Using Fluid Models for AQM Evaluation

by

Boris Taillard

Dissertation

Presented to the

University of Dublin, Trinity College

in partial fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

____
Boris Taillard

September 9, 2005

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

____

Boris Taillard

September 9, 2005

Acknowledgments

I would like to thank my supervisor, Meriel Huggard, for suggesting such an interresting subject for my dissertation and for the time she spent helping me with this work.

I would also like to thank Mathieu Robin, for making himself available for discussions and sharing his excellent knowledge in the subject area of my project with me. This saved me a lot of time during my work on this thesis.

Finally, my classmates and the whole population of Ireland deserve a special mention as I have spent a fantastic year studying in this wonderful country.

BORIS TAILLARD

University of Dublin, Trinity College
September 2005

Abstract

Using Fluid Models for AQM Evaluation

Boris Taillard
University of Dublin, Trinity College, 2005

Supervisor: Meriel Huggard

In spite of the congestion management mechanisms included in the TCP/IP protocol, the Internet still suffers from a lack of optimization in the way network overload is managed. One of the reasons for this is that the original congestion avoidance techniques specified by TCP/IP are implemented at the edges of the network. This means that the critical mission of reacting to link saturation relies on hosts that don't have a global view of the network. Moreover they do not have any obligation to respect a set of common rules. One possible solution to this problem is to introduce congestion management inside the core routers of the network. One way of doing this, Active Queue Management, consists of designing intelligent algorithms that start dropping packets before the routers' queues get full, so that they are always ready to accommodate bursts of traffic.

The rapid growth of the Internet has presented many challenges for those wishing to design and validate new protocols. Almost all such advances are evaluated through simulation (e.g. using the *ns* packet-level simulator). Because the internet is growing very quickly, researchers had to invent new methods to simulate its activity. Indeed, it has been quite a few years since the processing capacity of regular packet-level network simulators was sufficient to simulate the behavior of a network like the Internet. One alternative is to simulate the network traffic at a flow-level. Such a higher-level simulator would be a useful tool for network protocol design, because this kind of simulator is able to scale a lot more easily than a packet-level one.

This project looks at a fluid-based model for flow-level network simulation, which can be easily extended to support a variety of Active Queue Management schemes. The work consisted of validating, optimizing and extending an implementation of this model, while also making sure that it is suitable for evaluating Active Queue Management techniques. Ultimately, the simulator was successfully used to evaluate a number of these schemes.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **x**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Context 1

1.2 Project goals and outcomes 2

1.3 Dissertation outline 2

Chapter 2 Background **3**

2.1 Introduction 3

2.2 TCP/IP and network simulation 4

2.3 Active Queue Management 4

2.3.1 Principles 4

2.3.2 An AQM scheme: RED 6

2.4 Network simulation 8

2.4.1 Simulation techniques and application to networking 8

2.4.2 Packet-level simulation and the *ns* network simulator 12

2.5 Flow-based simulation of TCP/IP 13

2.5.1 A fluid-based model for network simulation 13

2.5.2 An implementation of the model 17

2.6 Conclusion 17

Chapter 3 Optimization and validation of the simulator **19**
3.1 The simulator 19
3.2 Optimization of the simulator 20
3.2.1 What to optimize? 20
3.2.2 Evaluation of the optimizations 24
3.3 Validation of the simulator 33
3.3.1 Validation with a simple predictable network 33
3.3.2 Validation against other results 36
3.3.3 Validation against *ns* 38
3.4 Conclusion 46

Chapter 4 Extension of the simulator **47**
4.1 Introduction 47
4.2 Better control over the classes of flow 47
4.2.1 Motivations 47
4.2.2 Implementation 48
4.2.3 Evaluation 48
4.3 Model reduction 50
4.3.1 Motivations 50
4.3.2 Implementation 52
4.3.3 Evaluation 55
4.4 Adding more AQM techniques 61
4.4.1 Preliminary work 61
4.4.2 PI 61
4.4.3 DRED 62
4.4.4 AVQ 62
4.5 Conclusion 63

Chapter 5 Evaluation of different AQM schemes **64**
5.1 A methodology to evaluate AQM schemes 64
5.1.1 Metrics for AQM schemes evaluation 64
5.1.2 A classic simulation scenario for AQM schemes evaluation . . . 65
5.2 Comparison of different AQM schemes 66

5.2.1 Experimental setup 66
5.2.2 Experimental results 67
5.3 Conclusion 72

Chapter 6 Conclusion **74**
6.1 Achievements 74
6.2 Future work 75

Bibliography **77**

List of Tables

3.1 Technical data and parameters used for the performance evaluation . . 26

4.1 Active classes of flows for the class-control validation 49
4.2 Active classes of flows for the model reduction validation 55
4.3 Result of the model reduction for the sample network 59
4.4 Execution time for the model reduction validation scenario 61

5.1 Active classes of flow for the evaluation of AQM schemes 67

List of Figures

2.1 The different elements to take in account in network simulation 5
2.2 RED drop function [13] 7
2.3 Packet-level network simulation 10
2.4 Flow-level network simulation 11
2.5 Flowchart of the operations to be performed during the simulation . . . 18

3.1 Class diagram for the network simulator 21
3.2 Defining a few links with the old configuration file format 21
3.3 Defining a link with the new configuration file format 22
3.4 The network topology used to evaluate the performance optimizations . 25
3.5 Influence of the number of nodes on the execution time 27
3.6 Influence of the simulation time on the execution time 28
3.7 Influence of the number of flows on the execution time 29
3.8 Influence of the number of nodes on the memory usage 30
3.9 Influence of the simulation time on the memory usage 31
3.10 Influence of the number of flows on the memory usage 32
3.11 The network used for analytic result based validation 33
3.12 Simulation results for the analytic result based validation 35
3.13 The validation network used by Liu et al. 37
3.14 Results obtained by Liu et al. (fig. 7 from [14]) 38
3.15 Simulation results with the Liu et al. test topology 39
3.16 Network topology used to validate against *ns* 40
3.17 State variables for the congested link (*ns* validation) 41
3.18 Arrival rate at each link (validation with *ns*) 42
3.19 Window size for each flow (validation with *ns*) 44

3.20 Network topology used to compare the performances with *ns* 45
3.21 Results of the performance comparison with *ns* 46

4.1 The network topology used to validate the class-control 49
4.2 Simulation results for the class-control validation 51
4.3 The network topology used to validate the model reduction 56
4.4 Simulation results for the model-reduction validation 58

5.1 The dumbbell topology [20] 66
5.2 Comparison of the network traffic for the different AQM schemes . . . 68
5.3 Comparison of the end-to-end delay for the different AQM schemes . . 69
5.4 Comparison of the loss-rate for the different AQM schemes 70
5.5 Total network utilization for the different schemes 71

Chapter 1

Introduction

1.1 Context

In spite of the congestion management mechanisms included in the TCP/IP protocol, the Internet still suffers from a lack of optimization in the way network overload is managed. One of the reasons for this is that the original congestion avoidance techniques specified by TCP/IP are implemented at the edges of the network. This means that the critical mission of reacting to link saturation relies on hosts that don't have a global view of the network. Moreover they do not have any obligation to respect a set of common rules. One possible solution to this problem is to introduce congestion management inside the core routers of the network. One way of doing this, Active Queue Management, consists of designing intelligent algorithms that start dropping packets before the routers' queues get full, so that they are always ready to accommodate bursts of traffic.

The rapid growth of the Internet has presented many challenges for those wishing to design and validate new protocols. Almost all such advances are evaluated through simulation (e.g. using the *ns* packet-level simulator). Because the internet is growing very quickly, researchers had to invent new methods to simulate its activity. Indeed, it has been quite a few years since the processing capacity of regular packet-level network simulators was sufficient to simulate the behavior of a network like the Internet. One alternative is to simulate the network traffic at a flow-level. Such a higher-level simulator would be a useful tool for network protocol design, because this kind of

simulator is able to scale a lot more easily than a packet-level one.

1.2 Project goals and outcomes

This thesis presents the outcomes of a five-month project whose goal was to enhance a large-scale network simulator so that it can be used to design and optimize Internet protocols. The simulator uses a fluid-based mathematical model presented in [14] that has the property of scaling very well.

The first objective was to become familiar with the above model and an existing implementation of it. Then, the work consisted of validating, optimizing and extending the code in such way that it would be perfectly suitable for simulating Internet-like networks. The validation had to be carried out by different means in order to guarantee the correctness of the results. The optimization efforts were concentrated on finding ways to make the application more scalable towards the size of the networks to be simulated. The extensions that were added address different concerns that made the simulator less suitable for large-scale simulation or protocol evaluation. The ultimate goal constisted of demonstrating that the result of this work can be used for protocol evaluation by using it to compare Active Queue Management schemes.

1.3 Dissertation outline

The first part of this dissertation presents the different concepts and principles that are needed to understand this project, including introductions to network simulation, Active Queue Management, and the model mentioned above. It also gives a first suggestion for the implementation of the simulator. Then, the optimization and validation operations that were carried out on the simulator are presented, before introducing the different extensions that were added to the application. These extensions are the improvement of the control that the user has over the flows that go through the network, a model-reduction that allows faster simulations, and the addition of support for more AQM schemes. The final chapter will present a way to use the simulator to evaluate the behaviour of previously added Active Queue Management schemes.

Chapter 2

Background

2.1 Introduction

In this chapter we introduce the necessary background information needed to understand the rest of this document. Special attention should be given to the last section of this chapter as it explains the *model* on which the simulator at the core of this work is based. The theoretical foundations [14] of this model are outlined.

The primary goal of this work is to implement a TCP/IP network simulator. Only those elements of relevance for simulation are explained; see [9] for a complete overview of TCP/IP.

A secondary goal of this work is to evaluate a number of *active queue management* mechanisms. A section is dedicated to the presentation of the general concept of AQM as well as the first AQM scheme to be implemented in the simulator: RED.

The basic theory and concepts used in network simulation are then introduced. We also describe *ns*, a network simulator that is widely used and was used as a validation tool for this work.

Finally, the last section presents the *model* used for the simulator and the chosen implemention methodology.

2.2 TCP/IP and network simulation

This section provides an introduction to the relevant terminology associated with TCP/IP networks. It is assumed that the reader is familiar with the concepts of *gateways* (also called *nodes*), *addressing*, *routing*, *packets*, *sequencing* and *windowing* as described in the original IP paper [1], as well as with the notion of *congestion avoidance* [2].

As stated in [9], the following entities have to be modeled in order to design a network simulator: *links*, *nodes*, and *load*. Those entities are illustrated in figure 2.1 and can be described as follows:

- Physical *links* may be simulated accurately. Each link has a given capacity and can transmit data from one *node* to another at a maximum rate given by this capacity.

- *Nodes* usually are either routers or hosts (in this document, the hosts, i.e. sources of traffic, are considered to be "part of" the router that connects them to the network). In the simulations studied in the following sections, as in a real-world TCP/IP network, a queue is associated with each *node*, or with each interface that connects the node to a *link*. These queues, which each have a maximum capacity, are used by router to store packets that can not be directly transmitted because of congestion on the outgoing link.

- Traffic *load* is the data that is sent through the network. Depending on the type of simulation used it can be either viewed as a set of *packets* present on the network or as a set of *flows*. To each flow is associated a *window size* that influences its current transmission rate (or more precisely the number of packets that can be sent before getting an acknowledgement).

2.3 Active Queue Management

2.3.1 Principles

Despite the improvements such as *congestion avoidance* mechanisms presented by Jacobson in the late 80s [2], Internet traffic is still subject to congestion. Indeed, this

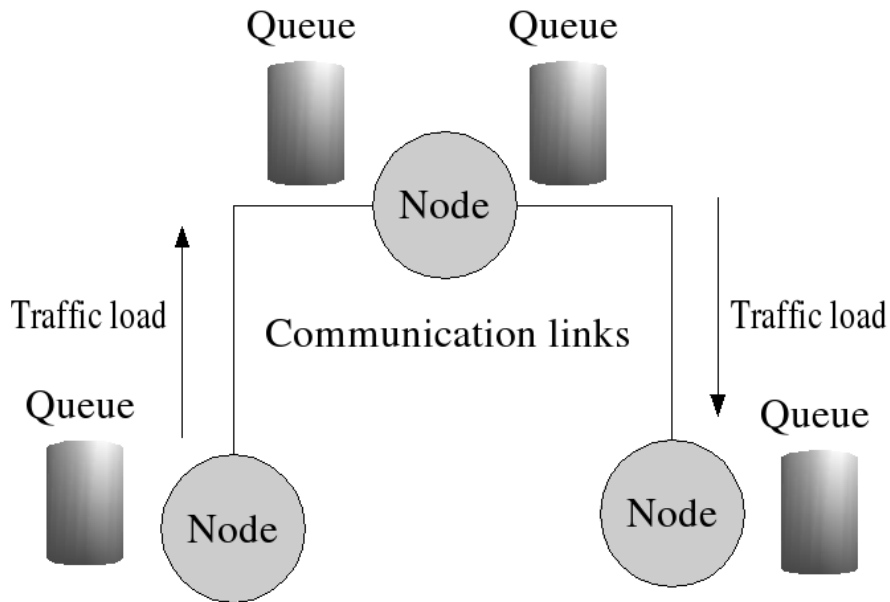


Figure 2.1: The different elements to take in account in network simulation

congestion avoidance mechanism only applies to the edges of the network, limiting the amount of control that can be performed. As a consequence, there was perceived a need to introduce congestion management within the network routers, in particular to manage the way in which network traffic is queued. The obvious - and most simple - technique is called *drop tail*. It uses a defined maximum queue length and accepts new packets into the queue as long as this queue is not full, and drops incoming packets when the queue is completely full. When packets from the queue have been transmitted, new incoming packets can be accepted again. *drop tail* has many drawbacks, among which are the fact that a single *flow* can monopolize queue space and prevent others from using the queue, or the fact that because the queues are always full in case of congestion no bursty traffic can be transmitted (contradicting the queue's primary goal of absorbing traffic bursts). Alternatives to *drop tail* exist, e.g drop the first packet of the queue when it is full (*drop front when full*), which solves the first problem, but not the second one.

The only solution to the full queue problem is to start dropping packets before the queue is full. This is called *active queue management* (AQM) and is the subjet of an RFC in the Internet community [5]. The original AQM scheme [4], RED, will be

described in the next section. According to the RFC the common characteristics of of all AQM techniques should be to:

- Reduce the number of packets dropped in routers. Moreover, by keeping queue-sizes small and limiting the storage of packets that are part of "steady-state" traffic, AQM makes the absorption of bursty traffic easier.
- Provide a low delay, interactive service. Because the queue-length is kept small, the latency tends to be better, which is an advantage for interactive applications.
- Avoid *lock-out* behaviour. Lock-out occurs when a single *flow* or connection monopolizes the queue, which is clearly undesirable because it makes the network less fair. This problem is naturally solved by *active queue management* as there will always be room in the queue for a new incoming packet.

2.3.2 An AQM scheme: RED

Random Early Detection (RED) was first described in [4] as a way to "keep the average queue size low while allowing occasional bursts of packets in the queue". In order to do so, some packets are dropped probabilistically before the queue becomes full. The probability of a packet getting dropped depends on the average queue length in the "recent past". The average queue size is computed using a simple exponentially weighted average. The decision to drop a packet or not is the following way:

- If the queue size is smaller than the t^{min} parameter, no packets are dropped.
- If the queue size is between the t^{min} and t^{max} parameters, the probability of a packet being dropped is chosen. This is between 0 and the p^{max} parameter and is proportional to the size of the queue.
- If the queue size exceeds t^{max} then all incoming packets are dropped.

The behavior of RED can be summarised by the following equations, which defines the probability $p(x)$ of a packet getting dropped given an average queue size x (figure

2.2 gives a graphical represention of this function):

$$p(x) = \begin{cases} 0 & \text{if } 0 \leq x < t^{min} \\ \frac{x - t^{min}}{t^{max} - t^{min}} p^{max} & \text{if } t^{min} \leq x \leq t^{max} \\ 1 & \text{if } t^{max} < x \end{cases} \tag{2.1}$$

In [13], the average queue size $x(t)$ is described by the following equation, which uses an Exponential Weighted Moving Average (EWMA):

$$\frac{dx}{dt} = \frac{log_e(1-\alpha)}{\delta} x(t) - \frac{log_e(1-\alpha)}{\delta} q(t) \tag{2.2}$$

where δ and α are the sampling interval and the weight used for EWMA, and $q(t)$ is the queue length at time t.

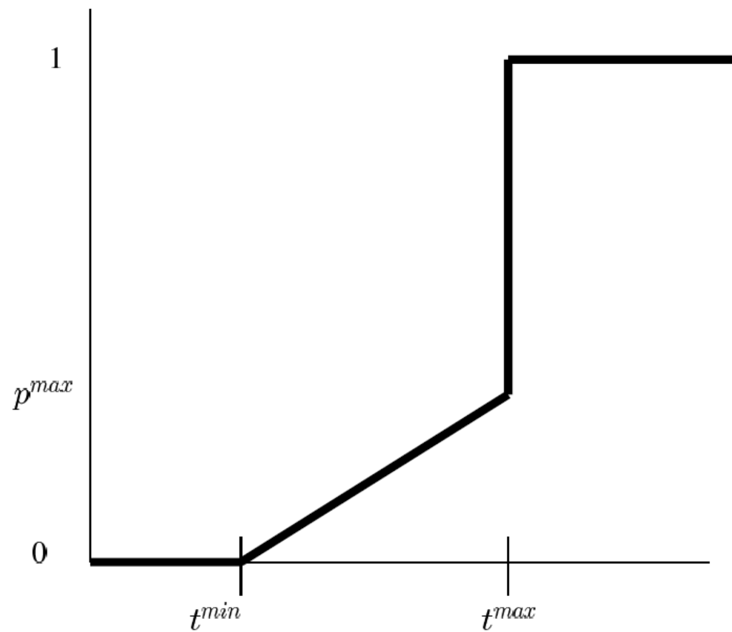


Figure 2.2: RED drop function [13]

The use of randomness in the choice of which packets to drop protects the router from the *lock-out* problem. The control of the queue size helps in the accommodation of traffic bursts. Indeed, if congestion occurs, the average queue length in the recent

past will be long and consequently a lot of packets will be dropped to keep some room in the queue. However, in the case of a bursty traffic, more packets will be accepted in the queue for the same incoming rate, as the queue length in the recent past will have been significantly smaller. In summary the algorithm will tolerate an almost full queue only if this is done to handle bursts.

2.4 Network simulation

2.4.1 Simulation techniques and application to networking

According to [8], simulating consists of imitating a real-world facility or process, usually using a computer. The process or facility to be simulated is called a *system*. In order to study a *system*, certain assumptions, usually mathematical or logical relationships, are made about the way it works. The whole set of assumptions constitutes a *model* that can be used to specify the behaviour of the system. If the *model* is simple enough and is known to capture all the *system* variables perfectly, it is then possible to obtain exact results by finding an *analytic* solution to this *model*. However, for most real-world cases, it is either impossible to take every possible parameter into account, or a complete *model* is too complicated to allow the exact computation of the result. In contrast to a complete *analytic* solution, simulation provides a numerical estimate of the desired characteristics of the *model* through simplifications that reduce the computational power needed to solve the model. However these simplifications reduce the accuracy of the results obtained. It is therefore important to pick the *model* simplifications - or reductions - carefully so that the principal factors that influence the *system* are simulated (for example, when simulating a flow of cars on several roads, the way a car works doesn't have to be simulated if the only field of interest is the flow itself, but it decreases precision as mechanical failure will no longer be taken in account by the model, and the failure of an individual car could influence the whole flow).

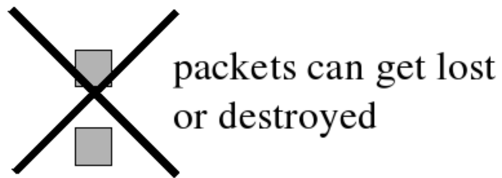
A simulation *model* can either be *deterministic* or *stochastic*. In a *deterministic model* the output is totally "determined" by the input, whereas a *stochastic model* introduces some randomness. A system of differential equations can be used to describe a complicated system in a *stochastic* way. In the case of network simulation, the behaviour of TCP flows can, for example, be described stochastically. This will be

discussed in more detail later.

A *discrete-event* simulation is a simulation in which state variables change instantly every given amount of time. Each point in time represents a step of the simulation; all the variables are updated according to the *model* and then the next step is processed, until the complete simulation is finished. This method of simulation works well for network simulation and is used by almost all the simulation techniques presented in this report.

