UNIVERSITY OF OSLO
Department of Informatics

Quality of Service in Virtual Cut-through Networks

Cand. Scient. Thesis

Frank Olaf
Sem-Jacobsen

Jaunary 2004

Preface

This master thesis is the culmination of two years of work which have been done as a fulfillment of the requirements for the Cand. Scient. degree at the University of Oslo. The work on this thesis has resulted in two articles. One has already been published, while the other awaits submission to the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA).

- Sven-Arne Reinemo, Frank Olaf Sem-Jacobsen, Tor Skeie, and Olav Lysne. "Admission Control for DiffServ based Quality of Service in Cut-Through Networks." In *Proceedings of the 10th International Conference on High Performance Computing*, 2003.

- Frank Olaf Sem-Jacobsen, Sven-Arne Reinemo, Tor Skeie, and Olav Lysne. "Achieving Flow Level QoS in Cut-Through Networks through Admission Control and DiffServ." To be submitted to *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2004.

The articles are included in appendix B.

Acknowledgments

I want to thank my advisor, Tor Skeie at Simula Research Laboratory (SRL), for his guidance through the work on this thesis. Additionally I want to thank Sven-Arne Reinemo (SRL), and also Olav Lysne (SRL), for their assistance in evaluating the simulation results used in the project. I also thank John Tibbals for catching my spelling and grammar mistakes, Nils Agne Norbotten for his assistance with a final read-through, and my parents and girlfriend for the support they have given me.

Abstract

This thesis explores the possibility of achieving class level and flow level Quality of Service guarantees in a Virtual Cut-Through network with a class based Quality of Service mechanism in conjunction with admission control. There is an increasing number of System Area Network technologies based on the Virtual Cut-Through principle. Many of these support Quality of Service mechanisms, but little work has been done on performing admission control in Virtual Cut-Through networks.

Three different admission control algorithms for use in Virtual Cut-Through networks are proposed in this thesis. All three algorithms operate in accordance with the DiffServ philosophy, but the basis for their admission control decisions differ. The first relies on apriori knowledge of the capacity of each link, and has information about the load on each link in the network. Its decision is based on whether the links can support more traffic. The second method performs measurements at the egress of the network to ascertain whether the network can tolerate an increase in traffic with a given latency requirement. The third and final method for admission control measures the jitter of special probe packets as the basis for its decision.

An evaluation of the proposed algorithms is presented through extensive simulation results. The Quality of Service properties that are studied are the ability to give bandwidth guarantees to each individual flow, and to the service class as a whole, and the latency and jitter characteristics that the traffic displays with the different admission control algorithms. Through these simulations the apparent limits of the admission control algorithms are discovered, and the range of QoS guarantees that may be achieved in Virtual Cut-Through networks becomes clear.

The simulations show that throughput guarantees on the class level and the flow level are achievable, but that latency and jitter in VCT networks are hard to control. Finally, packet dropping is investigated as a method for reducing packet jitter. The results show that this method is able to reduce the jitter perceived by the network traffic, but it does not outperform some of the admission control algorithms.

Contents

Preface 1
Acknowledgments . . . 1

1 Introduction 7
1.1 Thesis . . . 10
1.2 Readers Guide . . . 10

I Background 12

2 Problem Domain 13
2.1 Switched Networks and Interconnection Networks . . . 13
2.1.1 Switching . . . 15
2.1.2 Routing and Topology Issues . . . 18
2.1.3 Scalability . . . 20
2.2 InfiniBand . . . 21
2.2.1 Elements of an InfiniBand Network . . . 22
2.3 Quality of Service in Internet . . . 24
2.3.1 End-to-End Quality of Service . . . 25
2.3.2 Heterogeneous Networks . . . 30
2.3.3 MPLS/GMPLS . . . 30
2.3.4 Quality of Service in Switched Networks . . . 31
2.4 Admission Control . . . 32

3 VCT Quality of Service 34
3.1 Related work . . . 34
3.2 Quality of Service in IBA . . . 37
3.3 Admission control in IBA . . . 39
3.3.1 Is Admission Control Necessary? . . . 39
3.3.2 Back-pressure issues . . . 40

4 Properties of Admission Control **42**
4.1 Characterisation of a good admission control algorithm 42
4.1.1 The Network Utilisation vs. QoS trade-off 44
4.2 Arbitration Tables and Admission Control 45

II Application of Admission Control **47**

5 Admission Control Algorithms **48**
5.1 Switch Level Admission Control, Link-by-Link 48
5.1.1 Parameter Based 48
5.1.2 Measurement based 50
5.1.3 AC Differentiation for Bandwidth Requirements 51
5.2 Endpoint Admission Control 56
5.2.1 Egress Admission Control 56
5.2.2 Jitter Probing 61
5.3 Centralised Admission Control 62
5.3.1 Link-by-Link 62
5.3.2 Combinations 64
5.4 Aiming for Low Jitter 64
5.5 Summary of the Proposals 65

6 Simulations **67**
6.1 Simulation Environment 67
6.1.1 Network Components in the Simulator 67
6.1.2 The simulator engine 68
6.1.3 Network Topologies 68
6.1.4 Routing 69
6.2 Simulation Parameters 69
6.2.1 Admission Control Criteria 71
6.2.2 The Nature of the Simulations 74
6.2.3 Other technological assumptions inherent in the simulator . 76
6.2.4 Traffic Distribution 77
6.2.5 Traffic Generation 77

7 Results and Evaluation **82**
7.1 Target for Admission Control 83
7.2 Throughput/Network utilisation 83
7.2.1 Total Throughput 83
7.2.2 The Throughput of Each Flow 89
7.3 Latency 94

7.3.1 Random Pairs . . . 94
7.3.2 Two Hot-spots . . . 96
7.3.3 Summary . . . 97
7.4 Jitter . . . 98
7.4.1 Random Pairs . . . 99
7.4.2 Two Hot-spots . . . 101
7.4.3 Summary . . . 101
7.5 Throughput/QoS Trade-off . . . 102
7.6 Achieving Low Jitter . . . 103
7.7 Concluding Remarks . . . 107

8 Conclusion **108**
8.1 Further work . . . 109

A Additional figures **117**

B Produced Articles **120**

C Simulator Source Code **149**

List of Figures

1.1 A typical Internet configuration, client on a LAN, server on a SAN 9

2.1 A switched network . . . 14
2.2 Deadlock in a packet switched network . . . 20
2.3 A figure illustrating the back-pressure mechanism in VCT networks 23
2.4 Two DiffServ domains, each controlled by a BB. Clients and servers connect to the boundary nodes, data is forwarded efficiently in the interior. . . . 28

5.1 Latency distribution for SL1, 3 hops in a unsaturated and saturated network . . . 63
5.2 Figure depicting saturated and non-saturated regions indicated with high and medium marker . . . 63

6.1 Average flow latency as a function of hops . . . 73
6.2 Jitter as a function of Hops . . . 75
6.3 Switch architecture . . . 76
6.4 Packet rate at an increasing timescale . . . 81

7.1 Aggregated Throughput . . . 84
7.2 Aggregated Throughput for Two Hot-spots . . . 87
7.3 Mean per flow rate . . . 90
7.4 Mean per flow rate for two hot-spots . . . 92
7.5 Mean of mean flow latency . . . 95
7.6 Mean of mean flow latency for two hot-spots . . . 97
7.7 Maximum packet jitter for a flow . . . 99
7.8 Maximum packet jitter of flows for two hot-spots . . . 102
7.9 Trade-off between QoS and network utilisation . . . 104
7.10 Throughput with packet dropping . . . 105
7.11 Maximum packet jitter with packet dropping . . . 105
7.12 Distribution of how many packets flows have successfully sent through the network . . . 106

A.1 Average packet jitter for a flow 117
A.2 Average packet jitter of flows for two hot-spots 118
A.3 Average packet jitter with packet dropping 119

List of Tables

6.1 Service Level and Virtual Lane Configuration 70
6.2 Latency requirement distribution for each hop for use with Egress Measurements 72
6.3 Jitter requirement distribution 74
6.4 The SL and VL added for use with Jitter Probing, SL 1 - SL 5 remain the same 74

7.1 Admission Control Schemes Sorted by Network Utilisation 89
7.2 Percent of flows with full throughput with different AC 91
7.3 Admission Control Schemes Sorted by Bandwidth Guarantees . . . 93
7.4 Admission Control Schemes Sorted by Latency 98
7.5 Mean and standard deviation for SL 1, 3 hops at a offered packet rate of 1,3 packets per cycle (high load) 101
7.6 Admission Control Schemes Sorted by Jitter 103

Chapter 1

Introduction

The Internet has shown an exceptional growth over the past few years. It has evolved into a global infrastructure supporting services from e-mail to complex multimedia applications and business transactions. Many of these new emerging applications place demands on the network with regards to throughput and latency. In essence they require a certain Quality of Service (QoS).

Quality of Service (QoS) may be described from two distinctly different points of view:

"QoS is the measure of how good a service is, as presented to the user. It is expressed in user understandable language and manifests itself in a number of parameters, all of which have either subjective or objective values."[1]

In other words, a user has subjective demands for QoS guarantees from the network, such as that the network traffic should arrive fast and correctly. These guarantees are realised in the network in the form of guarantees for bandwidth, latency/jitter, low packet loss etc. The user demands are translated to QoS parameters in the network, and the QoS guarantees the network is able to provide is translated to a user understandable form.

The Internet today is basically a best effort network. The Internet Engineering Task Force (IETF) has therefore undertaken the task of defining Quality of Service mechanisms for use in the Internet. This has resulted in the definition of IntServ (Section 2.3.1), MPLS/GMPLS (Section 2.3.3) and DiffServ (Section 2.3.1). IntServ is a Quality of Service scheme based on per flow reservation of resources in the network by the use of the Resource Reservation Protocol (RSVP). GMPLS is a generalisation of MPLS, a forwarding scheme using label switching to provide efficient forwarding and service differentiation based on labels. DiffServ

[1] RACE D510, F. Fluckige 1995

on the other hand defines a relative Quality of Service concept with no explicit signaling or per flow information in the core of the network. Neither of the IETF QoS concepts specify how QoS is to be achieved on the link level, e.g. how routers and switches should treat different traffic. For IntServ however, Integrated Services over Specific Link Layers (ISSLL) is an organ that provides specifications and techniques for mapping the QoS requirements to the different link layers [4].

As Internet applications become larger and involve larger data transfers, and as the number of users of each application increases, so does the load on the servers providing the application. Applications supporting a huge number of users, such as streaming media, network storage, and large information databases place great demands on the resources of the application server. There has therefore been a move from single server environments to applications running on a cluster of machines. To facilitate this we have in the recent years seen the development of several new technologies for use in "System Area Networking" and "Local Area Networking" (SAN/LAN). These technologies include InfiniBand [9], Myrinet [16], Autonet [28], Tnet [33], and Gigabit Ethernet [58]. Gigabit Ethernet is the only listed technology relying on store and forward switching, each of the other technologies rely on virtual cut-through (VCT) or wormhole switching, technologies supporting a back pressure mechanism. These switching techniques are presented in Section 2.1.1.

TNet [33] is profiled as a reliable System Area Network for use as an interconnection technology for interconnecting CPUs and peripheral hardware in a cluster. The technology is based on wormhole switching to achieve bounded worst-case latencies. Both Myrinet [16] and Autonet [28] rely on virtual cut-through switching. Similar to TNet, Myrinet is a high-speed interconnection technology for use in SANs. Autonet is an older LAN technology operating at lower speeds than Myrinet and TNet. None of the technologies described here support Quality of Service mechanisms for differentiating traffic, but are instead designed with certain QoS characteristics, e.g. the bounded latency in TNet.

In a typical client/server transaction, such as a user requesting a media stream from a multimedia distribution system in the Internet, the distribution system is typically located in a high-capacity System Area Network that forms a server cluster. On the client side the user will typically be located on a computer terminal connected to a LAN, e.g. Ethernet, with the traffic between client and server traversing several different network technologies over the Internet, see figure 1. It is crucial that we are able to achieve predictable communication between the different nodes making up the cluster for a server cluster to operate satisfactorily. In order for the traffic from the server (in this example a video stream) to receive the demanded Quality of Service, every network technology step must support some sort of (unified) Quality of Service mechanism. As DiffServ seems to become

the most prominent Quality of Service paradigm for the Internet, it is important that the "Quality of Service"-mechanism in each network can operate with DiffServ at the higher level. This is challenged by the emerging SAN/LAN technologies such as InfiniBand [9] and Gigabit Ethernet [58]. These technologies are equipped with their own Quality of Service mechanisms for inter-operation with DiffServ or some other mechanism at the higher level.

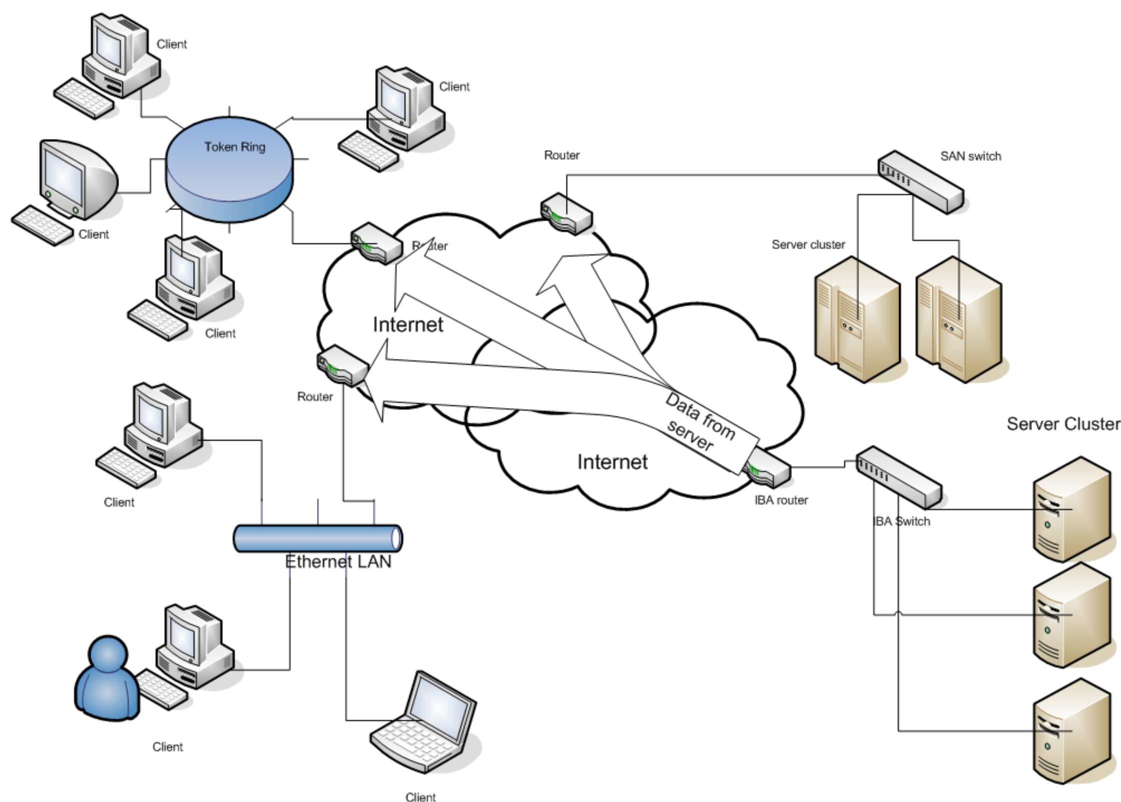


Figure 1.1: A typical Internet configuration, client on a LAN, server on a SAN

To ensure the predictability of the interprocessor communication in a cluster, regardless of the demands on the network, it is necessary to introduce some method of admission control (AC). The admission control should limit the amount of traffic in the network, so that the Quality of Service always remains within predictable limits [68] [74].

As mentioned, most emerging SAN technologies are based on virtual cut-through switching or other mechanisms supporting back-pressure. The back-pressure mechanism in virtual cut-through networks complicates achieving predictable transfers, and we will see in section 3.1 that there have been few contributions in the field of admission control in virtual cut-through networks, none of which are, to our knowledge, associable with DiffServ.

The back-pressure mechanism in virtual cut-through networks makes latency demands hard to meet. It is therefore necessary to find an admission control algorithm that efficiently provides latency guarantees to flows in the network. With the aid of admission control the network should be able to provide absolute guarantees as to bandwidth per flow/per service level regardless of the amount of admission requests sent by the hosts.

1.1 Thesis

In light of the situation outlined above, this project aims to propose and evaluate several admission control algorithms for use in virtual cut-through networks to achieve per class and per flow guarantees with regards to throughput, latency and jitter. This will be done in conjunction with a class-based, DiffServ compatible and thus flow negligent, Quality of Service scheme. The admission control schemes will be tested in a simulated Infiniband inspired network using the Quality of Service mechanisms as specified by the InfiniBand Trade Association.

1.2 Readers Guide

This thesis is divided into two parts. Part I describes the current state of affairs in interconnection networks and argues the need for admission control to be deployed to achieve better QoS. This is done by introducing the necessary background: interconnection networks, switching techniques, Quality of Service, and the concept of admission control. Also, InfiniBand is introduced as the technology of choice for the simulations in this project, and the relevant characteristics of this technology are presented. The second part starts by presenting several algorithms adapted from other environments, e.g. Internet. Next, the simulation environment is given, and finally the obtained simulation results are presented and the proposed algorithms are evaluated.

Part I – Background:
Chapter 2 introduces interconnection networks and the various switching techniques used in these. The different challenges pertaining to such networks are introduced in section 2.1. Section 2.2 introduces the InfiniBand Architecture model, the architecture used as a technology basis in this thesis. Section 2.3 provides an overview of the current Quality of Service mechanisms used in the Internet today. Finally Section 2.4 introduces admission control.

Chapter 3 gives an overview over related work pertaining to Quality of Service

and admission control in virtual cut-through networks, before the Quality of Service mechanisms available in the InfiniBand Architecture are introduced in Section 3.2 and an introduction to the admission control problem domain is given in section 2.4.

Chapter 4 gives criteria for evaluating the efficiency of the admission control algorithms. Section 4.2 discusses admission control in relation to the InfiniBand arbitration tables.

Part II – Application of Admission Control:
Chapter 5 proposes and describes several admission control algorithms to be evaluated in Chapter 7.

Chapter 6 gives an overview of the simulator in Section 6.1, and describes other simulation specific details in section 6.2.

Chapter 7 presents the target for admission control in Section 7.1. Subsequently the proposed admission control algorithms are evaluated with regard to throughput in section 7.2, latency in section 7.3, and jitter in section 7.4 using the evaluation criteria given in Chapter 4. In Section 7.6 results are presented for an alternative method for achieving low jitter.

Chapter 8 presents a conclusion to this thesis and further work is outlined in section 8.1.

Appendixes:
Appendix A contains additional simulation results.

Appendix B contains the papers that have been published/submitted during this project.

Appendix C contains the source code added to the simulator for this project.

Part I

Background

Chapter 2

Problem Domain

2.1 Switched Networks and Interconnection Networks

In switched networks every device/node in the network is connected to the rest of the network via a serial line. At the other end of the serial line there may be another node, a switch, or a network router, the purpose of which are described below. These nodes, switches, and routers may again be interconnected to other such devices, forming a large switched network as the one in Figure 2.1.

A switched network may in principle be of arbitrary size. It might consists of as little as two nodes and a switch, or be a large Wide Area Network (WAN) spanning a whole city. A special type of switched networks are interconnection networks. These are System Area Networks interconnecting nodes in a tightly coupled group, typically a cluster, a parallel computer, or even within the switching fabric of some switches. Interconnection networks are further divided into two categories. If every switch in the network is connected to one processor and one or more other switches, we have what is known as a Direct Interconnection Network [60]. If there are switches that are not directly connected to any processor, such as in a Multistage Interconnection Network (MIN), the network is termed an Indirect Interconnection Network [60].

Common for interconnection networks is that they require low latency and do not tolerate dropping of packets. To inhibit packet dropping such networks employ a credit-based link-level flow control ensuring that no packets have to be dropped due to insufficient buffer space. Such a flow control mechanism may lead to blocking of packets throughout the network, leading to an increased possibility of deadlocks. This places strict demands on the design of interconnection networks, and especially the routing protocol to be used.

The process of forwarding data (switching), deciding where to send the data

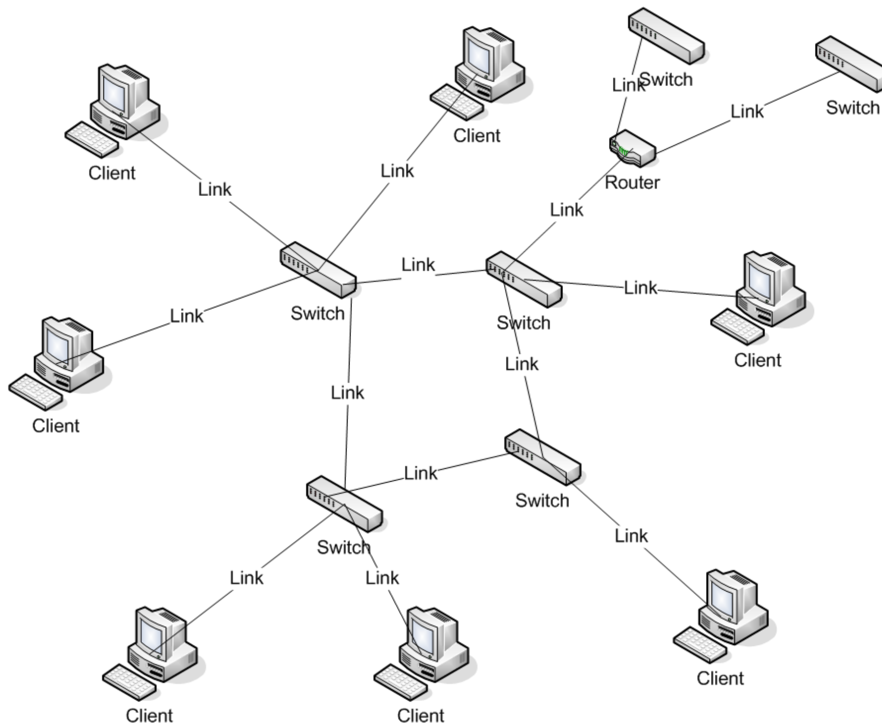


Figure 2.1: A switched network

destined for a specific end node (routing) and how to avoid deadlocks is the topic for the next few sections in this chapter.

2.1.1 Switching

Moving data from one link to another is known as switching, a process most commonly performed by switches as described above. We will now take a closer look at some of the more common switching techniques.

Circuit Switching

One of the oldest widespread switching methods is line- or circuit-switching. In a circuit-switched network, a physical connection between an incoming line and outgoing line is established so that the traffic in the network may pass effortlessly through the switches from the source to the destination [25]. This switching technique was originally used in telephony networks where a physical line was established through the network from the caller to the callee as the caller dialed up the numbers on the telephone [23]. Nowadays most of this is done digitally and the telephone switch boards are replaced with microcircuits, but the concept still applies, there is a dedicated line from the caller to the callee.

There exists several data networks based on this switching principle where a line is established from the sender to the receiver. Optical networks are switched networks based on optical fiber instead of electrical wires. Data in optical networks are represented by light waves, a medium which is difficult to buffer and control. To avoid the time-consuming process of converting between electrical and optical signals, many optical switches use a technique based on separating light waves through multiplexers and demultiplexers to propagate traffic to the correct output link. In essence the data travels on a continuous optical circuit. An overview of different optical switching techniques may be found in [67], and a technology based on optical circuit switching is TeraNet [29]. Another network technology based on (virtual) circuit switching is ATM [44], a technology developed by the telecom industry. In this case the circuit switching is done at a slightly higher level than in optical networks. ATM is basically packet switched (cell switched), but in order to send data through the network a virtual circuit (VC) has to be established from the sender to the receiver. The packets are then forwarded based on a virtual circuit identifier (VCI) present in every packet header. Associated with each VC is a set of resources which gives the VC the ability to give a certain Quality of Service guarantee. The result of these mechanisms is similar to that of having a circuit-switched link through the network.

Given a general packet switched network, it is possible to achieve a form of circuit switching similar to that used in ATM through the use of IntServ and

RSVP. This method is presented in Section 2.3.1.

Circuit switching is a technique well-suited for telephony since there is a reserved path through the network from the sender to the receiver once the call has been set up, ensuring that everything said at one end will arrive at the other. This works well as long as the participant at either end of the line is talking, but if the participants stop talking to each other without hanging up, the line will still be reserved without any traffic on it. A reserved line may not be utilised by other applications and the network resource is therefore wasted [25]. This means that any other traffic in the network is unable to utilise the resources used by an already established connection, the resources are effectively reserved. This fact makes it possible for the network to provide a clearly defined and strict Quality of Service to the traffic using a given connection at the expense of adaptability.

Packet Switching

In a data network circuit switching is not necessarily an ideal technique since much of the traffic in such a network consists of short to moderate length bursts of data, for example, a single web page from the Internet. If the client were to initiate a circuit to the web server for each web page to be downloaded, and then tear down the line when the web page is downloaded, a considerable amount of time would be wasted setting up and tearing down circuits. If, on the other hand, the client initiated a circuit to the web server and then left it open, even while not downloading web pages in case another page would be downloaded later, network resources would be wasted which might be more efficiently utilised by other traffic. This is where the invention of packet switching gains its importance. Instead of reserving a line through the network and sending a continuous stream of data over the line, the data is divided into several packets of a fixed or variable size [25]. These packets are then individually sent through the network leaving the switches to handle the packets one by one as they arrive and forward them to the correct destination based on the destination address and other information present in every packet header. The actual forwarding is done by looking up this header information in a forwarding table that indicates which output port should be used for each packet destination. The forwarding table may be statically configured at system initialisation or by a distributed routing protocol running on the network. Packet routing will be explained in section 2.1.2.

It is the nature of packet switching that packets from several packet streams are interleaved (statistically multiplexed) in various ways on different links through the network. As opposed to circuit switching where the circuit only holds traffic of that specific connection, packets in packet switched networks may be held back if there currently are other packets being transmitted on the link it is destined for. The Quality of Service received by a certain packet stream is thus dependent

on the amount of additional traffic in the network, traffic which possibly has no relation to the stream in question.

Store and Forward Switches differ in the way they treat packets as they are forwarded through the network. When a packet arrives on an input link the switch may buffer the whole packet regardless of the state of the output link, or buffer parts of the packet depending on whether the output link is busy or not.

The traditional switching principle is store and forward switching. When a packet starts arriving the switch gathers up all data for that packet. Only when the whole packet has arrived will the switch perform a forwarding table lookup based on the header information in the packet and possibly perform checksum calculations to detect packet errors. This requires that the whole packet has to arrive, be stored in a buffer, processed, and then transmitted. Depending on the size of the packet it takes some time for it to arrive at the switch and the switch has to have buffer space for several packets. When the packet's path through the network takes it across several switches this might constitute a considerable delay.

Store and forward is the switching principle used in Switched Ethernet. This gives the switches the opportunity to perform CRC checks and drop bad packets. Switched Ethernet is an extension of the original Ethernet definition [43], but using point-to-point links and switches instead of a broadcast medium.

Virtual Cut-Through In an effort to minimise latency through the network and buffer requirements in the switches a switching method known as Virtual Cut-Through (VCT) switching was developed [39][25]. Each packet is divided into small data units called flits, with a size typically between one and several bytes. When a switch receives the first flits of a packet it gathers them up until it has the necessary header information to forward the packet to the correct output link if it is free. As the rest of the flits of the packet arrive, they are forwarded directly to the correct output link without the need for internal buffering. If it should happen that the output link is busy when the first flits of a packet arrive, the rest of the flits are gathered up as they arrive and the packet is buffered as a whole in the switch. It is then possible to perform error discovery and correction routines if necessary. The necessity of buffering every packet in every switch is avoided and the network is effectively speeded up.

In order to ensure that the receiving end of a link does not become overloaded by the sender transmitting too many packets, VCT networks may utilise flow control. Flow control is a mechanism for limiting the sending rates of the sender at one end of the link according to the capabilities of the receiver at the other end of the link [65]. Typically the flow control mechanism will prevent the sender from sending packets if there is no buffer space available at the receiver. Note that this

flow control is performed at a link by link level, there is no specific end-to-end flow control involved at the link layer in VCT.

The flow control causing the buffering of packets on a busy output link leads to a back-pressure mechanism where a queue builds up in switches upstream from the switch in question, so that a queue forms and causes ripples throughout the network, affecting the QoS properties of the networks. This effect will be studied in more detail in section 3.3.2.

Wormhole Switching Wormhole switching is a switching technique that takes the VCT ideas to the extreme by only providing buffer capacity in the switches to store a couple of flits [25]. Wormhole switching utilises flow control in the same manner as VCT switching. When an output link is busy, the current switch only buffers a few flits of the packet. This blocks the upstream switch which also buffers a few flits of the packet and so on upstream. The results is a packet that is spread out over the whole network. This decreases the buffer requirements of the switches to the bare minimum, but the danger is an increased likelihood of deadlocks since a single packet occupies resources in several switches. Therefore, and also because the price of buffer space has decreased, wormhole switching is not widely deployed today. Deadlocks will be explained more fully in the next section (2.1.2).

While store and forward switching is widely in use today, wormhole switching and especially virtual cut through switching are the techniques most commonly used in high-performance interconnection networks.

2.1.2 Routing and Topology Issues

We have seen the many ways in which a switch handles a packet while it consults its forwarding tables to decide via which link the packet is destined to depart. The question now is where do their forwarding tables come from. The answer is: they are built by a routing protocol running on the network. The routing protocol utilises a routing algorithm. The routing problem is basically how to compute the forwarding tables for the switches in such a way that a packet is forwarded to its destination. In order to achieve this, the routing algorithm has to satisfy several goals. The main goal is of course to compute routes through the network which ultimately conduct every packet to its final destination. To achieve this the routes must be created in such a way that a packet will never be forwarded from switch to switch, in a loop, without ever reaching its destination. Such a loop is called a "livelock". Packets are continuously forwarded through the network, but the packets never reach their destination. Livelocks are difficult to discover since the network seems to operate correctly. The difficulty of creating livelock-free routes depends on the complexity of the network topology. In a network with a regular

topology the nodes and switches are interconnected in well defined and well known patterns, such as Multi-stage Interconnection Networks (MIN) [60]. It is possible to use the knowledge of the well defined patterns to build loop free and shortest path routes. Networks with irregular topologies on the other hand are a much bigger problem, they require a generic routing algorithm which is able to handle any type of network topology. Ad hoc networks are typically irregular networks, there is no plan or system behind the network connections. Regular networks which are expanded in some direction might become irregular, as do networks in which a node or a link fails. There has been done much research in the area are of routing in irregular networks, and several generic algorithms such as Up*/Down* routing [55], Destination Renaming [48], and Layered Shortest Path (LASH) [64] have been developed. Also refer to [49] for another method for achieving adaptive routing in cut-through networks.

Other properties of routing protocols are their adaptability and tolerance for faults. A fault tolerant routing algorithm detects network errors such as a link fault or other topology changes, and rebuilds the forwarding tables to avoid the problem. An adaptive algorithm will try to balance the load on the network, that is, when there is much traffic on one link and less traffic on another link, some of the traffic from the heavily loaded link may be moved to the lightly loaded link.

The Deadlock Problem

In switched networks links and buffers are shared resources with packets competing for access to them. In store and forward and VCT, the critical resources are buffers since a packet is buffered when it encounters a busy link. As in all cases involving critical resources a deadlock is also possible here. A deadlock occurs in this case when several packets are holding a resource, a buffer in one switch, and requesting a new one, a buffer in another switch, forming a cyclic dependency. This is illustrated in Figure 2.2. The figure shows four switches organised in a ring, each switch has a link to the next one. The buffer space in each switch is occupied by a packet waiting to be transmitted on the link to the next switch forcing the packet in the previous switch to be held back. A circular dependency between the packets in the different switches is therefore formed. The situation in Figure 2.2 may be described in terms of a Channel Dependency Graph (CDG). A Channel dependency graph is a graph illustrating the dependencies between different channels in a network. A possible deadlock is characterised by a cycle in the CDG.

There are two different ways of handling the deadlock problem [23].

- Deadlock recovery
- Deadlock avoidance

One approach is deadlock recovery where one tries to recover from a potential deadlock situation by, for example, introducing time-out on the packet forwarding. When the timer expires and the packet has not yet been forwarded, the packet may either be dropped or for example sent out on a free link to let it continue to its destination from there [60]. Although such schemes work it is difficult to detect all deadlocks correctly and then recover from them. It is important that no deadlocks remain undetected, and that few false deadlocks are detected. The detection of deadlocks often rely on timeouts, values which are very difficult to set in such a way as to operate most efficiently.

The most commonly used technique is deadlock avoidance. In this case the routing protocols calculate the routes in such a way as to avoid the cyclic dependencies that otherwise might cause a deadlock.

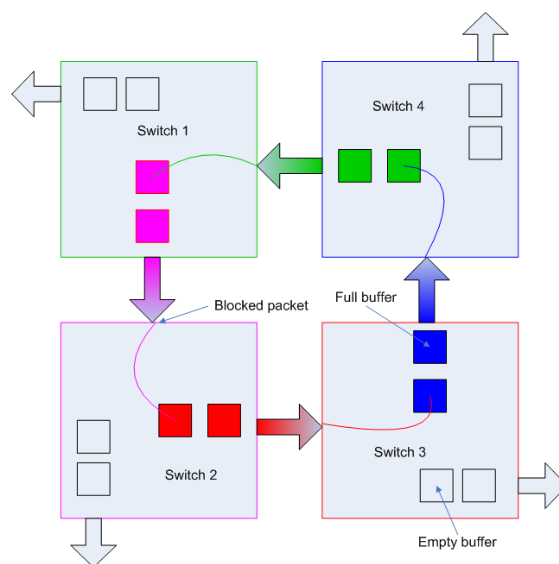


Figure 2.2: Deadlock in a packet switched network

2.1.3 Scalability

A switched network can be scaled to increase its size by adding switches and links to the network. As it becomes necessary to add more nodes to an existing network each node has to be connected to the network through a new link and possibly through new switches if all the available switches are in use. Adding additional links and switches to the network will increase the network's theoretical data transfer capacity, but the actual throughput from the network is not increased to the same extent. Each individual link is still only able to support the same

transfer rate, and as the number of nodes in the network increases so will the amount of data crossing some links in the network. Depending on network topology and traffic distribution, some links experience an increase in traffic as the number of nodes in the network increases, while their capacity remains the same. This means that in most cases the actual capacity of the network and the load put onto the network by the processors does not have an equal growth rate. Consequently there is a limit to the capacity of a network which is expanded solely by branching out with new links and switches. The scalability of the network throughput is limited.

