Efficiency of Load Balancing Schemes in a Message Queuing System

Nicolas P. Jones

Master Thesis

Presented in March 2005

Acknowledgements

I would like to thank Dr. Daniel Bauer, my supervisor at IBM Research Rüschlikon (Switzerland) for his constant support and infinite patience. With his drive he pushed me to explore new areas, thus giving my project wider dimensions and scope. His constructive criticism and his encouragement in moments of doubt have been essential in the finalisation of my master thesis. Thanks to Dr. Bauer I have been given the chance to broaden my knowledge considerably over just six months. I am truly thankful.

The support I have received from both my supervisor Prof. André Schiper at EPFL and my manager at IBM, Dr. Paolo Scotton has been much appreciated.

I am grateful to Dr. Guy Anastaze for pointing me in the direction of IBM Research Rüschlikon, thus providing me with a privileged opportunity to spend six months at such a prestigious world laboratory.

Finally, I would like to say that it has been a real pleasure to work with my colleagues at IBM who have given me valuable advice, helped me to concentrate and also to relax.

The work for this thesis was carried out at

IBM Research GmbH, Zurich Research Laboratory

Contents

List of Figures

List of Tables

1 Overview

1.1 Introduction

This paper describes the research performed on the efficiency of load balancing schemes in a message queuing system.

A message queuing system is an element of a Message Oriented Middleware (MOM). MOM are a specific class of middleware that support the exchange of general-purpose messages in a distributed application environment. Asynchronous messaging is possible because of the message queue system that buffers and then distributes the messages to the receivers. The act of distributing the messages is the "load balancing" that this thesis focusses on.

The study was performed on a system that had already been implemented, and which used a message queuing system in an environment for dealing with messages sent by sensors at a high frequency. In the pipelined processing architecture under consideration here, messages are preprocessed and aggregated by multiple servers, called Edge Servers, before being delivered to an Application Server. Between the sensors and the multiple Edge Server, a Message Queue Manager entity receives the messages. It buffers these until Edge Servers are ready to process them, and then distributes these messages to the servers.

The original system, base of this research, had been implemented with a single policy for spreading messages. However, no research had been attempted as to which types of policies would be suitable for which arrangement or setup. This architecture can be implemented in very different environments where networks can be variable in speed, bandwidth, delays, and other similar parameters.

Hence there is a need to explore the performance of each chosen load balancing policy, and to implement new ones for comparison. This is the framework topic of this research.

1.2 Chapter Overview

This section summarizes the contents of each chapter.

Chapter 2: Problem Statement Position of the problem, current implementation and challenges to be tackled are explained in this chapter.

Chapter 3: Related Work Load balancing policies constitute a vast field of re-

search. They play a main role in this architecture. This chapter gives an overview of many popular algorithms, analyses them with respect to the architecture of this project and proposes a selection of them to be retained for experiments.

Chapter 4: Design Overview The architectural hurdles imposed by the nature of the chosen algorithms and the current implementation of the system are described and addressed at this stage of the document.

Chapter 5: Analytical Model Based on the understanding of the functioning of the system, an analytical model is proposed. It is used to predict the performance of the selected algorithms.

Chapter 6: Experiments This chapter presents the experiments that were run and the results that were measured. It provides comprehensive explanations and analysis of these.

Chapter 7: Further Improvements The diversity of the field of research lead to the emergence of some interesting ideas. The time limitation on the project imposed restrictions on the choice and not all ideas could be explored. Some further improvements and abandoned ideas are presented in this section.

Chapter 8: Conclusions Finally some conclusions to this project are written in this last chapter.

2 Problem Statement

The goal of this Masters Thesis is to explore the advantages and disadvantages of different load balancing algorithms, in an environment where messages from sensors are preprocessed. This study of performance of load balancing algorithms was carried out on an already existing architecture. The following chapter first presents the already existing setup, before specifying the challenges of this research.

2.1 Introduction and Background

Edge Servers are intermediate computation platforms in which data is processed, filtered and/or reformatted before arriving at its ultimate destination. Large distributed sensor applications where messages are sent to a central location do not scale well. In fact all event model architectures (and similar) that depend on a centralized mediator component do not scale well [Mei00]. Edge Servers reduce the message volumes that need to be handled by application servers and thus enable scalability. The architecture follows the idea of pipelined - or staged - processing that is realized using the Edge Server model. Figure 2.1 shows the general idea of pipelined processing and the Edge Server model. Data is gathered by sensors and forwarded to Edge Servers, where a first preprocessing step takes place. Depending on the application, multiple stages of Edge Servers process data and forward it towards the application server. The application server runs the main application and also interacts with the end-users. In the model, an application server consists of a software platform running over one or more machines. This software platform supports a set of APIs which simplify the application's interaction with platform components such as databases, directories, HTTP servers. An Edge Server, in contrast, is much simpler. It consists of a software platform running on a single machine. The platform contains a set of communication libraries as well as application dependent processing code.

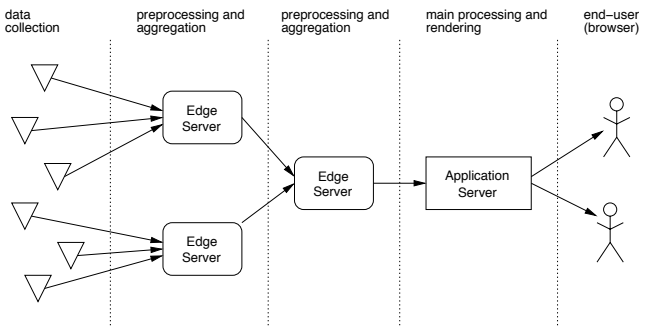

Figure 2.1: *Pipelined Processing*

2.1.1 Integration of Sensors

The standard approach to the connection of sensors and activators (collectively called transducers) within a control network is through the use of a field bus, for example (CAN) [Bos91]. Several approaches for integrating sensors into larger enterprise networks have been developed. These differ in the capacities they assume of the transducers. An excellent example on what can be done with sensor network is presented in [RBS05].

One approach is Arcom and IBM's MQ Telemetry Transport (MQTT) [mqt]. It is a TCP/IP based protocol that has been designed to connect SCADA (Supervisory Control and Data Acquisition) devices to a publish / subscribe messaging system. MQTT minimizes the required bandwidth by using a very small header of two to three bytes. It offers three levels of delivery assurance. "At most once" involves no MQTT acknowledgment, "at least once" involves a receipt from the pub/sub system and "exactly once" uses a receipt and receipt acknowledge. Furthermore, MQTT actively monitors connectivity using heart-beat messages in order to detect failures, which is important if a wireless environment is used. It supports the notion of a "last will" message that is published by the pub/sub system once the connection to a client is lost. MQTT implicitly assumes that sensors are capable of supporting TCP/IP, which is inappropriate for low-end sensors.

MQTT has been designed for message dissemination rather than message consumption and is therefore no solution for the sensors in this model. While sensors require a very light-weight protocol for producing messages, Edge Servers are able to handle a more complex protocol that allows for more functionality. In order to cover the different requirements of sensors and Edge Servers, an asymmetric message queuing system (AMQ) was developed, that uses a very light-weight protocol for senders and a more complex protocol that includes flow-control mechanisms for receivers. A typical usage scenario is depicted in Figure 2.2, where sensors enqueue messages using UDP/IP and Edge Servers receive these messages using a channel protocol over TCP/IP. While UDP/IP was used to attach the sensors, other light-weight protocols could also be used.

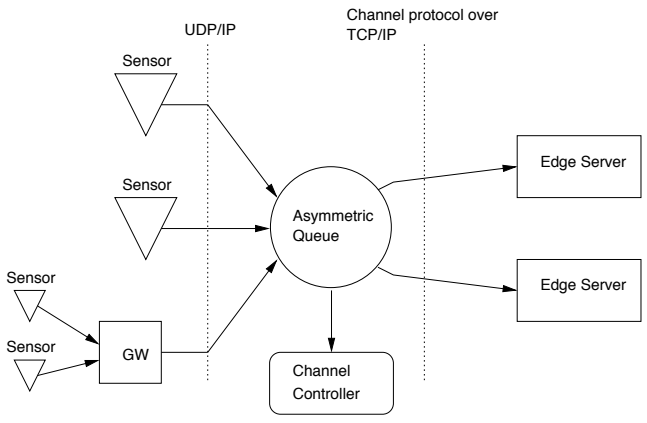

Figure 2.2: *Use of an Asymmetric Message Queue*

This approach allows the movement of data away from the sensors as quickly as possible, such that the sensors do not have to buffer messages. Typical sensor data

is redundant in nature and some loss can be tolerated, in which case UDP's "best effort" semantic is adequate. Alternatively, sensors might use TCP to transport messages reliably. However, message loss might still occur in case the queue overflows. In order to prevent such a situation, the AMQ sends warnings to its associated channel controller as soon as the queue size crosses a threshold. In this case, administrative counter-measures can be taken such as installing additional Edge Servers to drain the queue faster, or redirecting sensors to another, less loaded queue.

2.2 Research Items

On the Edge Servers, the receiver side, a channel protocol with a credit-based flow control mechanism on top of TCP is used. When subscribing to the AMQ, each receiver reports its initial capacity, i.e. configures the initial credit. The AMQ drains this credit each time it dispatches a message to a receiver. Receivers, on the other hand, renew their credits after they have processed a given number of messages. In the originally implemented architecture, this first scheme allowed the implementation of a load-balancing mechanism within the AMQ. As a prototype, a "priority-based" (also called "credit-priority") scheduler was implemented, that dispatched messages to the receiver with the largest outstanding credit.

The priority-based algorithm, as implemented, is globally expected to produce good results, thanks to the credit updates that reflect the relative processing capabilities of each Edge Servers. However some parameters in the system might diminish the performance of this scheme. The topic of this thesis is to investigate how good this load balancing algorithm is, and to investigate various other possible load-balancing mechanisms for the AMQ system. The proposed goals are to address the following points:

- To identify different load-balancing schemes that are or could be used by queuing systems. Examples are request/reply approaches, simple schedulers such as random or round-robin and feed-back based systems such as priority scheduling based on credit-dependant flow control.
- To establish a set of characteristics by which load-balancing schemes can be compared. Keywords here are performance, fairness, requirements on underlying message delivery protocol.
- To identify the parameter space of the system. Parameters here include relative performance of the involved components, network parameters such as throughput and delay.
- To compare the different schemes using analytical methods, simulation or measurement of implementations.

The first goal is crucial as it defines the direction of the whole research. Existing research in the field is vast. Algorithms have many different approaches to the task of load balancing, each-time with some advantages and some disadvantages. Their way of functioning is not always compatible with the current architecture. It is therefore worth considering a wide range of different algorithms. For this reason, a

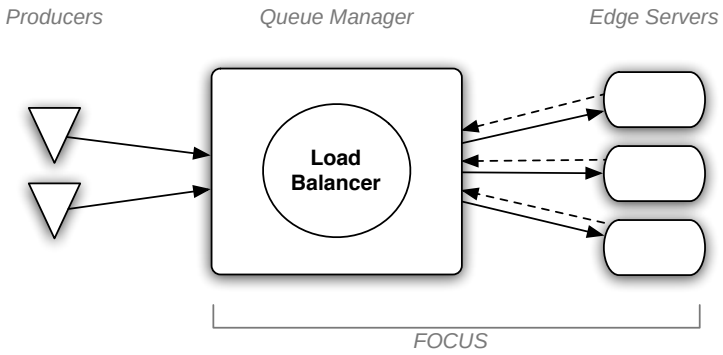

Figure 2.3: *Overview of the architecture of the chosen system for the experiments. The focus is on the load balancing and its effects.*

broad selection of load balancing algorithms is presented and discussed in Chapter 3.

The second point defines which are the elements that are important in the measures, and therefore influence which algorithms are chosen to be studied. Testing an algorithm that is known to guarantee fairness, when fairness is not useful, makes no sense. These characteristics are discussed at the end of Chapter 3, when talking about the chosen algorithms.

The third item, identifying the parameter space, is especially important for running some tests and measuring them. The higher the level of understanding of the elements involved in influencing the system, the more adequately the measurements can be made, and the easier it will be to explain the measured behaviors. The parameter space is described briefly a first time in Chapter 5.3 and in detail in Chapter 6.

The final goal, the comparison of the different schemes, investigates how to set up suitable experiments in order to reach the general aim of this thesis, namely the comparison of load balancing algorithms. It is further described in Chapter 4.

2.3 Miscellaneous

This section presents a few specific aspects that have not been mentioned before, or that merit an explicit explanation.

2.3.1 System architecture

Following what has already been said about the original implementation as a pipelined processing architecture, this subsection presents the architecture that is used for the reasoning and the measures.

The chosen architecture is fundamentally the same as the one setup in the original experiment. It is presented in figure 2.3. The main difference is that the focus is on the inter-relation between the Queue Manager (where the load balancer is) and the Edge Servers. This study focusses on the maximal performance of the load balancer. For this reason, it is assumed that the publishers are publishing faster than the rate at which the load balancer can distribute his load. Therefore the producers are included in the system, but they are not part of the focus of this study.

2.3.2 Emulation or Simulation

The title of this thesis implies the setting up of an experiment such as to measure the differences between a selection of load balancing algorithms. This statement brings immediate questions like: how should this be done? Is it more adequate to emulate or simulate the whole sensor-to-edge-server architecture? It is often faster to setup a simulation of the architecture on which experiments have to be run. Therefore it is an important question to ask and solve, especially as this project was conducted over a short period of only six months. The following subsection exposes the differences between emulation and simulation.

Emulation can be defined as the process by which a device imitates the actions of another. Simulation, on the other hand, can be described as the process of designing a model of a real system. It enables the exploration of the behavior of the system with this model and/or the evaluation of various strategies for the operation of the system. Emulation can be considered as more theoretical, and simulation more practical [emu].

The goal of this thesis is to explore the behavior of load balancing algorithms in a system under stress, which is complex and where timing issues are crucial. Load balancing is a subject where the theoretical expectations are well known most of the time and often easy to work out. However when such a distribution act is performed in a complex environment, predicting the results becomes more complicated. In order to make sure to capture all unforeseeable events and their influence on the load balancing algorithms, it was decided to use neither simulation nor emulation: the real system was actually implemented in order to give more importance to the results.

What is precisely done is described in the following list.

- The load balancing architecture is implemented for real.
- The operations of the network are emulated. The emulator used is *NIST Net* and is described in section 6.2.3.
- The sending of messages from the publishers is emulated.
- The processing task by the subscribers is emulated.

Although the processing of the subscribers is emulated, the time they spend waiting is real. Said more precisely, the subscribers do not perform any real task with the

received messages. From this point of view, their processing is emulated. However, in order to do this, the subscribers wait for a certain period of time. This task of waiting is real and blocks any reception of another messages, just as it would happen for a real subscriber.

The conclusion is that the real system is used and some components are emulated.

2.3.3 Connection

The connection between the Queue Manager and the clients is over TCP/IP. For this reason, the following assumption can be made:

- It can be assumed that messages don't get sent to clients that are down. If a client has crashed, the TCP connection gets closed and messages won't be delivered on this path.

3 Related Work

A great deal of research has already been done on the topic of Load balancing algorithms. This chapter presents a certain number of algorithms that were considered in the early stages of the project. They are briefly described and their advantages and disadvantages are discussed with regard to the project's system architecture. In some cases, ways of converting these to the current architecture are presented. The structure of this chapter first presents the system architecture more deeply, as it is crucial in deciding what are the advantages and disadvantages of the algorithms. Then a comprehensive list of interesting load balancing algorithms is discussed. Finally the chosen algorithms are presented and the reasons why they were chosen are explained.