The general concepts about simulation apply to network simulation, but some, more specific, problems have to be taken into account. An important issue is to decide what - or more precisely at what scale - to simulate. TCP traffic on a network can be viewed as packets that are transmitted on communication links and go through routers. And obvious way to simulate the traffic is then to implement a packet level simulator. This models all the packets travelling on the network and updates their states at each step of the simulation (see figure 2.3). This technique perfectly represents what is actually going on in the network, but doesn't scale very well. Indeed, with a large scale network, the simulator has to handle a very large number of packets, and carry out operations, such as routing and queue management, that would, in the real world, be distributed among all the routers present in the network .

An more scalable alternative to packet-level simulation is to view the traffic as a set of TCP *flows*. A *flow* is a group of packets that have the same characteristics: they have the same source, the same destination, and the same window size. Each *flow* can either be valid for the whole simulation time or start and end at given times. With this approach, it is not necessary to simulate all the packets but only to find an appropriate *model* that matches the behaviour of a *flow* (see figure 2.4). This approach scales well because the processor time needed to simulate a *flow* is usually a lot less than the processor time needed to simulate all the packets that are part of the *flow*. One particular advantage is illustrated in the following way: if the transfer rate for a certain flow is multiplied by 10 then, with a packet-level simulator, the processing time is also multiplied by 10 as there are 10 times more packets to simulate; whereas the simulation speed is not affected for a flow-level simulator, as there is still only one flow to simulate (the only change is that the values are going to be 10 times bigger in the equations that model the *flow*).



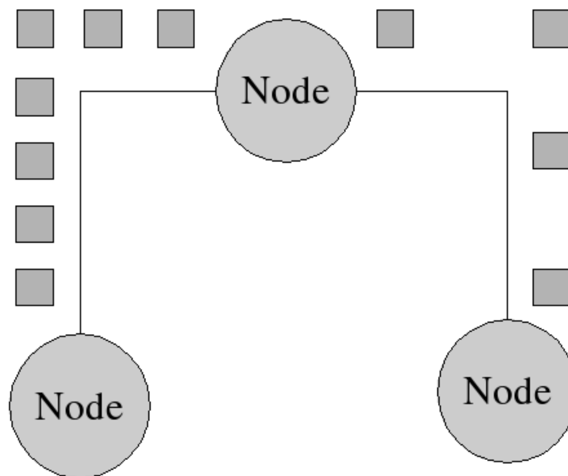


Figure 2.3: Packet-level network simulation

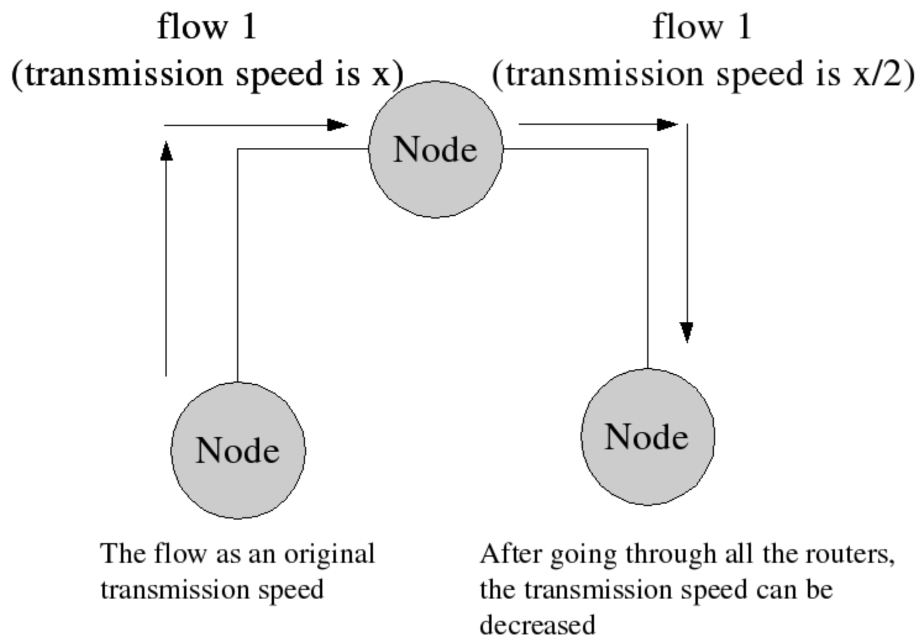


Figure 2.4: Flow-level network simulation

2.4.2 Packet-level simulation and the *ns* network simulator

ns (network simulator) [11] is very well known in the network simulation community. It offers a lot of possibilities in term of types of networks that can it can simulate, and it has been around long enough to be thoroughly tested and validated. It began as a variant of the REAL network simulator [10] in 1989 and has evolved substantially ever since.

ns provides a framework that allows researchers to extend it easily. The simulator works at a *packet level*, which requires quite a lot of processing power, but as a counterpart allows it to simulate network traffic accurately, makes it easier to simulate the interactions of multiple protocols, and allows for the possibility of interacting with real-world networks. However, the need for a fast simulation speed is taken into account by the presence of different mechanisms which make the simulation faster but at the cost of accuracy (for example replacing routing messages by centralized routing). All these mechanisms may be enabled or not depending on the goals of the particular simulation, the expected level of precision and the desired execution speed.

Even though it is not directly linked to the fluid-model that is studied in this dissertation, *ns* has an important role to play in the work described herein because it is a very usefull tool for validation purposes. Indeed, it can be used to:

- Evaluate the simulator to be implemented in terms of *correctness*. Because *ns* is probably the most widely used and validated tool in the network simulation research field, it is totally adapted for validation. By simulating the same scenarios with *ns* and the implementation of the *model* studied in this dissertation, it should be easy enough to find out if the results are correct and to establish how accurate the *model* is.

- Evaluate the simulator to be implemented in terms of *performance*. Normally, a flow-level simulator should be at least as fast as *ns* to compute the result for the same scenario. In theory, it should actually be a lot faster, but the multiple optimizations that have been made in *ns* over the last few years [11] are expected to easily compensate the speed gain compared to a code that was written by one individual in a few weeks. However, the advantage should normally rest with the *flow-level* simulator when the complexity of the network to be simulated increases, because of the economies of scale to involved. In particular, the execution time

should be longer in *ns* when the link capacities increase as the number of packets increases; whereas it should remain constant in the *flow-level* simulation as the number of *flows* is constant.

Finally, because *ns* was intensively used during the conception of *Random early detection* (RED), it is certainly the best tool available to validate a RED implementation, such as the one in the *flow-level* simulator described in this work. Indeed, early work on RED began on an ancestor of *ns*, and is is now present in the simulator as a standard component.

2.5 Flow-based simulation of TCP/IP

2.5.1 A fluid-based model for network simulation

Because the Internet both growing in size and evolving very rapidly, researchers have to invent new methods to simulate its activity. It has been quite a few years since the processing capacity of regular packet-level network simulators was sufficient to accurately simulate the behavior of a network like the Internet. The implementation of a higher-level simulator is a good alternative, as such a simulator should be able to perform much faster simulations than a packet-level one.

A basic model for flow-level simulation

One solution, proposed in [13], models TCP data *flows* as a *fluid*. The model uses *Poisson Driven Stochastic Differential Equations* to represent the traffic, and includes differential equations to model an AQM policy [5] (the AQM policy that is used is RED [4]).

As stated in [14], the following equation assumes that the network is represented as a directed graph $G = (V, E)$, with V a set of routers and E a set of links. To each link $l \in E$ is associated a capacity C_l and an AQM policy defined by its discarding probability function $p_l(t)$. N classes of *flow* are transported on the network, and each *class* i contains n_i *flows*. All the flows that belongs to a given class share the same characteristics (same route, same propagation delay and same window size).

Note that in the context of the *flow-level* simulator the words *flow* and *class of flow* often refer to the same notion. Indeed, a *class of flow* is just a collection of *flows* that share the exact same characteristics, and thus can be processed as one unique *flow* whose transmission rate is the sum of all the *flows* that are part of the class.

For a given router, Misra et al. [13] describes the characteristics of TCP *flows* this way (for each variable X, the notation X_i refers to the value related to a given *flow* i):

- The *round trip time* for a *flow* of class i is denoted $R_i(t)$ and defined by:

$$R_i(t) = a_i + \frac{q(t)}{C} \tag{2.3}$$

  where a_i is the (fixed) propagation delay, $q(t)$ is the queue length, and C the transmission capacity (and thus $\frac{q(t)}{C}$ is the queueing delay).

- The *window size* for a *flow* i at the time t is called $W_i(t)$ and satisfies the following differential equation:

$$\frac{dW_i(t)}{dt} = \frac{1}{R_i(t)} - \frac{W_i(t)}{2}\lambda_i(t) \tag{2.4}$$

  where $\lambda_i(t)$ is the loss rate experienced by the *flows* of class i.

- $q_l(t)$ represents the *queue length* for each queue l, N_l being the set of flows that go through the queue, and C_l the capacity of the associated link:

$$\frac{dq_l(t)}{dt} = -1(q_l(t) > 0)C_l + \sum_{i \in N_l} n_i A_i(t) \tag{2.5}$$

  where $1(x)$ takes the value 1 if x is true, and 0 otherwise. $A_i(t)$ is the expected sending rate of the *flow* and is linked to the window size by the equation $A_i(t) = \frac{W_i(t)}{R_i(t)}$.

A topology-aware refinement of the model

In [13], the above *model* is extended to the whole network simply by introducing a binary matrix A, where the rows represent the *flows* and the columns represent the routers. If the *flow* i goes though the router j, $A_{i,j} = 1$, otherwise $A_{i,j} = 0$. This representation makes it easy to generalize the equations to the whole network, but has

the disadvantage of ignoring the topology (the order in which the routers are crossed is not taken in account).

[14] overcomes this problem by refining the *model* to include information about the exact path followed by the different *flows*. This is done by introducing the ordered sets $F_i = (k_{i,1}, ..., k_{i,m'_i})$ and $I_i = (j_{i,n'+1}, ..., j_{i,m_i})$, that are respectively the queues (and therefore links) traversed by the data (initial path) and the acks (return path) for a *class* i. The whole path E_i for the *flow* is also defined as $E_i = F_i \cup O_i$.

To characterize the variation of the *flows* as they traverse the links, [14] also introduces the following two quantities:

- $A_i^l(t)$ is the arrival rate at the queue $l \in F_i$ for the *flows* of class i
- $D_i^l(t)$ is the departure rate from the queue $l \in F_i$ for the *flows* of class i

Note that in both cases only the queues included in F_i are taken in account. The reason for this is that the traffic generated by the acks is ignored as it is considered to be negligible. As a consequence, O_i is not used for the computation of the departure and arrival rates, but is present in the model anyway to evaluate the *round trip time* for each *class of flow*.

The two quantities that were introduced above can be evaluated as follows:

- The *departure rate* is the same as the *arrival rate* when the queue is empty. When the queue size $q_l(t)$ is not zero, the capacity is divided among the *flows* proportional to their *arrival rates* at the link. Let d_l denote the delay experienced by traffic starting from l at time t, then $d_l = \frac{q_l(t-d_l)}{C_l}$ and D_i^l can be expressed as:

$$D_i^l(t) = \begin{cases} A_i^l(t) & \text{if } q_l(t) = 0 \\ \frac{A_i^l(t-d_l)}{\sum_{j \in N_l} A_j^l(t-d_l)} C_l & \text{if } q_l(t) > 0 \end{cases} \tag{2.6}$$

- At the first queue it goes through, the *arrival rate* for the *class* i simply is the original sending rate for that *class*. For the other queues, it is the departure rate from the previous one after a delay that corresponds to the propagation time along the link. This behavior is summarised by the following equation:

$$A_i^l(t) = \begin{cases} A_i(t) & \text{if } l = k_{i,1} \\ D_i^{b_i(l)}(t - a_{b_i(l)}) & \text{otherwise} \end{cases} \tag{2.7}$$

The equations governing the system then have to be updated to take the new information in account:

- The *window size* is described by:

$$\frac{dW_i(t)}{dt} = \frac{1(W_i(t) < M_i)}{R_i(t)} - \frac{W_i(t)}{2}\lambda_i(t) \tag{2.8}$$

  where M_i is the maximal window size. The term $1(W_i(t) < M_i)$ prevents the *window size* from increasing when it is equal to the maximum *window size* for the *flow*.

- The equation representing the *queue length* is updated so that, for each *class*, it takes into account the arrival rate A_i^l at the queue, and not the global, expected sending rate for the *class* as in the previous version of the model:

$$\frac{dq_l(t)}{dt} = -1(q_l(t) > 0)C_l + \sum_{i \in N_l} n_i A_i^l(t) \tag{2.9}$$

Introducing AQM into the model

In addition to studying the *fluid-based* simulation of TCP/IP networks, one of the main goals of this thesis is to implement *active queue management* (AQM) techniques in the simulator. The first AQM scheme to be implemented is *random early detection* (RED), which is described by equation 2.1 and illustrated by figure 2.2. Moreover, the average queue length $x(t)$ needed to evaluate the RED function is described by equation 2.2. The two previous equations completely describe RED's behaviour and so it is easily implemented.

Other *active queue management* controllers, like PI [3] are to be implemented in the simulator, but will be considered as extensions and described later.

Solving the model

The simulation can be processed by putting together all the equations given in this section. According to the classification of simulators described in [8], the one to be implemented will be based on a *stochastic fluid model* and will use *discrete-event* (equivalent to *time-stepped*) simulation.

The differential equations can be solved using the Runge-Kutta algorithm [17], as described in [14]. The time has to be discretized (*time-stepped* simulation), and at each step the whole set of variables has to be updated according to the equations and using the solver. The order of the steps and the operations to be performed are described in detail in [14] and summarised by figure 2.5.

2.5.2 An implementation of the model

An implemention of the *model* as described above was carried out by D. Bruno, intern in the computer science department of Trinity College Dublin in 2004. The details of this implementation are given in [18]. It follows the steps presented in figure 2.5.

2.6 Conclusion

In this chapter, we presented the relevant *TCP/IP* and *network simulation* terminology to understand the rest of the document, along with the concept of *active queue management* and a *fluid-based model* that enables us to perform high-speed simulation. The next chapter will give more details about the implementation of the *model* and the first tasks that were accomplished on this implementation during this work.

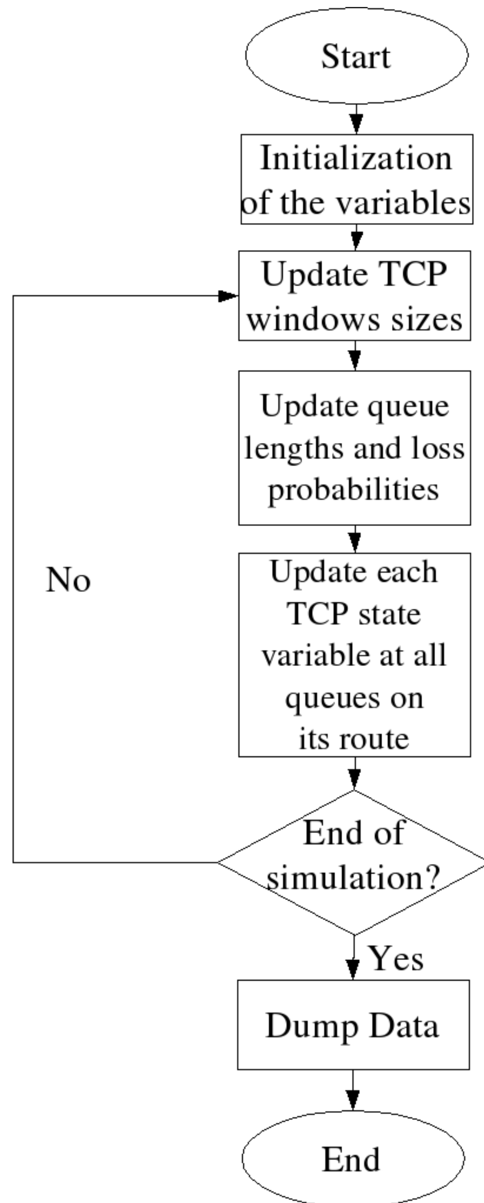


Figure 2.5: Flowchart of the operations to be performed during the simulation

Chapter 3

Optimization and validation of the simulator

3.1 The simulator

The work described in this dissertation builds on a *flow-level* network simulator implemented at Trinity College Dublin in 2004. This simulator [18] is based on the model presented in the first part of the dissertation, but was coded over a short period of time, and consequently was not fully optimized and did not take into account all the design improvement suggested in [14]. Moreover, even though it was totally functional, the application had not been completely validated. Indeed, apart from the Runge-Kutta solver included to solve the differential equations on which the *model* is based, none of the components functioned in the manner specified. The first part of this work was to find ways in which the software could be optimized to improve the execution speed. It was then necessary to validate the correctness of the results given by the implementation of the *model*.

The simulator was written in *Java*, and follows most of the implementation suggestions given in [14], apart for a few modifications that made the coding easier. Figure 3.1 shows a simplified version of the class diagram, that only represents the main classes (without their properties or methods) and the links between them, as well as the main class and its properties. Note that this diagram includes a few of the changes that were made on the original code to make it more object-oriented, as disscussed in the next

section. In particular, the *Node* class was not present in the original version and the relationships between *Link* and *SingleLinkFlow* were represented by references numbers corresponding to one given node and stored as an integer in the different classes.

3.2 Optimization of the simulator

3.2.1 What to optimize?

Making the code clearer to read and the simulator easier to use

Even though it can not truly be considered as optimization, the first step taken before working on performance issues was to make the code clearer to read, and to improve the input interface of the simulator. Indeed, the original code followed the suggestions of [14] very closely which sometimes lead to situations where the code didn't really respect the object-oriented programming paradigm. Moreover, making some parts of the code easier to read was useful in that it made it easier to identify possible optimisations.

Half of the code in the main class was dedicated to reading the network topology and the simulation parameters from an ASCII file. To make the code easier to read and the simulator easier to use, it was necessary to:

1. Change the input file format. Even though the original version had the advantage of being able to represent a rather complicated network in a very compact way, it was not intuitively understandable in the sense that it was just rows and columns of numbers (see figure 3.2) and therefore rather obscure. This increased the number of errors when editing or writing a configuration file.

2. Move the configuration reading code from the main class, and simplify it. The purpose of this code is not obvious to someone reading it for the first time and it was not necessary to have it in the main class.

To address the two above issues, the following changes were made:

1. A new input format was defined, based on XML. This format is a lot easier to read (see figure 3.3), and as it is XML based, the code to parse it is easier to understand and takes advantage of existing libraries.

