University of Twente
The Netherlands

Modeling Active Queue Management algorithms using Stochastic Petri Nets

Master Thesis

Author:
S. Dijkstra

Supervising committee:
prof. dr. ir. B.R.H.M. Haverkort
dr. ir. P.T. de Boer
ir. N.D. van Foreest

Design and Analysis of Communication Systems
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

December 14, 2004

Abstract

Active Queue Management (AQM) algorithms have been designed to be able to actively control the average queue length in routers supporting TCP traffic, and thus to be able to prevent congestion and resulting packet loss as much as possible. Many different algorithms have already been presented in literature, of which the RED algorithm has been given the most attention. In order to be able to compare these algorithms, various approaches can be applied to model the algorithm behavior, and subsequently to derive interesting performance measures. These modeling approaches have certain advantages over algorithm simulation (for example using the ns-2 tool), which has mostly been applied to derive algorithm performance measures up to now.

In this assignment, a first step will be to structure and classify the wide range of available AQM algorithms. Then, three modeling approaches, being classical control theory, the concept of stochastic differential equations and the concept of stochastic Petri nets, will be discussed, and their modeling power with respect to AQM algorithms (and in particular to RED) will be presented. In the second part of the assignment, the applicability of the stochastic Petri nets modeling approach with respect to the modeling of TCP/AQM characteristics will be investigated. Not only will we try to model RED behavior in more detail, but we will also see to what extent it is possible to model the behavior of other AQM algorithms.

Contents

Chapter 1

Introduction

In the last fifteen to twenty years, the use of the Internet has taken off rapidly. The public Internet is a world-wide computer network, which interconnects millions of computing devices around the world. Most of these devices are so-called *hosts* or *end systems*, e.g., traditional desktop PCs, Unix-based workstations or servers that store and transmit information such as web pages and email messages. Each of these end systems, as well as most of the other pieces forming the Internet, run so-called *protocols* that control the sending and the receiving of information within the Internet. Two of the most important of these protocols are the Transmission Control Protocol (TCP) and the Internet Protocol (IP).

End systems are connected by *communication links*, made up of different types of physical media, such as coaxial cable, copper wire, fiber optics and the radio spectrum. Usually, these end systems are not directly connected to each other via a single communication link, but are indirectly connected via intermediate switching devices, which are called *routers* or *gateways*. A router collects incoming information, determines the destination of this information, and forwards it again. The path traveled by the information from the source host to the destination host, via one or more routers, is called a *route* or *path* through the network. Figure 1.1 shows a typical network layout, including several hosts and routers.

Each of the routers in a network has a finite amount of buffer space, in which it can store information before forwarding it to the network. This storage space is necessary for prevention of information loss whenever data is coming in at a rate higher than the maximum processing rate of the router. Information that cannot be processed and forwarded directly is stored in the router buffer until the router processor is available. In order to prevent buffering delay at the router from getting too high, and also to keep the amount of information loss as low as possible, this buffer needs to be managed intelligently. Therefore, a variety of buffer management algorithms, which are commonly known as *active queue management* algorithms have been designed. These algorithms are the main subject of this report.

When one wants to analyze the performance of a certain (active queue management) algorithm, in order to be able to compare it to other algorithms, there are two possible approaches. One could either set up and run a simulation, for which various tools are available, or one could make an analytic model of the algorithm and the environment that influences its performance, and carry out mathematical analysis on this model in order to obtain the required performance measures. Using the results of either the simulation or the mathematical

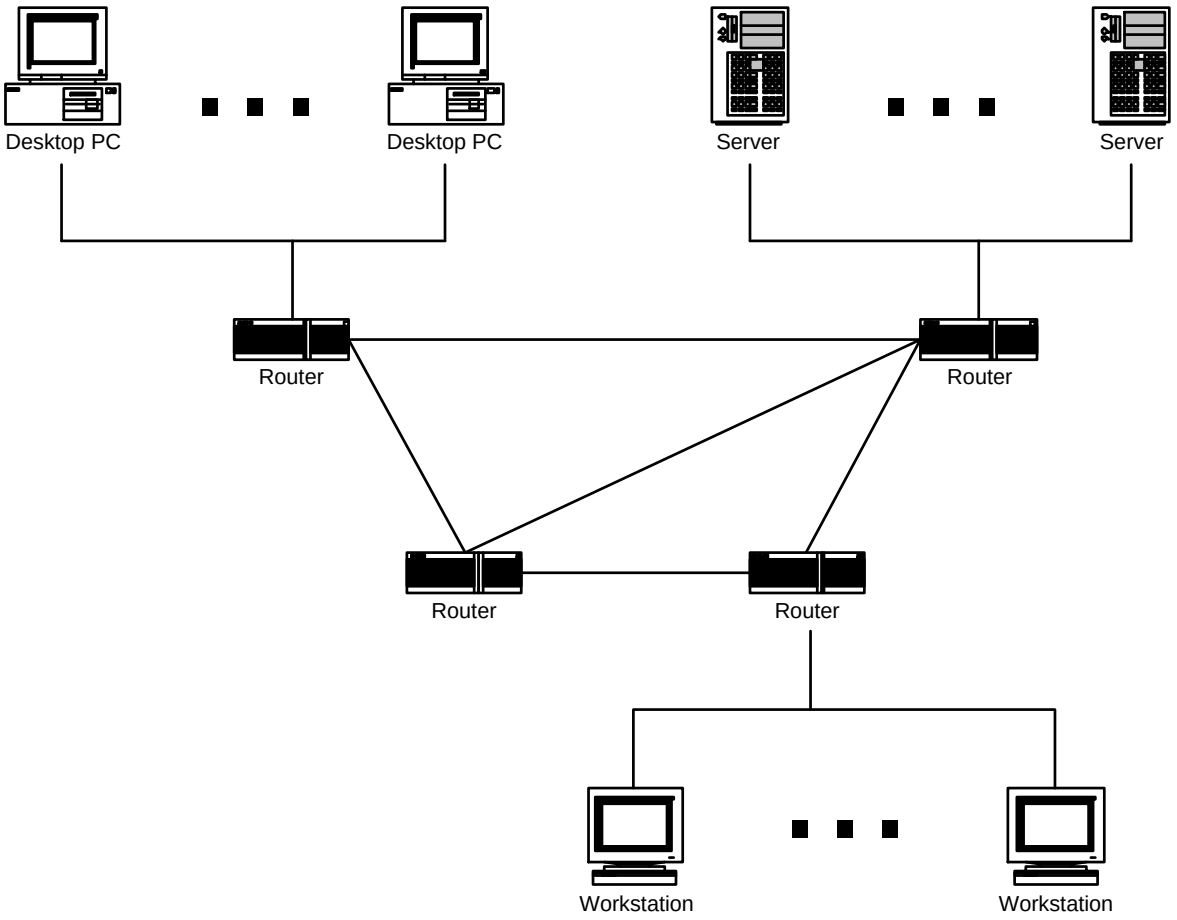

Figure 1.1: A typical network topology

analysis, a comparison between various algorithms can be made, or the performance of a single algorithm can be judged. This report focuses mainly on mathematical and numerical analysis of models, a choice that will be advocated in Chapter 4.

The rest of this report is structured as follows: In Chapter 2, TCP and its congestion control mechanism are introduced in some more detail. Chapter 3 contains an overview and a classification of a range of active queue management algorithms. As has been said, Chapter 4 compares simulation and modeling as the two approaches to algorithm performance analysis, and additionally gives an overview of three well-known modeling approaches. Next, in Chapter 5, an analytical model of TCP source and buffer interaction is presented. This model is extended in Chapter 6, where it is made suitable for the modeling of active queue management algorithms. The model is verified for correctness using an application to one of the most widely used active queue management algorithms. Chapter 7 presents some guidelines on the use of the extended model in combination with other algorithms. Finally, Chapter 8 provides us with some conclusions on the work that has been presented in this report, and gives some pointers on future work on this subject.

Chapter 2

About TCP and congestion control

In this chapter, the Transport Control Protocol (which is better known as TCP) will be introduced. The text from this chapter has been derived from [1]. After a short overview of the general purpose and design of the protocol, the congestion avoidance algorithm of TCP will be summarized. This algorithm forms the basis for a wide range of Active Queue Management algorithms, which will be the topic of Chapter 3.

2.1 Purpose and design of TCP

The main goal of the TCP protocol (which has been defined in RFCs 793, 1122, 1323, 2018 and 2581) is to provide a reliable data transfer service for Internet connections. It does so in a connection-oriented fashion, which means that before a TCP sender entity can send its data to a TCP receiver entity, a connection between sender and receiver has to be set up, which is done by means of a so-called "three-way handshake". In this handshake, the connection parameters are being negotiated by the sender and receiver entities. Once the connection has been established, the sender and receiver can start transferring data to each other. Since TCP is a full-duplex protocol, they can both send data at the same time.

Since the network layer service provided by the Internet Protocol (IP) is unreliable, TCP has to create a reliable data-transfer service on top of IP's unreliable best-effort service. This reliable TCP service ensures that the data stream coming out of the connection is exactly the same as the ingoing data stream. The TCP protocol ensures this by having the receiver send acknowledgement packets (better known as ACKs), which communicate back to the sender that packets have arrived correctly. These ACKs can be either first-time ACKs, for packets for which the sender has yet to receive an acknowledgement, or duplicate ACKs, which re-acknowledge a packet for which a first-time ACK has already been received earlier. In the case of the arrival of a first-time ACK, the sender knows that all data up to the packet being acknowledged by the arriving ACK has been received correctly. A duplicate ACK is sent in case a packet is received that does not have the expected sequence number (which is used to keep track of the order in which packets were sent). When the sequence number of an incoming packet is higher than the expected sequence number value, the receiver knows that there is a gap in the data stream, i.e., that there are packets missing. Since there are no negative acknowledgements in TCP, the ACK for the last packet received correctly (assuming it has sequence number n) is simply being sent again to the sender. Thus, the sender knows that

the packet with sequence number $n+1$ has not been received yet. Additionally, a TCP sender entity keeps a timer for every packet it has sent. When the timer of a packet expires before an ACK is received for that packet, the packet is considered lost and will be retransmitted. When a TCP sender entity receives three duplicate ACKs for a certain packet, it assumes that the succeeding packet is lost. In this case, the sender performs a *fast retransmit*, which means that the missing packet is sent again before that packet's timer expires.

2.2 The TCP congestion control mechanism

The packet loss mentioned in the previous section typically results from the overflowing of router buffers as the network gets congested. Typically, routers accept packets until a buffer overflow occurs, after which all incoming packets are discarded until there is free buffer space again. This is known as the *drop tail-algorithm* of TCP. Note that packet retransmission after a loss occurrence is a treatment for the symptom of network congestion, but it provides no solution for the cause of network congestion. To be able to treat the cause of network congestion, a *congestion control* mechanism is needed to make sure senders do not send data at a speed that is too high and that will therefore cause congestion.

The congestion control mechanism implemented in TCP introduces two important variables: the *congestion window* and the *threshold*. The size of the congestion window imposes a constraint on the amount of data that can be sent by a TCP sender entity, that is, the amount of unacknowledged data that a sender can have within a TCP connection is limited by the size of the congestion window. The use of the threshold becomes clear when one studies the congestion control-mechanism.

Initially, the congestion window size is limited to the maximum segment size (MSS) of exactly one packet. When a packet has been sent and ACKed before a time-out occurs, the size of the congestion window is doubled, resulting in a window size of 2×MSS. Then, the sender can send two packets simultaneously, without having to wait for an ACK. When these two packets are also ACKed in time, the window size is again doubled. This process repeats itself until the congestion window size exceeds the value of the threshold variable, and is known as the *slow-start phase* of TCP congestion control. After the threshold value has been reached, the congestion window size is not doubled anymore upon the correct receipt of all packets fitting within the window, but is rather incremented by MSS. After a packet loss occurs, the congestion window size is reduced. Here, there are two possibilities:

- Either packet loss is detected via the reception of a triple-duplicate ACK, and the congestion window size is halved,
- or packet loss is detected via the reception of a connection time-out indication, and the congestion window size is reset to MSS.

In both cases, the threshold is set to half the value of the congestion window size as it was just before the detection of the packet loss. Then, the slow-start phase is repeated, until the new threshold value has been reached.

The fact that TCP increases its window size linearly when there is no packet loss (except for the slow-start phase) and decreases it by a factor 2, or resets it to MSS when packet

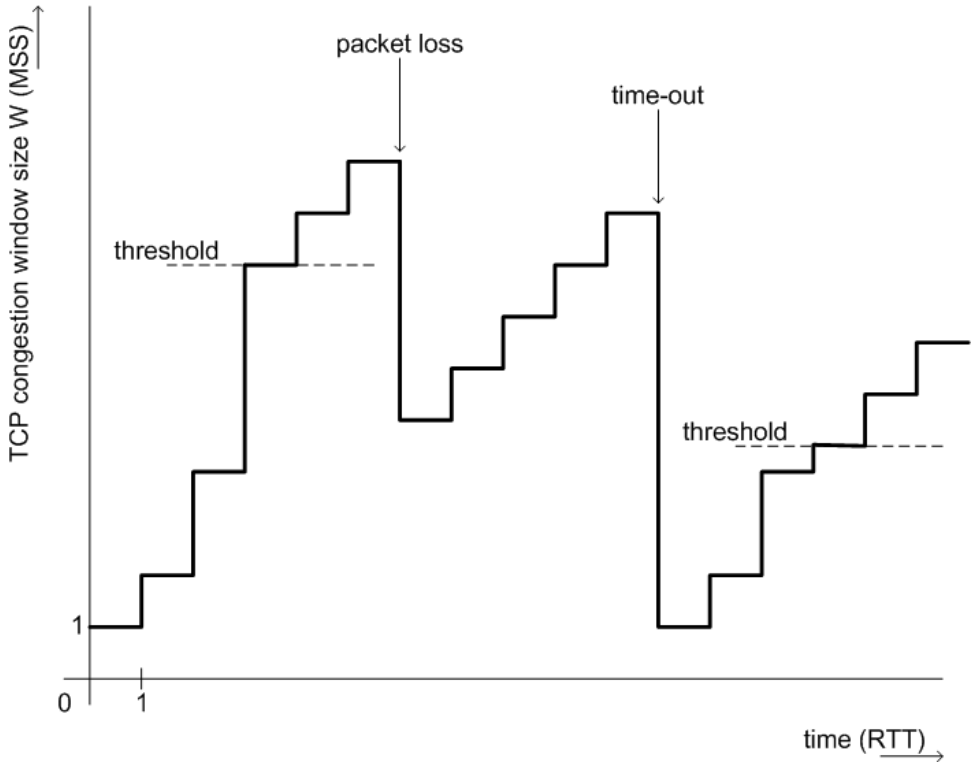

Figure 2.1: Evolution of TCP Reno window size

loss is detected, is the reason why TCP is often called an additive-increase, multiplicative decrease (AIMD) algorithm. Moreover, the version of TCP implementing the congestion control algorithm as described above is referred to as TCP Reno, and is implemented by most operating systems. Another version of TCP, called TCP Tahoe, has a slightly different congestion control algorithm. In TCP Tahoe, the slow-start phase is also included after a packet-loss, which is indicated by a triple-duplicate ACK, which is not the case in TCP Reno. A drawback of this approach is that in TCP Tahoe, when a packet is lost, the sender may have to wait a long period of time for the congestion window to be back to its original size. An example of the evolution of the congestion window size in TCP Reno is given in Figure 2.1. The multiplicative decrease of the size of the congestion window W, at the event of a packet loss, can be seen clearly.

A mechanism known as Explicit Congestion Notification (ECN) has been defined in RFC 3168 [27]. This mechanism enables routers to notify TCP sources of experienced congestion not by dropping packets, but rather by setting a flag in the IP header of packets from ECN-capable transports, which is known as *packet marking*. Throughout this report, packet dropping and packet marking will be used interchangeably, since the exact implementation of congestion notification (be it either by dropping or marking) is irrelevant to the subject. It both cases, it indicates that TCP sources are being notified of congestion experienced by a router.

Chapter 3

An overview of existing AQM algorithms

This chapter attempts to give a representative overview of existing active queue management (AQM) algorithms, as they have been presented in the literature. Since many different algorithms have been proposed, and many of these only differ in minor details, this overview does not pretend to be complete, but is limited to the most important algorithms and algorithm classes.

First, the general purpose of AQM algorithms is presented. Then, a classification of the algorithms is given, in order to provide a clear structure for the algorithms mentioned in the rest of this chapter. For each algorithm class, the most significant algorithms belonging to this class are then presented in some detail.

3.1 The general purpose of AQM algorithms

The TCP protocol, which has been presented in the previous chapter, detects congestion only after a packet has been dropped from the queue (according to the drop tail-algorithm). However, it is clearly undesirable to have large queues that are full most of the time, since this will significantly increase the delays. Therefore, and also keeping in mind the ever increasing speed of networks, it is ever more important to have a mechanism that keeps the overall throughput high, but at the same time keeps average queue size as low as possible. Note that in order to maximize the network throughput, queues should not necessarily be kept completely empty all the time, since this will result in under-utilization of the link, but in order to have a small queueing delay, the queue length should be kept sufficiently small.

In order to fulfill the above needs, a wide range of AQM algorithms have been proposed. The purpose of these algorithms is to provide a mechanism to detect network congestion early, and to start dropping packets from the router queue before this congestion will affect the network throughput too much. The definition of too much depends on the Quality of Service (QoS) to be delivered by the network.

AQM algorithms have been designed for implementation at network routers, as opposed to implementation at end nodes, such as TCP sender or receiver entities. This choice is advocated by the fact that detection of congestion can be carried out most effectively in

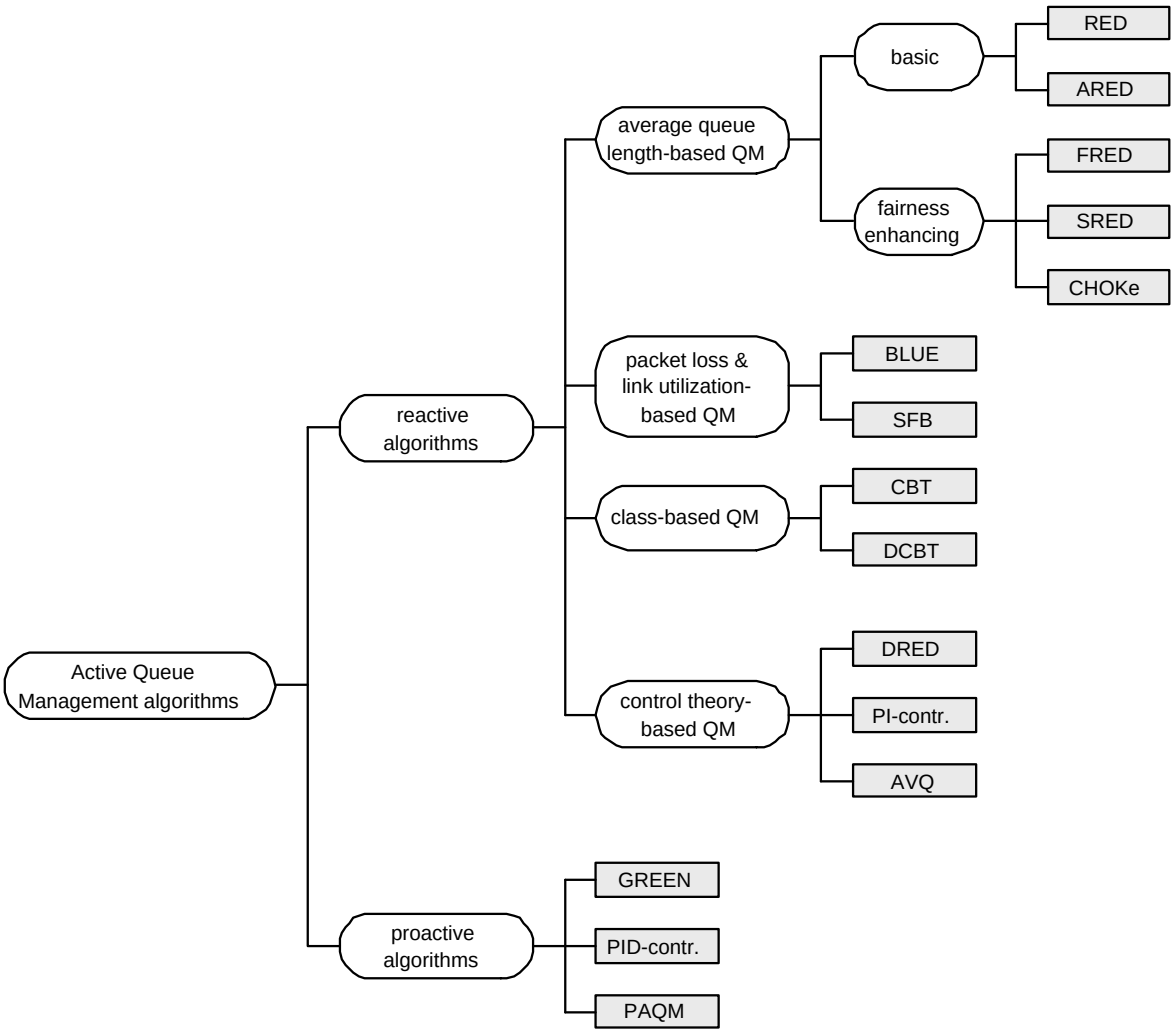

Figure 3.1: Classification of AQM algorithms

the router itself. A network router can reliably distinguish between propagation delay and persistent queueing delay. Only the router has a unified view of the queueing behavior over time; decisions about the duration and magnitude of congestion to be allowed at the router are therefore best made by the router itself.

3.2 A classification of AQM algorithms

The AQM algorithms to be presented in the rest of this chapter can be classified according to the criteria on which the decision whether or not to drop packets from the queue (when the link suffers from congestion) is being made. Four different strategies in this can be identified:

- Average queue length-based queue management (QM)
- Packet loss & link utilization-based QM
- Class-based QM

- Control theory-based QM

Alternatively, algorithms can be classified as being either reactive or proactive:

- A reactive AQM algorithm focuses on congestion avoidance, i.e., active early detection of and reaction to congestion. Congestion can occur in this case, but it will be detected early. Decisions on actions to be taken are based on current congestion.
- A proactive AQM algorithm focuses on congestion prevention, i.e., intelligent and proactive dropping of packets, resulting in prevention of congestion from ever occurring and ensuring a higher degree of fairness between flows. Decisions on actions to be taken are based on expected congestion.

Figure 3.1 gives an overview of the classification structure and the algorithms that will be addressed in the following sections. This classification is a slightly adapted version of the one presented in [26]. All of the following sections, except for the last one, are devoted to reactive AQM algorithms, ordered according to the classification as given above. The last section describes a few proactive algorithms, which are not classified according to their packet drop strategy.

3.3 Average queue length-based algorithms

This class of reactive AQM algorithms bases the decision on whether or not to drop packets from the queue on the observed average queue length. The algorithms in this class can be divided further based on whether or not the algorithm pays special attention to fairness as it comes to the distribution of available bandwidth over the active data flows.