3.1 System architecture

As explained in Chapter 2.1.1, the architecture that has already been setup in this project uses one queue at the Asynchronous Message Queue (AMQ) level[1]. However, load balancing algorithms have emerged in many different contexts, and therefore also consider many different architectural schemes. The very commonly found situation consists of a number of queues that dispatch their information to a single consumer acting as an Edge Server equivalent. The situation in this research is rather the opposite: there is a single information entry point where messages arrive. They then get get dispatched rapidly between the multiple treatment centers, the Edge Servers. In these two architecturally different environments, the dispatching of messages happens in the same place, but not on the same objects, as shown on figure 3.1. Arguably the first situation is closer to the task of "scheduling" rather than that of "load balancing", even if fundamentally it is still aiming a balancing a load. For clarity, the first situation will be referred to as *scheduling* within this document. The mentioned difference between *scheduling* and *load balancing* is depicted on figure 3.1.

This important structural specificity of the project's design means that some of the classical algorithms can not simply be applied in this case and need to be adapted (or the design changed). Quite often there is an easy way to flip-round the normal load balancing techniques in order to fit the current situation, but not always. At the queue manager level, messages are always treated in a FIFO[2] order. This means that some algorithms are just not possible.

[1] This will also be referred to as "Queue Manager" or "QM" further in this report.
[2] First In, First Out

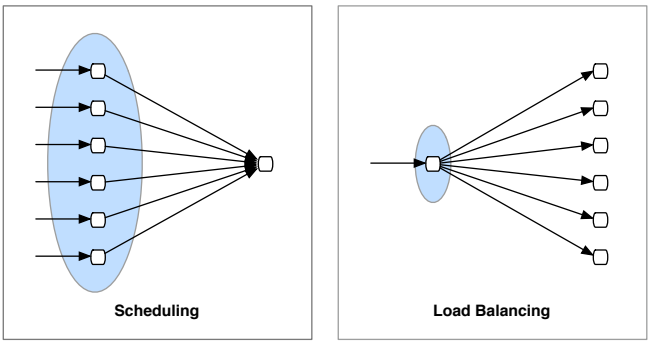

Figure 3.1: *Difference between scheduling and load balancing.*

3.2 Interesting algorithms

The following chapter presents a selection of interesting and popular algorithms. References in literature will not be proposed for each kind of algorithm. The reason is that many very global knowledge papers present in a very good way most kind of algorithms. Only when the algorithms become more technical and specific are papers cited.

3.2.1 Random

The random algorithm, as its name suggests, randomly selects the client to which to pass a message. Its advantages include its simple implementation, no feedback is needed (which is good for low power consumption), it is said to be good for homogeneous systems and it has no state information. However on the downside it is said to be bad for dealing with uneven packet sizes in a classical scheduling setup. In this load balancing architecture, this probably translates into being bad for unevenly loaded Edge Servers. This algorithm is also bad because it can not profit from the maximal capabilities of the clients. If these have different processing speeds, then some servers might finish processing faster than others, and then idle.

3.2.2 Round Robin

A round robin policy sends messages to each Edge Server, one after the other. This is done whether the client is running or down, overloaded or empty. Its main advantages are its simplicity and the fact that it does not need feedback. It also guarantees fairness and is said to be good for homogeneous systems. Its main disadvantage is that it is poor in case of uneven packet lengths as it gives priority to large ones, since they take up more server processing. Is also has the same problem as Random, since it does not have feedback, it can not profit from the maximal capabilities of the servers.

3.2.3 Credit Priority

This algorithm is also referred to as a "credit based", or "priority based" algorithm. The load balancer sends messages to the Edge Server with most buffer space. To be precise, this algorithm works in the following way. Each subscriber has a buffer and has a credit value that represents the space in the buffer. This credit parameter, representing the buffer occupation, is sent to the load balancer each time the buffer frees up again to a certain level (i.e. goes past a threshold). This operation is called an update, because it updates the local credit value that is stored on the load balancer. The later diminishes his local credit each time he sends a message. There is one local credit value per subscriber on the load balancer. Since the subscribers are processing messages one after the other, but only send an update of their credit status at a certain frequency, it means that the local credit values of the load balancer are not exact. The higher the amount of feedback, the better the correspondence between these two values will be.

This is the algorithm that is already implemented in the original architecture, which is the starting point of this research. It has the advantage that it works with feedback, and hence models the server occupation to a certain granularity. However this granularity is also a problem as it is not exact, and idling can still happen. Furthermore, feedback creates more data transmission and hence can diminish the system's performance. These points will be discussed later in this document.

3.2.4 Weighted Round Robin

With this algorithm, incoming requests are distributed in a round-robin fashion, weighted by some measure of the load, in this case of the Edge Server queues. Typically it would be possible to give more weight to certain less loaded servers, and hence have more messages sent to these. This algorithm should be easy to use with the current architecture, since there already is an indication of the load: the local credits. However, the algorithm can also work without feedback. In that case, it then has fixed weights for each consumers, set up upon startup. The positive aspects are that it should still, to some extent, evenly balance the load on all Edge Servers. The negative points come from the fact that it can need feedback. Another aspect could be negative: if the credits are an indication of the number of messages in the buffer, then the problem due to message size remains (This problem is described in 3.2.1). An good overview of this algorithm in the classical case is found in [Hah86].

3.2.5 Shortest Remaining Processing Time

This algorithm originally comes from pure realtime scheduling, where objects get partially processed. They then wait some more time before being processed further. The object with the shortest remaining processing time gets priority. In the architecture of this project, processing times are not known. Predicting processing time is difficult. It is not even linked to message size, since the latency that appears at the beginning of message processing is quite big. The interest in this policy comes from the fact that this algorithm is known, in classic situations, for "well balancing

the load" (for clustered web servers). However there is no direct indication that this would also be the case in this situation. An excellent overview of the topic can be found in [Decie].

3.2.6 Shortest Process Next

Shortest Process Next is a policy where the whole processing time is know, and the client with the shortest process gets all the priority until it has been dealt with. It has the disadvantage of creating a risk of starvation for long processes. In the current architecture, an equivalent would possibly be to send messages to the server whose buffer is going to be empty first, but again this is not something that can be known. Again, [Decie] is an excellent source for explanations.

3.2.7 Content Aware Policies

These policies analyze the type of data that arrives at them, and dispatch them according to type rules. It is unclear how these could work in this project. Often, these algorithms are interesting as servers can cache some information and gain time through the processing process. Projects where not all Edge Servers perform the same task, and profit from this caching, could be imagined. If one wanted to implement content aware policies, it would have to be through some filtering mechanism at the Queue Manager level. The disadvantage of such policies comes from the fact that the analysis of the content can take extra time, depending on the kind of messages, their encoding and other parameters. References can be found in [HNTer].

3.2.8 Locality Aware Policies

These policies send data to the nearest client for processing, and are specific to the architecture of the whole system. The interest comes from the idea that under certain circumstances, such as in case of a big network where sending data to the nearest clients in priority makes sense since delays are minimized. It also would reduce unnecessary network traffic and probably speedup the average total time, in comparison to a policy that solely looks at raw processing speed. Again, this is a policy that would need feedback. Furthermore it is unclear how the concept of "locality" could be defined (time, distance, packet loss, etc.).

3.2.9 Weighted Fair Queuing

Weighted Fair Queuing (often referred to as WFQ), can be seen as a combination of priority queuing (queues serviced with different levels of priority) and fair queuing (ensures that all queues get serviced, through a round robin policy, even if higher-priority queues are in demand of attention). WFQ ensures that queues do not starve for bandwidth, and that traffic gets predictable service. (Typically, this flow-based queuing algorithm would on the one hand schedule interactive traffic

to the front of the queue to reduce response time and on the other, fairly share the remaining bandwidth between the high bandwidth flows.). Information can be found in [DKS89]. The classical situation works as follows. There is a scheduler, implementing the WFQ discipline, which would remove requests from the buffers for execution in a round-robin fashion. The scheduler would associate a weight with each service class and the proportion of requests selected from each buffer would be a function of this weight. In the original architecture of the system, this idea would not be implementable.

3.2.10 Deficit Round Robin

This algorithm uses a fixed quantum that is employed to determine how many of the current packets gets send. It has the advantage of sending data in groups. When the communication's establishment takes time, then it is an interesting algorithm. In the current situation it is not of interest since the TCP connections are always active.

3.2.11 Pull

The pull algorithm is a very basic algorithm. It is decision driven by the client. When he wants to receive a message, he asks for it. It has the clear disadvantage that it needs a substantial amount of communications, but the advantage is that the load balancer has an exact picture of who is ready to process messages.

3.3 Retained algorithms

Between all these presented algorithms, there is one big division. Some balance the load without any information about the clients' state (feedback-less), and some do it with this kind of information (feedback). In the category of feedback-less algorithms, random and round robin are the most common algorithms. The main difference between random and round robin is that the later ensures fairness. Feedback-less algorithms have the advantage of having a minimal load of data being sent across the network, and a low computational power needed to implement the decision policy. They are therefore interesting in this architecture where power and communication can be a killer. These two algorithms were selected for the experiments based on these criteria.

Amongst the algorithms with feedback, many solutions exist. Two algorithms that are interesting are the 3.2.5 "Shortest Remaining Processing Time" and 3.2.6 "Shortest Process Next". Admittedly, processing times cannot be foreseen (it would actually be a challenging and interesting problem). Hence they were not chosen. But what these two algorithms try to do is send the messages to the clients that are working the fastest.

The implemented algorithm into the already existing architecture, 3.2.3 "Credit Priority ", actually tends to do exactly the same. By giving feedback when the

buffer occupation goes below a certain threshold, a client informs the load balancer that it has a short processing time left. It can therefore be suspected that the algorithm is already very good since both original algorithms are rated highly in general. For this reason, and in order to compare the current implementation with some further algorithms, this algorithm was selected.

The most significant problem with "Credit Priority" is that it has feedback, and hence augments the amount of communication, which can be a problem. In order to evaluate the impact of this increase in traffic, the pull algorithm was selected.

Content aware policies present some interesting aspects, but the complexity to implement them, combined with their disadvantages have lead to leaving these aside. An interesting idea, for future work, is described in section 7.3.1, "Future Work", based on these observations.

Finally 3.2.9 "Weighted Fair Queuing" is an algorithm that has become increasingly popular in recent times because of the guarantees it brings. Its complexity means it was not selected for this project. However an idea based on its way of functioning is presented in 7.3.1. No other algorithms were selected for the exploration space of this project.

4 Design Overview

The following chapter briefly presents the design challenges that were addressed. No information is given about the code that was written, since this is specific to the programming language used for the implementation. Only the abstract concepts are presented.

The main design challenge in this project was to modify the original implementation of the system, such as to be able to change in a transparent way from one algorithm to the other. Originally the credit-priority algorithm was implemented in a way that it could not even be considered to be an algorithm as such. It was simply a set of rules, directly encoded and embedded into the Message Queue Manager.

In order to reach the set goal, the design challenges were:

1. to extract the embedded set of rules (of the priority-based algorithm) and put them together, so as to form an "real" algorithmic entity.
2. to adapt the architecture in order to be able to add other algorithms, whilst keeping the architecture highly portable and easily expandable. Furthermore, the independence with respect to the underlying protocols had to be kept as well.
3. to modify the other system elements, for which the behavior depends on the chosen algorithm, such as to know what algorithm is running and behave accordingly.
4. to modify the subscribers and producers in order to carry out measurements, according to Chapter 2

As such, the first point was an easy task since the implementation was working originally. However the difficult task was to extract the algorithm parts and fitting them into an adequate structure, allowing the accomplishment of the next steps.

The second step was quite challenging. It meant differentiating between the elements (i.e. code) that were specific to an algorithm, and those that were not, but belonged to the system's fundamental architecture. Having identified these elements, an adequate structure had to be found in order to be able to switch from one algorithm to another in a single operation. Based on these observations, it was possible to select an adequate design pattern: a factory pattern [GHJV94] was chosen. This pattern helps to model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate. It is called a *Factory Pattern* since it is responsible for "Manufacturing" an object. The advantage of such a structure is that each algorithm can be instantiated individually, with its respective specific classes, and superfluousl rewriting of code is prevented since this

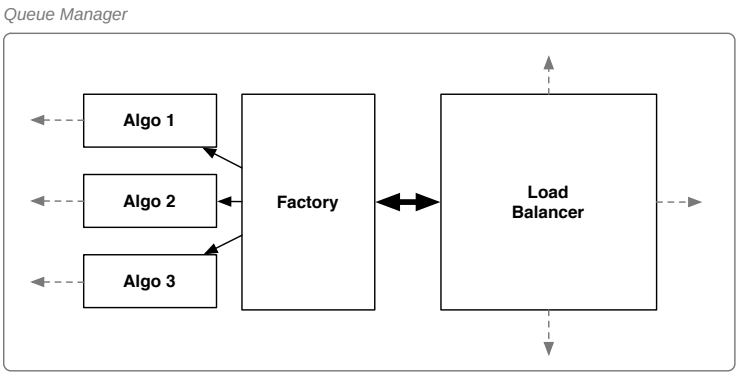

Figure 4.1: *Overview of the architecture of the system. The focus is on the load balancing and its effects.*

whole pattern "hooks" onto the original structure. This coupling is represented in figure 4.1. This figure must by no means be seen as the exact representation of what was implemented. However it gives an overview of the selected solution for an easy, centralized choosing of an algorithm for the load balancing.

The third step, modifying the other system elements, followed naturally from step two. The factory pattern was implemented for another element again, on the subscribers side this time.

The last step is aimed at precisely setting up the foreseen experiments. The previously implemented structure had been tested for other topics than load balancing. Therefore the subscriber and publisher code could not be re-used, and had to be rewritten in order to behave as wanted. The main challenge of these two elements of code was the memorization of time-stamps, whilst not perturbing the subscription, respectively publishing. This problem is explained in section 6.2.2.

5 Analytical Model

5.1 Overview

The current chapter presents an abstract model of the load balancing architecture and behavior, with the reflections and assumptions that are derived from it. Its pretension is not to be an exact theoretical model from which every aspect can be computed and foreseen. But it does give a good insight on how the system should behave and why, with regard to a set of defined parameters that will be made explicit later.

A model allows the prediction of the behavior of the new algorithms. It defines a theoretical value of the expected performance[1] in various situations and gives credit to the pertinence of the experiments. Ultimately it can confirm that some elements play a major role, others a minor one, in the system's performance.

5.2 From Reality to a Model

5.2.1 Defining the key elements

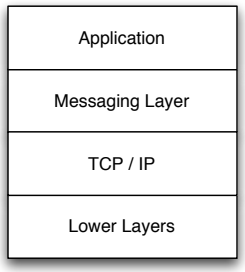

Figure 5.1: *Layers*

The path of a message that is sent from the Load Balancer to a client (or the other way) clearly goes through several layers, transits and comes back through other layers. Not all of them are of interest in this study. The abstraction will keep four layers, depicted in figure 5.1: application layer, messaging layer, TCP/IP and lower layers. These selected layers contain possibilities to slow down, block or even refuse a message. Hence they are potential candidates for influencing the performance of the load balancing. The name *Application Layer* is self explanatory. The *Messaging Layer* uses IBM's MQM protocol [mqt] to manage the connections. This layer deals with the decoupling of components, which in this experiment are the clients. It also sets up the asynchronous communication. This layer is application independent. This implemented protocol has a queue buffer associated with it, and therefore plays a major part in the efficiency of the load balancing. Lower down the layers, on the Transport layer, TCP is used. With such a connexion, a buffer is used to receive data from the network.

[1] "Performance" is explicitly left vague, since different elements can be selected to be measured.

