Università
Ca'Foscari
Venezia

Master's Degree programme

in Computer Science

"Software Dependability and
Cyber Security"

On the optimality of size-based scheduling in networking

Supervisors
Prof. Andrea Marin
Prof. Sabina Rossi
Candidate
Giorgio Magnan, 846314
Academic Year
2017 / 2018

Abstract

In recent years flow scheduling on the Internet has attracted a lot of interest in scientific research, in particular the study of how the distribution of flow size can influence system performance. Many queuing models have been studied and designed to prove that size-based schedulers improve performance for small flows without degrading overall system performance. On the other hand, however, it has been demonstrated that it is not easy to identify small size flows.

In this thesis we propose a new queuing system model, starting from the study of existing ones, with a multiple level priority queue that can separate small flows from bigger ones in order to prioritise them. We derive the *mean response time* for the job conditioned on their sizes and we compare them with those of the systems already studied in the scientific literature. Our results have been validated by using a stochastic simulator. Finally, we discuss an idea to implement the model in the reality analysing some schedulers implemented in Linux systems.

Contents

List of Figures

Listings

List of abbreviations

AF Assured Forwarding

AQM Active Queue Management

CoDel Controlled Delay Management

CPU Central processing unit

DiffServ Differentiated Services

EF Expedited Forwarding

FB Foreground Background

FCFS First Come First Served

FIFO First In First Out

FqCodel Fair Queuing Controlled Delay

LAS Least Attained Service

IETF Internet Engineering Task Force

IntServ Integrated Services

IP Internet Protocol

OS Operating System

PHB Per-Hop Behaviours

PS Processor Sharing

QoS Quality of Servise

RED Random Early Detection

RR Round Robin

RSVP Resource Reservation Protocol

SJF Shortest Job First

SJN Shortest Job Next

SPN Shortest Process Next

SRPT Shortest Remaining Processing Time

TCF Target Communications Framework

TCP Transmission Control Protocol

UDP User Datagram Protocol

Chapter 1

Introduction

In recent years computer networks have grown exponentially, more and more devices are connected and the number of services accessible via the network have increased. Many applications like voice call, streaming and online games require connections with low delay, while other applications like p2p try to exploit the available bandwidth at the maximum possible. For these reasons, many studies have concentrated on improving performance of routers and switches in order to maximise the amount of data transferred, focusing above all on scheduling algorithms.

Routers and switches can assign a class to each packet (or flow) that they route in such a way that it can determine its priority and therefore the order in which it will be served. Many scheduling algorithms use classes to assign priority or bandwidth to a specific job. Among these, some algorithms assign priority statically and other dynamically (as we will see in Section 3). Many scheduling algorithms have been studied in these years and many others have been designed and implemented to improve the performance of systems.

What clearly emerges from this literature is that the distinction between large and small TCP flows play a crucial role in the minimization of the expected response time. However, with the current TCP/IP network design it is impossible to understand the TCP flow size in advance, and the possibility of changing the protocols in order to allow for this information to be embedded in the packets is unfeasible for at least two reasons. First, the sender could give the routers wrong information about the size either intentionally or because it cannot know it in advance. The second reason is that changing the TCP/IP architecture appears to be prohibitive and similar attempts previously proposed failed. Therefore, the main goal is that of proposing a discipline capable of distinguishing large and short flow sizes by using network statistics.

Among the solutions proposed in the literature we will focus on the multi-level systems proposed by Kleinrock in [1, 2]. Although the multi-level queues have been proposed several years ago, the literature in networking seems not to have taken full advantage of this discipline.

The idea consist in the introduction of several thresholds to distinguish the flows based on the resources used up to a certain time. Under some conditions on the hazard-rate of the distributions of the job sizes, it can be proved that giving priority to the jobs that have requested less resources up to a certain epoch reduces the overall expected response time.

From a practical point of view, it is important to understand if this kind of scheduler is possible to implement on modern routers. Many routers on the network use an operating system that is a light version of Linux since it provides a modular architecture. As a consequence, it is easy to add and remove

modules to extend and modify scheduling algorithms and other networking operations.

1.1 Objective of the thesis

The purpose of this thesis is to study, analyse and model a set of scheduling algorithms proposed in the scientific literature and compare them with those already implemented and used in real systems. In particular our attention will be focused on *size-based* scheduling algorithms. We introduce a queuing system model with a multilevel scheduling structure able to recognise small flows from large ones in order to prioritise the first ones. We describe the structure and functionality of our model and we compute all the formulas to evaluate some performance indices to understand the effectiveness of our system. We design a simulator in order to verify the theoretical results and we propose a route to implement the scheduling discipline in a Linux OS after analysing the schedulers already present in this system.

1.2 Contributions of the thesis

The major contributions of this thesis can be summarised as follows:

- In [2] the author proposes an analysis of the multilevel queue which relies on the solution of a system of differential equations which is not trivial to address. The first contribution of this thesis is that of providing such a solution for jobs whose size follows a negative exponential random distribution.

- As a second point, we give an approximate solution for the multilevel queue model in the case of non exponential job sizes. We evaluate the accuracy of such an approximation by comparing the estimates of the simulation with those derived by the approximate model. According to our experiments, we asses that the error introduced by the approximation is around 3%.

- We develop two different simulators to compare the results obtained by the theoretical model with those obtained from the simulations. The first simulation model resembles the queuing model and has been mainly used to evaluate the approximations proposed in the previous step. The second simulator is closer to the actual implementations of the networking system and models the packet arrival process with an implementation of the considered queuing discipline.

- As a final step, we analyze how the networking scheduling works in Linux systems, focusing our attention on some scheduling algorithms already implemented. We discuss the implementability of size based scheduling in such real systems.

1.3 Structure of the thesis

This thesis is organised in 9 chapters. In Chapter 2 we present the general problem of scheduling in networking which is a topic addressed by a lot of scientific researches. Moreover, some QoS techniques are described.
Chapter 3 presents the main scheduling algorithms currently present in the scientific literature.

Chapter 4 presents some basic concept of the queuing theory that are necessary for the understanding of the entire research work.
Chapter 5 shows our model, with the description of the architecture, the calculations of the system indexes and the application of the model to some distributions taken into consideration.
Chapter 6 shows the simulation of the model and the results obtained by our system.
Chapter 7 presents the traffic control of Linux, taking into account queuing disciplines, classes and filters.
Chapter 8 analyses some schedulers used by default by the operating system.
In the last chapter our data (obtained from theoretical calculations and simulations) are compared with those of the other models in the scientific literature.

Chapter 2

Scheduling in networking

2.1 Introduction

The aim of this chapter is to give a general idea about the problem of the scheduling in networking. Starting from what it has been studied from the scientific literature, we have focused our attention on the works that analyze the distribution of the flow dimension. Secondly we give an introduction to the quality analysis in network systems and finally we report some existing technique to analyse the quality of service.

Scheduling of flows in the network is a subject that has attracted particular interest in scientific and technological research.

Modern computer network still rely on the TCP/IP architecture. This consists of two main protocols of the transport layer: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). While the latter is a mere incapsulation of the IP protocol and therefore provides an unreliable and connectionless service, TCP is a connected and reliable protocol. All

the segments belonging to the same TCP connections can be easily identified by the network devices and therefore we can talk about TCP flows or, more simply, flows. TCP is widely used in nowadays Internet services even in those domains for which the literature suggested to use UDP. At the current time, video streaming of YouTube and that of other providers like Netflix use TCP flows for the transmission of massive amount of data. Similarly, audio/video call software prefer to exploit the reliability of TCP rather than handling the characteristics of UDP. Finally, the relatively small pages and web elements of the world-wide-web are accessed by the browser thanks to the TCP. As a consequence, the size of the flows that populate the Internet varies a lot according to the content. A TCP flow can range from few bytes to some gigabytes.

Traffic in network is composed of flows. We can divide these flows in two groups, the first one is composed by a few number of flows that transport the majority of the traffic of the network (also called *elephant flows*), the second group is composed by a huge number of flows that carry a very small traffic (also called *mice flows*). Elephant flows have a huge impact on network performance, and for this reason it is important to identify them when we develop a network system.

Many studies have focused on the analysis of the distribution of the dimensions of the flows that pass through the networks. As a consequence, many researchers have focused their attention on designing scheduling algorithms that are able to recognise the small flows from the large ones in such a way to maximise the performance. All these studies have the aim to minimize the delay of small flows without degrading to much the performance of big

flows. But, as proved in many articles, it is not easy to understand the real dimension of a flow. For example, in [3], the authors try to understand the size of the flows by sampling the packets that enter in the router, while others like Least Attained Service (LAS) [4], try to reach the same goal without having any information about the size of the flows.

The performance index on which the researchers have focused is the average response time. For example, in [5], the authors prove that Shortest Remaining Processing Time (SRPT) minimise this index, in [6] they prove that a multilevel processor sharing system is always better or equal with respect to a processor sharing scheduling algorithm under the assumptions that the hazard rate of the job size distribution is decreasing.

Other studies have focused their attention on the relationship between hazard rate of the job size distribution and the average response time. In [7] and [8] the authors prove that some schedulers (like Foreground Background (FB) and LAS) minimize the average delay if the hazard rate of the job size distribution is decreasing.

Remark (Hazard rate)**.** *The hazard rate is a theoretical measure that examines the probability of occurrence of an event in an interval of time. For a random variable X, with cumulative density function (cdf) $F(t)$ and probability density function (pdf) $f(t)$ we define it as:*

$$h(t) = \frac{f(t)}{1 - F(t)} = \frac{f(t)}{R(t)}$$

2.2 Quality of service in networking

The term quality of service (or QoS) is used to denote some parameters and measures associated with the performance of a system. Typically, the parameters that are analysed are transmission delay, jitter, bit rate, throughput, and packet loss.

QoS can be associated with different types of network traffic, it can take into account the priority to be assigned to each flow and it can be used for the management of congestions.

The Internet Engineering Task Force (IETF) has proposed a lot of methods to achieve a certain quality of service in networking: the most important are Best Effort, Integrated Services (IntServ) and Differentiated Services (DiffServ).

2.2.1 Best Effort

The Best Effort model is the easiest one and is used by the Internet Protocol (IP). Each node of the network tries to send a packet to the destination but it does not guarantee that the data arrive. Data can be delivered later or even may not arrive at destination: if they arrive at destination there is not even the guarantee that they will arrive in the order in which the sender sent them.

It does not provide any type of QoS mechanism and for this reason it is very easy to implement but it is not usable in real time systems or in other networks with limited resources[9, 10].

2.2.2 Integrated Services (IntServ)

Integrated Services (IntServ) was introduced as a standard for the analysis of quality of service in 1994 by IETF. It is based on flows, each node asks the network to have a resource reservation for a flow. In this way, when an application receives a resource it is possible to predict the behaviour of the network because the resource is guaranteed. In the reservation phase each node specifies the minimum bandwidth, maximum acceptable delay and some other parameters. The network has to confirm the request and if the answer is affirmative, it must provide the resources required for all the duration of the transmission [9, 11].

It has not been widely used in networks for different reasons. First it assumes that each node maintains a history with all the flows and so it is hard to use it in fast growing networks. Second it requires a strict standard for the description of flows and it needs the use of Resource Reservation Protocol(RSVP) [12].

2.2.3 Differentiated Services (DiffServ)

Differentiated Services (DiffServ) was born as an alternative to *IntServ* due to the complexity and massive use of resources of the latter. Differently from the previous method, it groups flows that enter the network into classes. In this way, each node must not keep track of the flows that have crossed it as it happen in *IntServ*. Each packet is analysed by a classifier that inspects some parameters like source and destination address, traffic type or other characteristics to assign it to a class. Moreover, there is no need to reserve

resources before using them since it is based on statistical preferences [9, 10]. Each router implements a Per-Hop Behaviours (PHB) specifying the type of forwarding for each type of class, the most common are:

- Default PHB: it is usually the Best Effort traffic model;
- Expedited Forwarding (EF): low loss and low latency traffic;
- Assured Forwarding (AF): it guarantees the delivery under some predefined conditions;

This model is easy to use in scalable networks, it has an easy configuration, it reduces the overhead of maintaining a history of flows for each node and it can process the traffic more easily than *IntServ* [9, 13].

Chapter 3

Scheduling disciplines

In this chapter, we present some of the most widely used scheduling algorithms, we classifying them into two different categories depending if they work independently or dependently with the size of the jobs. Henceforth, we use the terms job, process and customer as synonyms to denote a unit of work for the server. In networks, traffic consists of a set of packets that move from one point to another thanks to the work of the routers. At each router, incoming packets are classified and then resent out based on the priority assigned, in particular packets with high priority have precedence with respect to the others. Nowadays, there are two different type of Internet Protocol (IP), one is Transmission Control Protocol (TCP) and the other is User Datagram Protocol (UDP). The TCP is a connection oriented protocol while UDP is a connectionless protocol. From this point forward we define a flow as a stream of packets that belong to the same TCP connection.
Scheduling algorithms are introduced to manage the available resources, they are fundamental for improving the performance of a system. Combining

scheduling and QoS we can regulate the traffic in a system, we can know how many packets a sender can send, how many packets an intermediate node can store and the number of flows that the system can support.
Scheduling algorithms are divided in two different sets, one consisting of the schedulers that takes decisions on the priority of the packets based on their sizes and one that contains those that does not take into account the dimension of the jobs and decides their priority statically.

3.1 Scheduling discipline independent of the job size

Scheduling algorithms that are independent of job sizes are the easiest to analyze and implement. These algorithms can be based on the order of arrival of the customers in the queue, giving more priority to those that arrived first (First In First Out or First Come First Served) or can assign a quantum of time that can be cyclically used by each job in queue (Round Robin). When the quantum of time shrinks to 0 the RR discipline takes the name of Processor Sharing (PS) and models the situation in which each job in the system receives the same amount of computational power.

3.1.1 First In First Out (FIFO)

First In First Out (FIFO) or First Come First Served (FCFS) is the simplest scheduling algorithm. It consists of only one queue in which customers are stored in the order of arrival. In this way, customers that arrive first have

higher priority than the last ones and it is impossible to change the priority of jobs already in the queue.

Given the easiness of implementation, FIFO discipline is widely used, but it is not good to use it in batch systems where the jobs can have size with different order of magnitude (minutes, days or weeks): in fact, if a job of small dimension comes in queue after a bigger job, it has to wait the completion of the bigger one [14]. For this reason, although FIFO discipline is still widely adopted for scheduling TCP flows, it is known to have poor disciplines and should be avoided unless some assumptions on the homogeneity of the flow size can be formulated.

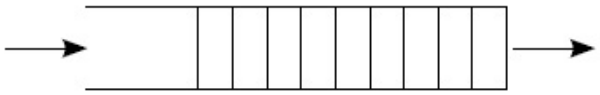

Figure 3.1: Example of FIFO queue

3.1.2 Round Robin (RR) and Processor Sharing (PS)

Round Robin (RR) is another simple scheduling algorithm in which the computational resource is assigned to each job in turn for a certain portion of time (*quantum*).

The scheduler maintains a queue of ready jobs, it takes the job from the head of the queue and processes it. If a job is not completed until the quantum of time, then it is put at the end of the queue (the scheduler uses a circular queue). The difference between RR and the FIFO algorithm is the assignment of a quantum of time for each job. In fact, while the FIFO extracts the

job from the head and completes it before moving on to the second job, the RR extracts the job from the head, works it for the quantum of time and if it is not finished it puts it back in the queue and moves on to the second. Each job can be processed more than once before being completed[14, 15].
This algorithm is widely used because it is easy to implement (it has only a circular queue and a constant that indicates the quantum of service provided each time) and it also solves the problem of starvation. Starvation occurs when a customer is not able to use a resource because it is used by other customers with higher priority that does not release it [14].
Although it is widely used, the RR algorithm has some critical issues in the choice of the quantum of time to be assigned to each job. In fact, if the quantum is too small it increases the overhead of the CPU while, if the quantum is too big, it increases the response time of the system (if the amount of time is too big, the system behaves like a FIFO system). Many studies have been made on the choice of the amount of time to assign, some algorithms, instead, change the size of the quantum dynamically in order to improve the performance of the system [16].
A specialisation of this model is the Processor Sharing policy (PS), in which each job in queue receive an equal quantum of time that tends to zero. In this way, the jobs perceive a sharing policy of the computational unit in which each one receives the same proportion of computational power.

3.2 Size based scheduling

Size based scheduling disciplines are used to improve the response time and reduce the waiting time to some categories of jobs. They are more complex to analyze and implement with respect to the previous types of scheduling algorithm and they base their decisions on the size of the job, which is retrieved dynamically. There are algorithms that give priority to smaller jobs (SJF), others give priority to job whose remaining processing time is the smallest (SRPT), others try to divide small and big jobs in order to minimise the response time of the small jobs using a multilevel scheduling structure and sampling the incoming packets (multilevel size based scheduling).

3.2.1 Shortest Job First (SJF)

Shortest Job First (SJF) (also known as Shortest Job Next (SJN) or Shortest Process Next (SPN)) is a non-preemptive scheduling algorithm that gives priority to the job that have the minimum execution time to be processed. It minimises the amount of time that each process has to wait before being processed. If two processes have the same execution time, then the first arriving has higher priority than the second one.
This algorithm has a drawback, in fact if other small jobs continuously arrive at the system, then these immediately overtake the processes with bigger size already present in the queue, blocking in this way the latter for a long time: for example, the IBM 7094 at MIT that used this algorithm was shut down in 1973 and in the system there were still present waiting jobs initialised in 1967 [17]. Some studies have been made in order to prevent that large jobs

have to wait indefinitely before being executed. An example is the Enhanced SJF algorithm proposed in [17] in which the shortest processes are scheduled as in the SJF algorithm but after an interval of time(selected dynamically) also big jobs could be processed. The processes are ordered in the queue, keeping the shorter ones in the head while the longer ones are kept in the back. There are two pointers which are used to take processes from the front or from the rear of the queue.

