities into play. Citing *Wikipedia*:

Queuing Theory is the mathematical study of waiting lines, or queues. A queueing model is constructed so that queue lengths and waiting time can be predicted.

This applies to a huge number of phenomena, if you think about it: from waiting lines at the airport or at the supermarket, to queue when trying to get the line to a call center operator, to systems of messages requests and consumptions in a message system.
The last case is what we had in mind when we started looking into queuing theory. We had a messaging system, where messages, arriving with a certain *arrival frequency*, are put into a buffer waiting to be processed by a certain number of threads which will perform some operations on these messages taking a certain *processing time*. What we wanted to do was trying to understand which are the most important factors which come into play and affect the total system time, for instance, and whether we could optimize some of these factors in such a way to improve the overall performance of the system.

## Some Theory

In this section we are going to briefly summarize the theoretical information we found out in the process. A lot of documentation about Queuing Theory is available, thus, if you want to learn more, just take a look on the Web (as we did).
First, some useful notation. A queue system is usually identified through an acronysm of the form *A/B/C/D/E*, where:

- $A$ stands for the distribution of the arrival time, meaning how the inter-arrival time with which the data come into the system is distributed;
- $B$ stands for the distribution of the processing time, meaning how the frequency at which the system performs whatever kind of operation it needs to do on the data is distributed;

- $C$ stands for the number of parallelism we have in our system; in our case this will represent the number of threads we have to process the data;
- $D$ stands for the system capacity, meaning the total number of data which are allowed in the system (both waiting to be processed and being processed) before some blocking or rejecting policy comes into play;
- $E$ stands for the total number of data (in case the system expects a finite number of data).

The case we studied was the so called $M/M/c/d/\infty$, meaning that the requests arrive according to a Poisson process with rate $\lambda$, that is the interarrival times are independent, exponentially distributed random variables with parameter $\lambda$, and the processing time follows an exponential distribution as well, with parameter $\mu$. We are assuming to have a total number of $c$ threads to process the data and a capacity of $d$, which will be the sum of the buffer size and the number of threads.

Queuing Theory helps us computing some useful quantities, such as:

- $\overline{\mathcal{L}}_q$: the average length of the queue (in our case, how many messages, on average, are in the buffer);
- $\overline{\mathcal{L}}$: the average length of the system (in our case, how many messages, on average, are in the system);
- $\overline{\mathcal{T}}_q$: the average time spent in the queue (in our case, the average time messages spent in the buffer);
- $\overline{\mathcal{T}}$: the average time spent in the system (in our case, the average time messages spent in the whole system).

In order to compute these quantities we need first to compute the probability, at the equilibrium, that the system is in an idle state (meaning no messages in the system), and the probability that $n$ messages are in the system. These two quantities are in general denoted with $\mathcal{P}_0$ and $\mathcal{P}_n$, respectively, and, for our case, they can be written as:

$$\mathcal{P}_0 = \begin{cases} \frac{1}{\frac{\rho^c}{c!}\frac{1-a^{d-c+1}}{1-a}+\sum_{i=0}^{c-1}\frac{\rho^i}{i!}} & \text{if } a \neq 1 \\ \frac{1}{\frac{\rho^c}{c!}(d-c+1)+\sum_{i=0}^{c-1}\frac{\rho^i}{i!}} & \text{if } a = 1 \end{cases} \tag{1}$$

$$\mathcal{P}_n = \begin{cases} \frac{\lambda^n}{n!\mu^n}\mathcal{P}_0 & \text{if } n < c \\ \frac{\lambda^n}{c^{n-c}c!\mu^n}\mathcal{P}_0 & \text{if } c \leq n \leq d \end{cases} \tag{2}$$

where $\lambda$ is the number of messages arriving in a unit time, $\mu$ the number of data processed in a unit time by each of the $c$ threads, $\rho = \lambda/\mu$ and $a = \rho/c$. Using these quantities we get:

$$\overline{\mathcal{L}}_q = \begin{cases} \frac{\rho^c a \mathcal{P}_0}{c!(1-a)^2}[1 - a^{d-c+1} - (1-a)(d-c+1)a^{d-c}] & \text{if } a \neq 1 \\ \frac{\rho^c a \mathcal{P}_0}{2c!}(d-c)(d-c+1) & \text{if } a = 1 \end{cases} \tag{3}$$

$$\overline{\mathcal{L}} = \overline{\mathcal{L}}_q + \rho(1 - P_d) \tag{4}$$

$$\overline{\mathcal{T}}_q = \frac{\overline{\mathcal{L}}_q}{\lambda(1 - P_d)} \tag{5}$$

$$\overline{\mathcal{T}} = \frac{\overline{\mathcal{L}}}{\lambda(1 - P_d)} \tag{6}$$

## Understanding the theory

After collecting the information from the theory, we wanted to have an idea of how all these expressions look like, just to have a feeling of what we were dealing with. Thus we created a simple setup which allows to compute the average system time, $\overline{\mathcal{T}}$, for different configurations, namely varying the arrival rate $(1/\lambda)$, the processing rate $(1/\mu)$, the number of threads $(c)$ and the system capacity $(d)$. Here are some plots which illustrate our findings.

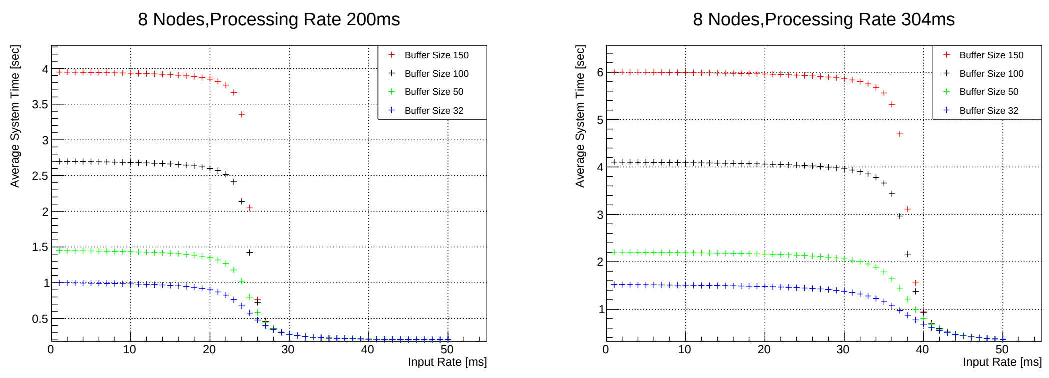

Figure 1: Average System Time (in sec) as a function of different input times ($1/\lambda$ in ms) for two fixed average processing rates of 1/200 mex/ms (left panel) and 1/304 mex/ms (right panel), 8 threads and different buffer sizes.

As you can see from the plots, regardless from the buffer size, which just regulates the maximum of the curve, the average system time remains quite stable for large values of the input frequency. When this reaches a certain *critical value*, then, we have a sudden decrease of the system time which immediately stabilizes at lower values. Looking at the same distribution for different processing rates it can be seen that the change in the curve slope depends on the ratio between the arrival and processing rate.
This was a good information for us, because this means that, if we are able to predict to some extent how fast the messages are arriving and how fast we will be in processing them, we could adjust the arrival time by delaying it in such a way to reduce the system time.

## Looking at the actual data

After obtaining some results from the theory, we needed to check whether actual data behave as predicted or whether there were factors we did not take into account in the proper way.
To do this, we performed some JUnit tests under different assumptions of arrival and processing rate distributions. As we already mentioned in the Introduction, our configuration is a bit more complex with respect to the one we treated in theory. In particular we are using OSGi Push Stream to consume the data from an event source. In our tests we considered an *event source* with a buffer of 100 elements, and a *push stream* with a buffer of 32 elements, a blocking queue policy and a back pressure time of 5 ms, which processes the data through 8 threads. We tested two queue policies to be applied to the event source:

- **Standard Blocking Policy (SBP)** in which the data enter the buffer as soon as they arrive into the system and when this is full new entries are blocked;
- **Delay Input Rate Policy (DIRP)** in which we keep track of the processing time of the last operation and of the arrival time between two consequent messages, in order to compute the ratio between the two rates and adjust the arrival time when the request comes too fast. In this way we are trying to shift all our messages to the right-hand part of the curves in Figure 1, making the average system time decrease.