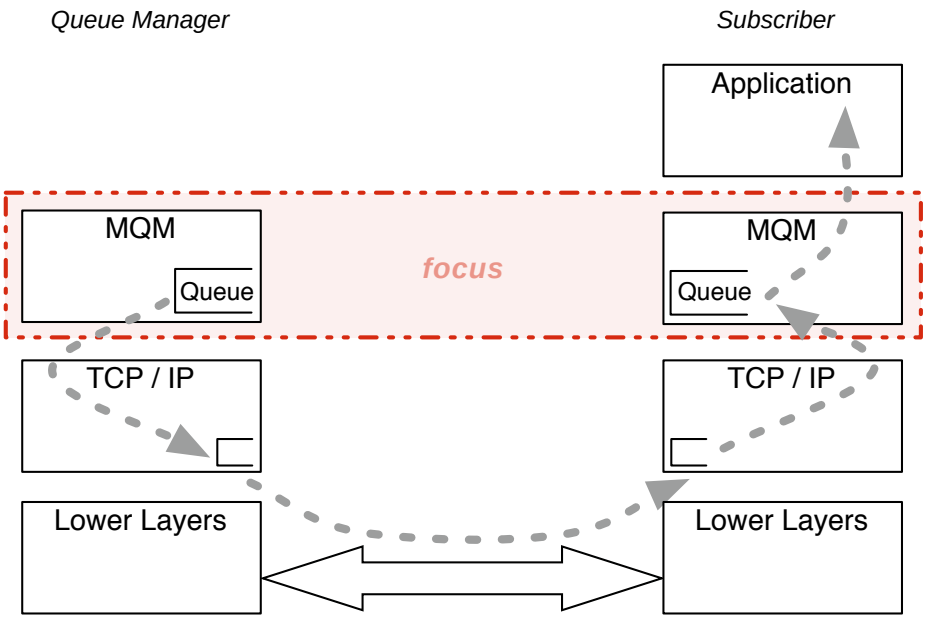

Figure 5.2: *Decomposition of the main layers for inter-client communication.*

This path between the Queue Manager and a Subscriber (client), though the different layers, is drawn on figure 5.2 by the dashed arrows. This "vertical" cut of the messages' path shows the three previously mentioned layers. At the MQM level, the messages depart from a queue of high capacity. At the TCP / IP level, they pass, on sender and receiver side, through a very small buffer that is used for making use of TCP/IP's sliding window advantages. This buffer is very small and is not of any interest for the behavior of the system since data is rapidly propagated further. The messages then go through the lower levels and get sent to the Subscriber, where they go back up the layers. Having arrived at the Messaging Layer, they are ready to be sent to the Application level.

Figure 5.3 is a "horizontal" extension of 5.2. First a message is created by some *producer* (typically a sensor for example) and is sent across any communication support to the *Queue Manager*. The Queue Manager manages multiple queues, and each Load Balancing entity has one assigned queue. Since the topic of this thesis is to study the efficiency of the latter, only one queue will be considered and is drawn (the queues are independent). The Queue Manager contains the load balancing entity marked *LB*. Messages get taken from the queue and are sent to subscribers, according to the algorithm's choice. The system uses a non-blocking architecture such that subscribers are served independently. This architecture makes use of Java NIO ([JNI] or [Hit02]). On figure 5.3, these extra queues are only lightly drawn, with the arrows going through them to indicate this extra feature. The *Edge Servers* receive the messages, and again store them in the queue of the Messaging Layer, where they get picked from to be processed when the *Application* is ready.

The buffers at the TCP/IP level only serve for the size of the sliding window protocol. They are very small, especially compared to the size of the queue used at the Messaging Layer. They can therefore be abstracted. As shown on figure 5.2, this leaves the focus for the model on the MQM level, where the load balancing is

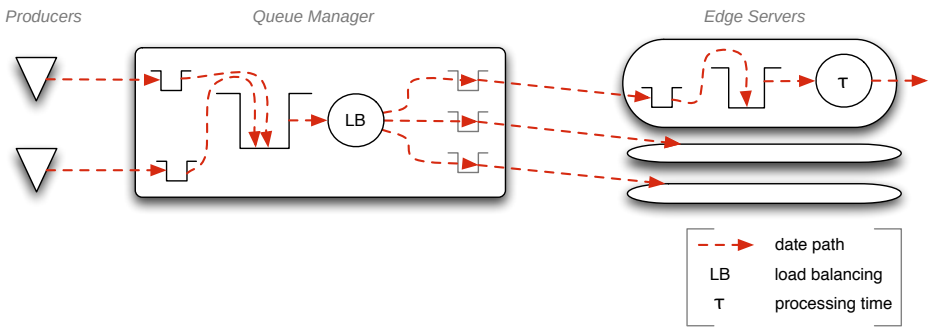

Figure 5.3: *Message path through the system.*

performed (on the Queue Manager) and where the messages are received (on the Subscribers).

5.2.2 Assumptions

Several elements are important in determining the time that a message takes to cross the system. The time is one of the measures of performance for the load balancing algorithms. *Delays* or *processing rates* are typical factors, since these might generate bottlenecks. It is because of the way the elements are linked that no loss of messages occurs. With the previous decomposition of key elements in the system, it is now possible to make some assumptions on which parts of the system are critical for keeping the global time as close to its optimum as possible. This is shown in figure 5.4 and can be decomposed as follows:

Producers The producers have a maximal rate at which they can send out information.

Data connection The communication means in the system (whatever they are) will have a certain bandwidth and link speed, creating delays and throughput rates.

MQM buffer The main buffer introduces delay.

Load Balancing entity The load balancer has a maximal rate at which it can extract messages from the main buffer, and decide to whom to send it. It subsequently introduces some delay.

Message processing The Edge Servers have a processing rate and insert delay into the messages' trip-time.

Given these observations, it is now possible to define a first basic model.

5.2.3 First sketch of a model

Before presenting a general model, the first sketches are outlined in order to gives a greater feel for what is expected to go on in the system. The final goal is to propose

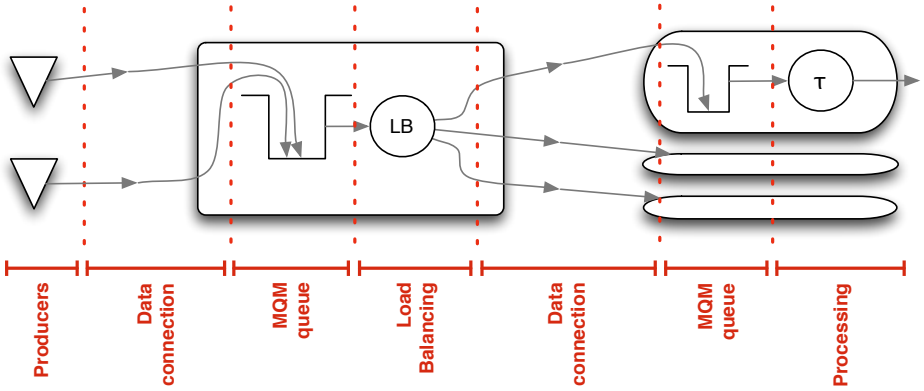

Figure 5.4: *Decomposing the message path*

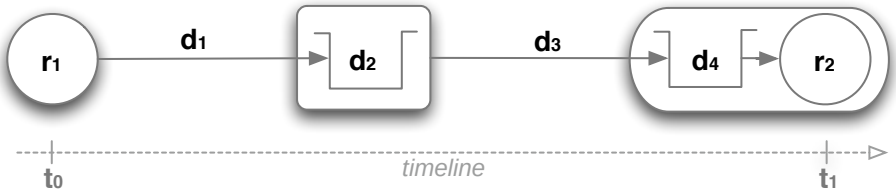

Figure 5.5: *Basic model between a client and a receiver.*

a model that allows to give a measure of performance of the various algorithms, in terms of the time they need to process a population of K messages.

The data connections can, as noted previously, introduce delay into the system and have a maximal rate at which they can transmit data. For a first sketch, the link throughput is abstracted as being infinite; delay is kept as an important factor.

An ideal load balancing entity, aiming to process messages as fast as possible, should not slow down the process because of a poor single processing rate or any form of delay producing a bottleneck. It must be as fast as all clients, whether senders or receivers. Therefore, in a first basic model, the processing rate of the LB is assumed infinite, and the LB can be removed from the model, in order to propose a first basic model. Furthermore, in a real implementation, the load balancer's decision takes orders of magnitude less time to compute compared to the transmission time of a message over the communication support. To keep it simple, an intuitive notation is given and a 1-to-1 model is presented in figure 5.5, with a single client and single receiver. The publisher has a publishing rate r_1 with a delay d_1 on its connection to the Queue Manager. There the queue inserts a delay d_2 before sending the message to a client. In this final place a delay d_4 is introduced before the final processing at rate r_2. Note that d_2 depends on r_2.

Based on this first 1-to-1 model, it is obvious that the global rate of the system is only dependent on the slowest element such that $r = \min(r_1, r_2)$ where r is the system's global processing rate. The relationship between t_0, the departure time of the first message, and t_1, the arrival time of the last message, is quite easy to write down. It is: $t_1 = t_0 + n/r + n\sum_{i=0}^{p} d_i$ where p is the number of delaying

elements in the architecture. In this case there is the first delay on the connection to the load balancer, the delay on the Queue Manager's queue, the delay on the communications between the LB and the servers and the delay from their respective buffer, so here $p = 4$. Such a model gives an excellent first idea of what is happening. However this first attempt has many deficiencies.

Wrong focus This project is only interested in the LB and what happens after, not before: the sending of the messages can be abstracted.

Inadequate terminology This first model is nothing more than a queuing network. However the terminology used is not the official and accepted one for queuing networks. Furthermore it is inadequate because it mixes different notions. In queuing theory, rate r_1 is called an arrival rate and is written λ. That which is here called r_2 is in reality a service rate, and should be written μ.

No multiple clients The model is a one-to-one representation, but the reality is more complex. By removing the LB from the first model, it is unclear how messages are distributed to the clients.

5.2.4 General model

The analytical model presented uses standard notation from queuing network theory. I will not try to go into any detailed presentation of queuing theory. Many people have done this before and much better than I ever could. Literature on queuing theory is vast. An excellent "general" coverage of the topic is [Jai91]. Main elements in queuing theory include for example: λ = mean arrival rate, s = service time per job, μ = mean service rate per server, n = number of jobs in system and w = waiting time.

The model presented in figure 5.6 corrects the aspects that the first sketch fails to model. The data arrives from the publishers at a rate λ. The LB has a mean service rate μ_{LB} and a service time per job of s_{LB}. Each k^{th} subscriber has an arrival rate λ_k, a service time per job s_k where $k \in 1 \dots i$.

In [DB78] an interesting method, called *operational analysis*, for analyzing queuing networks is presented. It defines A , B and C, as respectively being the number of arrivals occurring during the observation period T, the total amount of time during which the system is busy during T (such that $B \leq T$), and the number of completions occurring during the same observation period T. A and C are drawn in figure figure 5.6. It also defines $X = C/T$ as the output rate, $U = B/T$ as the utilization and $S = B/C$ as the mean service time per completed job.

As mentioned previously, the final goal is to propose a model that allows a measurement of performance of the various algorithms, in terms of the time they need to process a population of K messages. The term *utilization* is defined as the fraction of time that a system is busy. In [DB78] it is called U, but in standard queuing theory (as mentioned in [Jai91]) this term is referred to as *traffic intensity*, denoted by the symbol ρ. Within the observation period T, and based on the final goal, the following assumptions can be made:

$$\rho = \frac{\lambda}{\mu} \geq 1 \tag{5.1}$$

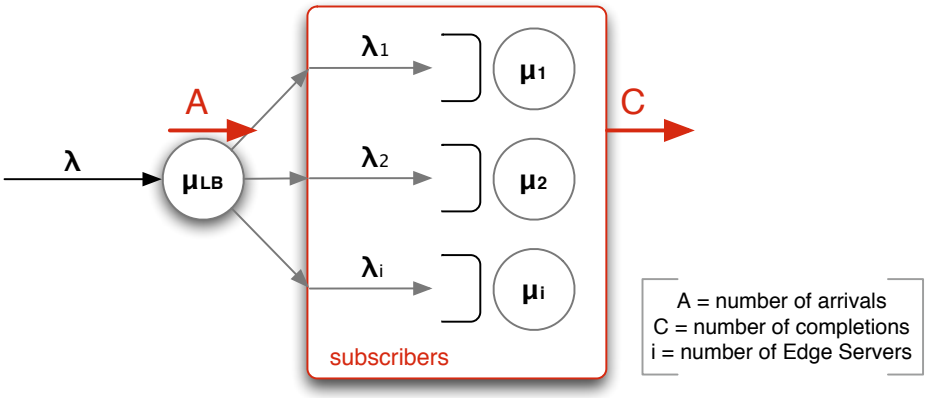

Figure 5.6: *General model for i Edge Servers.*

and it can be assumed that the sizes of the queues are infinite. The first assumption can be made because of the nature of the main goal. If utilization ρ is smaller than 1, even a suboptimal load balancer can achieve the same "performance" as an optimal one. Hence the interest of the model only exists for the case where $\rho \geq 1$. The service rate μ_{LB} of the Load Balancer gets split into the respective feeds for the subscribers, according to the algorithm's choice, such that $\mu_{LB} = \sum_{k=0}^{i} \mu_k$. Equation 5.1 not only applies globally, but also for each individual subscriber. If $\lambda_k \leq \mu_k$ for any one of the subscribers, one of them is spending some time idling and therefore the system is not optimal. This provides a optimality condition for the random algorithm:

$$\overline{\lambda} \leq \mu_k, \ (\forall k, k \in 1 \ldots i) \tag{5.2}$$

The observation period is considered in this project as long, since the aim is to measure the total time for a whole population K. It can therefore be assumed that the number of arrivals is equal to the number of completions during this time (this is not always the case, but makes sense for this experiment). The assumptions can be written:

$$A = C \tag{5.3}$$

It is called *job flow balance* because it implies $\lambda = X$. The time to process a load C of messages is determined by the slowest of the servers that acts as a bottleneck: $T = \max(C_i/\mu_i)$ where C_i is the number of messages from the i^{th} subscriber. The system's arrivals can be split up into the individual arrivals for each subscribers, such that $A = \sum A_i$. Similarly the sum of the individual completions is the system's completion: $C = \sum C_i$. Thanks to 5.3, it it can be said that $A_i = C_i$. Finally, the total time needed can be written as:

$$T = \max(A_i/\mu_i) \tag{5.4}$$

This is the model that will be used to foresee the behavior of the algorithms in specific cases. Finaly equation 5.3 says that $A = C$, which comes to saying that $\lambda = \mu$. Given that equality, a load balancer is then optimal if it guarantees:

$$\lambda_k = \mu_k \ \ (\forall k, k \in 1 \ldots i) \tag{5.5}$$

This equation is very important for the sections that follow, and will be called the "optimality condition".

5.3 Evaluation scenarios

In order to evaluate what is expected from the algorithms, a set of scenarios are defined. They are based on the key elements that are thought to be able to influence the system's performance. The elements are:

Processing variance The variance between the processing rates μ_i of the different subscribers is an important element. If subscriber k has a $\mu_k << \mu_{k+1}$, then both servers can not process as much data, and the system becomes unbalanced. When subscribers have different processing rates, it will be said that there is as *pattern* and will be referred to as "with or without pattern".

Delay Delay in the system can slow down the whole processing.

Bandwidth The limitations in bandwidth can create an extra bottleneck.

Based on these three kinds of influential factors, five scenarios have been defined. The bandwidth problem is a general problem and is only compared in scenario 1 and 5. When there is a bandwidth limitation, it is considered to be the same for all subscribers.

1. no pattern - no delay - unlimited bandwidth.
2. pattern - no delay - unlimited bandwidth.
3. no pattern - delay - unlimited bandwidth.
4. pattern - delay - unlimited bandwidth.
5. no pattern - no delay - limited bandwidth.

5.4 Expectations for each algorithm

The following section focusses on what behaviors can be expected from the chosen algorithms, with respect to the model presented in section 5.2.4 and this for each scenario described in 5.3.

5.4.1 Scenario 1

Scenario 1 is characterized by no pattern - no delay - unlimited bandwidth. In this situation all subscribers have an identical mean service rate μ_i, which means that for all algorithms:

$$\mu_k = \overline{\mu}, \quad (\forall k, k \in 1 \ldots i) \tag{5.6}$$

Random

A random algorithm is such that when observed over a long period of time, it can be considered as evenly distributing data such that the arrival rate of the k^{th} subscriber is $\lambda_k = \lambda_1 = \ldots = \lambda_i = \overline{\lambda}$. It can be written:

$$\lim_{t \to \infty} \lambda_k(t) = \overline{\lambda} \tag{5.7}$$

As said in assumption 5.1 of the model, $\lambda \geq \mu$. Therefore based on the equations above, in this scenario random is expected to be optimal since λ_k and μ_k will be in accordance.