3.2.2 Shortest Remaining Processing Time (SRPT)

Shortest Remaining Processing Time (SRPT) is the pre-emptive version of the SJF algorithm. Unlike the SJF, in which a job is processed until it runs out, in the SRPT a job can be interrupted if a shorter job enters in the queue. In this way, it minimises the average response time of the system and the queue length distribution [18].
SRPT, like SJF, is not widely used because it requires to know an estimation of the execution time of each job [19].

3.2.3 Multilevel size based scheduling

Queuing systems with a multilevel scheduling structure consist on a series of queues ordered according to their priority. Each queue processes the jobs until they reach a certain level of obtained service and then they are passed to a queue of another level with lower priority. We define a bound for each level in order to have:

$$0 = a_0 < a_1 < ... < a_N < a_{N+1} = \infty$$

For example, a job whose size is x, where $a_i < x < a_{i+1}$ will start its service at the queue with highest priority. Once it has received an amount of work equal to a_1 it is moved to the queue with the second highest priority. This continue until it reaches the queue with the i_th top priority. All the computational effort is concentrated in the queue with the highest priority which is not empty. We can define different scheduling disciplines for each level (it can be FCFS, RR or PS). As previously stated, each level has a different priority than the others in such a way that the schedulers choose to take the jobs that are in the high-level queue (those with the highest priority) so that small jobs are prioritised over large ones. In this way, a job that is in queue i can be served only if the $i-1$ previous queues are empty, otherwise it has to wait.

Chapter 4

Introduction to queuing theory

4.1 Introduction

In this chapter, we give a general introduction to queuing theory, we introduce its notation, we introduce the *Poisson process* that is widely used to describe the arrival process and, finally, we present two different queuing systems that are widely used and that we will exploit in this work.

Queuing theory is used to model and understand the performance of a system with waiting queues. It is widely used in a lot of technological fields [20].

For the purpose of this thesis, we consider a set of identically distributed random variables that forms the arrival process. They represent how customers arrive at the system and how they are distributed in the time. On the other hand, there is a set of independent random variables that models the service time. Let λ^{-1} be the expectation of the inter-arrival time of the jobs and let μ^{-1} be the expected service time. Then, if the queue has an infinite buffer, in order to have a stable system it is necessary that $\mu > \lambda$. In this work,

we take as definition of stability of queuing system the property that states that when time t goes to infinity the expected number of jobs in the system is finite. Therefore, if $\mu > \lambda$, the queue length tends to grow indefinitely because the system is unable to serve the input workload [20].
In general, we can have one or more servers, and we take the system capacity (maximum number of jobs that could be present in the system) to be infinite. All the customers that arrive at the system have to wait in a queue. Although in reality the queues have a limited size, it is common to consider them infinite from a theoretical prospective [21].
The general description of queues is done with the Kendall's notation $A/S/c/K/N/D$ where:

- A denotes the inter-arrival distribution, S indicates the service time distribution and they can be:
  - M: Exponential;
  - H_k: Hyper-exponential with k phases of service;
  - D: Deterministic;
  - G: General;
- c indicates the number of servers;
- K indicates the system capacity (if not specified, we assume that it is ∞);
- N indicates the population size (if not specified, we assume that it is ∞);

- D indicates the service discipline, the most common are FIFO, LIFO and PS (if not specified, we assume that it is *FIFO*);

Typically, arrivals at queues are described by a *Poisson processes*. This is a random process that has some fundamental characteristics such as [12]:

- the inter arrival times between two consecutive jobs are identically and exponentially distributed random variables;
- the inter-arrivals are independent and memoryless:

$$P(T \leq t_0 + t | T > t_0) = P(T \leq t)$$

- the probability that there are n arrivals in a interval of time t is:

$$P(n) = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$$

  where λ is called intensity of the process;

- the probability of one arrival in a small interval of time of width ϵ is proportional to its length:

$$P(1 \text{ arrival in } [t, t+\epsilon]) = \lambda \epsilon$$

4.2 M/M/1 queue

The simplest queuing system is the *M/M/1* in which the arrivals are distributed according to a Poisson process (denoted by the first *M* in the classification) with exponentially distributed interarrival times. There is one server which serves the clients in the queue (denoted with the *1* in the classification) and the job service time has an exponential distribution (denoted by

the second M in the classification) [12]. Thanks to the independence and the exponential distribution of all the times involved in the system, we have that the stochastic process underlying a *M/M/1* queue is a time-homogeneous Markow chain. Its state transmission diagram can be easily represented as shown in Figure 4.1. Jobs arrive with rate λ and are served with rate μ. Each node in the figure represent a state of the system, the number represents how many jobs are present in the system in a specif moment.
For this model, we know that the load factor is $\rho = \lambda/\mu$ and the stationary probability distribution of the number of costumers in the system (that is equal to the stationary probability to find the system in state i of the state space) is $\pi_o = (1-\rho)\rho^i$. We also know that:

- the expected number of jobs in the system is:

$$\bar{N} = \frac{\rho}{1-\rho}$$

  while the variance is:

$$Var(N) = \frac{\rho}{(1-\rho)^2}$$

- the expected response time computed with Little's theorem is:

$$\bar{R} = \frac{1}{\mu - \lambda}$$

Figure 4.1: State space of M/M/1 queue

If we give a maximum size to the buffer we have an *M/M/1/N* queue. All the other assumptions are equal to the *M/M/1* queue, but the probability to be in state i becomes:

$$\pi_o = \frac{1-\rho}{1-\rho^{n+1}}\rho^i$$

and the queue is unconditionally stable.

4.3 M/G/1 queue

Another popular queue is the *M/G/1* in which the arrivals are distributed with a Poisson process with rate λ (denoted with the first M in the classification), there is a general service time distribution with mean $\frac{1}{\mu}$ and variance σ^2 (denoted with the G in the classification), a single server (denoted with the *1* in the classification) and the queuing discipline is *FIFO*.
For this type of queue we know that:

- the load factor is $\rho = \lambda/\mu$;
- the expected number of jobs in the system is given by the Pollaczek Khinchine formula (P-K formula):

$$\bar{N} = \frac{\rho^2 + \lambda^2\sigma^2}{2(1-\rho)} + \rho$$

- the expected response time is:

$$\bar{R} = \frac{\rho + \lambda\mu\sigma^2}{2(\mu-\lambda)} + \frac{1}{\mu}$$

- the expected waiting time is:

$$E[w] = \frac{\rho + \lambda\mu\sigma^2}{2(\mu-\lambda)}$$

It is worth of notice that the stochastic process underlying a *M/G/1* queue is not Markovian. As a consequence, several techniques have been developed to study its stationary behaviour (see [2] and [22]). However, the importance of the *P-K formula* relies on the fact that the expected performance indexes depend only from the first two moments of the service time distributions although it can be shown that the stationary state distribution depends on the whole description of service time.

4.4 Conclusion

In this chapter we introduced some queuing models with some of the most important indices of performance that we will use in the following chapters. in queuing theory, we can focus either on the transient properties of the system, i.e., its behaviour over a finite horizon, or on the stationary behaviour, i.e., we consider the system in the long-run. In this chapter we have focused only on the latter results which one extremely useful for the performance evaluation of networking systems. In fact, transient behaviour is usually important for reliability analysis. An interesting aspect that characterises the stationary behaviour of Markovian queues is that, if the stochastic process underlying the queue is ergodic, then the performance indices do not depend on the initial state of the queue. For the Markovian queues, it can be shown that the ergodicity is granted by the stability condition.

Chapter 5

Analysis of multilevel size based scheduling[2]

5.1 Introduction

This section presents the multilevel size based scheduling technique presented by L. Kleinrock in [1] and then shows its analysis by means of a queuing system with a multilevel scheduling structure. We give a general idea of our model and then we compute some indexes that are useful to understand its power compared to those already existing in the scientific literature.

The model presented by L. Kleinrock in [1] allows us to describe a mixed scheduling strategy that is used in a large set of algorithm. We use Poisson arrivals and an arbitrary service time distribution. There are different levels of obtained service:

$$0 = a_0 < a_1 < ... < a_N < a_{N+1} = \infty \tag{5.1.1}$$

where $a_i \in \mathbb{R}^+ \cup \{\infty\}$ denote thresholds on the size of the job in the system. When a job arrives at the queue it start being served at the highest priority queue according to the scheduling discipline associated with that level. If its size is lower or equal than a_i, at the service completion it leaves the system. Conversely, if its size is greater than a_i, it receives service from the first level up to size a_i, and then it is moved to the second level to receive the residual service according to the discipline associated with the second level. Therefore, it is possible to define $N + 1$ scheduling discipline (one for each interval/level) followed by a job when it has a specific attained service τ:

$$a_{i-1} \leq \tau < a_i \qquad i = 1, 2, ..., N + 1 \tag{5.1.2}$$

The scheduling discipline for each level could be FCFS, FB or RR. Henceforth, we assume that all the levels implement a processor sharing queuing discipline. We make this choice because we aim to use this model for studying TCP-flow scheduling. These flows consist of numerous packets, therefore from the prospective of routing device, the job is divided in many small pieces (the packets) that are enqueued in its buffer. Clearly, among the available queuing disciplines, the PS is that with analytical tractability that resembles this behaviour more accurately.

5.2 Analysis of the queuing system

In this section we present the analytical study of the multilevel queuing system. The main idea consists in conditioning the expected service time on the job size x. Therefore, we want to calculate the value of $T(x)$. To this aim we introduce the quantity $T_i(x)$ that represents the time spent in queue-level

i for a job of size x, i.e.:

$$T_i(x) = E\{\text{Time spent in the } i_{th} \text{ interval } [a_{i-1}, a_i) \text{ for customers with size } x\} \tag{5.2.1}$$

We note that $T_i(x) = T_i(x')$ for all $x' \geq a_i$ while for $a_{k-1} \leq x < a_k$, we have that:

$$T(x) = \sum_{i=1}^{k} T_i(x) \tag{5.2.2}$$

If $\Delta(x)$ is the *p.d.f.* of the job size, then the expected response time not conditioned on the job size can be derived as:

$$T = \int_0^\infty T(x)\Delta(x)dx$$

We want to analyze the behaviour of the jobs that reach the ith-level queue and depart from the system before passing to the *(i + 1)th* level.
We also know that, by the assumption of preemptive priority of the lower level queues, those jobs in level higher than the ith-level can be ignored.
So, if a customer departs during the first level ($0 \leq x < a_1$), then we simply have:

$$T(x) = \frac{x}{1 - \rho_{a_1}} \tag{5.2.3}$$

that is a pure RR system in which the service time distribution is truncated at a_1.

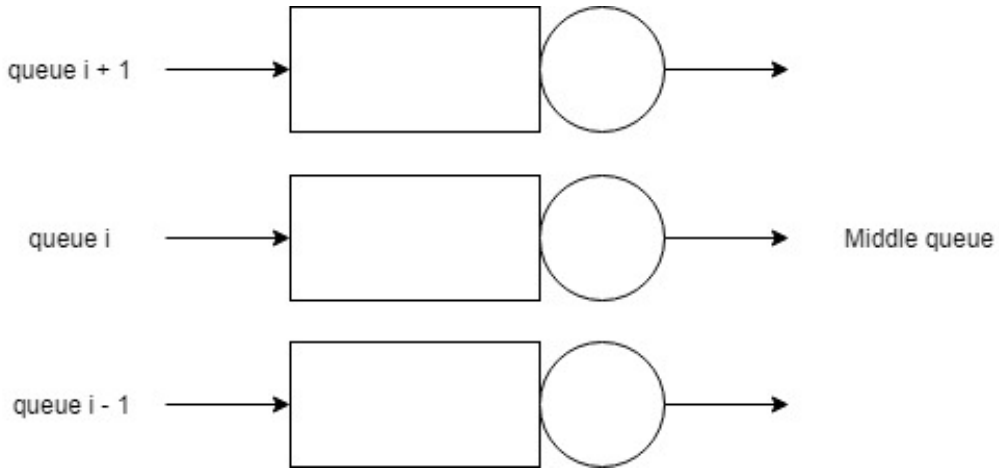

Figure 5.1: Example of multilevel queue, we define *middle queue* the i_{th} level queue that receives jobs from queue of level $i-1$ and passes to level $i+1$

Now, consider a job that requires $x = a_{i-1} + \tau$ sec of service, let α_1 be the mean *real* time the job spends in the system until its arrival to the middle level queue and $\alpha_2(\tau)$ be the mean *virtual* time that the job spends in the middle level queue.

α_1 is equal to the mean work the job finds in the lower level on arrival plus the a_{i-1} sec of work performed in the lower level, this delay is multiplied by a factor of $1/(1-\rho_{a_1})$ due to new jobs arriving at the lower level:

$$\alpha_1 = \frac{1}{1-\rho_{a_1}}(W_{a_{i-1}} + a_{i-1}) \tag{5.2.4}$$

where $W_{a_{i-1}}$ is the mean remaining work in the previous level.

If a customer departs from the second level, the conditioned response time is much harder to compute. The formula for the conditioned service time of the second level is:

$$T(\alpha_1 + \tau) = \frac{1}{1 - \rho_{a_1}}[W_{a_{i-1}} + a_{i-1} + \alpha_2(\tau)] \tag{5.2.5}$$

The only unknown term in the previous equation is $\alpha_2(\tau)$. In order to compute it, we have to consider a M/G/1 system with *bulk* arrivals and RR processor-sharing.

First, we have to compute the average bulk size $\bar{a}$ at the second level using the z-transform of the bulk size:

$$G(z) = \sum_{i=0}^{\infty} P[\bar{a} = n] z^n \tag{5.2.6}$$

The z-transform is defined from the equation [2]:

$$G(z) = [1 - B(a_1)] z e^{-\lambda a_1 [1 - G(z)]} + \int_0^{a_1} e^{-\lambda t [1 - G(z)]} dB(t) \tag{5.2.7}$$

The service time distribution $B(\tau)$ is defined as follows:

$$1 - B(\tau) = \begin{cases} q(\tau) e^{-\mu\tau}, & \text{if } 0 \leq \tau < a_i - a_{i-1} \triangleq t_1 \\ 0, & \text{if } t_1 \leq \tau \end{cases} \tag{5.2.8}$$

We have that $\bar{a} = G^{(1)}(1)$ and the value of b could be expressed in terms of $G(z)$ and we obtain:

$$b = \frac{G^{(2)}(z)}{G^{(1)}(z)} \tag{5.2.9}$$

Now we can obtain the value of $\alpha_2^{(1)}(\tau)$ from the following equation:

$$\alpha_2^{(1)}(\tau) = \frac{1}{1-\lambda\bar{a}\bar{\tau}} - \frac{b}{\lambda\bar{a}} \sum_{i=1}^{d+1} \frac{(\mu^2 - \beta_i^2)}{Q_2^{(1)}(\beta_i)} \Big[\frac{Q_0(\beta_i)e^{\beta_i\tau} - Q_1(\beta_i)e^{\beta_i(t_1-\tau)}}{Q_0(\beta_i) + Q_1(\beta_i)e^{-\beta_i t_1}} \Big] \tag{5.2.10}$$

where:

$$Q_0(y) = (y+\mu)^{d+1} - \lambda\bar{a} \sum_{i=0}^{d} q^{(i)}(0)(y+\mu)^{d-i} \tag{5.2.11}$$

$$Q_1(y) = \lambda\bar{a} \sum_{i=0}^{d} e^{-\mu t} q^{(i)}(t_1)(y+\lambda)^{d-1} \tag{5.2.12}$$

$$Q_2(y) = Q_0(y)Q_0(-y) - Q_1(y)Q_1(-y) \tag{5.2.13}$$

and where the solutions of the equations $Q_2(y) = 0$ occur in pairs denoted by $(\beta_i, -\beta_i)$ for $i = 1, 2, ..., d+1$.[2]

In the next figure we report the Response time of an *M/M/1* queue with parameters $\bar{x} = 1$, $\lambda = 0.75$, $a_1 = 2$. The image is taken from [2].