3.3.1 Random Early Detection (RED)

RED is one of the first AQM algorithms ever developed, proposed in [2] in 1993. It has been widely used with TCP and has been recommended by IETF. RED has served as basis for many other algorithms that are to be discussed later in this chapter. The algorithm detects incipient congestion by computing the average queue size at the router. When this average queue size exceeds a certain preset threshold, the router drops or marks each arriving packet with a certain probability p_a, which is a linear function of the average queue size. Connections are notified of congestion either by dropping packets arriving at the router, or rather by setting a bit in packet headers (which is referred to as 'marking a packet'). RED can be specified in pseudo-code as in Figure 3.2, where min_{th} and max_{th} are minimum and maximum thresholds, which can be set by the algorithm user, and which are both smaller than the maximum queue size allowed by the router. Whenever a packet arrival is detected by a router implementing RED, the average queue size q_{avg} is calculated using a low-pass filter to get rid of the potentially significant influence that short-term traffic bursts could have on the average queue size. Calculation of q_{avg} is carried out using

$$q_{avg} = (1 - w_q)q_{avg} + w_q q \tag{3.1}$$

where q is the instantaneous queue length as observed at the router, and w_q is the weight applied by the low-pass filter to the 'old' average queue size. By increasing this value w_q, the

```
for each packet arrival:
   calculate the average queue size q_avg
   if min_th ≤ q_avg < max_th
      calculate probability p_a
      with probability p_a:  mark/drop the arriving packet
   else if max_th ≤ q_avg
      mark/drop the arriving packet
   else
      do not mark/drop packet
```

Figure 3.2: Pseudo-code of the RED algorithm

influence of the current queue size on q_{avg} is also increased. This also increases the influence of traffic bursts, or periods of relatively little traffic on the average queue length. After q_{avg} has been calculated, it is compared to two threshold values, min_{th} and max_{th}. When q_{avg} is smaller than min_{th}, no packets are marked, and each arriving packet is appended at the end of the queue. When q_{avg} lies between min_{th} and max_{th}, each arriving packet is marked with probability p_a. When q_{avg} is greater than max_{th}, all incoming packets are marked with probability 1. The marking probability p_a is calculated as follows:

$$p_a = p_{max} \frac{(q_{avg} - min_{th})}{(max_{th} - min_{th})} \tag{3.2}$$

In [2], p_a is increased even further as a function of the number of packets that have arrived since the last packet was marked, resulting in a new marking probability p_b. Since this feature is often omitted in the literature, whenever the RED dropping probability is mentioned here, it refers to p_a; we will use p_a as the RED dropping probability in the rest of this report as well. Figure 3.3 shows a graphical representation of the possible values for the marking probability p_a.

Despite being widely used in combination with TCP for several years, RED has not found acceptance in the Internet research community. The reason for this are several drawbacks of the algorithm, of which the most important are:

- The packet loss fraction is equal for all flows, regardless of the bandwidth used by each flow;
- No restriction exists against aggressive, non-adaptive flows, which has a negative impact on well-behaving, adaptive flows;
- It is difficult to parameterize RED queues to give good performance under different congestion scenarios;
- The equilibrium queue length strongly depends on the number of active TCP connections;
- Simulations have shown that RED does not provide clear advantages over the TCP drop tail-algorithm in realistic scenarios.

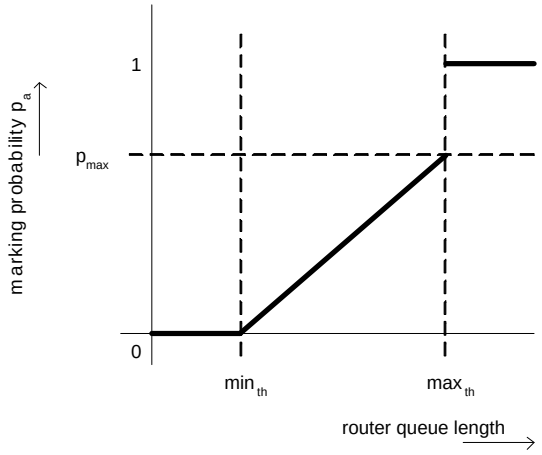

Figure 3.3: The RED marking probability p_a

These drawbacks have been the main reason for the development of a vast collection of improved AQM algorithms, of which the most significant ones are presented in the rest of this chapter.

3.3.2 Average Rate Early Detection (ARED)

ARED, proposed in [3], uses the average packet enqueue rate as a congestion indicator, and signals end hosts of incipient congestion with the objective of reducing the packet loss ratio and improving link utilization. Unlike most other AQM algorithms based on average queue length, ARED attempts to control the rate of queue occupancy change rather than controlling the queue occupancy itself. It intends to keep the queue length stabilized at an operational point at which the aggregate packet enqueue rate is approximately equal to or slightly below the link capacity. An incoming packet is dropped with a probability p, which is calculated according to the pseudo-code as seen in Figure 3.4. This pseudo-code is executed at every packet arrival event. From this pseudo-code, it can be derived that there are two situations in which packet loss might occur in ARED:

- The estimated packet enqueue rate R is higher than the link capacity μ (the congestion condition), or
- the instantaneous queue length is larger than a predetermined value L and the ratio of R to μ is larger than a predetermined value ρ (the congestion alert-condition).

Here, L is the average queue length to which the ARED algorithm automatically converges, and ρ is usually set to 0.95. A packet is dropped when one of the two above conditions holds, and p is larger than a uniformly distributed random variable. In addition, the probability p is increased with an amount d_1 whenever either the congestion condition or the congestion alert-condition holds for a time span longer than a preset $freeze\_time$. When the link has been idle for a $freeze\_time$ amount of time, p is decreased by an amount d_2, which is typically an order of magnitude smaller than d_1.

```
Upon every packet arrival:
   if (link_idle)
      reset link idle timer;
      link_idle = false;
   if (R > μ || (q_len > L  &&  R > ρ × μ))
      drop packet with probability p;
      if (current_time − last_update > freeze_time)
         p = p + d1;
         last_update = current_time;

When link becomes idle:
   set link idle timer to freeze_time;
   link_idle = true;

When link idle timer expires:
   p = p − d2;
   set link idle timer to freeze_time;
```

Figure 3.4: Pseudo-code of the ARED algorithm

3.3.3 Flow Random Early Drop (FRED)

A problem with the algorithms described in sections 3.3.1 and 3.3.2 is that they do not explicitly protect fragile, low-bandwidth flows from aggressive flows that tend to consume all of the available bandwidth. Aggressive flows typically have small RTTs, which allows them to increase their congestion window size faster after loss. This loss is caused by these aggressive flows with a reasonably high probability, and has a strongly negative influence on available bandwidth for fragile flows. FRED has been proposed in [4] as an alternative to RED in order to protect these fragile flows and thus to maintain a higher degree of fairness. It attempts to do so by using per-active-flow accounting, to impose on each flow a loss rate that depends on that flow's buffer usage.

FRED acts just like RED, but with the following additions: it introduces the parameters $min_q(i)$ and $max_q(i)$, which represent goals for the minimum and maximum number of packets that each flow i is allowed to buffer. FRED also has a global variable avg_{cq}, which is an estimate of the average per-flow buffer count; flows with fewer than avg_{cq} packets queued are favored over flows with more packets in the queue. FRED also maintains a variable $q_{len}(i)$, a count of buffered packets for each flow that currently has any packets buffered. Finally, FRED maintains a variable $strike(i)$ for each flow, which is a count of the number of times the flow has failed to respond to congestion notification; FRED penalizes flows with high $strike(i)$ values.

Despite the additional fairness provisions taken by FRED, the algorithm has a known drawback concerning this fairness. FRED has a potential problem: its TCP-favored per-flow punishment could unnecessarily discourage responsive UDP flows. Under FRED, incoming packets for a well-behaved TCP flow consuming more than their fair share are randomly dropped applying RED's drop rate. However, once a flow, although flow-controlled, is marked as a non-TCP friendly flow, it is regarded as an unresponsive flow and all incoming packets

of the flow are dropped when it is using more than its fair bandwidth share. As a result, a responsive UDP flow, which may have a higher chance to be marked, will experience more packet loss than a TCP flow and will thus be forced to do with less than its fair share of bandwidth.

Another reasonably important drawback of the FRED algorithm is the fact that the per-active-flow accounting strategy requires a certain amount of overhead data to be stored for each active flow. This could result in significant additional processing delay at a router, especially when there are many flows active at the same time.

3.3.4 Stabilized RED (SRED)

The SRED algorithm has been presented in [5] as an attempt to stabilize the TCP router buffer occupation at a level independent of the number of active connections, by estimating the number of active connections or flows statistically. The goal of the algorithm is to identify flows that are taking more than their fair share of bandwidth, and to allocate a fair share of bandwidth to all flows, without requiring too many computations. For this purpose, it uses a finite *zombie list*, in which it stores information about recently active flows, together with a *count* variable and a time stamp, which is used for administrative purposes, for each flow in the zombie list. The list starts out empty, and whenever a packet arrives, its packet flow identifier (source address, destination address, etc.) is added to the list. Once the zombie list is full, every packet arriving at the router is compared to a randomly drawn flow from the list (the 'zombie'). After this comparison, one out of two possible actions will be taken:

- When the arriving packet's flow matches the zombie (a 'hit'), that zombie's *count* variable is increased by one, and the time stamp is set to the latest packet arrival time, or,
- when the arriving packet's flow does not match the zombie (a 'no hit'), with a preset probability p the flow of the new packet overwrites the zombie chosen for comparison. Its *count* variable is initialized to 0.

The number of active flows, which is used by SRED to stabilize buffer occupation, can then be estimated using the average 'hit' rate. How this is done is presented in [5]. The *count* variable is used in the identification of misbehaving flows: a flow with a high *count* value is more likely to be a misbehaving flow, and will therefore experience a higher loss probability.

3.3.5 CHOKe

The basic idea behind the CHOKe algorithm [6] is that the contents of a buffer form a sufficient statistic about the incoming traffic, and that these contents can be used in a simple fashion to penalize misbehaving flows. CHOKe attempts to provide a fair queueing policy by discriminating against these misbehaving flows. When a packet arrives at a congested router, CHOKe draws a packet at random from the buffer and compares it with the arriving packet. If they both belong to the same flow, then they are both dropped, else the randomly chosen packet is left intact and the arriving packet is admitted into the buffer with a probability p that depends on the level of congestion (where p is computed in exactly the same way as in RED). The reason for doing this is that the router buffer is assumed to be more likely to

contain packets belonging to a misbehaving flow, and, hence, these packets are more likely to be chosen for comparison. Furthermore, packets belonging to a misbehaving flow are assumed to arrive more numerously and are more likely to trigger comparisons.

3.4 Packet loss & link utilization-based algorithms

The main idea behind this class of algorithms is to perform active queue management based on packet loss and link utilization rather than on the instantaneous or average queue lengths. Algorithms in this class maintain a single probability p, which is used to mark (or drop) packets when they are queued. If the queue is continually dropping packets due to buffer overflow, p is incremented, thus increasing the rate at which congestion notifications are sent back. Conversely, if the queue becomes empty or if the link is idle, p is decremented. Note that, in contrast to the class of AQM algorithms discussed in the previous section, no queue occupancy information is used.

3.4.1 BLUE

The BLUE algorithm [7] maintains a single probability, p_m, which it uses to mark (or drop) packets when they are enqueued. If the queue is continuously dropping packets due to buffer overflow, BLUE increments p_m, thus increasing the rate at which it sends back congestion notification. As has been said above, if the queue becomes empty or if the link is idle, p_m will be decremented. This allows BLUE to "learn" the correct rate at which it has to send back congestion notification. The algorithm can be specified in pseudo-code as in Figure 3.5.

```
Upon packet loss event:
   if ((now − last_update) > freeze_time)
      p_m = p_m + d_1
      last_update = now

Upon link idle event:
   if ((now − last_update) > freeze_time)
      p_m = p_m − d_2
      last_update = now
```

Figure 3.5: Pseudo-code of the BLUE algorithm

The $freeze\_time$ parameter should be a fixed variable, which is random for each router, in order to avoid global synchronization (the undesired phenomenon where each flow updates its window size at the same time). Note that the BLUE algorithm looks much like the ARED algorithm specified in section 3.3.2.

A slightly different variant of the BLUE algorithm, in which the marking probability is updated when the queue length exceeds a certain threshold value, has also been proposed. This modification allows room to be left in the queue for transient bursts, and allows the queue to control queueing delay when the size of the queue being used is large.

3.4.2 Stochastically Fair BLUE (SFB)

Since the original BLUE algorithm does not ensure fairness among flows, and more specifically does not provide protection for fragile flows against aggressive, non-responsive flows, SFB has been proposed in [7]. SFB is an algorithm that identifies and rate-limits non-responsive flows based on accounting mechanisms similar to those used with BLUE. It maintains $N \times L$ accounting bins, which are organized in L levels containing N bins each. Additionally, SFB maintains L independent hash functions, each associated with one level of accounting bins. Each hash function maps a packet flow into one of the N accounting bins in that level. The accounting bins are used to keep track of queue occupancy statistics of packets belonging to a particular bin. Each bin in SFB keeps a marking (dropping) probability p_m, which is the same as in BLUE, and which is updated based on bin occupancy. When a packet arrives at the queue, it is hashed into one of the N bins in each of the L levels. If the number of packets mapped to a bin reaches a certain threshold (i.e., the size of the bin), the marking probability p_m for that bin is increased. If the number of packets in a bin drops to zero, p_m for that bin is decreased. The pseudo-code specification of SFB is given in Figure 3.6.

```
define B[l][n] = L × N array of bins (L levels, N bins per level)

at packet arrival:
   calculate hash functions h_0, h_1, ..., h_{L-1};
   update q_len for each bin at each level;
   for i = 0 to L − 1
      if (B[i][h_i].q_len > bin_size)
         B[i][h_i].p_m += delta;
         drop packet;
      else if (B[i][h_i].q_len == 0)
         B[i][h_i].p_m −= delta;
   p_min = min(B[0][h_0].p_m, ..., B[L][h_L].p_m);
   if (p_min == 1)
      limit flow's sending rate;
   else
      mark / drop packet with probability p_min;
```

Figure 3.6: Pseudo-code of the SFB algorithm

The main idea behind SFB is that a non-responsive flow quickly drives p_m to 1 in all of the L bins it is hashed into. Responsive flows may share one or two bins with non-responsive flows, however, unless the number of non-responsive flows is extremely large compared to the number of bins, a responsive flow is likely to be hashed into at least one bin that is not polluted with non-responsive flows. This bin thus has a normal p_m value. The decision to mark a packet is based on p_{min}, the minimum p_m value of all bins to which the flow is mapped into. If p_{min} is equal to 1, the packet is identified as belonging to a non-responsive flow, and the available bandwidth for that flow is then limited.

3.5 Class-based algorithms

For an algorithm belonging to this class, the treatment of an incoming packet (under congestion circumstances) depends on the *class* this packet belongs to. Theoretically, there are many possible class definitions, but in practice, it is most common to categorize incoming packets based on the transport protocol (i.e., TCP or UDP) that has been used to send the packet. With this type of algorithms, for every non-TCP class, a threshold is defined, setting a limit to the maximum amount of packets that a certain class can have in the queue.

3.5.1 Class-Based Threshold (CBT)

The CBT algorithm, proposed in [8], divides incoming packets into two classes: TCP packets and UDP packets. The goal of the algorithm is to reduce congestion in routers and to protect TCP flows from UDP flows, while also ensuring acceptable throughput and latency for well-behaved UDP flows. This is done by constraining the average number of non-TCP packets that may reside simultaneously in the queue. CBT attempts to realize a "better than best effort"-service for well-behaved multimedia flows that is comparable to that achieved by a packet or link scheduling discipline. However, CBT does this by active queue management rather than by scheduling. The algorithm is an attempt to construct an AQM scheme that will maintain the positive features of RED, limit the impact of unresponsive flows, but still allow UDP flows access to a configurable share of the link bandwidth. Moreover, it tries to do this without having to maintain per-flow state information in the router. Two different CBT variants can be identified:

CBT with RED for all: When an UDP packet arrives, the weighted average number of packets enqueued for the appropriate class is updated and compared against the threshold for the class, to decide whether to drop the packet before passing it to the RED algorithm. For the TCP class, CBT does not apply a threshold test, but directly passes incoming packets to the RED test unit.

CBT with RED for TCP: only TCP packets are subjected to RED's early drop test, and UDP packets that survive a threshold test are directly enqueued to the outbound queue. Another difference from CBT with RED for all is that RED's average queue size is calculated only using the number of enqueued TCP packets. CBT with RED for TCP is based on the assumption that tagged (multimedia) UDP flows as well as untagged (other) UDP flows are mostly unresponsive, and it is of no use to notify these traffic sources of congestion earlier.

3.5.2 Dynamic CBT (DCBT)

DCBT [9] is an extension of "CBT with RED for all". This algorithm fairly allocates the bandwidth of a congested link to the traffic classes by dynamically assigning the UDP thresholds such that the sum of the fair share of flows in each class is assigned to the class at any given time. The fair class shares are calculated based on the ratio between the number of active flows in each class. The key differences between CBT and DCBT are:

- Dynamically moving fairness thresholds, and
- the UDP class threshold test that actively monitors and responds to RED indicated congestion.

In addition to this, DCBT adopts a slightly different class definition compared to CBT: unlike the class categorization of CBT in which responsive multimedia flows are not distinguished from unresponsive multimedia flows (they are all tagged), DCBT classifies UDP flows into responsive multimedia (tagged) UDP flows and other (untagged) UDP flows. In order to calculate the per-class thresholds, DCBT needs to keep track of the number of active flows in each class.

3.6 Control theory-based algorithms

This category of algorithms has been developed using classical control theory techniques. The queue length at the router is regulated to agree with a desired value, by eliminating the "error", that is, the difference between the queue length and this desired value (see [10] for more details). A typical feedback control system for an AQM algorithm consists of

- a desired queue length at the router q_{ref} (i.e., a reference input),
- the queue length at a router as a system variable (i.e., a controlled variable),
- a system, which represents a combination of subsystems (such as TCP sources, routers and receivers),
- an AQM controller, which controls the packet arrival rate to the router queue by generating the packet drop probability as a control signal, and
- a feedback signal, which is a sampled system output used to obtain the control error term.

AQM algorithms based on control theory attempt to maintain the instantaneous queue length as close as possible to q_{ref}. The packet drop probability p is adjusted periodically, based on a combination of both the current queue length deviation (the error signal) and the sum of previous queue length deviations from q_{ref}. As a result of these periodic updates, an AQM control system can be modelled in a discrete-time fashion. Figure 3.7 presents a graphical representation of a typical control theory-based AQM system. Apart from the typical control system parts mentioned above, this system also includes a low-pass filter, which will be introduced in 3.6.1, and influences from external disturbances, such as noise, which have an impact on the controlled output.

3.6.1 PI-controller

A linearized dynamic model of TCP, which has been presented in [14], has been chosen as the basis for the proportional-integral (PI-) controller algorithm [12]. PI-controller is basically proposed as a combination of two controller units, a *proportional controller* and an *integral controller*. In the proportional controller, the feedback signal is simply the regulated output (i.e., the queue length) multiplied by a proportional gain factor α:

$$p(n) = \alpha e(n) \tag{3.3}$$

where $e(n)$ is the error signal at time n, i.e., the difference between the actual queue length at time n and the reference queue length q_{ref}. This error signal is usually normalized by

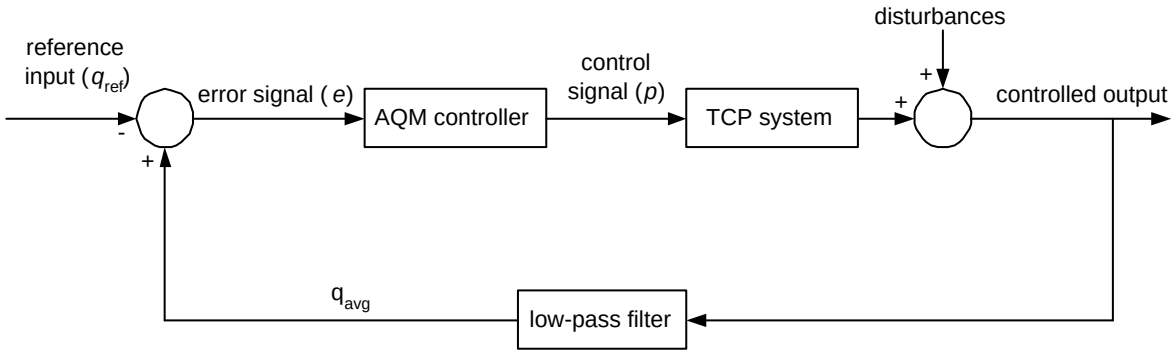

Figure 3.7: TCP and AQM as a closed-loop control system

the router buffer size B, since queue length fluctuations (and hence the error signal) grow linearly with buffer size. This normalization step is often used in practice, and therefore in the rest of this chapter, $e(n)$ will be assumed to be the normalized error signal $e(n)/B$. The α factor 'steers' the queue length to its desired length (the reference input). One of the drawbacks of the use of a purely proportional controller is the fact that its resulting steady-state output will never be equal to the reference output, due to the multiplicative nature of this mechanism. Therefore, an integral controller has been added to the algorithm. The purpose of this integral controller is to remove the *steady-state regulation error* of an AQM algorithm, which is defined as the difference between the steady-state output and the desired reference value. The generic structure of a PI-controller is given by

$$p(n) = \alpha e(n) + \beta \int_0^n e(\tau) d\tau \tag{3.4}$$

where β is the integral gain constant. The proportional and the integral part of this equation can be identified easily. Note that when discrete time systems are being considered, β is often replaced by β/T_I, where T_I is the integral time. In [15], the structure of the PI-controller under attention has been put differently, namely as

$$p(n) = p(n-1) + \kappa e(n) + \lambda(e(n) - e(n-1)) \tag{3.5}$$

In [15], it is claimed that this is a representation of a PD-controller, although with some mathematical transformations, this equation can be rewritten as (3.4). The pseudo-code of the digital implementation of PI-controller, which should be functionally equal to (3.4), is given as

$$p(n) = p(n-1) + \alpha e(n) + \beta e(n-1) \tag{3.6}$$

When we suppose $\alpha = \kappa + \lambda$, and $\beta = \lambda$, (3.5) and (3.6) are equivalent, implying (3.5) is equivalent to (3.4). The integral part of (3.5) is not as clearly visible as in (3.4), but it is present in the $p(n-1)$ element. The use of this element implies a summation over previous values of $p(n)$, and implicitly over previous values of $e(n)$, which corresponds to the integral term of (3.4). The fact that the system described in [15] is really a PI-controller in nature, instead of the PD-controller that it claims to be, is confirmed by the fact that this system is based on a PI-controller mentioned in [11]. The authors of [15] claim that the system from

[11] is a simple P-controller, which is the reason for erroneously claiming their own system is a PD-controller.

A considerable difference between the systems from [12] and from [15] is the fact that the latter uses a low-pass filter in the calculation of the average queue length, whereas the former deliberately refrains from doing this. The first step in the application of the low-pass filter in [15] is the sampling of the current queue length $q(n)$ from the system. This $q(n)$ is then used to compute a filtered queue length $q_{avg}(n)$ using

$$q_{avg}(n) = (1 - \chi)q_{avg}(n - 1) + \chi q(n) \tag{3.7}$$

where χ is a preset filter gain, which affects the response speed of the queue length filter. The filtered queue length $q_{avg}(n)$ is then used in the calculation of the error signal $e(n)$, which is given by

$$e(n) = q_{avg}(n) - q_{ref} \tag{3.8}$$

The intention of the application of a low-pass filter is to remove undesired effects of traffic bursts on the average queue length. The system described in [12] refrains from using the filter because:

> "Although one of the design goals of the low-pass filter was to let transient bursts pass through, from a control standpoint the averaging can lead to instability and low frequency oscillations in the regulated output. In fact, the averaging mechanism is built into the queue dynamics, and the queue essentially acts like a low-pass filter."