Round Robin

The round robin algorithm ensures fairness by equally distributing messages, in the same order, to each subscriber. It is therefore very similar to the random case, except that it is immediately fair (no need for $t \to \infty$). For $k \in 1 \ldots i$, it can be written that:

$$\lambda_k = \overline{\lambda} \tag{5.8}$$

Again this means that round robin, is expected to be optimal, according to the general model proposed.

Credit Based

The credit based algorithm will periodically make sure that each subscriber's arrival rate is in accordance with its service rate. This can be mathematically written as $\lambda_k = \mu_k - \beta$ where β is a granularity term. It is subtracted since it models the loss of precision in the λ-μ equality. Indeed, each feedback returns the value of the subscriber's current load, but this is only done at a certain frequency, hence the β term. The higher the threshold for credit (i.e. the more updates are made), the smaller β gets. β can be considered as a function of the threshold φ and delay δ. In this scenario there is no delay so it solely depends on the threshold. Since all μ_k are the same, and stay the same, the level of adequacy does not evolve throughout a run, and only depends on the threshold. If the latter is above a certain value, that ensures that the queue never runs out of messages, then $\beta(\varphi)$ is null, and the credit based algorithm is optimal.

Pull

The pull algorithm, because this scenario has no delays, will make sure that arrival and processing rates are exactly equal. Thanks to equation 5.6, it can be written that for all $k \in 1 \ldots i$:

$$\lambda_k = \mu_k = \overline{\mu} \tag{5.9}$$

5.4.2 Scenario 2

The characteristics of scenario 2 are pattern - no delay - unlimited bandwidth. In this situation the subscribers have different mean service rates from each other. This can be mathematically defined as, for all j and $k \in 1 \dots i$:

$$\mu_j \neq \mu_k \ \ (j \neq k) \tag{5.10}$$

Random

The assumptions made about the general model and the load balancer's performance lead to the equation $\lambda = \sum_{k=1}^{i} \lambda_k$. Therefore a random policy, as $T \to \infty$, ends up evenly distributing data, $\lambda = k\overline{\lambda}$. Hence each subscriber is expected to have an average arrival rate of $\lambda_k = \lambda/i$. Unfortunately, because in this scenario $\mu_j \neq \mu_k$ (5.10), this implies that the optimality condition 5.5, won't hold, and means that random will be suboptimal.

Round Robin

The round robin algorithm has the same problem as the Random in this scenario: because it evenly distributes the messages, $\lambda_k = \overline{\lambda}/i$. Now because of the equation imposed by the setup of this scenario, 5.10, the optimality condition 5.5 can not hold and hence the algorithm is not optimal. There is one difference between Random and Round Robin. The second always distributes messages in the same order. This means that fairness is assured. With Random, any subscriber can receive multiple messages in a row. It is only on average that fairness is approximated. In both cases, when fairness is assured, the time to complete is:

$$T = \max_i (A_i S_i) = \frac{A}{i} * \max_i (\frac{1}{\mu_i}) \tag{5.11}$$

Credit Based

The credit based algorithm behaves very similarly as it did in the previous scenario. With or without pattern, as long as the threshold is high enough to ensure that new messages arrive in the queue before the old ones have finished being processed, then the algorithm can get close to the optimum. Indeed with adequate threshold, the β term is null and although this time the μ_k are all different, $\lambda_k = \mu_k$ is still true.

However, in this scenario the buffer size plays an important role: it determines the granularity precision of the optimum. When all service rates are the same, this plays no role. But when they are different, the time to process a full buffer of data is different depending on the different rates. When all messages have been dispatched, not all buffers will be in the same state: some full, some nearly empty, some half full, etc. If a slow subscriber has just had his buffer filled as the last message gets distributed, it will be slow to process its load of message. In the meantime, other

faster subscribers might just have finished processing their last buffer, but because there are no more messages available, they will just idle until the end. This is what is called "granularity" of this algorithm.

The total time is in fact the optimal total time and the extra time to finish processing the buffer of the slowest subscribers: $T = T_{opt} + \nabla T$, where T_{opt} is the optimal total time, and ∇T the extra time induced by the slowest subscriber. It can mathematically be defined as:

$$\nabla T = \frac{n_j}{\mu_j} - \frac{n_k}{\mu_k} \tag{5.12}$$

where n_i are the number of messages that can be contained in the buffer of the i^{th} subscriber, μ_j is the slowest service rate of all subscribers, and μ_k the fastest.

Pull

Again, as long as no delays exist, the pull algorithm will make sure that arrival and processing rates are exactly equal, and hence be optimal. This is why, although the algorithm adapts to the pattern, it has a certain granularity linked to the variance of the service rates. The longer the runs is, the smaller the time lost by the granularity is, in comparison to the total time.

5.4.3 Scenario 3

Scenario 3 is characterized by no pattern - delay - unlimited bandwidth. In this situation messages are penalized by a delay d between the load balancer and the destination subscriber. The subscribers' service rates are all identical again.

Random

As noted above, a delay changes the time for a message to get from the LB to a subscriber. However it does not change the rate at which messages are sent. Therefore just as in scenario 1, Random is expected to stay optimal. The total time however changes. Assumption 5.1 on the utilization means that the additional delay must only be counted for once. The total time is expected to be here:

$$T = \max(A_i/\mu_i) + d \tag{5.13}$$

Round Robin

There is no reason for this algorithm, based on the general model, to behave differently than Random in this situation.

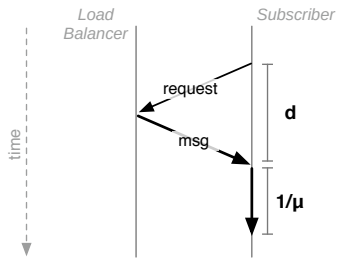

Figure 5.7: *Round Trip Time of a pull request and its processing.*

Credit Based

Delay affects this algorithm because feedback takes time to arrive, hence making it (the feedback) imprecise, such that:

$$\lambda_k \neq \mu_k \tag{5.14}$$

When a credit update gets sent it is still precise, such that $\lambda_k = \mu_k$ but the delay must be taken into account and gives: $\lambda_k = \mu_k - \beta(\delta)$. As δ grows, β grows too, making λ_k always smaller. Said otherwise, the bigger the delay gets, the less precise the feedback gets, and at one point, if condition 5.1 does not hold anymore, that is if $\lambda_k < \mu_k$, then the algorithm becomes sub-optimal.

Pull

Since the pull algorithm sends a request for each message to be received, it is going to be strongly influenced by delays. Figure 5.7 illustrates what happens; d is defined as the round trip time for asking for and receiving a message[2]. Once received it is processed for a time $1/\mu$. The inter-arrival time τ between two requests can be expressed as: $\tau = d + 1/\mu$. Remembering that $\lambda = 1/\tau$, the arrival rate for the k^{th} subscriber can be written as:

$$\lambda_k = \frac{1}{d + 1/\mu_k} \tag{5.15}$$

The expression clearly shows the more the delay grows, the smaller λ_k becomes. Since $\lambda_k < \mu_k$, the utilization is smaller than one. Hence the algorithm is suboptimal.

5.4.4 Scenario 4

Scenario 4 is characterized as pattern - delay - unlimited bandwidth. In this situation the subscribers have different mean service rates μ_i amongst each other, and there is delay.

[2] d could obviously be defined as the sum of two different delays, since the incoming delay need not be the same as the outgoing.

Random and Round Robin

As seen in the previous scenarios, neither Random nor Round Robin can adapt to a pattern of different service rates. However they are not influenced by any delay, if not for the additional time to send the first message. Indeed, as in scenario 2, the first sent messages takes longer to go from the LB to the server, but the next messages are sent as fast as possible. The processing time for the first messages is greater than the time between two sent messages. This scenario expects to see them both behaving in the same way.

Credit Based

Scenarios 1 and 2 show how a credit based algorithm behaves the same way with or without a pattern of processing times. However scenario 3 demonstrates that the delay is the main drawback for this policy. This scenario should therefore be similar to scenario 3, and should see this algorithm to be sub-optimal.

Pull

As seen previously, the pull algorithm will exactly adapt to the pattern. However, it was also shown previously that Pull is expected to have its main problems from delays. It will therefore behave exactly as in the previous scenario.

5.4.5 Scenario 5

Scenario 5 is characterized by no pattern - no delay - limited bandwidth. In this situation the communication supports have a limited bandwidth. It is important to observe that such a limitation can force all algorithms to be sub-optimal. Indeed, if the bandwidth limitation is too high, then it will not be possible to have a sufficiently large arrival rate in order that equation 5.1 can be possible. For this reason, the exposure of the scenario to each algorithm will only attempt to show which elements are influenced by this bandwidth limitation.

Random and Round Robin

As in all previous scenarios, Random and Round Robin are expected to have $\lambda_k = \overline{\lambda}$. The main difference is that $\overline{\lambda}$ will be smaller than in the previous examples, because of the bandwidth limitation, as mentioned in the introduction of this fifth scenario.

Credit Based and Pull

It is difficult to quantize the loss due to bandwidth, based on the model. A bandwidth limitation penalizes the number of data transfers. The same kind of

mathematical representation as that used previously can be written here such that $\lambda_k = \mu_k - \beta(\omega\varphi)$, where β is some granularity issue, undefined, that is suspected to depend on ω and φ. The value ω is the bandwidth limitation[3]. This equation expresses that the individual arrival rates diminish as a factor of the bandwidth limitation and the amount of feedback (which directly depends on the threshold).

When the communication support is *unidirectional* i.e. only supports one message transiting at once, then a credit-based algorithm will be penalized. Since the amount of feedback is higher than for a feedback-less algorithm, the bandwidth limit will be reached faster and less messages will be able to be sent. More importantly, the pull algorithm has a feedback call for every message and will be even more diminished by any bandwidth limitation. If the data channel is *bidirectional*, both algorithms are not expected to suffer more than Random or Round Robin. This is true because the feedback will be requiring less capacity on the backwards channel than the sending of the messages. Indeed this is true because feedback messages are smaller in size than data messages. In this case, bandwidth limitations only add to the delay since transmission time increases.

NOTICE: in all the preceding scenarios, it has been discussed whether such or such algorithms can be optimal or not based on certain assumptions. The assumption that there is no delay allows the pull algorithm to be optimal in certain cases. However it is important to understand that the "no delay" is never possible in real experiments. This means that the pull algorithm will always be sub-optimal to the others that are also supposed to be optimal in a specific setting. Indeed each "pull" for the next message will oblige the server to idle the time of the delay, however small it is, thus making the algorithm suboptimal.

[3]No units are mentioned since this is an abstract model

6 Experiments

This chapter describes the experiments that were performed in order to compare the four chosen algorithm. First the configuration and setup of these system tests are described. Afterwards the results are presented and immediately analyzed.

6.1 Test cases

As explained by Jean-Yves Le Boudec in his lecture notes ([Bou04]) on Performance Evaluation, it is necessary to have thought about the outcome of the experiments prior to running them: defining the *goal* of the experiments first, defining a *metric* and a load *model* are his recommended first steps that require a good understanding of how the system functions. The *factors* that can influence the system must be thought of; *bottlenecks* must be known, as they will be a key to the system's performance. This motivated the choices for the experiment scenarios.

Goals The main goal of these experiments is to compare the efficiency of multiple algorithms. The focus will be on both understanding and classifying the effects of multiple parameters on the total amount of time that a group of subscribers needs to process a batch of messages. These parameters include elements like bandwidth limitation or network delay.

Metric The main metric for measuring the system's performance is the time needed to process the number of messages sent by the system, per time unit.

Model See Chapter 5 for the proposed model.

Bottlenecks See Chapter 5 for considerations about bottlenecks in the system.

Factors The factors that play a role in this experiment are numerous. The the list of parameters selected for the experiments is given below. They were chosen thanks to the model's predictions and the main goal that the measures want to highlight. The choice of these parameters is discussed in section 6.3 in the details of each individual experiment:

- Distribution of the processing times
- Network delay
- Network bandwidth limitation
- Number of subscribers
- Number of publishers
- Number of messages published
- Processing time of the subscribers

- Time between two sent messages

The number of times a experiment is run is also a parameter (but not a what is considered here as a "factor") in the experiment. It will be further referenced as "runs". This parameter does not influence the results but changes the confidence in the measured values.

The processing times are not real, but emulated. Upon reception of a message the subscriber sleeps for the time of the desired processing. Two different processing-time distributions were setup: *constant* and *exponential* in order to cover multiple real-life situations. Their differences are further explained in section 6.3. A more global distribution, between the different subscribers, was also set up and is called a pattern. This *pattern*, as defined in the scenarios of part 5.3, across the receivers, is either used or not. A difference is made between the setup of different processing times across the subscribers, which is called a pattern, and the evolution of the processing time of a single subscriber over time, which is called a distribution. The possible combination of these four elements (*pattern* or *no pattern*, *constant* or *exponential distribuition*) are supposed to highlight the behavior of different parts of the architecture. Each experiment is run with the constant and exponential distributions, for direct comparison.

6.2 Experiment setup

6.2.1 Machines

The experiments were ran on an IBM eServer BladeCenter, with Gigabit ethernet. Each blade had two Network Interface Controllers: eth0 and eth1, and were bi-processor. Two machines were used for simulating the whole Sensors-QM-EdgeServer architecture, and a third one was used for installing the network emulator. In order to facilitate further reference to these machines, these blades will be referred to as *Opera* and *Optimus* from now on. The third machine, configured for controlling and modifying network behavior (as explained in subsection 6.2.3) will be called *Organum*.

The first is called *Opera* because the latin meaning of the word "opera" is "to work hard". This name is given to the first machine since it must work hard to publish and receive so many messages, as explained in subsection 6.2.2. The second machine is called *Optimus*. The latin meaning of "optimus" is "one of the best". The word *optimal* originates from it. This name was chosen for the machine where the load balancer is, since the latter must be as good as possible, or even optimal. Finally the third machine is called *Organum* which means "organ" in latin. This machine works as a router (see explanations in subsection 6.2.3). The name was chosen because a router "pipes" data. The exact behaviors of these three machines are explained in the following sections.

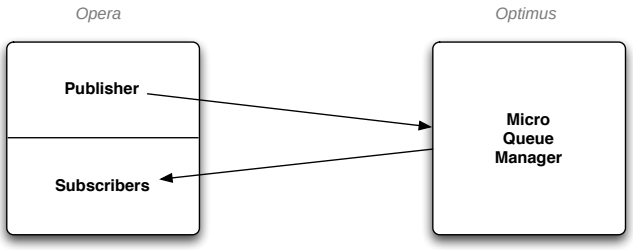

Figure 6.1: *Setup of the blades with the publisher and subscriber on one machine, and the MQM on another.*

6.2.2 Setting up the experiment

Since the main topic in this research is the time that the algorithms take to process a fixed number of messages, time-stamps have to be made. However, measuring a time difference based on two time-stamps from different machines, is not precise since the machines' clocks need to be perfectly synchronized, a difficult task. The precision of a time-stamp is of one millisecond. In order to guarantee the precision of the time measures, it was decided to implement the timing mechanism on the same machine. Hence the publisher and subscribers were installed on the same machine, *Opera*. The other entity needed in this experiment, the MicroQueueManager (MQM), was setup on another machine *Optimus* (no timing is needed between either a subscriber and the MQM, or a publisher and the MQM). This setup is shown in figure 6.1 where the publisher and subscribers are on one machine, and the MQM on another. The act of publishing and receiving messages is quite light. The installation of a publisher process and several subscriber processes on the same machine should not slow anything down, especially as the blades are bi-processor. The time measures were made upon reception of each messages by each subscriber. Making a time measure takes no time at all. However storing this measured value can be more of a problem, especially if it must be stored on disk. Each storage operation takes time and holds back the reception of messages. In order to be sure that nothing is slowing down the reception process, the messages are stored in memory (in a buffered writer) and only written to a file (i.e. to disk) once all messages have been received.