As the number of nodes in the network increases so does the number of addresses the network must be able to route packets to. If every switch should have listed the output port to use for every node in the network, the switches would quickly use a vast amount of space to hold the information. The network size is thus limited by the method used for assigning addresses and organising the network forwarding. The size of the network also influences the time spent on building the routing tables [23].

We will see in Section 2.3.1 that the choice of Quality of Service mechanism may also affect the expandability of a network. Quality of Service mechanisms which require a certain amount of information in the network depending on the amount of network traffic will not scale as well as mechanisms which have a constant amount of information stored in the network components regardless of the amount of traffic [47].

For a network to be scalable, every component of the network should operate with an efficiency unrelated to the amount traffic in or the size of the network. This means that the amount of information stored in the network nodes should not increase (much) as the size of the network increases, and that the time spent performing critical tasks should remain low [23].

2.2 InfiniBand

InfiniBand is a newly specified serial-line switched network technology. The specification has been developed by the InfiniBand Trade Association, an association which is supported by several large technological companies including IBM, Intel and SUN [8]. The InfiniBand Architecture (IBA) is an interconnection network intended for use in System and Local Area Networks. It is basically a layer 2 technology, but it also specifies higher layer protocols. The switching principle used in IBA is VCT with the addition of a Quality of Service mechanism built into the switch architecture. These properties make IBA a good choice for the network architecture to be used in this project; it supports QoS mechanisms and relies on VCT, giving us the opportunity to evaluate the degree of QoS guarantees that

may be achieved in such networks through the use of different admission control schemes.

2.2.1 Elements of an InfiniBand Network

As a serial line switched network an InfiniBand network is comprised of several major components. In the following paragraphs the main components of IBA as they are described in [9] will be presented.

Links

Every network is dependent on what links the different components of the network together. According to the specification of InfiniBand[9] an InfiniBand network can support both twisted pair and optical cable. The bandwidth of each link is specified to be 1.2 Gbps. Additionally, IBA offers the option of combining several such links in parallel configurations of 4 and 12 links with the respective bandwidths of 10 Gbps and 30 Gbps. A link may be divided in up to 16 logical channels, called Virtual Lanes (VL). Each virtual lane has separate send and receive buffers at both ends of link. VL 0 is always required to be present as the basic data VL, a virtual lane for ordinary network traffic. Each link must also support VL 15 which is reserved for management traffic as described later on in Section 2.2.1.

Switches

Switches are one of the types of network nodes the links interconnect. The main purpose of an InfiniBand switch is to accept incoming packets, inspect the header, perform a forwarding table look-up, and forward the packet on to the correct outgoing link in a way that brings the packet closer to its final destination. In order to achieve this the switch looks at a part of the packet header called the Local Route Header (LRH). Every InfiniBand network component; switches, nodes, and routers, are identified by a Local Identification (LID). In addition to this every switch has a forwarding table constructed by the Subnet Manager (SM), to be described below, in which the correct outgoing link for every LID in the subnet is listed.

A switch has three built in Quality of Service mechanisms: a service level to virtual lane mapping table, two arbitration tables, and a Limit of High-Priority (LHP), all of which will be described in Section 3.2. The specification states that the switches in an InfiniBand network should operate as VCT switches as described in section 2.1.1. Thus if a packet is delayed in a switch, packets will queue-up in every switch upstream until the end-node is prohibited from sending. The switches shall not and cannot drop packets.

Flow control At the link level InfiniBand utilises flow control in order to prevent packet loss at the receiving end of a link. This is done by using a strict credit based scheme in which a packet is sent out on a link only if there is available credit for that particular receiver. The flow control operates on a VL by VL basis and there is no end-to-end flow control specified. The terms sender and receiver correspond to the sending and receiving side of a physical link. Every node and switch in the network keeps track of the total amount of data that has been sent since link initialisation. The receiver uses this information to send credits back to the sender indicating the additional amount of data it is allowed to send. The receiver will continue to send more credits to the sender as long as it has available capacity in its buffers. In this way, when a receiver's buffer is full, the sender will be blocked from sending new packets to the receiver. The sender will then quickly fill up its own buffers and stop providing credits to the nodes sending to it. In this way blocked packets will lead to packets being blocked in the upstream switches (switches sending data to the current switch). This behavior continues in a chain reaction ultimately forcing the sender to cease sending packets into the network. This mechanism is known as the back-pressure mechanism, which is illustrated in Figure 2.3. Two packets are destined for the same link. One is forwarded and link 1 is blocked. This leads to a blocking of link 2, which further leads to blockages of links 3, then 4, and so on, even though the packets are not en route to the same destination. One of the level 3 packets may be destined for link 5, but is affected by the back-pressure nonetheless.

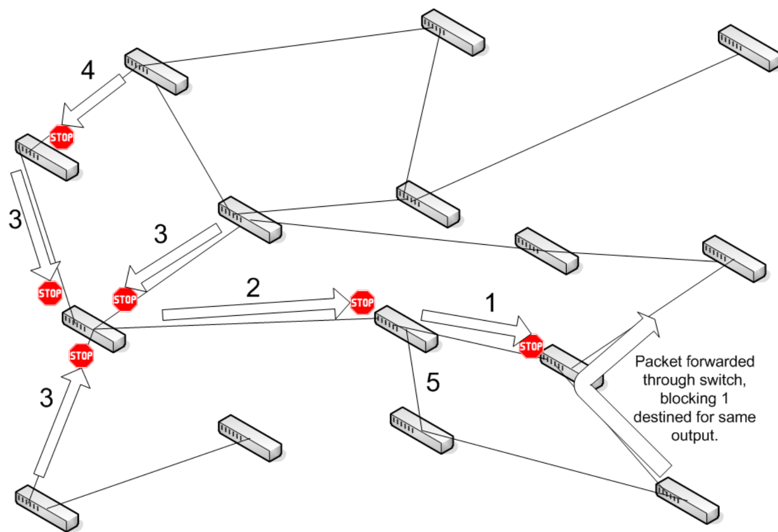


Figure 2.3: A figure illustrating the back-pressure mechanism in VCT networks

Hosts

Hosts are computing nodes producing and consuming network packets. There is also a small amount of traffic generated by the switches in order to perform flow control, but this is just in order to ensure that the nodes do not overflow parts of the network with too much traffic.

At the transport layer, between nodes, communication takes place between queue pairs (QP). A QP consists of both a send work queue and a receive work queue. With these QPs InfiniBand supports five transport service types: reliable or unreliable connection, datagram, raw IPV6 datagrams and raw EtherType datagrams. The architecture supports up to 2^{64} QPs per channel adapter.

Subnet Managers

The subnet manager is the control-center of a subnet. The main purpose of the subnet manager is to discover the network topology, perform routing and construct forwarding tables for every switch in the network. The IBA specification [9] leaves many of the implementation specific aspects of the subnet manager unspecified. In [12] the authors present a subnet management mechanism for use in IBA, a mechanism including among others, methods for topology discovery and updating the switches' forwarding tables.

Routers

Routers are used to link several subnets together. In addition to its LID, every network component has a unique Global Identification (GID) which is used when routing between subnets. It is the responsibility of the router to translate the GID into a LID in one of its local networks, or forward the packet to the appropriate router if the GID is not present in the local subnet.

2.3 Quality of Service in Internet

The Internet is one of the largest and most diverse packet switched networks in existence today. It spans the entire globe with millions of connected computers creating a huge international network for transporting information and data all over the world. Internet is a network layer concept built upon a large diversity of link layer technologies. Many of the link layer technologies are based on the switching principles presented in section 2.1.1. The QoS concepts for Internet described further down must therefore co-operate with the QoS mechanisms, if any, of the link layer technology.

The Internet structure is based on a best-effort model where as little information and as few guarantees as possible should be present in the core of the network. The forwarding of packets through the Internet is done using the Internet Protocol (IP), a best-effort protocol giving no guarantees as to packet delivery. The end nodes should have all the intelligence necessary to ensure safe delivery of data across the Internet. This means that the basic operation of an Internet relay/network node is to forward received packets if possible, without guaranteeing that the packet will ever reach its destination. Because of this lack of guarantee from the network, it is up to the transport layer protocol residing at the end nodes to give the illusion of a better Quality of Service than Internet really provides. There exist today basically two transport protocols for ordinary computer traffic: the Unreliable Datagram Protocol (UDP) and the Transport Control Protocol (TCP). UDP is a connection-less protocol which gives no further guarantees than the Internet protocol itself, packets may get lost or reordered along the way. TCP, on the other hand, is a connection-oriented protocol able to guarantee packet delivery in correct order end-to-end using timeouts and retransmissions as long as there is an end to end connection. It is, however, not able to give any guarantees as to bandwidth or latency through the network [31].

2.3.1 End-to-End Quality of Service

Achieving the ability to give Quality of Service guarantees in the Internet is a much sought-after goal [68][74]. An increasing number of companies use the Internet as part of their daily business transactions, IP telephony and video on demand are rapidly growing fields and online gaming also shows a rapid increase. Each of these applications require a particular level of service from the Internet with respect to packet loss, bandwidth and latency/jitter. This is complicated by the fact that Internet is basically a best-effort network, with no guarantees as to how packets will be treated. One domain may offer the packets an extremely good service while another domain may drop many packets and impose huge delays on the packets which make it through. The Quality of Service perceived end-to-end is not much better than the worst Quality of Service received in any of the domains along the path. This makes it almost impossible to give any sort of guarantees as to how data traffic will be serviced in the Internet.

Attempts have been made to give QoS guarantees in Internet. IntServ and DiffServ are the two main QoS concepts for use in Internet put forward by the IETF (Internet Engineering Task Force). They represent two widely different viewpoints of QoS in switched networks. Each of these two concepts are presented below.

IntServ

In conventional packet switched networks, several flows may share the same physical link and buffer space. This makes it difficult to service different flows individually; there exists no mechanisms for treating one flow differently from another. In line- or circuit-switched networks this is not a problem since every flow has its own dedicated line through the network which is allocated a certain amount of bandwidth. Furthermore, no packet loss can be guaranteed as long as the transmitting end of the line does not exceed its allocated bandwidth, and the flow receives Quality of Service.

An intuitive way of providing Quality of Service in a packet-switched network would be to emulate circuit-switched networking. This is the basis for the Integrated Services (IntServ) [17] Quality of Service paradigm. The idea behind IntServ is to emulate circuit-switched networks by reserving a certain amount of the resources in every switch and router for each flow from source to destination [17][71]. Such a reservation may be a certain priority for the packets if the switch or router supports priority scheduling, a certain amount of minimum bandwidth on the outgoing link, no packet dropping, and so forth. A reservation message is sent through the network along the path the flow will follow to set up the reservations in the network. A specific protocol has been developed for this purpose, the Resource Reservation Protocol (RSVP) [71]. RSVP is a receiver oriented reservation protocol in that it is the receiver that initiates the reservation of resources in the network. First, a PATH message is sent through the network to potential receivers, typically using multicast. The receiver then responds with a reservation message (RESV) which reserves resources on its way to the source of the PATH message. Every network node along the path of the reservation message through the network reads the reservation message and tries to reserve the resources specified. If the reservation in a node is unsuccessful, a message is sent back to the receiver indicating that the reservation failed, terminating the reservation process. If the reservation is successful, the reservation message is forwarded to the next hop on the path and the process is repeated. The reservations are associated with a timeout function. This makes RSVP a soft-state reservation protocol forcing the receiver to periodically update its reservations in the network. In case the sender should stop sending packets the reservation will eventually time out and be deleted from the network. Upon completion of a transaction using a reservation, a tear-down message is sent through the network to free the reserved resources.

In order to classify what kind of behaviour each packet should receive on entering a router, the router has to determine to which flow the current packet belongs. This is found through a combination of several fields in the packet header like source address, destination address, transport protocol, and port number. This

classification has to be done at every router/switch to provide differential service. The classification is a time-consuming process imposing unnecessary delay on the network traffic. The move towards IP version 6 (IPv6) [3] will avoid this problem since the packet header has a specific flow label for this purpose.

A scheme like this requires that every node in the network has information about every flow passing through it. As the network size increases the number of flows through each node in the network increase, and so does the amount of stored state information. Additionally there is an overhead associated with reserving and tearing down the resource reservations. As discussed in Section 2.1.3 this breaks one of the properties of a scalable network: the amount of information stored in the network should be independent of the amount of traffic (number of flows) in the network.

The result is a Quality of Service scheme which is able to give very specific guarantees to each individual flow, but does not scale very well.

DiffServ

Realising that per flow state information in every node in the core of the network does not scale very well, Differentiated Services (DiffServ) [14] was developed in an attempt to push the complexity to the extremities of the network, much in accordance with how the Internet is designed [14][71]. Instead of reserving resources for each flow in every core node in the network, the traffic is divided into a certain number of classes: Expedited Forwarding (EF), Assured Forwarding (AF) and Best Effort (BE). The core nodes are configured to provide different service to the different classes. Each class has several attributes associated with Quality of Service defining the Per Hop Behavior (PHB) of that class. The PHB defines the treatment of the packets associated with it. This includes giving the packets a certain priority in the routers and switches, allowing them a certain portion of the link bandwidth and specifying to what degree packets may be dropped.

To facilitate the different PHBs the switches and routers in the network have to support certain mechanisms for differentiating packet treatment. For instance the service classes associated with the different PHBs should have separate queues. This gives the router/switch the opportunity to treat the service classes differently by giving the packet queues different priorities.

A mechanism for achieving the desired variation in treatment of the packet queues is Weighted Fair Queuing (WFQ). WFQ is a packet scheduling technique which assigns portions of the outgoing link's bandwidth to the different packet queues relative to their priorities. The EF class provides low delay and low jitter by using a queue with high priority. AF is divided into several classes with different properties, offering higher-priority classes lower drop rates and higher queue priorities than ordinary best effort traffic.

Instead of having information about every flow, the core nodes of the network are required only to hold information about a fixed number of service classes regardless of the number of flows in the network. The packet differentiation, deciding what service the different packet should receive on entering a network node, is done through reading the the DSCP (Differentiated Service CodePoint) field in the packet header indicating which service class the packet belongs to. The DSCP field is set either by the sender, assuming that the sender has information about the different service classes available in the network and has permission to use them, or by the ingress router to the network if the packet comes from another network.

There is a clear distinction, as can be seen in Figure 2.4, between boundary nodes at the perimeter of the clouds, and interior nodes in the center of the clouds in a DiffServ domain. The boundary nodes in the DiffServ domain classifies and marks packets with the appropriate DSCP value. This is a function that requires time and computational resources which are not available in the interior nodes. The task of the interior nodes is to forward packets and treat them based on the DSCP field in the packet header.

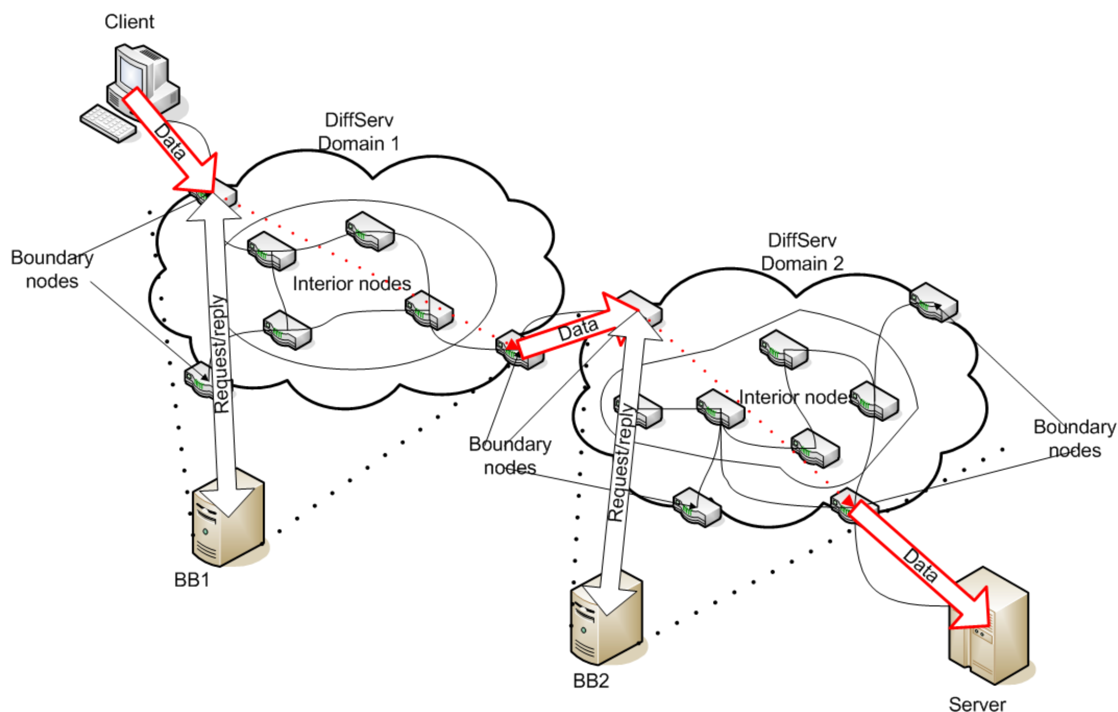


Figure 2.4: Two DiffServ domains, each controlled by a BB. Clients and servers connect to the boundary nodes, data is forwarded efficiently in the interior.

In order for an application to utilise the different service classes present in a DiffServ domain, the sender must have a Service Level Agreement (SLA) with the domain in question. This agreement specifies what kind of service the packets from the specific sender should receive and is used by the ingress router to mark the packets from that sender correctly. The SLA information may be located at a Bandwidth Broker (BB), a host in the DiffServ domain with information about the service level agreements and the current traffic through the domain. The BB is not a necessary part of a DiffServ domain, but it may often be included to function as a admission control mechanism and a resource controller ensuring that a party with which it has a SLA does not send more traffic than specified. Figure 2.4 illustrates the integration of a BB into a DiffServ domain. The figure shows how the boundary nodes communicate with the BB when it receives a QoS requests. When the client sends traffic to the boundary router, the boundary router communicates with the BB to ascertain whether the client has a SLA with the DiffServ domain and whether the traffic may be admitted. The boundary router receives the response with the appropriate service class and marks the packet accordingly. The packets are then forwarded through the interior nodes based on the destination and the DSCP field. Upon entering a new DiffServ domain the boundary router of this new domain must perform a similar communication with its BB before forwarding the traffic.

This scheme is not able to give any per flow guarantees. The guarantees given by the different service classes are relative to the other service classes. This means that a service class is not able to give an absolute guarantee as to latency, jitter, and so forth, it can only guarantee a service no worse than that of the service classes with lower priority. This is only partly true. If the EF class has much traffic and the AF class has little, it might be conceivable that the traffic in the AF class receives better QoS, e.g. drops fewer packets, than the traffic in the EF class.

Having too much traffic in the EF class will degrade both the performance of every flow in that class and every service class below. Entering too much traffic in a lower service class will not degrade the service perceived by the flows in the higher classes to the same extent. This scheme achieves high scalability at the cost of no flow level control and seems to be the only viable option with regard to achieving Quality of Service in large-scale networks.

The deployment of DiffServ and IntServ, possibly in some combination, might be able to correct the lack of QoS in Internet if it is ever possible to unify the QoS mechanisms in Internet by implementing a global Quality of Service scheme [68]. The combination of DiffServ and IntServ may be achieved by using DiffServ in the core of the Internet, the backbone, where there are large amounts of traffic requiring high-speed mechanisms. IntServ may then be used in the endpoint networks

offering precise control over local traffic.

2.3.2 Heterogeneous Networks

One of the main reasons for the lack of guarantees from the network layer in Internet is the diversity of the hardware that makes up the network, its heterogeneity. Internet is built up from many autonomous domains which are created and managed by separate organisations [68]. The rest of the world may not necessarily have any information about the hardware used in a particular domain; it could be Ethernet, Token Ring, ATM or others, each with its own properties as to latency characteristics and packet dropping. It is therefore not safe to assume anything about the kind of service that the traffic through any arbitrary domain will receive.

Many of the link layer technologies that make up the Internet provide different QoS mechanisms, if they provide any at all. Supposing a flow has an end-to-end Quality of Service request, that request needs to be mapped to every QoS mechanism present in the link layer technologies traversed by the flow. This requires vertical integration between the high-level and low-level QoS mechanisms [30], but many low-level QoS mechanisms are not associable with the high-level IETF QoS concepts such as IntServ and DiffServ.

Additionally the various QoS mechanisms must be able to inter-operate in such a way that the different networks using different link layer technologies may provide similar QoS. This is known as horizontal integration [30]. Both types of integration are necessary to be able to provide a unified Quality of Service throughout the Internet.

The IETF has developed a method for seamless integration of heterogeneous networks, i.e. a method for horizontal integration. This approach consists of a protocol called Generalized Multiprotocol Label Switching (GMPLS) [34] and is described below.

2.3.3 MPLS/GMPLS

GMPLS is a generalisation of Multiprotocol Label Switching (MPLS) [69], a label switching protocol developed for use in packet switched networks to provide a common forwarding method for ATM, Frame Relay and IP [34]. It allows the use of a uniform protocol for data forwarding in a larger variety of link layers, technologies based on time domain multiplexing (TDM), Lambda switching (LSC) and Fiber Switching (FSC) in addition to ordinary packet switching [34]. As with MPLS, GMPLS provides a forwarding label which is independent of the packet's network layer header and any specific routing function. Depending on the link layer technology this label may be a number, a specific wave-length, a time slot, or information encoded in another way which is easily accessible for the switches,

making the switching action as efficient as possible. For example, in an optical network the labels are light-bursts at specific wave-lengths which may be easily switched by optical switches. When a switch/router receives a packet with a specific label, it uses the packet's label and input port to determine the output port and the label to be used on that hop, much as in MPLS [69].

When a new flow is initiated a label switched path (LSP) through the network is set up and the flow is assigned a label for use in the first network. When the flow reaches a new network, the flow is assigned a new label for the new network of the type that network supports. Several flows destined for the same destination in the local network may be bundled and assigned the same label, reducing the need for label information in the switches. GMPLS supports tunneling of packets through another network technology (at a higher hierarchical level) with a larger multiplexing capability (the capacity to aggregate more LSPs on one link), so that the packet emerges another place in the original network technology by providing for hierarchical LSPs[11, 10]. A new label is then added to the packets entering the higher hierarchical level, aggregating several LSPs together. The label is removed as the packet emerges to the lower hierarchical level, demultiplexing the LSP[10].

Setting up the paths and assigning labels is done by a special label distribution protocol which operates together with the routing algorithm[11].

GMPLS leads to efficient transportation of for example IP packets over various link layer technologies, and it also provides for service differentiation in the process of setting up the LSPs.

2.3.4 Quality of Service in Switched Networks

IntServ and DiffServ are two high level IETF QoS concepts. They focus on how traffic may be classified and at which resolution the Quality of Service should be provided. They do not, however, specify in what way the Quality of Service should be realised in the lower layers such as the link layer. The link layer requires specific mechanisms for treating packets differently based on some sort of classification, either per flow (IntServ) or per class (DiffServ), for example WFQ described above. This has led to the development and implementation of several QoS mechanisms which vary greatly in the way they provide quality of service. As noted in Section 2.3.2 the parameters of the different QoS mechanisms are not easily mapped from one mechanism to another. Little attention has been given to this vertical integration of QoS concepts.

InfiniBand is an example of a VCT network technology with a specific QoS implementation. Although it may not inter-operate well with other link layer QoS mechanisms, the QoS mechanisms implemented in IBA are designed for cooperation with higher level QoS concepts. The QoS mechanisms present in IBA will be presented in chapter 3.2.

2.4 Admission Control

Admission control is the act of restricting the admittance of a new flow into a network when the acceptance of the new flow would cause the network to not be able to satisfy the service commitments it has already undertaken [35].

Regardless of the complexity of the QoS mechanism, the QoS received from a network depends upon the amount of traffic to receive that particular QoS. In both the IntServ and DiffServ QoS paradigm Quality of Service is dependent on the traffic of that "service level". If a flow with reserved resources in IntServ was to increase its sending rate, the service given each individual packet will be degraded. Packets will be delayed longer in each switch and the switches' buffer capacity may be insufficient for the amount of traffic, forcing further delays or packet dropping. This situation is avoided by having the switches deny reservation requests if there is insufficient resources to accommodate them. The client wishing to increase its flow bandwidth is hindered, avoiding too much traffic in the network. The mechanism in operation here is known as admission control. DiffServ on the other hand does not have such an admission control mechanism, with the exception of the optional BB. If any client increases the packet rate of a flow this will affect the quality of service received by the other flows in that service class and the service classes with lower priorities. The addition of the optional BB gives DiffServ the ability to perform a general form of admission control.

As more and more SAN technologies arrive with their own QoS mechanisms it is natural to also extend the concept of admission control down to the link layer. Neither DiffServ nor IntServ specify exactly how QoS and admission control is to operate on the link layer. If the network technology offers these mechanisms they may be used in a DiffServ or IntServ context. The application of both QoS and admission control at the link level will provide a good framework on top of which to build end-to-end large-scale Quality of Service schemes such as IntServ and DiffServ in the Internet.

A network usually employs an admission control algorithm to perform the admission control. Whenever a node has a flow it wishes to start sending through the network, the admission control algorithm activates and checks to see if admitting the new flow would reduce the Quality of Service offered to the flows already admitted to an unacceptable level. There are a number of ways to perform admission control, some of the differences being in which part of the network the admission control decision is being made, according to what criteria the decision is being made and so on. Chapter 5 will present and describe different admission control algorithms that may be applied to a VCT network.

The following chapter presents the QoS mechanisms present in IBA. Furthermore it discusses some aspects of admission control related to VCT and the de-

scribed QoS mechanisms.

Chapter 3

VCT Quality of Service

Users are typically greedy. They post requests for data and try to gain as much bandwidth from the network as they are able to. It is therefore necessary to have some sort of QoS mechanism combined with Admission Control (AC). There exists several interconnection network technologies for use in SANs, LANs and Cluster Networks, many of which use VCT as the switching technique. As stated above, InfiniBand is such a communication technology which might be used as a interconnection network technology for server clusters, it utilises VCT switching and contains support for Quality of Service at the lower network levels, e.i. at the link level. What is missing from the technology and the many other VCT switched technologies with regards to Quality of Service is a reliable admission control scheme.

This chapter first presents an overview of the previous work done in the field of Quality of Service and admission control in VCT networks. The chapter is continued by presenting the various Quality of Service mechanisms in InfiniBand, describing the concept of admission control and concludes with explaining why it is difficult to achieve predictable transfers in VCT networks.

3.1 Related work

This section will present an overview of the work done with regards to Quality of Service and admission control in System Area Networks.

With the emergence of SAN/LAN technologies such as InfiniBand [9], Myrinet [16], Autonet [28], Tnet [33] and Gigabit Ethernet [58] some work has been done in the field of Quality of Service and to some extent admission control in these types of networks.

Switched Ethernet is not a SAN technology as such, but rather a LAN technology. The switches in Switched Ethernet may be equipped with a priority mech-

anism to support a certain degree of Quality of Service differentiation of network traffic. This mechanism includes several queues and priority tagging of packets to achieve traffic differentiation as is specified in IEEE 802.1p [37]. For Switched Ethernet several bodies of work have been presented that analyse the delay characteristics of the technology. In [36] the authors concentrate on the latency characteristics of a Switched Ethernet network, whilst in [62] and [37] the authors focus on the deterministic properties displayed by the priority mechanisms present in such networks.

Switcherland [26] is a switching technology for use in SANs. As many other technologies it is based on point-to-point links interconnected by VCT switches. The technology relies on a fixed packet size and uses separate queues in the switches to provide different service to CBR and VBR traffic in combination with rate-based flow-control for CBR traffic and credit-based flow-control for VBR traffic. The technology is optimised for low latency by building the switch and managing queues in such a way as to minimise the maximum packet latency in each switch. Additional speed gains are achieved by assuming that nodes are well behaved, avoiding the need for admission control.

In relation to the InfiniBand Architecture several bodies of work have been presented suggesting methods of utilising the Quality of Service mechanisms supported by this technology. Some of the articles presented below include some sort of admission control in their systems, but as we shall see, these methods of admission control are not necessarily applicable to real-life networks.

In [46] Pelissier gives a presentation of the different Quality of Service mechanisms present in the InfiniBand Architecture and shows how these may be used to enable support for DiffServ over IBA. He divides traffic into four classes, one supporting time sensitive traffic which is mapped to a virtual lane in the high-priority table. The rest of the traffic is mapped to virtual lanes in the low priority table, thus giving the best service to the time sensitive traffic. Pelissier does not specify the way in which the different virtual lanes should be weighted in order to achieve the correct differentiation between the service levels.

The work is carried further by Alfaro et al. in [6] and [7] where the authors in [6] define a method for calculating the arbitration tables for the low priority virtual lanes based on the bandwidth requirements of the traffic assigned to the service levels mapped to the different virtual lanes. In [7] the authors include time sensitive traffic into the scheme from [6] . The time sensitive traffic is assigned to a virtual lane in the high-priority table just as in [46] and the worst case latency for this high-priority time sensitive traffic is calculated for several switch architectures. The authors conclude by performing simulations showing that all time sensitive traffic traverses the network with latencies below the worst-case calculations.

Common for these three contributions is that they imply some sort of admission

control mechanism in the network in order to avoid overloading the network and thus degrading the Quality of Service received by the flows. Furthermore, none of the authors specify what type of admission control is needed or indeed have any detailed description of an admission control algorithm. The authors of [6] and [7] assume the presence of an admission control algorithm modifying the contents of the arbitration tables of every switch affected by the arrival of a new flow. This is a time-consuming process and not compatible with DiffServ. Methods relying on run-time modification of the arbitration tables in the switches are further examined in section 4.2.

In [51] Skeie et. al analyse the effect of a DiffServ inspired Quality of Service concept applied to VCT networks. The network on which they perform the simulations relies on the QoS mechanisms specified by IBA as described in Section 3.2. The authors show that the InfiniBand Quality of Service mechanisms are able to differentiate the traffic with regards to throughput. The authors also state that as long as the network operates below the saturation point, the configuration of the InfiniBand arbitration tables and the limit of high priority is nearly insignificant for the throughput of each flow. On the other hand, when the network reaches and passes its saturation point the throughput of the different service levels is dominated by the Quality of Service mechanisms. Another finding is that low latency and especially low jitter is hard to achieve in back-pressure networks such as InfiniBand. It is shown that the maximum possible latency of a packet grows exponentially as the packets path through the network increases in length, and methods for calculating the maximum latency are given.

A body of work giving a more thorough description of an admission control algorithm to be used in an InfiniBand environment is [73]. Here the authors propose one method for admission control and one method for congestion control. They show how these two methods cooperate in a wormhole-switched InfiniBand network to keep the network traffic at a level at which the Quality of Service characteristics are better than without them. The admission control algorithm is based on per flow bandwidth reservations in every switch/router in the network. This is done through the use of setup and tear-down messages much in the same way as in IntServ [17]. This method requires both per flow signaling and recalculation of the arbitration tables used in the weighted round-robin scheduler and is therefore not very scalable nor in accordance with DiffServ.

To the best of my knowledge there has not been published any previous work regarding admission control in VCT networks and therefore no detailed admission control algorithm for virtual cut through networks has yet been proposed.

After this brief summary of related work, the question remains to be answered: how does one perform admission control efficiently in a virtual cut through network whilst adhering to the DiffServ philosophy? It is apparent that some sort of

admission control is necessary in order to guarantee a certain level of Quality of Service. It is also seems like no previous work has been done regarding admission control in VCT networks.

3.2 Quality of Service in IBA

What follows is a description of the QoS mechanisms available in IBA as they are presented in [9]. There are three mechanisms in InfiniBand for Quality of Service differentiation. First there is the service level to virtual lane mapping table. Second is the virtual lane arbitration tables, and third, the Limit of High-Priority (LHP). All of these mechanisms and how they interrelate is described below.

Service level to virtual lane mapping An InfiniBand packet may be sent at one of 16 Service Levels (SL). On which SL a specific packet is to operate is decided at the sending node, possibly with some communication with the subnet manager. The definition of what type of service a specific SL offers may vary from network to network. The purpose of the SLs becomes clear when we look at the SL to VL mapping table which exist in every switch, Host Channel Adapter (HCA), and router in an InfiniBand network. The goal of these tables is to map a packet's SL to the VL which is to be used on the next hop. As mentioned earlier, each network component may have a different number of VLs, and so the SL to VL mapping table may also vary from switch to switch. Examples of what service is expressed in a given SL might be a minimum bandwidth guarantee, delay guarantee, and so forth. The SL to VL mapping table must set up so that each SL is mapped to the VL which in the current switch has the right priorities/properties in order for it to fulfill the requirements of the service level. The properties of VLs are described in the next section.

Arbitration tables If nothing else was said one would expect that each Virtual Lane would receive an equal amount of bandwidth on the link. One may then raise the question: "So what is the point of virtual lanes?"