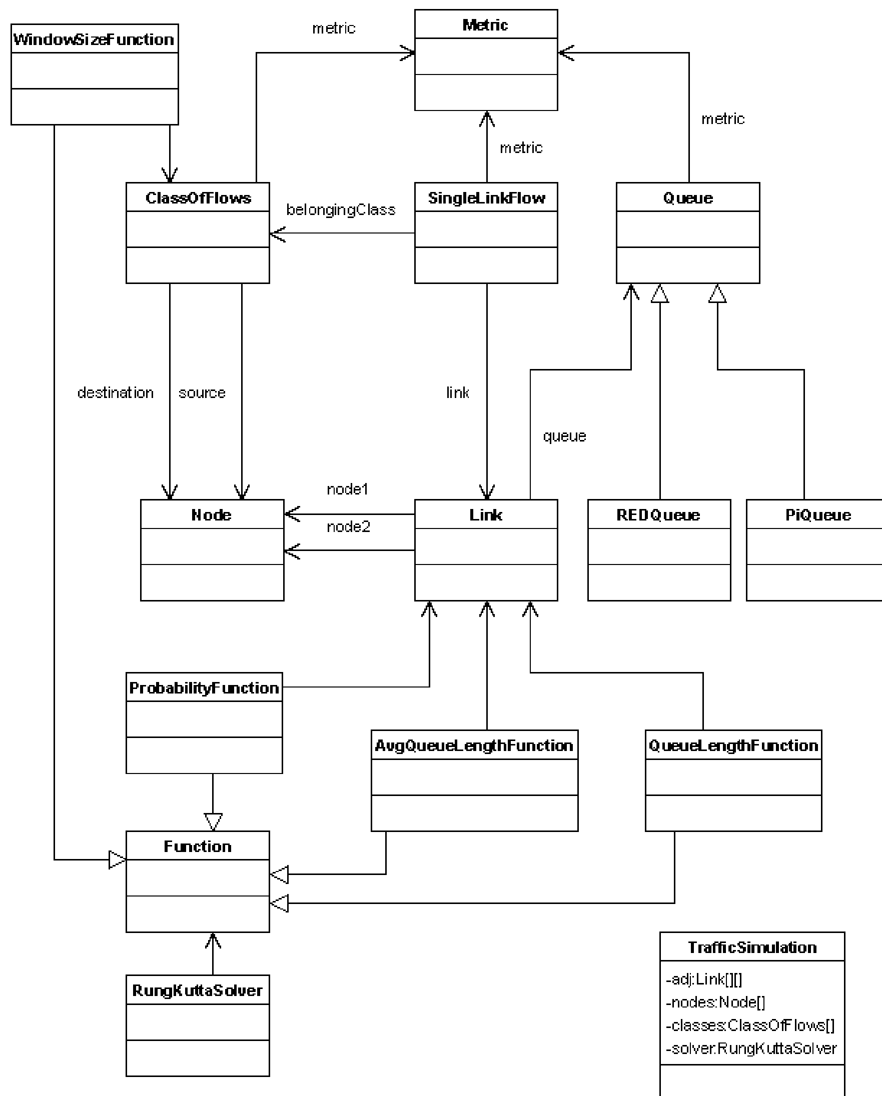


Figure 3.1: Class diagram for the network simulator

```
NumberOfNodes: 4
0 1 20 0.01 RED 20 50 0.02 0.005 0.5 100 RED 20 50 0.02 0.005 0.5 100
1 2 20 0.01 RED 20 50 0.02 0.005 0.5 100 RED 20 50 0.02 0.005 0.5 100
2 3 20 0.01 RED 20 50 0.02 0.005 0.5 100 RED 20 50 0.02 0.005 0.5 100
```

Figure 3.2: Defining a few links with the old configuration file format

2. A new generic class called *ConfigReader* was created. This abstract class specifies the functions to be implemented to both read a configuration file and transmit the information to the main class, which just has to instantiate a *ConfigReader* instead of implementing the parsing of the file. In order to maintain compatibility with the previous files, two specialization of the *ConfigReader* have been implemented, on that can read the old ASCII format, and one that reads the XML-based format.

```
<link node1="0" node2="1" capacity="20" delay="0.01">
    <aqm
        direction="forward"
        type="red"
        tmin="20"
        tmax="50"
        alpha="0.02"
        delta="0.005"
        pmax="0.5"
        maxlength="100">
    </aqm>
    <aqm
        direction="backwards"
        type="red"
        tmin="20"
        tmax="50"
        alpha="0.02"
        delta="0.005"
        pmax="0.5"
        maxlength="100">
    </aqm>
</link>
```

Figure 3.3: Defining a link with the new configuration file format

Making the code more efficient

The optimization of the code was done by trying to find the most efficient optimizations, while respecting the application's design. The two main concerns were to improve:

- The *simulation speed*, with a particular focus on *scalability*. The simulation time for a relatively small network was acceptable, but not as short as expected. However the simulation time increased rapidly with the complexity of the network and this needed to be addressed.

- The *memory usage*, again focusing on *scalability*. As the size of the network got bigger, or the simulation time increased, the amount of memory needed was such that it was simply not possible to run the simulation. As a consequence, improvements in term of speed would have been worthless without improvement in term of memory usage. Moreover, being able to simulate large-scale networks is one of the original goals of *flow-level* simulation techniques.

A number of "basic" optimizations were carried out. These did not involve important changes to the code but brought about significant improvements. Because of their straight-foreward nature, these changes won't be detailed, however, the main aspects which led to optimization were:

- Abusive instantiation of objects. In some cases, the objects needed to compute the simulation data (especially the equation solvers) were instantiated at each step of the simulation. This was done because some parameters needed to instantiate the object were assumed to be only available at that time, but all the information needed was available at the start of the simulation and therefore the objects can be instantiated only once and reused at each step, which saves CPU power. The result of this optimization is not obvious on a small scale, but is clearly evident when the network to be simulated gets bigger.

- Access to the data. It appeared that one of the slowest operations was data access. The model sometimes requires access to past data values of a data (e.g. queue length) for the evaluation of certain functions. These values are stored in *Metric* objects. Because this data access occurs frequently, and because the values that need to be accessed usually are at time $t - timestep$; optimizing the data access method in the *Metric* class with this information in mind drastically decreased the amount of time needed to simulate a given network.

- Output of the simulation results. The conversion of the *float* values stored in the *Metric* objects needed to write them into file was a very slow operation

in Java, and couldn't be optimized. A proposed solution was to introduce a *dumpPrecision* parameter, to specify how much data has to be dumped into the file. For example, if the simulation step is 0.001 second, with a dump precision of 1/100, the value of the metrics are stored in the output file only every 0.1 second. This reduces the amount of information in the output file, but doesn't decrease the simulation precision. This parameter should be set to a relevant value depending on what the simulation results are to be used for. For example, in order to generate a graph for a 10 second simulation with 0.001 second time step, a file dump precision of 1/100 is more than sufficient and will make the simulation a lot faster.

- On the fly output of the results. As suggested in [14] and illustrated in figure 2.5, the original simulator stores the evolution of the value of the different metrics during the whole simulation process, and dumps all the data into files when the simulation is completed. This method it totally unsuitable for a large scale network, or even for a small network when the simulation time is long, as a huge amount of data has to be stored in memory. The optimized version of the simulator was modified so that the data that is no longer needed by the simulator to compute the current values of the metrics is automatically dumped to the corresponding output file and erased from memory as the simulation is running.

3.2.2 Evaluation of the optimizations

Tests run with a small size network (3 nodes, 2 links, and 4 classes of flow) show that the execution time is 10 times faster for that particular network, and the amount of memory needed is about 10 times smaller. These results are satisfying, and the difference is expected to be bigger with larger-scale networks. Because evaluation on a bigger scale was non-trivial (the original simulator can not really handle large-scale networks because of its high requirements in terms of available memory), the evaluations presented in this section were done with simple scenarios. These scenarios are designed to evaluate the impact of the different parameters on the simulation speed for both simulators, and to lead to some conjectures on how they would compare with bigger networks.

Experimental setup

The experiments were done for the network represented in figure 3.4 and with the parameters specified in table 3.1. Unless differently specified, all the experiments in this dissertation will use this same setup. For each set of experiments, either the number of *nodes*, the *simulation time*, or the number of *flows* was changed to one of the four different values to evaluate the evolution of the simulation speed and memory usage with both the original and the optimized simulator. When considering the number of nodes, the previously described network was used with one unique flow going through a succession of nodes. However in that case the number of nodes varied from 2 to 16. For the simulation time, the topology remained unchanged but the simulation time was varied from 2 to 16 seconds. Finally, to evaluate the impact of the number of classes of flow, certain classes were added to the original network (a maximum of eight).

Note that the measured execution time includes the time needed to read the configuration file, the simulation time, and the time to write the output files. The XML configuration file reader used by the new version of the simulator seem to be a little slower that the ASCII one, but this will be neglected as the difference is rather small, and the results lead to the conclusion that the new simulator is faster and scales a lot more easily.

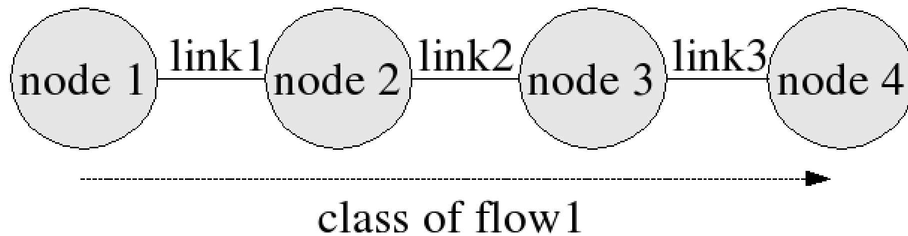


Figure 3.4: The network topology used to evaluate the performance optimizations

Performances

Figures 3.5 to 3.7 present the influence of the number of nodes, the simulation time, and the number of flows on the CPU time needed to complete the simulation.

Figure 3.5 shows that for both versions of the simulator the execution time is proportional to the number of nodes. Moreover the execution time is faster for the

Computer	Intel Pentium M 1.6 Ghz, 512MB
Operating system	Linux 2.6.10
JRE	Standard Edition 1.5.0
Simulation time	15 seconds
Simulation precision	0.001
Data dump precision	1/100
Links capacities	20 packets/second
Links delays	0.01
AQM types	RED
t^{min} (for each RED queue)	20
t^{max} (for each RED queue)	50
alpha (for each RED queue)	0.02
delta (for each RED queue)	0.005
p^{max} (for each RED queue)	0.5
Max length (for each RED queue)	100
Number of flows per class	1
Max window size for each flow	1200

Table 3.1: Technical data and parameters used for the performance evaluation

optimized version, and more importantly this version scales better when new nodes are added (the execution time is multiplied by 4 when the number of nodes is multiplied by 8, whereas it is multiplied by 7 for the original simulator).

Figure 3.6 doesn't show execution times that are directly proportional to the simulation time. This result is not expected as the number of operations to be performed should be proportional to the simulation time, and there is not supposed to be any "memory effect" that would make the computation of the latest values fastest. As the execution times are quite small for this experiment, this is very likely to be caused by the influence of the configuration reading code. Indeed, in spite of the fact that precaution were taken to avoid the effect of file caching during the experiments, the reading of the input files doesn't exactly take the same amount of time each time the simulator executes the same simulation, and this difference may have caused the value not to be exactly proportional. However, the results clearly show that the optimized version is faster and scales better as the execution time increases. The improvement in term of scalability is limited if the one second time needed to read the configuration is subtracted from the execution times. This is not disappointing, as there is no reason

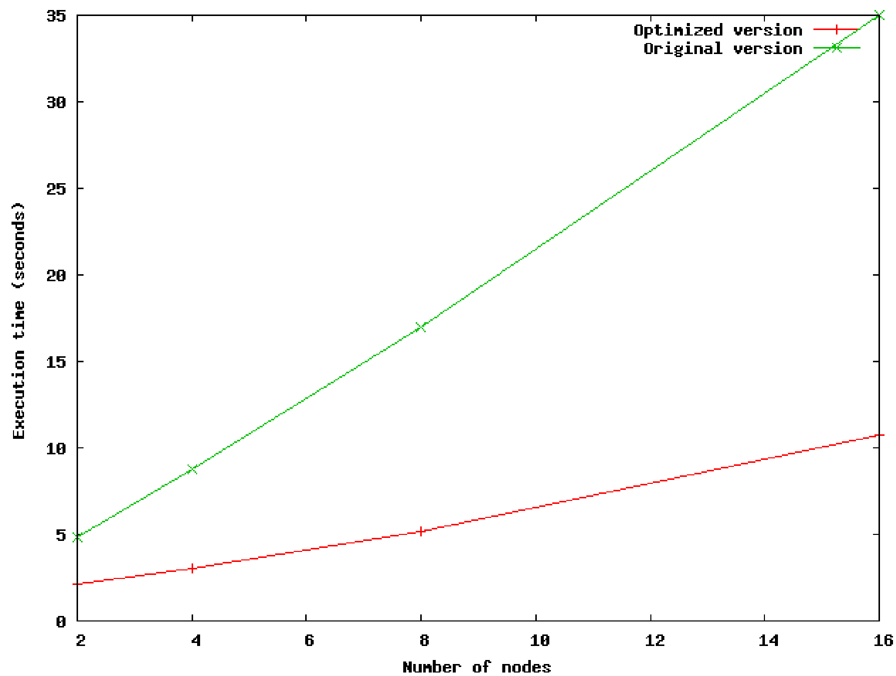


Figure 3.5: Influence of the number of nodes on the execution time

for the new version to scale better as the simulation time increases (if the simulation time doubles, the amount of operations logically doubles in both cases).

Figure 3.7 give valuable information as it shows a linear increases in the execution time as the number of flows increases for the optimized version of the simulator. In the case of the old version the evolution is linear at first, but then grows rapidly for the 8 flow test. Without taking this into account, the new version already scales a lot better. But the difference is even greater when ones understands why the last value is higher than expected for the original simulator. For the first tests, only simple flows were added to the network (going though only one node), whereas the network was too small to add such flows for the final test. As a consequence, the later flows follow a more complicated path and therefore go through more nodes. It is very interesting to note that these flow don't require extra computation time with the optimized version, whereas the original one requires more time to treat them. In other words, the optimized version is not significantly influenced by the complexity of a flow. This can be explained by the fact that the extra work needed to process these flows is mostly

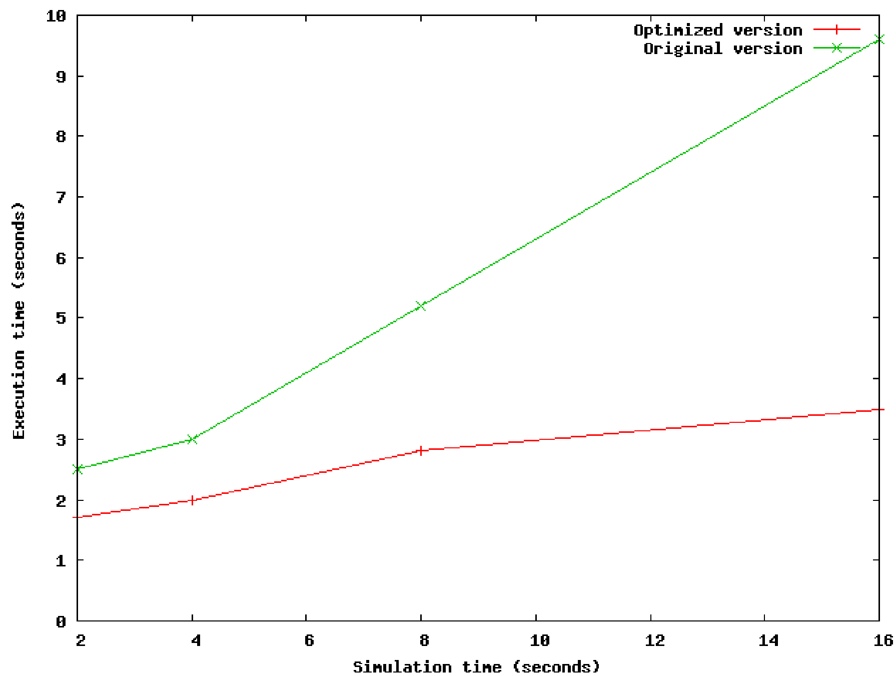


Figure 3.6: Influence of the simulation time on the execution time

required to manage access to the previous queue length values and to write data in the output files, and both these operations were identified as critical and optimized as much as possible.

In conclusion, the new version of the simulator is definitely faster and more scalable. The differences observed in the previous tests don't seem to be gigantic, but show that in the case of big networks, especially if there are numerous flows that go though a lot of nodes, the new version will be a lot faster.

Memory usage

Figures 3.8 to 3.10 present the influence of the number of nodes, the simulation time, and the number of flows on the amount of memory needed to complete the simulation.

Figures 3.8 and 3.10 show that the optimized simulator always needs less memory than the original one when the number of nodes or flows increases. The more flows or nodes present, the bigger the difference. This was expected because metric data is progressively written to external files, while the previous version of the simulator was

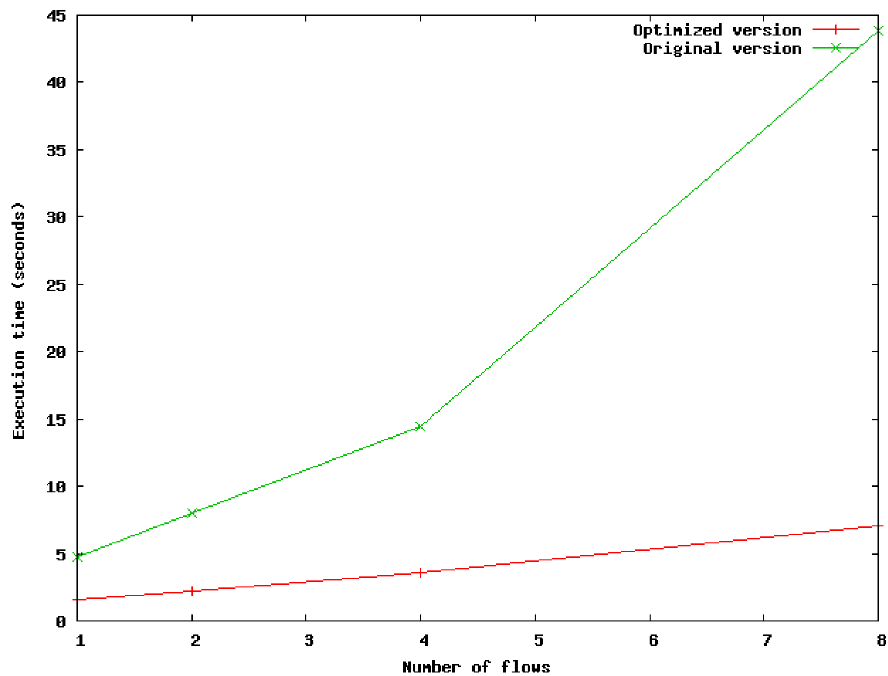


Figure 3.7: Influence of the number of flows on the execution time

storing that information in memory.

Figure 3.9 gives even more interesting information about the optimizations. As expected the amount of memory required by the original simulator gets very large for the longest simulation time (and it quickly becomes impossible to run any simulations), whereas the value remains tractable for the optimized one. However, after a certain point (from the graphs, this is reached at a simulation time between 3 and 4 seconds), the required amount of memory should not increase as all the values that are no longer needed are dumped to a file on the harddrive. The fact that this is not totally the case suggests that either some metrics are not taken into account by the on-the-fly dumping system, or that a memory leak is present somewhere in the code. Because the impact of this is limited within the scope of this study, no further research was carried out about this problem, but it should be looked at for future work involving this simulator.

In terms of memory usage, the new version is definitely better and solves the problem of ever-increasing memory usage as the simulation time increases. However, even though this aspect is now well optimized, other, more refined solutions could be found

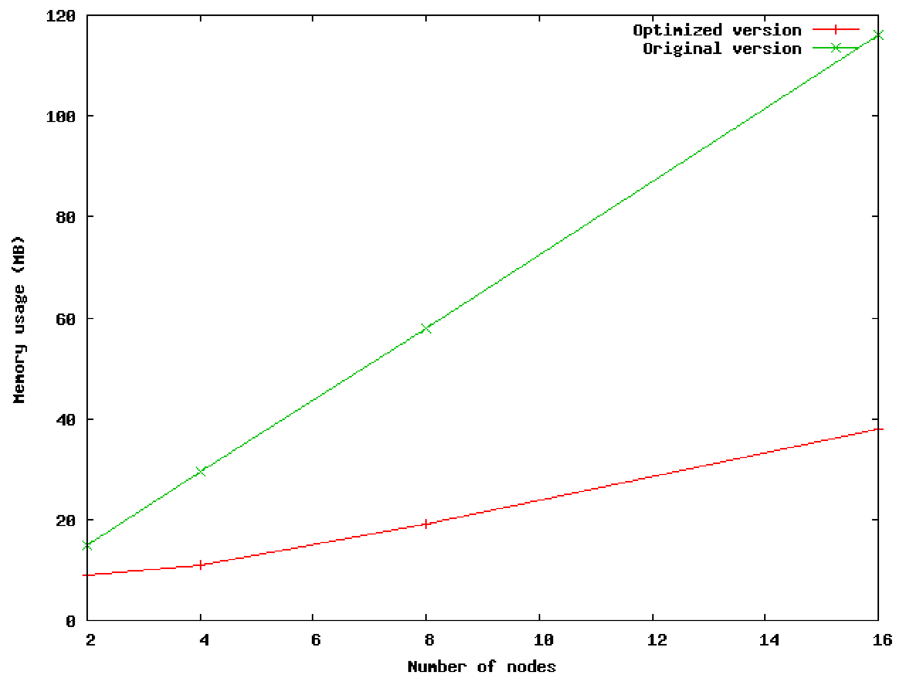


Figure 3.8: Influence of the number of nodes on the memory usage

to reduce the memory usage even further in the future.