To make sure that the experiments can be repeated, and in order to ensure precisely the same configuration, the whole experiment was launched through a single script. All parameters were set up in a single configuration file. The messages that are sent are very small in size (around 100 bytes).

6.2.3 Network emulator

In order to be able to control network behavior easily, a network emulation tool was used. There was no hardware controller available for the experiment. A software solution therefore chosen: *NIST Net*. The NIST Net is a network emulator for emulating performance dynamics in IP networks, that has been developed by the U.S. National Institute of Standards and Technology (NIST). References can be found through the NIST Net website [NIS] and in the technical paper [CS04]. A

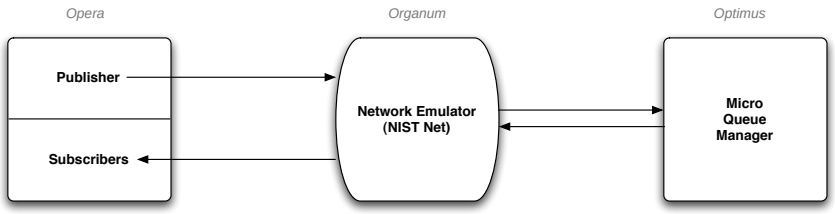

Figure 6.2: *Setup of the blades with the publisher and subscriber on one machine, the network emulator in the middle and the MQM on the last machine.*

high number of university researches have been conducted on and with NIST Net. The feedback is very good and gives this tool a lot of credit. There is therefore no reason to believe that this could be a problem to use this solution rather than a hardware one. NIST Net is implemented as a kernel module extension for Linux.

The authors of [CS04] describe this tool as a "network-in-a-box". It actually behaves as special router which emulates a whole network in one single point, hence being able to simulate such diverse parameters as packet delay, network throughput limitations, loss, jitter, and more. Its working model consists of a source and a destination node, between which rules can be applied. It does not act as a network controller for the machine on which it is run. The original setup in the experiments was to have the network emulator on the same machine as the MQM but this did not work. NIST Net really acts as a router. For this reason, a third machine had to be used. This machine is called *Organum*, as mentioned previously. Figure 6.2 presents the setup from a general point of vue.

In the experiments of this project, the two main parameters that need to be varied, in order to observe the behavior of the algorithms, are *delay* and *bandwidth*. These two parameters were chosen in Chapter 5 because of their possible influence on the feedback messages. Therefore they are expected to influence only the returning communications between the MicroQM and the Subscribers. Furthermore, as explained in the model of Chapter 5, the focus is on the load balancing efficiency, not on the Publisher-to-MQ link performances. Hence only the returning communications need to be controlled. For this reason, the network emulator needs to work between *Optimus* and *Opera*. The chosen configuration is described in figure 6.3

This setup takes into account a few constraints that are described hereafter. In order to be able to login remotely, and because the user files are not stored on the machines, *Opera* and *Optimus* have to stay connected to the main network, which will be called "Lab" network further in this paper. Another constraint is the fact that NIST Net has to be set up as a router, as explained earlier. How can this be set up? As mentioned in 6.2.1, each blade has two network interfaces. The BladeCenter interconnects all machines through two main network interfaces. In order to be able to control the returning communications between *Opera* and *Optimus*, there must only be one possible path between the two, that goes through the network emulator on *Organum*. The implemented solution is to have the first network interface eth0 of *Opera* connected through the first channel into *Organum*:eth0. *Organum*:eth1 is connected through to the second channel, where *Optimus*:eth1 is also connected. *Organum* is set up to forward all requests arriving on eth0, directed to 10.10.10.0, onto eth1, and also forwards all requests arriving on eth1, directed to

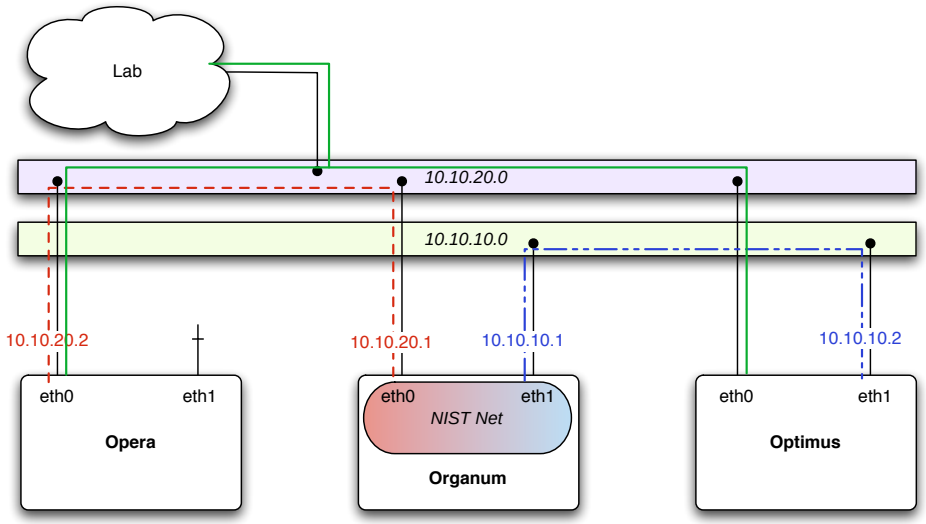

Figure 6.3: *Architecture for using the NistNET network emulator.*

10.10.20.0, onto eth0. With such a setup, when *Opera* tries to send a message to *Optimus*, on 10.10.10.2, the message goes through the blades first network interface and arrives through eth0 on *Organum*. Since the message is destined to a message on the 10.10.10.0 address, *Organum* forwards it through its second interface, eth1. The message then goes through the BladeCenter's second network interface before arriving at *Optimus*. The messages going the other way take the same path but in reverse order.

Opera:eth1 est purposely *not* connected to BladeCenter's 10.10.10.0 interface, so that messages are forced to go through NIST Net. If it were connected would there would be no reason why a message sent between *Optimus* (10.10.10.2) and *Opera* (10.10.20.2) would go through *Organum*, since a shorter way would be possible through *Opera*:eth1. When experiments must be run without network emulation, this can be done through the BladeCenter's first network interface 10.10.20.0, as the full-line path shows on the figure (6.3). Thanks to this architecture, the experiments can be efficiently controlled, and repeated multiple times.

6.3 Scenarios and Results

This chapter presents the scenarios used for the experiments and the results.

Just like in the analytical model of Chapter 5, a set of scenarios is used for the experiments. The scenarios are basically the same as in Chapter 5, but for the addition of one extra parameter. In Chapter 5, subscribers either all have the same service rate, or a pattern is set. However in both cases, the situation does not change: processing times are fixed across time. If the first subscriber is twice as fast as the second, the situation stays such throughout the whole experiment run. This is not something that is guaranteed in real experiments and needs to be added as an extra parameter that can influence the performance of load balancing algorithms.

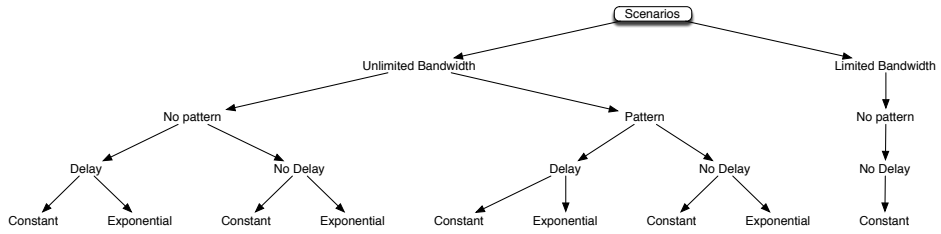

Figure 6.4: *Set of parameters selected for the experiments.*

In some experiments, it can be perfectly reasonnable to expect that the so called "processing" is always the same and that hence the time required to treat the data is always the same. Hence, it was decided to use a constant distribution, also called constant distribution, to model the evolution of service rates of each subscriber. Obviously in many experiments there is no constance and this case just does not happen. The processing times keep changing and evolving. In order to model this behavior, an exponential distribution was chosen. The exponential distribution is used extensively in queuing models because it is the only continuous distribution[1] with the memoryless property such that the time since the last event does not help in predicting the time till the next event. In reality, there is often no reason to believe that processing times are always constant, nor that there is a link between the previous and current processing time.

Therefore two different processing-time distributions were set up: *constant* and *exponential* in order to cover multiple real-life situations. In the first case all subscribers always have exactly the same processing time for each message. In the second, the simulated processing times are modeled as exponentially distributed, with the same average value as the constant distribution (this makes the two models comparable).

The experiments start by focusing on a single factor in the system, and then grow to more complex arrangements. The set of parameters, described in section 5.3, combined with the distributions mentioned above, give a broad set of possible scenarios. Figure 6.4 presents the set of those that were kept for running experiments.

Measurements

For each of the scenarios that follow, two measurements are plotted on the graphs. The first is called *Total time* and the second is called *Average time*. How they are calculated is described hereafter.

For every run of an experiment, each of the six subscribers has a time-stamp for each received message. The difference between the last and first time-stamp gives an individual total time for each subscriber. The average of these six totals is called the "average time of one run". With five runs, for example, the average of these "average time of one run" is computed and that is what is called the *Average time*. The earliest and latest time-stamps of all the six subscribers together are compared

[1] It is the only one, according to [Jai91]

to determine the total time that they altogether needed to process all messages, and this is the "individual total time of one run". Each run gives this "individual total time of one run", and its average is what is called the *Total time*.

The *Total time* is a direct measure of an algorithm's performance, since it measures the total amount of time that a combination of subscribers needed to process a certain load of messages. The *Average time* is more a measure of the variation across the subscribers.

As said, the *Total time* is an immediate indicator of performance. However it is more sensitive than the *Average time* since typically variations upon startup are recorded and influence the result. Furthermore, although subscribers might each need the same amount of time to process a fixed load of message, if they don't start processing at the same time, the total time needed by all of them will be higher than the individual totals. The average will not highlight this difference. Therefore, although the measure is very direct when looking at the totals, the average is a softer value that still indicates whether the algorithms are fundamentally behaving differently. If a real case scenario needs to have performant algorithms immediately after the startup, then the *Total time* is a more relevant indicator of performance. But if what is important is what happens when the system is already running, then the *Average time* is more relevant.

6.3.1 Optimal situation

The first scenario is called "optimal" because many parameters are configured such that they least influence the experiments. Unless mentioned explicitly, the bandwidth is assumed *not* to be a problem and only physically limited by the double Gigabit ethernet connections of the setup, which is much more than needed. The parameters for this first experiment are listed in table 6.1. The number of subscribers is fixed to six by choice. More subscribers could be launched since the TCP connections would hold (within reason). However, increasing the number of subscribers would require an increased rate of publishing messages and more publishers would have to be launched. Given the physical setup of the experiment, this would require adding machines and would make the measurements more complicated. Six consumers is enough to observe how the load is distributed and makes sure that there is enough processing power for each subscribers to work without being biased.

Table 6.1: *Parameters for the "Optimal situation" scenario.*

Parameters	*Value*
Nb Subscribers	6
Processing time	60ms
Nb Publishers	1
Send rate	as fast as possible
Delay	-
Nb of msg published	25'000

This first experiment brings out the fact that all algorithms are close to their optimal. The graph on figure 6.5 shows the results. Be aware that the time scale on it

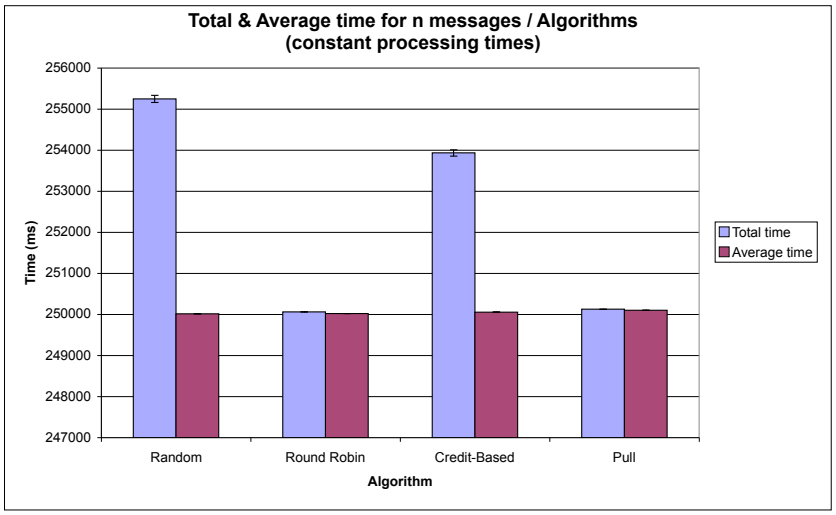

Figure 6.5: *Total and average time for processing n messages / Algorithm (constant).*

has a suppressed zero. This was done in order to be able to see the small differences between the algorithms. Many other graphs that follow also have suppressed zeros. When the scale starts from zero, all algorithms are indistinguishably fast. Error bars are drawn and represent the 95% confidence interval.

On this graph, the error bars are so small they are hardly visible. They underline the high stability of the experiment. The total times are all within 3% of each other. Round Robin and Pull are indistinguishably close. Random and Priority are both not as good, with respectively five and four seconds difference. This is explainable for the random algorithm. Indeed, although on average, a random selection gives the same number of messages to every subscriber, some might receive messages multiple times in a row, whereas other have not received anything for a few runs. This means that at some moments, some subscribers might have finished processing their buffer before receiving new messages, therefore idling and explaining why this algorithm takes more time. The difference for the priority algorithm is much harder to understand. Indeed it is expected, based on the model, to be an optimal algorithm, but it clearly is not as fast as the pull (which seems optimal). This is even more surprising under this configuration where delays are insignificant. Notice that on average, all four algorithms are extremely close (within 0.04%) which means that all the subscribers of a run take the same amount of time to process their messages.

The same experiment with the exponential distribution of processing times highlights even more differences, as shown on figure 6.6. The confidence interval bars are again extremely small. First of all the averages of the four algorithms are not the same, although they are within 0.5% of each other. This most certainly comes from the exponential distribution itself, which is implemented with the same mean, but has this *memoryless* property. By nature, the same exponential distribution

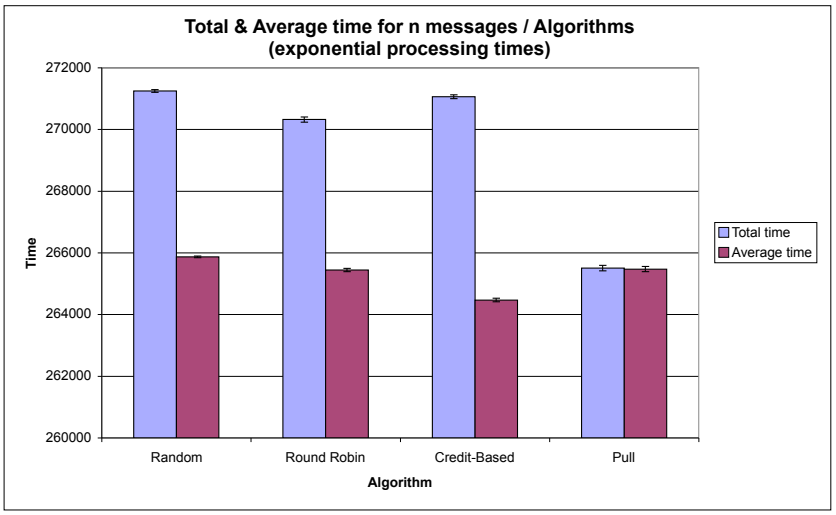

Figure 6.6: *Total and average time for processing n messages / Algorithm (exponential).*

can not be recreated twice in a row creating this variance in the average times. One observation is strange: the average of these average times is 6% higher than that in the constant case. This should not be the case since the mean of the exponential distribution uses what is defined as processing time in the constant case, as its mean for the distribution. However, after checking the simulated processing times, it appears that these 6% come from the imprecision of the implementation of the exponential rate, and not because of idling from the subscribers.