The VL serves three basic purposes:

- To be used for solving the deadlock problem
- Increase throughput
- Provide for service differentiation

It is possible to exploit the existence of VLs in a routing algorithm to avoid deadlocks. Recall form Section 2.1.2 that deadlocks occur when packets occupy buffers

when waiting for available buffer capacity in another switch, forming a cyclic dependency. Having several VLs may be utilised by assigning routes between different source/destination pairs to different VLs. If a particular assignment causes cycles in the channel dependency graph, a source/destination pair may be moved to another VL to break the cycle. LASH [64] and Up*/Down* combined with VLs [54] are examples of a routing algorithms using several VLs to avoid deadlocks. A short description of LASH is presented in Section 6.1.4. As to increased throughput, each VL is associated with its own set of send and receive buffers. The blocking of packets in one VL will not affect the packets in the other VLs. This contributes to increase network throughput by allowing more efficient link utilisation.

In our context the role played by the VLs in providing service differentiation is perhaps the most important. The separation of traffic into VLs makes it possible to treat various packet streams in different ways by controlling the service given to each VL. The service received by a VL is controlled by the arbitration tables. Every switch has two arbitration tables, one high-priority table and one low-priority table, with the high-priority table being able to preempt the low priority table. The entries in both the high-priority and low-priority table consist of a VL number and a weight, with a value from 0 to 255, indicating the number of 64 byte-units which may be transmitted from that VL when its turn comes. The decision as to which VL's turn it is may be done in a Round Robin fashion. The tables may have up to 64 such entries which means that the VLs may be listed several times in order to increase the overall weight of the VL.

Limit of High-priority There is a Limit of High-Priority Counter (LHPC) which counts the number of bytes sent from the high-priority VLs between each packet from low-priority VL. It is initsialised to 4096 * LHP. The LHP may have a value between 0 and 255. This sets a limit on how many bytes may be sent from the high-priority table before an entry in the low-priority table has to be serviced. The LHPC prevents the virtual lanes in the low priority table from being starved. As long as $LHPC > 0$ a packet may be sent, even if the remaining value of LHPC is less than the packet size. If $LHP = 255$ the high-priority arbitration table always preempts the low-priority table, there is no guarantee against starvation of the low-priority traffic.

In essence this is a combination of preemptive scheduling and Weighted Fair Queuing, with the high-priority table preempting the low-priority table, and with WFQ used within the tables. On top of this VL 15 always has top priority, it preempts any other VL.

This technique allows us to give quite firm guarantees on some VLs by putting them in the high-priority table without stopping flows in the low priority table and

thus still be able to provide best effort service. A flow may receive relatively high bandwidth by giving the appropriate VL a large weight in the low or high-priority table. In [6] the authors propose a method for computing the weights to be used in the InfiniBand arbitration tables. The authors calculate the relationship between the arbitration table weights and the actual bandwidth on the link corresponding to the weight. The weights may then be distributed among the different VLs based on the priority and amount of traffic that VL is to service. Low delay can be achieved by giving the VL an entry in the high-priority table, and thus giving that VL priority, the delay may then be calculated as described in [51] and in Section 3.3.2.

The mechanisms described here form a DiffServ compliant QoS scheme. There is some complexity in the client/boundary routers required for selecting the appropriate initial SL for the traffic, leaving the core switches free to forward traffic based only on destination and SL much as in DiffServ.

3.3 Admission control in IBA

3.3.1 Is Admission Control Necessary?

The Quality of Service mechanisms of IBA are only able to provide class based Quality of Service at a VL level. All traffic present in a VL will receive exactly the same treatment no matter to which flow it belongs. This means that as long as there is more than one flow per VL the network is unable to give per flow Quality of Service guarantees. To illustrate this consider the following example: suppose we have a link supporting a single VL with the total bandwidth 100 Mbps and flow requests admittance with a throughput requirement of 50 Mbps. Since the requested bandwidth is below the link bandwidth the flow may be admitted without problems. If the second flow requests admittance with the same bandwidth requirement, 50 Mbps, it too may be admitted. The link bandwidth is now fully utilised by the two flows, and both flows receive a throughput as requested from the network. Now, consider what would happen if a flow increases its rate from 50 to 100 Mbps. The network would obviously no longer be able to handle the amount of traffic generated and it would have to restrict the sending of a large number of packets. Since both flows are using the same VL, and since IBA is unable to differentiate traffic within a VL, all of the present flows will receive a degradation of service. The network will block packets from both flows resulting in that none of the flows receive the bandwidth they initially requested. By preventing the second flow from increasing its sending rate, e.g. rejecting its request to send more traffic, the situation above may be avoided and the network can continue providing adequate service to the admitted flows.

A typical system where these sorts of problems are relevant would be a network supporting a multimedia server. Suppose this server contains video files to be streamed to nodes in the network. Each video stream has bandwidth and jitter requirements. That is, the delay between packets in the video stream may not exceed a certain threshold. If this was to happen, the video playing on the node would have to skip video frames and the resulting video would not be a pleasant sight. By employing admission control in such a network one could limit the number of video streams from the server to the different nodes in such a way that as many nodes as possible may request video streams while the quality of each individual stream remains acceptable.

3.3.2 Back-pressure issues

The back-pressure mechanism of VCT switches described in section 2.1.1 may cause very large and variable delays since the blocking of one packet in a switch may lead to blocked packets in other switches in the network. In [51] Skeie et. al present a set of equations to calculate the worst-case latency in the network of a certain number of switches with only one VL per port. These equations show that the maximum latency experienced by packets traversing the network increases exponentially as the number of hops the packet has to traverse increases.

The maximum latency given in [51] is dependent on the amount of ports in the switches and the number of hops the packet traverses. More specifically:

$$P = \sum_{sw=1}^{n} (nLinks - 2)^{sw} \tag{3.1}$$

This calculates the maximum number of packets a specific packet might have to wait for before it may be sent itself. The maximum latency L may then be given as

$$L = T * P \tag{3.2}$$

T is here the transmission speed in cycles.

Using this equation as presented in [51] it should be possible to calculate the worst latency the packets would get on any path through the network. This worst case latency would be the only one the network is able to guarantee 100%. This is similar to the experiments performed in [7] where the authors calculate the delay a packet would experience through several different switch architectures and show through simulations that the network is able to provide a packet latency less than that specified by the worst case calculations. Skeie et. al show in [51] that the average latency experienced by the packets injected into the network is far below the worst-case calculations, but with a large amount of traffic in the network the latency may approach the worst-case scenario.

By their very nature, VCT networks do not drop packets. As the network load increases, more and more packets are held back for a short while in buffers in the switches. If the network is saturated, that is, there is always a packet ready to be sent on any link when the link becomes free, the packet latency will approach the maximum given by the above equation (equation 3.2). As the network load decreases, so does the number of packets in the network at the same time. A smaller amount of packets in the network means that there is smaller probability of having to wait for other packets. This again leads to lower network latency.

It is clear in this case that admission control is necessary to limit the load on the network to such a degree as to be able to guarantee a worst-case network latency with a lower bound than worst-case. This view is supported by the authors of [51] as they conclude by advocating the need for admission control to keep the network load at a level below saturation in order to enhance the latency and jitter characteristics of the network.

The DiffServ nature of the Infiniband Architecture puts certain restrictions on the types of admission control that may be utilised. In order to comply with the DiffServ spirit, the admission control algorithms should not require any signaling or state information in the core of the network. Additionally it should not be necessary to alter information in any switch at the admission of a new flow.

The next chapter discusses admission control more thoroughly and gives criteria by which admission control algorithms may be evaluated.

Chapter 4

Properties of Admission Control

The previous chapter introduced admission control as an important mechanism for providing QoS. In this chapter a more detailed presentation of admission control will be given. Furthermore criteria will be given against which the admission control algorithms to be proposed in the next chapter will be evaluated.

4.1 Characterisation of a good admission control algorithm

Based on the QoS definition presented in the introduction, there are two key goals admission control algorithms have to meet.

Network Throughput One is the ability to provide the bandwidth that the flow is requesting. If a flow is requesting a certain guaranteed bandwidth, the admission control routine should only admit the flow if it can guarantee that the flow will receive the requested bandwidth throughout its lifetime, and that the service experienced by the already admitted flows should not be degraded. If the flow is admitted and does not receive its requested bandwidth, the admission control routine is not strict enough when making its decisions. If, however, every flow that is admitted receives its required bandwidth, but the admission control routine starts rejecting flows while there is still enough available capacity in the network in order to ensure that the requirements are met, the admission control routine is too strict. A good admission control algorithm should in other words utilise as much bandwidth as possible whilst still being able to guarantee the required bandwidth of the individual flows [72].

There are of course modifications to this statement. For some applications it might be absolutely essential that the flows receive the requested bandwidth and one might accept wasting some of the networks bandwidth in order to give absolute

guarantees to such flows. On the other hand the flows might ordinarily be sending at a rate lower than the peak rate which they originally reserved in the network. If these flows accept that they might not get the reserved rate absolutely all the time, the admission control algorithm may accept more flows so as to utilise the network bandwidth to its fullest extent by assuming that not all the flows will send at their reserved rate at the same time. High network utilisation and absolute bandwidth guarantees may not necessarily combine easily and the importance of one over the other may vary according to the type of traffic in the network.

Timeliness The second goal is latency and jitter. A requesting flow might require that the latency its packets experience through the network should never exceed a certain threshold, or the packet inter-arrival time, jitter, should remain within certain bounds. These demands may be made separately or in combination with bandwidth requirements as described above.

The latency experienced by the packets of a flow depends heavily on the characteristics of the network through which the packets traverse. It is clearly impossible to achieve a network latency lower than the time it takes to process the packets in the network switches and forward them across the links, and it is clear that the longer the network path is the larger latency the packets will experience. Since the network latency is so heavily dependent on the topology and on the physical characteristics of the network, it is difficult to achieve a latency guarantee beyond a certain minimum. Whether or not the packets of a flow arrive at the end node within the latency bound is therefore not a good indication of an admission control algorithm's efficiency, unless one takes into account the network topology when calculating the latency bounds. The jitter on the other hand is less dependent on the network topology and is therefore better suited as an efficiency measurement.

We see here again a conflict regarding jitter versus network utilisation. As the network utilisation increases, the traffic in each switch in the network will also increase and the jitter will presumably worsen. The only jitter requirement that can be met absolutely is obtained from calculating the difference between the best case and the worst-case latency in the network [46]. In order to provide good jitter characteristics to a flow, the admission control algorithm must therefore limit the network traffic in such a way as to make the worst-case latency small enough that the difference between the worst-case and best case does not exceed the requirement.

The relationship between bandwidth and latency in the network is not very clear, but as the network load increases there will be an increase in the number of packets in the network and in the chance of a packet having to wait for a longer time. It follows that if every flow in the SL reserves more bandwidth than is actually used all the flows will experience less network latency.

There is no relationship between bandwidth requirements and latency requirements of a flow, a flow may request high bandwidth and ignore latency or *vice versa.* On the other hand the relationship between latency and jitter is well-defined in that low network latency necessarily leads to low jitter since the difference between the maximum and minimum time the packet spends in the network decreases [46].

4.1.1 The Network Utilisation vs. QoS trade-off

As discussed in the previous section there is a trade-off between the Quality of Service received from the network and the utilisation of the network. It would seem impossible to give guarantees beyond a certain point without going too low in the network utilisation. This indicates that it should be possible to get the best Quality of Service technically possible from the network at the expense of network utilisation, a postulate which is evaluated and confirmed in Section 7.5. It is therefore necessary to include the network utilisation as the criteria for measuring the efficiency of admission control algorithms in addition to the goals of latency/jitter and throughput mentioned above.

What follows is a list that summarises the criteria against which the admission control algorithms described in this project will be tested and evaluated.

1. The admission control algorithm should be able to guarantee the requested bandwidth to each flow admitted into the network.

2. The admission control algorithm should be able to satisfy the latency/jitter requirements of the flows (if applicable).

3. The admission control algorithm should be able to differentiate between different service levels with different latency/jitter requirements (if applicable). In other words, two SLs with different latency demands should receive different latency from the network.

4. The network utilisation should be kept as high as possible in relation to the admission control requirements. This means that an admission control scheme with high network utilisation but with slack QoS requirements is not necessarily better than a network with low utilisation, but with strict QoS demands. Any best-effort service levels which will use the remaining available bandwidth and thus push the network utilisation up towards the maximum point must be considered when assessing the admission control algorithm.

5. The admission control algorithm should keep the network below saturation in order to guarantee the requested bandwidth to all service levels. Recall

from [51] that in a saturated network, the bandwidth distribution between service levels is controlled by the QoS mechanism in InfiniBand, not by the offered load.

Item 2 and 3 will be difficult to test on an admission control scheme that bases its decisions entirely on available bandwidth, such as the one described in section 5.1. It should however be possible to configure the different algorithms so that they yield approximately the same network utilisation and compare the latency and jitter characteristics of different algorithms.

It is against these criteria that the admission control algorithms proposed in this thesis will be evaluated in a VCT environment, taking into account the characteristics of each individual algorithm as to the type of guarantees it can give in the framework described above.

4.2 Arbitration Tables and Admission Control

As touched upon in Section 3.1 regarding related work, some authors have introduced papers relying on admission control based on modifications of the arbitration tables. This section will discuss this as a method of admission control and point out some strengths and weaknesses of this approach.

The values present in the arbitration tables represent the relative difference in priority between the different VLs. A certain value indicates that the VL in question should receive a certain part of the total bandwidth available to the link. If the link in question is unsaturated, that is, there is available bandwidth, traffic on every VL will be serviced as it arrives in the switch. This means that as long as there is available capacity on the link the VLs may receive a larger part of the link's bandwidth than specified in the arbitration tables, the weight values in the arbitration tables have no real significance [51]. If on the other hand the network is saturated, that is, the amount of traffic pushed on to the link approaches the link bandwidth, the limit of high-priority and arbitration tables takes effect and the traffic load is distributed as they dictate. This means that for a given flow with a certain bandwidth a VL is only able to guarantee specific bandwidth to a limited number of flows with the same bandwidth, calculated by dividing the bandwidth available to a VL, $BW_{VirtualLane}$, calculated later in equation 5.6 in section 5.1.3, by the flow's bandwidth, BW_{Flow}.

$$NumberOfFlows = \frac{BW_{VirtualLane}}{BW_{flow}} \tag{4.1}$$

This is a property that should be utilised in admission control. When receiving a request for admission of a flow it could be possible to modify the weights in the

arbitration tables of switches through which the flow should pass in such a way as to increase the number of flows the VL is able to support. These modifications could be done for example by leaving the total of the assigned weights constant and increasing the weight of the VL to which the flow will be mapped, and decreasing the weight of a VL to which a low priority SL is mapped. This scheme will guarantee available bandwidth for high priority flows at the cost of the bandwidth available to the low priority flows. The downside is that upon acceptance of a flow every switch of a flow's path will have to take an active part in the admission control process in order to modify its arbitration tables. Modification of the arbitration tables leads to the reservation of resources in each switch, exactly as is done in IntServ. This is unacceptable in a DiffServ context which requires that all complexity is limited to the boundary nodes.

An alternative technique assumes that a fully reserved switch would contain the maximum possible weight values in every entry in the arbitration tables. One could then use this information to calculate the bandwidth corresponding to a weight of "1" in the high-priority and low priority table. A flow with a specific bandwidth would then require a certain weight added to the VL to which it is mapped. As long as there is available space in the arbitration tables one could continue admitting flows and increase the weight of the appropriate VL accordingly. A flow is rejected when it is no longer possible to assign the necessary weight to the appropriate VL. This method is able to guarantee bandwidth to every flow on every SL and is much simpler than the one first mentioned.

Both these methods are able to give bandwidth guarantees and are flexible in the sense that they can recalculate the weights of the different arbitration tables as the flows' demands varies. If there are many flow requests on a specific SL/VL the VL will be assigned appropriate weights as long as there is room in the arbitration table.

In [6] and [7] the authors assume an admission control algorithm which operates through modifying the arbitration tables in the switches, possibly in the way described above. The large and relevant downside of this method is that the arbitration tables of every switch along a flow's path has to be modified every time a flow is admitted or torn down. This leads to a considerable overhead when setting up and tearing down flows and is not associable with DiffServ. This means that although the methods have some very pleasant properties in their ability to give bandwidth guarantees, they are virtually useless in large-scale networks with many flow events. It is therefore necessary to find alternative methods of admission control which adhere to the DiffServ paradigm by not demanding runtime reconfigurations of the core elements of the network, the switches and routers.

Part II

Application of Admission Control

Chapter 5

Admission Control Algorithms

In this chapter several admission control algorithms for use in VCT networks will be proposed and described. The range of algorithms represents widely differing approaches to admission control, and when evaluated they should provide insights into what type of admission control is the most effective and what type of guarantees that may be achieved. As there is no previous work regarding admission control and VCT, all the algorithms presented in this chapter are adapted from other networking environments, such as Internet.

5.1 Switch Level Admission Control, Link-by-Link

5.1.1 Parameter Based

The simplest and most intuitive form of admission control would be to have every part of the network through which the flow is to pass decide whether it can support the requesting flow or not. In this way every part of the network is able to say with a degree of certainty that it is able to accept the flow or not. One such method is introduced in [35]. If every part of the network which is affected by the requesting flow indicate that they are able to handle it, it is reasonable to assume that the network as a whole is able to accept the flow.

In an Infiniband network central elements of the network are the links and switches. The links are just unintelligent wires running between the switches, and since every link is connected to a switch (or an end node), the switches have complete control over the traffic on every link in the network. By letting the requesting node and every switch on the flows path from the node to the sender check to see whether the outgoing link is able to support the requesting flow, one can achieve this form of admission control. This scheme operates in accordance

with IntServ [17] where it is up to the core of the network to perform the admission control decisions.

There are several ways in which the switches can decide whether the outgoing link can handle the flow or not. The absolutely safest way to perform the admission control, a method which guarantees that the link is able to accept the flow, is one described as the "simple sum" algorithm in [35]. In this algorithm the sender includes in the reservation message the peak rate of the requesting flow. The peak rate of a flow is the maximum rate at which a node will be sending packets belonging to the flow. Upon receiving the flow admittance request the switch adds the peak rate to the peak rate of the already admitted flows and performs a check to see whether the result is greater than the bandwidth of the link. If this sum does not exceed the link bandwidth, the flow is accepted and the peak rate is added to the sum of the peak rates of the admitted flows. The reservation is then forwarded to the next switch on the path for it to perform the same test. If, however, the sum exceeds the link bandwidth, the flow must be rejected and a message sent back to the sender along the path by which it arrived. The switches that have already admitted the flow must remove the flows peak rate from its sum of admitted peak rates since the flow is no longer admitted in the network. If p is the peak rate of the requesting flow, s the sum of the admitted peak rates and bw the link bandwidth, the requesting flow will be admitted if the following check succeeds [35]:

$$p + s < bw \tag{5.1}$$

In Infiniband where SLs and VLs play an important role in the assurance of Quality of Service it might be desirable to split the system up from the link level admission control as described here to VL level admission control. In VL level admission control one divides the link's bandwidth among the VLs according to whichever rules one wishes. Since different SLs may have different bandwidth and latency requirements it is natural to introduce some sort of differentiation into equation 5.1 by exchanging bw for AC_{bw}, AC_{bw} being the available bandwidth for a VL. The differentiation may for example be done according to the offered load by giving each VL a portion of the link bandwidth relative to the traffic load as described in section 5.1.3.

$$p + s < AC_{bw} \tag{5.2}$$

The method described here is a so-called parameter based approach where the admission control decision is made using parameters calculated *a priori*, the link capacity and the peak bandwidth requirements of the requesting flows. In order to give bandwidth guarantees, it is assumed that the sending rate of the flows will be the same as the peak rate. This is often not the case, flows often have variable send rate with the peak rate being the absolute highest sending rate. Parameter

based systems are known to give better QoS guarantees at the expense of network utilisation since the switches tend to reserve too much bandwidth for each flow.

Switched networks have a property that complicates this scheme a little. A network is unable to utilise the maximum of the theoretical bandwidth of a link in the network. For a typical network link utilisation might lie around 60% [38] of the link capacity in a switch using input queuing and processing only one packet per packet slot. This is due to inefficiencies in the switches' forwarding mechanisms, such as head of line blocking and the fact that the packets spend a small amount of time in the switch when being forwarded. In VCT networks the flow control mechanism is also responsible for some of the reduction of the actual link utilisation.

Head-of-line blocking is a phenomenon which occurs in certain switch architectures as the network load increases. It is a consequence of the way buffers are handled and where in the switch they are placed. Consider a switch with a packet queue at each input link where the packets wait before they cross the switching fabric to the output port. If the first packet is destined for an output port which is busy the packet must wait until the output port is available. Using FIFO (first in, first out) queuing the packets that are further back in the queue will also be forced to wait, even though they are destined for an available output port. The head of the line blocks the rest of the queue. There exists switch architectures that avoid the problem of head-of-line blocking, shared queuing [19] and virtual output queuing [19] among others, but they require complex buffer management schemes.

The reduced actual link utilisation indicates that one should reduce the link capacity in the above calculations to the amount indicated as the actual link utilisation. This information can be obtained by gathering throughput statistics from the network and calculating the link utilisation from them.

5.1.2 Measurement based

Measurement based admission control schemes are known to give better network utillisation than parameter based admission control schemes [72], but they may lead to the switches overbooking the network capacity, thus decreasing the ability to give QoS guarantees. A variation of the parameter based link-by-link scheme is the "measured sum"-approach, also mentioned in [35]. Instead of just accumulating the peak bandwidth of the admitted flows which results in a guaranteed service, the measured sum approach measures the load of the network and uses this to make admission control decisions. While the simple sum approach ensures that every flow may send at its peak rate continuously without overloading the network, the measured sum approach only ensures that the current rate at which the nodes are sending does not exceed the link capacity. It works as follows: every

switch keeps track of its throughput rate, and when a flow requests admittance the throughput rate is added to the peak rate of the admitting flow. If the result is less than the link capacity the flow is admitted. The link capacity should also here be modified by its percentage of actual utilisation. Additionally the modification must take into account the fact that the load might increase anytime if several flows decide to send at their peak rate instead of mean rate at the same time.

The parameter based link-by-link scheme is able to guarantee that a flow may send at its packet rate without overflowing the network and disturbing the already admitted flows, it can give absolute QoS guarantees. The price of the absolute guarantees is network throughput. The method will tend to minimise throughput in order to make hard guarantees, giving a good indication of how low delay/jitter it is possible to achieve in this kind of network. The results achieved with this method may therefore act as a reference against which the other algorithms to be described here may be compared.

5.1.3 AC Differentiation for Bandwidth Requirements

The link-by-link scheme operates by comparing the link capacity to the aggregate bandwidth of the admitted traffic. Since Infiniband allows the existence of up to 16 VLs on each link supporting traffic of different priority, the admission control should be done on a VL by VL basis rather than on a link-by-link basis. The relative portion of the links bandwidth available to each VL is specified by the weights in the arbitration tables. A VL with a large weight in an arbitration table will have a larger amount of the link bandwidth than a VL with a small weight in the same table. Similarly a VL with an entry in the high-priority arbitration table will have a larger proportion of the link bandwidth than a VL in the low-priority arbitration table with equal weight. Since the traffic in the different VLs may have a different priority, it is desirable that the admission control routine treats the different VLs in a manner related to their priority. Traffic with high priority typically use VLs with large weights in the low or high-priority arbitration tables, but it is not always the case. Traffic on a VL with a lower weight than other VLs in the same arbitration table may receive better service than the other VLs if the amount of traffic using the VL is restricted appropriately, allowing it to support traffic of higher priority.

As mentioned earlier, the link-by-link (LBL) admission control scheme bases its decision on the available link bandwidth. Therefore, when introducing several VLs supporting traffic with different priorities, LBL requires a method for relating the VL weight and traffic priority to the link bandwidth parameter, the VL capacity, used in the admission control decisions. In other words, LBL needs a way of calculating the amount of traffic that may be admitted on each VL based on its weight and priority of the traffic on it, a way of differentiating the admission control

parameter of the different VLs.

There are at least three possible ways of having the admission control routine differentiate the VLs based on bandwidth. These are

1. Statically: having the admission control routine admit and equal/predefined amount of traffic on each VL.

2. Available bandwidth for each VL (AB): tailoring the admitted traffic to the bandwidth available to the appropriate VL based on the QoS mechanisms LHP and the arbitration tables.

3. Offered load (OL): admitting traffic according to the expected load on the VL, admitting much traffic on VLs with higher load and less traffic on VLs with less load.

The first item is trivial, the portion of link bandwidth to be used in the admission control decision may be set to any value, possibly an equal share of the link bandwidth, without any guarantees as to how it will work out. The other two methods, one based on calculating the available bandwidth for each VL based on the weights in the arbitration table and the LHP and the other relating the offered load to the admission control parameter, are described in the following sections. One major difference of these two methods is that OL requires a priori knowledge of the load to the offered on the different VLs while the AB method does not.

The third method requires that we have a one-to-one or one-to-many relationship between SLs and VLs. Packets operate on SLs, and the way in which we can know the load a priori is to specify the amount of traffic allowed to use certain SLs, e.g., each node is allowed to send 10% of its traffic on a certain SL. The method requires that there is a clear relationship between the expected SL load and the VL load. This is most easilly done if each SL is mapped to exactly one VL. If a single SL is mapped to several VLs, as is the case with LASH (Section 6.1.4), the routing algorithm used in the simulations, we have to give each VL an equal share of the total bandwidth available to that VL. If we have a many-to-many mapping between SLs and VLs, things become much more complicated and depend heavily on whether the different VLs have the same weights and so forth. Many-to-one mapping, or aggregating, may also present a problem, but as long as every switch offers an equal number of VLs the problem may be avoided since aggregating of SL's is avoidable. In this project we limit ourselves to use only a one-to-one or one-to-many mapping depending on the routing algorithm (see section 6.2) and therefore avoid these problems.

Differentiation based on each VL's available bandwidth

Recall that Infiniband already supplies the mechanisms necessary to provide Quality of Service. These mechanisms are the SL to VL mapping, the VL arbitration tables and the limit of high-priority. Together these mechanisms form a framework for guaranteeing a certain minimum bandwidth to each VL existing in the network. Using the equations to be presented in Section 5.1.3 one could calculate the bandwidth available to each VL based on whether the VL is in a high- or low-priority arbitration table, the LHP and the VL's weight. It can be seen from these equations that as the weight of a VL increases so will the portion of bandwidth allocated to that VL increase. This means that VLs with higher weights and/or which reside in the high-priority table will have more available bandwidth than VLs with less weight, which is exactly what is supposed happen. Now, if one were to use the available bandwidth of a VL as the basis for admission control, one would end up admitting more traffic on VLs with higher weight since they have more available bandwidth. If AC_{bw} is the parameter given LBL as the bandwidth available to a VL as described above, i.e. the amount of traffic LBL may admit for a VL, and $BW_{VirualLane}$ is the bandwidth available to a VL in saturation (an equation for $BW_{VirtualLane}$ is given below), the calculation of AC_{bw} is trivial:

$$AC_{bw} = BW_{VirtualLane} \tag{5.3}$$

This forces the network into the behaviour displayed when it is saturated, where the traffic is dictated by the QoS mechanisms instead of by the offered load as is the case when the network is unsaturated. This behaviour is undesirable since time sensitive traffic should be given a VL in the high-priority table [46], a VL with large available bandwidth. LBL will use this large available bandwidth to admit many flows so that there will be a large amount traffic in the high-priority VL. As the amount of traffic in the high-priority VL is increased by admitting traffic, the individual flows will receive poorer service, effectively neutralising the benefit of having the traffic in a high-priority VL in the first place. An extreme situation might illustrate the point: if all the traffic were to use the high-priority VL, the service of the individual flows would be no better than if all the traffic used a low priority VL. Time sensitive and high-priority traffic requires a VL a large amount of available bandwidth and a small amount of traffic.

If one assumes that high-priority traffic is assigned large weights in the arbitration tables, it is possible to utilise this by differentiating the VLs based on the inverse of the available bandwidth, admitting few flows on VLs with high available bandwidth and vice versa. The calculation of the admission control parameter used as the available bandwidth in this case, AC_{bw}, is as follows:

$$AC_{bw} = BW_{link} - BW_{VirtualLane} \tag{5.4}$$

Differentiation based on offered load

Instead of using the available bandwidth for each VL as an admission control parameter, one can assume that there will be less high-priority traffic than low-priority traffic, e.g., 10% of the total network traffic is high-priority and uses SL 1. This can utilised to divide the bandwidth according to the load given to the different SLs mapped onto the VLs. This requires that the load on the SLs is known a priori, something which is not always the case. We achieve the differentiation by statically dividing the link bandwidth into portions with a size relative to the traffic load of the SLs as in equation 5.5.

$$AC_{bw} = bw_{link} * \frac{load_{SL}}{load_{total}} \tag{5.5}$$

AC_{bw} is the bandwidth used by LBL for the VL to which the SL with the load $load_{SL}$ is mapped. We assume here a one-to-one mapping between SL and VL. $load_{total}$ is the total load put on the network from one node for all the SLs combined.

A SL with high load would be permitted to send large amounts of traffic, thereby giving them the chance to utilise the available network bandwidth, and leaving it up to the QoS mechanisms in the network to force these flows down to the level of the bandwidth available to the SL as necessary if the network reaches saturation. The SLs with a light load will be restrained by the admission control routine giving them, depending on the weighting and prioritising of VLs, less throughput and allowing them to experience lower latency. As mentioned, this method requires that the approximate load on every SL is known a priori. This is easily done in a simulation environment where there are parameters to determine the load to be put on each SL, but might not be so easily achieved in real life. However the theory still applies, admit less traffic on SLs requiring lower latency.

The bandwidth differentiation used in this thesis for LBL is based on the offered load approach as described above. This makes it easier to ascertain whether the admission control algorithm is able to give bandwidth guaranties on the class level as the differentiation resembles the offered traffic. In the special case of the simulations, the relationship between offered load and available bandwidth on the different SLs is such that the SLs with high available bandwidth, the SLs with the ability to provide best Quality of Service, has the lowest offered load (see table 6.1 in section 6.2). This should be similar to basing the differentiation on the inverse available bandwidth.

Bandwidth Calculation

Calculating the available bandwidth in an InfiniBand network is a tricky affair due to the nature of the arbitration tables. As described earlier, the scheduling mechanism of an Infiniband switch consists of a two level priority scheme with weighted

fair queuing within each priority level. The amount of available bandwidth for a VL is dependent upon four factors:

1. the weight assigned to the VL
2. whether it is a high-priority VL or a low priority VL
3. the total weighting assigned to the VLs of the current priority level
4. the limit of high-priority.

According to the specification ([9]) one packet may be sent from the low-priority table each time the limit of high-priority counter reaches 0. Within a given priority level the amount of bandwidth available to a VL is just the weight of the VL divided by the total weights of all the VLs in that priority level, times the bandwidth available to that arbitration table ($BW_{HP/LP}$), as made explicit in equation 5.6:

$$BW_{VirtualLane} = BW_{HP/LP} \frac{Weight_{VirtualLane}}{Weight_{HP/LP}} \tag{5.6}$$

with HP or LP depending on whether the VL is in the high-priority table or low-priority table.

The bandwidth available to the low-priority arbitration table in a saturated network is calculated as follows: The amount of bytes that may be sent from the high-priority (HP) arbitration table is $LHP * 4096$ bytes. Since the switch may send a HP packet as long as $LHP * 4069 > 0$, part of the last packet sent is sent in addition to the $LHP * 4096$ bytes, $packet_size \bmod (LHP * 4096)$. The total amount of HP bytes becomes

$$Bytes_{HP} = LHP * 4096 + packet_size \bmod (LHP * 4096) \tag{5.7}$$

The total amount of low-priority (LP) bytes that may be sent between each HP batch is

$$Bytes_{LP} = packet_size \tag{5.8}$$

The fraction of bandwidth available to the LP arbitration table is then

$$\begin{aligned} BW_{fraction} &= \tfrac{Bytes_{LP}}{Bytes_{LP}+Bytes_{HP}} \\ &= \tfrac{packet_size}{packet_size+(LHP*4096)+packet_size \bmod (LHP*4096)} \end{aligned} \tag{5.9}$$

The fraction from 5.9 may then be multiplied with the link bandwidth to arrive at the bandwidth available to the LP arbitration table according to the rules specified in [9] and in section 3.2:

$$BW_{LP} = \begin{cases} 0.5 * BW_{total} & \text{if } LPH = 0 \\ BW_{total} * BW_{fraction} & \text{if } 0 < LPH < 255 \\ 0 & \text{if } LHP = 255 \end{cases} \tag{5.10}$$

The amount of bandwidth available to the high-priority VLs is just the remaining bandwidth,

$$BW_{HP} = BW_{total} - BW_{LP} \tag{5.11}$$

Equation 5.6 calculates the minimum guaranteed bandwidth available to each VL, i.e. the amount of bandwidth the VL is assigned even when the network is saturated. Thus this is a description of how the throughput will be distributed between the VLs in the saturation region of the network, the region we intend to avoid using the admission control algorithm.

5.2 Endpoint Admission Control

In a large and high-speed network it is desirable to keep the switches as simple as possible. In a network which operates at high speeds the switches are required to process packets extremely fast and there is usually no time to perform any additional operations other than forwarding packets. Any sort of modification of switch parameters would take too much time and the network would slow down. Additionally if one were to store per flow information in the switches, the amount of data stored in each switch would rapidly increase as the size of the network and number of flows in the network increases. LBL described above is an example of such a scheme, requiring per flow alteration of the switches.

Because of these limitations one wishes to keep as much of the information and intelligence outside the core of the network as possible, mainly in routers or even ingress/egress nodes. In Infiniband the Quality of Service mechanisms are already built into the core elements in the network, the switches and routers, in an effective class-based manner much as in a DiffServ system. When implementing admission control into this scheme it is desirable to leave the switches unchanged and only change the endpoints in order to avoid increasing the load of the switches. In such a scenario where the endpoints perform the admission control it is natural to utilise a measurement based admission control algorithm since measurements are the most practical way for the endpoints to gather information about the current load in the network.