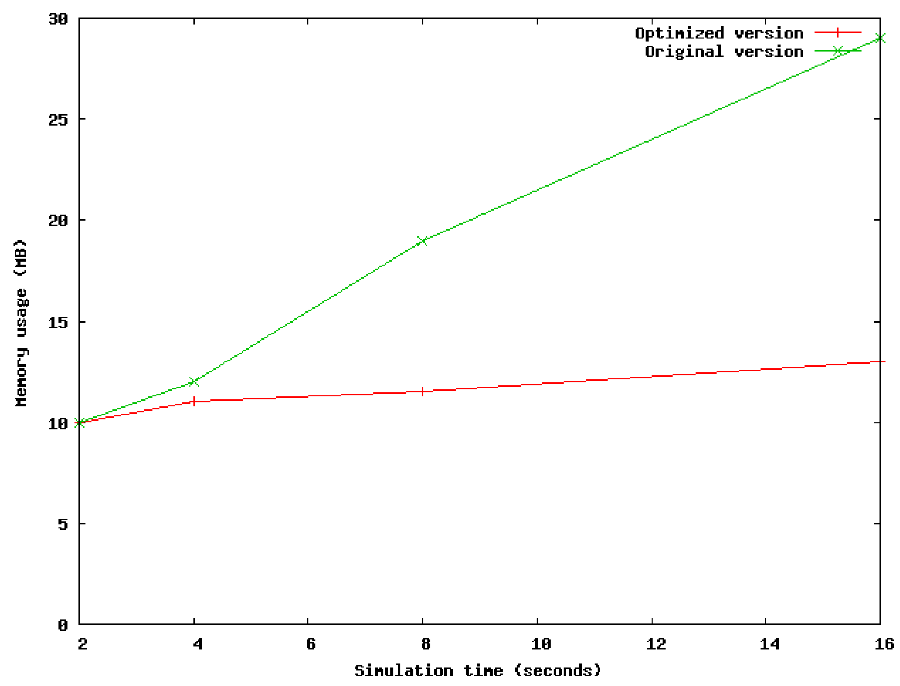


Figure 3.9: Influence of the simulation time on the memory usage

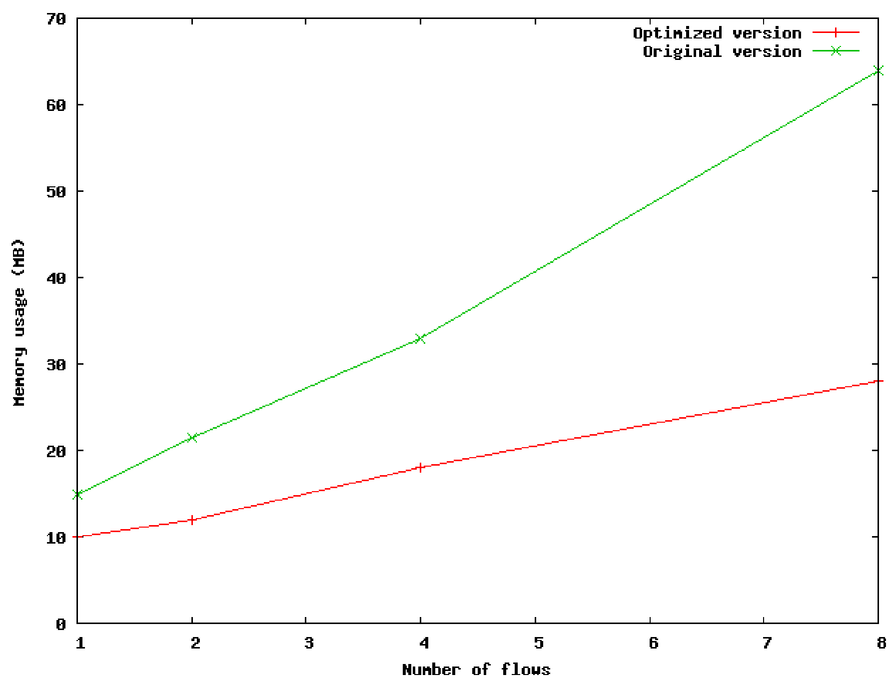


Figure 3.10: Influence of the number of flows on the memory usage

3.3 Validation of the simulator

After optimizing the code, a lot of effort was put into thoroughly debugging it. Indeed, some of the results obtained were clearly incorrect and it was an absolute essential to fix these problems. The validation methods used to prove the correctness of the implementation are explained in this section.

3.3.1 Validation with a simple predictable network

The purpose of this test was to validate the behaviour of the simulator with the most basic possible network, so that the results obtained are easily predictable and can be compared to the simulation results.

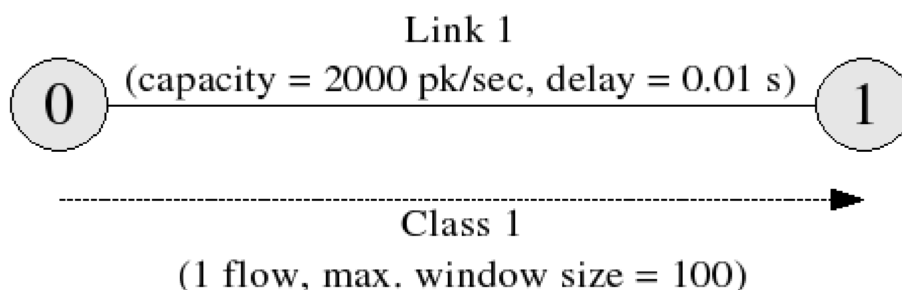


Figure 3.11: The network used for analytic result based validation

The network used for this test is illustrated in figure 3.11. It simply consists of two nodes and one link, which is only used by a single flow. The values used for the different parameters are given in figure 3.11 and the RED parameters are the same as those specified in table 3.1. Finally, the simulation time is 10 seconds and each time step is 0.001 seconds long.

Ignoring the queueing system, the flow is supposed to saturate the link when its windows size reaches 40. Indeed, the round-trip time is 0.02 seconds and the capacity is 2000 packets/second, which means that the maximum number of packets that can be transmitted in one time unit without any packet drop is $2000 * 0.02 = 40$. The window size is then supposed to increase by 1 every timestep until it reaches 40 (TCP window *additive increase*), and then be divided by 2 (TCP window *multiplicative decrease*), start increasing again, and so on.

If the queueing system is added to the system, the window size will actually increase as the queue gets filled, until the RED policy decides that the queue is too long and starts dropping packets. At that time, the TCP window size will be divided by 2 as explained before. The last piece of information to fully specify the theoretical behavior of the system is the lower RED drop probability threshold. RED starts dropping packets when the average queue length exceeds the t^{min} parameter, which is set to 20 for our scenario. To determine precisely when the first packet will be dropped, it would be necessary to take into account the technique used to evaluate the average queue size (EWMA) and the fact that not all packets get dropped before the queue length exceeds t^{min}. However, given the nature of the traffic (one unique flow increasing its window size regularly), it is clear that packets will start being dropped almost straight after the queue exceeds 20 packets.

To summarise, the expected behavior of the flow is to increase its window size until it reaches 60 (40 to saturate the link plus 20 to fill the queue without RED dropping packets), then back-off and divide the window size by 2 to a value of 30, start increasing it again, and so on. The queue length is expected to start growing after a short period as the link gets saturated, attain a value that is a little higher that 20, get smaller as the flow backs-off and reduces its transmission speed, and then increase again, and so on.

The results of the simulation are given by figure 3.12. The evolution of the metrics is close to the expected behaviour of the system, except for a few points that can be easily explained:

- The window size takes values up to 70, and not 60 as predicted. This can be explained by the fact that, as illustrated by the graphs representing the actual queue length and the average estimated queue length, after the queue exceeds 20 packets it takes some time for the average queue length to exceed 20 and trigger the dropping of packets. When the average queue length takes on the value 21, the actual queue is actually almost full with 30 packets. This is the reason why the window size reaches the value of 70 (40 packets to saturate the link, and 30 to fill the queue before packets get dropped).

- When the link and the queue saturate, the window size is divided by 4 instead of being divided by 2 as expected. This can also be explained by the average

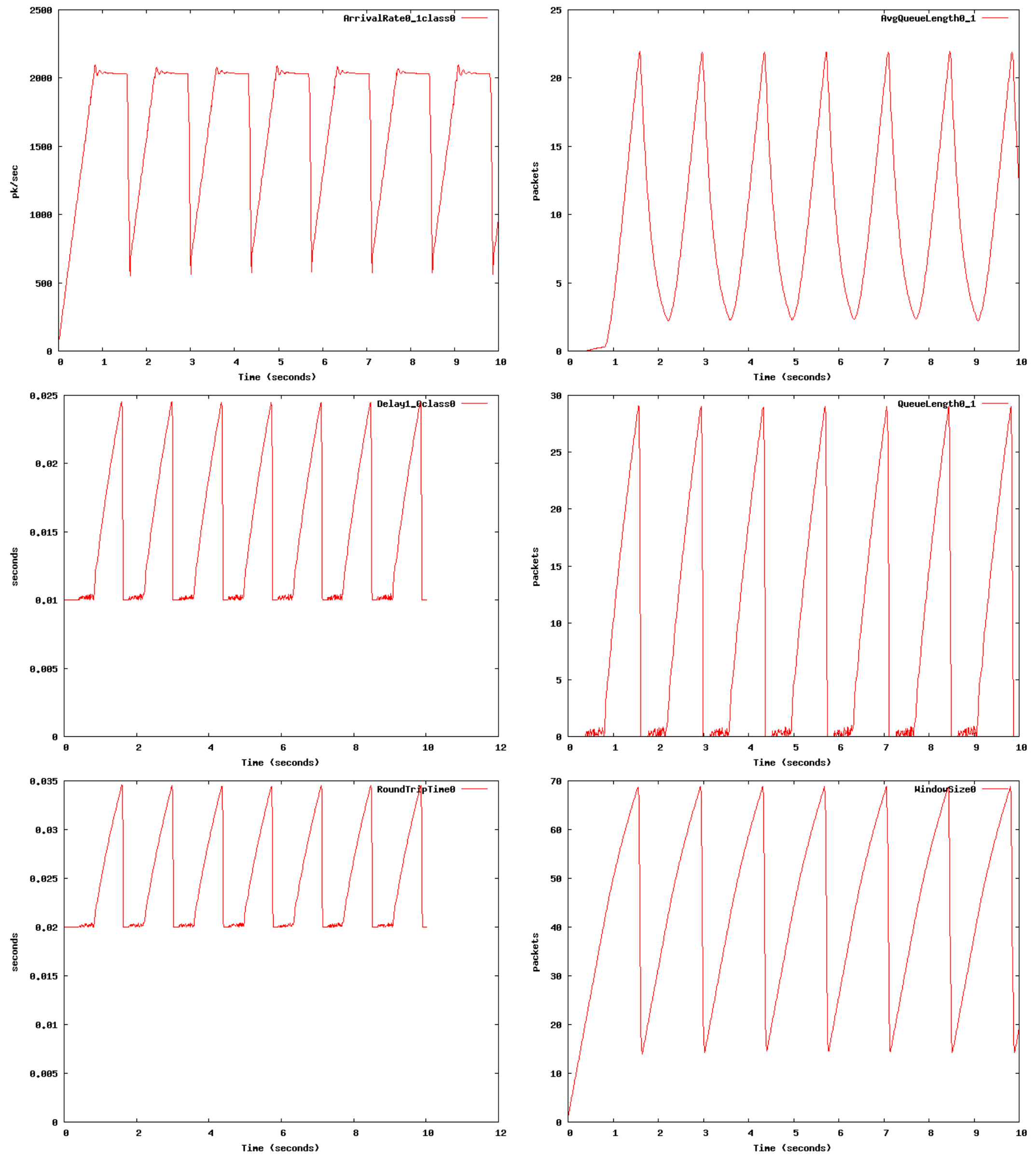


Figure 3.12: Simulation results for the analytic result based validation

queue length evaluation method. Because the queue is actually longer than the average queue length when packets start to be dropped, and because the average suffers from some latency compared to the actual value, it takes some time for the average queue length to return to a value that is lower than t^{min} after the first packet drop. For that reason, RED doesn't only drop one packet, but keeps on dropping until the average queue length is small enough. In our case, it seems that packets kept on being dropped after the first time TCP backed-off, and so it backed-off a second time, and hence the window size was divided by 2 again (and in the end the max value was divided by two twice). This was difficult to predict, but makes perfect sense upon reflection and confirms that the simulator is working well.

In the end, in spite of those few differences which were actually due to the omission of certain details in the predictions, for this small scenario, the simulator seems to behave exactly as expected. However, it has to be validated with more complex scenarios. Even this simple example showed that predicting the results is very tricky and that is why the following evaluations will be based on a comparison with other results presented in the literature or by validation against the *ns* simulator.

3.3.2 Validation against other results

Motivations

The purpose of this section is to validate the correctness of the results obtained by comparing with others from the literature on the same simulation technique. If the two outputs are close to each other, it will mean that the implementation of the model is correct. Further validation which compares results with those obtained using a completely different simulation technique were also carried out.

Please note that, as opposed to all other validation tests, the following results were obtained with a version of the simulator which included the refinements described in chapter 4. Indeed, the improved flow control presented in section 4.2 was necessary to reproduce the results presented in [14].

Validation against *Liu et al.*

The fluid-based, flow-level network simulator which was validated uses the exact same model as in [14]. The authors did not provide sufficient information to reproduce their results exactly (some of the parameters are missing). However, because the network topology is clearly defined, it was possible to reproduce the results presented in section 5.1 of the Lui et al. paper. The author recognizes that reproducing results by adjusting the input parameters is not ideal, but the original paper did not provide the necessary information. The general shape of the different graphs was the same regardless of the value of the input parameters, and adjustments were only needed to obtain values that were at the same scale.

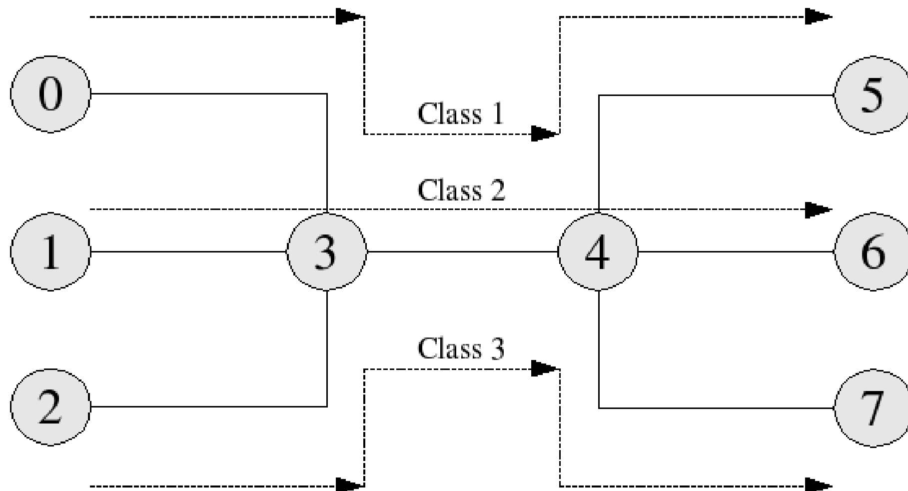


Figure 3.13: The validation network used by Liu et al.

Figure 3.13 shows the network topology used for the test, and figure 3.14 is a copy of the results obtained by Liu et al. (window size for class 1, and expected queue size at the node $3 \rightarrow 4$ link). The graphs include results obtained with *ns* and the *fluid-flow model* (FFM).

The TCP class 1 is active during the whole simulation. Class 2 starts being active at time $t = 0$ and stops at $t = 40s$. Finally, class 3 starts at $t = 70s$ and remains active until the end of the simulation. All the links have the same capacity and the maximum window sizes are big enough to saturate any of them. The bandwith of the

node 3 $\rightarrow$ 4 link will then be shared when two flows are active simultaneously, which should have an impact on the TCP window sizes of each flow and the behavior of the queue for that link.

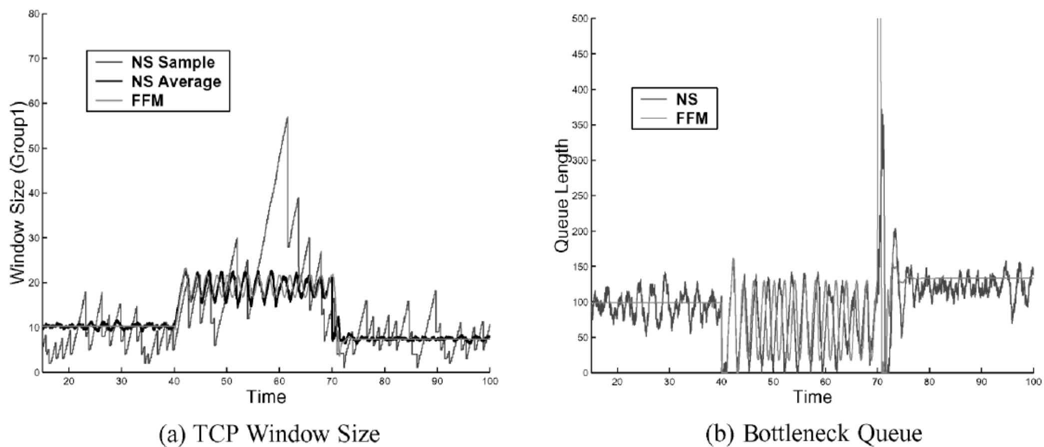


Figure 3.14: Results obtained by Liu et al. (fig. 7 from [14])

Figure 3.15 shows the values of the same metrics obtained with the flow-level simulator from this work. Even though the results are not exactly the same, they are quite close to each other and the general behavior of the network is the same. As expected, and confirmed by the Liu et al. results, the average window sizes for the first flow doubles when only that flow is in action (from time 40 s to time 70 s). During that time the queue length is globally shorter with both simulators (as well as with *ns*), even though its evolution is slightly different (which is probably simply caused by different RED parameters or a different implementation of RED).

The previous results give a good idea of the reliability of the simulator. However, it was decided to compare it in more depth with a well known and recognized packet based simulator.

3.3.3 Validation against *ns*

Correctness

ns is widely used in the field of network simulation, which is why it was a tool of choice to validate the behavior of the *flow-level* simulator. Moreover, as it uses a *packet-level*

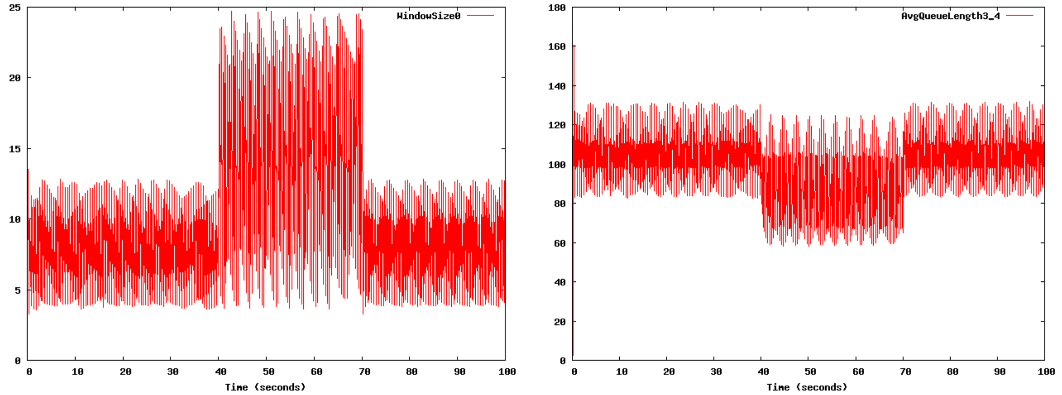


Figure 3.15: Simulation results with the Liu et al. test topology

model, it was a good way to illustrate the differences between the two techniques as regard their outputs.

The topology used to compare the simulators is illustrated in figure 3.16. It consists of a "backbone" network (nodes 1 to 3) and "network access points" (nodes 4 to 7). All the links have the same propagation delay of 10 ms, the connections between backone nodes have a capacity of 5000 pk/sec while the others have a capacity of 2000 pk/sec.

The network traffic assumes that clients are downloading file from a server located at node 0 from each access points. Each of those classes of flows has a maximum window size that is big enough to saturate any of the links and consists of one unique flow. All the links could become congested by the flows that go through them, but it is clear that, as all the flows go through it , link $0 \to 1$ will saturate first and shape the traffic for the following ones.