We assumed different combinations of distributions for the inter arrival time ($t_a$) of the messages and the processing time ($t_p$). In particular we studied the case:

- $t_a$ taken from a uniform distribution centered at different values between 5 and 50 ms, and $t_p$ fixed at 200 ms;
- $t_a$ and $t_p$ extracted from an exponential distribution of the form $\exp^{-\lambda t}$ with $\lambda = 30$ ms and $\lambda = 225$ ms, respectively;
- $t_a$ extracted from an exponential distribution of the form $\exp^{-\lambda t}$ with $\lambda = 30$ ms and $t_p$ extracted from a log-normal distribution with mean $\mu = 225$ ms and deviation $\sigma = 50$ ms;
- $t_a$ fixed at 30 ms for the first 300 messages and then fixed at 15 ms for the last 200 messages, while $t_p$ fixed for all messages at 200 ms;
- $t_a$ fixed at 30 ms for the first 300 messages and then fixed at 15 ms for the last 200 messages, while $t_p$ extracted from a log-normal distribution with mean $\mu = 225$ ms and deviation $\sigma = 50$ ms;
- $t_a$ fixed at 30 ms and $t_p$ varying from 200 ms for the first 300 messages to 500 ms for the last 200 messages;
- $t_a$ extracted from an exponential distribution of the form $\exp^{-\lambda t}$ with $\lambda = 30$ ms and $t_p$ varying from 200 ms for the first 300 messages to 500 ms for the last 200 messages.

In the following we are going to summarize the results of our tests.

## Case 1: $t_a$ uniform/$t_p$ fixed

In this test we wanted first to study the distribution of the average system time as a function of the arrival frequency, in order to check whether the data were well described by the theoretical distributions we obtained.
In order to do that, we generated 500 messages with a variable input frequency, extracted from uniform distributions centered at different values between 5 and 50 ms, and with a fixed processing frequency of 1/200 mex/ms. We then computed the average input frequency and the average system time. We repeated the same experiment 50 times, for both the queue policies under study. The results are shown in the following figure.