5.2.1 Egress Admission Control

Egress Measurements (EM), which is described in [21] and [56], is an admission control scheme developed for use in conjunction with IP (Internet Protocol). The admission control decision is located at the egress of the network, or the destination node. In [56] the authors implement the admission control scheme described in [21], while making some adjustments and clarifications appropriate for the system in which the admission control scheme is implemented. In order to be able to

make any admission control decisions, the egress nodes/routers have to gather information about the current load of the network. The only way an egress node has of knowing anything about the network to which it is connected is to analyse the traffic it receives locally from the network.

The egress node constructs two functions which together provide the information necessary to make an admission control decision. First is a service envelope, a function describing the packet service rate. To make more information available to the egress nodes, each packet in the network is timestamped as it departs from the networks ingress node/router. These timestamps are then used by the egress node to calculate the service envelope, basically the amount of packet units serviced by the network as a function of time intervals [21]. Second is an arrival envelope, a function describing the packet arrival rate at the egress node.

Service Envelope To calculate the service envelope the egress node measures the time each packets spends in the network, that is, the time a_j from which the packet j arrives in the network to the time d_j when it departs from the network. As long as there is only one single packet in the network en-route to a specific egress node, the service class (SL in InfiniBand) is said to be backlogged for $k = 1$ packet transmission. If a new packet enters the network before the previous has departed, that is, $a_j < d_{j-1}$, the service class is said to be backlogged for $k = 2$ packet transmissions. If a third packet enters the network before one or both of the previous packets have departed, then the service class is backlogged for $k = 3$ and so on. Mathematically it can be said that a traffic flow is continuously backlogged for k packet transmissions in the interval $[a_, d_{j+k-1}]$ if

$$d_{j+m} < a_{j+m+1} \text{ for all } 0 \leq m \leq k-2 \tag{5.12}$$

for $k \geq 2$ [21]. The egress node gathers the maximum of these measurements in an increasing function of packet units according to the following equation:

$$U_i = max(U_i, d_{j+k-1} - a_j) \tag{5.13}$$

where

$$i = \sum_{m=0}^{k-1} l_{j+m} \tag{5.14}$$

and l_{j+m} is the size of packet $j + m$ expressed in packet units [21]. U_i is the time it takes for the network to service i packet units.

This is done during a measurement interval $T\tau$ (explained below) and stored. After M such measurement intervals the mean ($\overline{U}$) and variance of these M measurements are calculated for use in the calculations. If $\overline{U}$ is inverted we get the number of packet units serviced as a function of time which is our service envelope $S(t)$.

Arrival Envelope The egress node measures the rate of incoming packets in order to calculate the aggregate maximal rate envelope $\overline{R}(t)$. Time is divided into T time slots of length τ and the number of a service class' arrivals in such a time slot is stored in a table. Using the following notation one can say that $A[s, s+I_k]$ is the number of arrived packet units in the interval $[s, s+I_k]$, and the arrival rate in the same interval is $A[s, s+I_k]/I_k$ [21]. The peak rate of this interval length is the maximum rate of every interval measured, $R_k = max_s A[s, s+I_k]/I_k$. If we say that the interval length is k times the interval length τ we have that

$$R_k^1 = \frac{1}{k\tau} max_{t-T+k \leq s \leq t} A[(s-k+1)\tau, s\tau] \tag{5.15}$$

[21] for $k = 1, ..., T$. This is done every T time slots and the final measured peak rate arrival envelope $\overline{R}(t)$ is calculated from the M most recent R_k. The measured peak rate arrival envelope is basically a table containing the maximum rate of arrivals for longer and longer intervals. As the interval length increases, the sensitivity to the current arrival rate is decreased as the arrival rate approaches the mean arrival rate.

The Admission Control Function To summarise, the egress node has information about the incoming rate and the outgoing rate in the network on every path to itself in $\overline{S}(t)$ and $\overline{R}(t)$ respectively. These two values are different in one respect, namely that $\overline{R}(t)$ is measured as packet units per time units in a given time interval t, e.g. bps, and $\overline{S}(t)$ is just the number of packet units serviced by the network for the same time interval t. For a time interval t, if one were to subtract $\overline{S}(t)$ from $t * \overline{R}(t)$ one would get a negative number indicating the remaining capacity of the network as the result (as long as the network is not overloaded)[21]. Then, if one were to add a new flow to the network in the state described by these variables with a peak rate P and delay bound D one would simply add the peak rate P to the measured peak rate arrival envelope and increase the interval for the service envelope with the delay bound D like this $\overline{S}(t+D)$. By computing the variance of the measurements of the measured peak rate arrival envelope and the service envelope and adding this to the equation described above, the result is as follows: A flow with peak rate P and delay bound D may be admitted to the network if the following condition is satisfied:

$$t\overline{R}(t) + Pt - \overline{S}(t+D) + \alpha\sqrt{t^2\sigma^2(t) + \Psi^2(t+D)} < 0 \tag{5.16}$$

for all interval lengths $0 \leq t \leq T$. $\overline{R}(t)$ and $\overline{S}(t+D)$ are as already described, and $\sigma^2(t)$ and $\Psi^2(t+D)$ are the variance of the arrival envelope and the service envelope respectively. This equation simply states that a flow may be admitted if the arrival rate plus the peak rate is less than the service rate within an acceptable

delay bound, corrected by the variance for all possible interval lengths less than T. The variable α in the above equation is a variable defining the confidence level of the equation.

In addition there is one other limitation on the number of flows that may be accepted, known as the stability criterion in [21], the peak rate arrival envelope plus the peak rate of the new flow may not exceed the service rate of the incoming links [56]. This is realised in the following equation:

$$\lim_{t\to\infty} \overline{R}(t) + P < \lim_{t\to\infty} \frac{\overline{S}(t)}{t} \tag{5.17}$$

where, according to [56], $\frac{\overline{S}(t)}{t}$ may be replaced by the incoming link's bandwidth.

A flow may be accepted if both equations 5.16 and 5.17 are satisfied for every t between 0 and T.

In order to get a flow accepted, a node wishing to initiate a new flow must send a reservation message through the network to the egress node in order for it to perform the admission control. This requires some sort of signaling or reservation protocol. In [56] the authors proposed to use RSVP as a signaling protocol, though with some modifications. The RSVP demon is only required to be running on the boundary nodes/routers in the system since the core nodes of the network are not required to take any part in the admission control routine other than to forward the packets.

Synchronisation

Since the admission control scheme requires time-stamping of packets, some sort of reliable synchronisation between endpoint nodes must be in place in order for the system to operate correctly. The authors of [56] propose to use the Network Time Protocol (NTP) [41] for this purpose. The accuracy with this type of synchronisation is not sufficient for use in a small to medium System Area Network such as a server cluster. It is more suited for synchronising computers in networks which do not require very high accuracy. The authors take this a step further by indicating that the Global Positioning System (GPS) might be utilised for synchronisation purposes by building a GPS receiver into the network nodes. In [32] and [61] however, the authors present methods for achieving high precision synchronisation in switched networks using NTP, making this a viable option.

In a simulator environment, such as the one to be described later, in which this admission control scheme will be implemented and evaluated, it is not necessary to provide any method of synchronisation to the system. This is because the simulator runs on a stand-alone machine and every node in the simulation therefore has the same simulation and system time.

Open Issues

The authors of [21] do not specify what is to happen at the beginning of a network's lifetime. How can the egress nodes perform any sort of admission control without any traffic yet having passed through the egress node? All the measurements will be 0 at the beginning of a network's existence and thus equation (5.16) cannot be < 0. One way of avoiding this is to admit flows until one has a certain amount of security/certainty in the measurements and then start the admission control seriously.

Another way of achieving this is a hybrid approach utilising some sort of probing scheme at the beginning of a network's existence until enough history has been built up at the egress nodes. One such probing scheme is described in [18], but the problems with these approaches are that the probe packets require either to be dropped when they are in overflow or some sort of marking mechanism which marks the probe packets that exceed some threshold. Infiniband does not drop packets. Infiniband switches are subject to VCT flow control and packet queues build up in the switches along the path to the sender. If one were to perform the probing in the same SL as the actual traffic would use, this queue-up caused by the excess probing packets would affect and degrade the service of the already admitted flows in the SL. It is possible to use marking of probe packets that should have been dropped, packets that are delayed, rather than dropping of probing packets. This is an approach which leads to shorter probing times [18], but one has the difficulty of relating the marking rate with the acceptance threshold [18], and of setting the marking threshold of the probe packets. The criterion for marking packets would be when the queue in a switch exceeds a certain length. Exactly at what queue length packets are to be marked is uncertain, but this could be determined through experiments. One could treat these marked packets as dropped, but as already marked packets could cause problems in many switches ahead there will be an excess of packets marked. [18] does not describe any method of relating packet marking rate to the acceptance threshold. Different probing schemes are discussed further in section 5.2.2.

Yet another way of achieving admission control before the egress routers have had the chance to build up an adequate history would be for the ingress node to send a certain amount of dummy/probe traffic to the egress node and have the egress node check to see if the already admitted flows are affected by the test traffic. If the egress node sees that the admitted traffic is affected by the test traffic it can send a message to the source of the test traffic indicating that the flow is rejected. The drawback of this approach is that already admitted traffic may receive poorer service during the test run. Another problem is that the test traffic may affect the traffic received by other egress nodes without them being able to let the source know.

In the results presented in the next chapters the method deployed is one in which the egress nodes accept all traffic until they have enough data to perform admission control in an orderly fashion.

5.2.2 Jitter Probing

Instead of passively monitoring the network activity in the egress nodes of the network, it is possible for the end nodes in the network to take a more active role in the admission control decision. This may be done by actively sending probe packets through the network from the ingress node and monitoring the probe packets' arrivals at the egress of the network. If the size and rate of the probe packets is designed correctly they should give the egress node the opportunity to calculate how the new flow will be treated by the network. As opposed to the EM scheme described in section 5.2.1 where the egress nodes had to calculate the available capacity in the network based on the already existing traffic, the additional probe packets give the end points the opportunity to inject more traffic into the network and see what happens. This may lead to other problems such as bandwidth stealing.

Bandwidth stealing is a phenomenon where the probe packets steal bandwidth from the already admitted flows, degrading the service they experience [31].

Several probing schemes have been proposed, some of which are described in [31], [18], and [13]. Both [31] and [18] require either that packets be dropped to indicate congestion, which is impossible when using virtual cut-through switching as Infiniband does, or that congested packets be marked in the switches in the network. This requires intelligence in the switches and partly negates the whole point of end-point admission control. In [13] the authors propose a probing scheme in which the load level of the network is calculated by measuring the jitter of the probe packets. The authors solve the bandwidth stealing problem by requiring that the probe packets be forwarded through the network with the lowest priority of all packets. This ensures that the probe packets will be unable to steal bandwidth from the already existing traffic in the network whilst additionally giving worst-case measurements of the network jitter and thus guaranteeing that the traffic when admitted will get at least the service that the probe packets received. When applying this to the InfiniBand network it is therefore natural to let the probe traffic be forwarded on a VL in the low priority table with a relatively low weight, possibly equal to 1. The probe packets will still be guaranteed delivery to their destination even if the network is saturated, but due to the nature of virtual cut-through networks the probe packets will most likely experience large jitter and the flow will therefore be rejected.

For each probe packet received, the receiver registers the packet's transmission time, e.g. the time the packet spends in the network. When an adequate num-

ber of probe packets have been sent and received the receiver calculates the jitter by subtracting the minimum packet transmission time from the maximum packet transmission time. This value is compared to the jitter requirements embedded in the probe packets and an admission decision is sent back to the sender. Additionally if any of the probe packets are rejected by the sender due to the limited size of the send queue buffer, the flow is also rejected. In other words a flow is accepted if

$$TransmissionTime_{max} - TransmissionTime_{min} < Jitter_threshold \tag{5.18}$$

and

$$Packets_rejected = 0 \tag{5.19}$$

The worst-case latency of a path through a virtual cut-through network grows exponentially as the network size increases [51] and as the network load increases so does the probability of a packet having to wait while other packets are transmitted on a link. One packet waiting in one switch may lead to several packets waiting in several switches upstream and so on. This increase in network load will lead to a measurable increase in packet jitter.

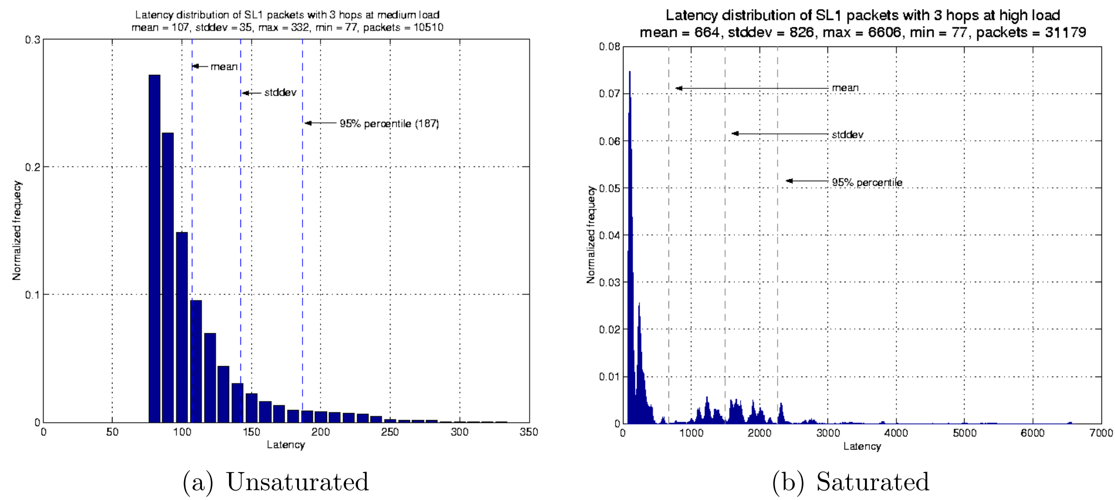
Figure 5.1 and 5.2 are figures taken from the simulations presented in chapter 7. The simulations use 5 SLs of which the two first are high-priority. The two plots in Figure 5.1 represent the distribution of packet latency on SL1 for all packets traversing 3 hops in the network. The x-axis is latency and the y-axis is the frequency of that latency. The state of the network as the load increases is displayed in Figure 5.2. We see here how the 5 SLs are treated by the network as the load increases along the x-axis. The y-axis represents the aggregated throughput of all the flows in the SLs.

Figure 5.1(a) is the latency distribution for an unsaturated network corresponding to the "'medium"'-marker in Figure 5.2. Similarly Figure 5.1(b) is the latency distribution for a saturated network, a network in a state corresponding to the "'high"'-marker in Figure 5.2. Recall from section 4.1.1 that the network should operate below saturation to be able to give bandwidth guarantees. We can clearly see from figures 5.1(a) and 5.1(b) that the variation in latency increases drastically as the network reaches saturation, indicating that packet jitter may be used as an indication of network saturation.

5.3 Centralised Admission Control

5.3.1 Link-by-Link

As explained in section 5.2.1 it is undesirable to put too much functionality in the switches of a large-scale network. The LBL admission control scheme described



(a) Unsaturated (b) Saturated

Figure 5.1: Latency distribution for SL1, 3 hops in a unsaturated and saturated network

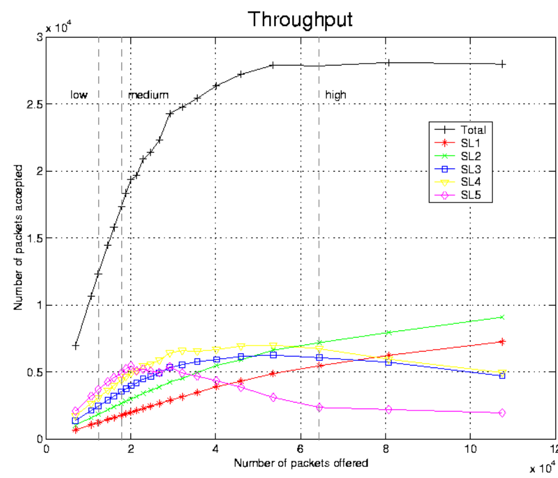


Figure 5.2: Figure depicting saturated and non-saturated regions indicated with high and medium marker

there is therefore impractical to implement in a real-life network. It is, however, possible to get the same functionality if one were to utilise something similar to a BB as it described in conjunction with DiffServ in Section 2.3.1. In this case the BB is a node with knowledge of the total network topology which holds information about the switches' reservations. When a flow requests admittance, the request is sent to the BB which traverses its tables and is able to perform the same type of control as if the real-life switches were traversed by the reservation message. This is the same setup as described previously in Figure 2.4 in Section 2.4. The "measured sum" approach is not as easily adapted for use with a BB since it requires that the switches perform measurements which are not suitable to be moved to the bandwidth broker. The switches may perform measurements and send reports to the BB so it can perform admission control, but this still requires much functionality in the switches which is what we are trying to avoid.

In a large network with much traffic and many flows there will be many flow admission requests and flow tear-down requests sent to the single BB in the system. The BB becomes a hot-spot in the system and may become a bottleneck, limiting the efficiency and scalability of this approach.

5.3.2 Combinations

It is possible to combine several AC-schemes into one using a BB much like the system described in [72]. One could for example let the egress nodes described in section 5.2.1 report to the BB in combination with the link-by-link scheme described above in 5.3.1. This can be done in order to provide parameter based AC for high-priority traffic and measurement based AC for the low-priority traffic much as in [72]. Such combinations will not be explored by this project.

5.4 Aiming for Low Jitter

As we will see in chapter 7 and as has been described in [51] and [50], low jitter may be hard to achieve in VCT networks. The main reason for this is the nature of the flow control mechanism utilised in such networks. A packet buildup in one switch in the network will lead to packets being blocked in several other switches leading to highly unpredictable network latency. All the methods described so far in this chapter focus on providing guaranteed bandwidth or low jitter through the use of admission control algorithms. As we will see in chapter 7 these methods are not sufficient to guarantee a network jitter within the bounds set by the admission control algorithms, it seems that a more radical approach may be necessary.

Suppose we have an application with hard jitter demands, but which might accept some packet loss. An example of such an application might be a video

stream. Now, suppose if one, instead of, or in addition to admission control, started dropping packets whenever they were held back by the flow control. By monitoring the queue length in the switches one could drop packets whenever the queue length reached a certain threshold which is below the point where the flow control mechanism starts holding packets back in the upstream switches. This mechanism will avoid the state where the flow control starts blocking packets and affecting network latency, and thus hopefully keep the network latency at a stable level.

It must be noted that this approach is rather radical and not really associable with the VCT paradigm. It seems however that some drastic means are needed in order to guarantee jitter within a certain bounds in VCT networks. This method will probably be able to provide good jitter characteristics, but whether they are any better than the admission control algorithms described previously remains to be seen in chapter 7.

5.5 Summary of the Proposals

The presentation and discussion of the different admission control algorithms in this chapter leads to six scenarios that will be investigated. A comparison and evaluation of an uncontrolled network and three different admission control algorithms will take place. These algorithms are described previously in this chapter, and a short summary is given here:

- Link-by-Link is presented in Section 5.1. The algorithm requires detailed information about the load on each link in the network. To achieve bandwidth differentiation, each VL is assigned a portion of the link's bandwidth in accordance with the offered load as described in Section 5.1.3. VLs to which SLs with low offered load are mapped receive a smaller portion of bandwidth than VLs to which SLs with higher offered load is mapped. The parameter based (simple sum) approach is chosen where the flow's peak rate is reserved on every link on the path.

- Egress Measurements is presented in Section 5.2.1. The algorithm measures the arrival rate and service rate of the packets and uses the allowed increase in service time to decide whether a flow with a given peak rate may be admitted.

- Jitter Probing is presented in Section 5.2.2. Probe packets are sent through the network, and the packet jitter is measured at the egress. The admission control decision is based on whether the measured jitter is below the requirement or not.

Additionally, the effect of packet dropping on throughput and jitter will be studied, both without admission control and with Link-by-Link admission control.

Chapter 6

Simulations

This chapter presents the environment in which the admission control algorithms proposed in chapter 5 will be tested. First an overview of the simulator used in this project is given, followed by a more detailed presentation of the different simulation and admission control parameters used.

6.1 Simulation Environment

The admission control algorithms proposed in chapter 5 are implemented in a simulator developed in house at the Simula Research Laboratory (SRL) by Ingebjørg Theiss and subsequently by Sven-Arne Reinemo. The simulator consists of a discrete event-driven simulation engine with several types of network elements generating events which are executed by the engine at the appropriate simulation time. The simulator is originally developed for virtual cut-through networks and has therefore been adapted to Infiniband by Sven-Arne Reinemo. The simulator code is written in Java.

The results received from the simulator are further processed in MatLab or some other appropriate mathematical program in order to analyse the results and present them in intelligible ways.

6.1.1 Network Components in the Simulator

The simulator provides code for every network component necessary to simulate a local or system area network. There are nodes/processors that produce and consume packets, switches that forward the packets further along the correct links to their destination and links that transport packets from one processor/switch to another. The simulator does not contain routers or subnet managers so the network parameters are statically configured at the start of simulation time. This also pre-

cludes the option of linking several subnets together with routers, or recalculating routing tables and forwarding tables during the simulation lifetime.

The links in the simulator are able to process one flit per simulation cycle, with a flit-size of 1 byte. One simulation cycle is defined to be 3 ns long, giving a link speed of 2.5 Gbps.

6.1.2 The simulator engine

The simulation engine, or kernel, is as already mentioned discrete event driven. Every event in the simulation lifetime, that is, the generation of a packet, the arrival of a packet on a link, the arrival of a packet at a switch, forwarding table look-up completion and so on, generates an event with a time stamp sometime in the future, indicating the time at which the event should be completed. Every such event is put in an event heap waiting to be scheduled by the simulator kernel. The simulator kernel has a dispatcher which goes through the heap of upcoming events executing the events scheduled for the current simulation time. When there are no more events for the current simulation time, the time is advanced one step and the process is repeated until the heap is empty or the maximum simulation time is reached.

There is one major problem inherent in such event driven simulators, this is the problem of concurrency. In a real-life network there are multiple processing units doing calculations in parallel. When one is simulating such a network on a single processing unit system such as an ordinary computer it is no longer possible to do multiple calculations in parallel. In order to simulate such parallel calculations the only option is to enqueue the events taking place at the same time in some specific order and execute them serially. This may cause problems since the outcome may depend upon in which order the events are executed. Much work has been done in the field of the event driven simulations, some of which can be read in [57] and [20]. In our simulator, the simple approach has been chosen of executing in a random order events scheduled for the same time tick.

6.1.3 Network Topologies

When dealing with simulations, several simulation runs with different topologies must be made in order to achieve a certain degree of statistical credibility in the results. We have chosen to use irregular topologies since this is the more general case. Regular topologies are a special case of irregular topologies. This means that every algorithm that works in a irregular network will also work in regular networks. Therefore, by testing the algorithms in irregular networks, we achieve results which represents the general case. Every simulation series is carried out on 16 different random irregular topologies.

A typical network configuration is a network containing 32 switches interconnected randomly, but in such a way that every network node has a path to every other. There are connected 5 processors, one for each SL, to every switch in the network. Each processor is connected to its switch with a separate link such that every switch is connected to five processors and its neighbouring switches. In the core of the network however, between switches, the SLs are mapped to different VLs on the same link. The processors connected to a switch form a logical entity as a single node, and when talking about packet production rates of a node in the next sections this is derived from the combined rates of a node's processors.

6.1.4 Routing

There exists few generic routing algorithms which are able to guarantee shortest path routing in irregular networks. One of the few algorithms to achieve this is Layered Shortest Path Routing (LASH) [64]. This is a deterministic routing algorithm which guarantees shortest path routing and in-order delivery in both regular and irregular networks [64]. Additionally, in [63] the authors show that generic LASH is as efficient as dedicated routing algorithms for regular topologies such as Dimension-Ordered Routing (DOR) [66].

LASH utilises VLs to create shortest path, deadlock free routes, a service which is already available in the Infiniband switches, and is therefore well-suited for use in this scenario. The routing algorithm works by assigning all source/destination pairs to exactly one virtual layer. It is then able to ensure that each layer is deadlock free by assigning the source/destination pairs to layers in such a way that the source/destination pairs assigned to one layer do not generate cycles. The idea of using VLs to avoid routing deadlocks is also explored in [54]. Here the authors use Up*/Down* routing and solve potential deadlocks by moving certain paths to other VLs. The authors of [48] present another method for calculating minimal routing in Infiniband networks through the use of destination renaming, exploiting the existence of multiple QPs in the hosts.

6.2 Simulation Parameters

The configuration of the SLs in the simulations are inspired by the traffic classes in DiffServ. In the DiffServ terminology, we have two SLs, SL 1 and SL 2, which are considered to be in the Expedited Forwarding (EF) traffic class. These two SLs are to support high priority, time sensitive traffic and are therefore mapped to VLs with entries in the high-priority arbitration table. Two more SLs, SL 3 and SL 4, are implemented to correspond to the Assured Forwarding (AF) traffic class. These SLs are available to traffic requiring throughput guarantees from the

network, and are mapped to VLs in the low-priority arbitration table. Finally SL 5 is implemented as a Best Effort (BE) SL, a SL available to elastic applications with little or no QoS requirements from the network. This SL is mapped to a VL in the low-priority table with very low weight so that it may not disturb SLs with higher priorities. Every SL is mapped to a unique VL, that is, there is a one-to-one relationship between SLs and VLs. However, the routing algorithm uses several VLs to achieve deadlock-free routing, so the SLs are in reality mapped to several VLs. This does not affect the performance of the admission control algorithms or QoS mechanisms, the relative relationship between SLs and VLs remains the same.

For SL 1 to SL 4 there is a clear relationship between the weights of the VL and traffic load on the corresponding SL. VLs with a small weight has a smaller percentage of the network node than VLs with higher weight. SL 5 is the exception, it has a weight of 1 and the most traffic. This is done to emulate the large amount of best effort traffic that may be found in a network and to try to utilise the network to its fullest extent while still giving QoS guarantees to the EF and AF traffic classes.

The relationship between SLs, priorities and VL weights are given in table 6.1.

SL	Priority	VLWeight	Load
1	high	4	10
2	high	6	15
3	low	8	20
4	low	10	25
5	low	1	30

Table 6.1: Service Level and Virtual Lane Configuration

The load is given as a percentage of the load of a node and the load of every processor on a node should therefore add up to 100%. The weights in the table are according to the specification meant to indicate the number of 64 bytes units to be sent on the VL when its turn comes, but due to the small packet size (32 bytes) used in most of the simulations, the values are redefined to mean the number of packets to be sent from each VL. The limit of high-priority is set to 32, again being redefined from meaning 4096 times the number of bytes to the transmitted from the high-priority table to mean the number of packets to be transmitted from the high-priority table.

Due to time limitations, most of the simulations performed in this project use a packet size of 32 bytes. Increasing the packet size to a more normal size,

for example 512 bytes, leads to an increase in simulation time from 200 000 to 3.2 million simulation cycles in order to get the same number of packets through the network. Using large packets leads to longer simulation times. This project requires a large number of simulations and it is therefore imperative to keep each simulation run as short as possible for the data to be available within a reasonable time.

With a larger packet size the packet rate of each flow must be reduced to achieve the same data rate for each flow. This results in that the data rate in the network remains the same, only the simulation runs for a longer period of time. As the admission control algorithms look at either data rate or latency/jitter, the increase of data in each packet should not affect the performance of the algorithms as the data rate is unaltered. The latency and jitter will increase, but this may be accounted for when choosing the latency and jitter admission control parameters.

Half of the processors in the network are configured to send flows with a packet rate of 0.0001 packets per simulation cycle. With a packet size of 32 bytes and a cycle length of 3 ns this adds up to a bit rate of 8.13 Mbps. The other half are configured to send flows with a packet rate double that of the first half, 0.0002 packets per simulation cycle, yielding an effective bit rate of 16.26 Mbps. Half of the processors in a SL are configured with the first packet rate, while the other half sends with the second packet rate. In this way we have flows with different bandwidth requirements in each SL giving us the opportunity to study whether each flow receives bandwidth it requests or if the service is unrelated to the requested bandwidth.

6.2.1 Admission Control Criteria

The following paragraphs discuss the admission control criteria used as parameters to the admission control algorithms in the simulations. Recall that the different algorithms use different parameters for admission control. LBL uses bandwidth as the admission control parameter while egress measurements uses latency and jitter probing uses packet jitter as the admission control parameter. Consequently it is difficult to configure the different admission control algorithms to operate in the same manner. There is no clear way of defining the relationship between bandwidth guarantees, latency guarantees, or jitter guarantees. Consequently, given the different admission control parameters used, the proposed algorithms will have different performance as to bandwidth, latency and jitter guarantees.

Link-by-Link

The LBL admission control algorithm bases its admission control decisions on detailed knowledge about the available unreserved bandwidth on each link . A

switched network such as the one mentioned here is not able to fully utilise the theoretical link capacity. Based on results presented in [51] the link utilisation percent is set to 50%. This is further divided amongst the VLs according to the Offered Load (OL) method described in 5.1.3 as the first admission control parameter. The peak rate for each flow, which is given to the admission control algorithm as the second parameter, is double that of the mean flow bandwidth when using self similar traffic generation (Section 6.2.5).

Egress Measurements

The egress measurements admission control algorithm uses both a flow's requested maximum latency and a flow's peak rate as parameters. The peak rate is, as for every other algorithm, double that of the flows the mean rate as described above. Recall from figures 5.1 and 5.2 in Section 5.2.2 that there was a marked difference in latency distribution within a SL in an unsaturated and a saturated network. Instead of limiting the network throughput to just below network saturation by monitoring the available bandwidth as is done in LBL, it should be possible to limit network latency to the level just below network saturation and achieve a result similar to that of LBL. The latency parameter has therefore been obtained by looking at the mean value of the latency as a function of hops at medium load (unsaturated network, see Figure 5.2) without admission control, see Figure 6.1 for the latency distribution.

Based on the mean latency distributions of larger hops the latency requirement is increased by 25 time units per hop. The latency requirements for each SL are listed in table 6.2.

SL	**1 hop**	**2 hops**	**3 hops**	**4 hops**
1 - 4	100	125	150	175
5	∞	∞	∞	∞

Table 6.2: Latency requirement distribution for each hop for use with Egress Measurements

The infinite latency values for SL 5 in table 6.2 indicates that this is a best effort SL and should therefore not be subject to admission control.

Notice that all the SLs have the same latency requirements regardless of the mean latency in Figure 6.1. The admission control algorithm is not configured to differentiate SL 1 - SL 4. Experiments have shown that admitting more traffic on low-priority SL 3 and SL 4 leads to poorer service for every SL. It has therefore been decided to have strict demands on every SL in order to achieve as good results as possible. A possible causes for this behaviour is that the value of LHP

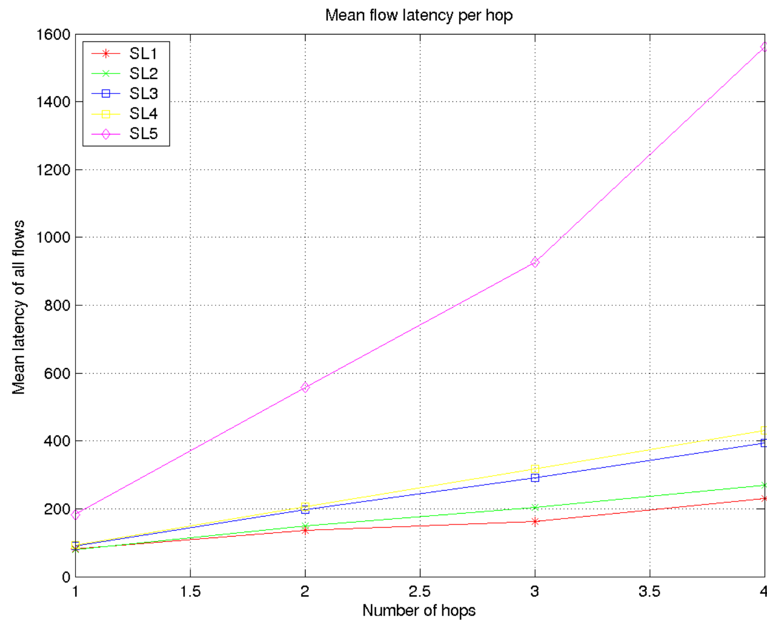


Figure 6.1: Average flow latency as a function of hops

is so low that as long as the network is below saturation, the addition of more low-priority traffic will worsen the service perceived by high-priority traffic until the LHP becomes the dominating factor. As long as we admission control the network, the network will never reach saturation and LHP will never dominate. This makes the high-priority traffic susceptible to be influenced by the low-priority traffic.

Jitter Probing

Jitter probing uses only desired maximum jitter as the admission control parameter. As for egress measurements we surmise from figure 5.1 in section 5.2.2 that packet jitter worsens as the network reaches saturation. The jitter requirements for each flow are therefore inspired by the jitter perceived by each flow at medium load with no admission control, see Figure 6.2. Every probe is sent on the same SL, SL 6, which for this occasion is an SL added for probe packets. Every probe will therefore receive the same service from the network without regard to the SL sending the probes. SL 5 is best effort and is therefore not subject to admission control and SL 6 is reserved for probing with a weight of one and zero load (see table 6.4). The jitter requirements for the other SLs are given in table 6.3. The

jitter requirements are based on the offered load to attempt to achieve a reasonable SL differentiation. As the offered load on the SLs increases so does the amount of allowed jitter. This places hard demands on the high-priority SLs SL 1 and SL 2 and less strict demands on the low priority SLs, SL 3 and SL 4. Path length is not taken into account here as it is with egress measurements. Given the increase in jitter in Figure 6.2 as the number of hops increase, there will be fewer accepted flows on paths containing many hops. This makes it easier to determine whether the jitter requirements have been met, every flow in a SL should display jitter within the same bound.