Figure 3.17 shows the evolution of the queue length, RED drop probability, and traffic arrival rate for that potentially congested link. The first thing to notice about the two simulators is that the results look similar but with *ns* the arrival rate seem to be more bursty. It is simply caused by the nature of the *packet-level* simulation that models each packet and therefore captures very small changes, whereas the *flow-level* simulator and the *fluid-model* behind it work differently and models a "perfect" *system* that evolves more smoothly. This difference will appear in all the results and is perfectly normal. Another difference is that with *ns* there is a peak in traffic at the very beginning that isn't present with the flow-level simulator. It can be simply

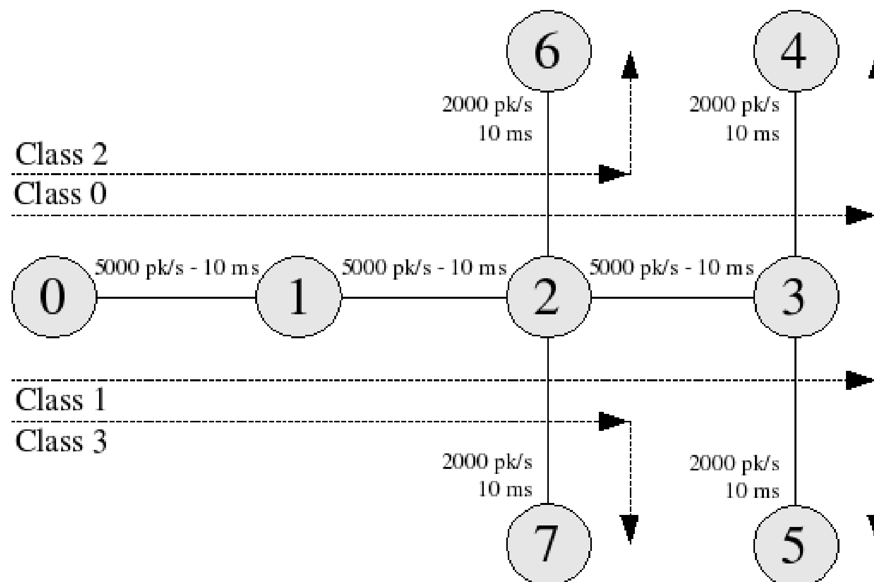


Figure 3.16: Network topology used to validate against *ns*

explained: *ns* models the TCP *slow-start* mechanism, whereas the fluid-model doesn't. Apart from those expected differences, the metrics' evolution is the same in both cases:

1. The arrival rate progresses at a regular speed as the different flows increase their window sizes.

2. Around time=6 seconds the link's maximum capacity is reached.

3. The sending-rate is maintained for a short period as the queue starts to fill.

4. Around time=6.5 seconds the average queue length reaches a value that makes RED start to drop packets (the dropping probability increases).

5. TCP backs-off and the sending rate drops to a very low value.

It is also interesting to note that the drop probability reaches a higher value with *ns*. This is probably due to different implementations of the average queue length estimation mechanism, however it has a reduced impact on the behaviour of the whole *system*.

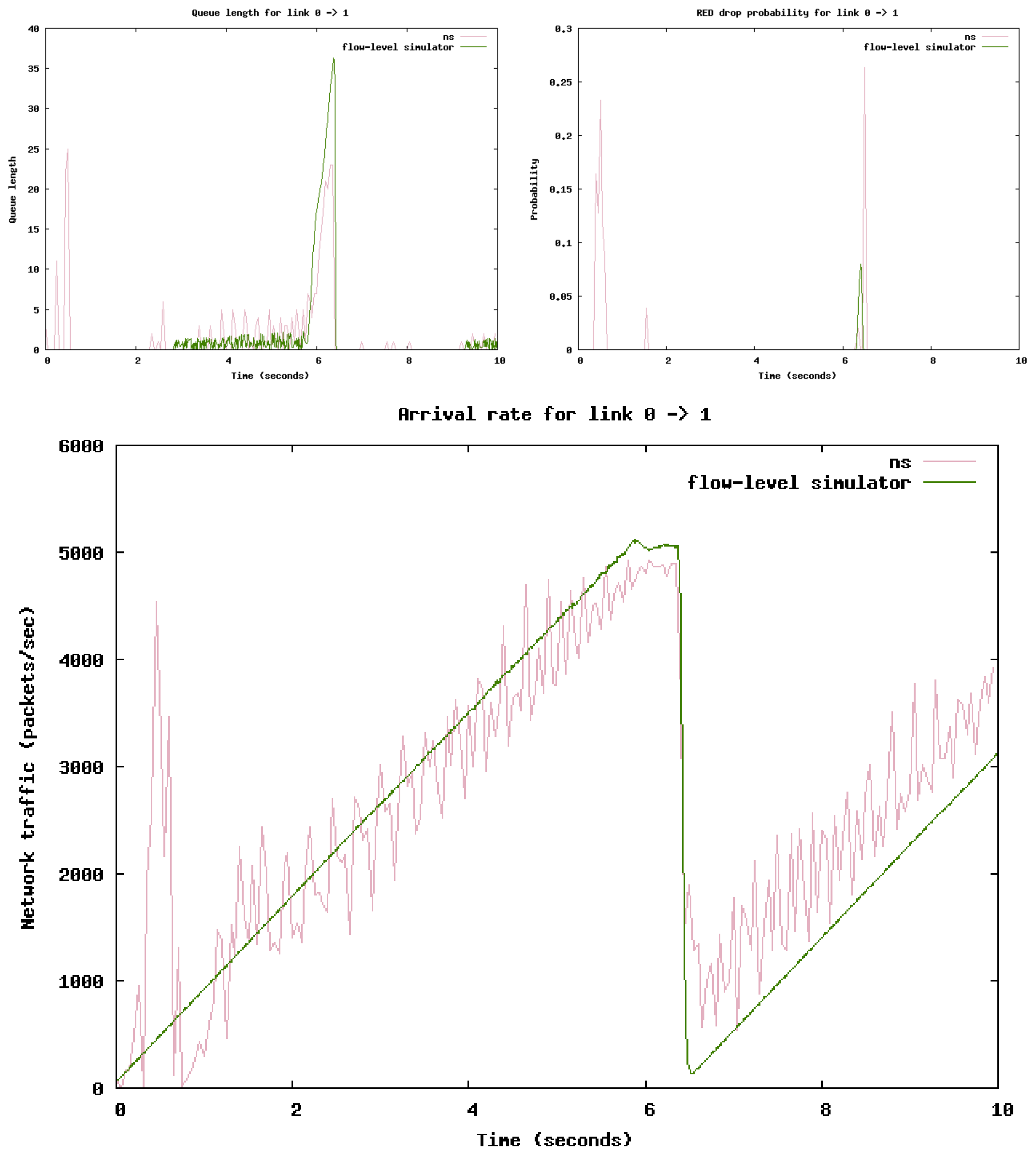


Figure 3.17: State variables for the congested link (*ns* validation)

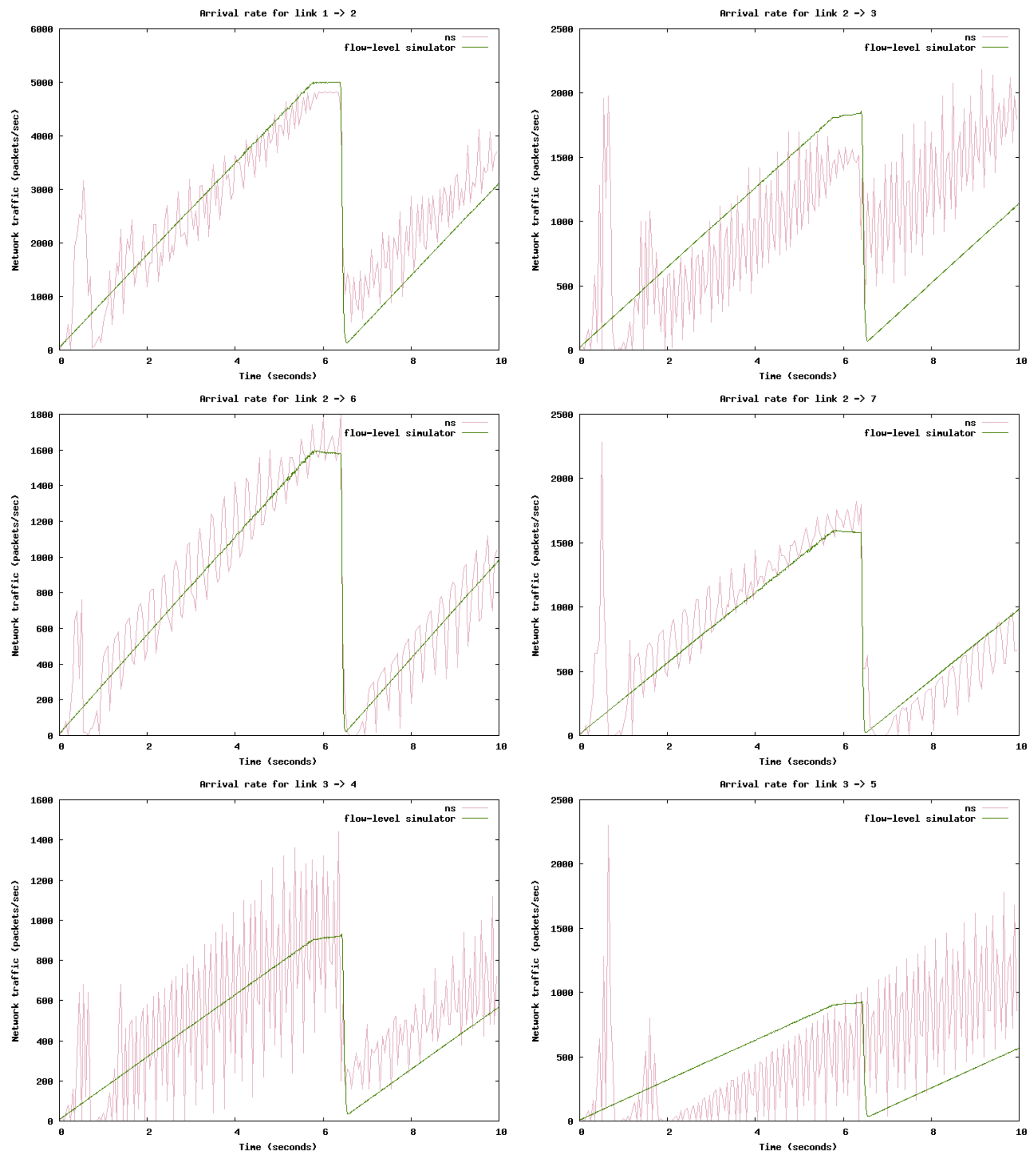


Figure 3.18: Arrival rate at each link (validation with *ns*)

Figure 3.18 shows the arrival rate at all the other links. It confirms that none of those links is congested (for a short period, the traffic is exactly equal to the link capacity for link $1 \to 2$, as it is shaped by link $0 \to 1$). The traffic on links $2 \to 6$ and $2 \to 7$ is logically higher than on $3 \to 4$ and $3 \to 5$, because the flows experience a shorter round trip time and their window sizes increase faster. The only notable difference between the two simulators occurs on link $3 \to 5$. With *ns*, the flows window size reaches two peaks at the beginning (probably both caused by the *slow-start* mechanism) which causes TCP to back-off twice and the flow to have a lower transmission rate than the others at any given time. As a consequence, it seems that the flow doesn't experience any loss when packets start to get dropped by link $0 \to 1$ (no impact of this is visible on the flow's transmission rate which should otherwise be reduced in line with the other flows). The cause for this is not clear, but in a "perfect world" the behaviour of the flow-level simulator makes more sense: links $3 \to 4$ and $3 \to 5$ are identical and support identical flows that follow the same path before entering those links, therefore the traffic is identical for both of them.

Figure 3.19 shows the evolution of the window-size for the different flows. Apart from the *slow-start* at the beginning and the fact that flow 1 doesn't back-off when the others do with *ns*, which was explained above, the results are almost identical for both simulators and correspond to what was previously observed.

To conclude, on this example the flow-level simulator gives an output that is very close to the one given by *ns*. Some differences are present, but they are either minor or caused by the differences between the simulation techniques.

Performances and scalability

Knowing that the final output is pretty much the same with the flow-level simulator and *ns*, it was interesting to explore the differences between them in term of processing speed. [15] and [16] have demonstrated that *flow-level simulation* often outperforms *packet-level simulation*, especially when the number of packets on the network starts to become significant. This section will aim to confirm these results by comparing the performances of the two simulators when the links on a given network get updated to support more traffic.

Figure 3.20 illustrates the network topology used for this scaling test as well as

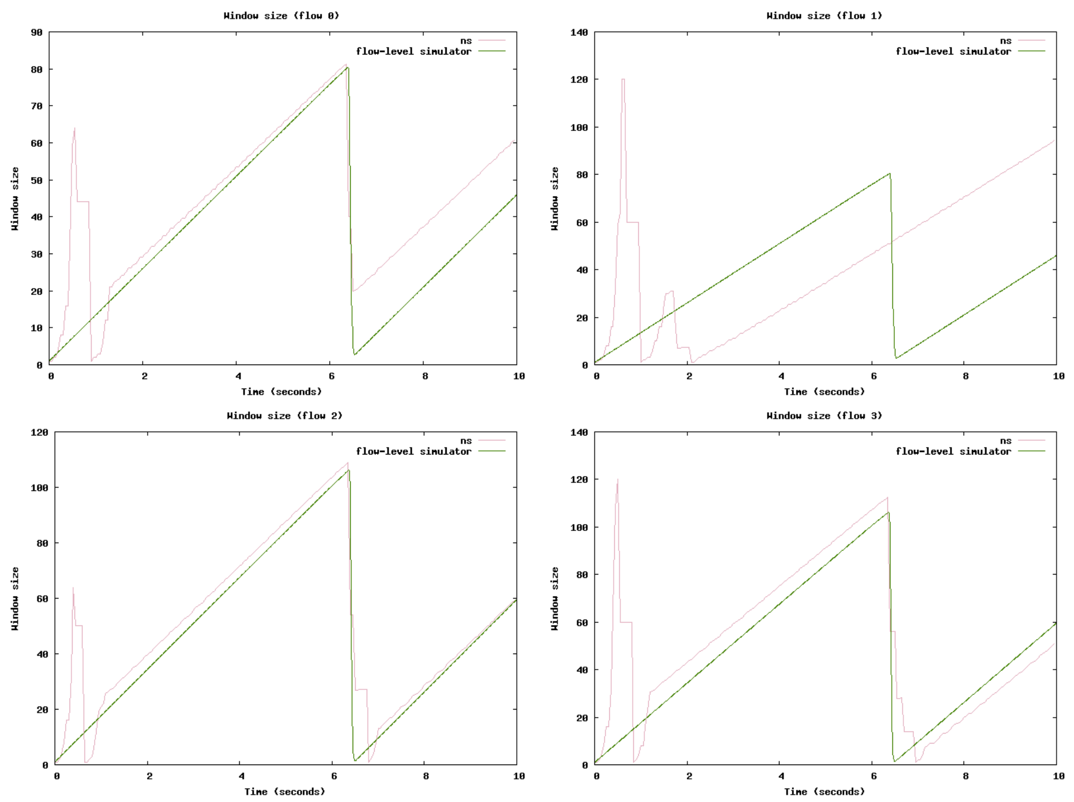


Figure 3.19: Window size for each flow (validation with *ns*)

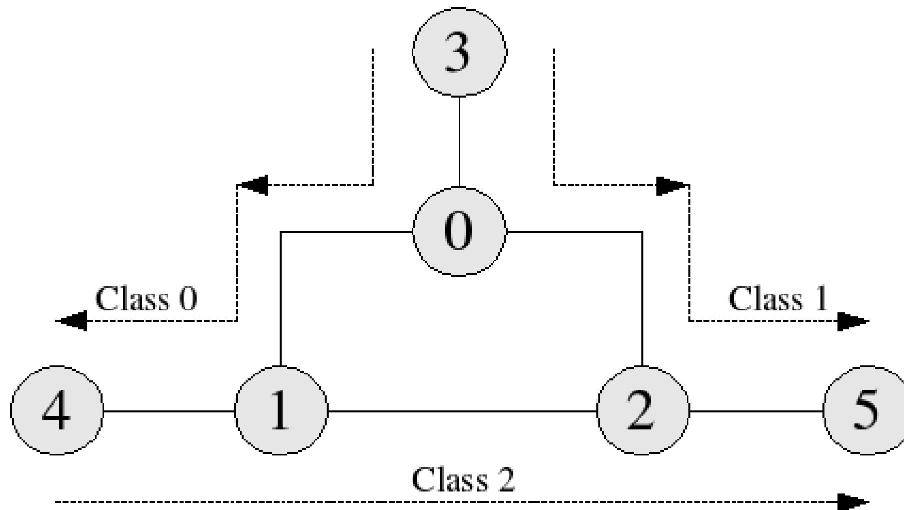


Figure 3.20: Network topology used to compare the performances with *ns*

the classes of flows that go through the network. The propagation delay is 0.01 ms for all the links and the maximum windows size is 2000 packets for each flow. Each class contains 10 flows and the rest of the parameters are the sames as in table 3.1. The purpose of the test is to evaluate how *ns* and the flow-level simulator scale when the network capacity increases. The network was simulated with increasing values for the link-capacity, knowing that the maximum window size was chosen so that it would always allow links to saturate.

Figure 3.21 shows the execution time with both simulators when the links capacities varies from 1000 to 100000 packets/second. As expected, for the flow-level simulator this time is constant (around 5.5 seconds), as only the sending rates changes but no new class is introduced into the network. With *ns* things are different, indeed as the sending rates increase the number of packets does as well, and it takes more time to simulate the network. It is not clear why the execution time doesn't progress proportionally to the traffic, nor why in certain cases the execution time is smaller when there is more traffic in the system. This might be due to specific optimizations implemented in *ns* that simplify the computation of the results when the traffic follows a certain pattern. However, it is certain that the computational needs globally grow with the total sending speed on the network (for a given packet size), which is not the case

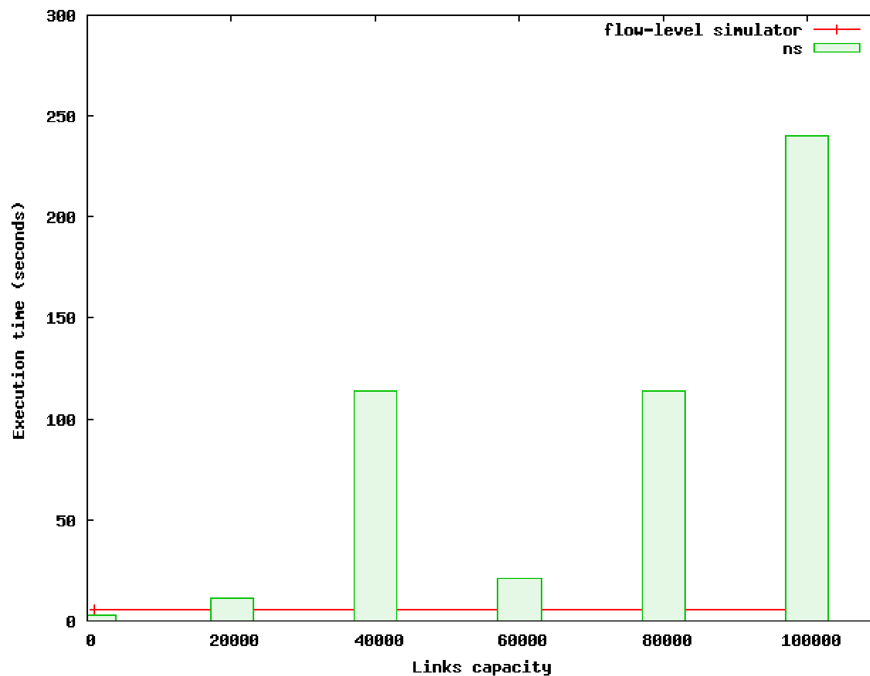


Figure 3.21: Results of the performance comparison with *ns*

with the fluid-based simulator (providing that the traffic patterns don't change). This invariance is an obvious advantage when simulating large-scale networks; as the bigger a network is, the more packets it will have to handle. Moreover, the high-capacity of backbone links in Internet-like networks will give an advantage to flow-level simulation, when compared to packet-level simulation.

3.4 Conclusion

This chapter presented the optimization and validation tasks that were performed on the simulator. It was proven that the current simulator is a lot faster and more scalable than the original implementation and that it produces correct results. Having proved these important facts, the following chapter will concentrate on the next step: the addition of a number of extensions to the simulator.