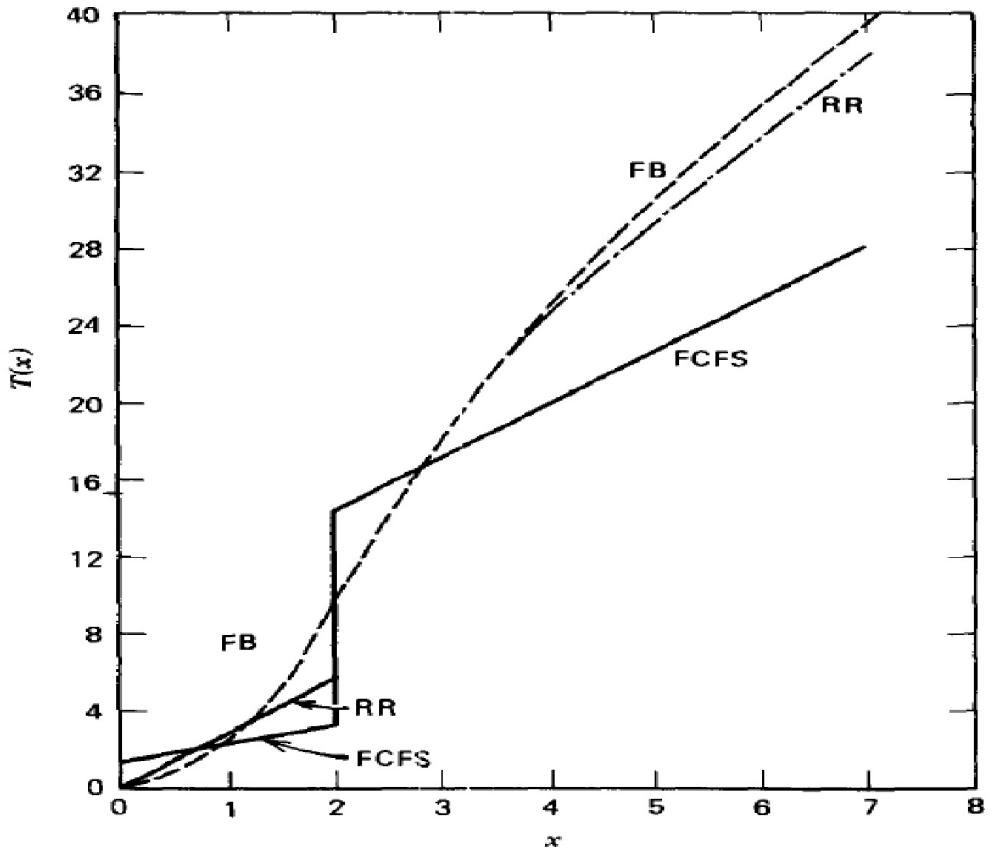

Figure 5.2: **Response time for M/M/1,** $\bar{x} = 1$, $\lambda = 0.75$, $a_1 = 2$

5.3 The 2-level processor sharing queue

This model was designed after studying the models presented in the articles [3, 4, 6, 8, 23] and is based on the algorithm developed by L. Kleinrock for the multilevel processor-sharing scheduler (chapter 4.7 of [2]). Since we focus our attention on data center TCP flows, real-world devices are capable of handling two priority queues. The complexity of the device with more priorities seems not to justify the benefits obtained in terms of performance improvements.

As reported in Figure 5.3 the model is composed of two queues, one with *high priority* and one with *low priority*. When a job arrives in the system it is inserted in the first queue with high priority, both the queues are served with a round robin policy but the second one is served only if the first one is empty. Each job can be processed in the first queue until it reaches a specific threshold of attained service a, if the job is smaller than a it is fully served in the first queue (it is a *short job*) otherwise it is considered as a *big job* and is moved to the second queue.

Only when the first queue is empty, a job that is in the second queue can be processed. There, it is processed until a new job arrives in the high priority queue, or the job ends or ends its quantum of time.

The jobs arrive at the system with rate λ and they are served with rate μ. All the derivations are done with the assumptions that the system works with Poisson arrivals and arbitrary service time distribution.

In Figure 5.3 the functionality of the model is presented schematically.

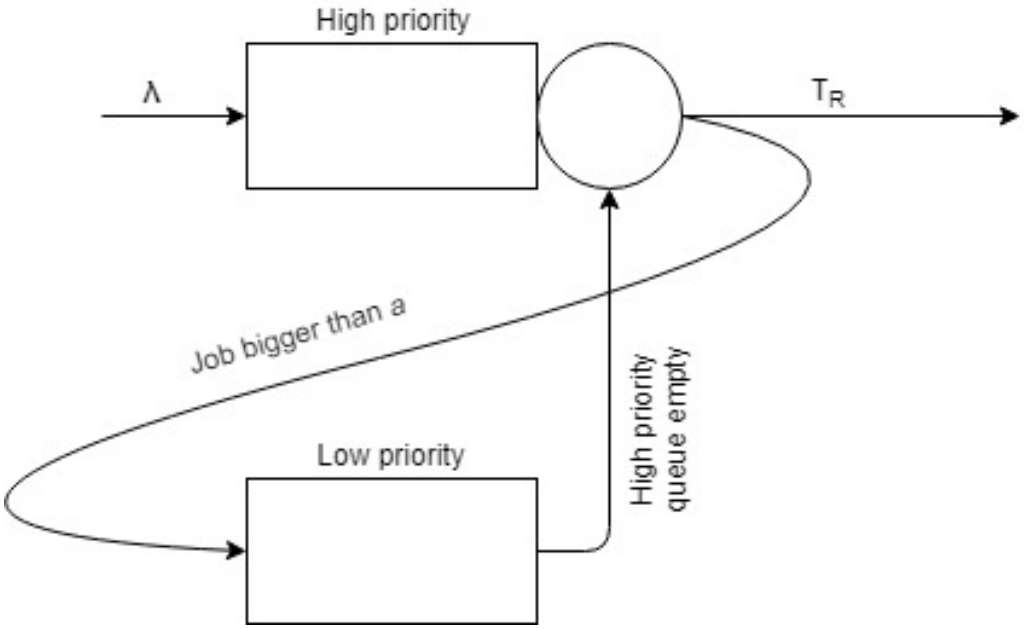

Figure 5.3: General idea of the model, we described it in the previous section. When a job arrives in the system it is inserted in the first queue. Each job can be processed in the first queue until it reaches a specific threshold of attained service a, if it is not completed it is moved to the second queue. Only when the first queue is empty, a job that is in the second queue can be processed.

5.3.1 High priority queue

In this section we focus on the analysis of the response time conditioned on the job size at the high priority queue. We compute the utilisation factor of the queue in the first level by knowing that $S(x) = Pr\{X \leq x\}$ is the cumulative distribution function of X, the random variable that describes the size of the job. The average size $\bar{S}_1$ of a job served at this queue is:

$$\bar{S}_1 = \int_0^a x s(x) dx + a \int_a^\infty s(x) dx \tag{5.3.1}$$

where $s(x)$ is the probability density function of the variable X, and a is the only threshold in our system used to distinguish the job sizes.
Therefore, the load factor ρ_1 at the high priority queue is:

$$\rho_1 = \lambda \bar{S}_1 \tag{5.3.2}$$

The average speed and the variance of the distribution are defined as::

$$\mu_1 = \frac{1}{\bar{S}_1} \tag{5.3.3}$$

$$Var(S_1) = \int_0^a x^2 s(x) dx + a^2 \int_a^\infty s(x) dx - \bar{S}_1^2 \tag{5.3.4}$$

The mean work that a random job sees at the first queue at the instant of its arrival is computed with the *PK-formula* (see Chapter 4) as the mean waiting time of a M/G/1 queue:

$$W_1 = \frac{\rho_1 + \lambda \mu_1 Var(S_1)}{2(\mu_1 - \lambda)} \tag{5.3.5}$$

The response time conditioned on the dimension of the job at the first queue is defined in the equation (4.36) of [2] for the jobs that have size $x \leq a$:

$$T(x) = \frac{x}{1 - \rho_1} \tag{5.3.6}$$

Now we compute the mean response time for the jobs that are smaller than the threshold a:

$$T_{\leq a} = \frac{\int_0^a T(x)s(x)dx}{Pr\{X \leq a\}} \tag{5.3.7}$$

The time that a job, with size greater than a, spends in the first queue it is defined in equation (4.37) of [2]:

$$\alpha_1 = \frac{1}{1 - \rho_1}(W_1 + a) \tag{5.3.8}$$

5.3.2 Low priority queue

Now, we discuss how to derive the performance of the low priority queue. Since the low priority queue works only if the high priority queue is empty, we can imagine a time line in which we remove the busy periods of the first queue. In this way the low priority queue sees a Poisson arrival process of batches consisting of the jobs moved from the first queue to the second. If

x is the job size, the remaining work on it is $x - a$. However its advantages relies on the fact that we do not have to solve the differential equation system reported in 5.2.11 and 5.2.12. The mean service time of the second level is computed as:

$$\bar{S}_2 = \int_a^\infty (x-a)s(x)dx \tag{5.3.9}$$

And the mean service rate is:

$$\mu_2^{-1} = \frac{\bar{S}_2}{Pr\{X > a\}}$$

We compute the *mean batch size* $\bar{a}$ and the average size of the bulk from which a tagged job is selected at random from the arrivals to the second level $b+1$ as described in equation (4.41) of [2].

The *generating function* of *bulk size* $\bar{a}$ is defined in the exercise 7.1 of [1] as the solution of:

$$\beta(z) = [1 - B(a)]ze^{-\lambda a_{i-1}[1-\beta(z)]} + \int_0^a e^{-\lambda t[1-\beta(z)]}dB(t) \tag{5.3.10}$$

where $B(a)$ is the service time distribution for the level.
From equation(5.3.10) we can compute $\bar{a}$ and b:

$$\bar{a} = \beta^{(1)}(1) \tag{5.3.11}$$

$$b = \frac{\beta^{(2)}(1)}{\beta^{(1)}(1)} \tag{5.3.12}$$

Now, to compute the expected response time of the second queue we restart to the *z-transform* approach, i.e., we introduce the function $g(z) = \sum_{i=0}^{\infty} \pi_i z^i$ where π_i is the stationary probability that queue 2 has i jobs. Notice that this approach provides the exact solution in case of exponentially sized jobs but it is approximate otherwise. To compute π_0 and *g(z)* we need to start from the balance equation of the system where b_i is the probability that the batch size is equal to i.

$$\begin{cases} \pi_0 \lambda(1 - b_0) = \pi_1 \mu_2 \\ \pi_n(\lambda(1 - b_0) + \mu_2) = \pi_{n+1}\mu_2 + \sum_{i=1}^{n} b_i \pi_{n-i} \end{cases} \tag{5.3.13}$$

We can multiply both members for z^n and sum up all the equations of the system (5.3.13)

$$\sum_{n=1}^{\infty}(\lambda(1 - b_0) + \mu_2)z^n\pi_n = \sum_{n=1}^{\infty} \mu\pi_{n+1}z^n + \lambda \sum_{n=1}^{\infty} z^n \sum_{i=1}^{\infty} b_i \pi_{n-i}$$

$$(\lambda(1 - b_0) + \mu_2) \sum_{n=1}^{\infty} z^n\pi_n = \frac{\mu_2}{z} \sum_{n=1}^{\infty} \pi_{n+1}z^{n+1} + \lambda \sum_{n=1}^{\infty} z^n \sum_{i=1}^{\infty} b_i \pi_{n-i}$$

We know that $g(z) = \sum_{n=0}^{\infty} \pi_n z^n$ so:

$$(\lambda(1 - b_0) + \mu_2)\big[g(z) - \pi_0\big] = \frac{\mu_2}{z}\big[g(z) - \pi_0 - \pi_1 z\big] + \lambda \sum_{n=1}^{\infty} z^n \sum_{i=1}^{\infty} b_i \pi_{n-i}$$

From equation(5.3.13) we know that $\pi_1 = \pi_0 \frac{\lambda(1-b_0)}{\mu}$

$$(\lambda(1-b_0)+\mu_2)\big[g(z)-\pi_0\big] = \frac{\mu_2}{z}\Big[g(z)-\pi_0-\pi_0\frac{\lambda(1-b_0)}{\mu_2}z\Big]+\lambda\sum_{n=1}^{\infty}z^n\sum_{i=1}^{\infty}b_i\pi_{n-i}$$

The last term can be rewritten as:

$$\begin{array}{llllll} n=1 & zb_1\pi_0 & & & & \\ n=2 & z^2b_1\pi_1 & + & z^2b_2\pi_0 & & \\ n=3 & z^3b_1\pi_2 & + & z^3b_2\pi_1 & + & z^3b_3\pi_0 \\ \cdots & & & & & \end{array}$$

this succession can be rewritten as:

$$\sum_{j=0}^{\infty}\pi_j\sum_{i=1}^{\infty}b_iz^{i+j} = \sum_{j=0}^{\infty}\pi_jz^j\sum_{i=1}^{\infty}b_iz^i = g(z)\big[\beta(z)-b_0\big]$$

$$(\lambda(1-b_0)+\mu_2)\big[g(z)-\pi_0\big] = \frac{\mu_2}{z}\Big[g(z)-\pi_0-\pi_0\frac{\lambda(1-b_0)}{\mu_2}z\Big]+\lambda g(z)\big[\beta(z)-b_0\big] \tag{5.3.14}$$

From equation(5.3.14) we can get π_0:

$$\pi_0 = \frac{\mu_2 g(z)-\lambda z g(z)-\mu_2 z g(z)+\lambda z g(z)\beta(z)}{\mu_2(1-z)} \tag{5.3.15}$$

We compute the limit of equation(5.3.15) with $z \to 1$, we recall that $g(1) =$

$\beta(1) = 1$. We know that $g'(1) = \bar{N}$ is the average number of jobs in the system and $\beta'(1) = \bar{a}$ is the average batch size.

$$\pi_0 = \lim_{z\to 1} \frac{\mu_2 g(z) - \lambda z g(z) - \mu_2 z g(z) + \lambda z g(z)\beta(z)}{\mu_2(1-z)}$$

$$\pi_0 \overset{\text{H}}{=} \lim_{z\to 1} \frac{g(z)\big[-\lambda - \mu_2 + \lambda\beta(z) + \lambda z\beta'(z)\big] + g'(z)\big[\mu_2 - \lambda z - \mu_2 z + \lambda z\beta(z)\big]}{-\mu_2}$$

And so we have that:

$$\pi_0 = 1 - \frac{\lambda\bar{a}}{\mu_2} \tag{5.3.16}$$

From equation(5.3.14) we obtain $g(z)$ as:

$$g(z) = \frac{\pi_0\mu_2 - \pi_0\mu_2 z}{\mu_2 + \lambda z\beta(z) - \lambda z - \mu_2 z} \tag{5.3.17}$$

It is easy to see that $g(1) = 1$, now deriving $g(z)$ from equation(5.3.17) we get $g'(z)$

$$g'(z) = -\frac{\lambda\mu_2\pi_0(-1 + \beta(z) - (-1+z)z\beta'(z))}{(\mu_2(-1+z) + \lambda z - \lambda z\beta(z))^2} \tag{5.3.18}$$

$$g'(1) = \lim_{z\to 1} -\frac{\lambda\mu_2\pi_0(-1 + \beta(z) - (-1+z)z\beta'(z))}{(\mu_2(-1+z) + \lambda z - \lambda z\beta(z))^2}$$

$$g'(1) \overset{\text{H}}{=} \lim_{z\to 1} \frac{\lambda\mu_2\pi_0(-1+z)(2\beta'(z) + z\beta''(z))}{2(\mu_2(-1+z) + \lambda z - \lambda z\beta(z))(\lambda + \mu_2 - \lambda(\beta(z) + z\beta'(z)))}$$

$$g'(1) \overset{\text{H}}{=} \lim_{z\to 1} \left[\frac{\lambda\mu_2\pi_0(2\beta'(z) + (-3+4z) + \beta''(z) + (-1+z)z\beta^{(3)}(z))}{2\big\{\big[\lambda+\mu_2-\lambda(\beta(z)+z\beta'(z))\big]^2 + \\ \quad + \lambda\big[\mu_2-(\lambda+\mu_2)z+\lambda z\beta(z)\big]\big[2\beta'(z)+z\beta''(z)\big]\big\}} \right]$$

As seen in chapter 4.7 of [2] $\bar{a} = \beta'(1)$ and $b = \frac{\beta''(1)}{\beta'(1)}$

$$g'(1) = \frac{\lambda\mu_2\pi_0(2\bar{a} + \beta''(1))}{2\big[\lambda+\mu_2-\lambda(1+\bar{a})\big]^2}$$

We can rewrite $\beta''(1) = b\bar{a}$ and remembering that $\pi_0 = \frac{\mu_2-\lambda\bar{a}}{\mu_2}$ as showed in equation(5.3.16) we obtain:

$$\bar{N} = g'(1) = \frac{\lambda\bar{a}(2+b)}{2(\mu_2-\lambda\bar{a})}$$

Now we can compute the *mean response time* $\bar{R}$ for the second level multiplying $\bar{N}$ by throughput:

$$\alpha_2 = \bar{R} = \frac{\lambda\bar{a}(2+b)}{2(\mu_2-\lambda\bar{a})}\frac{1}{\lambda\bar{a}} = \frac{2+b}{2(\mu_2-\lambda\bar{a})} \tag{5.3.19}$$

Let's check if we have an *M/M/1* queue in which $\bar{a} = 1$ and $b = 0$ we have that:

$$\bar{N} = \frac{\lambda}{\mu_2-\lambda} = \frac{\lambda/\mu_2}{1-\lambda/\mu_2} = \frac{\rho}{1-\rho}$$

$$\bar{R} = \frac{1}{\mu_2 - \lambda}$$

We compute the mean response time for the system:

$$\bar{T}_R = T_{\leq a} Pr\{X \leq a\} + \alpha_1 Pr\{X > a\} + \frac{1}{1-\rho_1}\alpha_2 Pr\{X > a\} \qquad (5.3.20)$$

5.4 The model with exponentially sized job

In this section we will show our model applied to systems with exponential arrival and service time distributions.