Random and RoundRobin both are not so close to their respective averages. Random is clearly having some idling going on, probably because of not only the randomness itself, but also because the processing times are exponentially distributed. The two effects do not seem to be adding themselves, since in the constant case, the total time was 2.1% off the average, and here it is 2.0% off it. The RoundRobin is clearly "suffering" from this exponentiality as it is 1.9% slower on total time than its average. Because of the nature of the algorithm, this must not come from any idling. Every subscriber receives the same number of messages. The only explanation for this difference is in the randomness of the processing times generated by the distribution. The pull algorithm works very well in this scenario. Its total time is less than 0.1% slower than its average. The priority algorithm is, once again, not doing very well, although it is expected to, according to the model. When running the experiments, it was noticed that the subscribers did not finish so closely in time, although they should be relatively close. The term relatively is used because the granularity of the buffer and threshold comes into account. This observation was in accordance with the expected effects of using a large buffer. A new experiment was launched with a diminished size of buffer. In the first case a big $128 * 1024$ bits buffer was used. For the new experiment a small $2 * 1024$ bits buffer was set up.[2]

[2]Further in the experiments, these two buffers will be called "big" and "small" buffers.

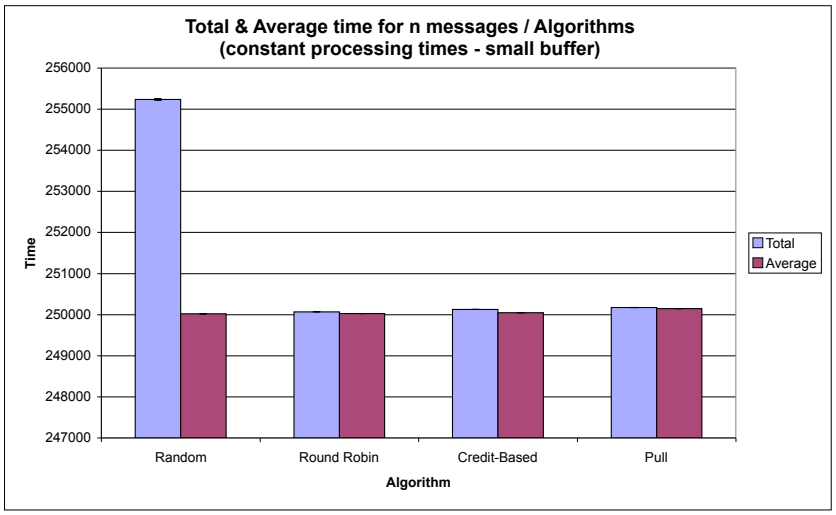

Figure 6.7: *Total and average time for processing n messages / Algorithm (constant - small buffer).*

Refining with a smaller buffer

First come the constant processing times. The error bars are extremely small and hardly visible on the graph 6.7. When repeating the same experiments with a smaller buffer, the results change in the expected way, as shown in the graph. The random algorithm still has some idling going on, but this is normal. Changing the buffer size should not influence this, nor should it influence the round robin's performance. Both algorithms behave in the same way as in the previous case with a big buffer. The pull algorithm also behaves just as precisely. There is no noticeable difference. And finally, the credit based algorithm is behaving as foreseen by the model. By reducing the size of the buffer, the granularity problem was treated, and the algorithm is much finer here. Its total is less that 0.01% off its average and the total time of the round robin.

It is difficult to see from the graphs, but the pull algorithm is in fact slower than the other algorithms. This is due to the fact that although the network connections used in the experiment are extremely fast, there is still some infinitely small delay, delay that is forcing the application to idle. When receiving 25'000 messages, the average time for the pull is 118 milliseconds (+0.047%) slower than that of round robin. This is extremely small for a measurement that lasts 250 seconds. However an additional run of the experiment, with this time 100'000 messages sent, allowed to measure a difference of 410 milliseconds, which is slower by 0.041%. This incredible stability in the experiment gives credit to the fact that this is not due to a random parameter, but that it does underline a behavior of the pull algorithm, which is in fact, ever so slightly sub-optimal, as foreseen by the model.

In the exponential situation, the results have again more variance, but the confidence

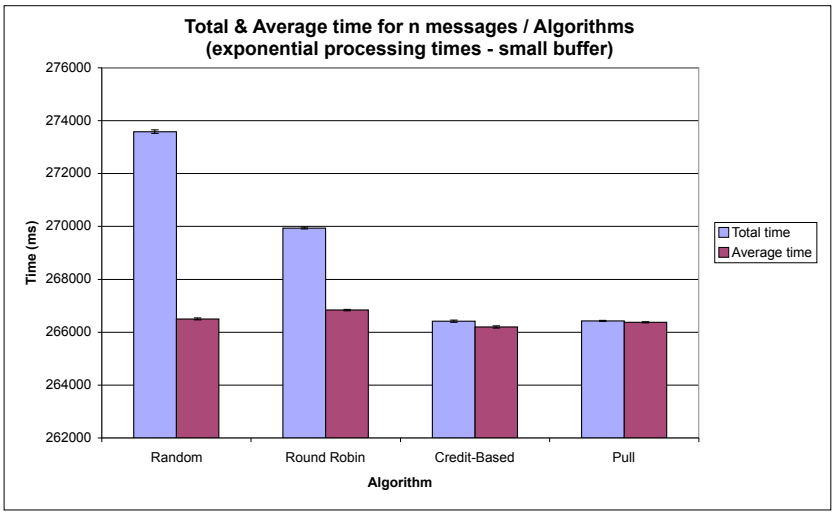

Figure 6.8: *Total and average time for processing n messages / Algorithm (exponential - small buffer).*

intervals stay very small. Graph 6.8 shows the results. The averages are sort of close. Just like in the first case with a bigger buffer, the values should be the same, but they are not. This variation means that the confidence becomes smaller. However, all algorithms are still very performant, since random is only 2.7% more than the total time of the pull, which seems optimal. The credit based algorithm is very close to the pull. Both are close to their respective average.

The confidence intervals shown so far are more distractions than useful information, when plotted on a graph. The stability in the runs of the experiments is impressive and had been detected already whilst doing some simple pre-analysis, before the experiments were run. The confidence intervals for all scenarios stayed very close to what was measured here. They are therefore no mentioned anymore in the next results for the following scenarios.

6.3.2 Basic setup with delay

This second scenario is called "Basic setup with delay" because it stays very similar to the first scenario, and just adds delay. This makes sure that the effects of delay can be clearly identified. The parameters for this first experiment are listed in table 6.2. In this second scenario, three experiments are run. The first with a communication's delay of 10 milliseconds, the second with 50 milliseconds and the third with 300 milliseconds. The delays are for the messages going from the subscribers to the load balancer (see section 6.2.3 for details on the setup of the network emulator NIST Net). The graph on figure 6.9 presents the total and average time for processing n messages in the constant situation, with all three delays side by side for direct comparison.

Table 6.2: *Parameters for the "Basic setup with delay" scenario.*

Parameters	*Value*
Nb Subscribers	6
Processing time	60ms
Nb Publishers	1
Send rate	as fast as possible
Nb of msg published	25'000
Delay experiment 1	10ms
Delay experiment 2	50ms
Delay experiment 3	300ms

First of all it is clear from the graph that the pull algorithm "suffers" from the delay in comparison to the other algorithms. Whether 10, 50 or 300 ms, in the three cases it is slower, and with 50 and 300 ms delay it is really much slower. The extra time is actually very precise and corresponds to what was predicted in the analytical model. As said earlier, the processing time is of 60 ms. In the constant case, each subscriber receives an equal share of the 25'000 messages, that corresponds to 4166 messages. An optimal algorithm needs $4166 * 60 = 250000$ ms to process his share of messages. This is exactly (to the precision of the measurements) what Round Robin needs. Pull on the other hand has exactly one delay per message. That means that with the first delay, it will need $4166 * (60 + 10) = 291620$ ms. In the experiment, the measured total time is of 291784, less than 0.1% off. With the second delay, the model predicts 458260 ms and 458361 ms is measured; in the third case, 1499760 ms are predicted, and 1499882 ms are measured. The precision is remarkable and underlines the plausibility of the model.

The graph also shows that the other three algorithms are very similar, performance wise, and neither of them looks influenced by the size of the delay. However the scale of the graph is not adequate to discuss fine differences between the policies. This will be done further in this section.

The second graph, figure 6.10, shows the same situation but with the exponential distribution and only the first two delays. (The third delay behaves similarly to in the constant case. It is of little interest and just blows up the scale of the graph.) Again, the pull algorithm is clearly losing performance because of the delay, and the higher the delay, the higher the loss. For all algorithms (whatever the delay), the same problem as mentioned in scenario 1 appears: the average of the measured total and average times is higher than that measured in the constant case, and there is some variance. A possible explanation for this is the following. Delays on the return path from the subscribers to the load balancer are not supposed to affect the algorithms that work without feedback. However, communication is done over TCP/IP, which means that there is a control flow using the return flow. The delays affect this flow, and hence most certainly slow down the whole sending process. But although this difference in time with a case without delay appears, for each of the remaining three algorithms, the delay does not make any difference to their performance. They seem very close.

As mentioned earlier, in order to get a precise look at what's going on with the three remaining algorithms (i.e. all but the pull), the scale of the graph must be adapted. Figures 6.11 and 6.12 show respectively a *zoom* (also called a *focus*) in the constant

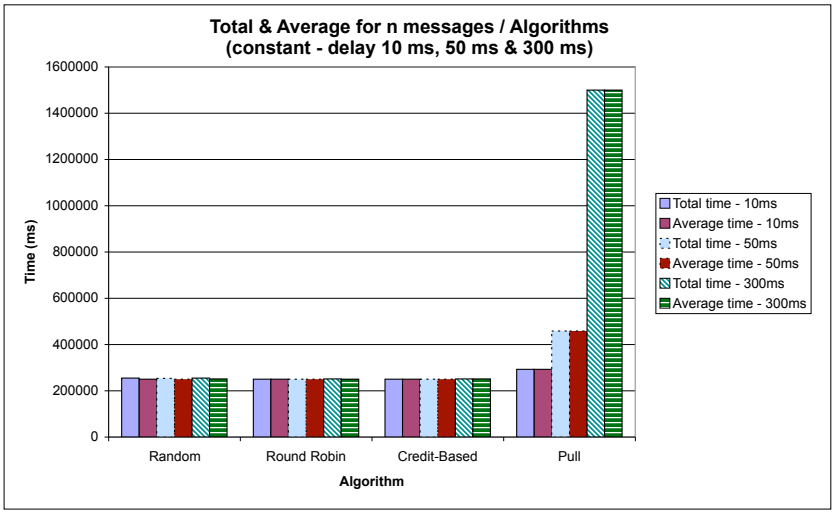

Figure 6.9: *Total & average time for processing n messages / Algorithm (constant - 10ms & 50ms).*

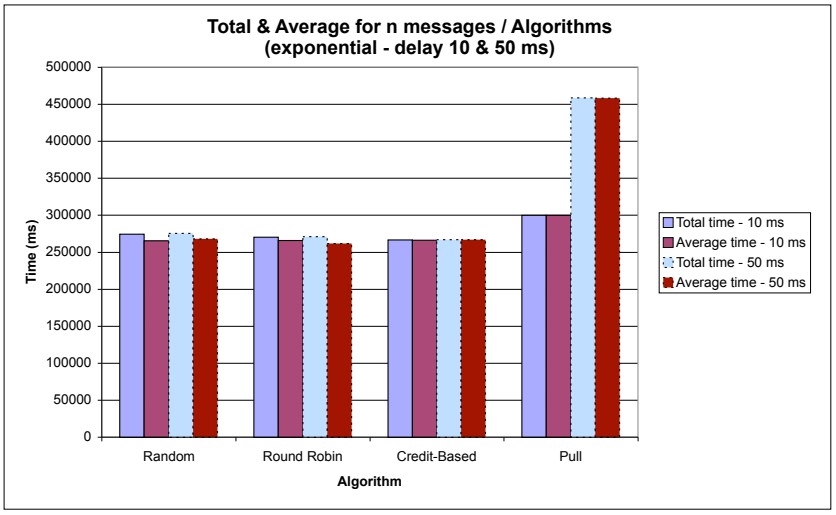

Figure 6.10: *Total & average time for processing n messages / Algorithm (exponential - 10ms & 50ms).*

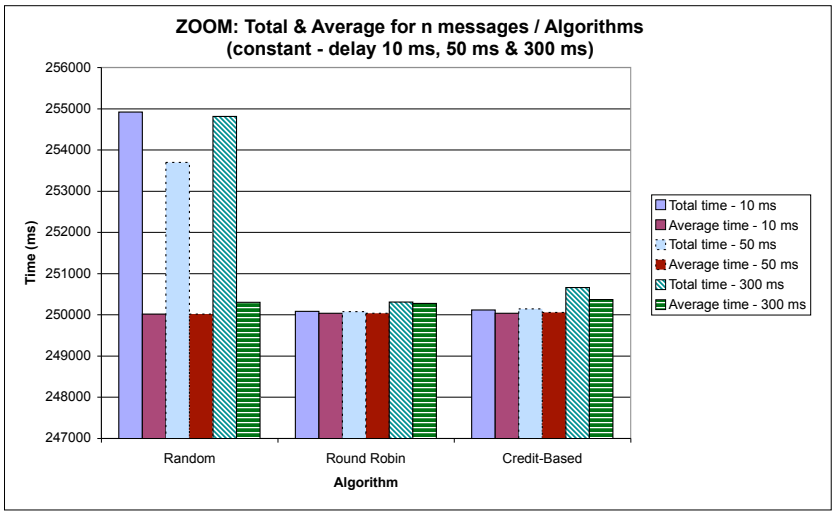

Figure 6.11: *ZOOM: total & average time for processing n messages / Algorithm (constant - 10ms & 50ms).*

and exponential case, with graphs where the time scale has a suppressed zero. The first graph shows that the averages are all the very close. The case with a delay of 300 ms is actually slower than the other cases. According to the predictions of the model, this should come from the difference in time to send the first message. Therefore the difference in average time should be the difference in delay between the measures. The results go in this direction, but the measures are so small that the precision is doubtful. It should be verified with a higher delay, before it can be considered as validating the model. The credit based algorithm seems to be just as effective, whether the delay is 10 ms or 50 ms, but somewhat slower with a delay of 300 ms. The difference in the total times for 10 and 50 ms is of 25 ms ($< 0.1\%$) which tends to support the idea that although there is delay, both cases are performing optimally. With 300 ms delay however, the difference with 10 ms delay for the total times is of 545 ms, which is 0.2% slower. It is difficult to explain this with certitude as the measurements are with very small delays. However high stability in the experiment tends to give credit to this value and therefore could be explained by the fact that this higher delay introduces some idling. The round robin is clearly working at its optimal value. And the random is like in scenario 1, and in accordance with the predictions from the model, saying that the delay does not influence its performance.

In the second graph, the focus is on the exponential distribution. The first thing that is striking is that there is quite some variance in the values. This, as previously explained, comes from the exponential distribution itself. Similarly it can very well be seen that the average of values is once again higher than in the constant case. All three algorithms behave the same way, whatever the delay. Since this is an exponential case, the total times are not exactly the same as their averages. The credit update policy adapts better to the ever changing processing times and has total times that are much closer to the average time.