3.6.2 Dynamic RED (DRED)

DRED [11] also uses simple control theory techniques to randomly discard packets with a load-dependent probability whenever a router becomes congested. The main goal of this algorithm, which is very similar to RED but uses a different (i.e. control theoretic) approach, is to stabilize the router buffer occupancy at a level independent of the number of active TCP connections. The operation of DRED is quite similar to the operation of a PI-controller (see the previous section), and can also be specified as a sequence of steps, carried out at time n. First, the the current queue length $q(n)$ is sampled. Then, the current error signal $e(n)$ is computed as $e(n) = q(n) - q_{ref}$, which is very similar to (3.8), with the difference that no low-pass filter has been applied yet to the observed queue length, meaning $q(n)$ represents the instantaneous queue length. To this error signal $e(n)$, a low-pass filter is then applied using

$$\hat{e}(n) = (1 - \chi)\hat{e}(n - 1) + \chi e(n) \tag{3.9}$$

where χ is again the preset low-pass filter gain. The dropping probability $p(n)$ can now be computed as

$$p(n) = \min\{\max[p(n - 1) + \alpha\hat{e}(n), 0], p_{max}\} \tag{3.10}$$

where p_{max} is again the preset maximum dropping probability, and α is again the proportional gain. The dropping probability $p(n)$ is then stored for use at time $n+1$, when a new probability

$p(n+1)$ will be computed using the steps defined above. For this purpose, $\hat{e}(n)$ needs to be stored as well. Simulations have shown that DRED is indeed able to stabilize the router queue length close to a predefined reference value, and accommodates well to traffic bursts without dropping too many incoming packets.

Note that when it comes to the application of the low-pass filter, DRED only differs from PI-controller in the moment at which the filter is applied. In DRED, it is applied to the error signal $e(n)$, whereas in PI-controller it is applied to the average queue length $q(n)$, which is then used to derive the error signal. Since the application of a low-pass filter is a linear operation, this does not make a difference when comparing DRED and PI-controller.

3.6.3 Adaptive Virtual Queue (AVQ)

The AVQ algorithm, proposed in [16], maintains a virtual queue, whose capacity (referred to as the *virtual capacity*) is less than the actual capacity of the router queue implementing AVQ. When a packet arrives at the router, the virtual queue is also updated to reflect a new arrival. Packets in the real queue are marked or dropped when the virtual buffer overflows. The virtual capacity at each link is then modified such that the total flow entering each link achieves a desired utilization of that link. Thus, instead of directly controlling the real queue length using a dropping probability, AVQ controls the virtual queue capacity, which implicitly imposes a dropping probability on packets in the real queue. No dropping probability is calculated directly.

Figure 3.8 shows an AVQ specification in pseudo-code, where V_Q is the number of bytes currently in the virtual queue, b is the number of bytes of the arriving packet, B is the total (real) buffer size, and the *last_arrival* variable is used to store the time of the most recent packet arrival. The '`update` V_Q' event consists of updating the variable holding the current virtual queue length, since this one might have changed since the previous packet arrival event, e.g., because of packets being served that have left the queue. Note that this is different from updating the virtual capacity. See [16] for more details on this.

```
at each packet arrival do:
   update V_Q;
   if (V_Q + b > B)
      mark packet in real queue;
   else
      V_Q ← V_Q + b;
   endif
   update virtual_capacity;
   update last_arrival;
```

Figure 3.8: Pseudo-code of the AVQ algorithm

3.7 Proactive algorithms

One important drawback of the AQM algorithms presented in the previous sections is that their congestion detection and control functions depend only on either the current queue status, or the history of the queue status (e.g., the average queue length). Hence, the congestion

detection and control in these algorithms are reactive to current or the past congestion, but do not respond proactively to incipient congestion. For example, the congestion detection method in RED can detect and respond to long-term traffic patterns using the exponentially weighted moving average queue lengths. However, it is unable to detect incipient congestion caused by short-term traffic load changes. In this case, the implicit congestion notification sent back to end hosts by a packet drop may be an inappropriate control signal, and can possibly make the congestion situation worse.

This section describes a few proactive QM algorithms, which are designed to react on incipient congestion, rather than react on current congestion, as the reactive AQM algorithms presented in the previous sections do. Basically, this means that the packet drop probability at time t is not only a function of the current system parameters (thus, the parameter values at time t) and/or previous values (at time $t-1$, $t-2$, etc.), but also on (estimations of) future parameter values (e.g., at time $t+1$).

3.7.1 GREEN

The GREEN (Generalized Random Early Evasion Network) algorithm [17] applies knowledge of the steady state behavior of TCP connections to intelligently and proactively drop packets, and thus attempts to prevent congestion from ever occurring and to ensure a higher degree of fairness between flows. The algorithm operates as follows. The bandwidth BW of a TCP connection can be approximated by

$$BW = \frac{\text{MSS} \times c}{\text{RTT} \times \sqrt{p}}, \tag{3.11}$$

where p is the packet loss probability and c is a constant depending on the acknowledgement strategy being used, as well as on whether packets are assumed to be lost periodically or randomly. Now consider a scenario with N active flows on a link of capacity L. The fair-share throughput of each flow is then L/N. The loss probability p can now be defined as

$$p = \left(\frac{N \times \text{MSS} \times c}{L \times \text{RTT}} \right)^2 \tag{3.12}$$

When this value is used as the dropping probability for congestion notification, GREEN forces flows to send at their fair-share rate. Since p depends on the number of flows and the RTT of each flow, congestion notification is more aggressive for large N and small RTT. By including the RTT as an inverse parameter in (3.12), GREEN also eliminates the bias of favoring TCP connections with smaller RTT with respect to throughput (flows with smaller RTT can increase their window size faster due to this smaller RTT and are therefore more aggressive. Hence, these flows grab more than their fair share of bandwidth, which leads to this bias). GREEN does not require any information about the congestion window size, which is usually hard to calculate and varies with the topology of the network. No end-to-end modifications are required either. The implementation of GREEN relies on the knowledge of flow RTTs and the total number of active flows N. In [17], it is assumed that routers have knowledge of these parameters, but at the end of the paper, some hints on how the GREEN algorithm can estimate these parameters itself (thereby removing the need for routers to have knowledge about these parameters) are briefly presented.

3.7.2 PID-controller and PAQM

The PID (proportional-integral-derivative) controller algorithm [10] is constructed by combining a PI-controller unit (see Section 3.6.1) with a PD-controller unit. It thereby combines the advantages of the PI-controller (removal of the steady-state error, at the cost of increased response time) with the most important advantage of a PD-controller unit, which is the improving of damping and of speed of response. The design of PID-controller is based on a control theoretic plant model of TCP, which will be presented later on in Section 4.3.

A generic PID control equation, which, in combination with the plant model of TCP, serves as the basis for the PID-controller algorithm, can be given as

$$p(n) = \alpha e(n) + \beta \int_0^n e(\tau)d\tau + \gamma \frac{de(n)}{dn} \tag{3.13}$$

The proportional, integral and derivative parts of the equation can be clearly identified here. A variant of PID-controller, the Pro-Active Queue Management or PAQM algorithm, does not rely on assumptions on the plant dynamic model, i.e. on the TCP flow dynamic model. Instead, it has been designed using the direct digital design method, based on a discrete representation of the PID control equation in (3.13) and on certain properties of network traffic. A detailed specification of the PAQM algorithm can also be found in [10]. Simulations presented in [10] show that both algorithms outperform reactive AQM algorithms such as RED in terms of queue length dynamics, packet loss rates, and link utilization, which in turn justifies the fact that a proactive queue management approach has some considerable benefits over "traditional" reactive queue management algorithms.

3.8 Summary of algorithm features

The most important features of the algorithms presented in this chapter have been summarized in Table 3.1, in order to give the reader a quick overview of these algorithms.

Average queue length-based QM	
Algorithm	**Characteristic features**
RED	Has served as a basis for all other algorithms mentioned in this chapter. It has been widely used in combination with TCP, and is known to have several drawbacks, which have lead to the design of many other, improved algorithms.
ARED	Attempts to control the rate of queue occupancy change, rather than controlling queue occupancy change itself.
FRED	Provides explicit protection for fragile flows and thus tries to maintain a higher degree of fairness. It attempts to do so by using per-flow accounting, which imposes significant overhead. Another drawback of FRED is the possible unfairness against well-behaving interactive multimedia UDP flows.
SRED	Attempts to stabilize the router queue length at a level independent of the number of active flows. It does so by estimating the number of active flows using a so-called zombie list.
CHOKe	Assumes that the contents of the queue form a 'sufficient statistic' about the incoming traffic, since the queue is more likely to contain packets from misbehaving flows. It attempts to provide fairness by discriminating against these misbehaving flows.

Packet loss & link utilization-based QM	
Algorithm	**Characteristic features**
BLUE	Tries to 'learn' the correct packet dropping rate by increasing it when congestion occurs, and decreasing it when the link is idle.
SFB	Attempts to ensure fairness by identifying and rate-limiting non-responsive flows based on accounting mechanisms that are similar to those used in BLUE. When a flow drives the dropping rate to 1 in all the bins it is hashed into, it is identified as being non-responsive, and the available bandwidth for that flow can be limited.

Class-based QM	
Algorithm	**Characteristic features**
CBT	Packet flows are divided into TCP and UDP flows. CBT tries to protect TCP flows from all UDP flows, and also to ensure an acceptable throughput for well-behaving UDP flows. The second is done by limiting the average number of non-TCP packets that can be in the queue simultaneously. Two variants of CBT can be identified: 'CBT with RED for all' and 'CBT with RED for TCP'.
DCBT	This is an extension of 'CBT with RED for all', and differs from it by dynamically determining fairness thresholds and the use of a separate UDP class threshold test. It also employs a slightly different class definition compared to CBT.

Control theory-based QM	
Algorithm	**Characteristic features**
PI-controller	Uses a proportional controller to steer the queue length to its desired length, in combination with an integral controller that is used to remove the steady-state regulation error in the queue length.
DRED	Operation is similar to RED, except for the control theoretic approach. Its goal is to stabilize the router queue occupancy at a level independent of the number of active TCP connections.
AVQ	Maintains a virtual queue, whose capacity is less than the actual capacity of the router queue. This virtual queue is updated whenever a packet arrives at the real queue, and gives an indication of upcoming congestion. AVQ attempts to regulate queue capacity instead of queue length.

Proactive QM	
Algorithm	**Characteristic features**
GREEN	Applies knowledge of the steady state behavior of TCP connections to attempt to prevent congestion from ever occurring and to ensure a higher degree of fairness. The loss probability is calculated using a well-known bandwidth approximation formula.
PID-controller	Created by combining a PI-controller with a PD-controller, thereby combining the advantages provided by both algorithms.
PAQM	A variant of PID-controller that does not rely on assumptions on the TCP flow dynamic model. It has been designed using the direct digital design method instead.

Table 3.1: Summary of AQM algorithm features

Chapter 4

AQM modeling approaches

The algorithms presented in Chapter 3 have certain properties in common, but are also different in several aspects, such as speed, memory utilization (size of the states information to be stored) and simplicity. When one needs to choose a certain AQM algorithm for implementation in a network, a comparison of possibly suitable algorithms needs to be carried out, in order to decide which one will fit best in the network, and which one delivers the desired result, or comes closest. Comparison of the various AQM algorithms can be carried out in several ways, of which simulation and mathematical or numerical analysis are the most common ones. Various analytical approaches have been used in research, of which three will be presented in this chapter. Each one of these approaches has its own particular characteristic features, which make each of them suitable for TCP/AQM behavior modeling in its own way. A logical first step in the comparison presented here is to identify for each of the three approaches in what way it has been used to model TCP/AQM behavior. Therefore, each modeling approach will be discussed briefly, and an example of its use with respect to AQM algorithms will be presented. As an introduction to this overview, the benefits of using an analytical approach over algorithm simulation will be discussed, so as to justify the use of mathematical and numerical analysis for the rest of this report.

4.1 Analysis vs. simulation

In this section, the benefits and drawbacks of mathematical analysis over a computer simulation of an AQM algorithm will be presented. First, the two terms are introduced properly, after which a short comparison will follow. The outcomes of this comparison provide an argument for the use of mathematical analysis in the rest of this report.

4.1.1 Computer simulation

Many different definitions of computer simulation have been used in the literature. With respect to the simulation of AQM algorithms, the following definition applies well:

> '...a computer program that defines the variables of a system, the range of values those variables may take on, and their interrelations in enough detail for the system to be set in motion to generate some output. The main function of a computer simulation is to explore the properties and implications of a system that is too complex for logical or mathematical analysis... A computer simulation

> generally has an ad hoc or 'home-made' quality that makes it less rigorous than a mathematical model.' [1]

The most widely used software tool for the simulation of AQM algorithms is the `ns-2` simulator [18]. Its developers describe `ns-2` as a discrete-event simulator, targeted at networking research. The tool provides support for simulation of TCP, routing, and multicast protocols over wired and wireless networks. Additionally, `ns-2` supports several algorithms in routing and queueing.

4.1.2 Benefits of mathematical analysis

Mathematical analysis (which will be referred to as 'analysis' in the rest of this report for convenience) has been defined as 'the resolving of problems by reducing the conditions that are in them to equations' [2]. The goal of analysis with regard to the subject of this report is to build a mathematical model of the behavior of an AQM algorithm (or a group of algorithms that have similar behavior), so that this algorithm can be analyzed and compared to similar algorithms using the properties and the outcomes of the model.

Using an analytical model of an AQM algorithm has several benefits over using the results of a simulation of that algorithm. First of all, when one wants to obtain qualitative information about a given algorithm, the use of an analytic model can result in vast analysis speed improvements. For example, a sensitivity analysis (in which the sensitivity of the algorithm outcomes to changes in the algorithm parameters is being determined) takes a few minutes using a model, but easily takes up to a few hours with a simulation in `ns-2`. Another aspect in which analytic models outperform computer simulations is accuracy. Certain elements of an AQM algorithm, such as the packet arrival process at a gateway, contain stochastic parameter distributions. When simulating such algorithms with a tool such as `ns-2`, these stochastic elements are never guaranteed to be accurately simulated. This effect, i.e., the variance of the stochastic variables in use, can be decreased by carrying out more simulations, and having the final simulation outcome to be the mean of the results of the individual simulations. This, however, causes an obvious increase in simulation time.

Using an analytic model also has a drawback when compared to computer simulation. Usually, it takes a significant amount of time to develop such an analytic model, and to verify its correctness. Implementing an algorithm in `ns-2` for simulation might be done faster, but as has been said before, the simulation results are less accurate and take longer to produce.

Several methods have been used in research to model the behavior of various AQM algorithms. In the rest of this chapter, the focus will be on three different modeling approaches, being the concept of stochastic differential equations, control theory (which has already been introduced in section 3.6), and the concept of stochastic Petri nets (SPNs). Each of these approaches has been chosen because research has shown that it is suitable for modeling TCP/AQM behavior in a certain way. For each approach, the way this has been done will be presented.

[1] From http://era.anthropology.ac.uk/Era_Resources/Era/Simulate/simwhat.html
[2] From http://www.wordiq.com/dictionary/analysis.html

4.2 Stochastic differential equations

The concept of stochastic differential equations (SDEs) has been applied to model characteristics of TCP/AQM. SDEs are on the one hand very similar to ordinary (partial) differential equations, which are widely used to describe a wide range of (physical) processes. On the other hand there is a big difference, since lots of processes can only be described in terms of some sort of probability. SDEs can be used when dealing with these random processes. The general form of an SDE is basically an ordinary differential equation, with a random component η added to it:

$$\frac{d(x)}{d(t)} = a(x,t) + b(x,t)\eta(t) \tag{4.1}$$

A more specific variant of SDEs, called Poisson counter driven SDEs (PCSDEs), has successfully been applied to model TCP/AQM behavior. It provides for relatively simple and fast analysis, especially when compared to algorithm simulation with `ns-2`. Also, this modeling approach can be easily applied to systems where a large number of flows are present. One drawback of the PCSDE approach is the fact that a lot of assumptions have to be made to keep the model relatively simple, which could eventually result in model outcomes that are quite different from reality. The models presented here, though, produced good results, even with the assumptions that have been made.

4.2.1 Modeling TCP behavior using PCSDEs

SDEs can be used to model TCP behavior under the assumption that the TCP window-size behavior is modeled as a fluid. This means that increments and decrements of the TCP window size are regarded as being continuous instead of discrete. Fluid modeling of TCP congestion window behavior has been presented in [20]. Another assumption to be made when modeling TCP using SDEs concerns the loss process. Usually, this loss process is seen from a source centric point of view, i.e., it is assumed that a source sends out TCP packets to the network that experience some loss probability p. In order to allow for SDE modeling, a different view on modeling packet loss is employed, referred to as a *network centric view*. This view assumes that the network is a source of congestion, which TCP flows try to detect using losses. Loss indications arrive at the source from the network in the form of duplicate ACKs or gaps in sequence numbers. Time outs occur when no ACKS are generated and sent to the sources for a given amount of time.

In [20], it is stated that when the loss arrival process is assumed to be a Poisson process, which is reasonable when the number of hops (with losses) increases, according to Khinchine's theorem [21], PCSDEs can be applied to model TCP window-size behavior. PCSDEs differ from the generic SDEs as given in (4.1) in the fact that $\eta(t)$ is replaced by a Poisson counter N_λ, where the underlying Poisson process has an arrival rate λ. N_λ is defined as

$$\begin{aligned} dN_\lambda &= \begin{cases} 1, & \text{at a Poisson arrival,} \\ 0, & \text{otherwise,} \end{cases} \\ E[dN_\lambda] &= \lambda dt. \end{aligned}$$

To obtain the general form of a PCSDE, (4.1) can now be rewritten as

$$\frac{dx}{dt} = a(x,t) + b(x,t)N_\lambda(t) \tag{4.2}$$

A model of TCP window-size behavior, constructed using these PCSDEs is presented in [20]. A few interesting results will be summarized hereafter, in order to illustrate the possibilities of this modeling approach. The TCP window-size evolution as it has been presented in Section 2.2 can be expressed by PCSDEs using the following equation, which expresses the change in window size dW:

$$dW = \frac{dt}{\mathrm{RTT}} + \frac{-W}{2}dN_{TD} + (1-W)dN_{TO} \tag{4.3}$$

The first term on the right-hand side expresses the additive increase part of TCP, the second term expresses the multiplicative decrease, where N_{TD} is a PCSDE expressing the arrival process of triple-duplicate (TD) ACKs. At the event of a TD-type loss, the congestion window size W is halved. Finally, the third term expresses the timeout behavior, where N_{TO} is a PCSDE that expresses the arrival process of timeout signals. When a TO-type loss event is detected, W is set back to 1 (i.e., to 1 $\times$ MSS, see Section 2.2). Note that the slow-start phase behavior is left out of consideration here, since a typical slow-start phase only lasts for a few RTTs, and thus its influence will be relatively small. Moreover, in TCP Reno, the slow-start phase only occurs after TO losses (see Section 2.2 for details on this), which decreases its influence even more. A PCSDE model of the TCP window size, where the slow-start phase is included, can be found in [20].

Another measure of reasonable interest is the maximum throughput R of a TCP connection. To obtain an expression for R, first, the expected value $E[W]$ of the window size W is calculated by taking expectations at both sides of (4.3), resulting in

$$E[dW] = E\left[\frac{dt}{\mathrm{RTT}}\right] + \frac{-E[W]}{2}E[dN_{TD}] + (1-E[W])E[dN_{TO}] \tag{4.4}$$

Solving this for $E[W]$ and taking the steady-state result (i.e., the result for $t \to \infty$) gives

$$E[W] = \frac{\frac{1}{\mathrm{RTT}} + \lambda_{TO}}{\frac{\lambda_{TD}}{2} + \lambda_{TO}} \tag{4.5}$$

From this, an expression for the maximum throughput R can be derived simply by dividing $E[W]$ by RTT:

$$R = \frac{1}{\mathrm{RTT}}\left(\frac{\frac{1}{\mathrm{RTT}} + \lambda_{TO}}{\frac{\lambda_{TD}}{2} + \lambda_{TO}}\right) \tag{4.6}$$

An aspect that has not been taken into consideration here is a possible limitation on the maximum window size. This is, in fact, an important parameter, which will alter the model presented above significantly. In order to introduce a maximum window size M in the model, an indicator function $I_M(W)$ is added to (4.3) as follows:

$$dW = I_M(W)\frac{dt}{\mathrm{RTT}} + \frac{-W}{2}dN_{TD} + (1-W)dN_{TO}, \tag{4.7}$$

where

$$I_M(W) = \begin{cases} 1, & \text{when } W < M \\ 0, & \text{when } W = M \end{cases}$$

The indicator function $I_M(W)$ simply disables the additive increase of the window size W when it has reached its maximum M. (4.7) can be used to derive new expressions for $E[W]$ and R, which can be found in [20]. The derivation of these equations is similar to the derivation of those belonging to the unlimited window-size model presented in this section.

4.2.2 Including RED behavior in the PCSDE model

The model presented in the previous section has served as basis for a PCSDE model of RED behavior, which has been presented in [22]. This model of RED is more detailed than the TCP model presented before, since the delay τ between the actual packet loss and the detection of the loss by a flow has been included in this model. Therefore, the time parameter t has also been included explicitly in this model, where it was only present implicitly in the TCP model in the previous section. To add even more detail, the RTT is not assumed to be constant anymore, as opposed to the previous model. The RTT $R_i(t)$ of a certain flow i at time point t is now given by

$$R_i(t) = \alpha_i + q(t)/C,$$

where α_i is a fixed propagation delay, $q(t)$ is the instantaneous queue length at time t, and C is the link capacity. Furthermore, the delay τ has been modeled as the solution to the following equations:

$$\begin{aligned} t &= R(q(t')) + t' \\ t' &= t - \tau \end{aligned}$$

Thus, τ represents approximately one round trip delay, which is the time it takes a loss notification to reach the source after packet loss has occurred at the router queue. A drawback of the model presented in the previous section is that the packet loss process was independent of the actual data flow. This drawback has been dealt with in [22] by modeling a complete system, where packet losses and packet sending rates are closely coupled. The model presented applies to a system with multiple flows, and adopts the following expression for the evolution of the window size W for a given flow i:

$$dW_i(t) = \frac{dt}{R_i(q(t))} + \frac{-W_i(t)}{2} dN_i(t) \tag{4.8}$$

Note that, as opposed to the model from the previous section, all packet losses are assumed to be of the TD type, which implies that the third term of (4.3) can simply be omitted, as is shown above. Taking expectations at both sides yields

$$dE[W_i(t)] \approx E\left[\frac{dt}{R_i(q(t))}\right] + \frac{-E[W_i(t)]E[dN_i(t)]}{2} dt \tag{4.9}$$

Note that this is an approximation, since independence between the flow window size $E[W_i(t)]$ and the loss arrival process $E[dN_i(t)]$ is assumed, which is generally not the case. This,

however, does not have a major influence on the model outcomes. As has been specified in Section 3.3.1, RED employs a proportional marking scheme, which means that the loss probability of a certain flow depends on the bandwidth share of that flow. This implies that when the throughput of a given flow i is $B_i(t-\tau)$, the rate of loss indications at time t $E[dN_i(t)]$ is equal to $p_i(t-\tau)B_i(t-\tau)$, or $p_i(t-\tau)(W_i(t-\tau)/R_i(q(t-\tau)))$. When this is applied to (4.9), the result is

$$\begin{aligned} dE[W_i(t)] &\approx E\left[\frac{dt}{R_i(q(t))}\right] + \frac{-E[W_i(t)]}{2} p_i(t-\tau)\frac{E[W_i(t-\tau)]}{R_i(q(t-\tau))}dt \\ &= E\left[\frac{dt}{R_i(q(t))}\right] + p_i(t-\tau)\left(\frac{-E[W_i(t)]E[W_i(t-\tau)]}{2R_i(q(t-\tau))}\right)dt. \end{aligned} \tag{4.10}$$

The evolution of the expected congestion window size over time can now be expressed as

$$\frac{dE[W_i(t)]}{dt} = \frac{1}{E[R_i(q(t))]} + p_i(t-\tau)\left(\frac{-E[W_i(t)]E[W_i(t-\tau)]}{2R_i(q(t-\tau))}\right)dt. \tag{4.11}$$

Finally, the router queue length evolution over time can be specified as

$$\frac{dq(t)}{dt} = \sum_{i=1}^{M(t)} \frac{E[W_i(t)]}{E[R_i(q(t))]} - C, \tag{4.12}$$

where $M(t)$ is the number of active connections at time t, and C is again the link capacity. Note that the model presented here applies for a single congested router. It can be extended to a network in a relatively straightforward manner, which is specified in [22]. For now, the above is assumed to be sufficient to give an example of the modeling power of PCSDEs with regards to TCP and AQM.