5.4.1 High priority queue

We compute the utilization factor of the first level starting from the equation (5.3.1) and by knowing that $s(x) = \mu e^{-\mu x}$:

$$\bar{S_1} = \int_0^a x s(x) dx + \int_a^\infty a s(x) dx = \int_0^a x \mu e^{-\mu x} dx + a \int_a^\infty \mu e^{-\mu x} dx$$

$$\bar{S_1} = \frac{1 - e^{-a\mu}}{\mu}$$

The load factor ρ_1:

$$\rho_1 = \lambda \bar{S_1} = \frac{\lambda(1 - e^{-a\mu})}{\mu}$$

The mean and the variance of the distribution:

$$\mu_1 = \frac{1}{\bar{S_1}} = \frac{\mu}{\lambda(1 - e^{-a\mu)}}$$

$$Var(\bar{S_1}) = \int_0^a x^2 \mu e^{-\mu x} dx + a^2 \int_a^\infty \mu e^{-\mu x} dx - \bar{S_1}^2$$

The mean response time of the first level requires the computation of the *PK-formula* of a *M/G/1* queue described in equation (5.3.5)

$$W_1 = \frac{\rho_1 + \lambda\mu_1 Var(S_1)}{2(\mu_1 - \lambda)} = \frac{\lambda e^{-a\mu}\left(-a\mu + e^{a\mu} - 1\right)}{\mu^2(1-\rho_1)}$$

The mean response time of the jobs that has size lower than the threshold a is computed as follow, knowing that $T(x)$ is calculated as reported in the equation (5.3.6):

$$T_{\leq a} = \frac{1}{1 - e^{-a\mu}} \int_0^a T(x)\mu e^{-\mu x} dx = \frac{1}{1 - e^{-a\mu}} \left[\frac{-a\mu + e^{a\mu} - 1}{e^{a\mu}(\mu - \lambda) + \lambda} \right] \tag{5.4.1}$$

Remembering that, as seen in equation (5.3.8), the time spent in the first queue from a job that is bigger than the threshold is:[i]

$$\alpha_1 = \frac{1}{1 - \rho_1}(W_1 + a) \tag{5.4.2}$$

5.4.2 Low priority queue

Now we want to evaluate the *average batch size* $\bar{a}$ and b using equation (5.3.10) knowing that $\mu_2 = \mu$ for exponential distribution

$$1 - B(t) = e^{-\mu t} \Longrightarrow dB(t) = \mu e^{-\mu t} dt$$

[i] the development of the following expression is not reported due to the complexity of the result

Equation(5.3.10) becomes:

$$\beta(z) = [1 - B(a_{i-1})]ze^{-\lambda a_{i-1}[1-\beta(z)]} + \int_0^{a_{i-1}} e^{-\lambda t[1-\beta(z)]}\mu e^{-\mu t}dt$$

$$\beta(z) = ze^{-a\lambda(1-\beta(z))-a\mu} - \frac{\mu\left(e^{a(\lambda\beta(z)-\lambda-\mu)} - 1\right)}{\lambda - \lambda\beta(z) + \mu}$$

Knowing that $\bar{a} = \beta'(1)$ as defined in equation(5.3.11) we have:

$$\bar{a} = \beta'(1) = \frac{\mu}{\lambda + e^{a\mu}(-\lambda + \mu)}$$

We also know that $b = \frac{\beta^{(2)}(1)}{\beta'(1)}$ as defined in equation(5.3.12)

$$b = \frac{\beta^{(2)}(1)}{\beta'(1)} = \frac{2\lambda\big[-\lambda + e^{a\mu}(\lambda - a\lambda\mu + a\mu^2)\big]}{\big[\lambda + e^{a\mu}(-\lambda + \mu)\big]^2}$$

Knowing $\bar{a}$ and b we can compute the *mean response time* of the second queue as seen in equation (5.3.19): [ii]

$$\alpha_2 = \bar{R} = \frac{2 + b}{2(\mu_2 - \lambda\bar{a})} \tag{5.4.3}$$

[ii] the development of the following expression is not reported due to the complexity of the result

Now we can compute the mean response time of the system starting from equation(5.3.20) and putting together (5.4.1), (5.4.2) and (5.4.3):

$$\bar{T_R} = T_{\leq a} Pr\{X \leq a\} + \alpha_1 Pr\{X > a\} + \frac{1}{1-\rho_1}\alpha_2 Pr\{X > a\} = \frac{1}{\mu - \lambda} \tag{5.4.4}$$

So the response time of our module with exponentially sized job depends only on the rate of arrival λ and on the rate of service of the system μ. The response time is constant and there are no improvements or worsening due to the change in the size of the threshold. However, we still have that we can observe an improvement in the expected response time of the small jobs. Albeit the worsening of the large jobs response time compensate these benefits. We can explain this result with the memoryless property of the exponential distribution: in fact, when we move a job from the high priority queue to the low priority one, we cannot predict if the residual size is big (and hence the use of two levels brings benefit) or is small (and hence the multi-level worsen the response time too much).

5.5 The model with hyper-exponentially sized job

In this section we will show our model applied to systems with hyper-exponential service time distribution (two exponential distribution with parameter η_1 and η_2) and exponential inter arrival distribution. Flows can be generated by the first exponential distribution with probability p or by the second distribution with probability $1 - p$.

A hyper-exponential distribution is a continuous probability distribution that mix a set of exponential distributions. Let $X_1, X_2, ..., X_n$ be independent exponential distribution with rate $\eta_1, \eta_2, ..., \eta_n$ and the probability of choice each variables are p_1, p_2, p_n. We can define the probability distribution functions as:

$$f(x) = p_1(\eta_1 e^{-\eta_1 x}) + p_2(\eta_2 e^{-\eta_2 x}) + ... + p_n(\eta_n e^{-\eta_n x})$$

The analysis that we propose is approximate since we adopt the generating function approach described in Chapter 5. We will access the accuracy of this approximation in the chapter presenting the simulation estimates.

5.5.1 High priority queue

We compute the utilization factor of the first level starting from the equation (5.3.1) and by knowing that $s(x) = p\eta_1 e^{-\eta_1 x} + (1-p)\eta_2 e^{-\eta_2 x}$:

$$\bar{S}_1 = \int_0^a x s(x) dx + \int_a^\infty a s(x) dx$$

$$\bar{S}_1 = \frac{\eta_1(p-1)\left(e^{-a\eta_2}-1\right)+\eta_2 p\left(1-e^{-a\eta_1}\right)}{\eta_1\eta_2}$$

The load factor ρ_1:

$$\rho_1 = \lambda\bar{S}_1 = \frac{\lambda\left(\eta_1(p-1)\left(e^{-a\eta_2}-1\right)+\eta_2 p\left(1-e^{-a\eta_1}\right)\right)}{\eta_1\eta_2}$$

The mean and the variance of the distribution:

$$\mu_1 = \frac{1}{\bar{S}_1} = \frac{\eta_1\eta_2}{\eta_1(p-1)\left(e^{-a\eta_2}-1\right)+\eta_2 p\left(1-e^{-a\eta_1}\right)}$$

$$Var(\bar{S}_1) = \int_0^a x^2 s(x)dx + a^2\int_a^\infty s(x)dx - \bar{S}_1^2$$

The mean response time of the first level is computed with the *PK-formula* of a *M/G/1* queue described in equation(5.3.5)

$$W_1 = \frac{\rho_1 + \lambda\mu_1 Var(S_1)}{2(\mu_1-\lambda)}$$

The mean response time of the jobs that has size lower than the threshold a is computed as follow, knowing that $T(x)$ is calculated as reported in the equation (5.3.6):

$$T_{\leq a} = \frac{1}{1-e^{-a\mu}}\int_0^a T(x)s(x)dx \tag{5.5.1}$$

Remembering that, as seen in equation (5.3.8), the time spent in the first queue from a job that is bigger than the threshold is:[iii]

$$\alpha_1 = \frac{1}{1-\rho_1}(W_1 + a) \tag{5.5.2}$$

5.5.2 Low priority queue

Now we want to evaluate the *average batch size* $\bar{a}$ and b using equation(5.3.10) knowing that μ_2 for hyper-exponential distribution is:

$$\mu_2 = \frac{\eta_1\eta_2\left(pe^{a\eta_2} - (p-1)e^{a\eta_1}\right)}{\eta_2 pe^{a\eta_2} - \eta_1(p-1)e^{a\eta_1}}$$

$$B(t) = p(1 - e^{-\eta_1 t}) + (1-p)(1 - e^{-\eta_2 t})$$

So $dB(t)$ becomes:

$$dB(t) = \eta_1 pe^{-\eta_1 t} + \eta_2(1-p)e^{-\eta_2 t}dt$$

Equation(5.3.10) becomes:

$$\beta(z) = [1 - B(a_{i-1})]ze^{-\lambda a_{i-1}[1-\beta(z)]} + \int_0^{a_{i-1}} e^{-\lambda t[1-\beta(z)]}dB(t)$$

[iii] the development of the following expression is not reported due to the complexity of the result

Knowing that $\bar{a} = \beta'(1)$ as defined in equation(5.3.11) we have:

$$\bar{a} = \frac{\eta_1\eta_2\left(pe^{a\eta_2} - (p-1)e^{a\eta_1}\right)}{e^{a(\eta_1+\eta_2)}(\eta_1(\eta_2+\lambda(p-1)) - \eta_2\lambda p) + \eta_1\lambda(p-1)\left(-e^{a\eta_1}\right) + \eta_2\lambda p e^{a\eta_2}}$$

We also know that $b = \frac{\beta^{(2)}(1)}{\beta'(1)}$[iv] as defined in equation(5.3.12).
Knowing $\bar{a}$ and b we can compute the *mean response time* of the second queue as seen in equation (5.3.19): [iv]

$$\alpha_2 = \bar{R} = \frac{2+b}{2(\mu_2 - \lambda\bar{a})} \tag{5.5.3}$$

Now we can compute the mean response time of the system starting from equation(5.3.20) and putting together (5.5.1), (5.5.2) and (5.5.3):

$$\bar{T_R} = T_{\leq a}Pr\{X \leq a\} + \alpha_1 Pr\{X > a\} + \frac{1}{1-\rho_1}\alpha_2 Pr\{X > a\}$$

$$\bar{T_R} = \frac{\eta_1(1-p) + \eta_2 p - \frac{\lambda(p-1)p(\eta_1-\eta_2)(\eta_1(e^{a\eta_2}-1)-\eta_2 e^{a\eta_1}+\eta_2)}{e^{a(\eta_1+\eta_2)}(\eta_1(\eta_2+\lambda(p-1))-\eta_2\lambda p)+\eta_1\lambda(p-1)(-e^{a\eta_1})+\eta_2\lambda p e^{a\eta_2}}}{\eta_1(\eta_2+\lambda(p-1)) - \eta_2\lambda p} \tag{5.5.4}$$

Unlike the average response time calculated with jobs originating from an exponential distribution (equation 5.4.4) in this case we have many variables

[iv] the development of the following expression is not reported due to the complexity of the result

within the final result. The average response time of the system applied to jobs with hyper-exponential size distribution depends on the means of the two distribution η_1 and η_2 , from the threshold a, from the arrival rate λ and from the probability p that a flow is generated by the first or by the second distribution. Notice that the parameter a is now present in the expected response time and hence the formula can be used to choose the best threshold in order to minimize the expected overall response time.

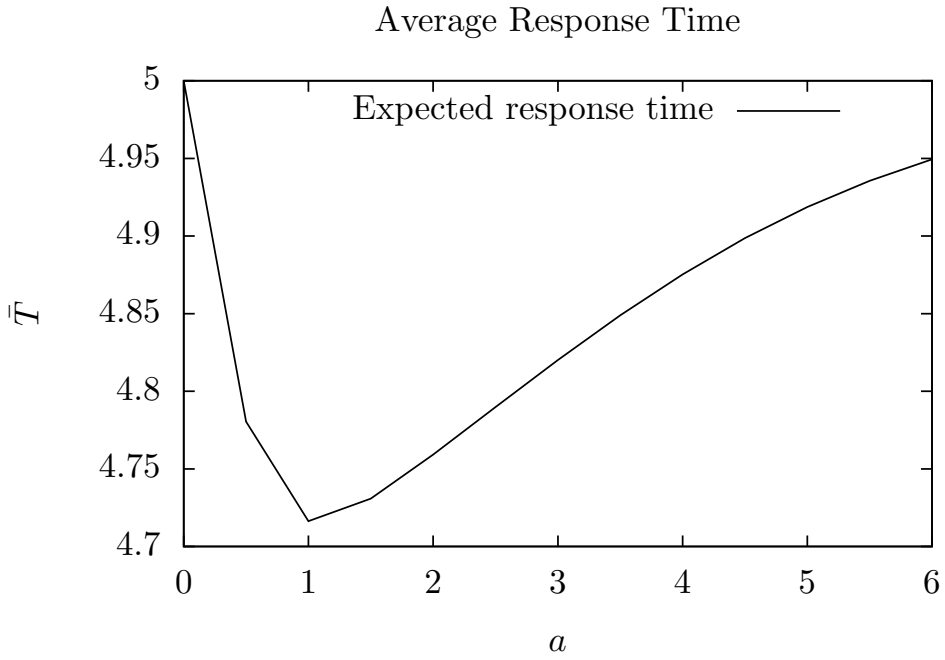

Figure 5.4: Average Response Time computed for hyper-exponentially sized job (the simulation was done with two exponential distribution has rate respectively equal to 1/0.2 and 1/1.8 and the probability to choice one distribution or the other is equal to 0.5). As describe in the next section we can notice an improvement of the average response time when the size of the threshold changes

As can be seen in Figure 5.4 in this case, based on the choice of the threshold size, we could have an improvement of the average response time. It is important to select a correct dimension of the threshold a to obtain the best possible improvement however, even with a non-optimal selection of the threshold, an improvement is always obtained even if minimal. The figure is obtained with two exponential distribution with rate respectively equal to 1/0.2 and 1/1.8 and the probability to choice one distribution or

the other is equal to 0.5, we can notice that. In this case the optimal value of the threshold is around 1. Moreover, we observe that, for the parameters considered in this example, the expected response time of a *M/G/1/PS* queue is 5 and hence the introduction of the multilevel queue produce a reduction of the expected response time of 6% plus the benefits for the small jobs. If the phases of the hyper-exponential distribution are very different in magnitude, the improvement can reach also higher level around 20%.

5.6 The model with uniformly sized job

In this section we will show our model applied to systems with uniform arrivals distribution $U(v, w)$ and exponential service distribution. Without loss of generality we consider that our threshold a is in between v and w and $v \geq 0$

5.6.1 High priority queue

As previously seen for the exponential model we calculate the utilization factor of the first queue by taking the equation(5.3.1), knowing that $s(x) = \frac{1}{w-v} \quad \forall x \in [v, w]$:

$$\bar{S_1} = \int_0^a x s(x) dx + \int_a^\infty a s(x) dx = \frac{1}{w-v} \int_v^a x dx + \frac{a}{w-v} \int_a^w dx$$

$$\bar{S_1} = \frac{a^2 + v^2 - 2aw}{2(v-w)}$$

The load factor ρ_1:

$$\rho_1 = \lambda \bar{S_1} = \frac{\lambda(a^2 + v^2 - 2aw)}{2(v-w)}$$

The mean and the variance of the distribution:

$$\mu_1 = \frac{1}{\bar{S_1}} = \frac{2(v-w)}{a^2 + v^2 - 2aw}$$

$$Var(\bar{S}_1) = \frac{1}{w-v}\int_v^a x^2 dx + \frac{a^2}{w-v}\int_a^w dx - \bar{S}_1^2 = -\frac{(a-v)^3(3a+v-4w)}{12(v-w)^2}$$

The mean response time of the first level is computed with the *PK-formula* of a *M/G/1* queue described in equation(5.3.5)

$$W_1 = \frac{\rho_1 + \lambda\mu_1 Var(S_1)}{2(\mu_1 - \lambda)} = -\frac{l\,(2a^3 - 3a^2w + v^3)}{3\,(l\,(a^2 - 2aw + v^2) - 2v + 2w)}$$

The mean response time of the jobs that has size lower than the threshold a is computed as follow, knowing that $T(x)$ is calculated as reported in the equation (5.3.6) :

$$T_{\leq a} = \frac{\frac{1}{w-v}\int_v^a T(x)dx}{Prob\{X < a\}} = \frac{(a-v)(a+v)}{l\,(a^2 - 2aw + v^2) - 2v + 2w} * \frac{1}{Prob\{X < a\}} \tag{5.6.1}$$

Remembering that, as seen in equation (5.3.8), the time spent in the first queue from a job that is bigger than the threshold is:[v]

$$\alpha_1 = \frac{1}{1-\rho_1}(W_1 + a) \tag{5.6.2}$$

[v]the development of the following expression is not reported due to the complexity of the result

5.6.2 Low priority queue

Now we want to evaluate the *average batch size* $\bar{a}$ and b using equation(5.3.10) knowing that $\mu_2 = \frac{2}{w-a}$ for uniform distribution:

$$1 - B(t) \Longrightarrow dB(t) = \frac{1}{w-v}dt$$

Equation(5.3.10) becomes:

$$\beta(z) = \frac{e^{a\lambda(b(z)-1)} - e^{\lambda v(b(z)-1)}}{\lambda(b(z)-1)(w-v)} + z\left(1 - \frac{a-v}{w-v}\right)e^{-a\lambda(1-b(z))}$$

Knowing that $\bar{a} = \beta'(1)$ as defined in equation(5.3.11) we have:

$$\bar{a} = \beta'(1) = \frac{2(a-w)}{a\lambda(a-2w) + \lambda v^2 - 2v + 2w}$$

We also know that $b = \frac{\beta^{(2)}(1)}{\beta'(1)}$ as defined in equation(5.3.12)

$$b = \frac{\beta^{(2)}(1)}{\beta'(1)} = -\frac{4\lambda(a-w)\left(a^3\lambda - 3a^2\lambda w + 3av(\lambda v - 2) + 6aw - \lambda v^3\right)}{3\left(\lambda\left(a^2 - 2aw + v^2\right) - 2v + 2w\right)^2}$$

Knowing $\bar{a}$ and b we can compute the *mean response time* of the second queue as seen in equation (5.3.19):[vi]

$$\alpha_2 = \bar{R} = \frac{2+b}{2(\mu_2 - \lambda\bar{a})} \tag{5.6.3}$$

Now we can compute the mean response time of the system starting from equation(5.3.20) and putting together (5.6.1), (5.6.2) and (5.6.3):

$$\bar{T}_R = T_{\leq a} Pr\{X \leq a\} + \alpha_1 Pr\{X > a\} + \frac{1}{1-\rho_1}\alpha_2 Pr\{X > a\}$$

$$\bar{T}_R = \left[\frac{\begin{array}{c}\lambda\left(a^4 + 2av^3 - 3v^4\right) - 2w\left(2a^3\lambda + v^2(\lambda v + 3)\right) + \\ + 6w^2\left(a^2\lambda + v(\lambda v - 1)\right) - 6w^3(a\lambda - 1) + 6v^3\end{array}}{3(v-w)(\lambda(v+w)-2)\left(\lambda\left(a^2 - 2aw + v^2\right) - 2v + 2w\right)} \right] \tag{5.6.4}$$

Also in this case (as in the case of jobs generated by a hyper-exponential described in the Section 5.5) the average response time depends on many parameters such as the arguments of the uniform distribution v an w, from the threshold a and from the arrival rate of the system λ.