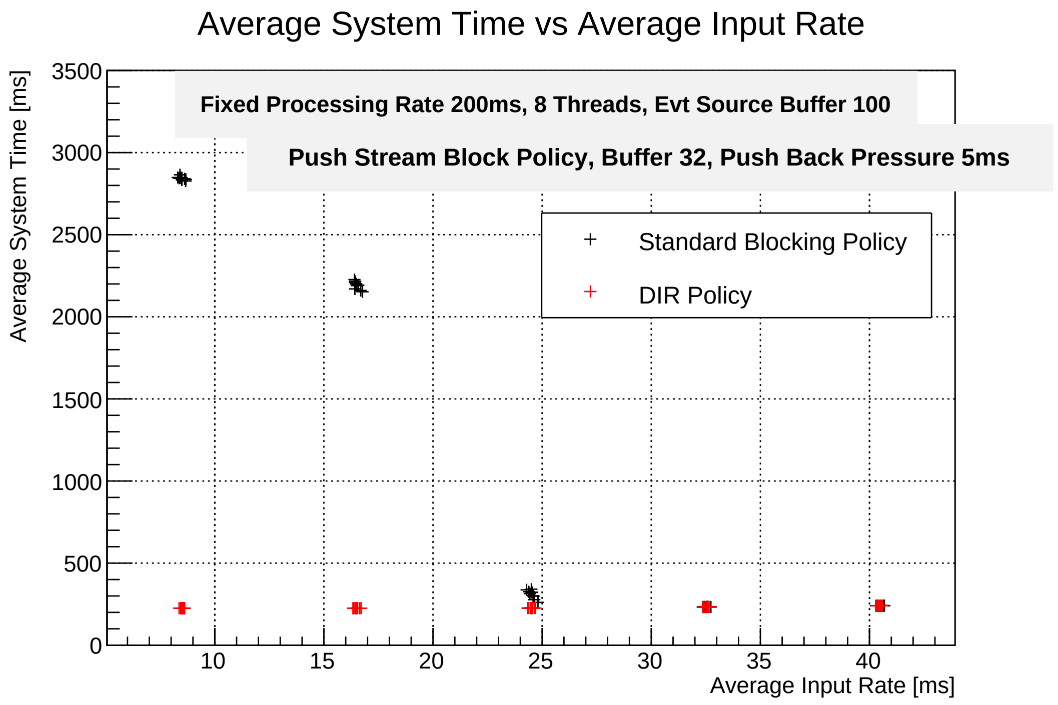

Figure 2: Average system time (in ms) vs average input time (in ms) for a fixed processing time of 200 ms. The SBP (black points) is compared to the DIRP (red points).

As you can see from the plot, the distribution for the SBP well reflects the one we was expecting from the theory, both in shape and in the overall scale. Furthermore, applying the DIRP we can see that the average system time for events which are arriving at higher frequency in the system can be drastically reduced at the level of those arriving slower.

## Case 2: $t_a$ exponential/$t_p$ exponential

In this test we repeated the procedure described in the previous case, but extracting now both $t_a$ and $t_p$ from an exponential distribution. In order to do that we extracted a random number $u$ from 0 to 1 and then compute our exponential random number as:

$$-\ln(u) \times \lambda \tag{7}$$

where $\lambda = 30$ ms for $t_a$ and $\lambda = 225$ ms for $t_p$. The results we obtained are shown in the following figure.

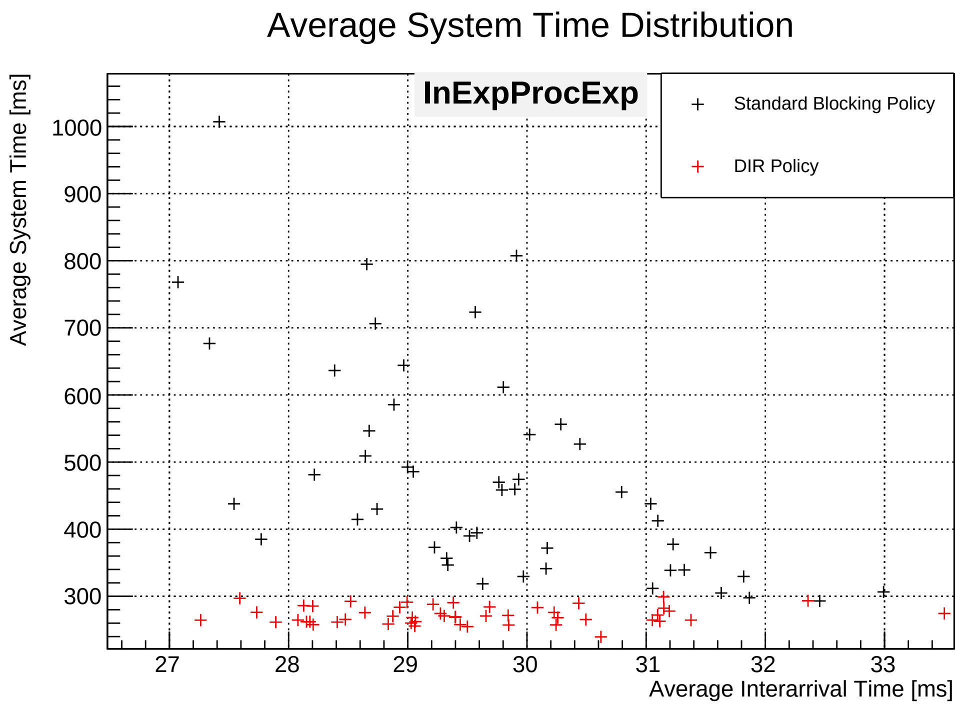

Figure 3: Average system time (in ms) vs average input time (in ms) for an exponentially distributed arrival and processing time. The SBP (black points) is compared to the DIRP (red points).