Chapter 4

Extension of the simulator

4.1 Introduction

After having proved that the simulator was working correctly, it was possible to add a number of extensions to its code. The first of these extensions consisted of improving the control that the user has over the flows that go through the network. The second extension was to implement a *model-reduction* capable of improving the performance of the simulator. Finally, new AQM schemes were added to the simulator.

4.2 Better control over the classes of flow

4.2.1 Motivations

The purpose of this extention is to improve the control that the user has over the flows that go through the network. Indeed, in the original *model* and its implementation, each class is permanently active and there is no way to specify a time when the class is idle and therefore doesn't use any bandwidth on the links. This both restricts the use of the simulator, especially regarding the possibility of estimating a network's behaviour when new flows appear and the traffic load increases, and inconvenient for validating the implementation, as the results presented in other papers are mostly based on scenarios where new flows appear during the simulation.

As a consequence, it was necessary to add new parameters to let the user decide when a given class of flows is active or not. The chosen solution was to associate a

starting time and an ending time to each class: data is transmitted only if the current time is superior to the starting time and inferior to the ending time.

4.2.2 Implementation

Updating the *model* to take this change into account is quite straight forward. Indeed the only thing to do is to introduce the quantities t^i_{start} and t^i_{stop}, the times at which a given flow i starts and stops being active. Then, it is necessary to refine equation 2.7 to take these new parameters into account:

$$A^l_i(t) = \begin{cases} A_i(t) & \text{if } l = k_{i,1} \text{ and } t^i_{start} \leq t \leq t^i_{stop} \\ 0 & \text{if } l = k_{i,1} \text{ and } (t < t^i_{start} \text{ or } t > t^i_{stop}) \\ D_i^{b_i(l)}(t - a_{b_i(l)}) & \text{otherwise} \end{cases} \quad (4.1)$$

The implementation is done by adding a test that checks whether the current time is between the start time and the end time for the flow before adding the flow's traffic to its first link at each step of the simulation. The parameters were also added in the XML description file and the corresponding properties are added to the *ClassOfFlows* object.

4.2.3 Evaluation

In order to validate the fact that the control over the flows is working properly, the network topology presented in figure 4.1 was used. The idea is to have two different flows that go through a common link, and to sometimes have only one active flow and sometimes two.

As described in table 4.1, two different flows are active on the network. Both will have a maximum transmission rate of $\frac{60}{2*(0.01+0.01)} = 1500$ pk/sec, the first one will follow the path $0 \to 2 \to 3$, and the second one the path $1 \to 2 \to 3$. The first class is active from the time $t = 0$ to $t = 10$, and the second one from $t = 5$ to $t = 15$. It means that only the first class is active during the first 5 seconds, then both classes are active for the next 5 seconds, and finally only the second one is active for the last 5 seconds of the simulation. Because of the maximum sending rates, no link should become congested when only one class is active and the sending rate should increase

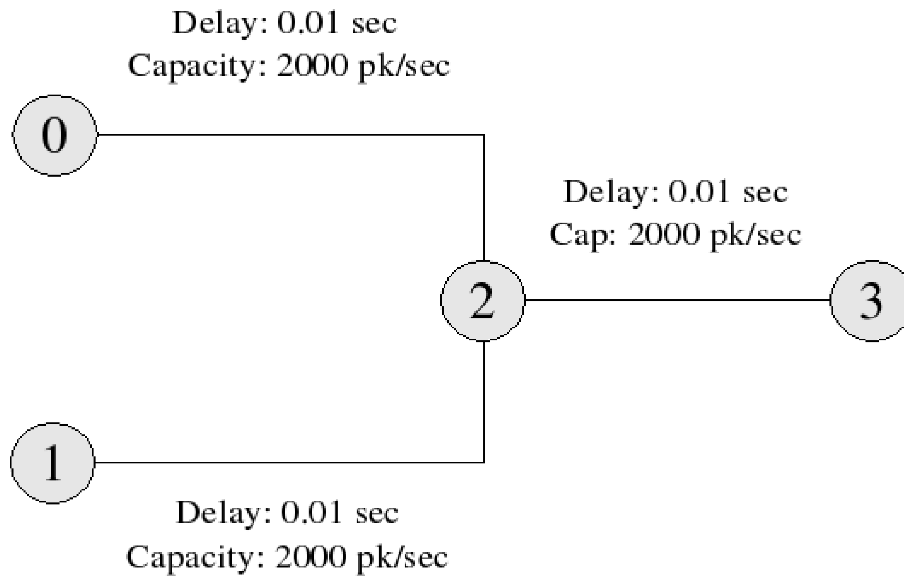


Figure 4.1: The network topology used to validate the class-control

to 1500 pk/sec. When the second flow becomes active, the bandwith of the link from node 2 to node 3 will be shared and the queue size should start increasing for that link. Packet should then start to get dropped and the sending rate for each flow should oscillate so that the traffic generated on the congested link doesn't get higher than its capacity of 2000 pk/sec. Finally when the first flow becomes inactive the sending speed for the second one should progressively reach 1500 pk/sec.

Source	Destination	Nb of flows	Max. win. size	Start time	Stop time
0	3	1	60	0	10
1	3	1	60	5	15

Table 4.1: Active classes of flows for the class-control validation

Figure 4.2 shows the evolution of the network traffic for each class and for the three links along with the evolution of the queue on node $2 \rightarrow 3$ link (the other queues are always empty). The system behaves as expected: at time $t = 6$ seconds the departure rate for the second class reach 500 pk/sec, and because the node $2 \rightarrow 3$ link already transmits the 1500 pk/sec, traffic from the first class, its queue-size starts growing. As

a consequence, packets start being dropped, TCP *backs-off*, and the transmition speed of the two flows oscillates so that the total value is lower than 2000. The queue starts filling a second time as the window sizes get bigger, and finally only the second flow remains active and it reaches its maximum sending rate.

This experiment demonstrates that the extentions that provide better flow control are working well. Other tests confirming the validity of the model, and the implementation, were completed successfully during the development phase.

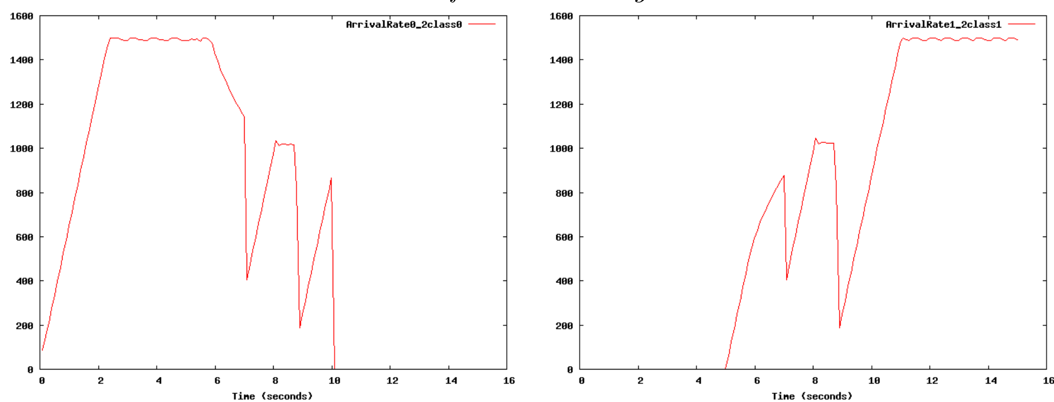
4.3 Model reduction

4.3.1 Motivations

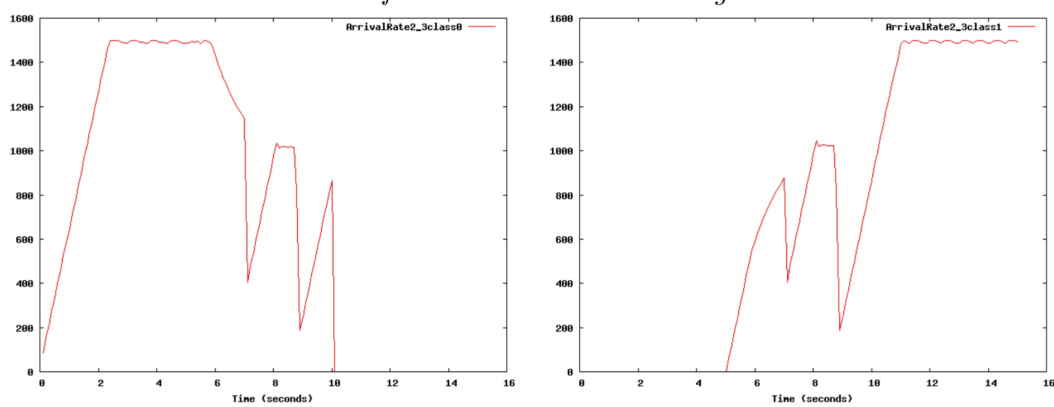
Even though the optimizations presented in the previous chapter greatly improved the execution speed, the overall results remained disappointing. The improvements made the simulator as fast or faster than *ns* in a lot of cases, but the difference was clearly not as significant as expected and in some situations the performance evaluations results were even quite disappointing. It is clear that this disappointment is due to the fact that the code remains quite complicated and could be made a lot simpler. A more challenging way of improving the execution speed was to implement a *model reduction* mechanism to reduce complexity of the simulation. The general idea is to detect which links on the network can not possibly get congested and to simplify the way those links are simulated. For example, one method is not to simulate them at all and to report the delay they introduce to the following links).

This technique can be especially efficient when simulating a typical telecommunication network or a big corporate network. Such networks are composed of access networks that are close to the hosts and a backbone network that is dedicated to transporting data between different sites. The latter is usually overdimensionned in order to be very efficient all the time. As a consequence most of the backbone network links can not be congested and it is not necessary to simulate queue management for those links.

Arrival rates for the uncongested links



Arrival rates for each class at the congested link



Queue length at the congested link

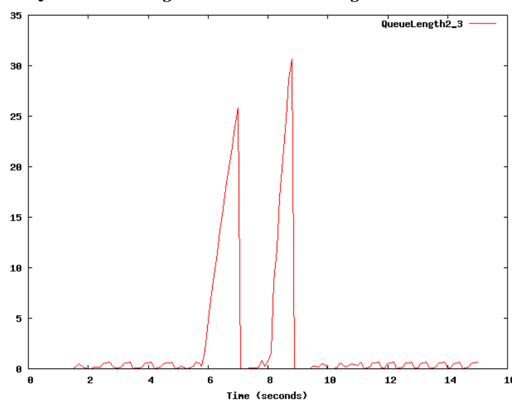


Figure 4.2: Simulation results for the class-control validation

4.3.2 Implementation

Previous work

[14] gives a method describing how to remove all the *uncongested* links (and therefore queues) from a network topology. The resulting simplified network is then simulated instead of the original one. This has the big advantage of significantly reducing the processing power required if the network in not very congested. However, a consequence of the fact that the simulated network is not the original one is that it is not possible to know exactly what is happening on the removed links.

The model reduction, as described by the authors of the paper [14] consists of four steps:

1. Define the matrix ADJ such that $ADJ(i,j) = 1$ if at least one *class of flows* traverses the queue (and link) j immediately after the queue i, and $ADJ(i,j) = 0$ otherwise. For each queue i, $O(i)$ is defined as the set of *classes* that are originating from i (i is their first hop).

2. Each queue i that satisfies the following condition is marked *uncongested*:

$$\sum_{l \in E} ADJ(l,i) * C_l + \sum_{k \in O(i)} \frac{M_k n_k}{\tau_k} < C_i \quad (4.2)$$

   M_k is the maximal *window size* for the class k, n_k the number of flows in the class, and τ_k the *two-way propagation delay*, given by:

$$\tau_k = \sum_{l \in E_k} a_l \quad (4.3)$$

   where E_k is the set of links traversed by class k (including the return path for the acks) and a_l the propagation delay for link l. The first sum in equation 4.2 is the maximal arrival rate from the preceeding links and the second one is the maximum sending rate of the classes originating from the link.

3. Remove all the *uncongested* links from the topology and adjust it accordingly along with the TCP routes. If all the queues that a class traverses are removed, the sending rate for that class will be given by $\frac{M_k * n_k}{\tau_k}$, its maximal sending rate.

4. If no queue was removed in the previous step, end the *model reduction*, otherwise go back to the first step.

To summarise, all the *uncongested* links are removed then the search for *uncongested* links is run again because removing some links could have introduced new *uncongested* ones into the system, and the operation is repeated until the system becomes stable (i.e. it is not possible to find any more *uncongested* links). The article doesn't specify how the topology and the TCP routes should be changed when a link is removed. It is quite clear that the propagation delay for the removed links has to be reported to the following links. However it is not obvious how to do this, nor is it obvious to know how the topology should be modified, especially in the case of complex topologies where several paths can lead from one node to another.

Modification and refinement of the technique

The *model reduction* method that was described above has two drawbacks that led to its modification prior to use in the simulator:

- It is not obvious how to modify the network when links are removed.
- It doesn't simulate the originally defined network exactly. This may be acceptable in certain situations (for example if the only purpose of the simulation is to detect bottlenecks) but for a general purpose simulator it does not seem appropriate.

In order to address these problems, the alternative method used in the simulator does not remove any link but simply simplifies the treatment of the *uncongested* ones by modelling their behaviour as follows:

$$\begin{cases} \forall t, l \in U, i \in C, & D_i^l(t) = A_i^l(t) \\ \forall t, l \in U, & q_l(t) = 0 \end{cases} \tag{4.4}$$

where U is the set containing all the uncongested links, and C is a set containing all the classes. Moreover, no AQM scheme will be associated with these links as the queue length will always be 0. The behaviour of the links that might be congested remains unchanged and is still defined by equation 2.6.

The remaining task to be performed in order to complete the *model reduction* method is to define how to detect uncongested links. We introduce the quantity max_l,

which represents the maximum traffic that can possibly go through a link l, and will be updated as described below. The detection of the *uncongested* links is done by executing the following steps:

Step 1. For each link i, initialize the value of max_i to the value of its capacity C_i

Step 2. For each link i, perform the following operations:

- Compute the quantity T_i (traffic on i) as follows:

$$T_i = \sum_{l \in E} ADJ(l,i) * max_l + \sum_{k \in O(i)} \frac{M_k n_k}{\tau_k} \tag{4.5}$$

  were the notations is the same as in equation 4.2 and τ_k is defined in the same way.

- If $T_i < C_i$, then mark the link as *uncongested*, and redefine max_i as $max_i = T_i$

- If the link was marked *uncongested*, update the max value for all the links that may be impacted by running the *updateFollowingMaxRates* function on the link (see the description of this function below).

Step 3. If no link was marked as *uncongested* during step 2, finish the *model reduction*, otherwise run step 2 again.

The *updateFollowingMaxRates(l)* function is intended to update the value of max_i for each link i that follows (directly or not) the link l. We define N_l as the set of (directed) links that originate from a node at which the (directed) link l ends. Note that a link betwen nodes n_1 and n_2 is considered here as two links, one starting from n_1 and going to n_2, and one starting from n_2 and going to n_1. *updateFollowingMaxRates(l)* consists of performing the following operations:

- For each link $i \in N_l$, if the link is marked as *uncongested*, compute $T_i = \sum_{l \in E} ADJ(l,i) * max_l + \sum_{k \in O(i)} \frac{M_k n_k}{\tau_k}$ and perform the next operation, otherwise process the next link or stop if there is no more links to process.

- If $T_i < M_i$, set $M_i = T_i$ and run *updateFollowingMaxRates(i)*, otherwise do nothing (stopping condition for the recursive function).

Instead of removing nodes, this version of the *model reduction* keeps track of the maximum traffic that can go through *uncongested* links. By using this value instead of the link capacity in equation 4.2, the real maximum arrival rate on the link (not ignoring the fact the previous links might be *uncongested*) is taken in account, which is equivalent to removing the link but does not introduce the disavantages listed above.

4.3.3 Evaluation

Experimental setup

To evaluate the *model reduction* mechanism, a network that resembles a telecommunications network (on a small scale) was simulated. This network is represented in figure 4.3, which shows the different nodes and links along with their capacities (each link has a propagation delay of 0.01 seconds). Apart from the link capacities and the window sizes which are specific to this experiment, all the parameters are the same as specified in table 3.1.

Nodes 0 to 3 and the links between them could represent a backbone network, and nodes 4 to 7 could be the points of presence of the operator in different places, linked to the backbone network at a high link rate and receiving flows from different hosts. All the traffic will be between these nodes and will go through the backbone network. All the classes of flow on the network are active for the full length of the simulation time and are described in table 4.2

Source node	Destination node	Number of flows	Maximum window size
4	5	1	100
4	7	1	100
5	7	1	300
6	7	1	100

Table 4.2: Active classes of flows for the model reduction validation

Three different classes have node 7 as their destination and the goal here clearly is to saturate the link between nodes 3 and 7. Moreover, depending on the routing decisions, the traffic generated by the third flow (between nodes 5 and 7) together with the one from the second (4 to 7) or fourth (6 to 7) one could saturate one of the backbone links.

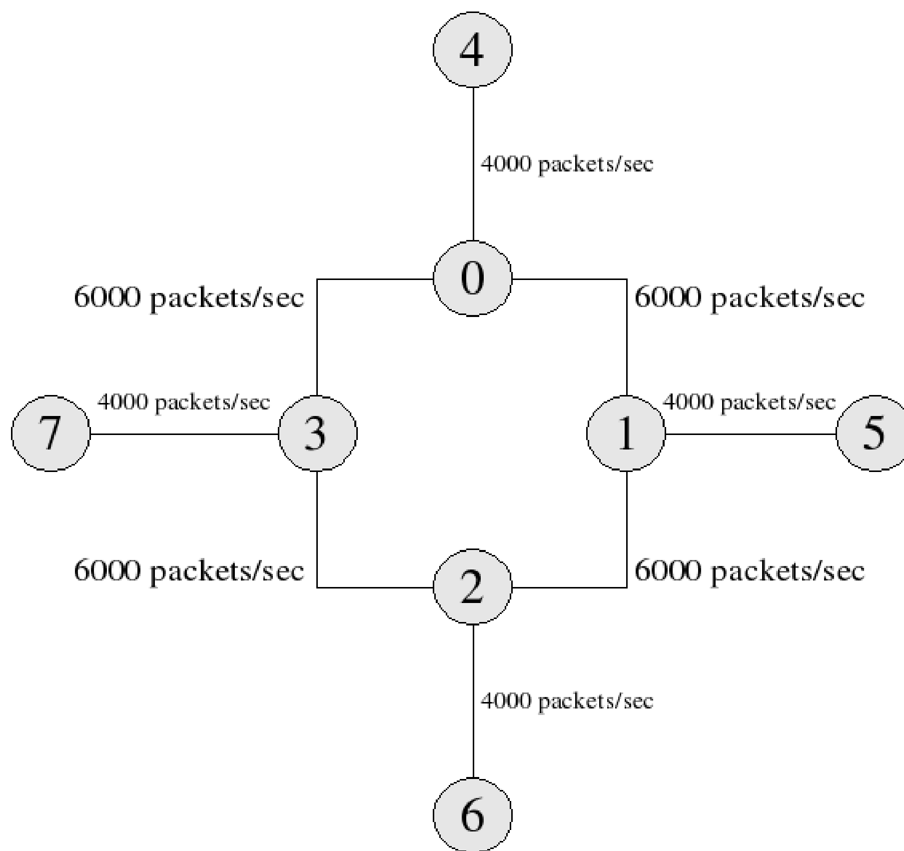


Figure 4.3: The network topology used to validate the model reduction

Evaluation of the correctness of the results

In order to evaluate the correctness of the model reduction, it is necessary to know which paths the classes of flows are going to take and what their maximum sending rates are. The shortest-path algorithm implemented in the simulator decides on the routes as follows:

- The class going from node 4 to 5 should follow the path $4 \rightarrow 0 \rightarrow 1 \rightarrow 5$ which is obviously the shortest. As a consequence, the maximum sending rate for that flow should be $\frac{100}{2*(0.01+0.01+0.01)} = 1666$ packets/second.
- The class going from node 4 to 7 should follow the path $4 \rightarrow 0 \rightarrow 3 \rightarrow 7$ which is obviously the shortest. As a consequence, the maximum sending rate for that flow should be $\frac{100}{2*(0.01+0.01+0.01)} = 1666$ packets/second.
- The path for the class going from node 5 to 7 is not obvious. Indeed the paths $5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$ and $5 \rightarrow 1 \rightarrow 0 \rightarrow 3 \rightarrow 7$ are equivalent and either of them could be chosen. In any case, the maximum sending rate for that flow should be $\frac{300}{2*(0.01+0.01+0.01+0.01)} = 3750$ packets/second.
- The class going from node 6 to 7 should follow the path $6 \rightarrow 2 \rightarrow 3 \rightarrow 7$ which is obviously the shortest. As a consequence, the maximum sending rate for that flow should be $\frac{100}{2*(0.01+0.01+0.01)} = 1666$ packets/second.