[vi] the development of the following expression is not reported due to the complexity of the result

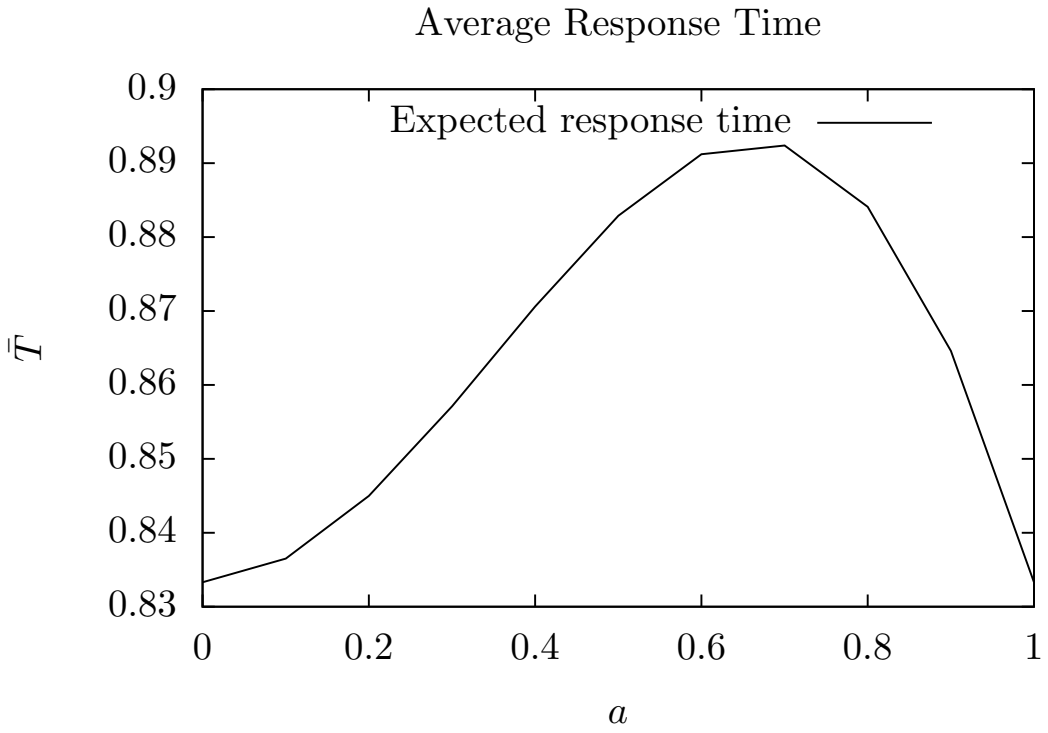

Figure 5.5: Average Response Time computed for uniformly sized job (the simulation was done with a Uniform distribution with parameter $(0, 1)$). As describe in the next section we can notice a worsening of the average response time when we use the threshold

In Figure 5.5 we reported the average response time of the simulation done with a Uniform distribution with parameter $(0, 1)$. Unlike the results obtained with the jobs generated by exponential and hyper-exponential distributions, with this distribution we always have a worsening of the average response time regardless of the choice of the value of the threshold a.

Chapter 6

Simulation and results

The aim of this chapter is to present our simulators developed after having computed some performance indexes with the theoretical calculations. We present the main functions of the two simulators and finally we compare the theoretical results with those provided by the simulators.

After evaluating the various indexes through the theoretical calculations we designed two different simulators: one that replicates the behaviour of the model and one that is closer to the behaviour of the real system. Basically, the first simulator works on the queuing model by implementing the processor sharing discipline with multiple levels. This will be used to access the accuracy of the approximations introduced in the previous section for job size distribution which are not exponential. In contrast, the second simulator does not resort the processor sharing abstraction and works by decomposing the flows in packets and exploring an an actual possible implementation of the multi-level discipline.

We have chosen to develop the two simulators with two different program-

ming languages: the first one is written in *C* because it did not require the use of large data types and hierarchical structures, the second one was developed in *Scala* because we needed objects to represent the various components of the system, their hierarchy, and the interactions between them.
In the next two subsections we present the two simulators that we have developed. At the end we compare the result that we obtained with our simulators (the results are very similar to each other except for some approximations or rounding errors) and with the theoretical calculations. The simulators are downloadable from `https://github.com/GiorgioMagnan/2-level-processor-sharing-queue`

6.1 Simulations of the model

Starting from the model described in Section 5 we designed a simulator that emulates its behaviour. As said previously, we decided to develop this simulator with a simple imperative language like C.
We used a lot of auxiliary variables to evaluate the various indexes that we had calculated theoretically and then we proceeded with a comparison. Furthermore, we used 3 arrays to store the flows that arrived in the system: the first two arrays were used to simulate the behaviour of the *high priority* and *low priority* queue while the third one was used to save the arrival time of the flows. The *main* function of the program (reported in Listing 6.1) initialises all the variables and the structures and then performs a cycle with thousands of interactions; for each interaction it evaluates what the next event will be (it could be a departure or an arrival of a flow) and then performs the action.

At the end we printed the statistics to be able to compare the values obtained with our simulator with those obtained from the theoretical calculations.

```
int main(){
    initialize ();
    event_type e; /*can be ARRIVAL or DEPARTURE*/
    for(int events = 0; events < MAX_EVENTS; events++){
        int ne = next_event(&e);
        process_event(e, ne);
    }
    print_stats ();
    return 0;
}
```

Listing 6.1: Simulator of the model: main function

The main part of the work is performed by the functions *next_event* and *process_event*.

The first function has to control the state of the queues and the relation between the completion time of the jobs and the *next_arrival* time. At the beginning, it assumes that the *type of next event* is an *ARRIVAL*, then it checks if the first queue is not empty: in this case it adds at each job in queue1 the quantum of works (calculated as $queue1\ length/MU$), then it checks if some job has been completed. If a job has been completed, then the *type of next event* becomes a *DEPARTURE*, otherwise it remains an *ARRIVAL*.

If the first queue is empty, it will check if the second one has elements: in positive case it performs the same operation of the first queue, adding to each job the quantum of service and checking if any of them has been com-

pleted. Also in this case, if a job is completed, then the *type of next event* will become a *DEPARTURE* otherwise it remains an *ARRIVAL*.
At the end of the function it returns the time elapses before the next event takes place.
In Listing 6.2 we report a schematization of the operations performed by the function.

```
double next_event(event_type *type){
    next_event = time_next_arrival;
    *type = ARRIVAL;
    if (queue1 not empty){
        for each job in queue1 do
            remaining_work -= (size(queue1) / MU);
            if (job completed){
                *type = DEPARTURE;
                next_event = time_next_departure;
            }
    }
    else if (queue2 not empty){
        for each job in queue2 do
            remaining_work -= (size(queue2) / MU);
            if (job completed){
                *type = DEPARTURE;
                next_event = time_next_departure;
            }
    }
    return next_event;
}
```

Listing 6.2: Simulator of the model: next_event function

The second important function is the *process_event*. At the beginning of the function we computed the work done that is performed in the elapsed time from the previous event, than we updated the number of jobs in the system. From this point forward, the function will have two different behaviours in the case that the *next event* is an *ARRIVAL* or a *DEPARTURE*.
In case of *ARRIVAL* the function updates the number of jobs arrived, then it updates the residual of each job already present in the queue, then again it adds the new job in the queue checking if it is greater than the threshold (*th*). If it is bigger, then it splits the job in the two queues, putting in the first queue a quantity of work equal to the threshold and the remaining part of the job in the second queue, otherwise it puts the job only in the first queue.
At the end, it computes the time for the *next_arrival* generated by an exponential distribution with parameter *LAMBDA*.
In case of *DEPARTURE* the function checks first of all if the first queue has only one job: in this case it is the end of its *busy period* and then it can update the value of *alpha1*; otherwise it checks if there is not any job in queue1 and if there are jobs in the second queue. If it is true it uploads the value of *alpha2*.
At this point it checks from which queue there is a departure:

- from queue1: it decreases the number of jobs in the queue1 and than it updates the residual job size in queue1. Then it has to check what is the dimension of the job:
  - if it is smaller than the threshold: it increases the number of *completions* and it evaluates the *response time* of the job;

  - if it is bigger than the threshold: it increases the number of jobs present in queue2;

- from queue2: it decrease the number of job in queue2, it increases the number of *completions*, it evaluates the *response time* of the job and then it updates the value of *alpha2*;

At the end of the function it increases the *simtime*.

In Listing 6.3 we report the structure of the *process_event* function.

```
1  void process_event(event_type t, double ts){
2      double elapse = ts - simtime;
3      double workdone;
4      if (jobs1 > 0)
5          workdone = elapsed * MU / jobs1;
6      else
7          workdone = elapsed * MU /jobs2;
8      totjobs += jobs*elapsed;
9      switch(t){
10         case ARRIVAL:
11             /*pseudocode in Listing 4*/
12         case DEPARTURE:
13             /*pseudocode in Listing 5*/
14     }
15     simtime = ts;
```

Listing 6.3: Simulator of the model: main function

In Listing 6.4 and 6.5 we report a schematization of the operations performed by the function when it has to manage an *ARRIVAL* or a *DEPARTURE*.

```
void process_event(event_type t, double ts){
case ARRIVAL:
    totalarrivals ++;
    /*update residual job sizes*/
    if(queue1 not empty){
        for each job in queue1
            remaining_work -= workdone;
    }
    else{
        for each job in queue2
            remaining_work -= workdone;
    }
    /*add job in the queues*/
    if(size > th){
        queue1 = th;
        queue2 = size - th;
    }
    else{
        queue1 = size;
        queue2 = 0;
    }
    next_arrival = ts + exponential(LAMBDA)
}
```

Listing 6.4: Simulator of the model:
process_event function (case ARRIVAL)

```
1  void process_event(event_type t, double ts){
2  case DEPARTURE:
3      if (size(queue1) == 1)
4          /*update the value of alpha1*/
5      else if (size(queue1) == 0 && size(queue2) > 0)
6          /*update the value of alpha2*/
7
8      /*Completion at queue1 and departure*/
9      if (queue1[jobs-1]>0.0 && queue2[jobs-1]==0.0 && size(queue1)>0){
10         /*update residual job size in queue1*/
11     }
12     /*Completion at queue1 and move to queue2*/
13     else if (queue1[jobs-1]>0 && queue2[jobs-1]>0 && size(queue1)>0){
14         /*update residual job size in queue1*/
15     }
16     /*completion at queue2*/
17     else if (queue1[jobs-1]==0 && queue2[jobs-1]>0 && size(queue1)==0 && size(
       queue2)>0){
18         /*update residual job size in queue2*/
19         /*update alpha2*/
20     }
21     else{
22         /*SOME ERROR IN THE SIMULATION*/
23     }
24 }
```

Listing 6.5: Simulator of the model:
process_event function (case DEPARTURE)

6.2 Simulations of the system

In order to simulate the model that we describe in Section 5, applied in a system with different components, we decide to use a programming language that allows us to emulate the behaviour of each entity and that guarantees a hierarchy among the components: for these reasons we decided to use an object oriented language like *Scala*.

The *main* function of the program initialises an object that represents the *Scheduler* of our system and then it calls the *simulate* method of the scheduler that starts the simulation. At the beginning of the method, we sent a series of flows to make sure that the queues are not empty at the beginning of the real simulation. At the end of the method, we printed the statistics in order to compare the values obtained with this simulator with those obtained from the theoretical calculations.

```
1  class Scheduler {
2      def simulate(): Unit = {
3          while(SW.getFlowsSent < Scheduler.WarmUpFlows)
4            executeOneEvent()
5          SW.reset()
6          while(SW.getFlowsSent < Scheduler.TotalFlows)
7            executeOneEvent()
8          SW.printStatistics()
9      }
10 }
```

Listing 6.6: Simulator of the system: Scheduler.simulate()

At each iteration the scheduler calls the *executeOneEvent* method that must update the residual job size of flows in the system (it call the *forwardTime* method) and it must check if the next event is a departure or an arrival. It checks in the list of *SimulationEntity* which element has the minimum *NextEventTime*, in fact if the minimum is of type *Flow* we have a departure, otherwise, if it is a *FlowGenerator*, we have an arrival.

In case the next event is of type *FlowGenerator*, it creates a new flow and adds it to the list of entities in the system, then it calls the *executeEvent* which simply calculates the time between the next arrival by generating a random number from a Poisson distribution, and then again it returns to the Scheduler and moves on to the next iteration.

In case the next event is of type *Flow* then it updates the remaining number of packets that the flow has to send and calculates the time between the next arrival with a Poisson distribution (it execute this two operation by calling the *executeEvent* method), and then again it sends the packet to the *Switch* object (with the *receivePacket* method). At the end, it checks if the flow is completed and in case of positive answer it removes the flow from the list of entities.

```
1  private def executeOneEvent():Unit = {
2      val Next = getNextEvent
3      forwardTime()
4      Next match{
5          case fg: FlowGenerator => {
6              val F: Flow = new Flow(Scheduler.FlowRate)
7              entities = F :: entities
8              fg.executeEvent()
9          }
10         case f: Flow => {
11             f.executeEvent()
12             SW.receivePacket(flowPackets(f.getID)
13             if(f.getRemainingPackets == 0L) {
14                 entities = entities. filter (_ != f)
15         }
16     }
17 }
```

Listing 6.7: Simulator of the system: Scheduler.executeOneEvent()

The *Scheduler* maintain a series of field that coordinate the operation of the entire system. The most important are *SWRate* and *FGRate* that represent respectively the rate service of the *Switch* and the rate of the *FlowGenerator*. Furthermore, it maintains the number of flows that must be generated in the warm up and in the simulation.

It maintains a referene to the switch, a reference to the flow generator, a list of entities (that initially contain only the switch and the flow generator) and a map with the history of packets sent for each flow.

In Listing 6.8 we report all variable that store the *Scheduler* that are useful to coordinate the system.

```
1  object Scheduler {
2    val SWRate: Double = 1
3    val FGRate: Double = 0.8 * SWRate
4    val FlowRate: Double = SWRate * Flow.AverageSize / Packet.AverageSize
5    val TotalFlows: Int = 90000
6    val WarmUpFlows: Int = 25000
7  }
8
9  class Scheduler {
10     private val SW: Switch = new Switch(Scheduler.SWRate * Flow.AverageSize /
       Packet.AverageSize)
11     private val FG: FlowGenerator = new FlowGenerator(Scheduler.FGRate)
12     private var entities : List[SimulationEntity] = Nil
13     private var flowPackets: Map[Long, Packet] = Map[Long, Packet]()
14     entities = SW :: FG :: entities
15 }
```

Listing 6.8: Simulator of the system: Scheduler fields

The fundamental component of the system is the *Switch*, it has a lot of auxiliary variables that are used to calculate the final statistics and a variable that indicates the dimension of the threshold. Furthermore, it has a map to store how many packets of a flow have already been sent, a variable that indicates the time between the next event and two lists to distinguish the two queue's levels.

```
class Switch(serviceRate: Double) extends SimulationEntity(serviceRate) {
    private val sizeThreshold: Long;
    private var packetSent: Map[Long, Long] = Map[Long, Long]()
    nextEvent = Double.MaxValue
    private var inputQueue = mutable.MutableList[Packet]()
    private var waitQueue = mutable.MutableList[Packet]()

    /*Auxiliary variables*/
    private var totalFlowsSent: Int = 0
    private var totalResponseTime: Double = 0.0
    private var simulationTime: Double = 0.0
    private var warmUpTime: Double = 0.0
    private var arrivalTime: Map[Long, Double] = Map[Long, Double]()
    private var avgQueueLength: Double = 0.0
}
```

Listing 6.9: Simulator of the system: Switch

Another important method of the Switch is the *receivePacket*, in this method we checked if the arrived packet is the first of the flow by checking into the *arrivalTime* map if the ID of the flows is already present, if it is not present it is added into the map.