4.3 Control theory

Control theory can be applied when one wants to control some sort of physical quantity in a system. This concept has been applied in a wide range of research areas, from computer science to chemistry. With regard to TCP/AQM, several algorithms have been developed that have a control theoretic structure, of which the most important ones have already been mentioned in Section 3.6. Control theory can also be used in a slightly broader sense, allowing for a general model of TCP/AQM behavior. Such a model will be summarized in this section.

In the previous section, a TCP/AQM model constructed using PCSDEs has been summarized. A very important by-product of PCSDE models is that they can be mapped onto control theoretic models more or less directly. For the RED model presented in the previous section, this has been done in [24]. The control theoretic model presented in that paper will be summarized below, so as to give an example of general TCP/AQM modeling using control theory. The first step towards such a control theoretic model is the linearizing of the PCSDE model that has been presented in the previous section, because a linear model can be mapped to a control theoretic model in a straightforward manner. The following expression for TCP window size dynamics is derived from (4.11):

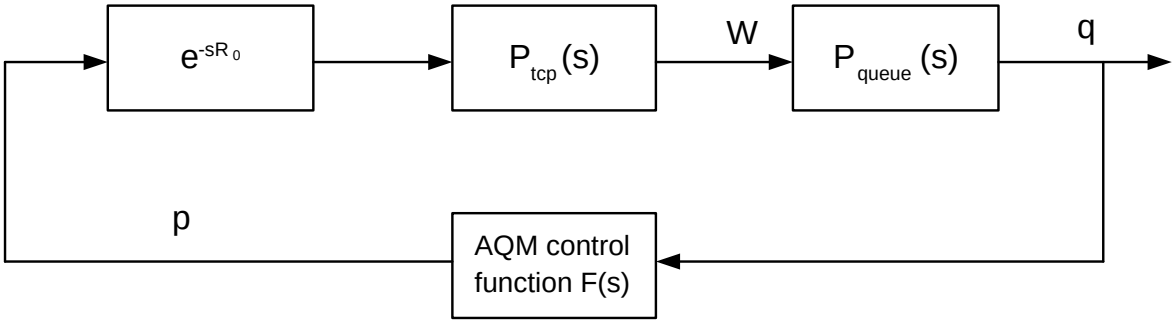

Figure 4.1: AQM as a feedback control system

$$\delta\dot{W}(t) = -\frac{2M}{R_0^2 C}\delta W(t) - \frac{C^2 R_0}{2M^2}\delta p(t - R_0), \tag{4.13}$$

where M is the (constant) number of active TCP sessions, $\dot{W}$ denotes the first-order differential of W, and δ denotes a (small) difference between the current value of a variable and its target value, e.g., $\delta W = W - W_0$, where W_0 is the target value of the window size W. R_0 is the RTT, which is assumed to be constant in this model. This implies that the delay parameter τ from (4.11) can be substituted by R_0, as has been done above. Note that (4.13) does not apply to any particular flow anymore, but rather is a general expression for the TCP window size. Linearizing the dynamics of the queue length $q(t)$, which has been specified in (4.12) results in

$$\delta\dot{q}(t) = \frac{M}{R_0}\delta W(t) - \frac{1}{R_0}\delta q(t). \tag{4.14}$$

The linearization process is described in further detail in [24]. From the linearized system specified in (4.13) and (4.14), an AQM control system can be derived directly. This AQM system can be modeled as the block diagram seen in Figure 4.1. In this figure, P_{tcp} denotes the transfer function from loss probability δp to window size δW, which has been specified in (4.13), and P_{queue} relates δW to the queue length δq as it has been specified in (4.14). The term e^{-sR_0} is the Laplace transform of the time delay, as it is included in the loss term $\delta p(t - R_0)$. The AQM control block is usually referred to as the *controller*, whereas the rest of the (uncontrolled) system is referred to as the *plant*. The overall plant transfer function, $P(s) = P_{tcp}(s)P_{queue}(s)$, can be expressed in terms of network parameters, which yields

$$\begin{aligned} P_{tcp}(s) &= \frac{\frac{R_0 C^2}{2M^2}}{s + \frac{2M}{R_0^2 C}} \\ P_{queue}(s) &= \frac{\frac{M}{R_0}}{s + \frac{1}{R_0}} \end{aligned} \tag{4.15}$$

Again, $P_{tcp}(s)$ represents the transfer function from the loss probability δp to the window size δW, whereas $P_{queue}(s)$ relates δW to the queue length δq. Next, the AQM control strategy, which is still to be added to the model, can be represented by the transfer function $F(s)$ as seen in Figure 4.1. For RED, a model for this transfer function is given by

$$F(s) = F_{RED}(s) = \frac{L_{RED}}{s/K + 1}, \tag{4.16}$$

where

$$L_{RED} = \frac{p_{max}}{max_{th} - min_{th}}; \qquad K = \frac{\ln(1 - w_q)}{\Delta}$$

Here, w_q is the low-pass filter weight (see Section 3.3.1), and Δ is the queue length sampling time. The addition of this AQM transfer function completes the control theoretic model of a TCP/AQM system.

4.4 Stochastic Petri Nets

The concept of Petri Nets (PNs) has been introduced by Karl Petri in 1964, as a graphical modeling formalism allowing for the convenient description of dynamic system behavior. Around 1980, the concepts of time and stochastics have been added, resulting in a formalism widely known as stochastic Petri nets (SPNs). The static structure of SPNs is defined by the union of a set of places and a set of transitions, together with the union of a set of input arcs (pointing from places to transitions) and a set of output arcs (pointing from transitions to places). Places represent system state variables, whereas transitions resemble actions that trigger changes of states. Finally, each place may have one or more tokens. A particular distribution of tokens over the set of places of an SPN is called a *marking*. As a result, the state an SPN is residing in at a particular moment in time is defined by the marking of the SPN at that moment. The dynamic structure of an SPN is defined by the possible movements of tokens from one place to another. These movements are described by a set of *firing rules*, which are applied to the transitions. For example, one of these rules states that a transition is only enabled when all of its input places contain at least one token. A more detailed description of the static and dynamic structure of SPNs can be found in [25].

Most classes of SPNs can be mapped automatically onto an underlying Continuous Time Markov Chain (CTMC) with a finite state space (given that state residence times are exponentially distributed). Such a CTMC can be analyzed numerically, thereby giving information about various stationary and transient performance measures of the SPN under consideration. These performance measures are formulated in terms of (the expected value of) reward functions on this SPN. For the mapping onto and analysis of CTMCs, various tools are available, of which the Stochastic Petri Net Package (SPNP) [27] will be used later on in this report. SPNP is a tool which enables the user to define rewards for an SPN, which can be used to derive a variety of performance measures for that SPN. For the specification of the SPN and its rewards, SPNP uses CSPL, an extension of the C programming language specifically developed for the specification of SPNs.

One of the aspects of SPN modeling that makes its use appealing, is the fact that modeling TCP/AQM characteristics using SPNs is relatively user-friendly. Instead of programming such a model, or constructing it using purely mathematical expressions or formalisms as is done in the control theory and SDE modeling approaches, an SPN model can be constructed using a predefined set of behavior rules. The graphical nature of this formalism adds to an easy understanding of the model by both the model designer and (future) users of that model.

When modeling systems that involve relatively large queues or buffers, which is typically the case in a TCP/AQM network, a special class of SPNs known as *infinite-state SPNs* (or iSPNs) is employed most of the time. The main reason for this is the fact that iSPNs allow for a distinguished place of unbounded capacity, which is usually used to model infinite buffers or to approximate large buffers. The underlying CTMC of an SPN incorporating such an unbounded place has an infinite number of states, but since this infinite state space has a special structure that makes it suitable for efficient numerical analysis, iSPNs can be successfully employed when modeling and analyzing a TCP/AQM network.

Chapter 5

An SPN-based TCP drop tail model

In this chapter, we investigate the SPN model of two TCP sources sharing a single buffer, which has been presented in [23], in more detail. The particular subject of interest in this chapter is the proportional drop tail loss model embedded within this SPN model. An attempt to enhance this loss model will be made in the next chapter, trying to add characteristics of active queue management to the SPN model. This chapter first gives a detailed overview of the SPN model as it has been presented in [23], and briefly introduces the features that are to be added to the model in order in be able to model AQM behavior.

5.1 An SPN model of TCP behavior

To illustrate the modeling possibilities of SPNs, a model of TCP behavior will be summarized here. This model has been presented in [23], and has been created to study the interaction of multiple TCP sources that share one or two bottleneck buffers. The model is based on the fluid PCSDE model of TCP and RED, which has been presented in [22] and has been summarized in Section 4.2. The SPN model presented in this section tries to work around two fundamental problems of the PCSDE model, being:

- Analysis of the PCSDE model is set up in terms of averages of connections, and does not include any knowledge of individual connections. This is an important aspect though, for example when sources switch off with a rate depending on their transmission rate.
- There is no mention of how to compute the expectations that are taken, cf. (4.4) and (4.9).

The first model presented in [23] is a model of the interaction between two TCP processes that share a buffer. This buffer uses a proportional loss model resembling drop tail-like behavior, cf. Section 2.2. When looking at the dynamic behavior of the SPN model, three phases can be identified:

1. *From initial state to congestion.* This corresponds to the additive-increase phase of the TCP window size evolution. The window size of each TCP source is incremented stepwisely, as is the buffer content due to the sources sending packets into the network up to the point where the buffer is full and starts dropping packets.

2. *The loss process.* The probability that a TCP source is selected for packet dropping is proportional to that source's transmission rate. This is modeled by a random switch between the two loss processes (there is one for each source), where each option from this random switch is assigned its corresponding proportional dropping probability.

3. *Congestion removal.* The selected TCP source's congestion window is reduced by a factor two. If the net buffer input rate is still positive, this is repeated, until this input rate becomes negative again, and the model goes back to the first 'phase'.

From this model, a number of performance measures can be derived, such as an expected transmission rate, an approximation for the throughput of each connection, and the link utilization. The way to do this is also specified in [23]. Also, a model proposition for a different loss model, being a synchronous loss model where both sources have their congestion window reduced upon a loss event, is given in this paper. As a final step, the original 'two sources, one buffer'-model has been extended to a scenario where there are multiple sources, which can exhibit on-off-behavior, and to a scenario where three sources are sharing two buffers. This has been done to show the extensibility of the model summarized above.

5.2 Structure of the SPN model

In order to be able to specify TCP window size behavior with SPNs, the source window size and buffer content processes have been made discrete. The reason for this is that SPNs can only be used to express models with a discrete state-space. The queue process from (4.12) can take any value in the interval $[0, B]$, where B is the maximum buffer size. This process can be modified into a corresponding discrete process $D(t)$ with state space $\{0, 1, \ldots, K\}$. When $D(t) = k$, $0 \leq k \leq K$, the corresponding buffer content equals kB/K. The RTT for a given source i, given that the buffer filling level is k, now becomes (similar to (4.12)):

$$T_i(k) = T_i + \frac{k}{K}\frac{B}{C}, \tag{5.1}$$

where T_i is the RTT for source i when the buffer is empty, and C is again the link capacity. The window process $W_i(t)$ of a given source i can also be modeled as a discrete process, with state space $\{0, 1, \ldots, N_i\}$, where N_i is the maximum window size of source i. When $W_i(t) = n_i$ and $D(t) = k$, source i sends data at a rate $\mathrm{MSS}_i n_i / T_i(k)$. In the rest of this report, the shorthand $r_i(k) = \mathrm{MSS}_i / T_i(k)$ will be used, which implies that the sending rate of source i is given by $n_i r_i(k)$.

The SPN model under attention is shown in Figure 5.1, and consists of two TCP sources that share a single buffer. It contains three subnets, S_1, S_2 and B, where S_1 and S_2 are the subnets for the sources, and B is the buffer subnet. These subnets are described first, before the dynamics of the complete model are discussed. Some basic knowledge of the structure and dynamics of SPNs is assumed here.

The source subnet `S1` contains three places (`win1`, `winF1`, and `loss1`), one immediate transition (`tLoss1`), and two timed transitions (`tIncr1` and `tDecr1`). The state of `S1` is given by the markings of `win1`, `winF1`, and `loss1`. The number of tokens in `win1` (#`win1`) represents the size of the congestion window for `S1`, while #`winF1` denotes how much further

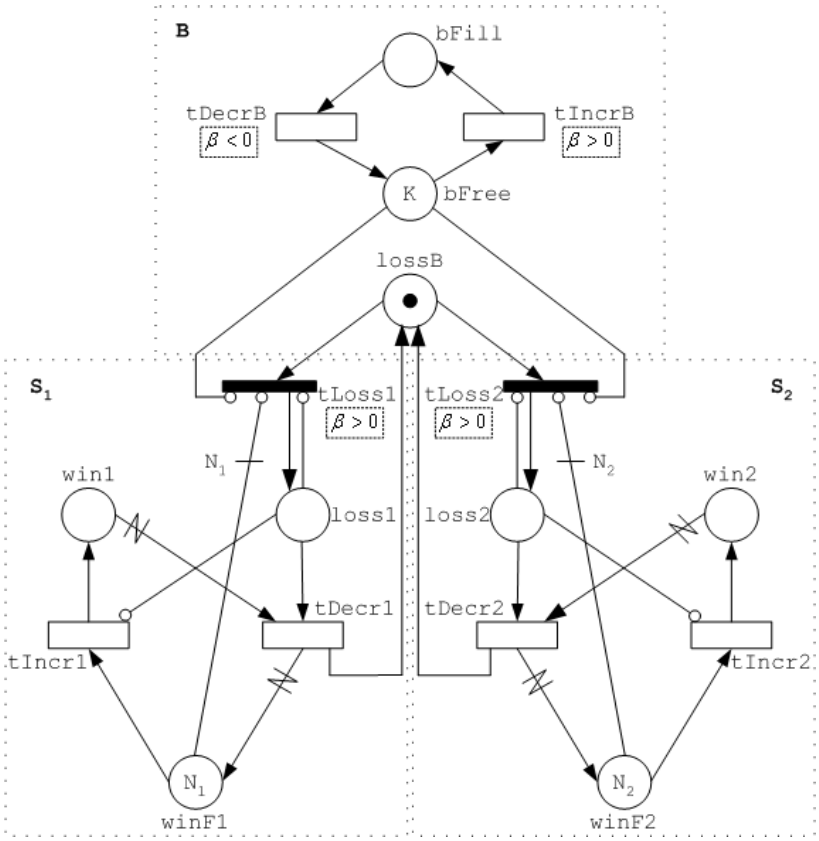

Figure 5.1: An SPN model of two TCP sources sharing a single buffer

the congestion window for source 1 can grow. For every state, #`win1` + #`winF1` should be equal to the maximum congestion window size N_1. The loss state of `S1` is represented by the `loss1` place. If #`loss1` = 0, `S1` is allowed to increase its congestion window, if #`loss1` = 1, `S1` should reduce its congestion window by a factor 2. Note that there is a one-to-one correspondence between the discrete window process $W_1(t)$ and #`win1`. The subnet for source 2 (`S2`) is specified similarly.

The buffer subnet `B` contains three places (`bFill`, `bFree` and `lossB`), and two timed transitions (`tIncrB` and `tDecrB`). The number of tokens in `bFill` represents the filling of the buffer, whereas the number of tokens in `bFree` represents how much free space there is left in the buffer. Again, for every state, #`bFill` + #`bFree` should be equal to the maximum buffer level K. The place `lossB` represents buffer congestion, with #`lossB` = 1 implying that there is no congestion. There is a one-to-one correspondence between the discrete buffer process $D(t)$ and #`bFill`.

Every transition has marking-dependent firing rates and guards associated with it, which are summarized in Table 5.1. The function $T_i(k)$ has already been specified in (5.1), and the function

$$\beta(W_1, W_2, K) = \frac{K}{B}(r_1 W_1 + r_2 W_2 - C) \tag{5.2}$$

Transition	Rate	Guard
tIncrB	$\beta(W_1, W_2, K)$	$\beta(W_1, W_2, K) > 0$
tDecrB	$-\beta(W_1, W_2, K)$	$\beta(W_1, W_2, K) < 0$
tLoss1	∞	$\beta(W_1, W_2, K) > 0$
tLoss2	∞	$\beta(W_1, W_2, K) > 0$
tIncr1	$T_1^{-1}(k)$	—
tDecr1	$T_1^{-1}(k)$	—
tIncr2	$T_2^{-1}(k)$	—
tDecr2	$T_2^{-1}(k)$	—

Table 5.1: Model transition rate functions and guards

represents the rate of in- and decrements in buffer size.

5.3 Dynamic behavior

As has already been stated in the previous section, the dynamic behavior of the SPN model of TCP/AQM can be specified in three steps. Each of these steps will be described in more detail below, so as to give a good overview of the dynamics of this SPN model.

5.3.1 From initial state to congestion

The initial marking of the SPN is shown in Figure 5.1. The buffer is empty, and in possession of the loss token, while the sources are off. Only the `tIncr1` and `tIncr2` transitions are enabled, and firing at rate $1/T_1$ and $1/T_2$, respectively. This models the additive-increase phase of the TCP model, as presented in Section 2.2. This means that a source spends (on average) $T_i(k)$ time in state i, given that $D(t) = k$.

After a number of firings of `tIncr1` and `tIncr2`, the windows W_1 and W_2 have become so large that $\beta(W_1, W_2, 0)$ becomes positive. This corresponds to the phenomenon that not all arriving traffic can be processed immediately, and as a result, data needs to be stored in the buffer before it can be processed. This will enable `tIncrB`, and each firing of this transition will increment the buffer process $D(t)$ by one. After K of these firings, $D(t) = K$, i.e., the buffer is full. This results in the disabling of the inhibitor arcs from `bFree` to `tLoss1` and `tLoss2`, so that `tLoss1` and `tLoss2` become enabled.

Now assume that `tLoss1` fires first, so that source 1 receives the loss token (i.e., the congestion signal sent to a source by the buffer). The inhibitor arc from `loss1` to `tIncr1` prevents W_1 from increasing any further. Note that here, `loss2` does not become marked, and as such, W_2 can still be increased.

5.3.2 The loss process

As has been said before, the buffer B incorporates a proportional loss model. When congestion is detected at the buffer, either one of the two sources is selected for congestion removal with a probability proportional to that source's momentary transmission rate. This mechanism has

	$r_1(K)W_1 \leq C$	$r_1(K)W_1 > C$
$r_2(K)W_2 \leq C$	$p_1 = \frac{r_1(K)W_1}{\mathbf{r}(K)\mathbf{W}}$	$p_1 = 1$
$r_2(K)W_2 > C$	$p_1 = 0$	$p_1 = \frac{r_1(K)W_1}{\mathbf{r}(K)\mathbf{W}}$

Table 5.2: Firing probability p_1 for transition `tLoss1`

been implemented by means of a random switch between the `tLoss1` and `tLoss2` transitions, where either `tLoss1` fires with probability p_1, or `tLoss2` fires with probability $p_2 = 1 - p_1$. The way in which the values for p_1 are computed, for all possible conditions on $r_i(K)$ and $W_i, i \in \{1, 2\}$, is specified in Table 5.2.
For the loss probability p_2 associated with `tLoss2`, a similar table can be constructed. Note that when $r_1(K)W_1 > C$ and $r_2(K)W_2 \leq C$, source 1 will certainly lose traffic. When both $r_1(K)W_1 \leq C$ and $r_2(K)W_2 \leq C$, both sources can lose traffic, with a probability proportional to their transmission rate. In the rare case that both $r_1(K)W_1 > C$ and $r_2(K)W_2 > C$, both sources will have to reduce their congestion window. Due to the structure of the model, this cannot happen at the same time for both sources, so this will be done in two consecutive steps.

The proportional loss model can be changed into a synchronous loss model, where both sources adapt their congestion window in the case of detected congestion. In this case, two loss tokens are needed, and the `tLoss1` and `tLoss2` transitions do not form a random switch anymore, but fire both with probability 1 when they are enabled (i.e., when congestion has been detected). Suppose that when `bFree` becomes empty (i.e., the buffer is completely filled), `tLoss1` is the first to fire. One of the loss tokens will then be moved from `lossB` to `loss1`. The inhibitor arc from `loss1` to `tLoss1` prevents this transition from firing again. As a consequence, `tLoss2` fires immediately, resulting in both sources receiving a loss token at the same time instant. As long as `loss1` is marked, source 1 cannot receive another loss token (which would become available after a firing of `tDecr2`), due to the inhibitor arc from `loss1` to `tLoss1`. Of course, a similar scenario can be outlined for source 2.

5.3.3 Congestion removal

When, for instance, source 1 has been selected for congestion removal, the timed, variable transition `tDecr1` will be enabled. When this transition fires, the loss token is removed from place `lossB`, and placed in `loss1`. Half of the tokens from `win1` (the place that represents the congestion window for source 1) are moved to `winF1`, corresponding to the multiplicative decrease-step in the TCP congestion control mechanism. When still $\beta(W_1, W_2, K) > 0$, meaning that the net input rate is still positive, either `tLoss1` or `tLoss2` will fire again immediately. When, after a number of firings, $\beta(W_1, W_2, K)$ becomes negative, `tLoss1` and `tLoss2` are disabled, meaning that the congestion windows cannot be decreased any further. The `tDecrB` transition decrements the buffer content, indicating that congestion has been removed successfully.

5.4 Loss process analysis and AQM extension

An interesting extension to the SPN model presented here is the inclusion of AQM behavior into it. There are two key differences between the loss models associated with AQM algorithms in general and the drop tail loss model included in the current SPN model. These key differences are:

1. Whereas drop tail does not start dropping packets until a buffer has been filled completely, AQM algorithms allow a buffer to drop packets before this point, thereby attempting to reduce buffering delay.

2. The percentage of packets that are dropped in drop tail is either 100 or 0, depending on whether the buffer is completely filled or not, respectively. With AQM algorithms, this percentage can also take intermediate values, since a dropping probability will be associated to each incoming packet. This probability can be derived from a range of different parameters, such as average queue length, link utilization, or the class a packet belongs to (as has been discussed in Chapter 3).

Another requirement (if possible) for the new model is to keep it as generic as possible, in order to be able to use it for the modeling of a range of different AQM algorithms, instead of being specifically designed for a single algorithm. The next chapter proposes an extended SPN model, which attempts to be as generic as possible. As a working example, it will be fine-tuned for the modeling of RED, but comments on the modeling of different AQM algorithms will also be given.