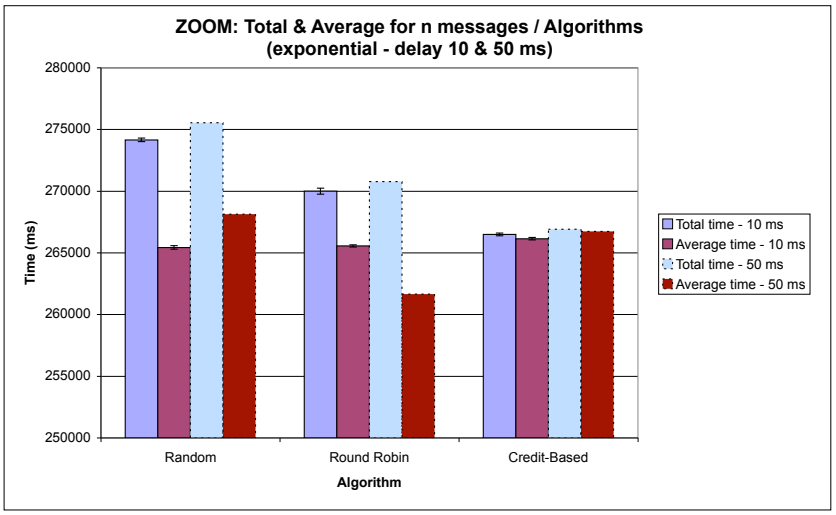

Figure 6.12: *ZOOM: total & average time for processing n messages / Algorithm (exponential - 10ms & 50ms).*

6.3.3 Pattern and no delay

This third scenario is called "Pattern and no delay". It follows the same configuration as the first scenario but there is a pattern of processing times through the subscribers. The direct effects of this pattern should be easy to detect, when being compared to the first scenario. The parameters for this first experiment are listed in table 6.3.

Table 6.3: *Parameters for the "Pattern and no delay" scenario.*

Parameters	*Value*
Nb Subscribers	6
Processing time	according to pattern
Nb Publishers	1
Send rate	as fast as possible
Nb of msg published	25'000
Delay	-

The pattern that was used for this scenario is described in figure 6.13. The same pattern was used for all experiments where a pattern was needed.

In this third scenario the conditions are optimal again (no delay, no bandwidth limitation) but this time the subscribers are not all the same, and each has a different service rate. In the case of the exponential distribution, the different service rates are used as a mean for the exponential distribution. The graph on figure 6.14 shows

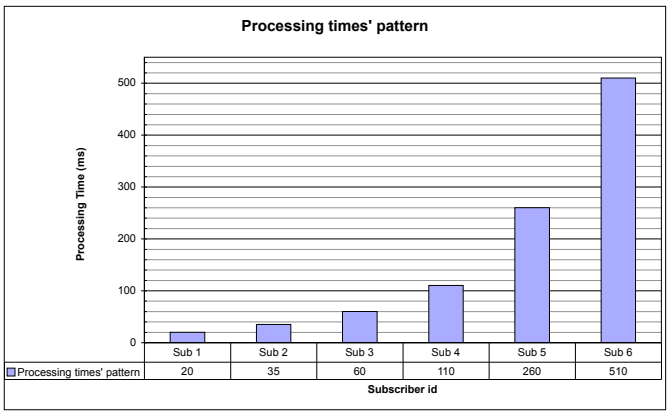

Figure 6.13: *Processing pattern setup across 6 subscribers.*

the total and average times with the pattern, and constant distribution, in the case of a small and bigger buffer.

Neither Random nor Round Robin are very good in this scenario. Their total times are very long. This is normal and had been predicted thanks to the model. By uniformly distributing messages to each subscriber, the system is limited by the slowest subscriber. In this case, the slowest has a service time of 510 ms. Theoretically it should take $(25'000/6) * 510 = 212500$ms. The measured total time is of 212524 ms for Round Robin, less than 0.1% off the foreseen mark, which is excellent. The pull and priority algorithm are both much faster at looking after this load of messages. The first is just as fast, whether with a small or a big buffer, as expected. The second, the credit based policy, is very fast, compared to the feedback-less algorithm, and can be nearly as fast as the Pull with a small buffer. The comparison of results shows that the credit update algorithm is tightly linked to this buffer size. By reducing it, it gained 60% in terms of the total time needed, and close to 30% on the average time. When the buffer is small (fine granularity) then the total time is very close to that of the pull algorithm: 2.3% more. When the distribution is exponential, the same problems appear, and nothing new can be said.

Another way to look at how the algorithms are balancing the load of messages, is to look at the runtime of each subscriber and to see how many messages they each received. The graph on figure 6.15 shows the total time for each subscriber, with the constant distribution and no delay, in the case where a pattern is set. The total times for each subscriber are determined by the average of the five runs for that subscriber. The difference between algorithms with and without feedback can immediately be seen. Random and Round Robin both have the total times of their subscribers that are proportional to the pattern of processing times that was setup. On the other side, the credit based and pull algorithms have their subscribers working the same amount of time. When looking at the number of messages received by each subscriber, the same distinction appears between the two categories of algorithms: this is drawn on figure 6.16. The first, without feedback, has very evenly distributed

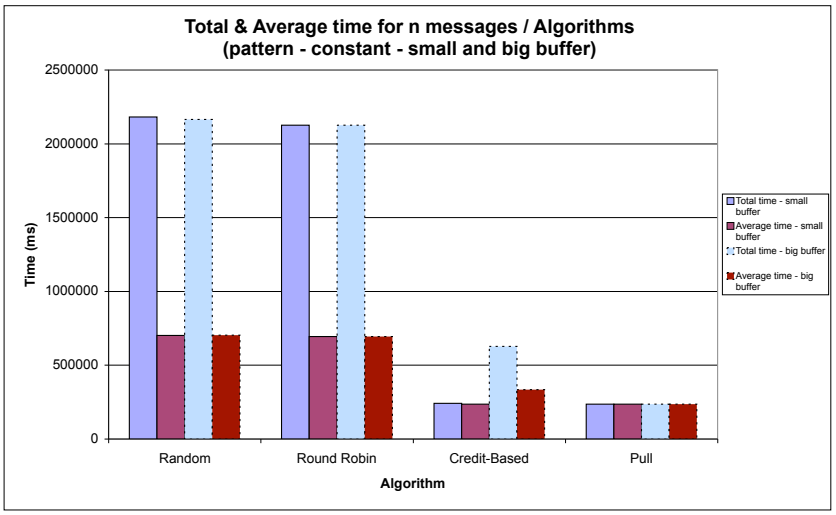

Figure 6.14: *Total & average time for processing n messages / Algorithm (pattern - constant - no delay.*

the load of messages, but the second, with feedback, has generated a distribution that is the opposite to that of the processing pattern. When the algorithms are compared by looking at their total and average time using a small buffer, there are two categories but within them, no differences can be detected with certainty. By looking at the individual total time and the number of messages received, the same statement can be made.

6.3.4 Pattern and delay

The fourth scenario is called "Pattern and delay". It mixes the parameters of scenario 2 and 3 and should be a good opportunity to explore the importance of pattern and delay, in comparison with each other. Measures from the preceding scenario will be used for comparison, in order to enlighten the role of delay in the load balancing policies. The parameters for this experiment are listed in table 6.4.

The pattern that was used for this scenario is the same as in the previous scenario (see figure 6.13). Three kind of delays were used. The first two are the same as in scenario 2. The third is a much bigger delay, in the order of magnitude of a satellite link delay.

In this fourth scenario the conditions are somewhat complex, since both a pattern and delay are active. However what most of all influences the results is the buffer size again. This experiment was run with both buffers, allowing for comparaison. Figures 6.17 shows the very sharp contrast that this buffer size brings, in the total times of the subscribers for the priority algorithm, with constant distribution. With

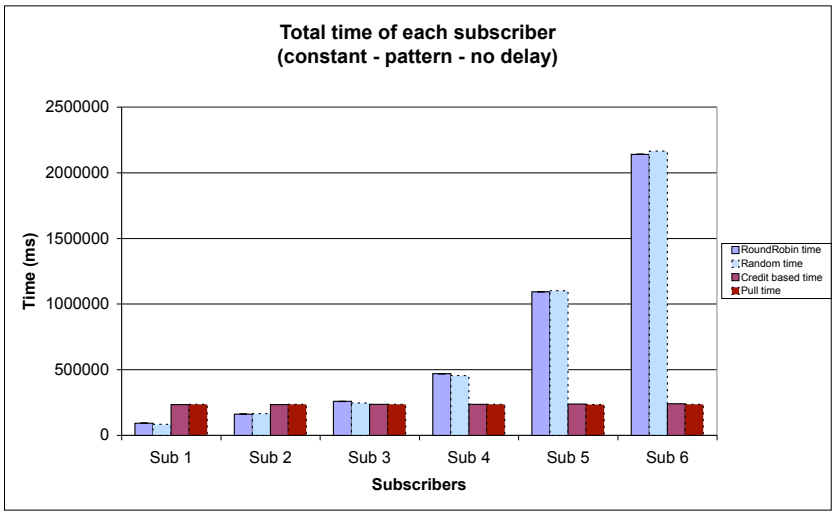

Figure 6.15: *Total time of each subscriber for processing his load of messages (pattern - constant - no delay.*

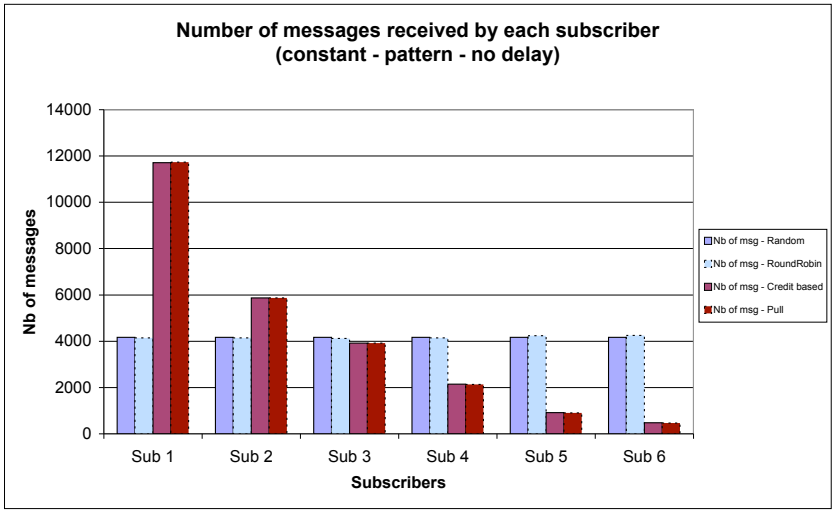

Figure 6.16: *Number of messages received by each subscriber (pattern - constant - no delay).*

Table 6.4: *Parameters for the "Pattern and delay" scenario.*

Parameters	*Value*
Nb Subscribers	6
Processing time	according to pattern
Nb Publishers	1
Send rate	as fast as possible
Nb of msg published	25'000
Delay experiment 1	10 ms
Delay experiment 2	50 ms
Delay experiment 3	300 ms

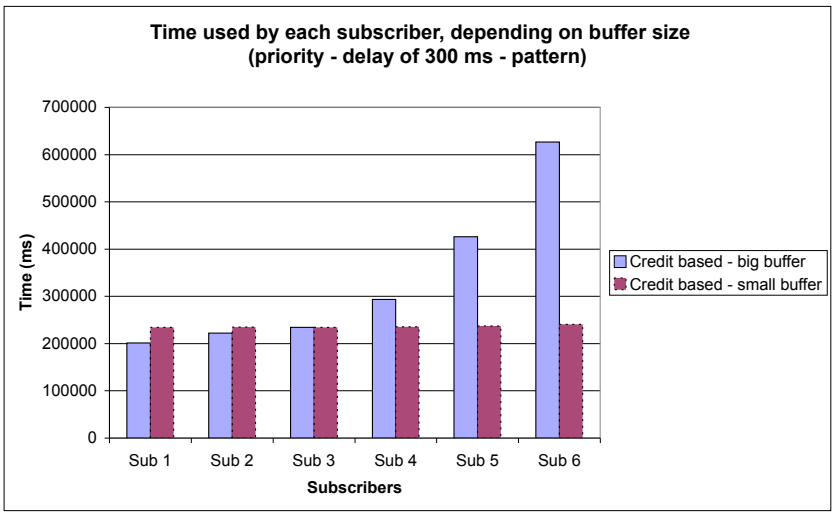

Figure 6.17: *Comparison of the time used by each subscriber of a credit based algorithm, with different buffer sizes (pattern - constant - delay 300 ms).*

a big buffer, the first three subscribers are somewhat close, as expected, but then the time grows steeply. With a longer run (more than 25'000 messages) the effect on the buffer size would diminish, comparatively speaking, to the time of the whole experiment.

By looking at the number of messages received, depending on the buffer size, the effect is very visible once again, as shown on figure 6.18. With a bigger buffer, the distribution of messages is flatter, and leads to having too many messages processed by the slowest of subscribers, and not enough from the fastest. By making the buffer smaller, the problem is much smaller and brings very good performance.

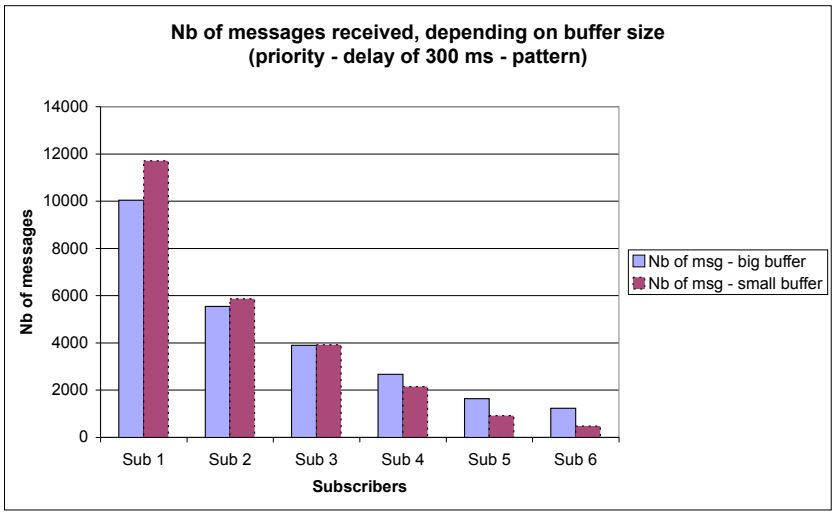

Figure 6.18: *Number of messages received by each subscriber of a credit based algorithm, with different buffer sizes (pattern - constant - delay 300 ms).*

6.3.5 Bandwidth limitation

This last scenario is called "Bandwidth limitation". It follows the same configuration as the first scenario but the bandwidth is limited in the direction "from the LB to the subscribers". The limitation is he same for each subscriber's data support. The parameters for this first experiment are listed in table 6.3. In this scenario, the

Table 6.5: *Parameters for the "Bandwidth limitation" scenario.*

Parameters	*Value*
Nb Subscribers	6
Processing time	60 ms
Nb Publishers	1
Send rate	as fast as possible
Nb of msg published	25'000
Delay	-
Bandwidth limit	1000 bytes/sec

bandwidth limitation shouldn't change anything to the rhythm at which messages are sent to the load balancer since the limitation is unidirectional. However it was rapidly seen that the sending was taking much more time than foreseen. After having analyzed the network more deeply, the behavior was understood. The network emulator works with a buffer. It uses it to simulate network limitation, such as bandwidth limitation, by keeping messages that are not "allowed" to go through the network, because the bandwidth limit has already been reached for the current second. The granularity of NIST Net is therefore a first problem. Secondly there

is a problem due to the implementation architecture for the experiments. Because the subscriber and the publisher are on the same machine, the effects on the one affect the other too. To be precise, after having sent a few messages to the load balancer. The latter starts distributing these to the subscribers. This uses up all of the bandwidth and messages get blocked in the network emulator's buffer until the end of the current second is elapsed. At this moment some messages are freed again until the bandwidth limitation for this second is reached again. This is the granularity problem of NIST Net. Because of this, the *control flow* for the sending of messages also gets buffered and delayed. Since the publishers are not receiving acknowledgments of the reception of messages, these get re-sent, although they originally arrived fine and are stocked in the emulator's buffer. This configuration, although arguably possible in some cases, is not in this case realistic. The goal of such a setup was to simulated the limitation only between the LB and the subscribers. The problems encountered with sending the messages makes it impossible to compare this experiment with the previous ones. However, interesting things can still be understood from the results.