SL	Jitter
1	15
2	20
3	25
4	30

Table 6.3: Jitter requirement distribution

With a simulation time of 200000 simulation cycles and a flow rate of 0.0001 packets per cycle each flow is able to send 20 packets through the network. In order to get an accurate picture of the network state a total of 6 probe packets is sent for each flow requesting admittance. This is quite a large amount of probe packets compared to the length of a flow, about 30% of the flows lifetime, and we should therefore be able to get a quite accurate picture of how the flow will be treated in the network.

SL	Priority	VLWeight	Load
6	low	1	0

Table 6.4: The SL and VL added for use with Jitter Probing, SL 1 - SL 5 remain the same

6.2.2 The Nature of the Simulations

A series of simulations consists of several individual simulation runs at different loads. The rates range from one packet per node per 15 time units (highest load) to one packet per node per 230 time units (lowest load). Given a typical network size of 32 nodes, packet size equals 32 bytes and a link speed of 2.5 Gbps leading to a time unit of 3 ns, mean load put on the network from one node at high

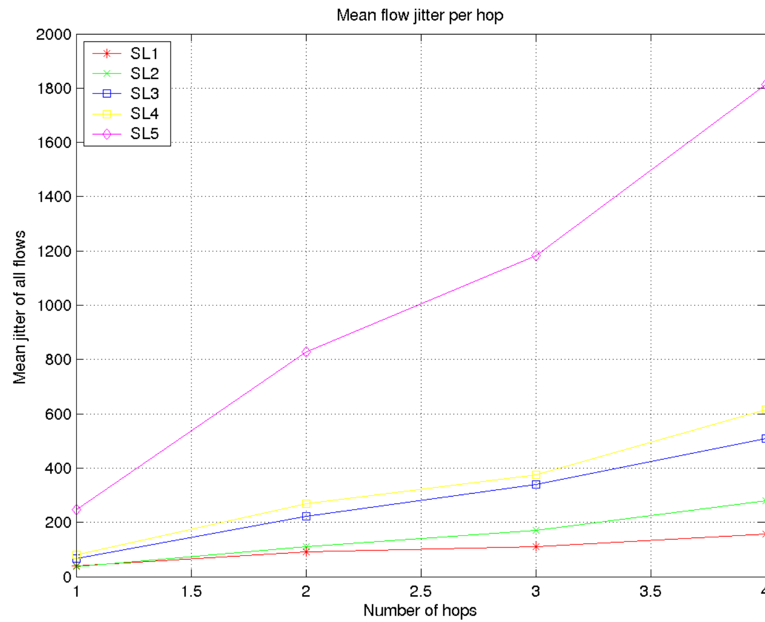


Figure 6.2: Jitter as a function of Hops

load equals 5.3 Gbps. This is twice that of the link rate divided between the five links connecting the five processors to the switch. This load is high enough to saturate the network. This is also evident from the simulation run with no admission control. At the lowest load, one packet per 230 time units, the load on the network from a given node equals 353 Mbps, a value well below the link capacity.

Each simulation produces a file containing latency data for every packet in the network with information about which flow and SL it belongs to, how many hops it has traversed and so on. This information is used to calculate the mean latency and jitter of every flow in the network, giving a picture of how the traffic is treated in different SLs.

A simulation starts by having each processor attempt to add a new flow to the network. This continues at regular intervals until every flow attempting admission is rejected or we have reached the target load of that specific simulation run. After there have been no flow admissions for a specific number of retries, the processors give up trying to insert more flows and the simulation goes into a stabilising period. When the network has reached its stable state the system starts gathering statistics for as many simulation cycles as specified.

It is important here to understand that as the target load of the simulation

runs increase, the packet rate of a flow remains stable. As the load increases the time between each new packet being produced in the processor decreases. Since the flow bandwidth remains stable, the effect of an increasing target load is the ability to add more flows to the network.

6.2.3 Other technological assumptions inherent in the simulator

The core of the switch built into the simulator consists of a crossbar to which each link has dedicated access. Every VL on a link is multiplexed on to the crossbar between the other VLs on the link. A crossbar is a interconnection matrix providing a connection from every input port to every output port in the switch. To which output port a given input port is to transmit its data is controlled by the switch logic. Each switch has enough buffer capacity for exactly one packet on the input end and buffer capacity of 2 flits at the output end for increased efficiency. At the input side VL arbitration is performed to decide which VL is to send next for this link. On the output side link arbitration is performed to decide which input link is to send to the specific output link. This arbitration is done in a round-robin fashion. The switch architecture is depicted in Figure 6.3. Every processor is able to buffer two packets should the output link be busy. The overall network design in the simulator is based on the canonical router architecture described in [25].

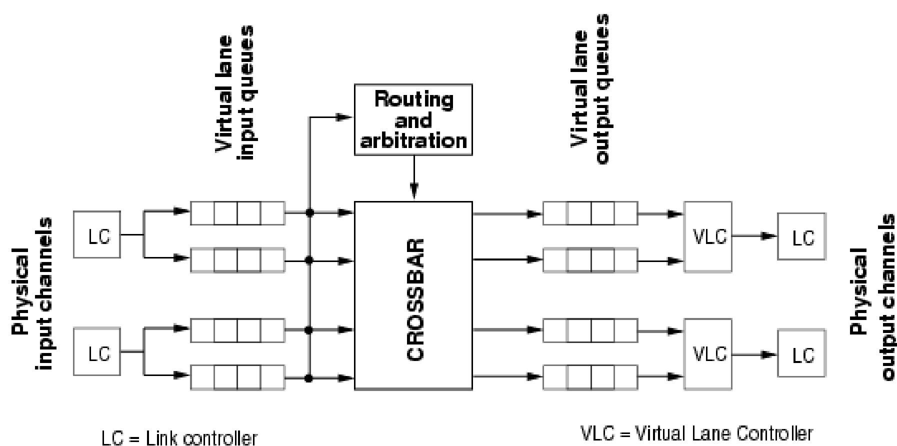


Figure 6.3: Switch architecture

6.2.4 Traffic Distribution

A variable that has some impact on the performance of a network in network simulations is how the traffic is distributed throughout the network. There are many ways in which traffic may be distributed in a network. With the coming of peer-to-peer file sharing programs such as Kazaa [5], Gnutella [2], Direct Connect [1] and others, we have a traffic pattern where many nodes both receive and send data to many other nodes. The traffic is reasonably evenly distributed in the network giving the possibility of high network utilisation. On the other hand, in a typical Internet environment much of the traffic flows between a huge number of clients (Web browsers, FTP clients, IRC clients, etc.) to a limited number of servers (Web servers, FTP servers, IRC servers, etc.), we have a typical many-to-one/one-to-many scenario. Given a certain amount of traffic these servers will become hot spots and function as bottlenecks in terms of network utilisation. This traffic distribution will not be able to achieve as high network utilisation as the peer-to-peer case.

The simulations in this project are performed using two distinctly different traffic patterns, one for each of the two scenarios described above. The first traffic pattern is called "random pairs" where each node is receiving data from one and only one source and sending data to one and only one destination. The source and destination are not the same node. This is somewhat similar to the peer-to-peer file sharing scenario with the difference being that each client is downloading only from one of the peers at a time and serving only one peer at a time. The other traffic pattern consists of choosing random nodes to become hot-spots in the network. Every node in the network sends its data to one of the designated hot-spot nodes in the network. The hot-spot nodes themselves send data to one of the other hot-spot nodes (if any) in the network. If there are no other hot-spot nodes in the network one of the other nodes is randomly chosen.

Random pairs traffic pattern will probe the limits of the network utilisation possible to achieve with different admission control algorithms whilst the hot-spot traffic pattern will challenge the ability of the admission control algorithm to limit the network traffic based on the load of a bottleneck link. In the simulations two hot-spot nodes will be used.

6.2.5 Traffic Generation

The results obtained from a simulator is dependent upon the process used to generate traffic. Since the simulated network is not a real network we have to rely on mathematical models or traces from real-life traffic to calculate when packets should be sent from the different sources. There exist many such algorithms for generating artificial traffic in the network, many of them tailored to the type of

traffic one wishes to generate.

Poisson Distributed Arrivals

One such commonly used method of emulating network traffic is known as the Poisson process. A Poisson arrival process is a process in which the probability of a packet arriving in a certain interval is independent of packet arrivals in other intervals. Given the mean inter-arrival time and standard deviation the Poisson process generates the next time a packet should be sent in such a way that the mean inter-arrival time of all the packets approach the desired mean value. This method works well for modeling the arrival of telephone calls [40] and FTP sessions [45], but as we shall see in the next section this method is not adequate for modeling general network traffic.

Self-Similar Traffic

Analysis of real-life network traffic traces [24] [40] have shown that the arrival of each packet in the network is not totally independent of the arrival of any other packet such as in a Poisson process, the arrival patterns display a degree of self similarity. The traffic pattern is repeated on smaller and larger time scales in accordance with fractal theory [45]. Several papers have been written on how to efficiently simulate such long-range dependencies in the network traffic. One of the findings in [70, 15, 53] is that such traffic can be modeled by a process with a finite mean and infinite variance. In [70] the authors show that an aggregation of Pareto distributed on/off sources with strictly alternating on and off periods are within the necessary mathematical criterion, with the proper α values between 1 and 2, to produce self similar traffic. To be within the necessary mathematical criterion in this respect means that the number-sequence generated by the Pareto-generator given an appropriate value of α has a finite mean value independent of the length of the number-sequence, and infinite variation [70].

A Pareto distributed on/off source produces alternating on and off periods of a Pareto distributed length. A formula to generate Pareto-distributed numbers on a computer is as follows:

$$X_{pareto} = \frac{b}{U^{\frac{1}{\alpha}}} \tag{6.1}$$

where U is a uniformly distributed value in the range (0,1] and b the minimum value of X_{pareto}. The parameter α controls the probability density function for the Pareto distribution. A small value of α (α not much greater than 1) returns a function with a higher probability of large values than a function with α closer to 2.

The theory in [70] states that the packet flow obtained from aggregating a number of Pareto on/off sources converges towards self-similarity as the number of sources generating the traffic flow and the time approaches infinity. In a simulation it is clearly impossible either to have infinite time or an infinite number of on/off sources for each traffic flow. Depending on the speed of the convergence towards self-similarity it might be possible to achieve a self-similar traffic flow within a limited amount of time and with a limited number of Pareto on/off sources. To investigate the self-similar property of a traffic stream generated within a limited time and only 10 Pareto on/off sources the plots in Figure 6.4 have been generated. Before the statistics were gathered the simulation was allowed to run for 100000 simulation cycles to allow the packet generation to stabilise. Each plot indicates the number of packets produced by a processor within a given timespan. The timespan of each figure is 10 times greater than that of the previous figure, each bar in the plots corresponds to the amount of packets produced in larger and larger time intervals. If one were using a normal exponential distribution such as Poisson to generate traffic one would see that the aggregate packet generation rates for the different sources would approach a mean value, with less variation in generation rate, as the timespan increases. The plots generated as we see them in Figure 6.4 do however not show the same tendency of leveling out as the timespan increases. The hight of the bars in plot 6.4(c) maintain the same high variability as the bars in plot 6.4(a) indicating a certain degree of self similarity in the traffic.

For simplicity the α values of the 10 on/off sources for each flow are equal and α_{on} is equal to α_{off}. When every Pareto-source used to generate a flow has the same α values there is a well-defined relationship between α and the Hurst parameter, or index of self similarity, H, described in equation 6.2 [70].

$$H = \frac{(3 - \alpha)}{2} \tag{6.2}$$

According to [22] a typical value of H is about 0.7 for naturally occurring time series such as Ethernet traffic. For this project $H = 0.75$ has been chosen giving $\alpha = 1.5$ for each of the 10 Pareto on/off sources generating traffic for each flow in the network.

The method used for producing self-similar traffic for the simulations in this project is as follows: Every flow has a packet generation engine consisting of 10 Pareto on/off sources, each producing packets at approximately one tenth of the flow's packet rate. The packets produced are deposited in a queue and are sent from the node at more or less regular intervals at double the packet rate of the flow to allow for the off-periods. This means that the peak rate of flows generated using Pareto on/off sources is double that of the flows' mean rates.

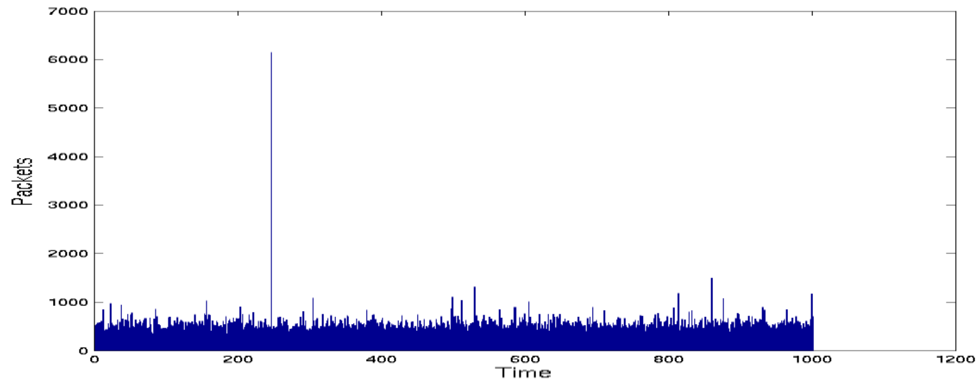
The main drawback of this method of traffic generation compared to the Poisson process described above is its infinite variability property. This means that for

relatively short time intervals, e.g. the time intervals used in the simulations to be presented in the next chapter, the mean value of the Pareto generated numbers show a great deal of variation. This means that the mean offered load of the flows also will vary leading to very few flows having the offered load specified for the simulations. As we shall see in section 7.2.2 this complicates the matter of finding out whether the flows receive their requested bandwidth or not.

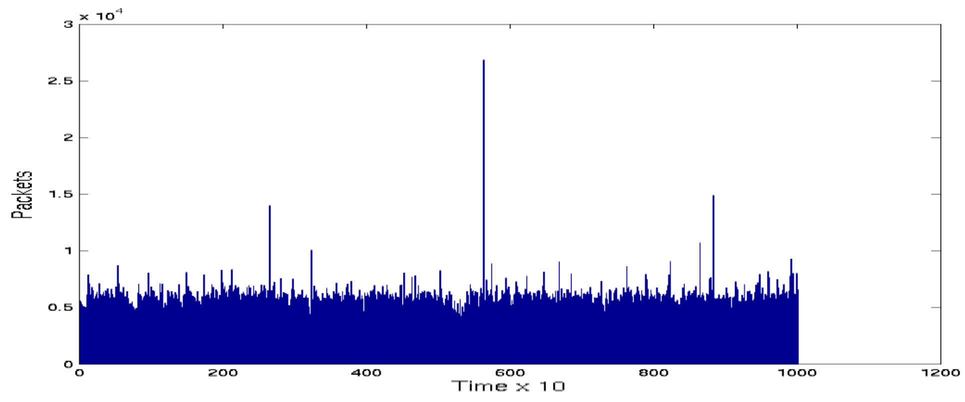
The reason for using self similar traffic despite the complications in analysing the results is the recent criticism regarding the lack of realism in simulations using a Poisson process for traffic generation.

Alternative methods for generating self similar traffic are described in [15, 42, 53, 52, 27].

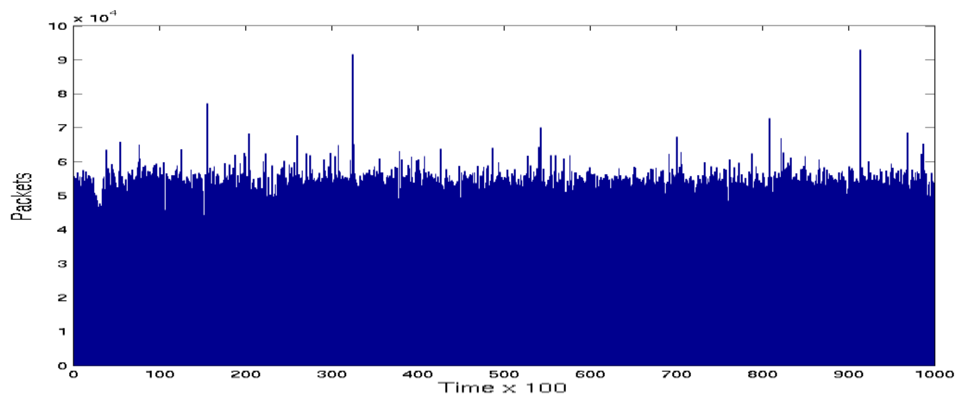
Jitter Measurements Using on/off Sources Normally jitter is defined as the inter-arrival time of packets. When using on/off sources the packet inter-arrival time may show very high variation when measuring two packets within one packet train compared to the last packet in one packet train and the first packet in the next packet train. Packet jitter will therefore be very much dependent on the packet generator and less dependent on the network architecture. To combat this, the jitter measurements in this project are based on the difference in packet transmission time, i.e. the time each packet spends in the network. In this way the only thing affecting packet jitter is the way in which the packets are treated as they traverse the network nodes, not the packet generation process.



(a) Packets produced at the normal timescale



(b) Packets produced at 10 times the normal timescale



(c) Packets produced at 100 times the normal timescale

Figure 6.4: Packet rate at an increasing timescale

Chapter 7

Results and Evaluation

In this chapter we are to evaluate the VCT admission control algorithms proposed by this thesis. Two types of experiments have been performed to evaluate different aspects of the admission control algorithms, namely the ability to give guarantees per SL and per flow. Some of the results from this thesis have been presented in [50] and [59]. In [50] we focus on the ability of the admission control algorithms to give throughput and latency guarantees *per SL* in conjunction with the Infiniband QoS mechanisms. In [59] the attention is turned towards achieving *flow level* guarantees using the same mechanisms as presented in [50].

The next sections will attempt to analyse the simulation results with respect to both SL guarantees and flow level guarantees. We analyse the admission control algorithms presented in chapter 5:

- Link-by-Link (LBL)
- Egress Measurements (EM)
- Jitter Probing (JP)
- no admission control for comparison

with respect to the evaluation points listed in section 4.1:

- network utilisation
- throughput guarantees
- latency
- jitter

7.1 Target for Admission Control

Recall from section 3.1 and [51] that as long as the network operates below the saturation point, each SL receives the throughput it requires and the QoS mechanisms have little significance. Thus by keeping the network below saturation each SL receives guaranteed throughput. Lets call the network throughput below which the SLs receive throughput guarantees for the admission control target (ACT). This point is where the throughput of some SLs are forced down by the QoS mechanisms in the network. To achieve high network utilisation and still be able to give throughput guarantees, the network throughput should be kept as close up to this point as possible. The ACT corresponds to the saturation point in Figure 7.1 where the network throughput is about 0.42 packets per simulation cycle.

7.2 Throughput/Network utilisation

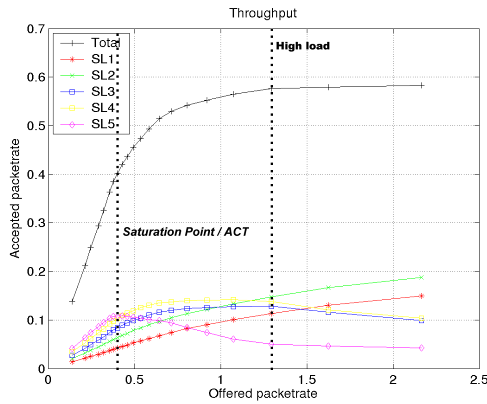
The term throughput may be divided into two categories. The first category is the total network throughput, or network utilisation. This information consists of the total number of packets arriving in the network for each SL divided by the simulation lifetime. This form of throughput may be used to determine the ability of the admission control algorithms to give throughput guarantees per SL, and the network utilisation. The other throughput category is flow throughput. This is the number of packets that have arrived at their destination for a flow in the network divided by the simulation time. This has not so much to do with network utilisation, but more with the Quality of Service received by each individual flow and is included to visualise the per-flow Quality of Service. These two categories are therefore described separately in the sections below.

7.2.1 Total Throughput

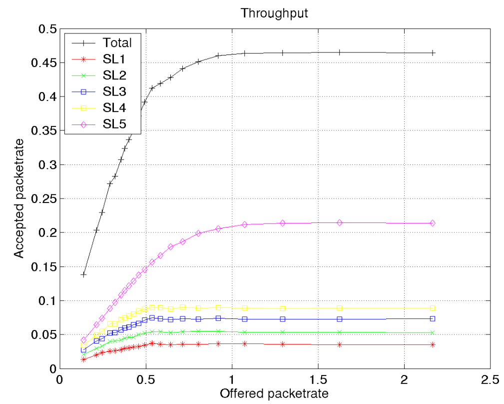
The results obtained from simulations using the two traffic distributions, random pairs and two hot-spots will be presented in separate sections below.

Random Pairs

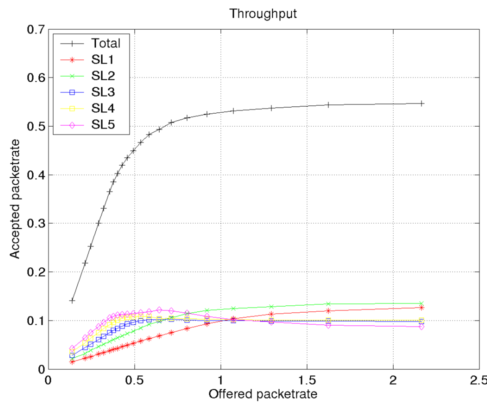
As stated in section 4.1.1 one of the goals of an admission control algorithm is to achieve high throughput whilst keeping the network below saturation. Figure 7.1 displays the throughput for each SL and the overall network throughput for each of the three admission control schemes tested. Additionally Figure 7.1(a) displays the throughput characteristics of a network without admission control. The x-axis in the figures is the load put on the network, the number of packets attempted



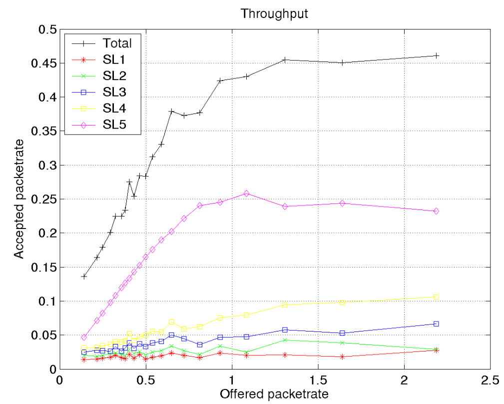
(a) No admission control



(b) Link-by-Link



(c) Egress Measurements



(d) Jitter Probe

Figure 7.1: Aggregated Throughput

injected into the network per simulation cycle. This is also the case for almost all the figures presented in this chapter. The y-axis is the achieved throughput in packets per simulation cycle.

The difference in the offered load and achieved throughput that may be seen in the figures may have one of two causes. In the case without admission control packets may be rejected by the sender if the send queue is full. When admission control is added flows will be rejected, forcing every packet to be produced by that flow to be rejected.

As can be seen in Figure 7.1(a) the network operates below saturation up to a point about midway between 0.25 and 0.5 on the x-axis marked by the saturation point. Up to this point the throughput increases linearly for each SL and for the total network. At about this point the low weight of SL 5 forces its throughput down by letting the other SLs have a greater share of the network bandwidth. At

about 0.5 the other two low priority SLs receive the same treatment and we see how the high priority SL 1 and SL 2 have high throughput at the expense of the low priority SLs. Note also that the maximum achievable network throughput without admission control equals a packet rate of 0.58 packets/cycle. This corresponds to a total network throughput of 46.1 Gbps.

This figure clearly shows how the high priority SLs preempt the low priority ones and how the SLs with high weight achieve better throughput than the SLs with low weight when the network is saturated. This indicates that the network is unable to give bandwidth guarantees. Instead of letting each SL have as much bandwidth as they desire as is the case below saturation, the network forces the throughput of the low-priority SLs down in favour of the high-priority SLs in the saturated state.

Moving on to Figure 7.1(b) which displays the throughput characteristics of the same network when LBL admission control is applied, we see that it is quite different from figure 7.1(a). Every SL except SL 5 is subject to admission control. As the offered load increases so does the throughput of every SL until the admission control starts rejecting flows. The admission control starts rejecting flows when the reservation of each SL has reached its target link utilisation. As the offered load continues to increase the admission control algorithm keeps the throughput on the controlled SLs constant. The throughput differentiation of the SLs is as specified by the algorithm even as the offered load increases. There is no hint of the service degradation seen in Figure 7.1(a), confirming that the network effectively is kept below the saturation point and thus giving bandwidth guarantees, at least on the class level. SL 5 continues to increase until it reaches its maximum possible throughput in the network and its packets are rejected in the sending nodes due to buffer overflow. The total network throughput lies around 0.46 packets/cycle, which is 22% lower than without admission control. The fact that the throughput is as high as it is, is mostly due to the large amount of best effort traffic in the network.

The LBL scheme bases its admission control decisions on reserved and available bandwidth, and it is therefore easy to force the method to differentiate between the different SLs. The EM scheme uses network latency as one of the key metrics and it is therefore difficult to achieve SL differentiation to the same degree. The throughput plot for EM can be seen in Figure 7.1(c). We see that this plot has more similarities with the plot without admission control than with the LBL plot. As the network load increases the network reaches, and passes, the point where the low priority SLs receive a degradation of service, their throughput is forced down. It is evident from figure 7.1(c) that the network is being admission controlled, but the admission control scheme is clearly not able to give bandwidth guarantees at this level. The network throughput is not as high as without admission control. An

interesting point is that as the load reaches its maximum, i.e. the offered packet rate increases, the throughputs of the high priority SLs are grouped together and the throughputs of the low priority SLs are close to each other. Recall that every SL requests the same latency guarantee from the network. The fact that the SLs in a priority level are grouped together regardless of the weight of the individual VLs, indicates that the VL weight does not affect the network latency, at least not to the same extent as whether it is a high- or low-priority VL.

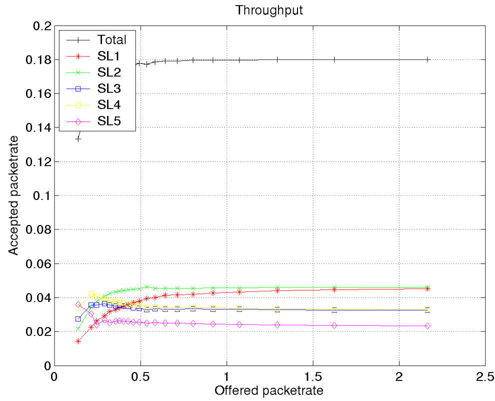
The total throughput in the case of JP is the lowest of all three admission control schemes, though it is almost as high as the LBL case. When looking at the throughput of the individual SLs we see that the throughput of SL 1 and SL 2 are about half of what they are in the LBL case. SL 3 and SL 4 on the other hand, slowly approach the same values that are present in LBL. The jitter requirements of the SLs were chosen from on the smallest values present in a unsaturated, uncontrolled network (see section 6.2.1). The relatively low throughput of the controlled SLs are caused by the fact that the jitter requirements are quite strict, forcing the algorithm to admit less traffic. Instead of the throughput of each SL flattening out as is the case of LBL, the throughput of every SL displays a more or less steady, though small, tendency to increase as the offered load increases. There is no sharp cut-off point as is the case with LBL and to some extent EM. The algorithm continues to admit traffic as the offered load increases, and we will see how this affects jitter in Section 7.4. Notice that the throughput of SL 5 is somewhat higher than in the LBL case, this can account for the relative high throughput despite the low throughput of the other SLs.

On a positive note we see that the throughput of each SL is differentiated in a manner consistent with the offered load on the different SLs and as dictated by the given jitter requirements. We may therefore conclude that the JP scheme is able to give bandwidth guarantees on a class level.

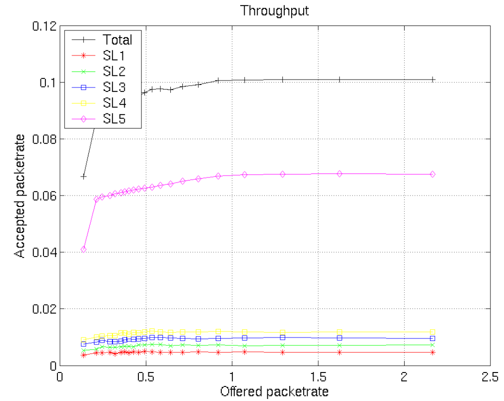
None of the admission control algorithms hit the ACT as well as they should. The throughput in every controlled network lies about 12% above what we defined as ACT. Despite of this, both LBL and JP are able to give class-based guarantees. In the latency and jitter results presented later however, the results might have been slightly better if the ACT was reached.

Two Hot-spots

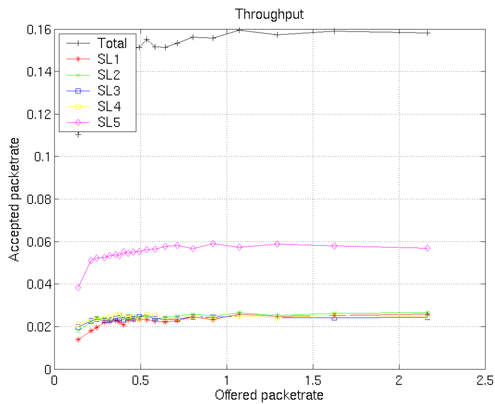
When changing the traffic distribution from random pairs to two hot-spots, the throughput characteristics of the network does, as expected, change. The throughput characteristics for all three admission control schemes in addition to the one without admission control is displayed in Figure 7.2. In the case without admission control, Figure 7.2(a), we see that instead of having the high-priority SLs steadily increase at the expense of the low priority SLs as we saw in the random traffic



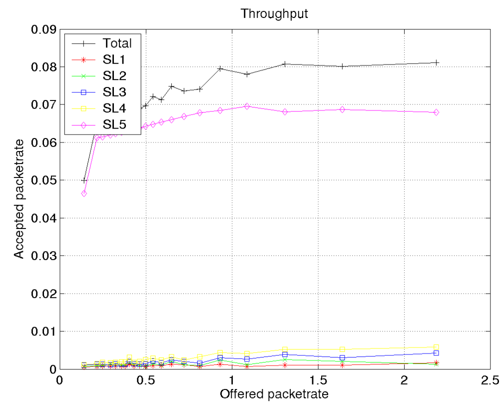
(a) No admission control



(b) Link-by-Link



(c) Egress Measurements



(d) Jitter Probe

Figure 7.2: Aggregated Throughput for Two Hot-spots

distribution, the throughput stabilises with the high-priority SLs SL 1 and SL 2 slightly above the low priority SLs SL 3 and SL 4, which again have four thirds of the throughput of SL 5. The high-priority SLs preempt the low-priority ones just as in Figure 7.1(a) and the network is saturated. The network is stabilised with a total throughput of 0.18 packets per cycle, with both SL 1 and 2 stabilising at about 0.042.

Both LBL (Figure 7.2(b)) and EM (Figure 7.2(c)) display quite different behaviour than without admission control. We see that the LBL scheme still is able to differentiate between the different SLs in the same way as it did using the random pairs traffic distribution. The differentiation is however not as distinct since the network throughput and the number of flows is significantly lower. The total network throughput achieved using the scheme is just above half as much as achieved without admission control.

The EM scheme does not display any differentiation between high priority and low priority SLs. Neither does it force the throughput of SL 5 down, the network throughput has not reached the saturation point, or ACT. The total throughput achieved using EM is higher than that of the LBL scheme, not too far below that with no admission control, the same as with random pairs. In other words, the network is kept below saturation, but the network does not seem to give any bandwidth guarantees. Bear in mind that every SL has the same latency requirement. If every SL had the same priority, we would expect no differentiation. The SLs do have different priority, but there still is no differentiation.

JP (Figure 7.2(d)) displays the lowest total throughput of all the figures. None of the controlled SLs are able to achieve much throughput, the algorithm seems to be a bit too strict when admitting flows. Despite the low throughput we are still able to detect a certain degree of SL differentiation indicating the ability to give bandwidth guarantees. Since all the network traffic is forced through a limited number of links there will be many more probe packets going through the bottleneck links than in the case of random pairs. This increased traffic will lead to larger jitter and thus fewer admitted flows. By easing the jitter requirements the overall throughput may be increased.

Summary

Of the presented admission control algorithms only LBL and JP are able to give bandwidth guarantees on a class level using the random pairs traffic distribution. When using two hot-spots traffic distribution network throughput is very low and it is therefore difficult to ascertain whether bandwidth guarantees are given. However it seems like only the LBL and JP admission control algorithms are able to give bandwidth guarantees in this case too. To summarise, the deployment of admission control in a network using either of the two traffic distributions discussed above leads to lower throughput on the controlled SLs and higher on the best effort. This might however not be a problem since the low throughput on the controlled SLs may lead to better latency and jitter characteristics, whilst the high throughput of the best effort SL helps in achieving a relatively high total network throughput.

Ranking the admission control schemes based on achieved throughput we get the following table (table 7.1). It is interesting to note that the rate pertaining to throughput is almost inverse of the rank pertaining to bandwidth guarantees. This is by no means the final rank of the admission control algorithms. As will become apparent in the next sections there are several other parameters that will influence the rank of the admission control schemes. Note that this ranking system is not provided to name a winner among the admission control algorithms, rather as a means of organising them as to their respective qualities.

AC	Throughput	Bandwidth Guarantees
No Admission Control	1	4
Egress Measurements	2	3
Link by Link	3	1
Jitter Probing	4	2

Table 7.1: Admission Control Schemes Sorted by Network Utilisation

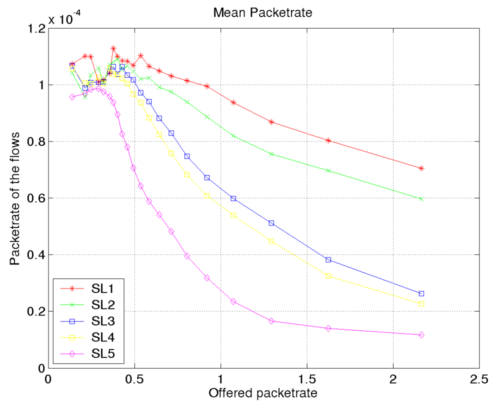
7.2.2 The Throughput of Each Flow

The quality of service at the class level does not represent well how the individual flows in the SL are treated. A SL may consist of several flows which receive good treatment and several flows which are treated badly, all the individual flows in the SL may receive highly variable Quality of Service, or all flows in the SL may receive almost the same treatment corresponding to the SLs QoS. This last case where each flow is treated equally within a SL is what we hope is the reality. To find out if this is the case, Quality of Service received by individual flows has to be studied.