Chapter 6

An SPN-based TCP model with AQM

In this chapter, a proposition for an extended SPN model, suitable for modeling the behavior of AQM algorithms, will be presented. This model is a relatively straightforward extension of the SPN model of TCP with a drop tail loss policy, which has been presented in detail in Chapter 5. The key differences between AQM loss models and the drop tail loss model of the SPN model presented in [23], which have been identified in that chapter as well, form the motivation behind the development of this model extension. Note that the extended model has been designed with the scenario of two TCP sources sharing a single AQM buffer in mind, but that it can be extended to incorporate more sources just as straightforward as the original model.

Just as the original model, this extended SPN model has been specified in the CSPL language, which is an extension of C suitable for the specification of Stochastic Petri Nets. This CSPL specification can then be used in combination with the SPNP tool to generate results and obtain various performance measures of the SPN model. The complete specification of the extended model can be found in Appendix A.

After the generic extension of the SPN model has been presented, an example of its use will be given by the implementation of RED behavior. In the last section, the outcomes of the model, when combined with RED characteristics will be discussed, in order to see whether this model gives a suitable reflection of RED behavior, and to verify whether the chosen approach is suitable for the modeling of AQM behavior.

Some more general thoughts on the use of the new SPN model in combination with AQM algorithms other than RED will be presented in Chapter 7.

6.1 Extensions to the original SPN model

The AQM process can be seen from the buffer point of view as a decision that is to be made at every packet arrival, since for each arriving packet it should be decided whether or not that packet is to be added to the end of the buffer. This view on AQM is reflected in the extended SPN model, which can be seen in Figure 6.1. In the original model (refer to Figure 5.1 for details), the loss token is held by the `lossB` place until the buffer has filled up completely. At

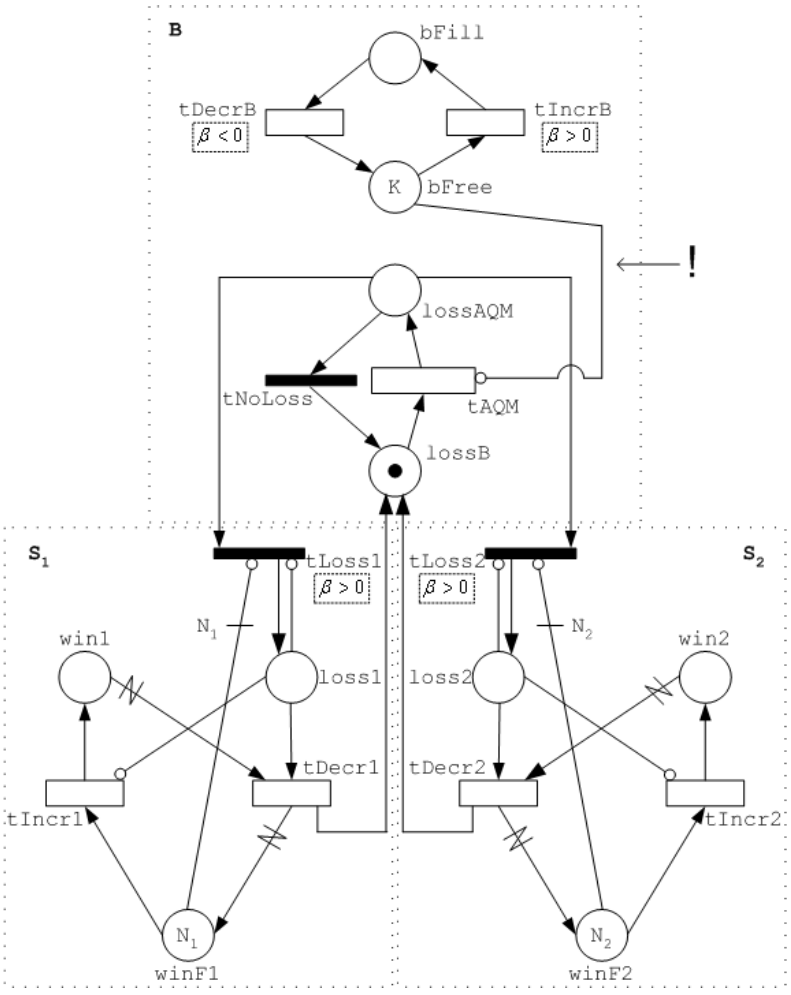

Figure 6.1: An extended SPN model, incorporating AQM behavior

that moment, the inhibitor arcs from `bFree` to `tLoss1` and `tLoss2` become inactive, allowing the loss token to be sent to either of the two sources, with a probability depending on the bandwidth fraction of the total traffic inflow that a source is responsible for. In the new model, this approach has been changed, since the loss token is now transferred from `lossB` to `lossAQM` via the timed transition `tAQM`. The rate R at which this timed transition will fire is given by

$$R = \frac{\#(\texttt{win1})}{T_1(k)} + \frac{\#(\texttt{win2})}{T_2(k)}, \tag{6.1}$$

where $T_1(k)$ and $T_2(k)$ are, as before, the RTTs for source 1 and 2 when the buffer filling level is k, respectively. Thus, R models the arrival rate of packets at the buffer, which is well-known to depend on both the RTT for each source and the window size of each source. Therefore, the packet drop-decision, which has been mentioned earlier, will be made at the modeled packet arrival rate, implying that this decision is made on average once for every modeled packet arrival.

When the loss token has arrived at `lossAQM`, it can be either forwarded via `tLoss1` to `loss1`, or via `tLoss2` to `loss2`, representing the sending of a congestion signal to source 1 or 2, respectively, just as in the original model. In addition to this, the loss token can also be sent back to `lossB` in the new model, modeling the event that no packet loss takes place at the event of a packet arrival. When the loss token is forwarded to either one of the source subnets, the congestion window of the source receiving the loss token is cut in half, which reflects TCP window size behavior, just as in the original model. This process repeats itself at the modeled packet arrival rate R. In addition to this, the function of the inhibitor arcs from `tLoss1` and `tLoss2` to `lossB` (see the exclamation marks in Figure 6.1) changes as well, since `tLoss1` and `tLoss2` will also be enabled when the buffer is not yet completely filled. The exact conditions under which `tLoss1` and `tLoss2` should be allowed to fire depend on the AQM algorithm under study. An example of the changed function of these inhibitor arcs will be given in the next section, when RED is implemented with this model.

Note that the decision made at `lossAQM` whether to forward the loss token to either one of the sources, or back to `lossB`, forms the implementation of the first of the two key differences between AQM and the drop tail loss model, which have been mentioned at the end of the previous chapter.

The implementation of the second key difference is carried out by correctly (i.e., according to the specification of the AQM algorithm under study) calculating the probabilities associated with each of the arcs going out of `lossAQM`, and then associating them with these arcs. As has been said before, the principle of AQM is that a dropping probability, which can take any value between 0 and 1, is associated with every arriving packet. With some probability p_1, the packet is added to the end of the buffer queue, and with probability $p_2 = 1 - p_1$ the packet is dropped. The differences between the various AQM algorithms are mainly found in the way these probabilities are calculated, and on which system and network parameters these probabilities depend. In the TCP/AQM model, the probability p_1 is associated with the event of the loss token being moved from `lossAQM` back to `lossB` via the immediate transition `tNoLoss`, representing a "no loss"-event. With probability p_2, the loss token is sent to either source 1 or source 2. The probability that either one of these sources is selected for loss is calculated in the same way as in the original model, and depends on the fraction of the total incoming traffic that a source is responsible for (see also Table 5.2). This fraction is assumed to be a good estimate of the fraction of the total number of packets in a queue that a given source is responsible for, and therefore this is a justified attempt to ensure fairness; algorithms contributing more to buffer content are more likely to suffer from loss. To obtain the probability that either source receives the loss token whenever a packet arrives, this fraction is multiplied by the overall probability p_2.

Note that in this model, no actual packets are modeled, but the traffic is rather modeled as a fluid. Since it is impossible to identify individual packets, nor is it possible to identify the source of a packet or flow, there is no distinguishing between separate flows, making the application of this model to algorithms that use different dropping strategies for different (types of) flows impossible. Still, this model can be applied to a significant part of the AQM algorithms presented in Chapter 3. As a first example of the application of the new model, the RED algorithm will be implemented using the generic SPN model of AQM presented in this chapter.

6.2 An application to RED

Embedding the functionality of a specific AQM algorithm into the generic SPN model presented in the previous section (referred to as 'the TCP/AQM model' in the rest of this section) is essentially a three-step process:

1. Check whether the algorithm is suitable for implementation within the TCP/AQM model;
2. Design a piece of code implementing the calculation of the dropping probability conforming to the specification of the algorithm under study;
3. Check whether additional changes have to be made to either the TCP/AQM model or the underlying calculations.

In this section, each of these three steps will be carried out for the RED algorithm, in order to provide a working example of the use of the TCP/AQM model. Although at first sight, it seems that the second step is the most significant step out of the three, the first and the last step are equally important for the correct embedding of any AQM algorithm within this model.

Step 1. The first step is to check the suitability of RED for implementation within the TCP/AQM model. In order for an algorithm to fit into the model, it has to meet certain requirements. The most important of these requirements is that the algorithm should not distinguish between different flows or different packets when determining a dropping probability, since the incoming traffic has been modeled as a fluid in the TCP/AQM model. In case of RED, the algorithm does not distinguish between different types of flows or packets, but rather assigns a dropping probability to each incoming packet regardless of where it comes from, making RED fitting this requirement. Another requirement that an algorithm has to fulfill is that the parameter values that are used in the dropping probability calculation can be derived from the state of the model, with the exception of fixed user-defined parameters. These fixed parameters can be given as an input before the model is evaluated. In the case of RED, the min_{th}, max_{th} and p_{max} parameters (refer to Section 3.3.1) are fixed, and the q_{avg} parameter, representing the average queue length, can be derived directly from the model, since it is represented by the marking of the `bFill` place, denoted as #`bFill`. This marking might seem to represent the instantaneous buffer queue length rather than the average queue length at first sight, but when we take into consideration that the traffic is modeled as a fluid, it can also be argued that #`bFill` may be used as a suitable estimate for the average queue length q_{avg}.

Step 2. Now that we have shown that RED conforms to both requirements set above, the second step in embedding the algorithm into the TCP/AQM model can be carried out. This step comprises the design of a piece of C code calculating the dropping probability, and the embedding of this piece of code into the model source. When doing this for RED, it is useful to note that the RED loss model (see also Figure 3.3) essentially consists of three different modes with regards to the buffer filling level #`bFill`, which can be summarized as:

- Mode 1: #`bFill` $\leq min_{th}$, where the dropping probability $p = 0$,

- Mode 2: $min_{th} < \#\texttt{bFill} \leq max_{th}$, where p increases linearly with #bFill up to the maximum dropping probability p_{max}, and
- Mode 3: $\#\texttt{bFill} > max_{th}$, where $p = 1$.

```
double REDLossProbability() {
    if(mark("bFill") > MAXTH){
        /* mode 3, p = 1 */
        return 1.0-1E-16;
    }
    else{
        /* mode 2, p grows linearly up to p_max */
        double p;
        p = MAX_RED*(mark("bFill")-MINTH)/(MAXTH-MINTH);
        return p;
    }
}
```

Figure 6.2: Calculation of the RED dropping probability in CSPL

The latter two modes can easily be identified in the piece of code that executes the calculation of p for RED, as given in Figure 6.2. The first mode is implemented indirectly by the multiple inhibitor arc that prevents `tAQM`, and, hence, the `tLoss1` and `tLoss2` transitions from firing whenever the buffer filling level is below min_{th}. This is explained in more detail when the third and final step is described. The `REDLossProbability()` method calculating the RED dropping probability is called whenever the `lossAQM` place receives the loss token. The resulting dropping probability p is then, for each source, combined with the fraction of the total traffic that source is responsible for. This yields the final dropping probabilities for each combination of loss model modes and shares of incoming traffic per source. For clarity, these dropping probabilities are also summarized in Tables 6.1 and 6.2, where p_{red} is the calculated RED dropping probability, and p_1 (p_2) is the loss probability for source 1 (2), which are the same as the probabilities for the original model presented in Chapter 5.

	mode 1	**mode 2**	**mode 3**
no loss	1	0	0
loss for source 1	0	see Table 6.2	$\frac{r_1(K)W_1}{\mathbf{r}(K)\mathbf{W}}$
loss for source 2	0	see Table 6.2 [1]	$\frac{r_2(K)W_2}{\mathbf{r}(K)\mathbf{W}}$

Table 6.1: Dropping probabilities for the SPN model with RED

Step 3. The third and final step of the implementation of an AQM algorithm within the SPN model consists of checking whether additional modifications need to be made to the model, and the execution of these modifications when this is the case. In the case of RED, and of other algorithms as well, this includes determining the correct multiplicity of the inhibitor arc from

[1] For the correct dropping probabilities for source 2, replace all 1's by 2's in Table 6.2, and vice versa.

	$r_1(K)W_1 \leq C$	$r_1(K)W_1 > C$
$r_2(K)W_2 \leq C$	$p_1 = p_{red} \cdot \frac{r_1(K)W_1}{\mathbf{r}(K)\mathbf{W}}$	$p_1 = p_{red}$
$r_2(K)W_2 > C$	$p_1 = 0$	$p_1 = p_{red} \cdot \frac{r_1(K)W_1}{\mathbf{r}(K)\mathbf{W}}$

Table 6.2: Dropping probabilities for source 1 in mode 2 of the RED loss model

`bFree` to the `tAQM` transition. This inhibitor arc, indicated by an exclamation mark in Figure 6.1, is assigned a multiplicity of $K - min_{th}$, where K is the maximum buffer level, in the case of RED. This implies that `tAQM` cannot fire whenever there are more than $K - min_{th}$ tokens in `bFree`. This corresponds to the fact that `tAQM`, and subsequently `tNoLoss`, `tLoss1` and `tLoss2` will not fire when the buffer filling level is below min_{th}, since $\#\texttt{bFill} + \#\texttt{bFree} = K$. Thus, no loss events will be triggered whenever the SPN model is in the first mode of the RED loss model (see Figure 3.3). The advantage of having such an inhibitor arc added to the model is a significant reduction in model state space, as is shown in later on in Section 6.3.4.

Apart from the determination of the multiplicity of this inhibitor arc, no additional modifications have to be made to the SPN model in order to correctly incorporate RED behavior. This means that the model is now ready for evaluation.

6.3 Evaluation of the model using RED

In this section, the extended SPN model, with the application of RED as proposed in the previous section, will be evaluated. To do this, a number of experiments have been carried out, of which the results will be summarized here. The experiments have been executed using the CSPL implementation of the SPN model incorporating RED behavior, in combination with the SPNP tool [27]. The complete model specification in CSPL can be found in Appendix A. The performance measures which are deemed the most illustrative for the correct functioning of the model have been selected, and will be summarized and elaborated on both textually and graphically in this section. For all of the evaluations that have been done in order to obtain these results, the network setup as depicted in Figure 6.3 has been used. For an evaluation of the throughput evolution, and of the utilization of the modeled system, just as it has been done in [23], the propagation delay d_1 of the link connecting source 1 (S_1) and the router implementing RED behavior (R_1) can be varied. Also, two different values for the buffer delay d_B will be taken into consideration: a relatively low delay $d_B = 16$ ms, and a relatively high delay $d_B = 160$ ms. Note that this buffering delay is used in the determination of the actual buffer size. By multiplying d_B with the buffer state space K, the actual buffer size B used in the simulation is acquired. It will be shown that the choice of the delay d_B has a significant impact on the model outcome.

In Appendix B, the plots of the throughput ratios and of the utilization of the original SPN model of TCP, as it has been presented in [23], are shown. These plots can be used when comparing these performance measures with their equivalents in the SPN model of TCP and RED, as it has been presented in this chapter. The plots for the original model and for the new model have been created using the same network setup, which has been displayed in Figure 6.3. By doing this, the throughput ratio and utilization performance measures of both models can be easily compared by comparing the respective plots.

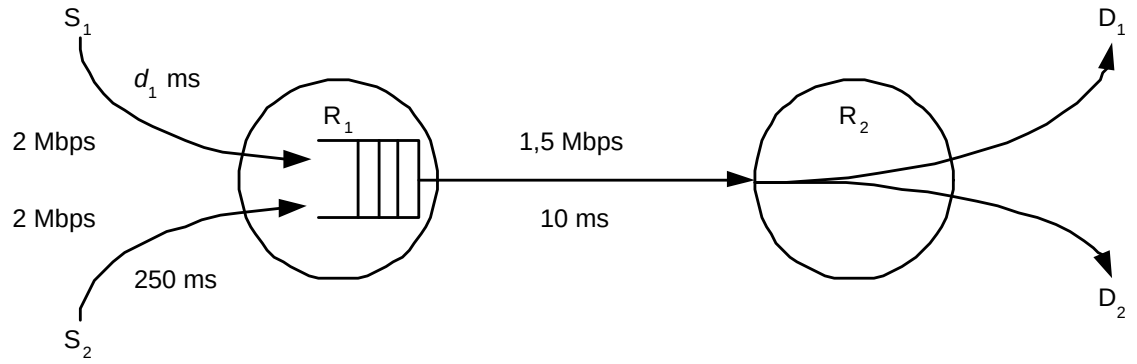

Figure 6.3: The network configuration for the simulation

6.3.1 Throughput

In [23], two possible definitions for the throughput of the SPN model have been adopted. These two alternatives, referred to as γ_i^{in} and γ_i^{out} for a given source i, differ in the fact that the former considers the traffic fluid entering the buffer, while the latter is considering the traffic fluid leaving the buffer. These two alternatives have also been used in the extended SPN model, which is the subject of this section.

For the SPN model of TCP, it has been shown in [23] that, in case of proportional loss and a relatively small buffer size, the ratio of the throughputs γ_1/γ_2 can be accurately estimated by $s^{-\alpha}$, where $s = T_1/T_2$ is the ratio of propagation delays of source 1 and 2, respectively, and $\alpha \approx 0.85$, for both $\gamma_i = \gamma_i^{in}$ and $\gamma_i = \gamma_i^{out}$. Figure B.1 in Appendix B shows these estimations. In this section, an attempt at finding a similar expression for this throughput ratio will be made for the situation in which the buffer employs a RED strategy. By plotting the throughput ratio as a function of s, i.e., as a function of the fraction of $T_1 = d_1$, the propagation delay for source 1, and the propagation delay of source 2, which is set fixed at $T_2 = 250$ ms, we empirically try to derive a good estimate for α, and try to find out whether this α is varying with the values of the various RED parameters.

First, in Figure 6.4, we see the overall throughput results as a function of p_{max}, for various values of min_{th} and max_{th}. As a reference, the throughput for the SPN model of TCP drop tail, labelled as "original model", has been plotted as well. The plot depicts the throughput situation for $d_B = 160$ ms. First of all, it can be seen in the figure that there is only a minor decrease in throughput when changing from the drop tail scenario to RED, especially for low values of p_{max}, which are commonly used in practice. We can also see from this figure that the throughput decreases when p_{max} increases for any given value of min_{th} and max_{th}. The very simple explanation for this is that when p_{max} is increased, the overall RED dropping probability p also increases, which causes more packets to be dropped actively, thereby decreasing throughput. Also, it can be concluded that the choice of min_{th} has a significant influence on the overall throughput. When min_{th} is lower, the buffer will start dropping packets earlier, because modes 2 and 3 of the RED loss model (see the previous section), where packets are getting dropped, are entered more often. This is again causing a decrease in the overall throughput. Finally, the choice of max_{th} is shown to have only a minor influence on the throughput. This can easily be concluded from Figure 6.4, where lower

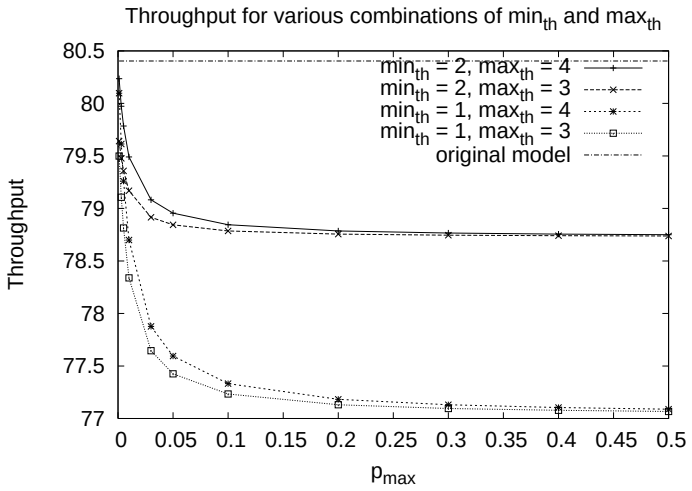

Figure 6.4: RED model throughput as a function of p_{max} for various values of min_{th} and max_{th}

values of max_{th} result in a slightly lower throughput for equal values of min_{th}. When max_{th} is set to a lower value, the model is, on average, in mode 3 of the RED loss model more often. In this mode, all packets are dropped, as opposed to mode 2, where only a fraction of the packets is being dropped. This results in a slightly lower throughput when max_{th} has a lower value.

Note that the maximum throughput, i.e., the link capacity L, is set fixed at $80.707107 = 80 + \sqrt{2}$. It can be seen from Figure 6.4 that the resulting throughput of the SPN model of TCP drop tail is very close to L. The throughput of the model using RED is slightly lower. This is due to the fact that the AQM algorithm causes the buffer to be empty relatively more often (as will be shown in Section 6.3.3), having a minor negative influence on the throughput. Also note that the overall throughput plot in Figure 6.4 only applies for the case where $\gamma_i = \gamma_i^{out}$, since the influence of the RED parameters on the throughput can be analyzed best when the traffic fluid is monitored just after this fluid has left the buffer.

Next, in Figure 6.5, we see the throughput ratios γ_1/γ_2 varying over $s = d_1/0.25$ for $p_{max} = 0.001, 0.1$ and 0.5, and for $d_B = 160$ ms. This figure can be compared to the right part of Figure 3 of [23]. Both $\gamma_i = \gamma_i^{out}$ (left plot of Figure 6.5) and $\gamma_i = \gamma_i^{in}$ (the right plot) have been taken into consideration. The min_{th} and max_{th} parameters have been set fixed at 2 and 4, respectively. Also, the fixed curve $s^{-\alpha}$ has been plotted for different values of α. The plots in Figure 6.6 correspond to the situation where the buffering delay is set to $d_B = 16$ ms, i.e., they can be compared to the left part of Figure 3 of [23]. From Figures 6.5 and 6.6, we can conclude that the throughput ratio increases with the value of p_{max} for low values of s, whereas the influence of the value of p_{max} decreases when s increases. This can be justified by considering the fact that when p_{max} is relatively high, the average buffer occupation will be relatively low (since more packets are dropped from the queue by the AQM algorithm), resulting in a decrease in queueing delay, and, hence, in an increase of the throughput. This

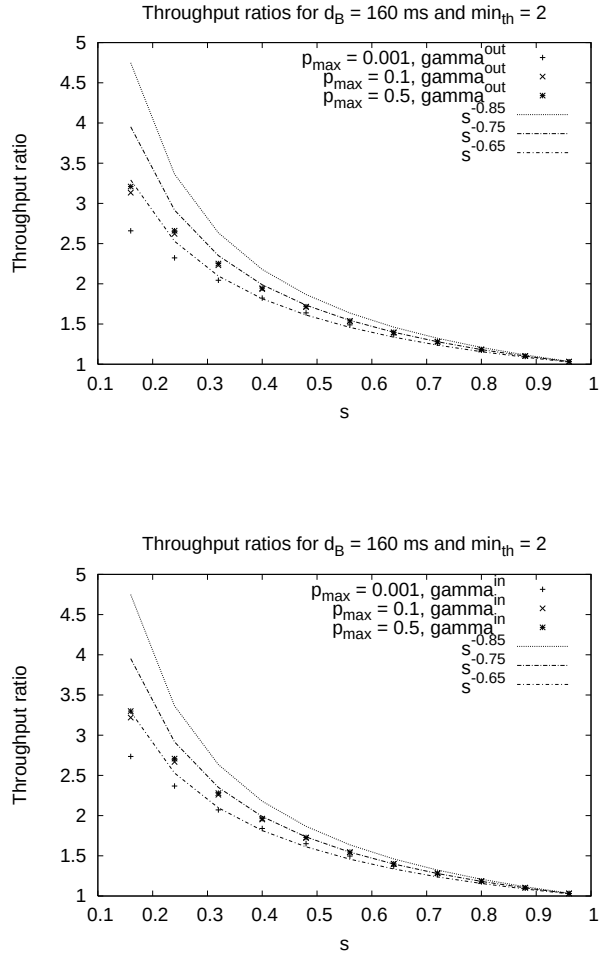

Figure 6.5: Throughput ratios as a function of s for different values of p_{max}, where $min_{th} = 2$, $max_{th} = 4$ and $d_B = 160$ ms

especially affects sources with a relatively low propagation delay, thus resulting in a higher throughput ratio $\gamma_1^{in}/\gamma_2^{in}$ or $\gamma_1^{out}/\gamma_2^{out}$. Note that the influence of p_{max} when s is relatively low decreases for higher values of this maximum dropping probability, i.e., when p_{max} would be increased beyond 0.5, the increase in the throughput ratio for low values of s would be even less.