Also in this case, it can be seen that the DIRP results in a more stable average system time with respect to the SBP.

## Case 3: $t_a$ exponential/$t_p$ log-normal

In this case we assumed $t_a$ distributed exactly as in the previous case, while for $t_p$ we assumed a log-normal distribution with a mean of 225 ms and a deviation of 50 ms. This distribution takes into account the fact that processing time distributions are typically skewed to the right, and never negative. In order to generate a random number extracted from a log-normal distribution with mean $\mu$ and deviation $\sigma$ we first generate a random number $y$ extracted from a Gaussian distribution with mean $m$ and deviation $s$ given by:

$$s^2 = \ln(1 + \frac{\sigma^2}{\mu^2}) \tag{8}$$

$$m = \ln(\mu) - \frac{s^2}{2} \tag{9}$$

and then we obtain our log-normal distributed number $x$ as:

$$x = e^y \tag{10}$$

The results we obtained in terms of average system time are shown in the figure below.

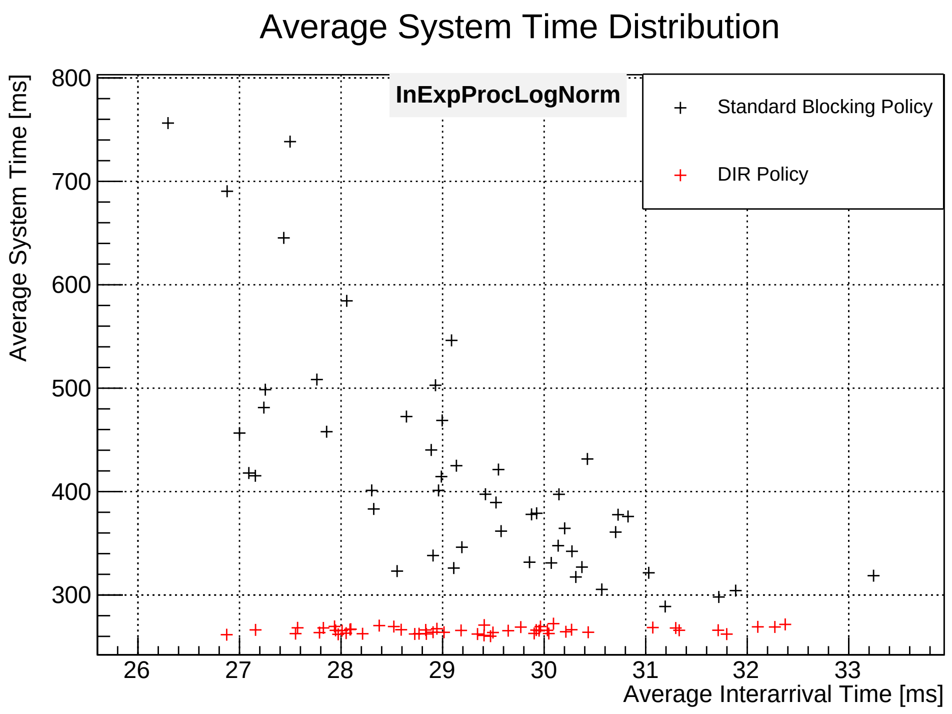

Figure 4: Average system time (in ms) vs average input time (in ms) for an exponentially distributed arrival time and a log-normal distributed processing time. The SBP (black points) is compared to the DIRP (red points).

Once again, the DIRP mantains the average system time to a lower and more stable value with respect to the SBP.

## Case 4: $t_a$ changing/$t_p$ fix

With this and the subsequent test we wanted to investigate what happens to the system time in case of a sudden increase of the arrival frequency. For this we generated 500 messages with both the policies under study, keeping the processing time fixed at 200 ms. The first 300 messages are generated with a frequency of 1/30 mex/ms while the last 200 with a frequency of 1/15 mex/ms. We stored for each of the messages the system time and an id, which simply indicates the order the messages were created. The results are shown in the figure below.