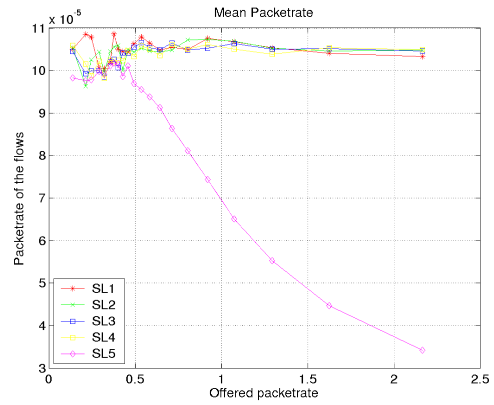
In an attempt to visualise the throughput of each flow in the network the following plots have been generated (Figure 7.3 for the random pairs traffic distribution and Figure 7.4 for the hot-spot traffic distribution). The figures depict the aggregation of the mean rate of flows in their respective SLs, e.g. the average packet rate of all the flows in the SLs. As mentioned in Section 6.2, each flow may have one of two packet rates. Only the results for the flows with a packet rate of 0.0001 packets per cycle are presented. The plots for the flows with a packet rate of 0.0002 packets per cycle show the same behaviour as the figures included here. As before, the x-axis represents the offered load in packets per simulation cycles. The y-axis represents the average achieved packet rate for the flows in the SLs. Ideally the achieved packet rate should lie around 0.0001 packets per cycle which is the configured packet rate for the flows presented in the plots.

Random Pairs

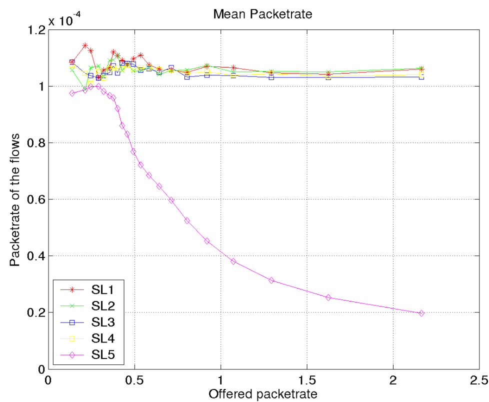
The simulations operate in an environment with limited buffer space. Specifically the size of the send queue in the processor is set to two packets as described in Section 6.2.3. As the network reaches saturation the queue buildup in the network will force the send queue buffers at the processors to become full and subsequent packets will be rejected due to insufficient buffer space. This will again force the throughput of each flow downwards and may therefore be used as an indication of network saturation and the ability to give bandwidth guarantees.



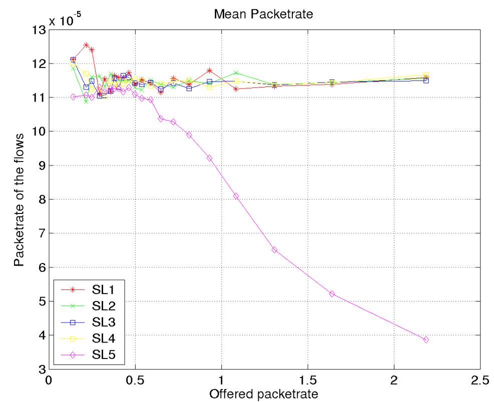
(a) No admission control



(b) Link-by-Link



(c) Egress Measurements



(d) Jitter Probe

Figure 7.3: Mean per flow rate

Figure 7.3(a) is a plot of the mean rate for every flow in their respective SLs with no admission control. We see that the curve for every SL drops from its initial value of about 0.0001 with a speed corresponding to the different SLs relative priority. SL 5 is the first to drop and drops the fastest while SL 1 and SL 2 hold on the longest. The downward tendency of the curves indicates that each flow gets fewer and fewer packets through the network, or alternatively that fewer and fewer flows get enough packets through the network whilst the other flows get full service. In any case, there are more and more flows added whilst the network bandwidth remains the same. This figure may not be used to determine which is the case, to this purpose table 7.2 is provided. The table indicates the percentage of flows which get 100% of their offered traffic through the network. As we can see the numbers for the case without admission control are not impressively high.

SL	NoAc	Link-by-Link	Egress	Probe
1	51.6	97.6	77.5	98.5
2	41.0	98.1	77.5	98.3
3	7.7	97.7	64.9	98.1
4	2.7	96.2	66.7	98.4
5	0.0	0.95	0.095	1.4

Table 7.2: Percent of flows with full throughput with different AC

It is worth noting at this point that although SL 2 has a larger weight than SL 1, giving SL 2 a higher relative priority than SL 1, SL 2 shows worse behaviour than SL 1. This is caused by the fact that SL 2 has a higher percentage of the offered load than SL 1, giving it the ability to insert more flows into the network. The increased priority of SL 2 vs SL 1 seems however not to be enough to handle the increase in the number of flows. The main conclusion to be drawn from this figure is, however, that without admission control none of the SLs are able to provide all their flows with the bandwidth they initially requested.

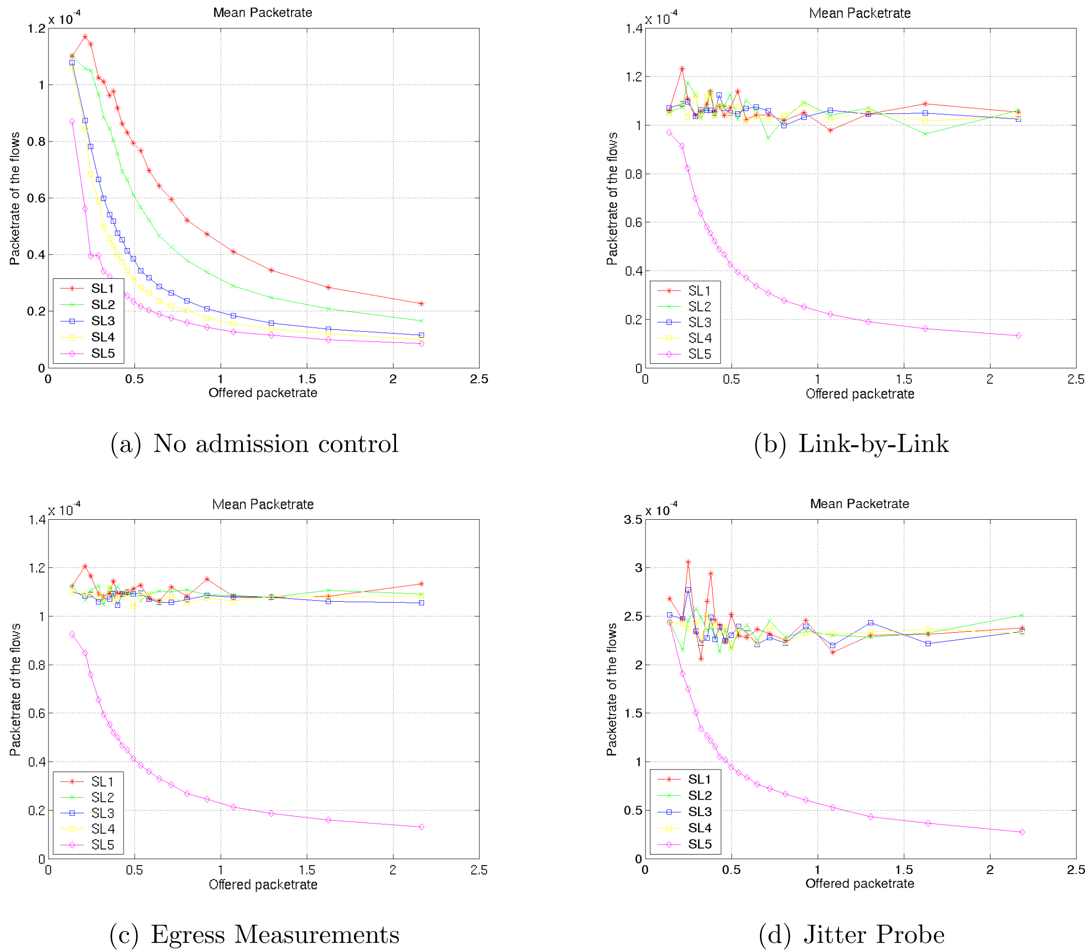
The other three plots in Figure 7.3, figures 7.3(b),7.3(c) and 7.3(d), display a markedly different behaviour, but are quite similar to each other. All three plots display a more or less constant bandwidth for SL 1 through SL 4. SL 5 on the other hand, which is not subject to admission control, plummets in much the same way as it does in Figure 7.3(a).

Despite the similarity there is a marked difference in the percentage of flows able to achieve full throughput. The numbers for egress scheme in Table 7.2 are certainly much better than the values without admission control, but they are still worse than both LBL (table 7.2) and the JP scheme which allows almost every flow full throughput, even for SL 4, despite the fact that SL 4 has laxer jitter requirements than SL 1.

All the four plots of Figure 7.3 exhibit more or less noisy curves. That is, there are few straight/smooth lines, the different points on a curve seem to jump a bit erratically up and down. This is due to the random nature of the self similar packet generation. The difference in level of “noise” between the different SLs and between the different plots is caused by the difference in the amount of flows admitted in the SLs. A large number of flows will produce a more stable value for mean flow throughput than a smaller number of flows.

Two Hot-spots

As we saw in Section 7.2.1 regarding the network throughput using two hot-spots as traffic distribution, the overall network traffic is quite small. This is worth



(a) No admission control

(b) Link-by-Link

(c) Egress Measurements

(d) Jitter Probe

Figure 7.4: Mean per flow rate for two hot-spots

bearing in mind when introducing the rate plots using the same two hot-spots traffic distribution, Figure 7.4.

For JP in this case, we have a situation where only the high bandwidth flows were accepted in the network. None of the flows with the lower packet rate were admitted. The figure presented in Figure 7.4(d) therefore displays the data of the flows with a packet rate of 0.0002. The figure shows that the admitted flows seem to be able to send at their specified rate as is the case with the random pairs scheme.

Comparing the rate plots using two hot-spots and no admission control (Figure 7.4(a)) with the corresponding figure using the random pairs traffic distribution (Figure 7.3(a)) we see that the mean flow rate has a far more drastic decrease for two hot-spots. This is expected since the same amount of traffic is sent through a

fewer number of links.

The LBL and JP schemes are however quite similar to the corresponding plots using the random pairs distribution, the only difference being a larger variation in rate due to the smaller number of flows. In the case of EM on the other hand, Figure 7.4(c), although they too seem quite similar, there seems to be a small tendency to a negative slope on the curves. This indicates that the admission control scheme is not quite able to guarantee each admitted flow its requested bandwidth.

Summary

The results in the previous section showed that some of the different admission control schemes were able to differentiate the throughput of each SL, based on the parameters given to the admission control algorithm. This meant that the admission control schemes were able to give bandwidth guarantees on a class level. In this section we have seen that two of the admission control schemes are able to give each individual flow bandwidth guarantees. No matter which one of the two packet rates used in the simulations a flow requested, the flow is given the requested throughput in the network.

In the EM scheme only 77% of the flows were able to achieve full throughput. We can therefore not say that the scheme was able to give bandwidth guarantees. The guarantees of the other two admission control mechanisms are, however, not 100%. As described in the tables presented in this section there was a certain percentage of flows that were unable to get every packet sent through the network. 100 percent guarantee was not even achieved with the LBL scheme which has detailed knowledge of each link in the network. The results are nonetheless quite uplifting as they are achieved in the network with only class based Quality of Service mechanisms. Table 7.3 ranks the admission control algorithms based on the ability to achieve flow level bandwidth guarantees.

Rank	AC
1	Jitter Probing
2	Link-by-Link
3	Egress Measurements
4	No Admission Control

Table 7.3: Admission Control Schemes Sorted by Bandwidth Guarantees

7.3 Latency

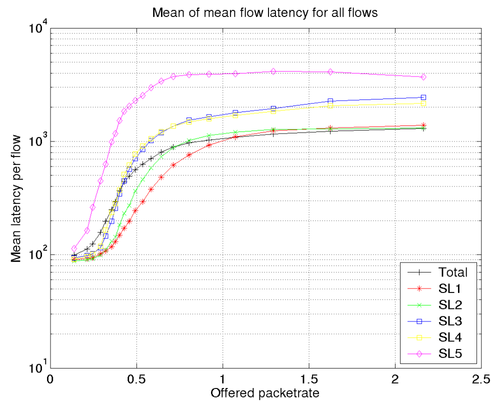
This section presents the network latency as it is observed with the different admission control algorithms. The figures presented depict the mean of the mean latency of every flow in the SLs. These figures are only slightly different from the figures depicting mean SL latency which have therefore not been included here. The figures are not able to give us as detailed information about each flow as we could with flow level throughput, but they give a fair indication of the latency characteristics of the different admission control algorithms. The y-axis in the figures represent the latency in simulation cycles.

Of the algorithms evaluated, only EM has any direct relationship with the network latency. One would therefore expect EM to be a good method for controlling network latency, but as we shall see in this section, this may not be the case.

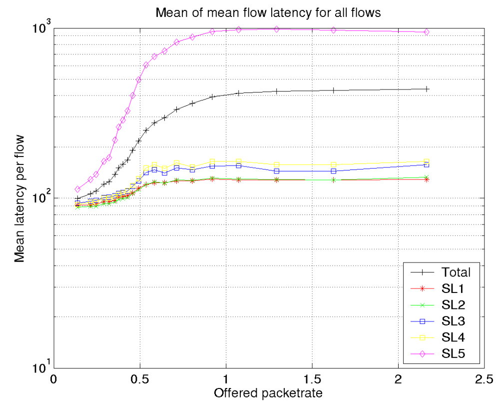
7.3.1 Random Pairs

The mean of each flows mean latency is displayed in Figure 7.5. To start with Figure 7.5(a), a plot of the system without admission control, we see that the latency of every SL, except SL 5, starts to increase around the point where the network reaches saturation, about midway between 0 and 0.5. This strengthens the claim that the admission control algorithms should not allow the network to reach saturation. We see that the latency of SL 5 starts to increase drastically almost immediately and stabilises at a higher value than the other SLs. This is consistent with the high load and low weight given to the SL. Also note that the latency of the SLs in the same priority arbitration table, e.g. the SLs in the high-priority table and the SLs in the low priority table, seem to converge to the same value. This indicates that although the arbitration weights control the throughput of the different SLs as we see in Figure 7.1(a), they have little influence on the latency experienced by the SLs at the same priority level. This may also indicate that the SL load and VL weight matches each other perfectly, except for SL 5, but this is somewhat unlikely, there is a linear increase in load through the SLs, but the VL priority is not increased linearly form SL 1 to SL 4.

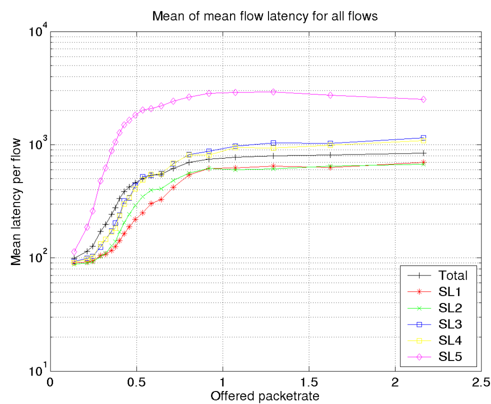
Moving on, we see that the latency characteristics of the LBL scheme as seen in Figure 7.5(b) are much improved. SL 5, which is not subject to admission control, displays a marked rise, though not to the same extent as SL 5 in Figure 7.5(a). The other SLs, SL 1 through SL 4, display an almost constant latency as the load increases. There is a small increase from around 90 to just above 100 simulation cycles at about load 0.5 from which the latency is constant. Although the SLs were well differentiated with regards to bandwidth as we saw in Figure 7.1(b), there does not seem to be much differentiation, apart from between priority levels as with no admission control, with regards to latency. This goes further to indicate



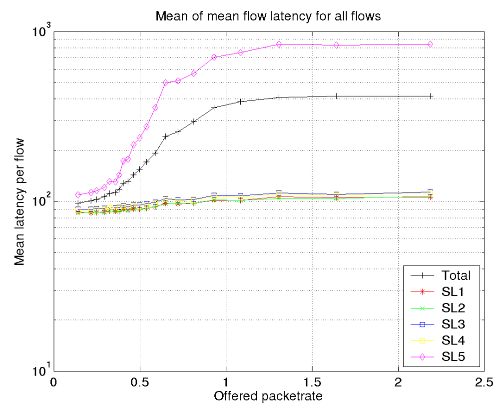
(a) No admission control



(b) Link-by-Link



(c) Egress Measurements



(d) Jitter Probe

Figure 7.5: Mean of mean flow latency

that the VL weight has little influence on network latency at high loads.

The EM scheme in Figure 7.5(c) displays a behaviour which is not in accordance with what one would expect given the admission control parameters. SL 1 and SL 2 are grouped and stable around 700, SL 3 and 4 are grouped and a bit less stable around 1000 and SL 5 with no admission control increases rapidly as in the other plots. For SL 1 and SL 2 we see that the latency is above 100 which was the basic requirement, and even above 200, the requirement for the longest paths. From this it can be concluded that the EM scheme is unable to give latency guarantees, even on high-priority SLs. A question may be raised, is a latency requirement of 100 realistic? As we saw in Figure 6.1 in Section 6.2.1, the mean of the mean flow latency for SL 1 in an uncontrolled network below saturation ranged from 90 to about 210 as the number of hops increased. It should therefore be possible

to achieve the same network latency by limiting the network traffic through the admission control. To achieve this with EM the latency requirements may possibly have to be mapped to lower symbolic latency requirements for use in the algorithm.

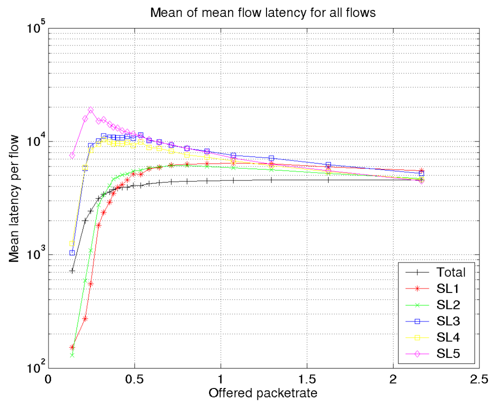
As with throughput the JP scheme displays much the same behaviour as LBL with regards to latency. The latency is slightly lower for SL 1 to SL 4 and the differentiation between the SLs is less prominent. We see the same tendency towards a convergence of the latencies of the high- and low-priority SLs. The lower latency and lack of differentiation with JP compared to LBL is most likely due to the fact that JP has slightly lower throughput than LBL. The lower throughput means fewer packets and less probability of being forced to wait during transmission of a packet through the network.

7.3.2 Two Hot-spots

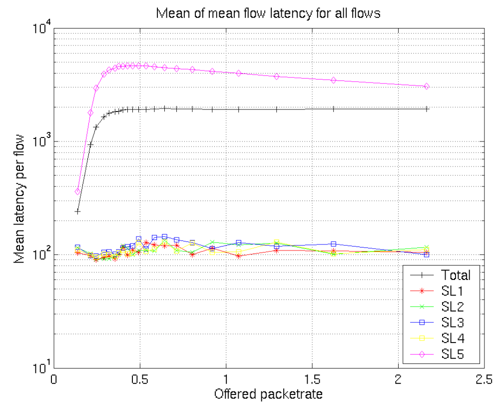
When switching to the two hot-spots address distribution it is evident from the plots in Figure 7.6 that the latency of the strictly controlled SLs decreases while the latency of the uncontrolled SLs increases. This means that for the LBL admission control scheme in Figure 7.6(b) SL 1 to SL 4 perceives a slight decrease in latency whilst SL 5 experiences much higher latency. The same goes for JP in Figure 7.6(d). The EM in Figure 7.6(c) shows the same behaviour for SL 1, SL 2 and SL 5, but SL 3 and SL 4 are almost unchanged. The uncontrolled SL, SL 5, shows worse behaviour since the same amount of traffic as with random pairs is trying to cross a limited number of links. Thus there is more traffic in each link and the latency worsens. The admission controlled virtual lanes, on the other hand, experience better latency because the limited number of links that all the traffic has to traverse force the admission control algorithm to stop admitting flows at an earlier time. There is less traffic and thus lower latency.

All of the admission control algorithms are able to give the same, or even better, latency guarantees to flows in a hot-spots traffic scenario. This may be explained by the fact that there are only a small number of links determining the amount of traffic to be allowed in the network and it is therefore easier to get an adequate picture of the state of the network for the admission control decision. One can further surmise from this that the algorithms with lower latency, especially LBL and JP, are able to adapt their admissions to the bottleneck links in the network.

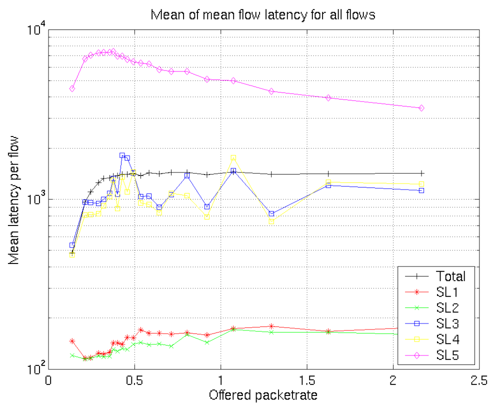
JP displays exceptionally good latency characteristics for the controlled SLs. This is as mentioned due to the fact that the algorithm is very strict when admitting flows and does not achieve very high throughput.



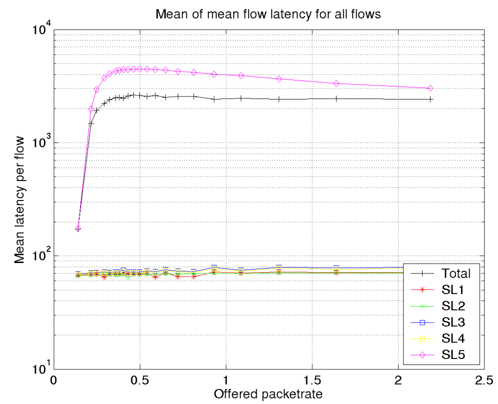
(a) No admission control



(b) Link-by-Link



(c) Egress Measurements



(d) Jitter Probe

Figure 7.6: Mean of mean flow latency for two hot-spots

7.3.3 Summary

As we have seen here, the addition of admission control in a virtual cut-through switched network is able to significantly reduce the average latency experienced by the admitted flows for different address distributions. Admittedly not all the admission control algorithms are able to reduce the latency by much, the EM scheme was unable to reduce the latency much and does not achieve the requested latency bounds. The situation was better for this and all the other admission control algorithms using the two hot-spots address distributions, but not even then was EM able to uphold the latency guarantees. As discussed in section 3.3.2 the nature of virtual cut-through makes it difficult to use measured latency as an indication of the network capacity, as EM clearly demonstrates. However, it should be possible to strengthen the latency requirements to for example 20 simulation

cycles instead of 100 for SL 1. One may then expect much improved results, but a latency requirement of 20 is unrealistic since the shortest time a packet uses in the network simulated here is about 60 simulation cycles. This requires a mapping between actual latency demand and the latency parameter given the algorithm.

Rank	**AC**
1	Jitter Probing
2	Link by Link
3	Egress Measurements
4	No Admission Control

Table 7.4: Admission Control Schemes Sorted by Latency

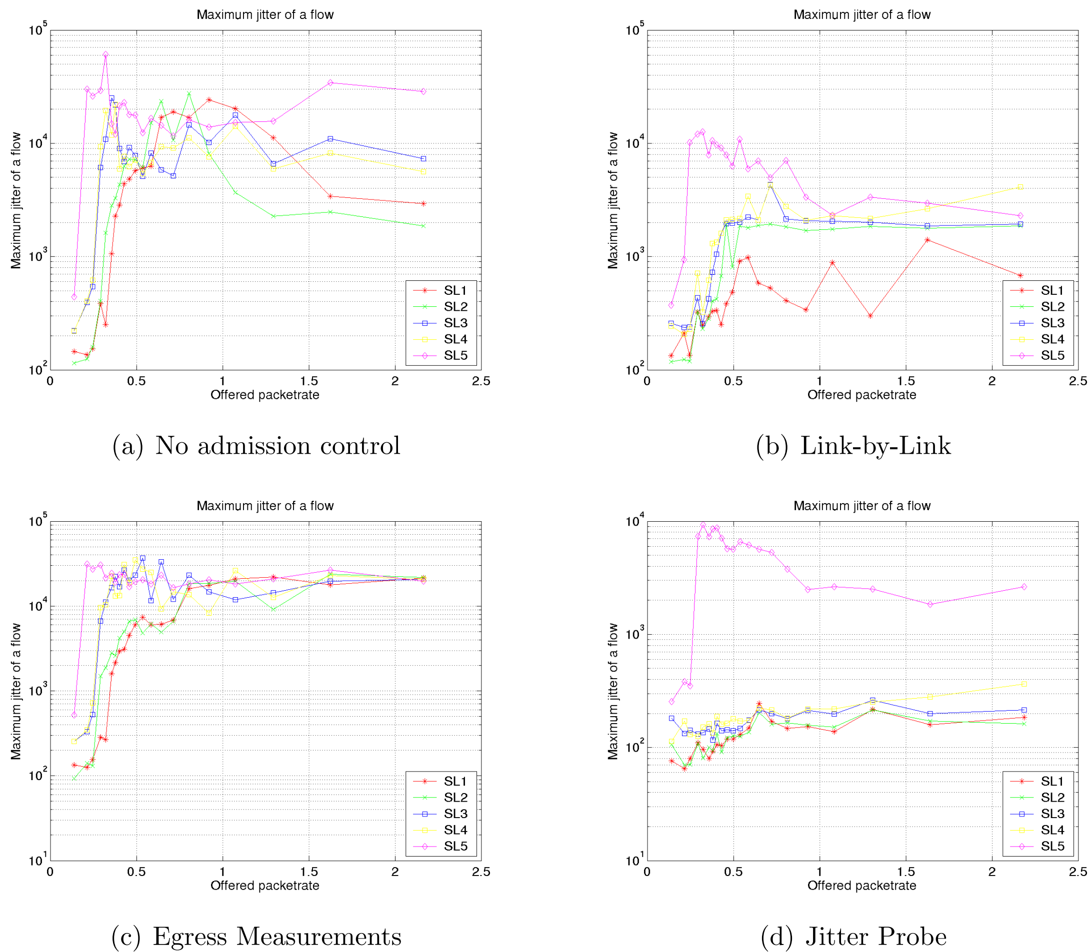
We see that there is a tight relationship between throughput and latency. The higher throughput of a SL the larger will the average latency for that SL be.

The ranking of best latency in Table 7.4 is the exact opposite of the best throughput ranking in Table 7.1. In other words, it may be possible to decrease throughput to obtain better latency. The question is however, how much one is willing to reduce throughput to achieve slightly better latency characteristics.

7.4 Jitter

Recall that jitter is the difference between the transmission time of two packets in a flow. The jitter of a flow with only one packet is therefore 0. The maximum jitter of a flow is the difference between the minimum and maximum transmissions time of the flows packets. As the number of packets in a flow decreases so does the probability of high variations in the transmission times, and the jitter therefore decreases.

The jitter plots in this section display the maximum jitter of all flows in the given SL. The average jitter of all the flows in the different SLs is somewhat lower and more regular, i.e. less chaotic, than the plots shown here. Although the average jitter plots may be more informative as to what generally happens to the jitter in the network using the different admission control schemes, it is when looking at the maximum jitter we see the most interesting differences. This will be especially noticeable when we come to the results from the packet dropping scheme at the end of this chapter. The plots for average jitter are presented as Figures A.1(Random pairs) and A.2(two hot-spots) in appendix A.



(a) No admission control (b) Link-by-Link

(c) Egress Measurements (d) Jitter Probe

Figure 7.7: Maximum packet jitter for a flow

7.4.1 Random Pairs

The jitter plot without admission control in Figure 7.7(a) is rather chaotic. The jitter curves of the different SLs cross each other and the overall jitter seems to decrease towards maximum load. There is no clear distinction between the maximum jitter of the different SLs, but for most of the time the high-priority SLs have the lowest maximum jitter. If one were to look at the average jitter plots without admission control (not shown here, see appendix 8.1) one would see that the average jitter for all SLs peaks at about a offered packet rate of 0.6 with an average jitter lying around 500 cycles, with SL 5 at about 1000 cycles. The interesting thing to notice is that as the offered load increases further the average and maximum jitter starts decreasing. As the offered load increases, the number of packets each flow gets through the network decreases, leading to lower

probabilities for large latency variation. This effect is most noticeable on SL 5 and less noticeable upward towards SL 1 (see Figure 7.3(a)). As the number of packets for each flow decreases so does the number of jitter measurements.

The deployment of LBL admission control scheme decreases the maximum jitter by about 1 order of magnitude. In Figure 7.7(b) we see that SL 2 to SL 4 experience roughly the same amount of jitter, about 3000 simulation cycles, while SL 1 is somewhat below and SL 5 is somewhat above. The average jitter in this case is much better. SL 1 and 2 stabilises with jitter between 60 and 80 while SL 3 and 4 stabilises around 150.

The maximum jitter of the EM scheme in Figure 7.7(c) resembles the corresponding plot without admission control. The jitter increases rapidly to about 20000 and stabilises, though with less variation than without admission control. This means that the EM scheme actually displays worse jitter characteristics than the case without admission control. Again, this stems from the fact that without admission control each flow has few packets through the network and thus there is a smaller probability of large variation in latency. The average jitter plot is similar to that of LBL, but the stabilising values for SL 1 and SL 2 are about 400 whilst SL 3 and SL 4 increase to 900.

JP in Figure 7.7(d) displays the lowest maximum jitter of all the presented admission control algorithms. This is as one should expect since JP bases its admission control decision on measured packet jitter in the network. On the other hand, EM was not able to provide good latency characteristics despite the fact that it bases its decisions on measured network latency. The maximum jitter for every SL, except SL 5, stabilises at a value between 100 and 200, migrating towards 200 as the offered load reaches its maximum. As with latency there is no real differentiation between the different SLs, most probably since all the probe packets are sent on the same SL and thus receive the same service. Despite the low jitter, neither the average nor the maximum jitter is as low as the requirements stated in table 6.3 in section 6.2.1.

Even though the throughput of each SL increased with the offered load, we do not see an increase in maximum jitter to the same degree. The algorithm is in other words able to keep the maximum packet jitter stable with different network loads. There is however a small increase in the average jitter, the algorithm is not perfect.

Even though the maximum jitter for LBL, and to some degree JP, seems quite high, we see from Table 7.5 that the mean and stddev for these algorithms are quite low. Table 7.5 presents the mean jitter and standard deviation for SL 1 in a network at high load (indicated with the "high"-marker in figure 7.1(a)) for flows with a hop length of 3 hops. 3 hops is chosen since the highest number of flows have this path length. The table clearly shows that LBL and JP are able to reduce

jitter, and that EM actually is worse than no admission control.

AC	Mean	Std. deviation
No Admission Control	392	600
Link by Link	79	39
Egress Measurements	490	898
Jitter Probing	62	21

Table 7.5: Mean and standard deviation for SL 1, 3 hops at a offered packet rate of 1,3 packets per cycle (high load)

7.4.2 Two Hot-spots

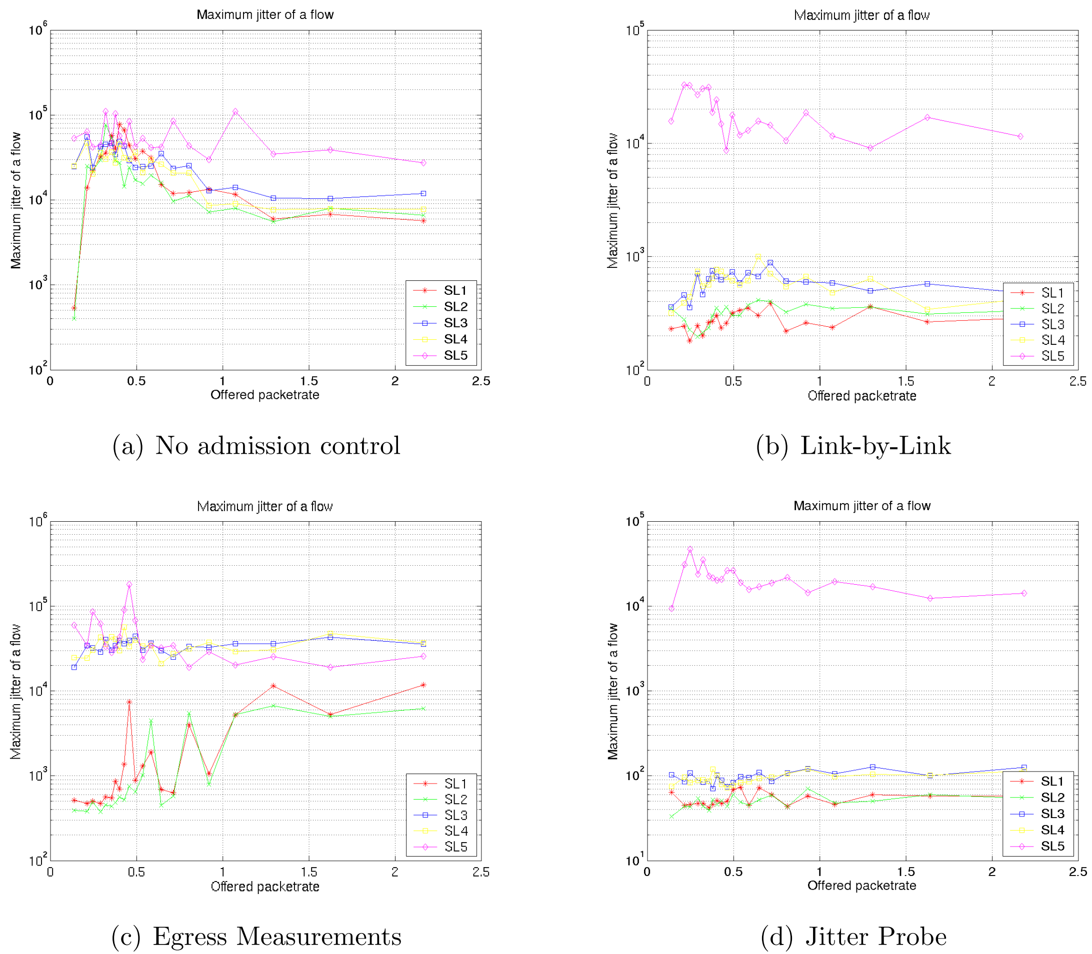
Without admission control (Figure 7.8(a)) the maximum jitter with two hot-spots is not much different from the case with random pairs. The maximum jitter shows the same high values, but there is less variation between the measurements. With the LBL scheme (figure 7.8(b)) the maximum jitter for the controlled SLs decreases as expected given the lower latency. The same goes for EM (figure 7.8(c)) though not at all to the same extent.