Table 4.3 presents the result of the *model reduction* applied to the network. For each link i, the implementation of the *model reduction* algorithm gives the value max_i of the maximum traffic that can possibly go though it, and the link is flagged as uncongested or not depending on this value.

From the results, it is clear that the path $5 \rightarrow 1 \rightarrow 0 \rightarrow 3 \rightarrow 7$ was chosen for the third flow. max_i is correctly set to the sum of the flows originating from the edges of the network for the links starting from nodes 4, 5, 6 and 7. For example the link from node 4 to node 0 has max_i set to 3332, which is the sum of the maximum sending rates for the first two flows of table 4.2. All the links in the backbone have max_i correctly set to the sum of the maximum traffic that go through their adjacent links. All the links but two are marked as *uncongested*. For the links that are not marked *uncongested*,

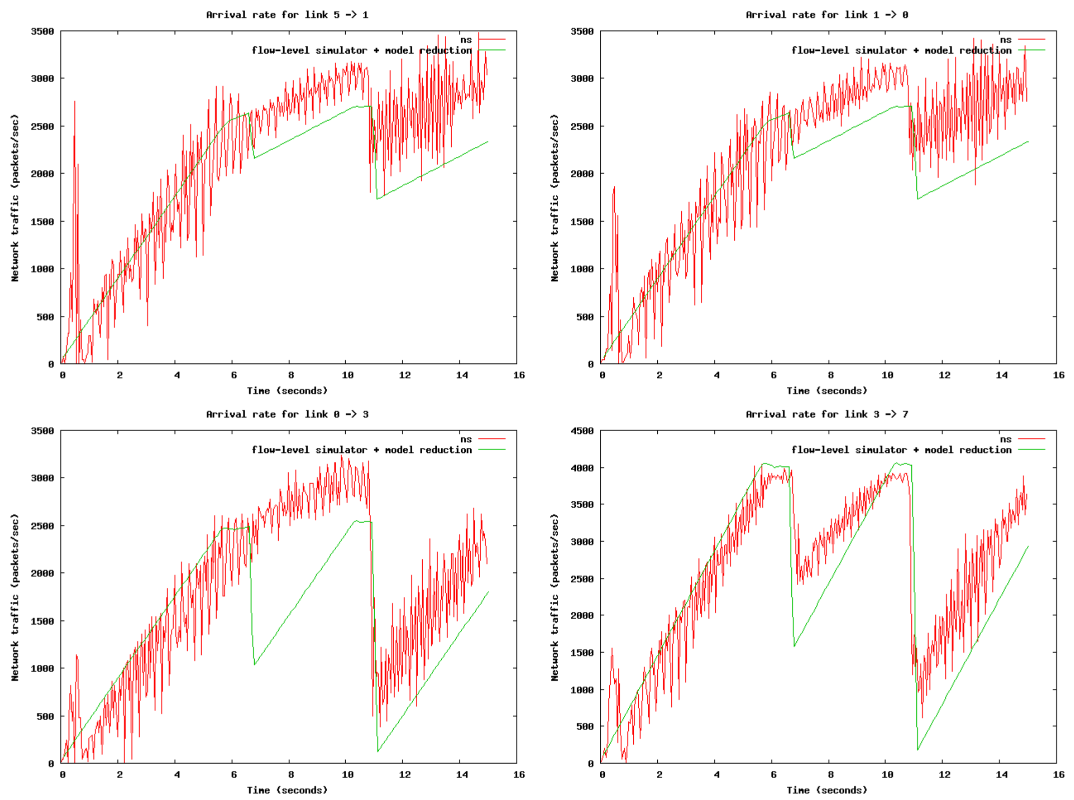


Figure 4.4: Simulation results for the model-reduction validation

Link source	Link destination	max_i (pk/sec)	Uncongested?
0	1	3332	yes
0	3	6000	**no**
0	4	0	yes
1	0	3750	yes
1	2	0	yes
1	5	3332	yes
2	1	0	yes
2	3	1666	yes
2	6	0	yes
3	0	0	yes
3	2	0	yes
3	7	4000	**no**
4	0	3332	yes
5	1	3750	yes
6	2	1666	yes
7	3	0	yes

Table 4.3: Result of the model reduction for the sample network

max_i is equal to their capacities, which is is perfectly normal (max_i cannot be bigger than the capacity and if it was smaller the link would be *uncongested*).

Figure 4.4 shows the total arrival rate for the queues that are on the path of the third flow in table 4.2. All the results obtained with *ns* or the flow-level simulator are similar to each other. Only those queues on the path contains both congested and uncongested links are present here. There are some minor differences that can be neglected. One of these is that the flow-level simulator neglects the traffic generated by acks, which usually is very low. However in this case it is not true because the packet size for the simulation in *ns* is set to its minimal value, which means that the amount of acks is equal to the data traffic. In order to partially correct the difference, the traffic on link $0 \to 1$ was added to the traffic on link $1 \to 0$ and the traffic on link $1 \to 5$ was added to the traffic on link $5 \to 1$ to generate the graphs for the flow-level simulator. This has the effect of adding the equivalent of the acks in the flow-level simulation results, and is only correct in this case because these links are completely *uncongested*, even including the acks (otherwise the acks would saturate the queues and

the behavior of the whole *system* would change). It is clear that the results obtained with *ns* and the flow-level simulator are very close to each other. Mainly, there are two differences, that can be easily explained:

- Even though the global average is the same, the results obtained with *ns* fluctuate a lot more. This is completely expected as the fluid model that is used in the flow-level simulator gives smoother results by its very nature.

- The traffic on link $0 \to 3$ drops just after 6 seconds for the flow-level simulator, whereas it doesn't with *ns*. This is caused by the fact that the queue length for link $3 \to 7$ start increasing to values that are higher than the RED drop threshold at this time. This causes packet drops and the TCP flows that go through that link *back-off*. It is unclear why the only flow to *back-off* is the one from node 6 to node 7 with *ns*; whereas all the flows going through the saturated queue *back-off* with the flow-level simulator. Both behaviours can be considered to be correct, and the difference probably comes from the fact that RED (and especially the evaluation of the average queue length) is not implemented the same way in both simulators, and as a consequence less packets are dropped with *ns*.

Those results clearly show that the simulator is still performing correctly after the application of the *model reduction*, but it would also be interesting know the impact of this *reduction* on the performance of the simulator.

Evaluation of the speed improvement

In order to evaluate the speed improvement brought about by the *model reduction*, the simulation was run with each of the versions of the simulator. The execution time with *ns* is also presented for comparison, but it should be clear that this scenario clearly doesn't intend to illustrate the advantages offered by the *fluid-model* compared to *packet-level simulation* (the network is small and there is only a small number of flows that don't generate much traffic).

Table 4.4 illustrates the result of the performance tests. Those results look quite disappointing, not only because *ns* is faster (which is actually not so disappointing on such as small network, as discussed in sub-section 3.3.3) but also because the *model reduction* doesn't seem to bring about a huge improvement (about 35%). However,

Simulator	execution time (seconds)
ns	3.7
flow-level simulator (model reduction)	3.9
flow-level simulator (no model reduction)	6

Table 4.4: Execution time for the model reduction validation scenario

when the network gets more complex the difference should greatly improve. Moreover, other tests showed that if the metrics relating to *uncongested* links are not written out to a file, the execution time is greatly reduced, which suggest that there is room for optimization in the way *uncongested* links are managed compared to links that might be congested.

4.4 Adding more AQM techniques

4.4.1 Preliminary work

An interesting refinement to the simulator was the addition of support for more AQM schemes. The most obvious application of this improvement is to enable the comparison of how those different schemes perform on the same scenario. Before adding anything new, it was necessary to improve the simulator design to increase its modularity. Indeed, in the original version, which was coded with the RED and PI [3] techniques in mind, some RED or PI-specific functions were present in the *Link* class, which was clearly not acceptable. After moving this code to suitable corresponding classes and defining a root class with a clear interface for the AQM queues, it was then possible to extend the simulator properly.

4.4.2 PI

The PI (proportional integrator) AQM controller was first introduced in [3]. It aims to kept the queue length $q(t)$ as close as possible to a desired queue-length q_{ref}. In [14], the evolution of the dropping probability p_l is described by the following differential equation:

$$\frac{dp_l}{dt} = K_1 \frac{dq(t)}{dt} + K_2(q(t) - q_{ref}) \tag{4.6}$$

where K_1 and K_2 are design parameters of the algorithm.

A test version of the PI implementation was already present in the simulator. The work on this AQM scheme consisted of making sure that the code was still working with the generic queue class defined earlier and to validate and correct the implementation.

4.4.3 DRED

DRED [6] (Dynamic-RED) is an AQM scheme that shares a common goal with RED: stabilizing the queue-length in routers, but aims to perform better at doing it. With DRED, the decision whether to drop a packet or not is made according to a target queue-size T. A simple version of the drop-function can be described by the following equation:

$$p(t) = p(t - \delta t) + \alpha \frac{x(t) - T}{B} \tag{4.7}$$

where t is the current time, δt the sampling interval, α the control gain, T the target queue-size, B the maximum queue-size, and $x(t)$ the estimated average queue-size, as described in equation 2.2.

As with other AQM schemes, adding DRED to the simulator consisted simply of creating a new class inheriting from the root AQM queue class and implementing the correct drop-function. No extra work was needed as the average queue-length function $x(t)$ had already been implemented for use with RED.

4.4.4 AVQ

Unlike the other AQM schemes presented in this dissertation, AVQ (Adaptive Virtual Queue) doesn't aim to stabilize the queue-length of a link, but to maintain the utilization of the link at a given value γ. The other parameter of this algorithm is a smoothing parameter α and a constant C represents the queue's capacity. [7] describes behavior of the AVQ algorithm as follows:

- The router manages a virtual queue whose capacity is $\tilde{C} < C$ (the virtual buffer-size is the same as the one of the real queue).

- Each time a packet arrives at the queue, a virtual packet is processed by the virtual queue. If the virtual queue drops the packet, the real packet is dropped

as well.

- At each packet arrival the virtual capacity is updated using the following equation:

$$\dot{\tilde{C}} = \alpha(\gamma C - \lambda) \tag{4.8}$$

where λ is the arrival rate at the link.

To summarise, the decision to drop a packet or not is made based on a virtual queue's capacity, whose median value is close to the desired link utilization. The size of this virtual queue increases when the link usage is low and decreases when the link usage is high.

The implementation of AVQ in the simulator required some adjustments to the algorithm. Because the simulation is done at a *flow*-level, it was not possible to implement the events that should happend at each *packet* arrival. The proposed solution is to perform those actions at each time-step. It makes the algorithm less reactive: all the packets going though the queue at a given time-step will be processed in the same way, whereas they could be processed differently by the original algorithm. However, if the time-step is short enough, the impact on the simulation results should be negligible.

4.5 Conclusion

This chapter presented and evaluated three different kinds of extensions that were added to the simulator. Those extensions will be very useful in the next chapter, which deals with the evaluation of AQM schemes. Indeed, they allow the simulation of short-lived TCP flows that cause bursts of traffic; they improve the simulation speed so that more complex topologies can be simulated in the same amount of time; and they implement the different AQM schemes to be evaluated.

Chapter 5

Evaluation of different AQM schemes

5.1 A methodology to evaluate AQM schemes

It order to use the fluid-based simulator to evaluate different Active Queue Management Schemes, it was necessary to define a precise framework that allows us to measure certain characteristics for each of the schemes to be compared. This framework should contain the following elements:

- Several *metrics* that are easy to measure and representative of what is expected from an AQM algorithm.
- One, or several, *scenarios* (network topology and description of the evolution of the traffic flows) that are easy to simulate and give a good idea of the behavior of the algorithms in real world application.

This section presents a framework that is outlined in [20]. This same method was also used in [21] to compare a number of different AQM schemes.

5.1.1 Metrics for AQM schemes evaluation

[20] clearly identifies five metrics that are relevant for the evaluation of AQM schemes:

- *Utilization*: This metric represents the total percentage of the links capacity that is used during the simulation. This give an idea of the ability of the AQM scheme to maximise the usage of the available bandwidth.
- *Delay*: This is the average delay that is experienced by a packet to reach its destination. It gives an idea of the ability of the AQM scheme to support applications like voice over IP that require short round-trip times.
- *Jitter*: This is the average variation in the time between each packet arrival for each flow. Isochronous application like voice transmission are also affected by jitter.
- *Drop rate*: This is the total percentage of packets that are dropped. Compared to *utilization*, this is just another way of evaluating the optimization of the usage of the network ressources.
- *Fairness*: This metric evaluates whether each flow gets its "fair share" of the available resources. If the system is completely fair, the value is 1, and if only one flow gets all the resources, the value is 0. See [20] for more details.

5.1.2 A classic simulation scenario for AQM schemes evaluation

Because it is well understood and easy to implement, [20] recommends the use of a *dumbbell* topology for AQM evaluation (see figure 5.1).

Basically, the dumbbell topology consists of a series of flows going through two routers at each extremity of a bottleneck link to which is associated a bandwith and a propagation delay. To make the simulation more realistic, the flows should experience different round-trip times. With the flow-level simulator, this can be done by adding high-bandwith links with different propagation delays after the bottleneck. Because their bandwidth is higher that that of the bottleneck this will only have the effect of changing the round-trip times experienced by the flows.

Finally, [20] suggest to mix different types of network traffic during the simulation:

- A high proportion of short-lived TCP flows that could, for example, model Web traffic.

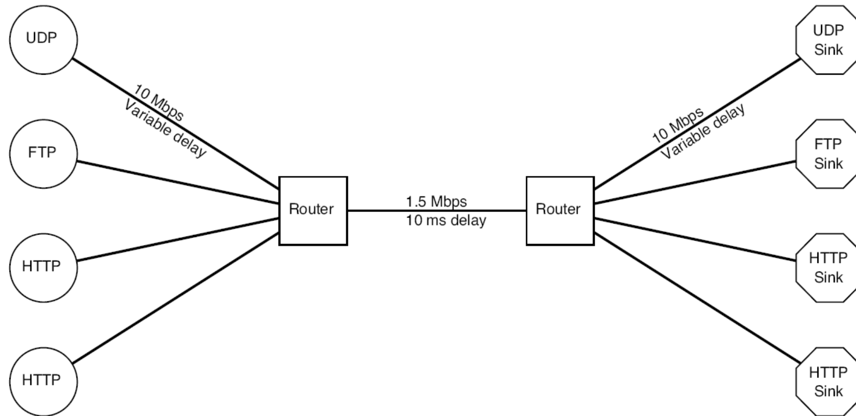


Figure 5.1: The dumbbell topology [20]

- A smaller proportion of long-lived TCP flow to model FTP traffic.
- One or several UDP flows to take into account the protocols that do not support congestion-control.

Unfortunately, the model used for the simulator does not support UDP traffic, and as a consequence it was not possible to include flows that don't implement end-to-end congestion control. An attempt was done to modify the model to include such traffic patterns (basically by having flows that don't have a window size and have a fixed departure rate) but this was not possible within the timeframe of this project.

5.2 Comparison of different AQM schemes

5.2.1 Experimental setup

In order to evaluate the different schemes, a dumbbell scenario was used. The bottleneck link capacity was 10000 pk/sec and it had a delay of 10ms. This link supported the traffic generated by different short lived and long lived flows that experienced different round-trip times. The list of those flows is given in table 5.1. All the flows have a maximum window size that is large enough to saturate the link.

All the queues have a capacity of 200 packets. The AQM parameters follow recommendations from other papers:

Class	RTT	Nb. of flows	Max. win. size	Start time	Stop time
0	20 ms	20	10000 pk	0 s	15 s
1	20 ms	5	10000 pk	5 s	6 s
2	20 ms	5	10000 pk	10 s	11 s
3	100 ms	20	10000 pk	0 s	15 s
4	100 ms	5	10000 pk	7 s	8 s
5	100 ms	5	10000 pk	10 s	11 s

Table 5.1: Active classes of flow for the evaluation of AQM schemes

- For RED, the value of the parameters t^{min}, t^{max}, and p^{max} are respectively 20, 50, and 0.5.
- For DRED, the values of α and T are respectively 0.005 and 50.
- For AVQ, the values of α and γ are respectively 0.15 and 0.8.

5.2.2 Experimental results

Figures 5.2, 5.3, and 5.4 respectively show the network traffic, end-to-end delay, and loss-rate for each class and with each AQM scheme. Figure 5.5 represents the total network utilization with the different schemes.

The first thing to note about those results is that the network traffic is fluctuating a lot, even in the absence of the temporary peaks of traffic generated by the short-lived flows. The explanation for this is that it is probably due to a weakness in the way similar flows are managed by the mathematical model used by the simulator. Because the model simulates a “perfect world”, as soon as packets start to get dropped at a given link, all the flows at that link back-off very quickly at the same time, which suddenly greatly reduces the traffic load. In the real world, random factors would cause the flows to react differently and back-off at different times, and therefore the average traffic load would fluctuate a lot less.

In spite of the previous conclusion, the following conclusions can be drawn about the different metrics:

- *Utilization*: According to the results presented in figure 5.5, AVQ seems to maximise network utilization compared to the other AQM schemes. Except from pe-