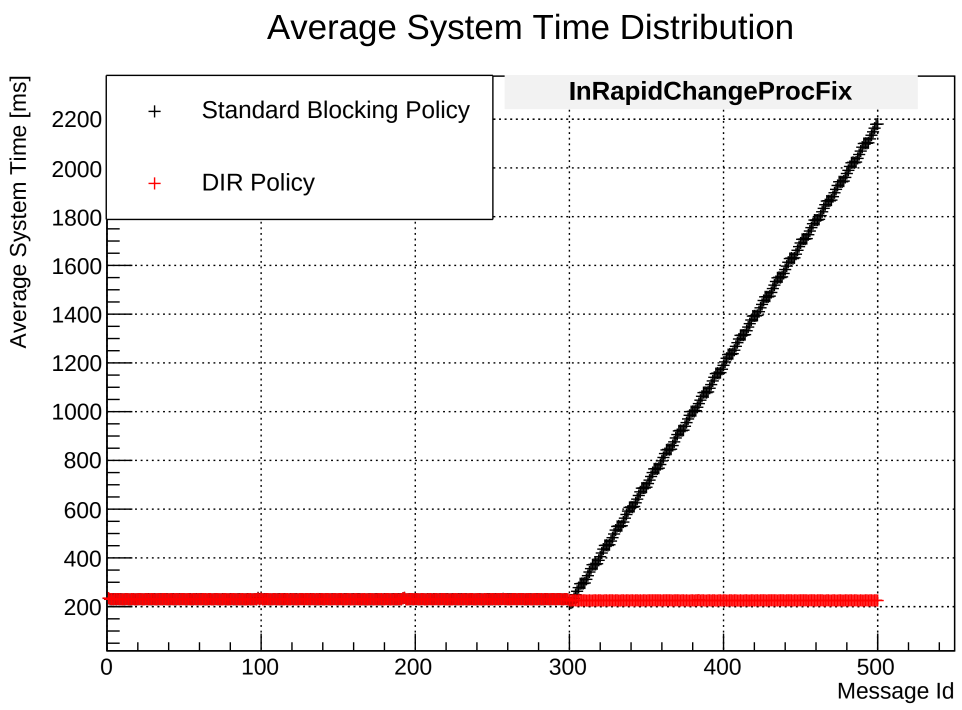

Figure 5: Average system time (in ms) vs message creation order for a fixed processing time of 200 ms, with a sudden change in input frequency. The SBP (black points) is compared to the DIRP (red points).

As you can see from the two curves, as soon as the input frequency is 1/30 mex/ms, the two policies behave almost identically, which is what we expected, since in this case the ratio between the input and processing frequencies is below the critical value, so we would be in the right-hand side of Figure 1. When the input frequency suddenly increases, meaning messages start to come into the system in a faster way, the system time increases quite drastically for the SBP, while for the DIRP it remains stable.

## Case 5: $t_a$ changing/$t_p$ log-normal

This test is exactly the same as the previous one, except for the fact that now the processing time is not kept fixed, but is assumed to be log-normally distributed. The results are shown in the figure below.