JP displays in this case the absolutely lowest maximum jitter observed in this project. Again this is caused by the low throughput achieved caused by the behaviour of this admission control algorithm when using two hot-spots.

As expected, lower latency of the controlled SLs leads to lower maximum jitter for the same SLs.

7.4.3 Summary

As stated in Section 4.1 jitter and latency is closely coupled. Low packet latency leads to low jitter since the variation in latency decreases as the maximum latency decreases. By using admission control algorithms to reduce throughput and thus packet latency it is possible to achieve jitter characteristics that are significantly better than in an uncontrolled network. For SL 1 using JP the average flow jitter is less than 30% of the average flow latency. For the worst case the jitter lies within 100% of the average latency. Although the results are better than without admission control, the jitter is still quite high relative to the latency. In conclusion, very low jitter seems very hard to achieve in VCT networks.



(a) No admission control

(b) Link-by-Link

(c) Egress Measurements

(d) Jitter Probe

Figure 7.8: Maximum packet jitter of flows for two hot-spots

7.5 Throughput/QoS Trade-off

The results that have been presented in this chapter demonstrates that, in the general case, lower throughput leads to lower latency. To find out to what degree this is the case, how low latency we can achieve by reducing throughput, experiments have been performed with different network utilisation targets for the LBL scheme.

In Figure 7.9 the LBL scheme is shown ranging from very strict admission control to not so strict. The purpose is to make clear the connection between network throughput and QoS, and to see how good QoS guarantees we are able to get from the network at the expense of network throughput. The values on the x-axis represents the level of theoretical link capacity used by the admission

Rank	AC
1	Jitter Probing
2	Link by Link
3	No Admission Control
4	Egress Measurements

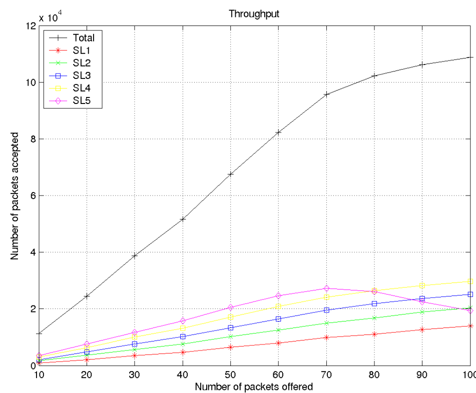
Table 7.6: Admission Control Schemes Sorted by Jitter

control algorithm ranging from 10 percent link utilisation to 100 percent link utilisation. Recall that the results from using the LBL admission control scheme presented previously are obtained with a link utilisation of 50%. In Figure 7.9(a) we see that the network throughput increases linearly to about 65 percent link utilisation where SL 5 starts to experience poor Quality of Service with regards to throughput. Similarly in Figure 7.9(b) the latency of each SL is quite stable until about 50 percent link utilisation where the average flow latency starts to increase. The maximum jitter plot (figure 7.9(c)) displays similar behaviour, only with the difference that the maximum jitter starts to increase at about 40 percent network utilisation. It is clear from these figures that there is a definite lower bound on the latency guarantee that can be given to a set of flows. This lower bound is closely related to the path length, fewer hops will give lower latencies. The lower bound on jitter is not as clear, it is not so closely related to path length. There is however a point below which a further reduction in throughput is not as effective.

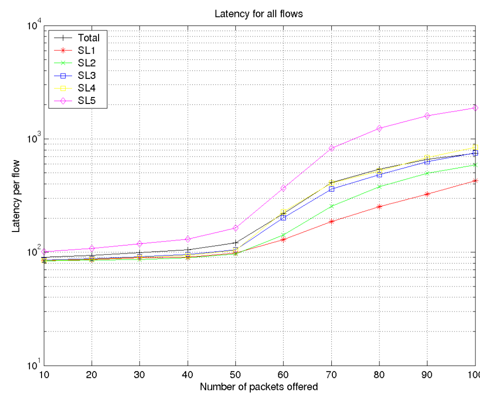
All this shows that although reduced throughput may lead to better latency and jitter characteristics there seems to be a limit as to how far it is effective to reduce throughput through the use of admission control. Beyond this point other methods must be found to reduce the latency (if possible) and especially jitter further.

7.6 Achieving Low Jitter

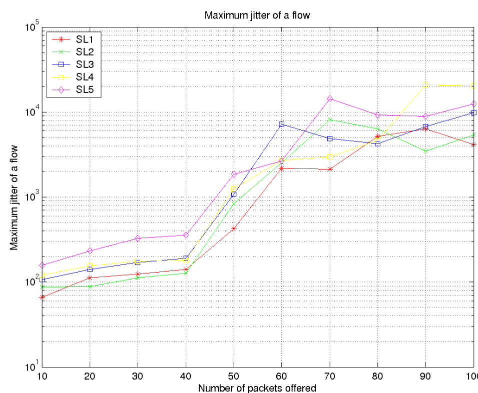
The dropping of packets on a SL has been proposed in section 5.4 as a method to further reduce jitter on that SL. To save space only two versions of packet dropping have been included. First there is packet dropping without admission control where packets on SL 1 are dropped if they are held back in the switches by the flow control mechanism. Second is the same setup only with LBL admission control. As can be seen in Figure 7.10 there is not much difference in throughput with and without (Figure 7.1 in Section 7.2.1) dropping packets. In the case without admission control the total network throughput is 11 percent lower than when not dropping packets. For the LBL scheme the total network throughput



(a) Link-by-Link TP



(b) Link-by-Link latency

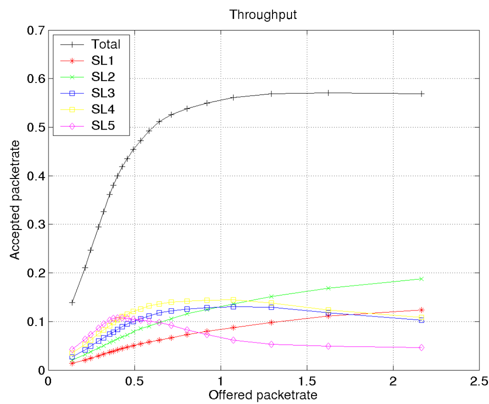


(c) Link-by-Link maximum jitter

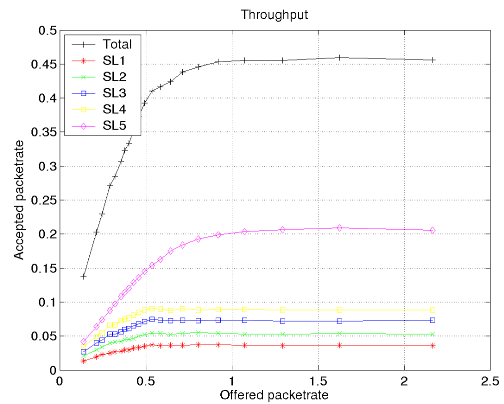
Figure 7.9: Trade-off between QoS and network utilisation

reduction is only two percent. The main difference that can be observed is that the throughput of SL 1 with no admission control and packet dropping in Figure 7.10(a) is not quite as high as the throughput of SL 1 without packet dropping.

Turning our attention to jitter (figure 7.11), we see a markedly greater difference, especially in the case without admission control. The average jitter in this case is forced down to just below 100 which is 75% lower than in the case without packet dropping where the average jitter of SL 1 lies around 4-500. But it is not in average jitter we see the most marked difference with packet dropping. Although the average jitter of the LBL scheme is almost unaltered when introducing packet dropping, when comparing the maximum jitter with packet dropping (Figure 7.11) with the maximum jitter without packet dropping (Figure 7.7 in Section 7.4.1) we see that there is a drastic decrease, 75%, in the maximum jitter perceived by any flow in the network for SL 1. In other words, the introduction of packet dropping



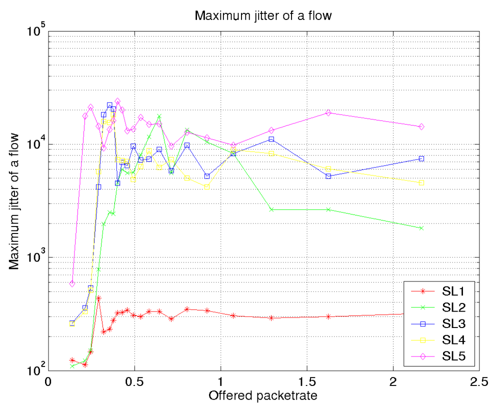
(a) No admission control



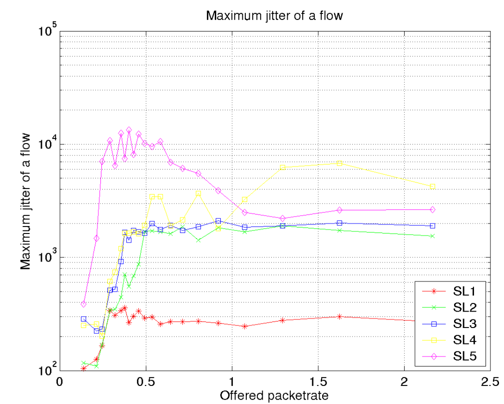
(b) Link-by-Link

Figure 7.10: Throughput with packet dropping

on SL 1 leads to a smaller variation in network latency and thus lower packet jitter. The maximum jitter of LBL is still higher than the maximum jitter achieved with JP, but not by much. When one takes the double throughput of SL 1 in the LBL scheme with packet dropping compared with SL 1 in JP into account, this small degradation in maximum jitter for SL 1 does not seem to have great significance. Halving the throughput rate of SL 1 is quite a price to pay for reducing maximum jitter by 20%, but depending on the application it might be worth-while.



(a) No admission control

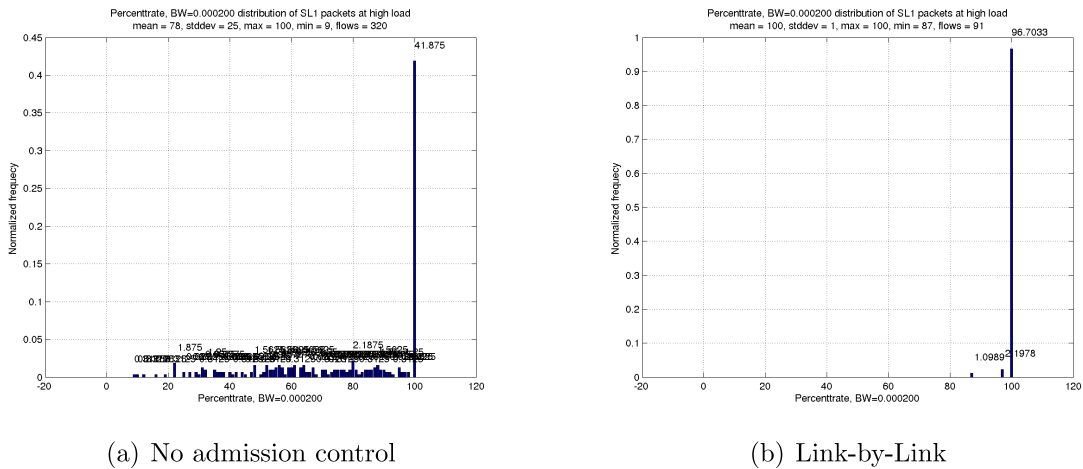


(b) Link-by-Link

Figure 7.11: Maximum packet jitter with packet dropping

It remains now to be seen at what cost this reduction in maximum packet jitter has been achieved. Figures 7.12(a) and 7.12(b) depict the distribution of

flows based on the amount of packets they were able to send through the network compared to the amount of packets that they tried to send. The height of the bar in the plot indicates the percentage of flows which were able to send a specific percentage of packets through the network. As we see in Figure 7.12(a) for the case without admission control, only 41% of the flows were able to propagate all their packets through the network without any of them being dropped by the network. A few flows did not even get 10% of the traffic through the network. LBL on the other hand, in Figure 7.12(b), is able to ensure that 95 percent of the flows get all their packets through the network. This is a small decrease of 2% compared to the value for SL 1 (around 97%) listed in Table 7.2 in section 7.2.2 for the case without packet dropping. As opposed to the plot without admission control the smallest percentage of packets a flow was able to successfully send through the network was about 85 percent.



(a) No admission control

(b) Link-by-Link

Figure 7.12: Distribution of how many packets flows have successfully sent through the network

To summarise, packet dropping is not allowed in VCT networks, but it seems that packet dropping can be a useful tool to provide better jitter characteristics in a VCT network without the necessity of reducing network throughput too much. It also seems that including packet dropping with a form of admission control gives the ability to provide this benefit without the cost of dropping too many packets, and without having to endure delays in the admission control decision while probing the network. The jitter is however still not lower than when using JP, it still remains to find a way of further reducing jitter.

7.7 Concluding Remarks

This section presents some concluding remarks regarding the performance and usability of the evaluated AC algorithms. Based on the tables presented in this chapter it is apparent that LBL and JP are the algorithms that are able to give best guarantees. EM is not able to give either bandwidth or latency guarantees and is not a viable approach in this scenario. LBL achieves higher throughput than JP and has better bandwidth differentiation, while JP displays lower latency and jitter. Additionally packet dropping with LBL displays low jitter and relatively high throughput. Packet dropping is however not supported or desirable in VCT, this is therefore not a method that is usable in the current situation.

Both LBL and JP have their drawbacks. LBL requires either direct modification of the switches, or a BB which might become a bottleneck in the AC fase. JP spends some time sending probe packets, delaying the admission of the flow until the probing is completed.

If the flows have time to wait for the probing to complete, JP is a good alternative offering low latency and jitter, though at the expense of throughput. If the AC decision must be performed quickly, LBL may be used, as long as there is a limited rate of flow events. LBL may also be tuned to provide even lower jitter as we saw in Section 7.5.

Chapter 8

Conclusion

This project has proposed and evaluated three admission control algorithms with the purpose of achieving Quality of Service guarantees with regards to throughput, latency, and jitter for both the service levels as a whole and for individual flows, as well as an alternative method for achieving low jitter. The admission control algorithms have been evaluated through extensive simulation experiments involving many different network topologies and several address distributions. The three admission control schemes represent fundamentally different approaches to admission control. The Link-by-Link scheme utilises a centralised bandwidth broker with a priori knowledge about the effective link utilisation and knowledge of the network topology. In the Egress Measurement method the egress nodes passively monitor the existing network traffic and uses mainly latency as the admission control criteria, whilst Jitter Probing actively probes the network for available capacity by observing the jitter of the probe packets. The results show that both Link-by-Link and Jitter Probing are capable of giving bandwidth guarantees both to the service level and individual flows. The Egress Measurements proved a disappointment, it was not able to give such good guarantees.

Latency and jitter on the other hand are properties that are difficult to control in virtual cut-through networks. Although Jitter Probing and Link-by-Link are able to improve on the network without admission control, Jitter Probing being the better one, it seems difficult to give strict guarantees.

The problem of unpredictable latency and jitter was challenged by allowing the switches to drop packets on Service Level 1 if there was any danger of the flow control mechanism holding packets back due to lack of buffer space. This solution was not able to improve the maximum jitter over that of the Jitter Probing, but it was able to achieve only slightly worse jitter with significantly higher throughput, almost double, for Service Level 1 when combined with Link-by-Link admission control.

There is a strong relationship between throughput and network latency/jitter.

Any improvements in latency/jitter come at the expense of lower network throughput. This may be partly rectified by having uncontrolled low priority traffic in the network as has been the case in the simulations presented here, but depending on the Quality of Service parameters, much low priority traffic may also worsen the Quality of Service perceived by the high-priority traffic.

8.1 Further work

There still remains to find a way of efficiently controlling latency and jitter in virtual cut-through networks. The nature of virtual cut-through networks makes it uncertain whether such methods will ever be found.

The results presented in this project have been achieved using a certain set of parameters for the different admission control algorithms. With a correct tuning of the different parameters, for example the effective link bandwidth in the Link-by-Link scheme, it might the possible to improve on the results as we present them here.

As for the results presented here, simulations should have been performed for a longer simulation time, using networks of smaller and larger sizes, using larger packets, presenting other parameter configurations of the admission control algorithms and so on. All of this may be done to study further the application of admission control in virtual cut-through networks.

Packet dropping should also be studied further. Although packet dropping is incompatible with the virtual cut-through philosophy, further work as to when the switches should drop packets may lead to solutions which are able to give jitter guarantees.

Bibliography

[1] The direct connect web page. http://www.neo-modus.com/.

[2] The gnutella web page. http://www.gnutella.com/.

[3] Ipv6: The next generation internet. http://www.ipv6.org/.

[4] The issll charter. http://www.ietf.org/html.charters/issll-charter.html.

[5] The kazaa web page. http://www.kazaa.com/.

[6] F. J. Alfaro, Jose L. Sanchez, Jose Duato, and Chita R. Das. A strategy to compute the infiniband arbitration tables. *Proceedings of International Parallel and Distributed Processing Symposium*, April 2002.

[7] Francisco J. Alfaro, Jose L. Sanchez, and Jose Duato. A strategy to manage time sensitive traffic in infiniband. *Proceedings of Workshop on Communication Architechture for Clusters (CAC)*, April 2002.

[8] Infiniband Trade Association. Infiniband web page. http://www.infinibandta.com/.

[9] InfiniBand Trade Association. *InfiniBand Architecture Release 1.0, Volume One - General Specifications*, final edition, October 24th 2000.

[10] Ayan Banerjee, John Drake, Jonathan Lang, Brad Turner, Daniel Awduche, Lou Berger, Kireeti Kompella, and Yakov Rekhter. Generalized mulitprotocol label switching: An overview of signaling enhancements and recovery techniques. *IEEE Communications Magazine*, pages 144–151, July 2001.

[11] Ayan Banerjee, John Drake, Jonathan Lang, Brad Turner, Kireeti Kompella, and Yakov Rekhter. Generalized multiprotocol label switching: An overview of routing and management enhancements. *IEEE Communications Magazine*, pages 144–150, January 2001.

[12] Aurelio Bermudez, Rafael Casado, Francisco J. Quiles, Timothy M. Pinkston, and Jose Duato. Evaluation of a subnet management mechanism for infiniband networks. *ICPP*, March 2003.

[13] G. Bianchi, F. Borgonovo, A. Capone, L. Fratta, and C. Petrioli. Endpoint admission control with delay variation measurements for qos in ip networks. *ACM SOGCOMM Computer Communications Review*, 32(2):61–69, April 2002.

[14] S. Blake, D. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. Rfc 2475, IETF, 1998.

[15] Stefan Bodamer and Joachim Charzinski. Evaluation of effective bandwidth schemes for self-similar traffic. *Proceedings of the 13th ITC Specialist Seminar on IP Measurement, Modeling and Management*, pages 21–1–21–10, September 2000.

[16] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[17] R. Braden, D. Clarke, and S. Shenker. Integrated services in the internet architecture: an overview. Rfc 1633, IETF, 1994.

[18] Lee Breslau, Edward W. Knightly, Scott Shenker, Ion Stoica, and Hui Zhang. Endpoint admission control: architectural issues and performance. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2000.

[19] Werner Bux, Wolfgang E. Denzel, Ton Enbersen, Andreas Herkersdorf, and Ronald P. Luijten. Technologies and building blocks for fast packet forwarding. *IEEE Communications magazine*, January 2001.

[20] John S. Carson. Modeling and simulation worldviews. *Proceedings of the 1993 Winter Simulation Conference*, pages 18–23, 1993.

[21] C Cetikaya and E. W. Knightly. Egress admission control. *IEEE infocom*, 3:1471–1480, 2000.

[22] Y. Chen and C. L. Williamson. A model for self-similar ethernet LAN traffic: Design, implementation, and performance implications. Technical Report DR-95-7, 27, 1995.

[23] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design*. Addison Wesley, third edition, 2001.

[24] Mark Crovella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. In *Proceedings of SIGMETRICS'96: The ACM International Conference on Measurement and Modeling of Computer Systems.*, Philadelphia, Pennsylvania, May 1996. Also, in Performance evaluation review, May 1996, 24(1):160-169.

[25] J. Duato, S. Yalamanchili, and L. Ni. Interconnection networks: an engeneering approach. *IEEE Computer Society*, 1997.

[26] Hans Eberle and Erwin Oertli. Switcherland: A qos communication architecture for workstation clusters. In *ISCA*, pages 98–108, 1998.

[27] Ashok Erramilli, R. P. Singh, and Parag Pruthi. Chaotic maps as models of packet traffic. *ITC*, 14, 1994.

[28] M. D. Schroder et. al. Autonet: a high-speed, self-configuring local area network using point-to-point links. *SMR Research Report 59, Digital Equipment Corporation*, 1990.

[29] R. Gidron. Teranet: A multihop multichannel ATM lightwave network. *Third IFIP WG 6.4 Conference on High Speed Networking; Berlin, Germany*, pages 61–76, 1991.

[30] Stein Gjessing, Olav Lysne, Audun Fosselie Hansen, and Amund Kvalbein. The vine project: Towards predictabe communication in heterogeneous networks. *To appear in proceedings from ICN'04, 3rd International Conference on Networking*, 2004.

[31] H. T. Hill, R. ; Kung. A diffserv enhance admission control scheme. *IEEE*, 4:2549–2555, 2001.

[32] Øyvind Holmeide and Tor Skeie. Switched synchronization. In *Industrial Ethernet Book (IEB)*, volume 7, pages 24–27. 2001.

[33] Robert W. Horst. Tnet: A reliable system area network. *IEEE Micro*, pages 15(1):37–45, 1995.

[34] Guillermo Ibanez. Gmpls. towards a common control and management plane: Generalized multiprotocol label switching in optical transport networks (gmpls).

[35] Sugih Jamin, Scott J. Shenker, and Peter B. Danzig. Comparison of measurement-based admission control algorithms for controlled-load service. *IEEE*, 1997.

[36] Jurgen Jasperneite and Peter Neumann. Switched ethernet for factory communication. *Proceedings of 8'th IEEE International Conference on Emerging Technologies and FActory Automation (ETFA'01)*, pages 205–212, October 2001.

[37] Jurgen Jaspernite, Peter Neumann, Michael Theis, and Kym Watson. Deterministic real-time communication with switched ethernet. *Proceedings of 4'th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.

[38] Mark J. Karol, Michael G. Hluchyj, and Samuel P. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications*, COM-35(12):13471356, 1987.

[39] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3(4):267–286, 1979.

[40] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In Deepinder P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[41] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. Communications COM-39*, pages 1482–1493, 1992.

[42] Anibal D. A. Miranda and Abzaloni A. Synthetizing of markovian and self-similar lan/wan traffic on data networks. *IEEE International Telecommunications Symposium*, 2002.

[43] Institute of Electrical and Electronic Engineers. Local area network - csma/cd access method and physical layer specifications. *American National Standard ANSI/IEEE 802.3, IEEE computer Society*, 1985.

[44] Jørgen Olsen. Atm internetworking. Master's thesis, 1996.

[45] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[46] Joe Pelissier. Providing quality of service over infinband architecture fabrics. *Proceedings of Hot Interconnects X*, 2000.

[47] Larry L. Peterson and Bruce S. Davie. *Computer Networks a Systems Approach.* Morgan Kaufman Publishers, Inc., second edition edition, 1996.

[48] P.Lpez, J.Flick, and J.Duato. Deadlock-free routing in infiniband through destination renaming. *IEEE Computer Society*, Proceedings of the 2001 International Conference on Parallel Processing (ICPP '01):427–434, 2001.

[49] Wenjian Qiao and Lionel M. Ni. Adaptive routing in irregular networks using cut-through switches. In *ICPP, Vol. 1*, pages 52–60, 1996.

[50] Sven-Arne Reinemo, Frank Olaf Sem-Jacobsen, Tor Skeie, and Olav Lysne. Admission control for diffserv based quality of service in cut-through networks. *Proceedings of the 10th International Conference on High Performance Computing*, 2003.

[51] Sven-Arne Reinemo, Tor Skeie, and Olav Lysne. Applying the diffserv model on cut-through networks. *Proceedings of the 2003 International Conference of Parallel and Distributed Processing Techniques and Applications*, 2003.

[52] Stephan Robert and Jean-Yves Le Boudec. New models for pseudo self-similar traffic. 1996.

[53] L.G. Samuel, J.M. Pitts, and R.J. Mondagon. Fast self-similar traffic generation. *Proceeding of the Fourteenth UK Teletraffic Symposium on Performance Engineering in Information Systems*, pages 8/1–8/4, March 1997.

[54] J. C. Sancho, A. Robles, J. Flich, P. Lopez, and J. Duato. Effective methodology for deadlock-free minimal routing in infiniband networks.

[55] Jos Carlos Sancho, Antonio Robles, and Jos Duato. Effective strategy to compute forwarding tables for infiniband networks. 2001.

[56] Julie Schlembach, Anders Skoe, Ping Yuan, and Edward Knightly. Design and implementation of scalable admission control. *QoS-IP*, pages 1–15, 2001.

[57] Thomas J. Schriber and Daniel T. Brunner. Inside simulation software: How it works and why it matters. *Proceedings of the 1996 Winter Simulation Conference*, pages 23–30, 1996.

[58] Rich Seifert. *Gigabit Ethernet*. Addison Wesley Pub Co., 1998.

[59] Frank Olaf Sem-Jacobsen, Sven-Arne Reinemo, Tor Skeie, and Olav Lysne. Acheiving flow level qos in cut-through networks through admission control and diffserv. *To be submitted to International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2004.

[60] Tor Skeie. *Topics in Interconnect Networking*. PhD thesis, University of Oslo, April 1998.

[61] Tor Skeie, Svein Johannessen, and Øyvind Holmeide. Highly accurate time sychronisation over switched ethernet. In *8th IEEE conference on Emerging Technologies and Factory Automation (ETFA)*, 2001.

[62] Tor Skeie, Svein Johannessen, and Øyvind Holmeide. The road to an end-to-end deterministic ethernet. *Proceedings of 4'th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.

[63] Tor Skeie, Olav Lysne, J. Flich, P. Lopez, A. Robles, and J. Duato. Lash-tor: A generic transition-oriented routing algorithm. *Submitted to International Conference on Parallel and Distributed Systems (ICPADS)*, 2004.

[64] Tor Skeie, Olav Lysne, and Ingebjørg Theiss. Layered shortest path (lash) routing in irregular system area networks. *Proceedings of Communication Architecture for Clusters*, 2002.

[65] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Inc., third edition edition, 1996. International edition.

[66] Y. Tseng. Multi-node broadcasting in hypercubes and star graph, 1998.

[67] R. Tucker and W. Zhong. Photonic packet switching: an overview, 1999.

[68] Nikolaos Vasiliou. Reading course paper, overview of internet qos and web server qos. April 2000.

[69] Arun Viswanathan, Nancy Feldman, Zheng Wang, and Ross Callon. Evolution of multiprotocol label switching. *IEEE Communications Magazine*, pages 165–173, May 1998.

[70] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

[71] Xipeng Xiao and Lionel M. Ni. Internet qos: A big picture. *IEEE Network*, pages 8–18, March 1999.

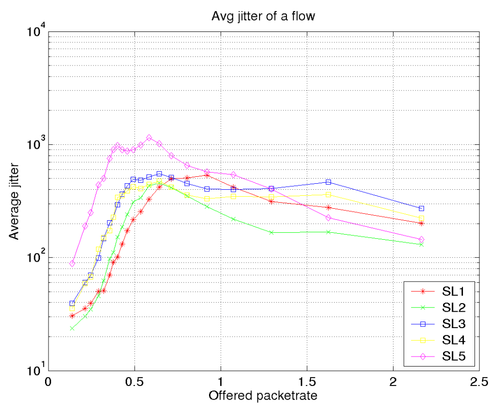
[72] Jia Yongxing and Ming-Chen. A new architecture of providing end-to-end quality of service for differentiated services network. *IEEE Military Communications Conference*, 2:1450–1455, 2001.

[73] Ki Hwan Yum, Eun Jung Kim, Chita R. Das, Mazin Yousif, and Jose Duato. Integrated admission and congestion control for qos support in clusters. *Proceeding of IEEE Internationa Conference on Cluster Computing*, pages 325–332, September 2002.

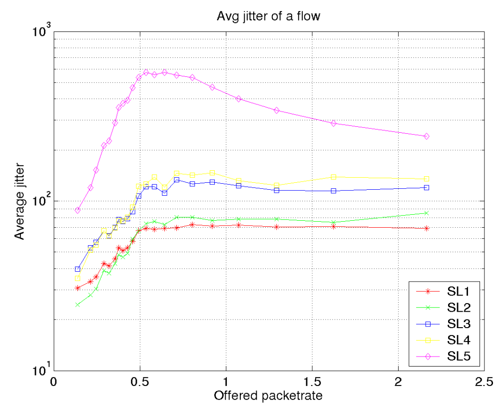
[74] Weibin Zhao, David Olshefski, and Henning Schulzrinne. Internet quality of service: an overview.

Appendix A

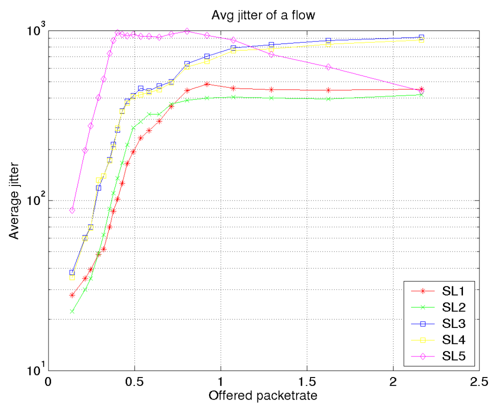
Additional figures



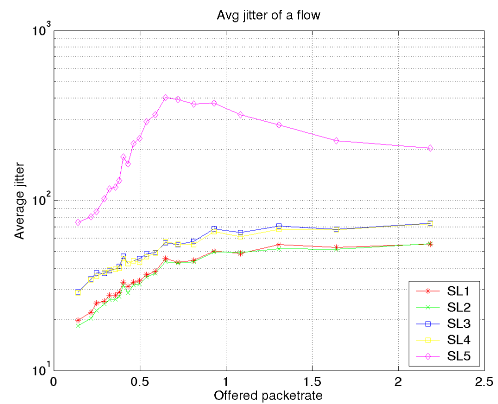
(a) No admission control



(b) Link-by-Link

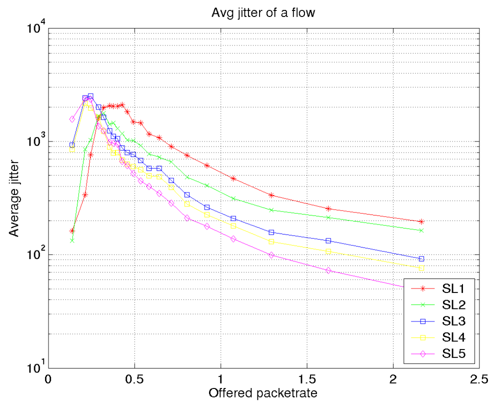


(c) Egress Measurements

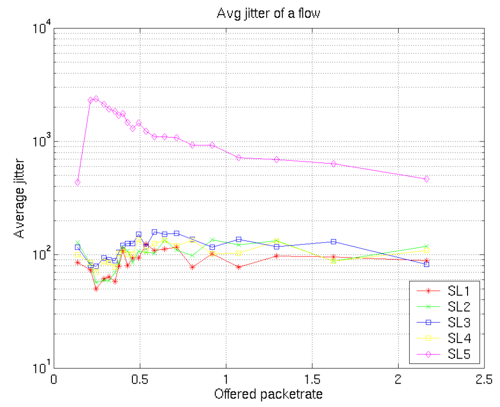


(d) Jitter Probe

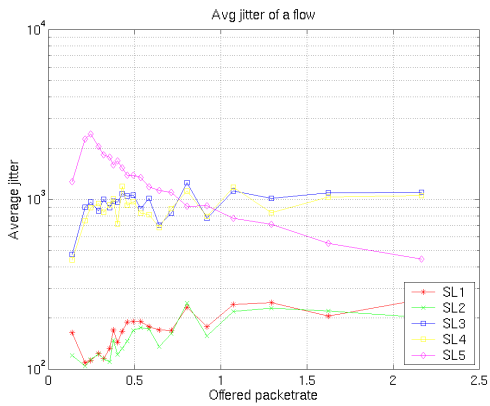
Figure A.1: Average packet jitter for a flow



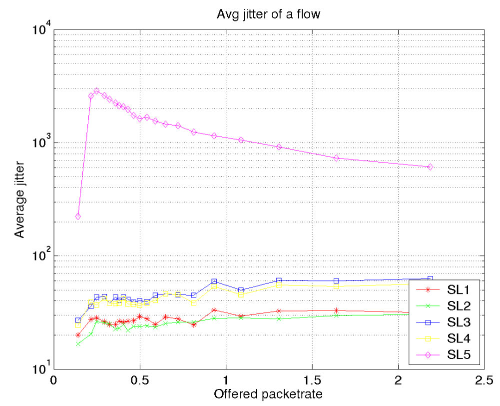
(a) No admission control



(b) Link-by-Link

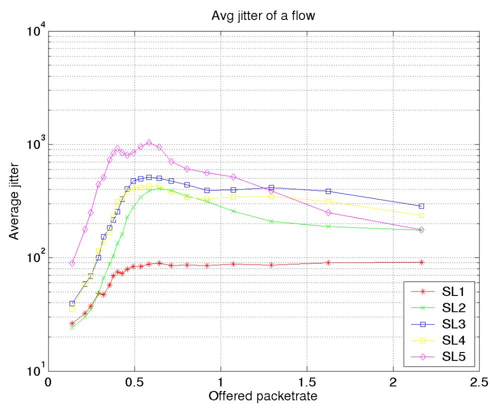


(c) Egress Measurements

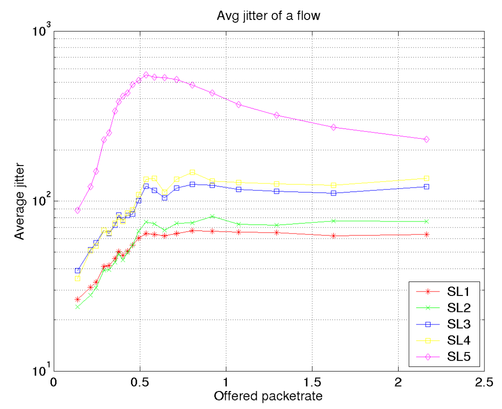


(d) Jitter Probe

Figure A.2: Average packet jitter of flows for two hot-spots



(a) No admission control



(b) Link-by-Link

Figure A.3: Average packet jitter with packet dropping

Appendix B

Produced Articles

Admission Control for DiffServ Based Quality of Service in Cut-through Networks

Sven-Arne Reinemo, Frank Olaf Sem-Jacobsen, Tor Skeie, and Olav Lysne

Simula Research Laboratory
P.O.Box 134, N-1325 Lysaker, Norway
svenar@simula.no, frankose@ifi.uio.no, {tskeie, olavly}@simula.no

Abstract. Previous work on Quality of Service in Cut-through networks shows that resource reservation mechanisms are only effective below the saturation point. Admission control in these networks will therefore need to keep network utilization below the saturation point, while still utilising the network resources to the maximum extent possible. In this paper we propose and evaluate three admission control schemes. Two of these use a centralised bandwidth broker, while the third is a distributed measurement based approach. We combine these admission control schemes with a DiffServ based QoS scheme for virtual cut-through networks to achieve class based bandwidth and latency guarantees. Our simulations show that the measurement based approach favoured in the Internet communities performs poorly in cut-trough networks. Furthermore it demonstrates that detailed knowledge on link utilization improves performance significantly.