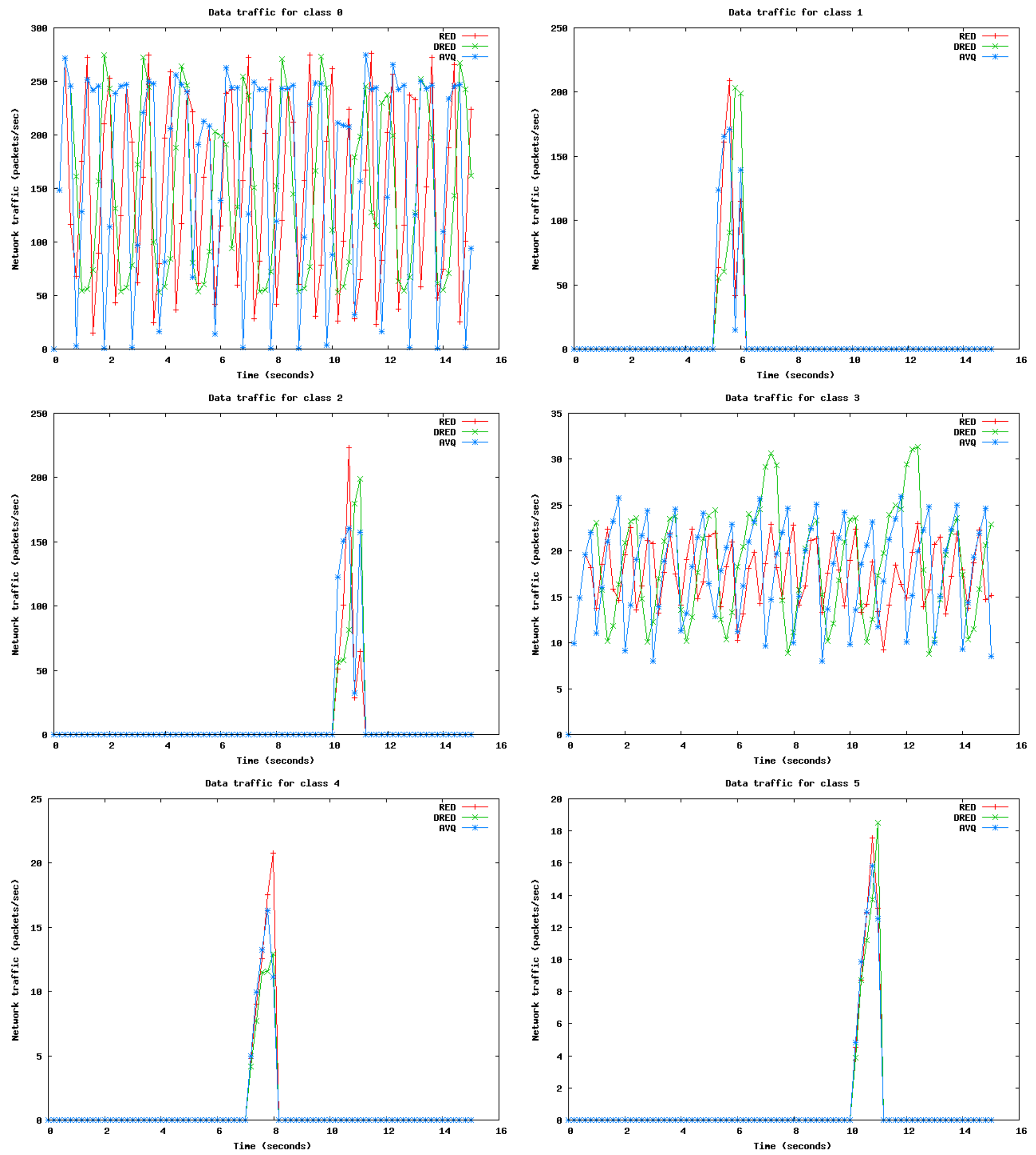


Figure 5.2: Comparison of the network traffic for the different AQM schemes

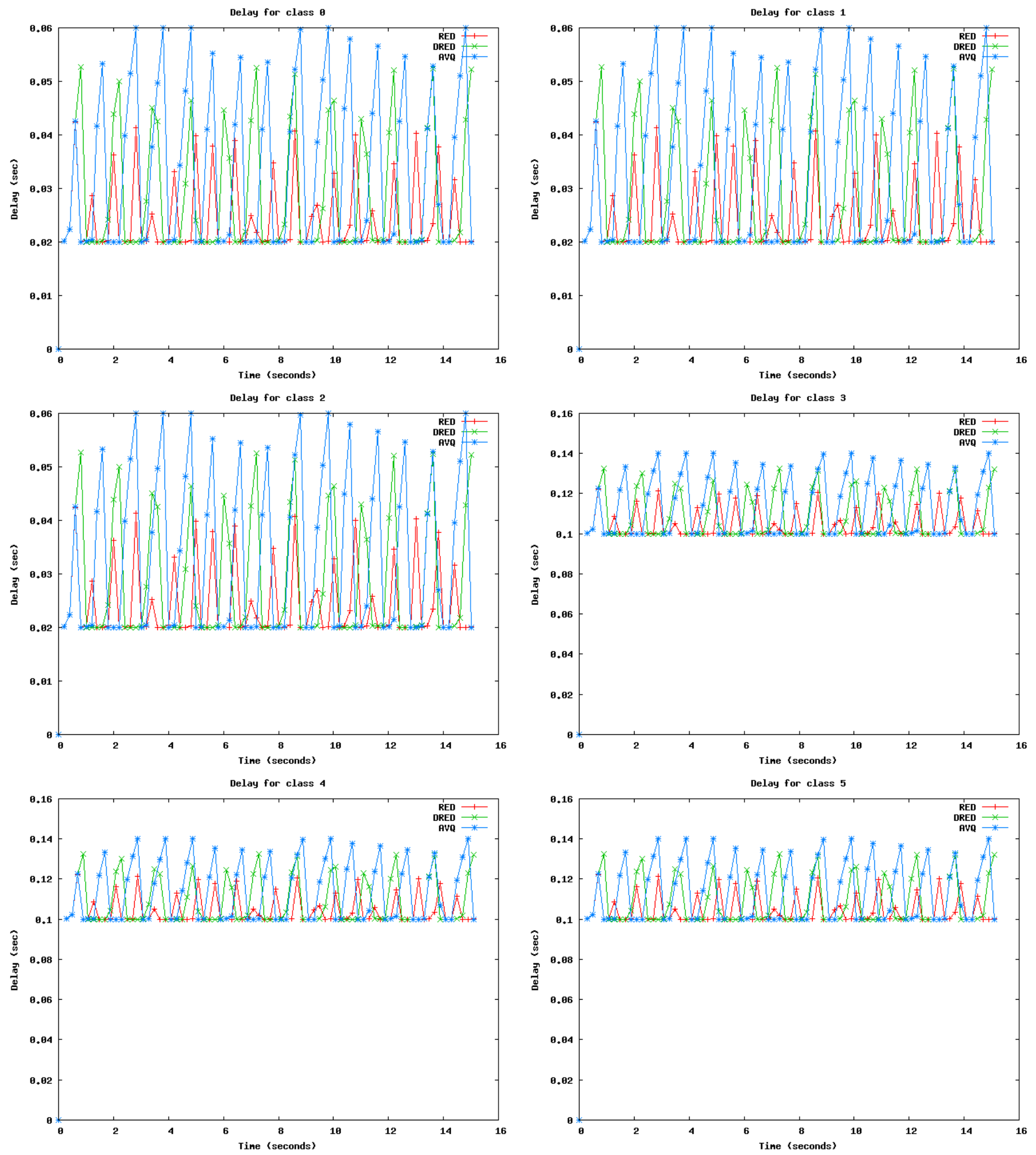


Figure 5.3: Comparison of the end-to-end delay for the different AQM schemes

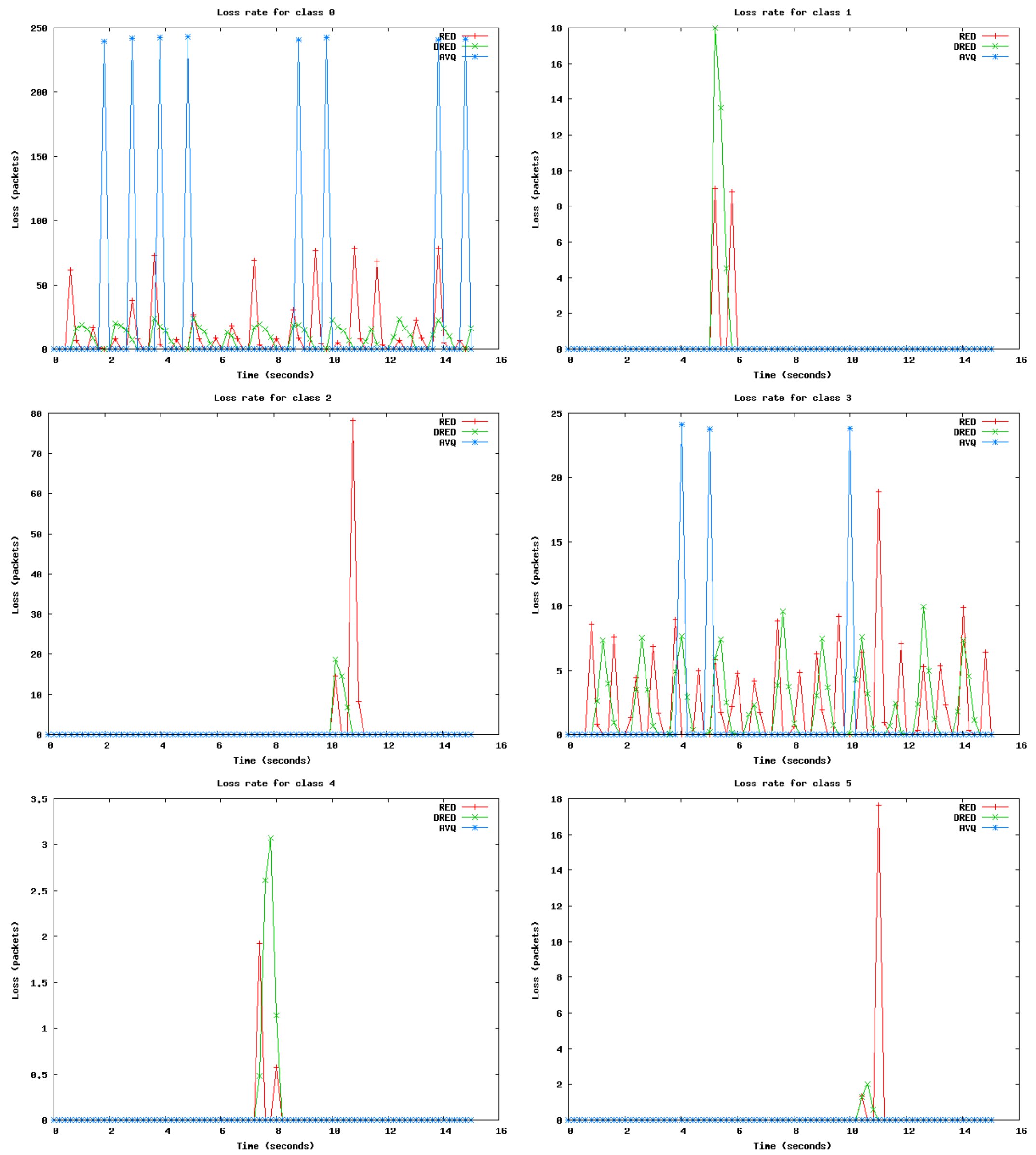


Figure 5.4: Comparison of the loss-rate for the different AQM schemes

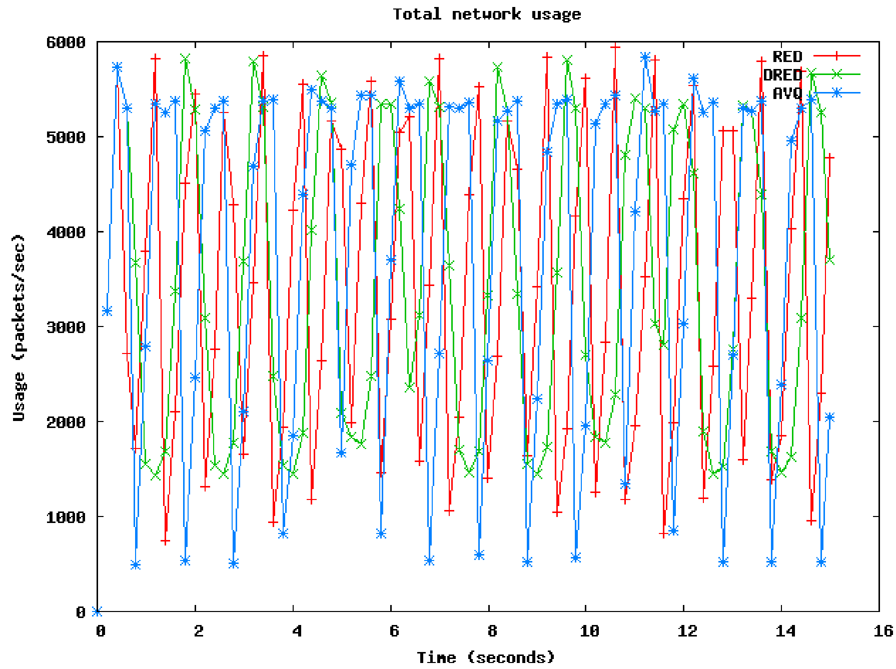


Figure 5.5: Total network utilization for the different schemes

riodical drops that are more significant than for the other schemes, the network usage is a lot more stable at high values. Moreover, these drops would probably be less important if the model didn't have the weakness described above. RED and DRED seem to exhibit similar performance, but the traffic is more stable with DRED. It also is interesting to notice that AVQ is close to maintaining the link utilization at 80 percent, as it was supposed to (the value of the γ parameter was set to 0.8).

- *Delay*: AVQ performed a lot better in terms of network usage, but the drawback is that this seems to be at the expense of the introduction of more delay (see figure 5.3). RED, which was the worst in terms of utilization, is the best scheme in term of delay.

- *Jitter*: In this simulation scenario, jitter is strongly linked to delay. Indeed, the delay tend to stay close to its minimum value and experience periodical peaks which introduce jitter. As a consequence, the ranking in term of jitter is the same

as in term of delay.

- *Drop rate*: In terms of drop rate, AVQ seems to drop a lot of packets from time to time and then not drop anymore for a given period, whereas RED, and even more so, DRED have a smother behaviour, which explains the lower jitter with these algorithms. It is interesting to notice that AVQ seem to be a lot more gentle with the short-lived flows as they almost experience no drops. This matches the idea of AQM enabling routers to accommodate peaks of traffic while regulating long-term sources of congestion. DRED seems to manage to maintain a lower drop rate than RED for traffic experiencing a short RTT, whereas the behaviour of the two algorithms is similar with flows which suffer from a longer transmission delay. Overall, it is difficult to say which AQM scheme is the best, but it seems that DRED and AVQ have a better control on the loss-rate; each one in a different way.

- *Fairness*: The fairness metric is difficult to evaluate in this scenario. Indeed, because the whole traffic is composed of TCP flows, the congestion-avoidance mechanism present in this protocol tends to satisfy the fairness expectations by itself. The transmission rates for the flows experiencing longer RTTs is a lot smaller, which might look unfair at first as they are capable of transmitting at higher speeds. However, there is nothing AQM schemes can do about this as it is simply due to the fact that their window-sizes increase a lot more slowly. In the end, it would be necessary to implement UDP in the simulator to obtain more accurate information on fairness.

5.3 Conclusion

The experiments conducted in this chapter give an overview of the different AQM schemes in a situation that is representative of specific real-life conditions. It did not allow us to draw a conclusion about the fairness of the algorithms but it gave some valuable information about their behavior and described a good methodology for the evaluation of these algorithms. Overall, from the experimental results it seems that in this particular situation AVQ is the best scheme for maintaining high capacity usage,

and DRED is the most efficient at accommodating applications with high isochronous requirements.

Chapter 6

Conclusion

6.1 Achievements

First, the field of network simulation has been presented (with a particular focus on flow-level simulation) in chapter 2. The concept of active queue management was also introduced, along with a detailed description of a particular AQM scheme: RED. The two notions of flow-level simulation and Active Queue Management were then used to introduce a fluid-based model for network simulation that can support different AQM algorithms. This was designed with scaling in mind. The presentation of the background information section of this dissertation ended with suggestions on how to implement this model.

The presentation of an existing implementation of the model, given in chapter 3, required thorough appreciation of its code. In the same chapter, a series of improvements to make the implementation easier to understand and to use were presented. Some optimizations that were made to the simulator, both in terms of CPU and memory usage, were also presented. This part of the dissertation ended with a thorough validation of the simulator. This was done though different means, in particular by comparing simulations with those done using the well-recognized *ns* network simulator. These experiments showed that the flow-level simulator works correctly and is faster than a well-optimized packet-level simulator in many situations.

Chapter 4 then introduced several extentions that were implemented within the simulator, while demonstrating their correctness. The first of these extentions was to

improve the control given to the user over the behaviour of the different flows during a simulation. This modification of the model was actually needed in the previous part of the work to validate the implementation against other results. The second extension presented was a model reduction that allows for the detection of links that will never be congested. The simulation of these links is simplified to increase the simulation speed. Finally, the chapter ended with the presentation of the different AQM schemes that where added to the simulator.

The implemented AQM schemes were evaluated in chapter 5. The chapter started by presenting a methodology for the performance evaluation of the algorithms. Experimental results were then analyzed to attribute eventual specific properties to each scheme.

As emphasised in the introduction, this work had two ultimate goals. The first one was to show that implementing a flow-level network simulator that scales well and performs better than a well-recognized packet-level simulator in many situation is a very reasonable objective. This goal was definitely achieved, as demonstrated in the performance evaluation tests that were described in this document. The second goal was to illustrate a way to use this kind of simulator for protocol evaluation. The comparison of different AQM schemes that was presented met these expectations. Indeed, the simulator clearly showed the differences between the congestion management schemes implemented. However, the presented results did not cover a very wide range of tests. Moreover, they uncovered a weakness in the model when several identical flows are present in the network. Even though it is not a positive result about the model itself, this unexpected outcome is a very good thing as it proves the relevance of the work that was carried out on this model.

6.2 Future work

Even though they gave good results, the experiments run to validate the simulator would lead to even more positive conclusions if they could be reproduced on a very large scale network topology. However, this involves further optimization of the simulator. Indeed, the improvement of the simulation speed following this work was important and encouraging, but the simulator still is too slow to process a very large scale network. Moreover, *ns* could also be an obstacle to this validation if it was used as a reference,

as it clearly wouldn't be able to simulate such a network in a reasonable amount of time.

Another obvious source of future work is to add even more AQM schemes to the simulator. Adding new schemes to the simulator is straightforward, provided that those algorithm can be described by simple equations that can be processed by the solver included in the code.

Finally, the last part of the dissertation illustrated a few weaknesses of the model when it comes to evaluating protocols, in particular, AQM schemes. It would be interesting to work on finding a way to simulate UDP traffic with the simulator, as this protocol is often a cause of network saturation because of its lack of a congestion management mechanism. This would be useful as it would provide more information on the fairness of the different AQM algorithms evaluated in chapter 5. It would also be interesting to look at the fact that the model doesn't always seem to give realistic results when several similar flows are present in the network. The fact that all the flows show a "perfect" behavior and back-off very quickly at exactly the same time does not correspond to reality. Incorporating a model of the random factor that cause the flows to react slightly differently from each other in the real world would be highly beneficial. Moreover, even though it gave some interesting results, the modest evaluation of different AQM schemes that was conducted should be extended to a wider variety of scenarios.

Bibliography

[1] V.G. Cerf and R.E. Kahn. *A Protocol for Packet Network Intercommunication.* IEEE Trans. on Comm., Vol. COM-23, May 1974, pp. 637-648.

[2] V. Jacobson. *Congestion Avoidance and Control.* ACM SIGCOMM '88, August 1988.

[3] C. Hallot, V. Misra, D, Towsley, W. Gong. *On designing improved controllers for AQM routers supporting TCP flows.*. INFOCOM, 2001.

[4] S. Floyd and V. Jacobson. *Random early detection gateways for congestion avoidance.* IEEE/ACM Transactions on Networking, 1(4):397–413, August 1993.

[5] B. Braden et al. *Recommendations on queue management and congestion avoidance in the Internet.* RFC 2309, April 1998.

[6] J. Aweya, M. Oulette, D. Y. Montuno. *A control theoric approach to active queue management.* Computer Networks, vol. 36, pp. 203235, Aug. 2001.

[7] S. Kunniyury, R. Srikant. *Analysis and Design of an Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management.* SIGCOMM01, August 27-31, 2001, San Diego, California, USA.

[8] Averill M. Law, W. David Kelton. *Simulation, Modeling, and Analysis.* McGraw-Hill, Third Edition, 2000.

[9] Lawrence S. Brakmo and Larry L. Peterson. *Experiences with Network Simulation.* University of Arizona, 1996.

[10] S. Keshav. *REAL: A Network Simulator.* Univ. California, Berkley, 1988.

[11] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. *Advances in network simulation.* IEEE Computer, 33(5):59–67, May 2000.

[12] V. Misra, W. B. Gong, and D. Towsley. *Stochastic Differential Equation Modeling and Analysis of TCP Windowsize Behavior.* Technical Report, University of Massachusetts, Amherst, 1999.

[13] V. Misra, W. B. Gong, and D. Towsley. *Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED.* In Proceedings of ACM/SIGCOMM, pages 151–160. ACM, August 2000.

[14] Y. Liu, F. Presti, V. Misra, D.F. Towsley, and Y. Gu. *Scalable fluid models and simulations for large-scale IP networks.* ACM Transactions on Modeling and Simulation, 14(3), pages 305-324, July 2004.

[15] T.K. Yung, J. Martin, M. Takai, and R. Bagrodia. *Integration of fluid-based analytical model with Packet-Level Simulation for Analysis of Computer Networks.* SPIE, 2001.

[16] B. Liu, D.R. Figueirdo, Y. Guo, J. Kurose, and D. Towsley. *A Study of Networks Simulation Efficiency: Fluid Simulation vs. Packet-level Simulation.* INFOCOM, 2001.

[17] J. W. Daniel, R. E. Moore. *Computation and Theory in Ordinary Differential Equations.* W.H. Freeman, San Francisco, CA, 1970.

[18] D. Bruno. *Evaluation of AQM Schemes through Fast Simulation of IP Networks.* Trinity College Dublin, Internal technical report, 2004.

[19] Chengyu Zhu, O.W.W. Yang, J. Aweya, M. Ouellette, and D.Y. Montuno. *A comparison of active queue management algorithms using the OPNET modeler.* IEEE Communications Magazine, 40(6):158–167, June 2002.

[20] A. Bitorika, M. Robin, and M. Huggard. *An evaluation framework for active queue management schemes.* MASCOTS'03, pages 200-206. IEEE, Oct. 2003.

[21] M. Huggard, M. Robin, A. Bitorika, and C. Mc Goldrick. *Performances Evaluation of Fairness-Oriented Active Queue Management Schemes.* MASCOTS'04, pages 105-112, IEEE, Oct. 2004.