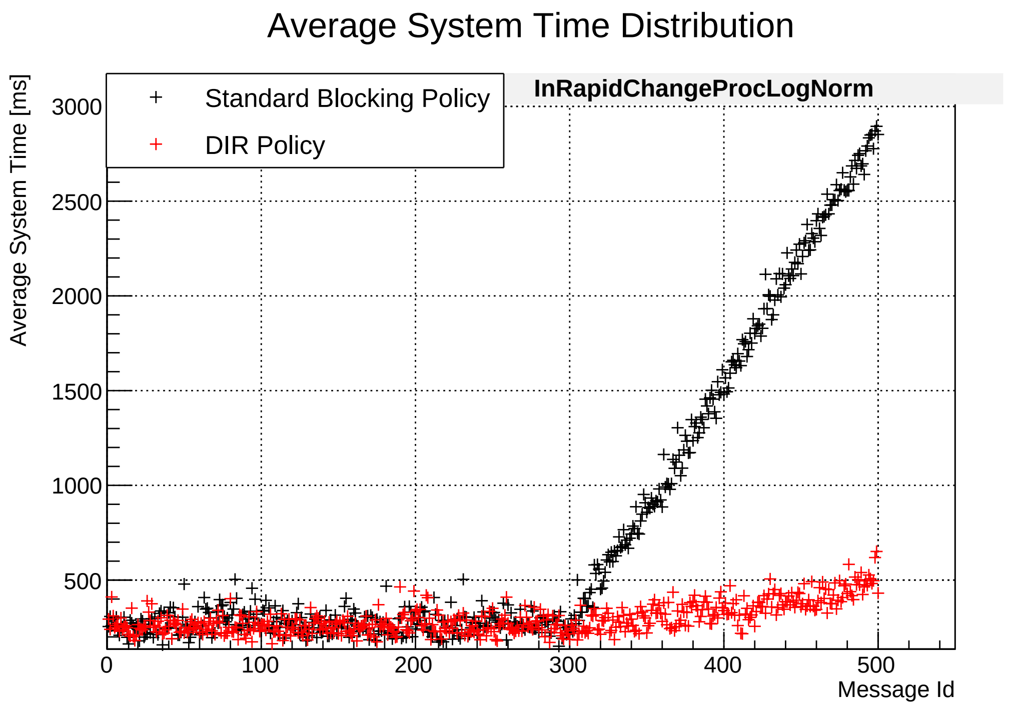

Figure 6: Average system time (in ms) vs message creation order for a log-normally distributed processing time, with a sudden change in input frequency. The SBP (black points) is compared to the DIRP (red points).

Also in this case, we can see how the DIRP makes the system time to remain quite stable with respect to the case of the SBP.

## Case 6: $t_a$ fix/$t_p$ changing

In this and the next test we wanted to investigate what happens to the system when the processing frequnecy suddenly decreases. For this we generated 500 messages for both the policies under study, keeping the input rate fixed at 1/30 mex/ms. The first 300 messages are processed with a processing time of 200 ms, while the other 200 with a processing time of 500 ms. We stored for each of the messages the system time and an id, which simply indicates the order the messages were created. The results are shown in the figure below.

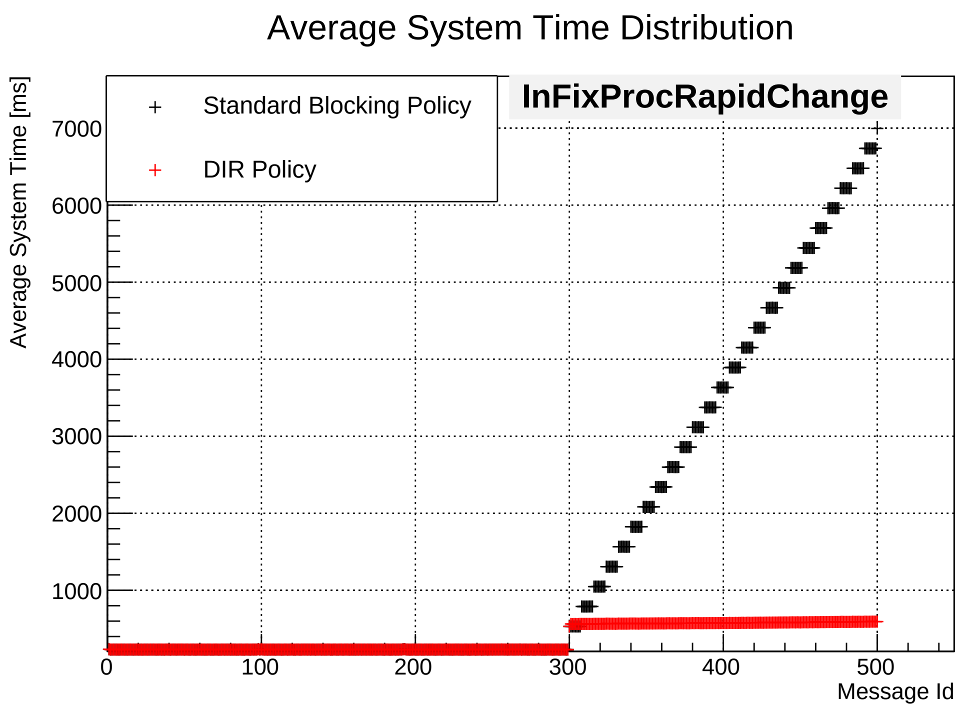

Figure 7: Average system time (in ms) vs message creation order for a fixed input time of 30 ms, with a sudden change in processing frequency. The SBP (black points) is compared to the DIRP (red points).