After that we checked if the packet that has just arrived should be added in the first or second queue by checking the history of the packets received from the flow to which the packet belongs. If the flow has already sent a number of packets greater the threshold then the packet it is put in the second queue otherwise in the first one.

```
1  def receivePacket(p: Packet): Unit = {
2      if (!arrivalTime.contains(p.getFlowID)) {
3          arrivalTime += (p.getFlowID -> simulationTime)
4          packetSent += (p.getFlowID -> 0L)
5      }
6      if (packetSent.get(p) < sizeThreshold){
7          insert (p, inputQueue)
8      }
9      else{
10         insert (p, waitQueue)
11     }
12 }
```

Listing 6.10: Simulator of the system: Switch.receivePacket()

The last important method is *executeEvent*, it has the task of determining from which queue it has to pick up the next packet tha has to be sent, after it checks if the packet that it has chosen is the last of the flow and in this case updates some variables that are used to calculate the statistics and the performance indices. In all the cases it call the *send* method of the packet and then it updates the number of packet sent in the map with the history of all flows.

```
override def executeEvent(): Unit = {
    val NextPacket = getNext
    if (NextPacket.getRemainingPacketsNumber == 1) {
        timeFrame = simulationTime - flowArrivalTime(NextPacket.getFlowId)
        flowTotalResponseTime += timeFrame
        NextPacket.endFlow()
        flowArrivalTime -= NextPacket.getFlowId
        totalFlows += 1
    }
    NextPacket.send()
    packetSent += (NextPacket.getFlowID -> (packetSent(NextPacket.getFlowID) +
    1))
}
```

Listing 6.11: Simulator of the system: Switch.executeEvent()

Other methods and other classes are present in our project but are only helpful in calculating final indices or for carrying out operations in our data structures. In Figure 6.1 we have reported the UML diagram to give a general idea of the project structure.

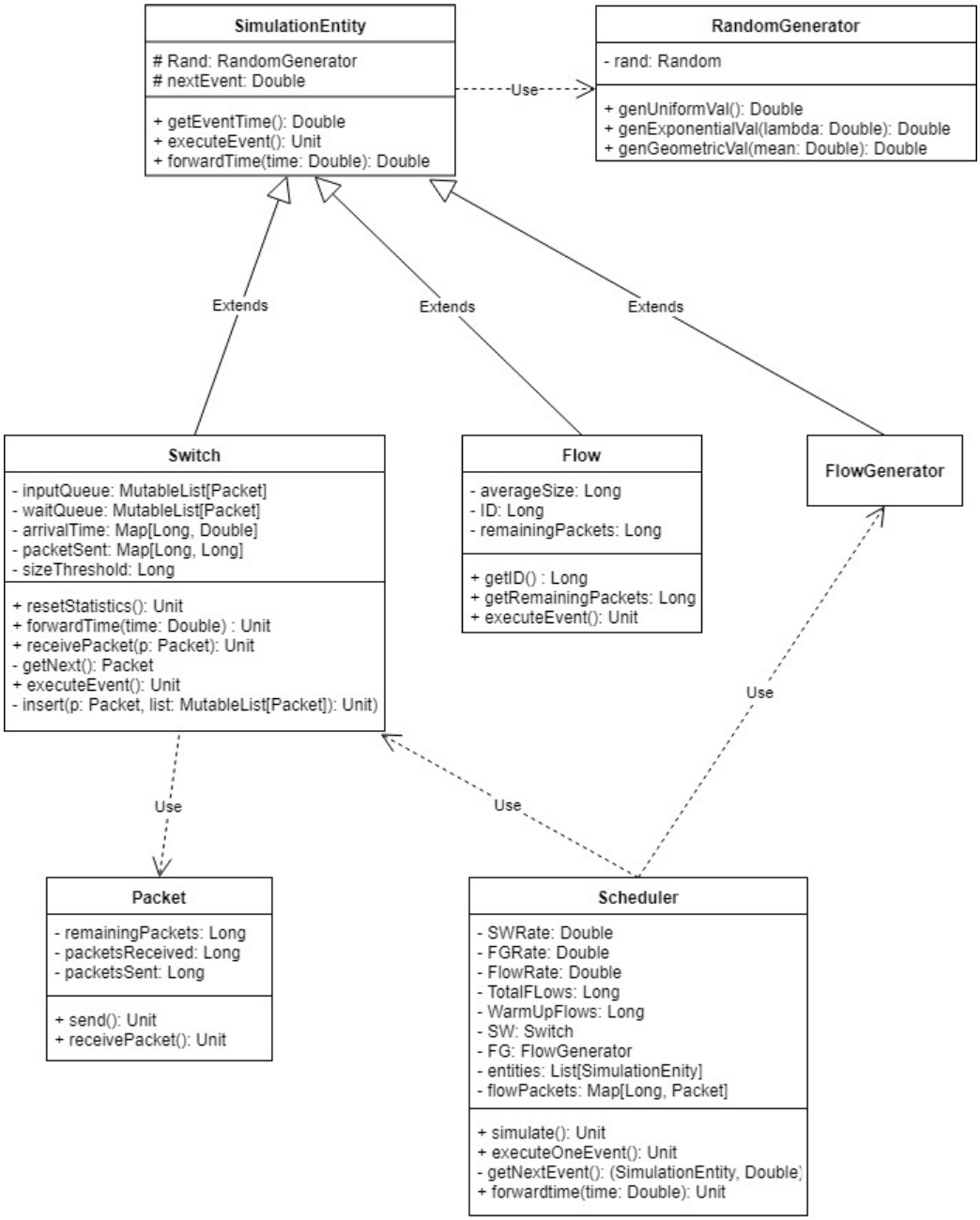

Figure 6.1: UML of the components of the simulator

6.3 Comparison of the results

In Table 6.1 we report the result obtained with our model in which the arrivals are distributed with an exponential process and the parameters are $\mu = 1$ and $\lambda = 0.8$. Henceforth, we will only report the data obtained from one of the two simulators because they are very similar to each other.

Remembering that as we have seen in equation (5.4.4) the average response time is $\bar{T}_R = 1/(\mu - \lambda)$ independently from the choice of the threshold so, with our data, we expect that $\bar{T}_R = 5$. As we can notice all our result obtained with the theoretical computations and the simulations are in between 4.71 and 5.00, we can consider that the values smaller than 5 are due to approximations of the calculations. No result below 5 can be considered as an improvement due to the use of our model or of the threshold.

a	Simulated	Computed
0.0	4.9856	5.0000
0.5	4.7094	4.7804
1.0	4.7067	4.7163
1.5	4.7346	4.7308
2.0	4.7633	4.7592
2.5	4.7831	4.7898
3.0	4.8046	4.8201
3.5	4.8425	4.8489
4.0	4.8820	4.8753
4.5	4.8987	4.8986
5.0	4.9416	4.9187

Table 6.1: Average response time with exponential distribution

In Figures 6.2 and 6.3 we report the average response time obtained with the theoretical computations and the simulations. We can notice that the result obtained with the theoretical computations and with the simulations (especially in the case of hyper-exponential distribution) are very similar to each other.

In Figure 6.2 we can notice that, when the threshold is set to 0, we have the maximum measure for the average response time; changing the level of the threshold, we obtain an improvement of the average response time.

We can be satisfied with the results obtained, the improvement achieved reaches up to 6/7%, the average response time is always under the maximum for all the possible values of the threshold and we are always under the

level that we can obtain with a system that use a *M/M/1* queuing discipline. Furthermore, the computed and simulated data are very similar to each other, the maximum difference in fact is around 2%.

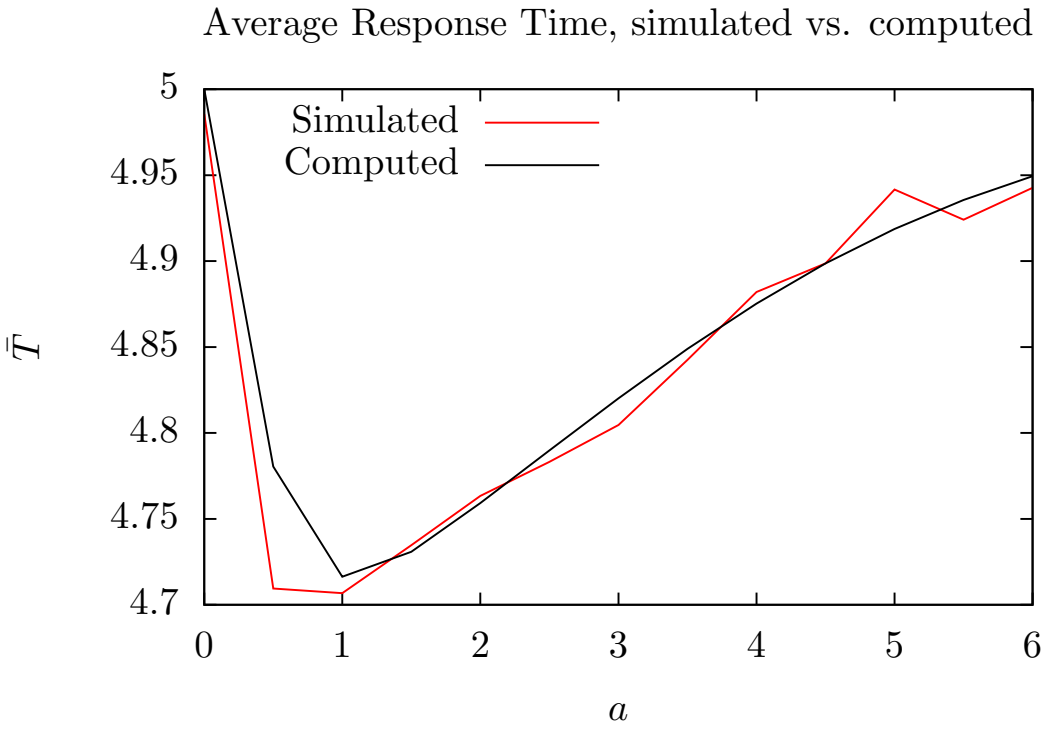

Figure 6.2: Average Response Time Hyper-exponential distribution (the simulation was done with two exponential distribution has rate respectively equal to 1/0.2 and 1/1.8 and the probability to choice one distribution or the other is equal to 0.5). As described in the previous section we can notice that we improved the response time of 7/8% with a difference from theoretical calculations and results obtained with the simulations of 2%

In Figure 6.3 we report the result obtained with flow generated by a *Uniform (0,1)*. In this case, however, we note that without threshold (so when $a =$

0 or $a = 1$) we have the best average response time; in all other cases, when a threshold is used (so when $0 < a < 1$), we have a deterioration of performance. Also in this case, the results obtained with the theoretical calculations and with the simulations, are similar to each other: the error is around the 3%.

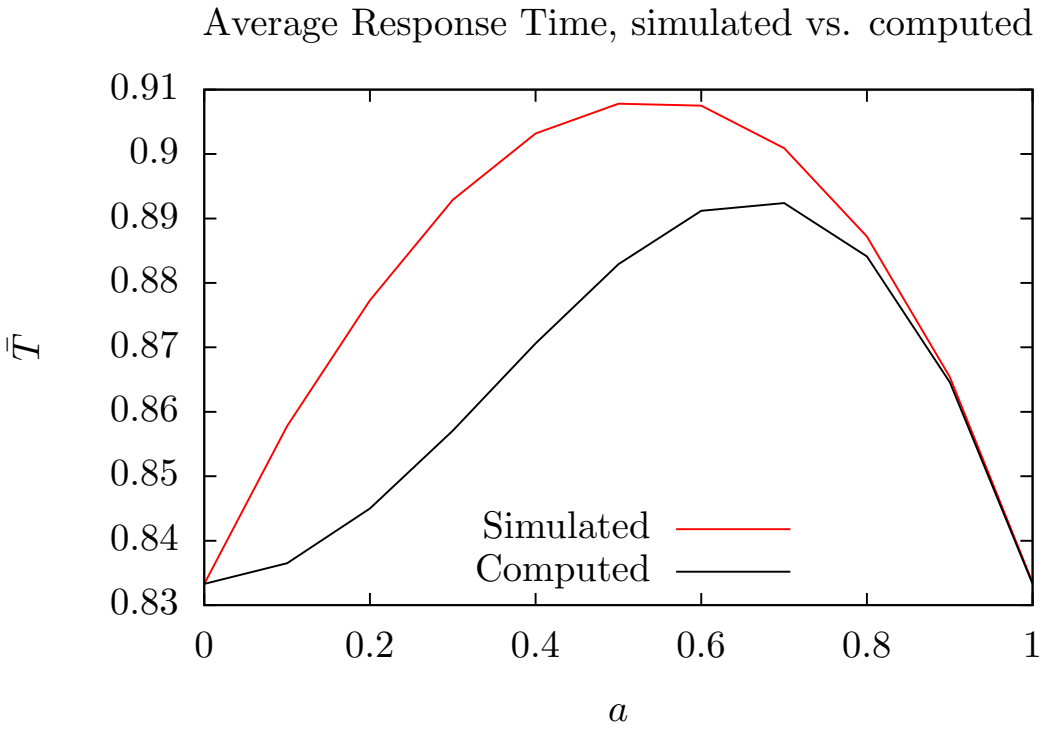

Figure 6.3: Average Response Time Uniform distribution (the simulation was done with a Uniform distribution with parameter $(0, 1)$). As described in the previous section we can notice that without the threshold we have the best value of the average response time, in the other cases we have a deterioration of the value. The difference between the theoretical calculations and the simulations is around the 3%

Chapter 7

Networking in the Linux kernel

In this chapter we introduce the Linux operating system, we focused our attention in particular on the traffic control of this system, taking into account queuing disciplines, classes and filters.

Linux is an open source operating system based on *Unix* built with a monolithic kernel. The kernel, which handles process control, networking (including facilities for firewalling, forwarding and traffic shaping), accesses to the peripherals and file systems, it is built in a modular way so that it is easy for developers to add or remove functionality [24].
The networking subsystem could be divided into four parts [24, 25]:

- network core: includes code that is independent from the protocol that the system could use, includes some generic datagram routines, data structures and device interfaces;

- network protocols: include specific code for specific type of networks

like Ip and Ethernet;

- network scheduler: includes routine to handle traffic priority and shaping;
- network drivers: include functionality to handle specif networking hardware;

7.1 Traffic control

From *Linux 2.2.0*, traffic control has been part of the distribution of the Linux kernel [26]. Traffic control consists of a set of operations and mechanisms that allows to manage packets sent and received by a machine. Figure 7.1 shows how the kernel processes data received by the network and how it generates data for the network. Incoming packets are examined and then forwarded to the network or passed up to the higher layer of the protocol in order to be processed. After the forwarding phase the packets are inserted in the correct output interface and, at this point, the traffic control takes action. [27]

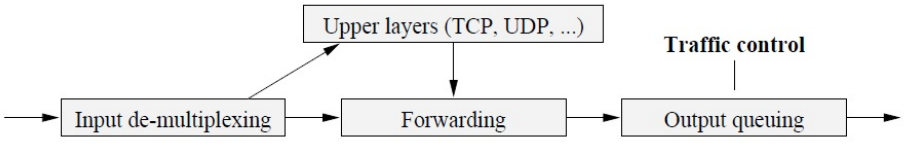

Figure 7.1: Networking data processing

Traffic control can decide if a packet can be queued or dropped, it can decide the order in which the packets have to be sent and it can slow down the sender. Traffic control is composed of three parts: queuing discipline, classes

and filters. As with most components, traffic control is also modular to allow the use of multiple different traffic control schemes.
Queuing disciplines are the basic of a traffic control system, it includes zero or more filters to direct traffic into classes and zero or more classes to prioritise traffic. Each class can contain another queuing discipline with its own classes and filters [26, 27].

7.1.1 Queuing disciplines

Queuing disciplines can be *classless* or *classful* but each network device has its own queuing discipline that controls how packets that pass from the device are treated. The most simple queuing discipline is a single queue that works with FIFO discipline where all packets are stored in the order in which they arrive and they are served one after the other respecting the order of arrival.