When considering the results of the different algorithms, it can be seen that everything is very chaotic, as shown on figure 6.19[3]. The graph shows the inter-arrival times on one subscriber, when bandwidth limitations are applied, for the four algorithms. For comparison, a fifth value is plotted: the same situation without bandwidth limitations, four the four algorithms (They are the same in this situation and mix into one single line). The situation here is actually closer to a unidirectional channel, than that of a bidirectional one, as mentioned in the model. In this case the pull is expected to suffer from this limitation. But actually the pull algorithm is the most constant in behavior. Because it sends a request for each message, it is purely limited by the emulator's speed to process what is in its buffer. This explains why it is constant. The credit based policy seems to have some kind of frequency that appears. This is certainly due to the credit updates sent, that take a long time to arrive to the load balancer, and then the waiting for the messages to arrive. Basically, because of the setup, there is some idling going on. Finally random and round robin are too chaotic to be interpreted.

6.4 Miscellaneous Results

This part briefly presents some miscellaneous results.

6.4.1 Precision of the model

In the third scenario of the model, whilst discussing the pull algorithm, it is predicted that the inter-arrival time between two requests will be $\tau = d + 1/\mu$. In this case, d is considered to be the round trip time. Because of the setup of the experiment, when a delay is imposed through the network emulator, it is only in one direction and therefore should be considered as $d/2$. When looking at the arrival times of a subscriber, with the pull algorithm and a 10 ms delay setup up

[3]The graph is complex and optimized for viewing in color. Using various dotted lines, and various widths for a grey and white visualization makes the graph totally impossible to understand, hence the choice for color.

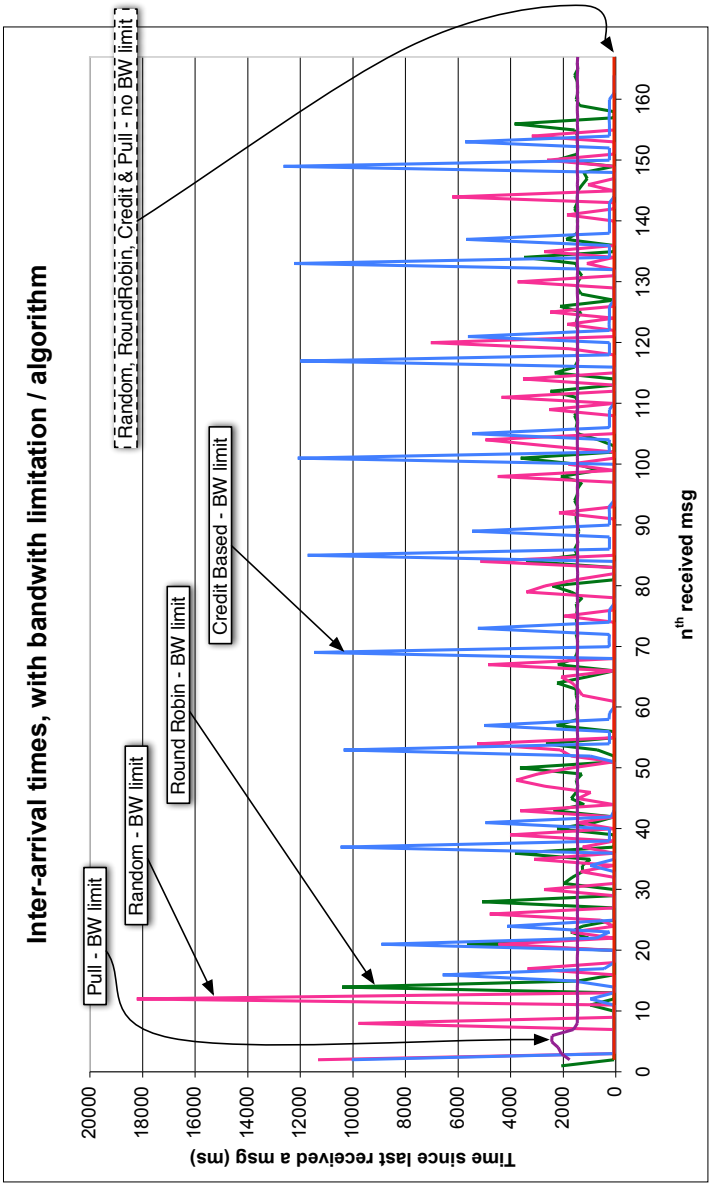

Figure 6.19: *Inter-arrival times of the sent message at the Load Balancer.*

(constant), since the processing time are 60 ms, the expected time between two arrivals, according to the model should be $\tau = 10 + 60 = 70$ ms. When looking at the measured values, the average inter-arrival time is 70.06 ms, with a variance of only 1.6 . In this situation, the accuracy of the model is excellent.

7 Further Improvements

7.1 Overview

This chapter presents a few ideas for future work, and improvements that could be brought to the architecture. The algorithms that were selected and tested were efficient, but still do not cover the full range of situations that could exist. A new algorithm, mixing two of the advantages seen in the tested policies is presented. Finally a completely different architecture is proposed for the load balancer, in order to be even more powerful in certain cases.

7.2 Possible improvements

7.2.1 Weighted Round Robin

Weighted Round Robin (WRR), in the way it is used classicaly, is not really foreseen for the architecture of this experiment. However, it has some similar characteristics to Weighted Fair Queuing (WFQ). They both combine a form of priority queuing and have some fair queuing activity. In the case of WRR, the fairness is an approximation since packet size can still bias this distribution, which is not the case for WFQ. Based on this observation, an adapted algorithm can be proposed that has two main parts. The figure 7.1 presents this solution graphically. The idea is, in a first step, to perform an equivalent to priority queuing, by having different priorities fixed for each subscriber. At this stage, it is easy to consider all subscribers with similar priorities as belonging to the same pool. Therefore pools have different priorities. In priority queuing, the higher priorities are always filled up first. In this case the priorities are more like probabilities. Each pool has a probability of receiving each message. Once a pool of subscribers has been created, it is easy to distribute messages in a fair way within the pool. This is the second step, that is inspired by fair queuing: the idea is to apply a round robin rule within each pool, when distributing messages. By behaving this way, the probabilities ensure that the most capable servers receive the heaviest loads, and within the members of a pool the load is balanced in a fair way. Such an adapted version of the WRR algorithm should be very interesting since it approximates the behavior of WFQ, which is known to be very efficient.

The main problem introduced by this algorithm is that is must be statically configured at the beginning. Creating "pools" of subscribers is obviously only efficient if the clients within the pool have similar processing capabilities. Typically slow server could be aggregated in a pool with lower priorities, and faster ones could be grouped in an another pool with much higher probability. In order to determine a

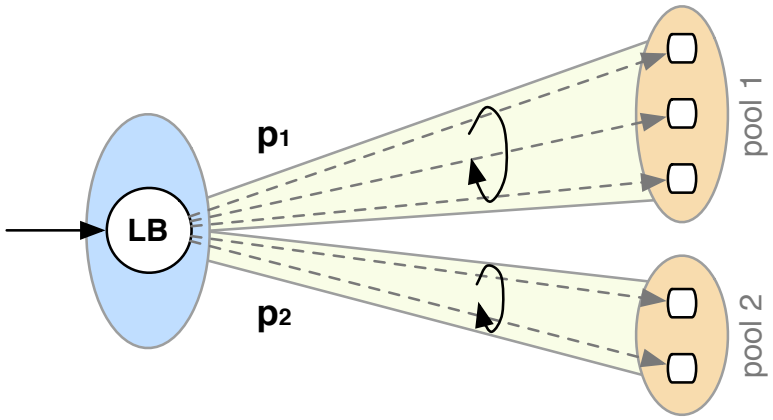

Figure 7.1: *Proposed architecture for a weighted round robin algorithm implementation.*

server's processing speed, benchmarks can be performed, but this speed might not always stay the same. The efficiency of the algorithm depends on the similarity between the benchmarks and the real value. A possible way to tackle this problem might be to introduce some lightweight control mechanism that supervises the number of messages processed over a certain period of time (which will become a new granularity issue). Such entities could ensure that subscribers would always be in the right pool. In case one subscriber's processing capabilities suddenly drops, the control mechanism would move it to a pool of lower probability. The opposite would happen when a server's processing capabilities gets suddenly augmented: it would be moved into a pool of higher priority.

During the course of the project, the algorithm was implemented as a test, but not as a chosen algorithm for the experiments. Some implementation issues were not resolved adequately, forcing this algorithm to be excluded from the experiments. The most difficult point being the flexibility needed by the algorithm in order to accept the creation and death of new and old pools of subscribers. However the prioritized distribution with round robin within the pool worked fine. It could be interesting to see if a control mechanism could be developed in order to evaluate the performance of this algorithm, and check whether the advantages over classical solutions that WFQ brings, can be found in this case.

7.3 Future work

7.3.1 Class based load balancer

Whilst reading multiple documents on the topic of classic load balancing, it is interesting to notice that a significant number of people talk about the benefits (i.e. speedups) of caching. The situations were obviously in no way comparable to the current Load Balancing architecture, but this inspired what follows.

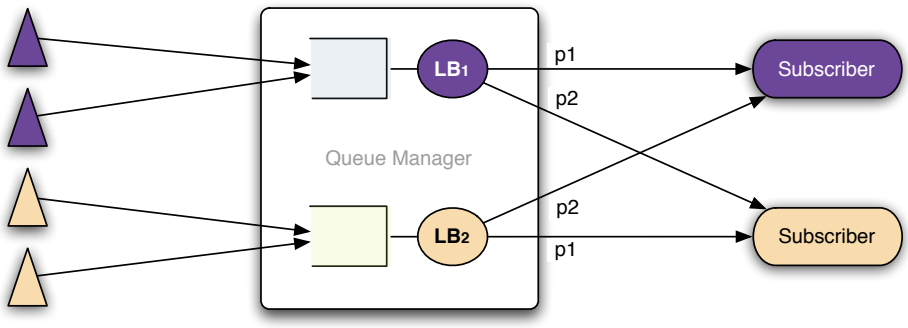

Figure 7.2: *Proposed architecture for class-based load balancing.*

The main idea is that if all similar packets could be sent to the same Edge Server, some performance gains might be achieved thanks to caching in the memory of the Edge Server. Content aware policies tend to do generate this caching advantage. Based on that observation, the following model has emerged. The assumptions made for the reasoning are:

1. there are two different kind of messages
2. there are two different kind of Edge Servers

In order to classify the packets of data, some *detection* has to be done. This detection shouldn't be done at the MQ level because this would lead to an application-specific choice, and this is not advisable in this case because of independence and scalability issues. However, there is an easy and realistic possibility for solving this situation. By having different kinds of sensors connected to different queues of the Queue Manager, the classification is already performed. Figure 7.2 shows this architecture. Two kind of sensors are drawn, and each kind is connected to a different queue. The Queue Manager then has two load balancing entities. These try and send their load of data to some subscriber with a high probability (p1 on the figure), and have a much lower probability p2 of sending messages to other server. The two load balancing entities do not consider the same subscribers as their main subscribers. By using priorities in such a way, each load balancer always tries to send his type of messages to the same subscribers, therefore encouraging possibilities of caching speedups on the subscribers. If these become overloaded, there is always the possibility to send messages to other subscribers, that are used by other load balancers, but that are not overloaded.

The current implementation of the MQ already has multiple queues. Hence it would be perfectly possible to set it up with multiple queues such as to implement this kind of class-based load balancing mechanism.

7.3.2 Resiliency

One of the possible aspects to study could have been to look at the system's resiliency, it capacity to return quickly to a previous good condition, whenever a

server crashed or a new one appeared. This subject was left aside because of the time limit on the project. In order to measure precisely the effect of a the appearance of a new server, or the closure of an existing one, a certain number of elements in the code need to by synchronized. Without a synchronization mechanism and an adequate structure for reproducing exactly the same event at the same moment in time, there is no point in trying to measure the impact of server addition or removal. This topic could be an good point to explore for future work.

7.3.3 Message Size

Another parameter that was not studied in the experiment is the variance in message size. The same sized messages were always used for measuring the algorithms performances. Such an element could also influence the performance of algorithms, since certain restrictions, such as bandwidth limitation could be reached faster. Message size could be considered as an even higher classification parameter than the bandwidth limitation in Chapter 6.3 and would hence double the number of experiments to be run for checking all situations.

8 Conclusions

This project has been very exciting. Despite the short time of six months, a number of interesting results were found. The fact that a real messaging architecture was implemented and not just simulated, was a wise choice. Certain behaviors would most certainly not have been discovered if the architecture had only been simulated. Furthermore the remarkable precision and consistency in the measured experiments would never have been underlined by a simulation.

The methodic and considered thinking in the early stages of the project led to the establishment of an analytical model. The model was an excellent way to think of the critical and influential parameters for the experiments. Above all, it proved itself to be remarkably precise, since the experiments often produced the foreseen results, within less than 1%. This model should be useful for further research on such architectures.

All four of the chosen algorithms behaved globally extremely well. Whenever no disturbing parameters were in action, the algorithms reached the foreseen optimum. It was reassuring to see that even when the situations were not favorable for some algorithms, they still had good performances. Furthermore, for each main experimental scenario, at least one algorithm was found to be optimal which is very useful.

One important aspect was also to see whether the complexity of the load balancing algorithms would influence the performance of the system. The experiments clearly underlined the fact that, whether complex or simple, the load balancing policies were not a bottleneck for the system's performance. The subscribers' abilities to process messages were much more important than the load balancing delays introduced by the complexity of any tested algorithm.

Based on a set of realistic parameters that could influence the global performance of the system, the project succeeded in highlighting the respective importance of each of these parameters. The model correctly anticipated that algorithms without feedback would not fundamentally suffer from delay. The biggest "surprise" is arguably the importance of the subscribers' buffer size. It is also very good to find out that a client-driven algorithm can still be very performant in many situations, but that as foreseen, the algorithm performs badly when the system has high delays. Finally it was very positive to see that the originally implemented policy was in fact the most polyvalent one. Whatever the parameters, it was always close to the optimum and even perfectly optimal in several configurations.

As with every project, some answers are found and new questions are raised. The research carried out in this project successfully performed an analysis of the behavior of load balancing schemes in a message queuing system. The results are very interesting and open the field for further investigations.

Bibliography

[Bos91] Bosch. Control area network specification, version 2.0. Technical report, Robet Bosch GmbH, D-70422 Stuttgart, 1991.

[Bou04] Jean-Yves Le Boudec. Performance evaluation of computer communication systems. `http://ica1www.epfl.ch/perfeval`, nov 2004.

[CS04] Mark Carson and Darrin Santay. Nist net - a linux-based network emulation tool. *CCR Computer Communication Review*, 33(3), 2004.

[DB78] Peter J. Denning and Jeffrey P. Buzen. The operational analysis of queuing network models. *Computing Surveys*, 10(3), 1978.

[Decie] Jean-Dominique Decotignie. Real-time programming. Lecture notes, EPFL, http://lamspeople.epfl.ch/decotignie/.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and similation of a fair queuing algorithm. *Proc. SIGCOMM '89*, 19(4), 1989.

[emu] Definition of the term "emulation" by wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Emulation/`.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, Octobre 1994.

[Hah86] E. Hahne. Round robin scheduling for faire flow control. Ph.d. thesis, Dept. Elect. Eng. And Comput. Sci. M.I.T., Dec. 1986.

[Hit02] Ron Hitchens. *Java NIO*. O'Reilly, 2002.

[HNTer] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 1991.

[JNI] Java nio homepage. `http://java.sun.com/`.

[Mei00] René Meier. State of the art review of distributed event models. Technical report, University of Dublin, March 2000.

[mqt] Telemetry integration. `http://www-306.ibm.com/software/integration/mqfamily/integrator/telemetry/`.

[NIS] Nist net homepage. `http://www-x.antd.nist.gov/nistnet/`.

[RBS05] Sean Rooney, Daniel Bauer, and Paolo Scotton. Edge server software architecture for sensor applications. *The 2005 Symposium on Applications and the Internet (SAINT'05)*, pp. 64-71, 2005.