When looking at the $s^{-\alpha}$-curves, it can be seen that in the case of $d_B = 160$ ms, i.e., in Figure 6.5, the curve where $\alpha = 0.65$ forms the best estimate of the throughput ratio curves if $\gamma_i = \gamma_i^{out}$, just as in Figure 3 of [23], although it can be seen that the accuracy of the $s^{-\alpha}$-approximation varies with the value of p_{max}. When $d_B = 16$ ms (Figure 6.6), the throughput ratios are closer to the $s^{-0.75}$ curve, where they were close to $\alpha = 0.85$ for the SPN model of TCP drop tail. An explanation for this is the fact that when the buffer delay is smaller, sources can update their window sizes faster (since the total RTT will be smaller as

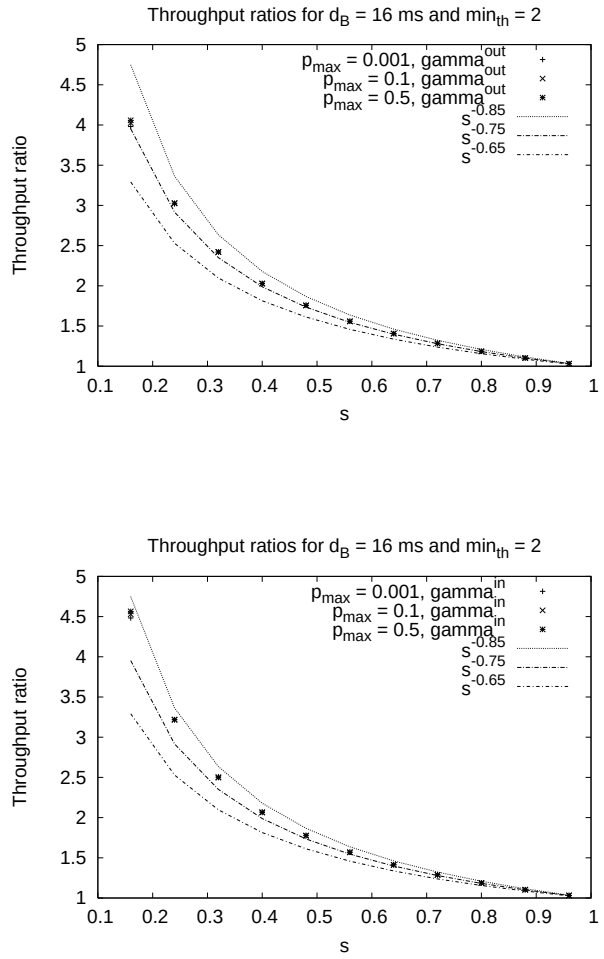

Figure 6.6: Throughput ratios as a function of s for different values of p_{max}, where $min_{th} = 2$, $max_{th} = 4$ and $d_B = 16$ ms

well), from which sources with a small propagation delay will benefit more on average. This results in a higher value for the throughput ratio, and the value for α will need to be higher accordingly, in order to form a good estimate of this ratio.

In Figures 6.7 and 6.8, the same plots are displayed as in Figure 6.5 and 6.6, respectively, but now for the situation where $min_{th} = 1$. It can be observed that the choice of a value for min_{th} has no significant effect on the accuracy of the estimation of the throughput ratio by $s^{-\alpha}$, i.e., the value of min_{th} does not significantly alter the value of α. This in turn implies that the throughput ratios remain roughly the same when min_{th} is changed (implying that the amount of fairness of RED also remains the same), while the throughput itself increases with the increase of min_{th}, when all other network parameters remain unchanged. Finally, it can be seen that increasing the buffering delay d_B results in a decrease of the throughput ratio, which means that an increase of d_B has a negative effect on the throughput advantage that

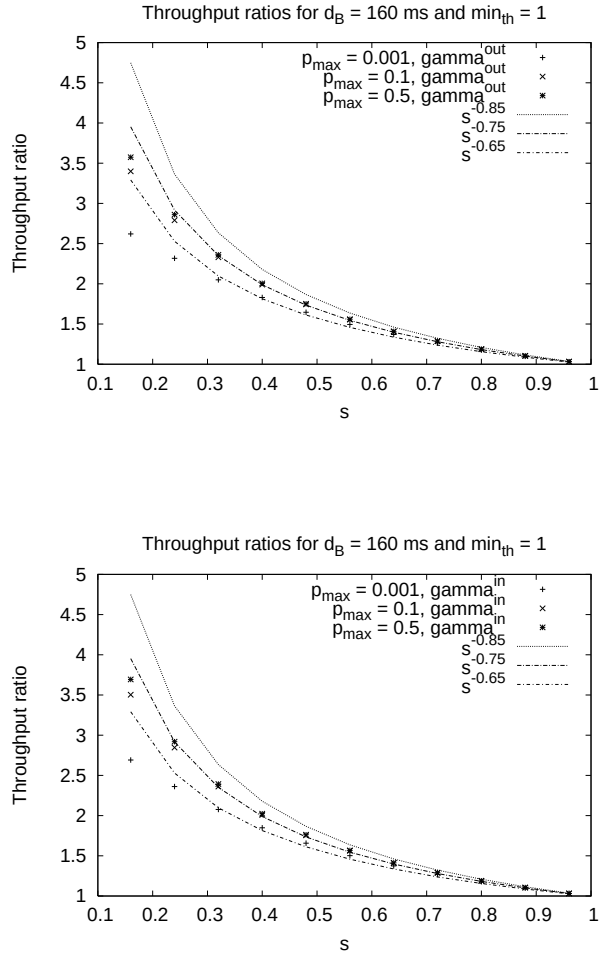

Figure 6.7: Throughput ratios as a function of s for different values of p_{max}, where $min_{th} = 1$, $max_{th} = 4$ and $d_B = 160$ ms

source 1 has due to its lower propagation delay. As a result, altering d_B requires adaptation of α, which can also be seen directly in the figures.

6.3.2 Utilization

For this SPN model of TCP/AQM, the utilization u of a link is defined as the fraction of the throughput γ and the link capacity C:

$$u_i = \frac{\gamma_i}{C}, i = 1, 2 \qquad\qquad u = \frac{\gamma_1 + \gamma_2}{C} = u_1 + u_2 \tag{6.2}$$

where γ_i is either γ_i^{in} or γ_i^{out}, depending on what proposal for the definition of the throughput has been chosen. Figure 6.9 shows the plots for various values of p_{max}, where $min_{th} = 2$, $max_{th} = 4$ and $d_B = 160$ ms. In Figure 6.10, the same plots are displayed for the situation where $d_B = 16$ ms. In Figures 6.11 and 6.12, the same situations have been plotted for

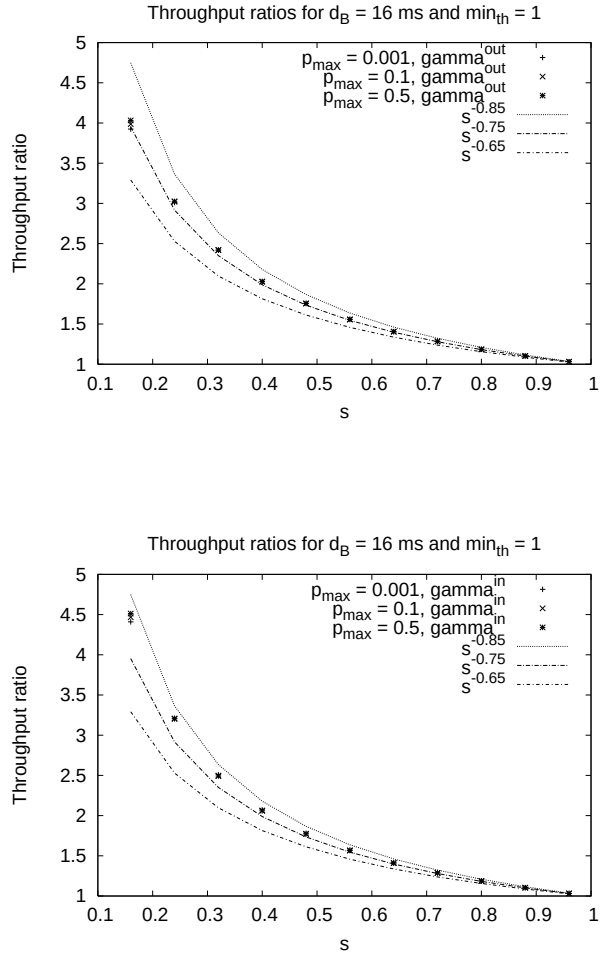

Figure 6.8: Throughput ratios as a function of s for different values of p_{max}, where $min_{th} = 1$, $max_{th} = 4$ and $d_B = 16$ ms

$min_{th} = 1$. We see from these figures that utilization increases slightly when the min_{th} threshold is set higher. The explanation for this is very simple; when the minimum threshold is set higher, the model drops less packets on average (assuming that all other parameters remain unchanged), since the first phase of the RED loss model, where the dropping probability p is 0, lasts longer. Utilization also increases when the maximum dropping probability p_{max} decreases. Here too, the explanation is simple: when p_{max} is relatively low, less packets are dropped in the second phase of the RED loss model, leading to a higher throughput (see also the previous section) and thus to utilization under the assumption that all other parameters remain unchanged. The max_{th} parameter has been left out of consideration in the discussion of the utilization measure, since this parameter has little effect on the overall throughput, as can be seen in Figure 6.4. Therefore, it will have only a minor effect on utilization as well, as the utilization is calculated directly from the throughput (see (6.2)).

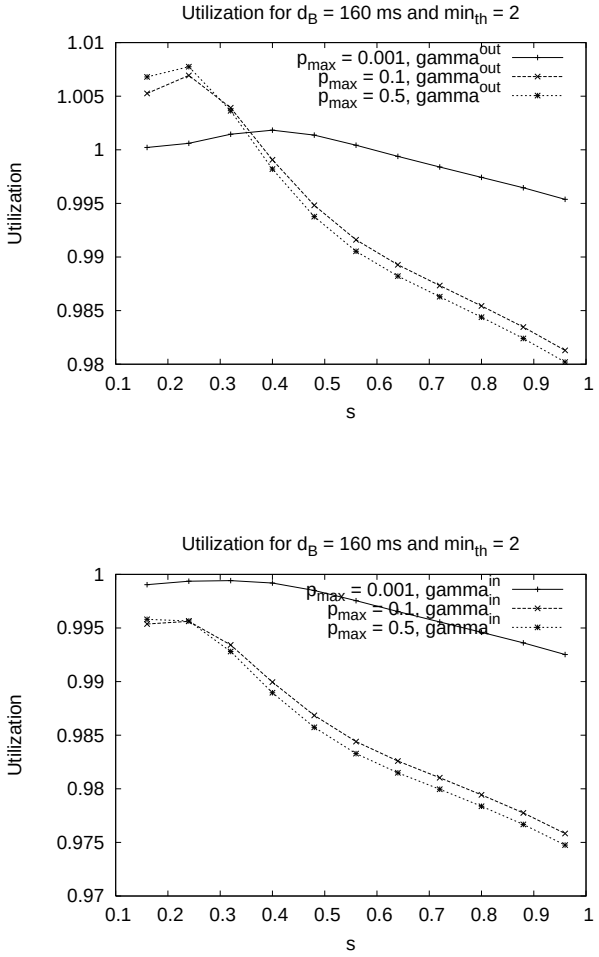

Figure 6.9: Utilization as a function of s for different values of p_{max}, where $min_{th} = 2$, $max_{th} = 4$ and $d_B = 160$ ms

Note that there is a strange peak in the utilization plot for the situation where $\gamma_i = \gamma_i^{out}$, and $d_B = 160$ ms, i.e., the upper plots of Figure 6.9 and Figure 6.11. Utilization even rises above 1 here, a phenomenon which can never occur in reality. A possible explanation for this flaw is the fact that congestion window sizes are updated quicker than the buffer filling level, resulting in a short time span in which the congestion window sizes (and especially the window of source 1) have already been doubled, but in which the buffer dynamics function, specified by (5.2), still corresponds to older and smaller congestion window sizes, leading to an inaccurate value for the link utilization. This phenomenon is most likely to occur when one of the sources (in this case, source 1) has a small propagation delay, and the buffer delay is relatively large. These conditions are satisfied in the upper plots of Figure 6.9 and Figure 6.11, where the incorrect peak occurs when d_1 is small. When the buffering delay is set to $d_B = 16$ ms, implying the buffer filling level is updated quicker in the model, the phenomenon does not occur anymore, as can be seen in the plots of Figure 6.10 and Figure 6.12.

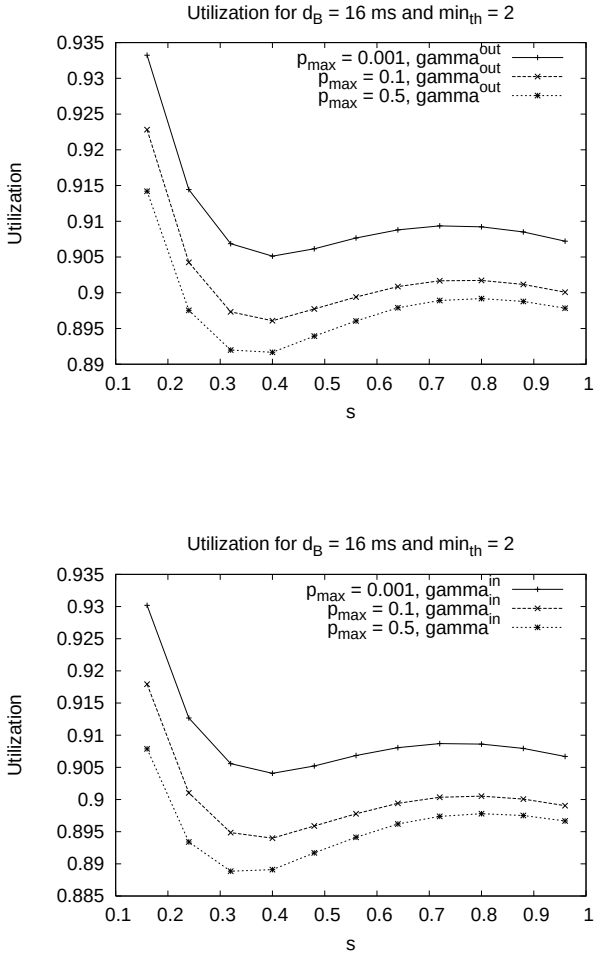

Figure 6.10: Utilization as a function of s for different values of p_{max}, where $min_{th} = 2$, $max_{th} = 4$ and $d_B = 16$ ms

One can see from Figures 6.10 and 6.12 that when $d_B = 16$ ms, there exists a minimum in utilization around $s = 0.3$. An explanation for the occurrence of this minimum has not yet been found. The same goes for the local maximum in these utilization plots around $s = 0.8$.

6.3.3 Cumulative buffer level distribution

The cumulative buffer level distribution $c_{\text{buff}}(k)$ is defined as:

$$\begin{aligned} c_{\text{buff}}(k) &= \sum_{i=1}^{k} p(\text{"buffer level} = i\text{"}), \\ c_{\text{buff}}(K) &= \sum_{i=1}^{K} p(\text{"buffer level} = i\text{"}) = 1, \end{aligned}$$

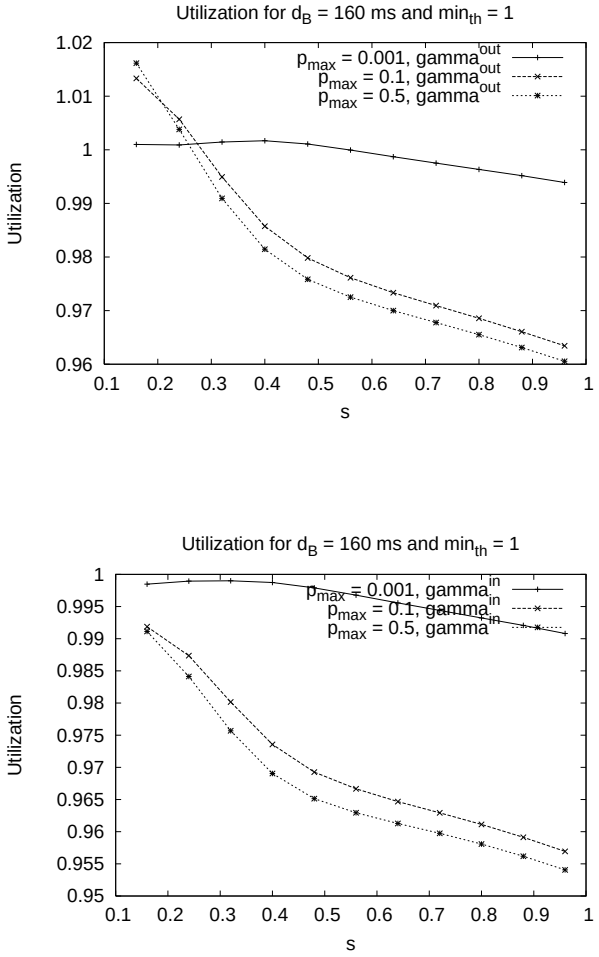

Figure 6.11: Utilization as a function of s for different values of p_{max}, where $min_{th} = 1$, $max_{th} = 4$ and $d_B = 160$ ms

where K is the maximum buffer filling level. This buffer filling level distribution is one of the most important measures when showing the benefit of the new SPN model incorporating AQM behavior (RED, in this case) over the original model, which used a drop tail buffer policy. The application of RED should result in the buffer being relatively empty compared to the scenario where drop tail has been employed, if all relevant network parameters are unchanged. To show the benefits of the application of RED, figures containing values of $c_{\text{buff}}(k)$ will be given for various RED parameter values. Also, the buffer filling level distribution for the original model will be included for an easy comparison of the old and the new model.

In Figure 6.13, the cumulative buffer filling level distribution is given for various values of min_{th}, max_{th} and p_{max}. In every plot, the cumulative distribution corresponding to the original model has been included as mentioned before. From these plots, it can be seen directly that, especially for higher values of p_{max}, the buffer filling level is generally lower compared to

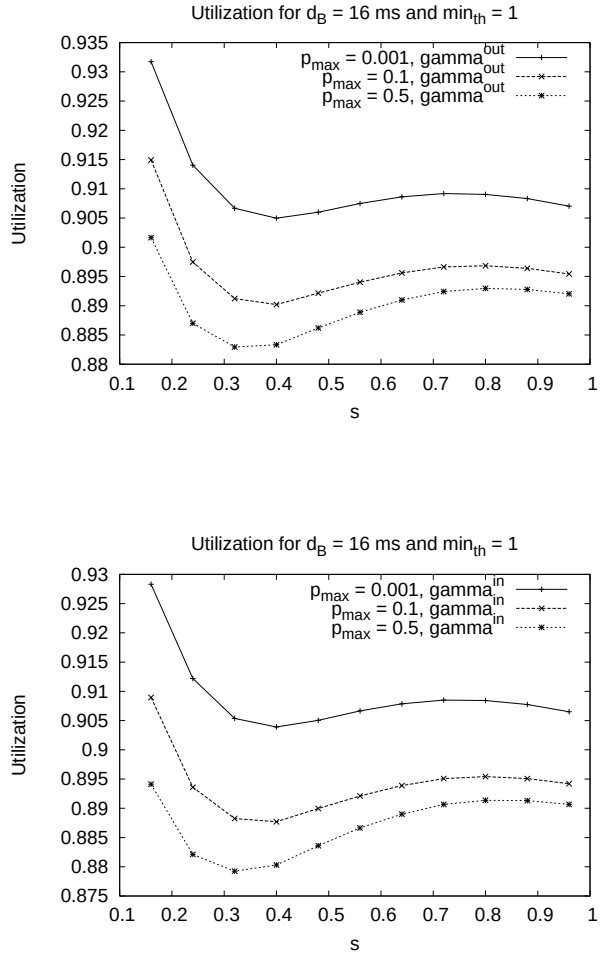

Figure 6.12: Utilization as a function of s for different values of p_{max}, where $min_{th} = 1$, $max_{th} = 4$ and $d_B = 16$ ms

the original model, i.e., the cumulative distribution approaches 1 faster, meaning that there is a higher probability of the buffer filling level being relatively low. For small values of p_{max}, $c_{\text{buff}}(k)$ for the new model is at least equal to the distribution for the original model, if not better, i.e., approaching 1 faster, indicating lower buffer filling levels on average.

As one can see in Section 6.3.1, one cannot just increase p_{max} indefinitely to reach an optimal buffer filling level distribution. The reason for this is the decline in throughput when p_{max} is increased. This gives reason to the claim that there must be some optimal value for p_{max}. It can be argued in an equal manner that there must be optimal values for min_{th} and max_{th} as well. Finding these optimal values for the RED parameters has been a research issue ever since RED was first introduced in 1993. The fact that optimal values are very hard to find (and are different for each individual network topology), let alone the fact that these parameters are set fixed in routers and therefore cannot be adapted automatically

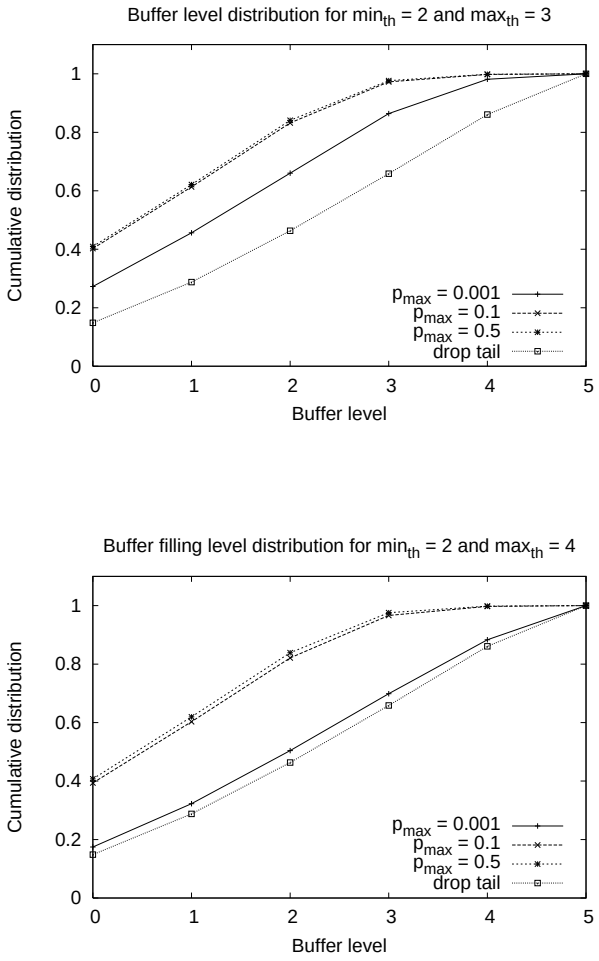

Figure 6.13: Buffer filling level distribution for various values of p_{max} and max_{th}, where $min_{th} = 2$

with changing network situations, has been identified as one of the main drawbacks of the RED algorithm. Modeling RED behavior using the SPN model presented in this report unfortunately does not present an easy solution for this problem, although the model provides an easy way for the evaluation of different RED parameter settings.