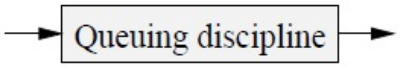

Figure 7.2: FIFO queuing discipline

On the other hand there are queuing disciplines that are *classful* and therefore use filters to distinguish packets in different classes and process each class in a different way. A classful discipline includes one or more classes with filters and a scheduler that manages the queues.
When a packet arrives, the system calls the *_enqueue()* function of the selected scheduler based on the filters configured in the discipline. The function examines all the filters until one of them corresponds to a match: at this point the packet is inserted in the queue of the corresponding class. If a packet

does not match any filter, it is associated with some default class.
When the kernel decides, it calls the *_dequeue()* function, the scheduler can make its decision based on different criteria like priority, delays of the queue and used traffic [24, 27].
In addition to the functions *_enqueue()* and *_dequeue()*, queuing disciplines provides several functions:

- _drop: drops one packet from the queue;
- _init: initialises and configures the queuing discipline;
- _change: changes the configuration of the queuing discipline;
- _reset: restore the initial state of the queuing discipline;
- _destroy: removes a queuing discipline;
- _dump: returns information used for maintenance [26, 27];

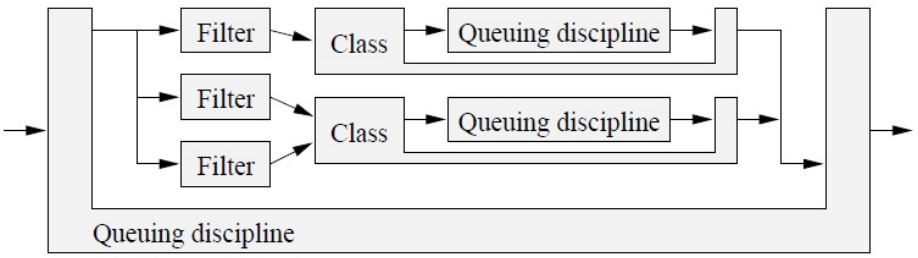

Figure 7.3: Queuing discipline with filters and classes

7.1.2 Classes

Classes are used by the queuing disciplines to differentiate the traffic, they are identified by an unique ID assigned by the user, they have a set of variable and a queue to put packet that belongs to the class.
Each class has a set of function like:

- get: returns the internal ID;
- put: called when a class that was previously referenced is dereferenced;
- change: used to create new class or to change the priority of an existing class;
- delete: deactivates and removes the class if it is not used;
- leaf: returns the queuing discipline of the class;
- graft: returns the queuing system in use and adds a new queuing discipline to the class;
- walk: invokes a callback function for each class of the queuing discipline;
- bind_tcf: binds a filter to the class (similar to *get*);
- unbind_tcf: removes a filter from the class (similar to *put*);
- tcf_chain: returns a link to the list of filters of the class;
- dump_class: returns information for maintenance and diagnostic data [27];

7.1.3 Filters

Filters are used by queuing discipline to manage the incoming traffic, in fact they assign each packet to a specific class. Filters are saved in a sorted list in ascending order based on priority. Each filter could access to all information of the packet and could have an internal structure that controls internal elements [24, 27].

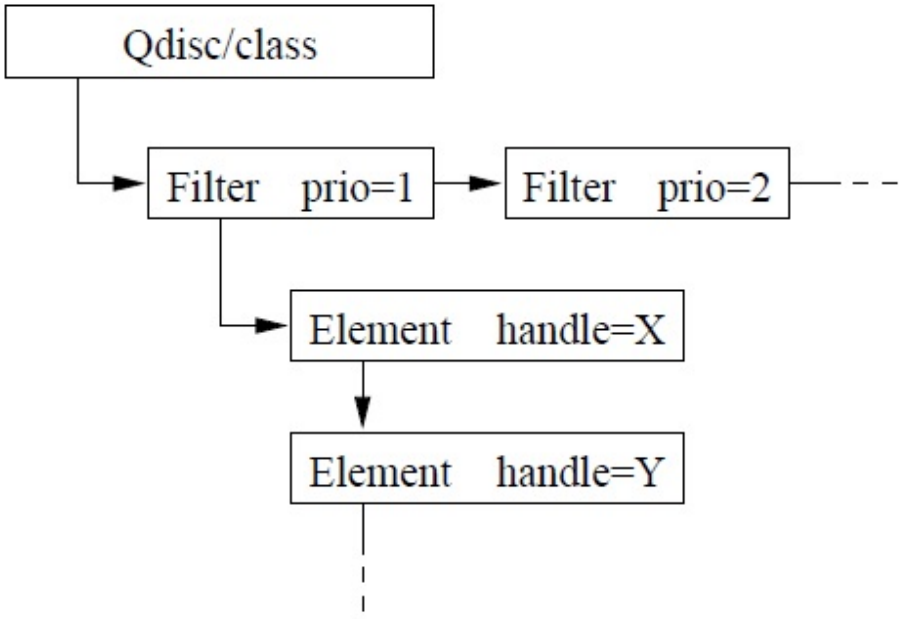

Figure 7.4: Structure of a filter with internal elements

As for the classes and the queuing disciplines also the filters have a series of operations to use them:

- init: initialises the filter;
- get: returns the internal ID of the filter;

- put: called when a filter that was previously referenced is no longer used;
- change: changes the configuration of an existing filter or configures a new filter;
- classify: returns the classification of the filter;
- destroy: removes a filter;
- delete: deletes an element of the filter;
- walk: invokes a callback function for each element of the filter;
- dump: returns information for maintenance and diagnostic data [27];

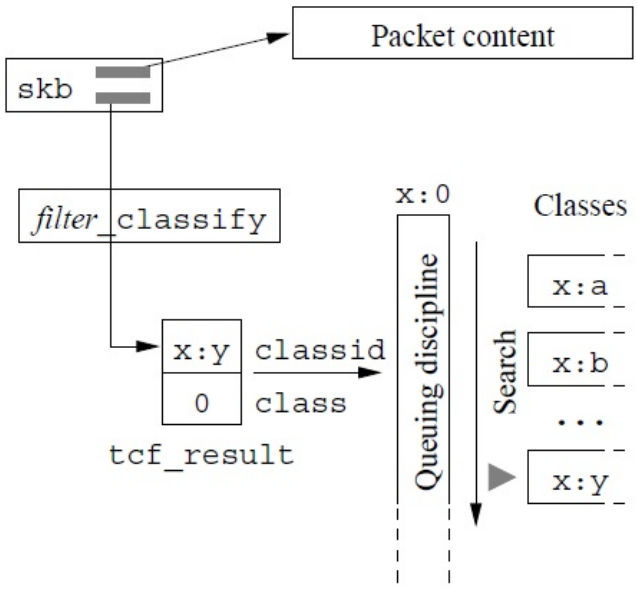

Figure 7.5: Traffic control: General procedure

7.1.4 Policing

Policing is used when one wants to impose traffic limits, they can be used in any part and in any moment in the traffic control, there is not a specific point where they could be applied. There are four types of policing:

1. policing decision by filter;
2. dropping of a packet from a queuing discipline;
3. dropping of a packet when a new one is enqueued;
4. refuse to enqueue a packet;

As we have seen in the section 7.1.3 the function *classify* of filters returns different types of policy value to indicate a different state, they are:

- TC_POLICE_OK: no treatment required;
- TC_POLICE_RECLASSIFY: packet was selected by the filter but it exceed some bound and it has to be reclassified;
- TC_POLICE_SHOT: packet was selected by the filter but it violate the bound and it should be rejected;

Furthermore, filters can use the *tcf_police* function to control if a flow is conformed or not.
The second policing is used when a queuing discipline decides to drop a packet to create new space for a packet that was already in queue but arrives from another class more important than the previous one.
The third policing eliminates the packets that were already in the queue, this

happens when the *enqueue* function is called and the new packet has higher priority than the previous one [24, 27].

Chapter 8

Linux schedulers

The scheduler is a component of the kernel and, as seen in Section 3, it manages the traffic entering and leaving the machine. There are different scheduling algorithms, in the next subsections we will see two scheduling algorithms widely used in Linux systems.

8.1 CoDel

CoDel (Controlled Delay Management) was developed by Van Jacobson and Kathleen Nichols in 2006 as an evolution of Random Early Detection (RED) algorithm. A first implementation was written in 2012 by Dave Tht and Eric Dumazet for the Linux kernel. It is designed to overcome the problem of bufferbloat in network links, it is parameterless, it controls delay and it scales from simple to complex routers [28].
It introduces three innovations [29]:

- it uses the *local minimum queue* instead of using queue size or queue

average size;

- it uses a single variable of the minimum delay to understand where the delay in the queue is;
- the queue size is measured in packet-sojourn time in the queue (it is not measured in bytes or packets);

Moreover, CoDel distinguishes between *good* and *bad* queues: the first one does not have bufferbloat, the network link utilization is maximised, and so, the management algorithm can ignore it; the second one, instead, has bufferbloat, low utilization and constant high buffer delay, and so, the monitoring is constant and a lot of packets are dropped.
Since CoDel measures the minimum packet sojourn time, there is no need for blocks in the implementation, in fact it can be modified only when a packet is dequeued. CoDel compares the minimum local queue with a target value: if it is lower, there is not the necessity to drop packets, otherwise, if it exceeds the target for an interval, the system starts to drop packets.

8.1.1 Main functions

In this section are presented some functions that are the core of CoDel algorithm, the code present in this subsection is taken from *sch_codel.c* of Linux v.18.04(64 bit) LTS.

The first thing that we analyze is the *enqueue* function, that uses the generic *enqueue* function to add the packet in the queue. Furthermore, it saves the actual timestamp that will be used in *dequeue* to calculate the sojourn time of the packet [30].

```
1 void codel_queue_t::enque(packet_t* pkt){
2     pkt->timestamp() = clock();
3     queue_t::enque(pkt);
4 }
```

Listing 8.1: CoDel enqueue function

The next piece of code is an auxiliary function that dequeues the packet and checks if the sojourn time is below the *target* [30].

```
dodeque_result codel_queue_t::dodeque(time_t now){
    dodeque_result r = { 0, queue::deque() };
    if (r.p == NULL) {
        first_above_time = 0;
    } else {
        time_t sojourn_time = now - r.p->tstamp;
        if (sojourn_time < target || bytes() < maxpacket) {
            // went below so we'll stay below for at least interval
            first_above_time = 0;
        } else {
            if (first_above_time == 0) {
                // just went above from below. if we stay above
                // for at least interval we'll say it's ok to drop
                first_above_time = now + interval;
            } else if (now >= first_above_time) {
                r.ok_to_drop = 1;
            }
        }
    }
    return r;
}
```

Listing 8.2: CoDel auiliary function for dequeue

The previous function returns a struct with two values, a *bool* that indicates if the packet has been over the *target* for more than *interval*(so it has to be dropped) and the dequeued packet [30].

```
1 typedef struct {
2     packet_t* p;
3     flag_t  ok_to_drop;
4 } dodeque_result;
```

Listing 8.3: CoDel struct dodeque_result

The core of the *CoDel* is in the *codel_queue_t::deque* function and it has some checks to manage different states of the system.
In Figure 8.1 we report a schematisation of the various branches of the function. From the figure we can notice that the function first checks that there is an element in the list. Then, before performing a dequeue of a packet, the function checks that the system is not in drop phase.

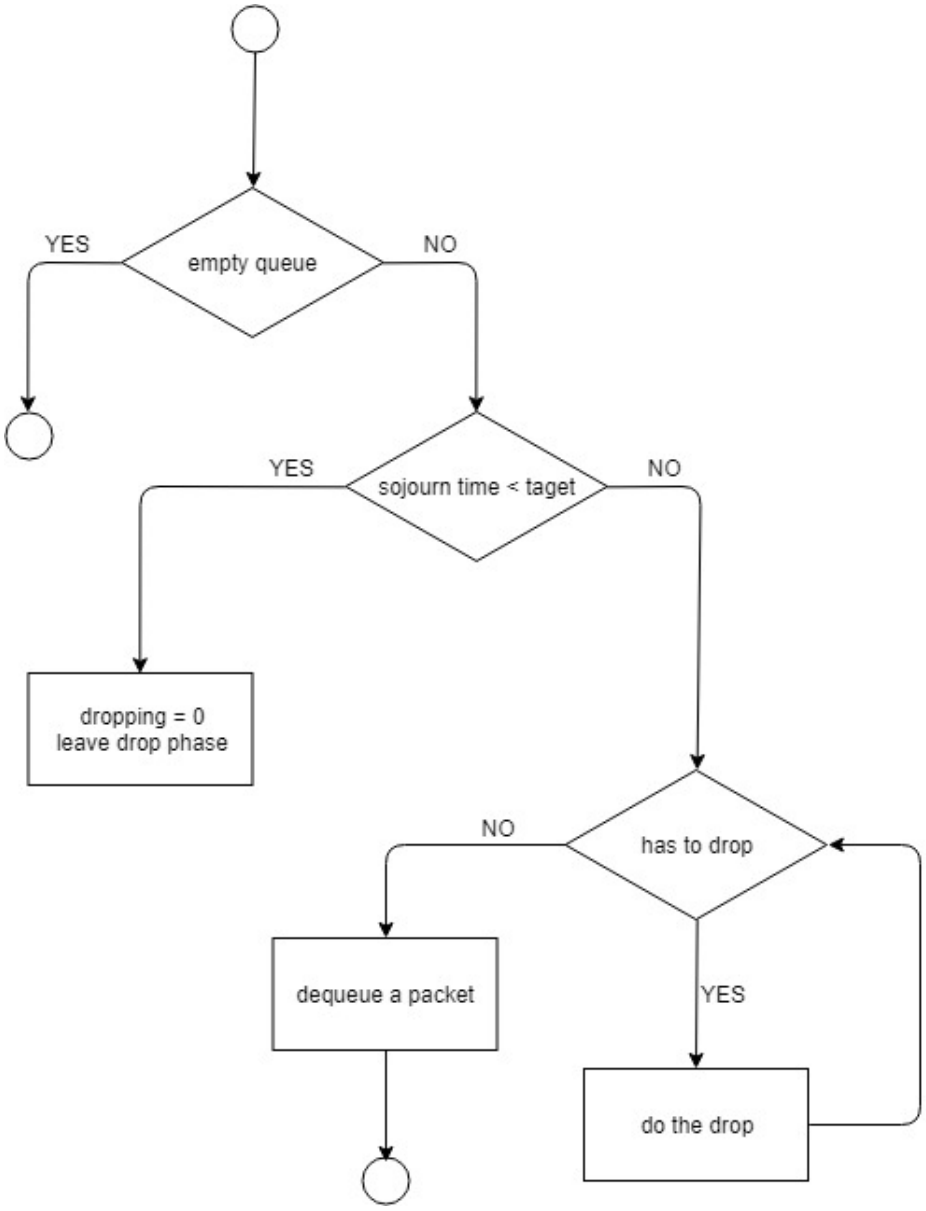

Figure 8.1: CoDel dequeue function general idea

At the beginning we check if we are in drop mode, then we check if we have to leave the drop phase (if the queue is empty or if the sojourn time is under the target) or if we have to continue to drop packets [30].

```
1  packet_t* codel_queue_t::deque(){
2      time_t now = clock();
3      dodeque_result r = dodeque();
4      if (r.p == NULL) {
5          // an empty queue takes us out of dropping state
6          dropping = 0;
7          return r.p;
8      }
9      if (dropping) {
10         if (! r.ok_to_drop) {
11             // sojourn time below target - leave dropping state
12             dropping = 0;
13         }
```

Listing 8.4: CoDel dequeue function: check drop mode phase

If we are in drop phase we drop the packet and we dequeue another one. After dequeuing, the system checks if it has to continue to drop other packets or if it has to leave the drop phase [30].

```
1        else if (now >= drop_next) {
2            while (now >= drop_next && dropping) {
3                drop(r.p);
4                ++count;
5                r = dodeque();
6                if (! r.ok_to_drop)
7                    // leave dropping state
8                    dropping = 0;
9                else
10                   // schedule the next drop.
11                   drop_next = control_law(drop_next);
12           }
13       }
```

Listing 8.5: CoDel enqueue function: drop packets phase

In the next branch, the system is not in dropping phase and it has to check if the sojourn time is bigger than the *target* for more then *interval*. In this case it has to drop a packet and then it enters in the dropping phase [30].

```
1     } else if (r.ok_to_drop && ((now - drop_next < interval) || (now -
      first_above_time >= interval))) {
2         drop(r.p);
3         r = dodeque();
4         dropping = 1;
5
6         // If we're in a drop cycle, the drop rate that controlled the queue
7         // on the last cycle is a good starting point to control it now.
8         if (now - drop_next < interval)
9             count = count>2? count-2 : 1;
10        else
11            count = 1;
12        drop_next = control_law(now);
13    }
14    return (r.p);
15 }
```

Listing 8.6: CoDel enqueue function: check sojourn time phase

8.2 FqCoDel

Th FqCodel (Fair Queuing Controlled Delay) algorithm is an evolution of *CoDel* and it combines a packet scheduler and an Active Queue Management (AQM) [31].

8.2.1 Main functions

As done for *CoDel* we present some main functions of *FqCodel*, the code present in this subsection is taken from *sch_codel.c* of Linux v.18.04(64 bit) LTS.

The first things that we analyze are that it has two ordered queues (*new_flows* and *old_flows*) in which packets are enqueued. When a packet arrives in the system it is put on the *new_flows* queue if the flow is not already in the queue. After an interval of time it is moved to the *old_flows* queue from which it is dequeued when the work of the flow is terminated [32].

```
1 struct fq_codel_sched_data {
2     struct list_head new_flows;
3     struct list_head old_flows;
4 }
```

Listing 8.7: FqCoDel struct sched_data

Differently from the *enqueue* function of the *CoDel* algorithm in this case the function is a bit more complicated. It has to manage some aspect of the system.

In Figure 8.2, we summarise the three main phases that characterise the functioning of the *fq_codel_enqueue*.