Also in this case it can be seen how the two policies perform in the same way in terms of system time when the processing rate is kept constant. When the change occurs then, the DIRP increases a bit the system time, which then remains stable under the 1 sec level; the SBP, instead, passes from a system time below 1 sec to 7 sec.

## Case 7: $t_a$ exponential/$t_p$ changing

This test is the same as the previous one, except for the fact that we assumed $t_a$ exponentially distributed. The results are shown in the figure below.

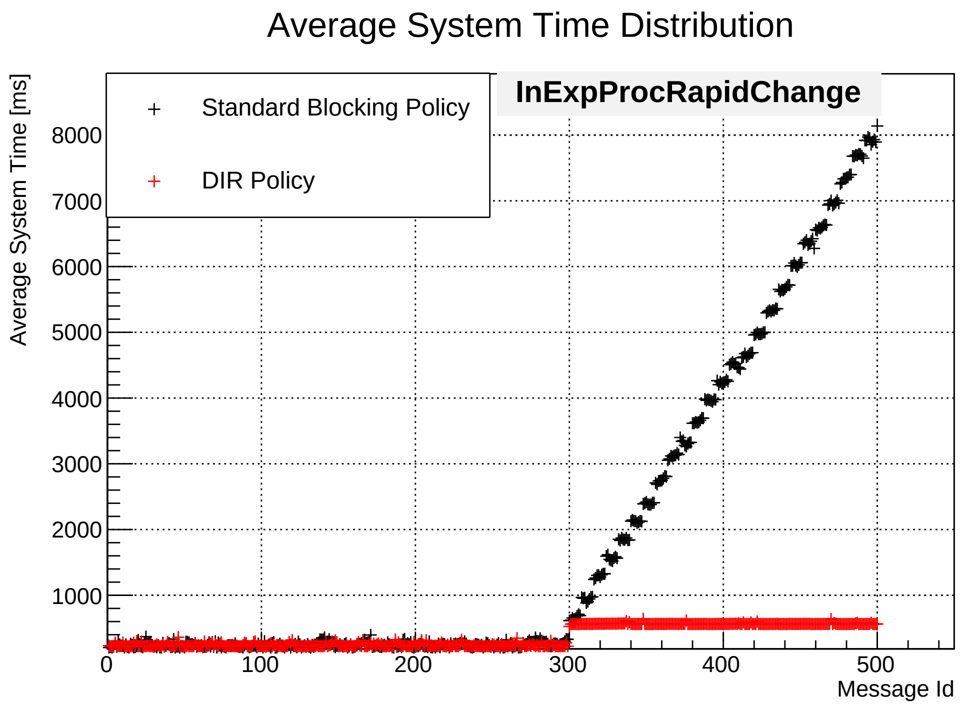

Figure 8: Average system time (in ms) vs message creation order for an exponentially distributed input time of 30 ms, with a sudden change in processing frequency. The SBP (black points) is compared to the DIRP (red points).

Despite some more fluctuations, not really appreciable at the scale of the plot but expected due to the fact that in this case the arrival rate is not constant, the results is again the same: the DIRP mantains the system time to a more stable value even after the sudden change of the processing time.

## Conclusions

We presented the study we did in order to better understand how the average system time behaves as a function of the arrival and processing rates, and of the other variables into play, such as the buffer size. We briefly described the theory we follwed in order to develop a proper new queue policy, suitable for our needs. We tested such policy and compared it to a standard blocking policy under different assumptions for the distributions of the arrival and processing rate, showing a significant improvement in terms of both stability and overall performance.