6.3.4 Model state space

Apart from the performance measures that can be derived from the model outcomes, as they have been presented in Sections 6.3.1 through 6.3.3, a performance measure of the model itself plays an important role when evaluating model effectiveness and efficiency. When one creates a model of any kind of system, one wants to assure that the model state space is as low as possible, since this will generally speed up model evaluation and system performance measure computation. Of course, reducing the model state space must not lead to a model that provides an incomplete representation of the system under attention. Regarding the SPN

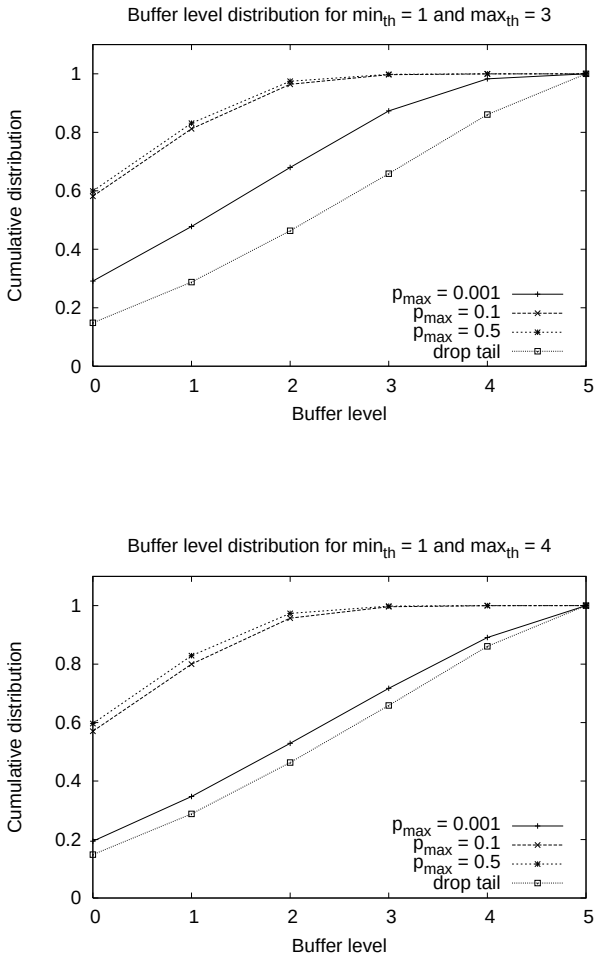

Figure 6.14: Buffer filling level distribution for various values of p_{max} and max_{th}, where $min_{th} = 1$

model of the TCP/AQM system, the model state space is represented by the state space of the Continuous Time Markov Chain (CTMC) underlying the SPN (see for a short explanation on this also Section 4.4). Every state in the CTMC corresponds to exactly one marking of the SPN. Hence, the less possible markings there are in the SPN, the less the size of the state space of the CTMC, and the less time is needed for the computation of the transient and steady-state probabilities of this CTMC (from which various system performance measures can be derived).

The size of the state space is one of the most important issues with the SPN model extension presented in this chapter. Compared to the original SPN model of TCP drop tail, incorporating AQM behavior in the SPN introduces a vast increase in state space. The reason this additional state space is introduced is the fact that the model allows loss to occur whenever the buffer is not yet completely filled. Whereas in the original model, loss was only possible when the buffer was completely full, in the new model, loss can occur whenever the

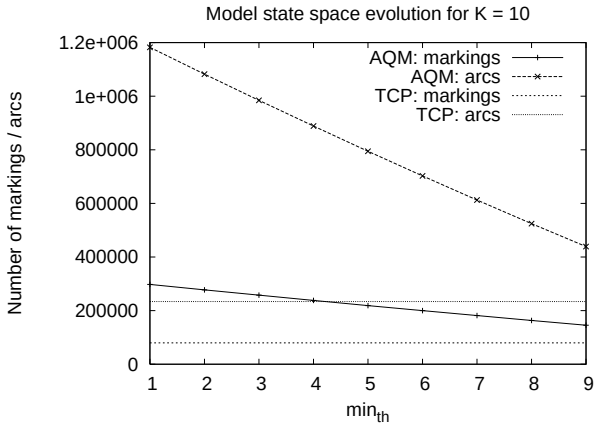

Figure 6.15: State space evolution of the SPN model of TCP/AQM with varying min_{th} for $K = 10$

buffer filling level exceeds the min_{th} threshold, which is typically set lower than the maximum buffer filling level K in order to benefit from the RED mechanism. Assume that of the CTMC of the original model, X states correspond to a situation where loss is enabled, i.e., to the situation where the buffer filling level is K. Each of these X states will occur in the CTMC for the new model, since loss is also enabled when the buffer is completely filled in the new model. It can be assumed that these X states will also be present for each buffer filling level $k < K$ where loss is also enabled, i.e., for each $k > min_{th}$. There are exactly $K - min_{th}$ buffer filling levels where loss is enabled, implying a multiplication of the X CTMC states where loss is enabled by a factor $K - min_{th}$. Even for a relatively small buffer size, this will cause the corresponding CTMC to increase in size fast, especially when $K - min_{th}$ is relatively large.

In addition to this, the addition of the extra `lossAQM` place will cause for an even larger CTMC state space. Instead of the loss token to be kept in `lossB` until one of the sources is selected for loss, the loss token is moved to `lossAQM` periodically. Assume that there are Y states of the CTMC where the loss token resides in `lossB`. If there would be no restraining condition on the firing of the `tAQM` transition, there would be an almost exact copy of each of these Y states, where the marking would only differ in the fact that the loss token would reside in the `lossAQM` state, instead of in the `lossB` state. Since the `tAQM` transition is only allowed to fire whenever the buffer filling level is larger than min_{th}, only a fraction $K - min_{th}/K$ of these Y states will have such an almost exact copy in the CTMC corresponding to the new model. Together with the additional state caused by the introduction of the possibility of loss before the buffer has been filled completely, this leads to the following empirically derived expression for the state space S_{new} of the CTMC underlying the new SPN model with RED behavior:

$$S_{new} \approx S_{old} + X(K - min_{th}) + Y\frac{K - min_{th}}{K}, \tag{6.3}$$

where S_{new} (S_{old}) is the state space of the CTMC corresponding to the new (old) SPN model, and X and Y are as defined above. As one can conclude from this expression, the state

space of the new model varies with K and min_{th}, where the evolution of the state space with changing min_{th} when K is fixed is probably best representing the overall state space issue of the new model. To illustrate this issue, the CTMC state space of the model for $K = 10$ and varying min_{th} is plotted in Figure 6.15. As a reference, the state space of the original SPN model using TCP drop tail for $K = 10$ is plotted as well. Note that the state space of the CTMC does not vary with either max_{th}, d_B or p_{max}.

A single experimental validation of (6.3) results in acceptable values for X and Y, e.g., for $K = 10$, $min_{th} = 2$, and S_{old} and S_{new} accordingly, the value of X lies around 20,000, whereas Y is approximately twice as big as X. These values are deemed realistic when one is considering the meaning of the X and Y variables.

As one can see from Figure 6.15, the state space of the CTMC underlying the SPN model of TCP and RED decreases linearly with the choice of the min_{th} parameter. This is caused by including this parameter in the multiplicity of the inhibitor arc from `tAQM` to `bFree`, which is set to $K - min_{th}$. The higher this parameter is set, the less often `tAQM` is enabled, thus generating less possible states. Hence, the model never generates a Markov chain having more states than necessary. For most other AQM algorithms, this probably does not apply, as they mostly have no minimum buffer threshold value, below which loss is disabled. An example of an algorithm that does use such a minimum threshold is the CHOKe algorithm (see Section 3.3.5).

6.3.5 Evaluation conclusions

Taking into account the evaluation results presented in Section 6.3.1 through 6.3.4, one could say that the SPN model using AQM behavior, which has been presented in this Chapter, provides a good model for the actual behavior of the RED algorithm. All of the derived performance measures take predictable values during the evaluation, and variation of the RED parameters min_{th}, max_{th} and p_{max} result in justifiable changes in model outcomes. When comparing the outcomes of the model incorporating RED behavior with the outcomes of the SPN model of TCP drop tail, as it has been presented in [23], the advantages of RED also show clearly. Model throughput remains roughly the same, whereas the average buffer content is lower, when all of the network parameters remain equal for both models. This is exactly what RED, and any AQM algorithm in general, tries to achieve.

The only drawback of the SPN model of RED, as it has been presented here, compared to the SPN model of TCP drop tail, is the large increase in the CTMC state space. Setting the multiplicity of the inhibitor arc from the `bFree` place to the `tAQM` transition successfully reduces the state space as much as possible, but despite this action, the state space is still far larger than that of the SPN model of TCP drop tail.

Chapter 7

Extensibility of the SPN-based TCP/AQM model

The goal of this chapter is to investigate the versatility of the SPN-based TCP/AQM model in terms of the ability to embed different types of AQM algorithms into it. For each class of algorithms, as they have been identified in Section 3.2, suitability for implementation within the SPN model will be discussed. When algorithms are found to be suitable for implementation, pointers on how the implementation process for these algorithms will look like, will be given. When an algorithm, or even a whole class of algorithms, cannot be fit into the model as it has been presented in Chapter 6, it will be discussed why exactly this (class of) algorithm(s) does not fit into the model, and pointers on how the model could be further adapted in order to be suitable for implementation of that particular (class of) algorithm(s) will be given as well, if possible. Note that these adaptations have not been carried out in the model as it has been presented in Chapter 6, since this model tries to be as generic as possible, allowing for as wide a variety of algorithms as possible. Further adaptations for the sake of implementation of a specific (class of) algorithm(s) could lead to a less generic model.

7.1 Introduction

Unfortunately, there is one aspect of the SPN modeling paradigm, and more specifically of the general idea behind Markov chains, which makes implementation of the majority of the algorithms in Chapter 3 very hard, if not impossible. This is the fact that the decision on which actions are to be taken next can only depend on information that can be derived directly from the current state, i.e., an SPN does not keep track of behavior in the past. This implies that all of the information needed by an algorithm (e.g., specific parameter values) should be derivable directly from the current state in which the CTMC resides, i.e., from the current marking of the SPN. No past parameter values can be stored for use in future calculations. The only way to work around this problem, is the storage of parameter values in places in the SPN model created specifically for that parameter. A certain amount of tokens in such a place would then represent a given value for that parameter. A clear drawback of this approach is the fact that more places, and hence more tokens, cause further expansion of the state space, which leads to an increase in the time needed to derive performance measures from the model. Another problem is that when parameter values are not limited to only integer values, the number of tokens in a place can only form an estimate of the current value

of such a parameter. This causes an obvious increase in inaccuracy of the model outcome. Of course, this does not hold for parameter values that remain the same for every state the model is in, such as the min_{th}, max_{th} and p_{max} parameters of RED. These can be included easily in the CSPL code using a standard C header file (with the `.h` extension), as has been done as given in Appendix A for RED.

7.2 Average queue length-based AQM algorithms

Apart from RED, which has been implemented into the new SPN model of TCP/AQM in Section 6.2, and of which the implementation has been evaluated in Section 6.3, there are several other algorithms which can be categorized as algorithms whose behavior is based on the average queue length in a router. Since RED has been implemented successfully into the model, a logical first step is to investigate whether other algorithms of this class (see Section 3.3) are equally suitable for implementation.

For ARED, FRED and SRED, the same motivation on why these algorithms are not particularly suitable for implementation can be used: they all make use of additional parameters which are stored and used throughout the model evaluation. In the case of ARED, this holds for the dropping probability p and the parameters used in the time-based operations. For FRED, there are even more parameters which need to be stored. This has already been identified as a drawback of the algorithm in Section 3.3.3, and this drawback is even bigger when one wants to implement this algorithm in the extended SPN model. SRED too has several variables which are used and updated throughout the algorithm. Note that it should not be impossible to implement these algorithms within the SPN model of TCP/AQM, but the fact that they all use multiple parameters which are adapted throughout the algorithm will ask for the addition of several new places, each having a significant amount of tokens, which would greatly enlarge the model state space. Other modeling techniques are probably better suited for the evaluation of these algorithms.

The CHOKe algorithm (see Section 3.3.5) provides better possibilities for implementation into the SPN model. It does not make use of any additional parameters stored and used throughout the algorithm, as its dropping probability is calculated in the exact same way as it has been done for RED. The only difference with RED is the fact that not only the incoming packet is not added to the queue, but also the packet to which the incoming one is compared is dropped if a hit occurs. This implies that, at the event of a packet arrival, with a probability p equal to the dropping probability of RED, the buffer content will decrease. With probability $1 - p$, the buffer content will increase. This can be modeled in the SPN by either firing the `decrB` transition with probability p whenever the `tAQM` transition fires, or by adding an extra transition that takes care of such a buffer decrement. With probability $1 - p$, the buffer level is increased in the usual way, and thus, this requires no additional changes to the model.

7.3 Packet loss & link utilization-based AQM algorithms

Concerning the algorithms which base the calculation of their dropping probability on packet loss and link utilization, comments on their suitability for integration with the SPN model

of TCP/AQM can be short. Both BLUE and SFB use parameters that need to be updated throughout the algorithm, making it very hard (in the case of BLUE) or even impossible (in the case of SFB) to implement them. BLUE shows many similarities with the ARED algorithm, since it updates its dropping probability in a very similar manner. The time parameters used in ARED also play an important role in BLUE, implying BLUE is as equally (un)suitable for implementation within the SPN model as ARED. Ideally, implementation of BLUE would consist of two small steps, of which the first one would be the replacement of the `REDLossProbability()` method by the code fragment as presented in Figure 7.1. The second step would be the adaptation of the multiplicity of the inhibitor arc going from the `tAQM` transition to the `bFree` place. This arc multiplicity should be changed from $K - min_{th}$ to $K - 1$, indicating that loss will not be triggered when the buffer is completely empty.

```
double BLUELossProbability() {

    if (arrivalRate() == 0) {
        link_idle = true;
    }
    if (link_idle) {
        if (current_time - last_update > freeze_time) {
            p = p - d2;
            last_update = current_time;
        }
    }
    if (mark("bFill") > L) {
        if (current_time - last_update > freeze_time) {
            p = p + d1;
            last_update = current_time;
        }
    }
    return p;
}
```

Figure 7.1: Calculation of the BLUE dropping probability in CSPL

Unfortunately, as one can see in Figure 7.1, there are several parameters in the BLUE algorithm, such as the dropping probability p and the parameter *last_update* indicating the most recent point in time at which p has been updated. The intermediate update of these parameters cannot be expressed in an SPN, unless the workaround pointed out at the beginning of this section is applied.

Regarding the SFB algorithm, it requires individual packets to be hashed into bins, requiring significant overhead by itself. To implement all these parameters in an SPN would cause an immense increase in model state space. Furthermore, the fact that SFB requires for the identification of individual packets (since they are to be hashed into bins separately), combined with the lack of individual packets in the SPN model, as the traffic has been modeled as a fluid, results in the conclusion that this algorithm cannot be implemented in the SPN model as it is.

7.4 Class-based AQM algorithms

This class of AQM algorithms also features a specific assumption which prevents the algorithms belonging to this class from being implemented successfully within the SPN model. The SPN model, as it has been set up, does not distinguish between different classes of packets, i.e., each modeled "packet" (note that there are no actual packets in the model) belongs to the same class in the model. As both CBT and DCBT distinguish between TCP and UDP classes, implementation of these algorithms is not feasible. Should the SPN model be able to make this distinguishing, then implementation would probably be relatively straightforward, as the calculation of the dropping probability in both CBT and DCBT is equal to that of RED. In Section 6.2, it has been shown that this calculation can be implemented in the SPN model.

7.5 Control theoretic AQM algorithms

For the algorithms based on control theory, the same problem holds as for ARED, SRED, FRED and most of the other algorithms mentioned in this section. These algorithms (and classical control theory in general) are all based on the continuous updating of actual network parameters according to a given reference value for these parameters. The only thing in which the control theoretic algorithms differ from one another, is the way in which they steer the network parameters that are to be controlled towards the desired reference value. This implies that network parameters need to be stored and updated continuously throughout the algorithm, which is exactly the main issue of the SPN modeling paradigm, as it has been identified in the beginning of this Chapter. It can therefore be concluded that this class of AQM algorithms is also not suitable for implementation within the SPN model. The fact that intermediate parameter values need to be stored in control theoretic algorithms can be seen directly in the code fragments calculating the dropping probability p for PI-controller (Figure 7.2) and for DRED (Figure 7.3). This code could be used when the workaround presented at the beginning of this chapter would be employed. Additionally, just as with the BLUE algorithm, the multiplicity of the inhibitor arcs should be set to $K - 1$ to prevent the loss transitions to fire whenever the buffer is completely empty.

7.6 Short evaluation of the SPN-based model of TCP/AQM

As has been shown in this chapter, the extended SPN model which has been presented in Chapter 6 is not as generic as had been assumed when the model has been designed. The main idea behind the concept of AQM, being the decision for every arriving packet on whether or not to actively drop it, based on a dropping probability which is calculated differently for every algorithm, has been implemented successfully. Unfortunately, one of the restrictions imposed by the SPN modeling paradigm, and the concept of Markov chains underlying it, provides a structural problem for the implementation of most of the AQM algorithms presented in Chapter 3, which has already been explained earlier in this chapter.

```
double PIControllerLossProbability() {

    /* calculate filtered queue length */
    q_avg = (1-filter_gain) * q_old + filter_gain * mark("bFill");

    /* calculate error signal */
    err = q_avg - q_ref;

    /* calculate dropping probability */
    p = p_old + prop_gain * err + der_gain * err_old;

    /* update variables for future calculation */
    p_old = p;
    q_old = q_avg;
    err_old = err;

    /* return dropping probability */
    return p;
}
```

Figure 7.2: Calculation of the PI-controller dropping probability in CSPL

```
double DREDLossProbability() {

    /* calculate error signal */
    err = mark("bFill") - q_ref;

    /* calculate filtered error signal */
    err_filt = (1 - filter_gain) * err_old + filter_gain * err;

    /* calculate dropping probability */
    p = min((max(p_old + prop_gain * err_filt ,0)),p_max);

    /* update variables for future calculation */
    p_old = p;
    err_old = err_filt;

    /* return dropping probability */
    return p;
}
```

Figure 7.3: Calculation of the DRED dropping probability in CSPL

Chapter 8

Conclusions and future work

In this chapter, conclusions on the work presented in the previous chapter will be drawn, along with some additional explanatory discussion where necessary. Following these conclusions, some pointers on possible future work on the subject of the modeling of Active Queue Management algorithms using Stochastic Petri Nets will be given as well.

8.1 Conclusions

The work that has been described in this report has had multiple goals, each of which will be mentioned separately in this section.

First of all, there has been a need for structuring the wide variety of AQM algorithms that have been developed since the introduction of RED in 1993. Organizing the existing algorithms into a decent structure makes it easier to see which ones are more or less alike, and therefore provides a strong basis for the comparison of different algorithms. A slightly adapted version of the classification first presented in [26] has proven to be a useful tool in this structuring. When categorizing an algorithm according to the criteria on which the decision whether or not to drop an incoming packet is being made in that algorithm, a well-structured classification tree can be set up, as can be seen in Figure 3.1.

Furthermore, it has been shown that the concept of Stochastic Petri Nets (or SPNs) is suitable for the modeling of the behavior of the RED algorithm. In fact, a straightforward extension of the SPN model of two TCP sources sharing a single (drop tail) buffer, which has been presented in [23], allows for the inclusion of generic AQM behavior in the buffer. The extension is formed by the addition of an option to not only drop packets, and to subsequently send a loss signal to a source when the buffer has been filled completely, but to allow a loss signal to be sent too when the buffer is not yet full. This relates to the main concept of AQM: the deliberate sending of loss signals before the buffer is entirely filled, in order to prevent the buffer filling level from getting too high, thereby avoiding high buffering delays.

Not only are SPNs suitable for the modeling of a set of TCP sources connected via a buffer deploying an AQM strategy, the SPN concept also has a general advantage when it comes to modeling. This advantage is the graphical nature of SPNs, which makes it easier to design a model when given a proper system definition, and also makes it easier to comprehend an already designed model. There are multiple tools available which can derive steady-state and

transient performance measures (which would cost a lot of time and is prone to errors when to be done by hand) from an SPN specification, such as SPNP [27].

A significant drawback of the SPN modeling paradigm can be seen when one wants to use the extended SPN model in combination with AQM algorithms other than RED. Almost all of these algorithms rely on information (e.g., intermediate parameter values) to be stored when the algorithm is executed. The concept of SPNs, and specifically of the Markov chains underlying these SPNs, does not allow for this type of information to be stored. The main idea behind Markov chains is the fact that decisions on future which actions are to be taken should be based only on information available in the current state. Relating this to AQM algorithms, only algorithms that exclusively use information which is directly available from the state of the SPN in their actions (such as the calculation of a dropping probability) are suitable for modeling using SPNs. Storing values in parameters, which are used in future iterations of the algorithm cannot be easily modeled using SPNs.

There is a possibility of working around this issue, which consists of the creation of a dedicated place in an SPN for each parameter for which intermediate values need to be stored. The number of tokens in this dedicated place would then represent a certain value for the parameter under attention. This approach, although being a solution to the issue presented above, comes with other drawbacks of itself. First of all, this workaround offers a significant increase in model state space, and therefore in simulation and computation time. More time will be needed to derive the desired model performance measures. Another drawback of this approach is the fact that when non-integer values are allowed for the parameters under attention, either a very large amount of tokens is necessary to express the current parameter values, which leads to a further (explosive) increase in state space, or the number of tokens in a place is only an estimation of the actual current parameter value. This can provide further inaccuracy of the model, and therefore its outcomes will represent the actual outcomes of the modeled system less precisely.

Summarizing the thoughts given above, it can be concluded that the concept of Stochastic Petri Nets has not yet proven its use in the modeling of general AQM behavior. While a slight moderation of the model presented in [23] is suitable for the modeling of RED, which is one of the best known and most used AQM algorithms, the general concept of Stochastic Petri Nets provides some limitations which make it very hard to adequately implement the behavior of most of the AQM algorithms that have been presented in the literature. There are some possible workarounds to these limitations, but these require additional assumptions, which negatively influence the accuracy of the model, and on top of that, result in an explosive increase in model state space.

8.2 Future work

There are several aspects of the modeling of Active Queue Management algorithm behavior using Stochastic Petri Nets that would benefit from further attention.

First of all, the behavior of the model in combination with the RED algorithm has only been evaluated for a relatively small buffer. Although this already provided good and realistic results, simulation of the model with a larger buffer space could either lead to an even better

foundation for the claim that the model provides a correct depiction of reality, or it could show some inaccuracies of the model which do not come to light when only a small buffer space is used. Furthermore, when using a larger buffer space, the choice of the min_{th} and max_{th} parameters is less limited, which could lead to a better evaluation of the impact on model outcomes when values for these parameters are varied.