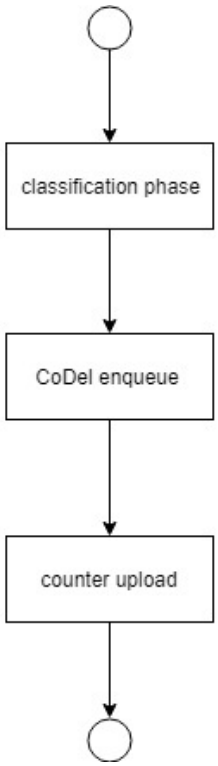

Figure 8.2: FqCoDel enqueue function general idea

The first phase of the enqueue function has the task of classifying the packets arriving in the system and redirecting them to the correct queue. By default, the classification is done with the hashing of the source and destination addresses and the port number modulo the number of queues [32].

```
1  static int fq_codel_enqueue(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff **
       to_free){
2      struct fq_codel_sched_data *q = qdisc_priv(sch);
3      unsigned int idx, prev_backlog, prev_qlen;
4      struct fq_codel_flow *flow;
5      int uninitialized_var(ret);
6      unsigned int pkt_len;
7      bool memory_limited;
8
9      idx = fq_codel_classify(skb, sch, &ret);
10     if (idx == 0) {
11         if (ret & __NET_XMIT_BYPASS)
12             qdisc_qstats_drop(sch);
13         __qdisc_drop(skb, to_free);
14         return ret;
15     }
```

Listing 8.8: FqCoDel enqueue function: classification phase

After having classified them, the packets are passed to the *Codel* algorithm to save the timestamp and then it is enqueued in the selected queue and the counter of bytes of the queue is updated by the size of the flow. It also checks if the *flowchain* is empty, in this case it moves the *new_flows* list in the *flowchain* [32].

```
1   codel_set_enqueue_time(skb);
2   flow = &q->flows[idx];
3   flow_queue_add(flow, skb);
4   q->backlogs[idx] += qdisc_pkt_len(skb);
5   qdisc_qstats_backlog_inc(sch, skb);
6
7   if (list_empty(&flow->flowchain)) {
8       list_add_tail(&flow->flowchain, &q->new_flows);
9       q->new_flow_count++;
10      flow->deficit = q->quantum;
11      flow->dropped = 0;
12  }
```

Listing 8.9: FqCoDel enqueue function: add flow in list phase

At the end, the counter is compared with the threshold and if it is above then some packets are dropped to get back into the limit [32].

```
    q->memory_usage += get_codel_cb(skb)->mem_usage;
    memory_limited = q->memory_usage > q->memory_limit;
    if (++sch->q.qlen <= sch->limit && !memory_limited)
        return NET_XMIT_SUCCESS;

    prev_backlog = sch->qstats.backlog;
    prev_qlen = sch->q.qlen;

    /* save this packet length as it might be dropped by fq_codel_drop() */
    pkt_len = qdisc_pkt_len(skb);
    /* fq_codel_drop() is quite expensive, as it performs a linear search
     * in q->backlogs[] to find a fat flow.
     * So instead of dropping a single packet, it drops half of its backlog
     * with a 64 packets limit to not add a too big cpu spike here. */
    ret = fq_codel_drop(sch, q->drop_batch_size, to_free);
    return NET_XMIT_SUCCESS;
}
```

Listing 8.10: FqCoDel enqueue function: threshold control phase

As for the *CoDel* algorithm, the bulk of the work is done by the function *fq_codel_dequeue*. It selects the queue, then it dequeues a job and then again it uploads the counter of bytes of the queue.

Obviously, it starts from the list of *new_flows*, if it is empty then it checks the *old_flows* queue [32].

```
1  static struct sk_buff *fq_codel_dequeue(struct Qdisc *sch){
2      struct fq_codel_sched_data *q = qdisc_priv(sch);
3      struct sk_buff *skb;
4      struct fq_codel_flow *flow;
5      struct list_head *head;
6      u32 prev_drop_count, prev_ecn_mark;
7
8  begin:
9      head = &q->new_flows;
10     if (list_empty(head)) {
11         head = &q->old_flows;
12         if (list_empty(head))
13           return NULL;
14     }
15     flow = list_first_entry (head, struct fq_codel_flow , flowchain);
```

Listing 8.11: FqCoDel dequeue function

If the queue has finished the credits then it is put in the *old_flows* list and the routine restarts again selecting another queue.

```
1    if (flow->deficit <= 0) {
2        flow->deficit += q->quantum;
3        list_move_tail (&flow->flowchain, &q->old_flows);
4        goto begin;
5    }
```

Listing 8.12: FqCoDel dequeue function: checking credits phase

When a queue is selected then it calls the *codel_dequeue* function, at the end it returns the packets that are removed from the list. If it does not return anything, then the scheduler has to manage different situations: in fact, if the function was applied to one queue of the *new_flows* list, then the queue is moved into the *old_flows* list. In the other hand, if the function was applied to the *old_flows* list then the queue is removed.
In Figure 8.3 is summarised the possible situation that the scheduler has to manage [32].

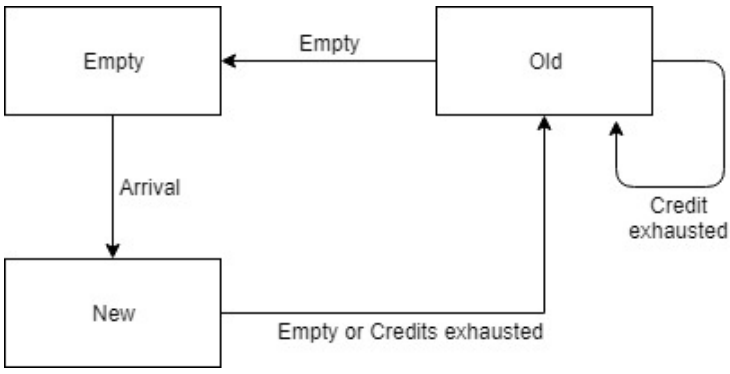

Figure 8.3: FqCoDel transition of queues

On the other hand, if a packet is returned to the scheduler then it uploads the counter of bytes already present in the queue and the available credits of the list [32].

8.2.2 Design of the multi-level queue in Linux

The implementation of the model presented in this thesis in reality is not very easy. Our idea for a possible future work is to start from one of the Linux schedulers analysed in the previous chapters (8.1 and 8.2) and modify them in order to adapt them to work with a multi-level queue. We thought about two possible solutions that would partially modify the structure of the two algorithms without twisting them. The first solution is to maintain the two lists *new_flows* and *old_flows* but changing the policy of moving the jobs from the first to the second queue according to our model (so as soon as a job in *new_flows* exceeds the threshold level of obtained service, it is moved to the *old_flows* list). The second idea is to double the number of lists in such a way to have 4 lists: 2 for *new_flows* and 2 for *old_flows*. In this way, we maintain the same politic used by *FqCodel* to distinguish flows (described in detail in Chapter 8.2). Moreover, for each group of flow, we add our policy distinguishing between large and small flows using two more queues. In this way we will have:

- *new_flows_high_priority* that contains the jobs that are classified by *FqCodel* as *new_flows* and whose obtained service is minor than the threshold;

- *new_flows_low_priority* that contains the jobs that are classified by *Fq-*

Codel as *new_flows* and whose obtained service exceeds the threshold;

- *old_flows_high_priority* that contains the jobs that are classified by *Fq-Codel* as *old_flows* and whose obtained service is minor than the threshold;

- *old_flows_low_priority* that contains the jobs that are classified by *Fq-Codel* as *old_flows* and whose obtained service exceeds the threshold;

The order with which the queues would be served will be:

1. *new_flows_high_priority*
2. *old_flows_high_priority*
3. *new_flows_low_priority*
4. *old_flows_low_priority*

In both cases, however, we must take into account the dropping policies of packets provided by the operating system in order not to overload the system. Furthermore, we have to add a data structure to count for each flow how many packets are processed. In this way we can identify the moment in which the obtained work threshold is exceeded.

Chapter 9

Conclusion

In this dissertation, we have presented some theoretical and practical aspects about scheduling in networking that have been addressed for years in the scientific literature. We have provided a new model that we believe could be implemented in reality and further developed.

Our study focused mainly on the current TCP/IP network design, we tried to understand the dimension of the flows using a series of queues and thresholds in order to distinguish the flows based on how much resources they used up to a certain moment.

In Chapter 2, we introduced the problem of scheduling in networking starting from what has been studied in the scientific literature. We focused, in particular, on works that concern the analysis of the distribution of the jobs' size. Then, we introduced some basic concepts that concern schedulers and queuing theory (discussed in Chapters 3 and 4) that underlies our research project.

All the concept discussed in Chapters 2, 3 and 4 have been used as a basis

for the development of our model.

Our model (described in detail in Chapter 5) was designed starting from the multi-level system proposed by Kleinrock in [1, 2]. Furthermore, in Chapter 5, after having described in detail the functioning of the model, we have calculated some indices (including the *average response time*, the *mean waiting time*, etc.) to understand the goodness of our model.

We have provided a punctual solution at the system of differential equations proposed by Kleinrock in [2] for jobs whose size follows a negative exponential random distribution. Secondly, we provided an approximate solution for jobs whose sizes do not follow an exponential random distribution.

After observing that the theoretical results obtained with our model were very satisfactory for jobs whose size follows an hyper-exponential random distribution (that corresponds to TCP flow distribution with high variance, as it happen in practice), we developed two simulators (that we described in detail in Chapter 6). The first simulator replicates the behaviour of the model, while the second one is closer to the behaviour of the real system. The first simulator was used to evaluate the accuracy of the approximation introduced with the theoretical calculation for jobs whose size does not follow an exponential distribution. We can be satisfied with the estimates obtained considering that, comparing the theoretically results with those obtained from the simulations, the introduced error was fewer than 3%.

The second simulator, instead, simulated the behaviour of our model applied in a real system, it decomposed the flows in packets, and simulated a possible implementation of the multi-level discipline in a real router.

In Chapter 6.3 we have analysed the results obtained with the theoretical

computations with those obtained with our simulators.

As far as the result obtained with the job whose size follows an exponential random distribution is concerned, we can say that the introduction of a threshold is useless with respect to the improvement of the expected response time, while it is still useful to improve the response time of the small jobs with respect to the bigger one. In fact, with the theoretical computation, we can observe that the *average response time* depends only on λ and μ, that are general parameters of the system. With the simulators we have obtained the same results of the theoretical computations without taking into consideration some errors introduced with approximations.

Instead, as far as the result obtained with jobs whose dimension follows an hyper-exponential distribution is concerned, we can observe an improvement of the *average response time* when the threshold levels change. With our simulation (executed with two exponential distributions with parameters, 0.2 and 1.8, considering that the probability of choosing one distribution or the other is equal to 0.5) we have observed an improvement of the 7% of the *average response time*.

Finally, as far as the result obtained with jobs whose size follows a uniform distribution is concerned, we can observe a deterioration of the performance (our simulation was done with a uniform distribution with parameter $(0, 1)$). The results that we have 0obtained are consistent with what has been said in [7] and [8] about the hazard rate of distribution. In fact, the hyper-exponential distribution has a decreasing hazard rate distribution, for this reason we can have an improvement of the *average response time*. On the other hand, the uniform distribution has an increasing hazard rate distribu-

tion, for this reason it cannot have an improvement of the *average response time*. In Chapter 7, we have seen that the traffic control of Linux consists of a set of elements that interacts in a lot of ways. Given the modularity with which the traffic control system has been implemented it is easy to add, modify or remove features to traffic control.

In Chapter 8, we discussed about two schedulers already present in Linux systems and we analysed their main functions in order to understand the functioning of the scheduling in the operating system. As described in Chapter 8.2.2 the implementation of a scheduler that respects our model is not simple because there are a lot of component that we have to manage in order to reproduce the multi-level queue system that we have described in this thesis (in particular we have to rewrite a module of the kernel which requires a complete knowledge of all the module that are used by the system to manage the scheduling of the packets). For these reasons we have proposed a general idea on how it could be developed.

In conclusion, we think that the results that we have obtained are satisfactory and we positively believe that the implementation can be done as a future work.

Acknowledgements

I would like to thank my supervisors Prof. Andrea Marin and Prof. Sabina Rossi for their support and guidance. They have given me the possibility to work to such an interesting project.

I also would like to thank Matteo Sottana and Fabrizio Romano that have collaborated to the study and the development of this project of thesis.

I would like to express my gratitude to my parents Sandra and Livio for their unconditional love and support. They have always sustained me and encouraged me during the whole run of study.

A particular thank goes to my classmates Andrea, Nicolò, Tommaso and Giorgia for the days we have spent together in university sharing studies and fun.

To Davide, Marco and all my other closer friends, I would like to express my gratitude for their friendship and their patience and support throughout these years.

Last, but not least, the greatest thanks goes to my girlfriend Veronica for her love and for having supported me every day. Without her support probably I would not have succeeded in reaching this objective.

Bibliography

[1] Leonard Kleinrock. *Queueing Systems*. Vol. I: Theory. (Published in Russian, 1979. Published in Japanese, 1979. Published in Hungarian, 1979. Published in Italian 1992.) Wiley Interscience, 1975.

[2] Leonard Kleinrock. *Queueing Systems: Volume 2: Computer Applications*. John Wiley & Sons New York, 1976.

[3] Konstantinos Psounis, Arpita Ghosh, Balaji Prabhakar, and Gang Wang. "SIFT: A simple algorithm for tracking elephant flows, and taking advantage of power laws". In: *Stanford University* (2018).

[4] Idris A. Rai, Guillaume Urvoy-Keller, Mary K. Vernon, and Ernst W. Biersack. "Performance Analysis of LAS-based Scheduling Disciplines in a Packet Switched Network". In: *SIGMETRICS Perform. Eval. Rev.* (2004).

[5] L. Schrage. "A Proof of the Optimality of the Shortest Remaining Processing Time Discipline". In: *Operations Research* (1968).

[6] Dinil Mon Divakaran, Giovanna Carofiglio, Eitan Altman, and Pascale Vicat-Blanc Primet. "A Flow Scheduler Architecture". In: ed. by Mark

Crovella, Laura Marie Feeney, Dan Rubenstein, and S. V. Raghavan. 2010.

[7] H. Feng, V. Misra, and D. Rubenstein. "PBS: a unified priority-based scheduler". In: *SIGMETRICS* (2007).

[8] Samuli Aalto, Urtzi Ayesta, and Eeva Nyberg-Oksanen. "Two-level Processor-sharing Scheduling Disciplines: Mean Delay Analysis". In: *SIGMETRICS Perform. Eval. Rev.* (2004).

[9] *QoS architecture models: IntServ vs DiffServ*. `https://learningnetwork.cisco.com/thread/121078`. Accessed: 2018-09-21.

[10] k. Ahlin. "Quality of Service in IP Networks". MA thesis. Linkoping University, 2003.

[11] R. Braden, D. Clark, and Shenker S. "Integrated Services in the Internet Architecture: an Overview". In: *IETF* (1994).

[12] K. Pandit. "Quality of Service Performance Analysis based on Network Calculus". MA thesis. Technische Universitat Darmstadt, 2006.

[13] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. "An Architecture for Differentiated Services". In: *IETF* (1998).

[14] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 1993.

[15] Sukumar Babu, Neelima Priyanka, and Sunil Kumar. "Efficient Round Robin CPU Scheduling Algorithm". In: *International Journal of Engineering Research and Development* (2012).

[16] Abbas Noon, Ali Kalakech, and Seifedine Kadry. “A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average”. In: *IJCSI International Journal of Computer Science Issues* (2011).

[17] Basit Shahzad and Muhammad Afzal. “OPTIMIZED SOLUTION TO SHORTEST JOB FIRST BY ELIMINATING THE STARVATION”. In: *Jordanian International Electrical Engineering and Electronic* (2005).

[18] Linus Schrage. “A Proof of the Optimality of the Shortest Remaining Processing Time Discipline”. In: *Operations Research* (1968).

[19] Nikhil Bansal and Mor Harchol-Balter. “Analysis of SRPT Scheduling: Investigating Unfairness”. In: *SIGMETRICS Perform. Eval. Rev.* (2001).

[20] L. Green. “Queueing Theory and Modeling”. In: *Handbook of Healthcare Delivery Systems* (2011).

[21] J. Sztrik. “Basic Queueing Theory”. MA thesis. University of Debrecen, Faculty of Informatics, 2016.

[22] M. F. Neuts. “The M/G/1 Queue with Several Types of Customers and Change-over Times”. In: *Advances in Applied Probability* (1977).

[23] Cristian Estan and George Varghese. “New Directions in Traffic Measurement and Accounting”. In: *SIGCOMM Comput. Commun. Rev.* (2002).

[24] Marko Luoma. “Implementation and Performance Analysis of a Delay Based Packet Scheduling Algorithm for an Embedded Open Source Router”. MA thesis. Helsinki University of Technology, 2007.

[25] Ankit Jain. *Linux Networking Subsystem, Desktop Companion to the Linux Source Code*. 2002.

[26] *A Practical Guide to Linux Traffic Control*. `http://borg.uu3.net/traffic_shaping/`. Accessed: 2018-02-15.

[27] Werner Almesberger. *Linux Network Traffic Control - Implementation Overview*. 1999.

[28] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar. "Controlling Queue Delay". In: *Internet Engineering Task Force (IETF)* (2012).

[29] *CoDel man page*. `http://man7.org/linux/man-pages/man8/tc-codel.8.html`. Accessed: 2018-09-03.

[30] *CoDel pseudocode*. `https://queue.acm.org/appendices/codel.html`. Accessed: 2018-09-03.

[31] *FqCoDel man page*. `http://man7.org/linux/man-pages/man8/tc-fq_codel.8.html`. Accessed: 2018-09-11.

[32] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. "The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm". In: *IETF* (2018).