Also, the workaround suggestion for the fact that the SPN paradigm does not allow for intermediate parameter values to be stored has not yet been evaluated in practice. It would be very interesting to see what the influence of the use of this workaround is on model performance. Without evaluating this approach in practice, it is very hard to say something about its usefulness. The approach, when combined with an AQM algorithm other than RED, could either result in relatively accurate model outcomes with a moderate and acceptable increase in model state space and, hence, in computation time, or it could provide unacceptable results due to the inaccuracy which is inherent to this workaround, with the additional problem of an explosively increasing model state space. Decision upon the question which of these two scenarios will be the most realistic therefore requires additional research. It is advised to use an AQM algorithm based on control theory, since these algorithms are fairly easy to implement once one has created the required additional places in the SPN model.

Bibliography

[1] J.F. Kurose and K.W. Ross, "*Computer networking - A top-down approach featuring the Internet*", 1^{st} edition, Addison Wesley, 2001.

[2] S. Floyd and V. Jacobson, "Random Early Detection gateways for congestion avoidance", *IEEE/ACM Transactions on Networking*, vol. 1 No. 4, pp. 397-413, August 1993.

[3] L. Su and J.C. Hou, "An active queue management scheme for Internet congestion control and its application to differentiated services", The Ohio State University, Columbus, OH, January 2000.

[4] D. Lin and R. Morris, "Dynamics of Random Early Detection", *Computer Communication Review*, Proceedings of ACM SIGCOMM 1997, vol.27 No.4, pp. 127-137, 1997.

[5] T.J. Ott, T.V. Lakshman and L. Wong, "SRED: Stabilized RED", Proceedings of IEEE INFOCOM 1999, pp. 1346-1355, March 1999.

[6] R. Pan, B. Prabhakar and K. Psounis, "CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation", Proceedings of IEEE INFOCOM 2000, pp. 942-951, April 2000.

[7] W.C. Feng, D.D. Kandlur, D. Saha and K.G. Shin, "BLUE: A new class of active queue management algorithms", Tech. report CSE-TR-387-99, University of Michigan, April 1999.

[8] M. Parris, K. Jeffay and F.D. Smith, "Lightweight active router-queue management for multimedia networking", *Multimedia Computing and Networking*, SPIE Proceedings Series, vol. 3020, January 1999.

[9] M. Claypool and J. Chung, "Dynamic CBT and ChIPS Router support for improved multimedia performance on the Internet", Proceedings of ACM Network and Operating System Support for Digital Audio and Video (NOSSDAV), June 2000.

[10] S. Ryu, C. Rump and C. Qiao, "Advances in active queue management (AQM) based TCP congestion control", Telecommunication Systems, vol.25 No.4, pages 317-351, March 2004.

[11] J. Aweya, M. Ouelette and D.Y. Montuno, "A control theoretic approach to active queue management", *Computer Networks*, vol.36, pp. 203-235, 2001.

[12] C.V. Hollot, V. Misra, D. Towsley and W.B. Gong, "On designing improved controllers for AQM routers supporting TCP flows", Proceedings of IEEE INFOCOM 2001, pp. 1726-1734, April 2001.

[13] E.D. Sontag, "Mathematical control theory - Deterministic finite dimensional systems", 2^{nd} edition, Springer, New York, 1998, ISBN 0-387-984895.

[14] C.V. Hollot, V. Misra, D. Towsley and W.B. Gong, "A control-theoretic analysis of RED", Proceedings of IEEE INFOCOM 2001, pp. 1510-1519, April 2001.

[15] J. Sun, G. Chen, K.T. Ko, S. Chan and M. Zukerman, "PD-Controller: A new active queue management scheme", IEEE Globecom 2003.

[16] S. Kunniyur and R. Srikant, "Analysis and design of an Adaptive Virtual Queue (AVQ) algorithm for Active Queue Management", Proceedings of ACM SIGCOMM, pp. 123-134, August 2001.

[17] W. Feng, A. Kapadia and S. Thulasidasan, "GREEN: Proactive queue management over a best-effort network", IEEE GLOBECOM, November 2002.

[18] NS-2 Simulator, available at http://www.isi.edu/nsnam/ns/

[19] B.R. Haverkort, "*Performance of computer communication systems - A model-based approach*", Wiley & Sons, New York, 1998, ISBN 0-471-197228-2.

[20] V. Misra, W.B. Gong and D. Towsley, "Stochastic differential equation modeling and analysis of TCP windowsize behavior", TR ECE-TR-CCS-99-10-01, presented at Performance '99, October 1999.

[21] A.Y. Khinchine, "Mathematical methods in the theory of queuing", Hafner Publishing Co., New York, 1960.

[22] 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", Proceedings of ACM SIGCOMM, pp. 151-160, September 2000.

[23] N.D. van Foreest, B.R. Haverkort, M.R.H. Mandjes, W.R.W. Scheinhardt, "Versatile Markovian models for networks of asymmetric TCP sources", April 2004. Submitted for publication.

[24] C.V. Hollot, V. Misra, D. Towsley, and W.B. Gong, "A control theoretic analysis of RED", Proceedings of IEEE Infocom, 2001.

[25] G. Balbo, "Introduction to Stochastic Petri Nets", in LNCS 2090: Formal Methods and Performance Analysis, Springer Verlag, September 2001.

[26] A.J. Warrier, "Active queue management: A survey", North Carolina State University, December 2003.

[27] Stochastic Petri Net Package, available at
http://www.ee.duke.edu/∼kst/software_packages.html

[27] K. Ramakrishnan, S. Floyd and D. Black, The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168, September 2001.

[28] E. Altman, T. Jimenez and R. Nuñez-Queija, Analysis of two competing TCP/IP connections. *Perf. Eval.*, 49:43-55, 2002.

[29] T.V. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high bandwidth-delay products and random loss", IEEE/ACM Transactions on Networking, 5(3):336-350, 1997.

Appendix A

The SPN model of RED in CSPL

This appendix shows the `moreSourcesRED.c` file, which contains the CSPL listing of the SPN model with an application to the RED algorithm as it has been presented in Section 6.2. It is preceded by the C header file `parameters.h` which includes the global parameters as they are used throughout the simulation.

```
/* header file including global parameters */

int numSources =2;
int numLossTokens = 1;
double propDelay[] = {0.5,0.52,0.2};
double fluidRate[] = {1.04,1};
int W[]={113,117,8};
double L=80.7071067811866;
double B=12.9131370849898;
int K=5;
int MINTH=2;
int MAXTH=4;
double MAX_RED=0.001;


/* multiple source TCP/AQM model in CSPL */

#include "user.h"
#include "parameters.h"
#define max(x,y) ((x)>(y) ? (x) : (y))

/* functions calculating RTTs and sending rates */
double T(int i) {return(propDelay[i]+((double)mark("bFill"))/((double)K)*
    B/L);}
double invT(int i){ return(1./T(i));}
double r(int i) {return(fluidRate[i]*propDelay[i]/T(i));}

/* decrease congestion window */
int decrWin(int i) {return(mark_1("win",i)/2. + .5);}

double tau1() {return (double)mark_1("win",0)*r(0);}
double tau2() {return (double)mark_1("win",1)*r(1);}
```

```
double tau3() {return (double)mark_1("win",2)*r(2);}

/* buffer occupancy distribution rewards */
reward_type bFillMarking0() {return(mark("bFill")==0);}
reward_type bFillMarking1() {return(mark("bFill")==1);}
reward_type bFillMarking2() {return(mark("bFill")==2);}
reward_type bFillMarking3() {return(mark("bFill")==3);}
reward_type bFillMarking4() {return(mark("bFill")==4);}
reward_type bFillMarking5() {return(mark("bFill")==5);}

double totTau()
{
    int i;
    double dum=0.;

    for(i=0;i<numSources;++i)
        dum+=((double)mark_1("win",i))*r(i);

    return dum;
}


double netInputRate() {return(totTau() - L);}

/* buffer dynamics functions */
double beta(){return (netInputRate()*((double)K)/B);}
double betaAbs() {double dum=beta(); return((dum >0.) ? dum: -dum);}
int betaPos() {return((beta() >0.) ? 1 : 0);}
int betaNeg() {return((beta() <0.) ? 1 : 0);}

/* determine firing rate for tAQM transition */
double arrivalRate(){

        int j;
        double total=0.;

        for(j=0;j<numSources;j++) {
            total+=(double) mark_1("win",j)/T(j);
        }
        return (total + 1E-16);
}

/*   throughput is equal to output process */
double gammaOut1()
{
    double dum;

    if(mark("bFill")==0){
      dum=tau1();
    } else {
      if(totTau()>0)
        dum=tau1()/totTau()*L;
      else /* totTau() = 0  => tau1() = 0 */
```

```
        dum=0.;
    }

    return(dum);
}

double gammaOut2()
{
    double dum;

    if(mark("bFill")==0){
      dum=tau2();
    } else {
      if(totTau()>0)
        dum=tau2()/totTau()*L;
      else /* totTau() = 0  => tau2() = 0 */
        dum=0.;
    }

    return(dum);
}
/* end of throughput is output */

double gammaIn1()
{
  double dum;

  dum=tau1();
  if(numLossTokens==1){
    if(mark_1("loss",0)==1)
      dum-=netInputRate();
  } else {
    if((mark("bFill")==K) && betaPos())
      dum-=netInputRate()*tau1()/totTau();
  }
  return(dum);
}

double gammaIn2()
{
  double dum;

  dum=tau2();
  if(numLossTokens==1){
    if(mark_1("loss",1)==1)
      dum-=netInputRate();
  } else {
    if((mark("bFill")==K) && betaPos())
      dum-=netInputRate()*tau2()/totTau();
  }
  return(dum);
}
```

```
double gammaIn1old()
{
  double dum;

  dum=tau1();
  if(mark_1("loss",0)==1) {
    if(numLossTokens==1){ /* in proportional loss model we assign the
                             excess fluid to the source with loss
                             token */
        dum-=netInputRate();
    } else {
     dum-=max(netInputRate(),0.)*tau1()/totTau();
    }
  }
  return(dum);
}

double gammaIn2old()
{
  double dum;

  dum=tau2();
  if(mark_1("loss",1)==1) {
    if(numLossTokens==1){ /* in proportional loss model we assign the
                             excess fluid to the source with loss
                             token */
        dum-=netInputRate();
    } else {
     dum-=max(netInputRate(),0.)*tau2()/totTau();
    }
  }
  return(dum);
}
/* end of ITC throughput */

/* loss probabilities */
double redLossProb() {

    if(mark("bFill")>MAXTH){

        /* phase 3, p = 1 */
        return 1.0-1E-16;
    }
    else{

        /* phase 2, p grows linearly up to p_max */
        double prob;
        prob = MAX_RED*(mark("bFill")-MINTH)/(MAXTH-MINTH);
        return prob;
    }
}
```

```
double redNoLossProb() {

        return((double) 1.0-redLossProb());
}

double lossProb1()
{
  if(totTau()==0) {
    fprintf(stderr,"fout\n");
    exit(0);
  } else{
    if((tau1()<=L) && (tau2() <= L))
      return(redLossProb()*tau1()/totTau());
    else if((tau1()<=L) && (tau2() > L))
      return(1E-16);
    else if((tau1()>L) && (tau2() <= L))
      return(1.0-1E-16);
    else /* this happens with (very) small probability */
      return(redLossProb()*tau1()/totTau());
  }
}

double lossProb2()
{
  if(totTau()==0) {
    fprintf(stderr,"fout\n");
    exit(0);
  } else{
    if((tau1()<=L) && (tau2() <= L))
      return(redLossProb()*tau2()/totTau());
    else if((tau1()<=L) && (tau2() > L))
      return(1.0-1E-16);
    else if((tau1()>L) && (tau2() <= L))
      return(1E-16);
    else /* this happens with (very) small probability */
      return(redLossProb()*tau2()/totTau());
  }
}

double lossProb3()
{
  fprintf(stderr,"moreSources: lossProb3() not yet implemented\n");
  exit(1);
}

/* Start of CSPL functions */

void parameters() {
    iopt(IOP_PR_FULL_MARK, VAL_YES);
    iopt(IOP_MC,VAL_CTMC);
    iopt(IOP_PR_MARK_ORDER,VAL_CANONIC);
    iopt(IOP_PR_MC_ORDER, VAL_TOFROM);
    iopt(IOP_PR_MC, VAL_YES);
```

```
    iopt(IOP_PR_PROB, VAL_YES);
    iopt(IOP_PR_RSET,VAL_YES);
    iopt(IOP_PR_RGRAPH,VAL_YES);
    iopt(IOP_ITERATIONS,1000);
    fopt(FOP_ABS_RET_M0,0.0);

}

void net()
{ /* Start of net */
    int i;

    if(numSources<1 || numSources > 3){
      fprintf(stderr,"Mind that this program does not yet\n");
      fprintf(stderr,"< 1 or > 3 sources\n");
      exit(1);
    }

    /* buffer */
    place("bFree");
    place("bFill");
    init("bFree",K);

    trans("incrB");
    iarc("incrB","bFree");
    oarc("incrB","bFill");

    trans("decrB");
    iarc("decrB","bFill");
    oarc("decrB","bFree");

    ratefun("incrB",betaAbs);
    guard("incrB",betaPos);
    ratefun("decrB",betaAbs);
    guard("decrB",betaNeg);

    /* loss node */
    place("lossB");
    init("lossB",numLossTokens);

    /* AQM node and transition */
    place("lossAQM");
    trans("tAQM");

    iarc("tAQM","lossB");
    oarc("tAQM","lossAQM");

    /* rate of the AQM transition */
    ratefun("tAQM",arrivalRate);

    /* tAQM only fires if buffer level > min_th */
    mharc("tAQM","bFree",(K-MINTH));
```

```
/* tAQM only fires when beta > 0 */
guard("tAQM",betaPos);

/* 'no loss'-loopback */
trans("tNoLoss");
iarc("tNoLoss","lossAQM");
oarc("tNoLoss","lossB");

/* sources */
place_1("winF",numSources);
place_1("win",numSources);
place_1("loss",numSources);

trans_1("incr",numSources);
trans_1("decr",numSources);

ratefun_1("incr",ALL,invT);
ratefun_1("decr",ALL,invT);

trans_1("tLoss",numSources);
iarc_1_0("tLoss",ALL,"lossAQM");

/* tLoss only fires when beta > 0 */
guard_1("tLoss",ALL,betaPos);

if(numSources==1){
    probfun("tNoLoss",redNoLossProb);
    probfun_1("tLoss",ALL,redLossProb);
} else if(numSources==2){
  if(numLossTokens==1){
    probfun("tNoLoss",redNoLossProb);
    probfun_1("tLoss",0,lossProb1);
    probfun_1("tLoss",1,lossProb2);
  } else if(numLossTokens==2){
    fprintf(stderr,"ERROR: only proportional loss mode implemented as
        of yet");
    exit(1);
  }
} else if(numSources==3){
  if(numLossTokens==1){
    probfun("tNoLoss",redNoLossProb);
    probfun_1("tLoss",0,lossProb1);
    probfun_1("tLoss",1,lossProb2);
    probfun_1("tLoss",2,lossProb3);
  } else{
    fprintf(stderr,"ERROR: only proportional loss mode implemented as
        of yet");
    exit(1);
  }
} else {
  fprintf(stderr,"Loss token is not yet implemented for\n");
  fprintf(stderr,"more than 2 sources\n");
  exit(1);
```

```
    }

    priority_1("tLoss",ALL,10);
    priority("tNoLoss",10);

    for(i=0;i<numSources;++i){
        init_1("winF",i,W[i]);

        iarc_1_1("incr",i,"winF",i);
        oarc_1_1("incr",i,"win",i);
        harc_1_1("incr",i,"loss",i);

        viarc_1_1("decr",i,"win",i,decrWin);
        voarc_1_1("decr",i,"winF",i,decrWin);
        iarc_1_1("decr",i,"loss",i);
        oarc_1_0("decr",i,"lossB");

        oarc_1_1("tLoss",i,"loss",i);
        harc_1_1("tLoss",i,"loss",i);
        mharc_1_1("tLoss",i,"winF",i,W[i]);
    }
} /* end of net */

assert()
{
    return(RES_NOERR);
}

ac_init()
{
    fprintf(stderr,"\n %d TCP Sources\n",numSources);
    pr_net_info();
}

void ac_reach() {
    fprintf(stderr, "The reachability graph has been generated\n\n");
    pr_rg_info();
}

ac_final()
{
    pr_mc_info();

    if(numSources==1){
      printf("1: %f\t%f\t%f\n", expected(gammaOut1),expected(gammaIn1));
    } else  if(numSources==2){
      printf("1: %f\t%f\t2: %f\t%f\n",\
             expected(gammaOut1),expected(gammaIn1),\
             expected(gammaOut2),expected(gammaIn2));
    } else {
      fprintf(stderr,"Computation of perf measures for\n");
      fprintf(stderr,"more than 2 sources not yet implemented\n");
    }
```

```
    /* print performance measures */
    pr_std_average();

    /* print buffer occupancy distribution */
    pr_expected("p(mark(bFill) = 0): ",bFillMarking0);
    pr_expected("p(mark(bFill) = 1): ",bFillMarking1);
    pr_expected("p(mark(bFill) = 2): ",bFillMarking2);
    pr_expected("p(mark(bFill) = 3): ",bFillMarking3);
    pr_expected("p(mark(bFill) = 4): ",bFillMarking4);
    pr_expected("p(mark(bFill) = 5): ",bFillMarking5);

    /* print total transition rate */
    pr_expected("Tot trans rate: ", totTau);

}
```

Appendix B

Performance measures of the SPN model of TCP drop tail

In this appendix, the performance measure plots for the SPN model of TCP will be shown, in order to be able to use them as a reference when evaluating the SPN model of TCP and RED, which has been presented in Chapter 6. These plots have been created using the exact same network setup as it has been used for the SPN model of TCP and RED, so that the performance measures for the original model (of TCP) and for the new model (of TCP and RED) can be easily compared.

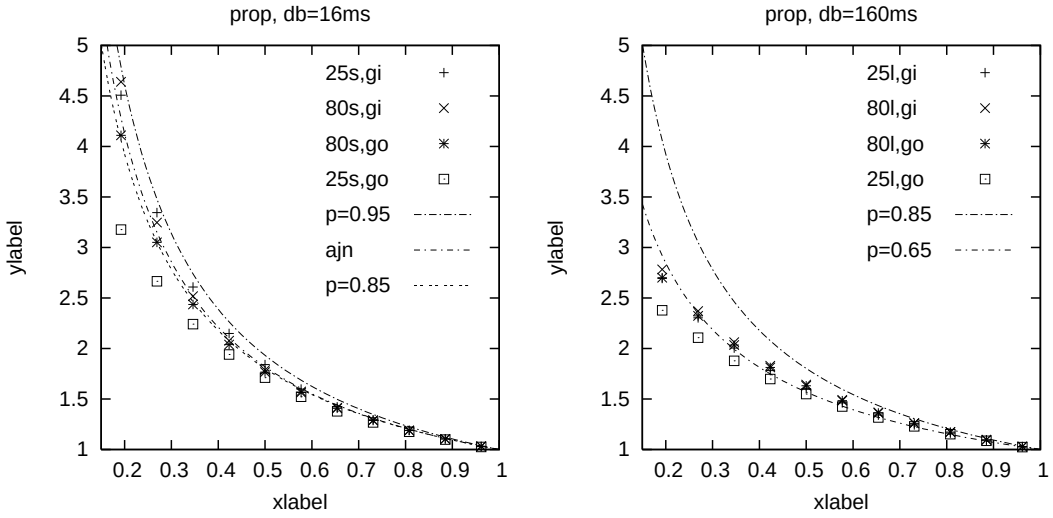

Figure B.1: Throughput ratio plots for the SPN model of TCP

In Figure B.1, the throughput ratios for the original model can be seen. Here, "prop" indicates that the loss model is proportional, i.e., only a single loss token has been used in the SPN, implying that each source suffers from loss with a probability proportional to its momentary transmission rate. The situation has been plotted for both $d_B = 16$ ms and $d_B = 160$ ms. Note that the plots in this appendix contain curves for both a high link capacity scenario ($80s$) and a low link capacity scenario ($25s$). The plots for the SPN model of TCP and RED only apply

to the high link capacity scenario ($80s$). Therefore, the plots for the low link capacity scenario can be left out of consideration. The curves denoted as $p = -\alpha$ in these plots correspond to the $s^{-\alpha}$-curves from Section 6.3.1. The "gi" and "go" suffixes correspond to the situations where $\gamma = \gamma^{in}$ and $\gamma = \gamma^{out}$, respectively. Finally, the "ajn"-curves correspond to results as they have been obtained in [28]. These results have been left out of consideration in this report as well, as they only apply to a network of TCP sources where no AQM is involved.

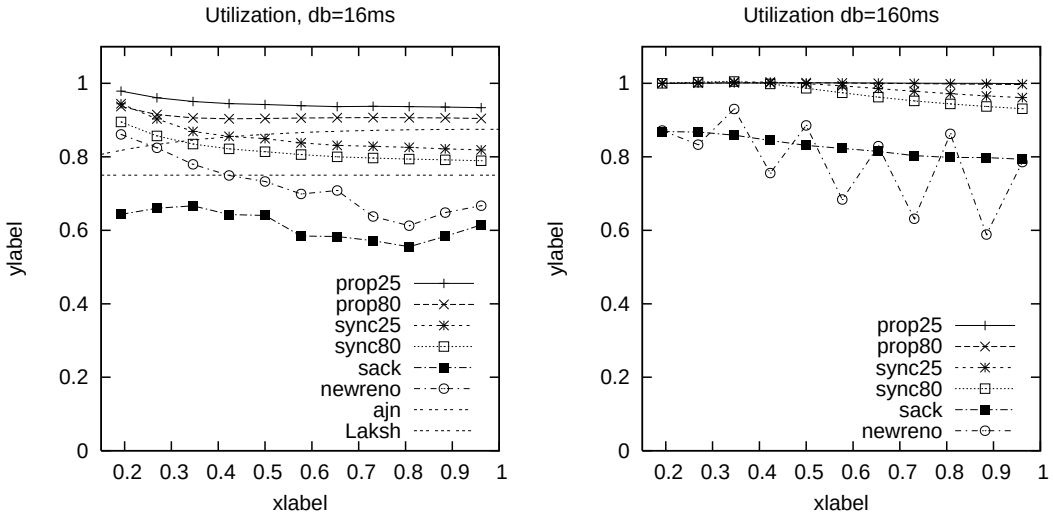

Figure B.2: Utilization plots for the SPN model of TCP

Figure B.2 displays the utilization plots for the SPN model of TCP, again for both $d_B = 16$ ms and $d_B = 160$ ms. The utilization plots for the SPN model of TCP and AQM, as they have been given in Figures 6.9, 6.10, 6.11 and 6.12, correspond to the "prop80"-curves from this figure. These curves depict the utilization for a proportional loss model with a link capacity of 80, hence, the "prop80" description in the figure. The other curves in Figure B.2 apply to scenarios with a lower link capacity, and/or with synchronous loss instead of proportional loss (i.e., there are two loss tokens, one for each source). The "sack"-curve corresponds to utilization using the TCP Sack-protocol as it has been specified in RFC 2018, the "newreno"-curve corresponds to TCP-NewReno (RFC 2582), the "ajn"-curve corresponds to the earlier mentioned results from [28], and finally, the "Laksh"-curve labels the 3/4 line, which is an estimate derived in [29].