1 Introduction

Internet has today evolved into a global infrastructure supporting applications such as streaming media, E-commerce, network storage, etc. Each of these applications must handle an ever increasing volume of data demanded as predictable transfers. In this context the provision of Quality of Service (QoS) is becoming an important issue. In order to keep pace with computer evolution and the increased burden imposed on data servers, application processing, etc. created by the popularity of the Internet, we have in recent years seen several new technologies proposed for System and Local Area Networking (SAN/LAN) [3,4,7,14,24]. Common for this body of technologies is that they rely on point-to-point links interconnected by off-the-shelf switches that support some kind of back-pressure mechanism. Besides, most of the referred technologies also adhere to the cut-through or wormhole switching principles - only Gigabit Ethernet is using the store-and-forward technique. For a survey of some relevant networking principles we refer to [6].

IETF has for several years provided the Internet community with QoS concepts and mechanisms. The best known ones are Integrated Services (IntServ) [8], Resource Reservation Protocol (RSVP) [13], and Differentiated Services (DiffServ) [5]. IntServ together with RSVP define a concept based on per flow reservations (signalling) and admission control to be present end-to-end. DiffServ,

however, takes another approach assuming no explicit reservation mechanism in the interior network elements. QoS is here realized by giving data packets differentiated treatment relative to the QoS header code information. In the underlying network technologies QoS has to a less extent been emphasised - the key metrics here have mainly been mean throughput and latency. To provide QoS end-to-end, possibly over heterogeneous technologies this means that the lower layers should also have support for predictable transfer including the ability to interoperate with a higher level IETF concept. This issue is being challenged by emerging SAN/LAN standards, such as InfiniBand^{TM} [4] and Gigabit Ethernet [24] providing various QoS mechanisms.

Recently we have also seen several research contributions to this field. Jaspernite et. al. [9,10] and Skeie et. al [15] discuss different aspects of taking control of the latency through switched Ethernet relative to the IEEE 802.1p standard aiming for traffic priorities. Another body of work is tailored to the InfiniBand^{TM} architecture (IBA) [1, 2, 12]. In [12] Pelissier gives an introduction to the set of QoS mechanisms offered by IBA and the support for DiffServ over IBA. In this approach the presence of admission control is assumed. Alfaro et. al build on this scheme and present a strategy for computing the arbitration tables of IBA networks, moreover a methodology for weighting of virtual layers referring to the dual arbitrator defined by IBA [2]. The concept is evaluated through simulations assuming that only bandwidth sensitive traffic requests QoS. In [1] Alfaro et. al also include time sensitive traffic, besides calculating the worst case latency through various types of switching architectures.

DiffServ is foreseen to be the most prominent concept for providing QoS in the future Internet [11, 17]. DiffServ makes a distinction between boundary nodes and core nodes with respect to support of QoS features. Following the DiffServ philosophy no core switch should hold status information about passing-through traffic. Neither should there be any explicit signalling on a per flow basis to these components. This means that within the DiffServ framework any admission control or policing functionality would have to be implemented by boundary nodes or handled by a dedicated bandwidth broker. The core switches are assumed to perform traffic discrimination only based on service class, which is decided by a QoS tag included in the packet header - all packets carrying the same QoS tag will get equal treatment. From that viewpoint DiffServ is apparently a relative service model having difficulties giving absolute guarantees.

None of the previous debated contributions comply with the DiffServ model. In [12] Pelissier, however, discusses interoperation between DiffServ and IBA on a traffic class and service level basis, but refer to RSVP with respect to admission control. The strategy proposed by Alfaro et. al has to recompute the IBA dual arbitrator every time that a new connection is honoured [1,2]. And such a scheme is not associable with DiffServ. Neither is the admission control scheme presented in [29] by Yum et. al, which use hop by hop bandwidth reservations and requires recomputations of the weighted round robin scheduler at every hop towards the destination. In [25] Reinemo et. al. studied the provision of QoS in cut-through networks by adhering to the DiffServ model. The problem was

approached without any explicit admission control mechanism, as a pure relative model. Empirically they examined the sensitivity of different QoS properties under various load and traffic mixture conditions, hereunder assessing the effect of back-pressure.

In this paper we endeavour to achieve class based QoS in cut-through networks by use of admission control. More specifically, we extend the concept described in [25] with admission control. However, still in compliance with the Diff-Serv paradigm where service classes, as aggregated flows, are the target for QoS. Three different admission control mechanisms are proposed and carefully evaluated through extensive simulations. Two of the schemes assume pre-knowledge of the network's performance behaviour without admission control, and are furthermore implemented as a centralised bandwidth broker. The third scheme is based on endpoint/egress admission control and relies on measurements to assess the load situation, inspired by Internet QoS provisioning. To the best of the authors' knowledge no detailed admission control methods have been proposed for cut-through networks before.

The rest of this paper is organised as follows. In section 2 we give a description of our QoS architecture and routing algorithm, in section 3 our three admission control mechanisms are described, and in section 4 our simulation scenario is described. In section 5 we discuss our performance results, and finally in section 6 we give some concluding remarks.

2 QoS Architecture

The architecture used in our simulations is inspired by IBA link layer technology [4] and is a flit based virtual cut-through (VCT) switch. The overall design is based on the canonical router architecture described in [6].

In VCT the routing decision is made as soon as the header of the packet is received and if the necessary resources are available the rest of the packet is forwarded directly to the destination link [23]. If the necessary resources are busy the packet is buffered in the switch. In addition we use flow control on all links so all data is organised as flow control digits (flits) at the lowest level.

The switch core consists of a crossbar where each link and VL has dedicated access to the crossbar. Each link supports one or more virtual lanes (VL), where each VL has its own buffer resources which consist of an input buffer large enough to hold a packet and an output buffer large enough to hold two flits to increase performance. Output link arbitration is done in a round robin fashion.

To achieve QoS our switch architecture support QoS mechanisms like the ones found in the IBA architecture. IBA supports three mechanisms for QoS which are mapping of service level (SL) to VL, weighting of VLs and prioritising VLs as either low priority (LP) or high priority (HP). A more detailed description of these QoS aspects can be found in [25].

The routing used is a newly introduced routing algorithm called *Layered shortest path routing* (LASH) [16]. LASH is a minimal deterministic routing algorithm for irregular networks which only relies on the support of virtual layers.

There is no need for any other functionality in the switch, so LASH fits well with our simple approach to QoS. An in-depth description of LASH is found in [16].

3 Admission Control

In this section we propose three different admission control (AC) mechanisms that we carefully evaluate in section 5.

3.1 Calibrated Load Based Admission Control

The *Calibrated Load* (CL) approach is a simple scheme relying on the fact that a BB knows the total rate of traffic entering the network. Our AC parameter is the amount of traffic which can be injected into the network while still keeping the load below saturation. As the rate of traffic entering the network reaches the CL parameter no more traffic will be admitted. In most cases the CL parameter must be decided by measurements on the network in question to find the saturation point - our CL is deduced from measurements performed in [25].

To keep HP and LP traffic separated we use two different CL parameters, one for the total HP traffic and one for the total LP traffic. For HP traffic this can be expressed as follows

$$\sum_{i=0}^{n} L_{HP,i} + P_{HP} < CL_{HP} \quad . \tag{1}$$

Here CL_{HP} is the calibrated load for outgoing HP traffic, $L_{HP,n}$ is the HP load in node n and P_{HP} is the peak rate for the requesting flow. Moreover, the flow is admitted if the total HP load $\sum_{i=0}^{n} L_{HP,i}$ plus the requested increase P_{HP} is below the calibrated load CL_{HP}. LP traffic can be expressed similarly just substituting HP values with LP values. The strength of this scheme is that it is simple, its weakness is that it is inaccurate since it does not take into account the distribution of flows in the network. And from that viewpoint it is less suitable for handling hot spots.

3.2 Link by Link Based Admission Control

Our second scheme is the Link-by-Link (LL) approach. Here the BB knows the load on every link in the network and will consult the availability of bandwidth on every link between source and destination before accepting or rejecting a flow. Compared to the CL approach, this solution assumes both topological and routing information about the network.

For the AC decision we adopt the *simple sum* approach as presented in [28]. This algorithm states that a flow may be admitted if its peak rate p plus the peak rate of the already admitted flows s is less than the link bandwidth bw. Thus the requested flow will be admitted if the following inequality is true [28]

$$p + s < bw \tag{2}$$

we view p as the increase in peak rate for the flow and s as the sum of the admitted peak rates. As for the *CL* method we deduce the effective bandwidth from the measurements obtained in [25]. Since we are dealing with service levels where each SL have different bandwidth requirements it is natural to introduce some sort of differentiation into equation (2). We achieve this by dividing the link bandwidth into portions relative to the traffic load of the SLs, and include only the bandwidth available to a specific service level bw_{SL} in the equation as follows

$$p + s_{SL} < bw_{SL} \tag{3}$$

where

$$bw_{SL} = bw_{link} * \frac{load_{SL}}{load_{total}} \tag{4}$$

and s_{SL} is the sum of the admitted peak rates for service level SL and bw_{link} is the effective link bandwidth.

3.3 Egress Based Admission Control

Our third scheme is the Egress Based (EB) approach. This is a fully distributed AC scheme where the egress nodes are responsible for conducting the provisions. Basically, we here adopt the Internet AC concept presented by Cetikaya and Knightly in [26]. This method does not assume any pre-knowledge of the network behaviour as was the case with our previous solutions. Also different from the previous approaches is the use of a delay bound as the primary AC parameter. For clarity we give a brief outline of the algorithm below, a more detailed description can be found in [27].

The method is entirely measurement based and relies on that the sending nodes timestamp all packets enabling the egress nodes to calculate two types of measurements. First, by dividing time into timeslots of length τ and counting the number of arriving packets, the egress nodes can deduce the arrival rate of packets in a specific timeslot. By computing the maximum arrival rate for increasingly longer time intervals we get a peak rate arrival envelope $R(t)$, where $t = 0, ..., T$ timeslots, as described in [26]. Second, by comparing the originating timestamp relative to the arrival time, the egress node can calculate the transfer time of a packet. Having this information available the egress node can furthermore derive the time needed by the infrastructure to service k following packets; i.e. a consecutive stream of packets where the next packet in the service class enters the infrastructure before the previous packet has departed the egress node. By doing this for larger and larger k sequences of packets within a measuring interval of length $T\tau$ and subsequently inverting this function we achieve the service envelope $S(t)$, giving the amount of packets processed by the network in a given time interval t. Now repeating this M times, the mean $\overline{R}(t)$ and the variance $\sigma^2(t)$ of $R(t)$, and the mean $\overline{S}(t)$ and variance $\Psi^2(t)$ of $S(t)$ may be calculated. If a flow request has a peak rate P and a delay bound D it

may be accepted if the peak rate P plus the measured arrival rate $R(t)$ is less than the service rate allowing for the delay $D, S(t + D)$.

The EB scheme derives its knowledge of the network from measurements of the traffic passing through the egress nodes. It is therefore difficult for the egress nodes to have a complete picture of the load in the network, moreover the packet latency is used to infer the network load utilising the fact that an increased network load will cause increase in latency as well. From that viewpoint it seems difficult to give a service class bandwidth guarantees since it has no concrete knowledge of the network load. The algorithm will admit as much traffic as it can without breaking the delay bound. The key instrument of the scheme is the given delay bound for the different flows, and the efficiency of the algorithm is linked to its ability to limit the service levels to operate within the delay bounds.

3.4 Target for Admission Control

The main findings for the work in [25] are that (*i*) throughput differentiation can be achieved by weighting of VLs and by classifying the VLs as either low or high priority, (*ii*) the balance between VL weighting and VL load is not crucial when the network is operating below the saturation level. In general this sets the target for the AC, since as long as we can ensure that the load of the various service classes is below saturation level we can also guarantee that each of these classes get the bandwidth they request. The target for admission control is thus the point where the amount of accepted traffic is starting to become less than traffic offered. The effective bandwidth at this point will be used as a steering vehicle by the CL and LL methods.

Another main finding in [25] is that though the latency characteristics below saturation were fairly good, significant jitter was observed. This problem we challenge by proposing the EB method, where a given delay bound is the requested quality of service. Since this concept is continuously monitoring the end-to-end latency characteristics for all pair of nodes one should possibly expect that delay guarantees could be given.

4 Simulation Scenario

For all simulations we have used a flit level simulator developed in house at Simula Research Laboratory. In the simulation results that follow, all traffic is modelled by a normal approximation of the Poisson distribution. We have performed simulations on a network with 32 switches, where each switch is connected to 5 end nodes and the maximum number of links per switch is 10 in addition to the end nodes. We have randomly generated 16 irregular topologies and we have run measurements on these topologies at increasing load. We use LASH [16] as routing algorithm and random pairs as traffic pattern. In the random pairs scheme each source sends only to one destination and no destination receives from more that one source. The link speed is one flit per cycle, the flit size is one byte and the packet size is 32 bytes for all packets.

The five different end nodes send traffic on one of five different service levels. One service level for each node (Table 1), SL 1 and 2 are considered to be of the expedited forwarding (EF) class in DiffServ terminology. And SL 3 and 4 are considered to be of the assured forwarding (AF) class. SL 5 is considered as best effort (BE) traffic and from that viewpoint is not a subject of AC.

For the CL and LL schemes all simulations where run with an ACT deduced from our measurements in [25]. In the first part of the simulation the send rate is steadily increased by adding more and more flows until admission is denied by the AC scheme. When this happens the current rate is not changed, but the node will continue to try to go beyond the ACT for a fixed number of times before it gives up. For the EB scheme the send rate is increased in the same way, but the AC decision is primarily based on measured latency as described in section 3.3.

Table 1. Services levels

SL	DS[1]	Load %	BW[2]	Pri	SL	DS	Load %	BW	Pri	SL	DS	Load %	BW	Pri
1	EF	10	4	high	3	AF	20	8	low	5	AF	30	1	low
2	EF	15	6	high	4	AF	25	10	low					

5 Performance Results

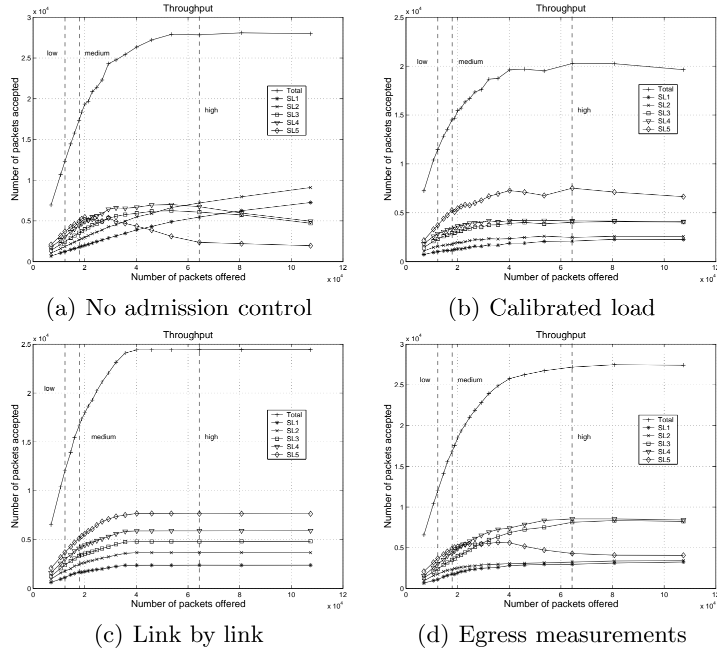
5.1 Throughput

Recall that the target for the AC is to make sure that the network operates below saturation at all times, since below this point we can guarantee that all SLs get the bandwidth they request. The relative requests for each SL are as shown in Table 1. Figure 1(a) shows what happens in a network without AC when it enters saturation. We are no longer able to give all service classes the bandwidth they request and the HP classes preempts LP bandwidth, i.e. the bandwidth differentiation is no longer according to the percentages in Table 1. In the CL scheme, we see from figure 1(b) that we are successful in keeping the accepted load below the saturation point, even as the offered load goes beyond this point. The bandwidth differentiation does not fail as is the case in figure 1(a), but it suffers slightly as we reaches *high* load. As the load increases the differentiation between SLs in the same class is diminished. Thus, the CL scheme is able to keep the load below saturation. However, it appears that it is too coarse to achieve good bandwidth differentiation between SLs of the same class since it makes its AC decisions based on the total load for a class and not for each SL.

Moving on to the LL scheme (figure 1(c)) we see several improvements compared to the CL scheme. First, we get a sharper bandwitdh cut-off with much

[1] The DiffServ equivalent service class.

[2] The maximum number of flits allow to transmit when scheduled.



(a) No admission control (b) Calibrated load

(c) Link by link (d) Egress measurements

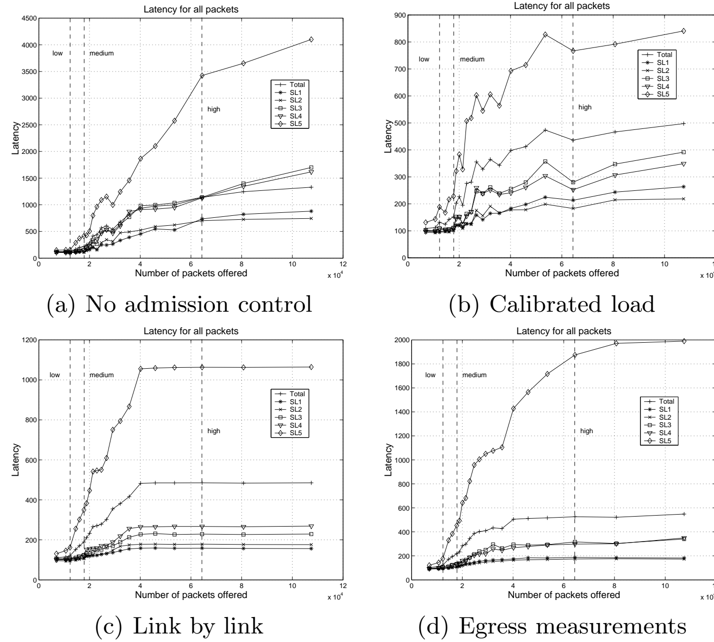
Fig. 1. Throughput

less hesitation than for CL. Second, we achieve a differentiation relative to the requests, meaning that we can give bandwidth guarantees. Third, we are able to utilise the network resource better as we get closer to the saturation point. This improvement is probably due to the fact that the LL scheme knows the load of every link in the network and is able to make the AC decision based on the load along the actual source/destination path.

Finally, we have the EB method. It is apparent from figure 1(d) that this method is unable to give bandwidth guarantees, as well as increasing the load beyond the saturation point and admitting too much traffic. Now as the load increases beyond saturation the best effort traffic (SL 5) is reduced as it must make way for traffic on the other SLs. The issue here is that delay is the most significant AC parameter in this scheme and bandwidth requirements have more or less been ignored.

5.2 Latency

Let us now turn our attention to the latency results. Figure 2(a) shows the average latency for increasing load values without admission control. Comparing it with the CL results in figure 2(b) shows that the CL scheme is quite close when we look at the same load values. The average latency for all packets is 436 for CL at the high mark which is a 6% increase compared to the scheme without AC at a corresponding load. A problem with the CL scheme is that the latency



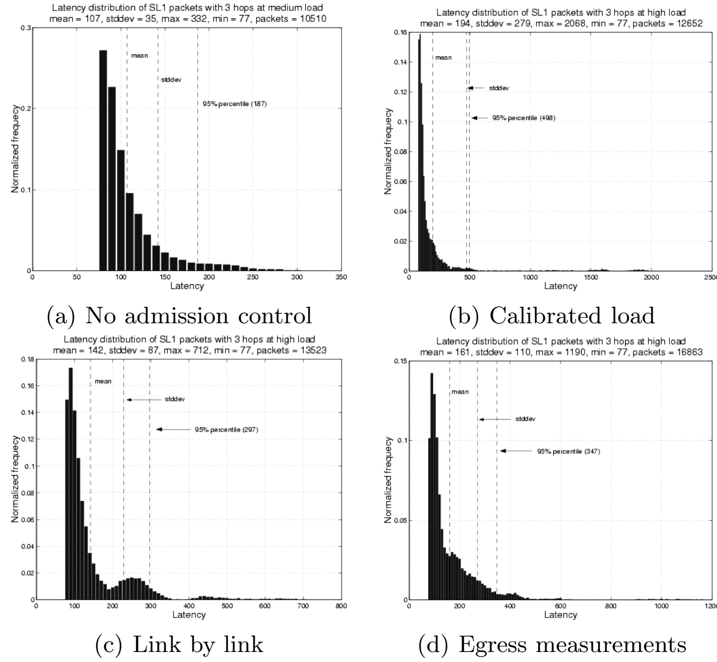
(a) No admission control (b) Calibrated load

(c) Link by link (d) Egress measurements

Fig. 2. Average latency

values are unstable as the load increases since the estimate of current throughput is to coarse to target the exact rejection point. The LL scheme overcomes this problem as show in figure 2(c). As soon as the LL scheme starts rejecting flows the latency stabilises. The latency values for high load is 485 which is slightly above the CL latency at this particular point. The LL method also gives a more linear increase in latency as we approach the rejection point for new flows. The LL scheme is the better of the two as it gives lower latency to SL 1-4 and higher latency to the best effort traffic in SL 5. Even if it has a higher average latency for all packets (compared to CL) it performs better since the increase in the average is caused by the best effort traffic in SL 5.

The EB scheme uses measured latency as its primary AC parameter. The results are presented in figure 2(d). This scheme produces average results which fall between the CL and LL scheme. Furthermore, it is capable of giving the same latency to SLs fairly independently of weight such as SL 1 and 2, but it is unable to satisfy the delay bound of 100 for SL 1 and 2, and 250 for SL 3 and 4. The achieved latency at the high mark is 187 for SL 1 and 316 SL 3. So even if using a measurement based method we are unable to give hard delay guarantees. It seems very difficult to give delay bounds in combination with good throughput in cut-through networks. To remedy this problem one possible could reduce the throughput by hardening the AC, or we could turn to other means such as modifying the flow control to better handle delay bound traffic.



(a) No admission control (b) Calibrated load

(c) Link by link (d) Egress measurements

Fig. 3. Latency distribution for packets with 3 hops

5.3 Jitter

Finally lets turn our attention to jitter. Figure 3 shows the latency distributions for packets with a path length of 3 hops. This was the most frequently occurring path length in our simulations. The mean, standard deviation and 95 % percentile are marked with a dashed line in the figures. The distance between the mean mark and the standard deviation mark reflects the standard deviation.

Figure 3(a) shows the latency distribution achieved without AC at the load marked as medium in figure 1(a). Figure 3(b) shows the distribution for the CL scheme at the load marked as high in figure 1(b). Note that the load at this point is about 20% above that without admission control. We see that the CL scheme has quite poor jitter characteristics, which is reasonable if we recall the latency curve we saw in figure 2(b). The mean is 194, the standard deviation 279 and the 95% percentile is 498. Thus 95% of the packets have a latency of 498 or below. Even if the mean value is not too bad, the large standard deviation and the 95% percentile shows that jitter is clearly very high. The LL scheme has better jitter characteristics. From figure 3(c) we see that the histogram has a shorter tail compared to figure 3(b). We have a mean of 142, standard deviation of 87 and a 95% percentile of 297. Which reduces the jitter potential and gives us almost a 40% reduction of packet latency for 95% of the packets. In addition the load for LL at this point is about 7% above the CL load. Thus, better results are achieved at a higher load. For EB scheme in figure 3(d) we see that it is

unable to improve on the results from the LL scheme. With a 95% percentile of 357 it has a 30% improvement over the CL. Note that this is achieved at a load 25% above the CL load. Still, even a measurements based delay bound scheme is unable to give good jitter characteristics in cut-through networks.

6 Conclusion

In this paper we propose and evaluate three different admission control schemes for virtual cut-through networks. Each one suitable for use in combination with a DiffServ based QoS scheme to deliver soft real-time guarantees. Two of the schemes assume pre-knowledge of the network's performance behaviour without admission control, and are both implemented with bandwidth broker. The third method is based on endpoint/egress admission control and relies on measurements to assess the load situation.

Our main findings are as follows. First, bandwidth guarantees for aggregated flows are achievable with the use of the Link-by-Link scheme. While the Calibrated Load and Egress Based methods are unable to achieve such good guarantees. Second, latency and jitter properties are hard to achieve regardless of the method used. This is due to the nature of cut-networks and the way flow control affects latency. Strict admission control can be used to improve latency, but at the cost of lower throughput. To achieve a combination of high throughput and low latency modifications to the flow control may be considered.

References

1. F. J. Alfaro, J. L. Sanchez, J. Duato, and C. R. Das. A strategy to compute the InfiniBand arbitration tables. In *Proceedings of International Parallel and Distributed Processing Symposium*, April 2002.
2. F. J. Alfaro, J. L. Sanchez, and J. Duato. A strategy to manage time sensitive traffic in InfiniBand. In *Proceedings of Workshop on Communication Architecture for Clusters (CAC)*, April 2002.
3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet – a gigabit-per-second LAN. IEEE MICRO, 1995.
4. InfiniBand Trade Association. Infiniband architecture specification.
5. Differentiated Services. RFC 2475.
6. J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks an engineering approach*, IEEE Computer Society, 1997.
7. R. W. Horst. Tnet: A reliable SAN. *IEEE Micro*, 15(1):37–45, 1995.
8. Integrated Services. RFC 1633.
9. J. Jaspernite, and P. Neumann. Switched Ethernet for Factory Communication. In *Proceedings of* 8th *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'01)*, 205–212, October 2001.
10. J. Jaspernite, P. Neumann, M. Theiss, and K. Watson. Deterministic real-time communication with switched Ethernet. In *Proceedings of* 4th *IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.
11. K. Kilkki. Differentiated services for the Internet. *Macmillian Tech. Publishing*, 1999.

12. J. Pelissier. Providing quality of service over InfiniBand^{TM} architecture fabrics. In *Proceedings of Hot Interconnects X*, 2000.
13. ReSource ReserVation Protocol. RFC 2205.
14. M. D. Schroder et.al., "Autonet: a high-speed, self-configuring local area network using point-to-point links," SRC Research Report 59, Digital Equipment Corporation, 1990.
15. T. Skeie, J. Johannessen, and Ø. Holmeide. The road to an end-to-end deterministic Ethernet. In *Proceedings of* 4th *IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.
16. T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (LASH) routing in irregular system area networks. In *Proceedings of Communication Architecture for Clusters*, 2002.
17. X. Xiao and L. M. Ni. Internet QoS: A Big Picture In *IEEE Network Magazine*, 8–19, March/April 1999.
18. J. S. Yang and C. T. King, "Turn-restricted adaptive routing in irregular wormhole-routed networks," in *Proceedings of the 11'th International Symposium on High Performance Computing (HPCS97)*, July 1997.
19. A. A. Chien and J. H. Kim, "Approaches to Quality of Service in High-Performance Networks," in *Lecture Notes in Computer Science*, vol. 1417, 1998.
20. J. Duato and S. Yalamanchili and B. Caminero and D. S. Love and F. J. Quiles, "MMR: A High-Performance Multimedia Router - Architecture and Design Trade-Offs," in *HPCA*, pages 300-309, 1999.
21. B. Caminero, C. Carrion, F. J. Quiles, J. Duato and S. Yalamanchili, " A Solution for Handling Hybrid Traffic in Clustered Environments: The MultiMedia Router MMR," in *Proceedings of IPDPS-03*, April 2003.
22. M. Gerla and B. Kannan and B. Kwan and E. Leonardi and F. Neri and P. Palnati and S. Walton, "Quality of Service Support in High-Speed, Wormhole Routing Networks," in *International Conference on Network Protocols (ICNP'96)*, 1996.
23. P. Kermani and L. Kleinrock, "Virtual Cut-through: A New Computer Communication Switching Technique," in *Computer Networks*, no. 4, vol. 3, 1979.
24. R. Seifert, *Gigabit Ethernet,* Addison Wesley Pub Co., 1998.
25. S. A. Reinemo and T. Skeie and O. Lysne, "Applying the DiffServ Model in Cut-through Networks," in *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2003.
26. C. Cetikaya and E. W. Knightly, "Egress admission control," in *INFOCOM (3)*, pages 1471-1480, 2000.
27. J. Schlembach and A. Skoe and P. Yuan and E. Knightly, "Design and Implementation of Scalable Admission Control," in *QoS-IP*, pages 1-15, 2001.
28. S. Jamin and S. J. Shenker and P. B. Danzig, "Comparison of Measurement-Based Admission Control Algorithms for Controlled-Load Service," in *INFOCOM (3)*, pages 973-980, 1997.
29. K. H. Yum, E. J. Kim, C. R. Das, M. Yousif, J. Duato, "Integrated Admission and Congestion Control for QoS Support in Clusters," in *Proceedings of IEEE International Conference on Cluster Computing*, pages 325-332, September 2002.

Achieving Flow Level QoS in Cut-through Networks through Admission Control and DiffServ

Frank Olaf Sem-Jacobsen   Sven-Arne Reinemo   Tor Skeie
Olav Lysne
Simula Research Laboratory
P.O.Box 134, N-1325 Lysaker, Norway
frankose@ifi.uio.no, {svenar, tskeie, olavly}@simula.no

Abstract

Cluster networks will serve as the future access networks for multimedia streaming, massive multiplayer online gaming, e-commerce, network storage etc. And for those application areas provisioning of Quality of Service (QoS) is becoming and important issue. DiffServ as specified by the IETF is foreseen to be the most prominent concept for providing predictability in the future Internet. To enable seamless interoperation with the higher level IETF concepts the QoS architecture of the lower layers should comply with the DiffServ paradigm as well. Previous work on predictability in cut-through networks has only studied class *based QoS. In this paper we set out to achieve* flow *level QoS using flow aware admission control in combination with a flow negligent DiffServ inspired QoS mechanism. Our results show that flow level bandwidth guarantees are achievable with the use of the Link-by-Link and the Probe based schemes. In addition we are able to achieve an order of magnitude improvement in jitter and latency in individual flows.*

1 Introduction

As the global Internet has evolved into a marketplace with a wealth of applications, the performance demands on the servers running these applications has grown too large to be handled by a single machine alone. This has resulted in a move from single server applications to applications running on a cluster of machines. Furthermore, new challenges have appeared, one of these are the interconnection of computers in a cluster, another is how to achieve predictable communication between machines in a cluster. This has renewed the focus on Quality of Service and resulted in several new technologies for System and Local Area Networking (SAN/LAN) [3, 4, 7, 14, 24, 31]. Common for this body of technologies is that they rely on point-to-point links interconnected by off-the-shelf switches that support some kind of flow-control mechanism. Besides, most of the referred technologies also adhere to the cut-through or wormhole switching principles - only Gigabit Ethernet is using the store-and-forward technique. For a survey of some relevant networking principles we refer to [6, 32].

IETF has for several years provided the Internet community with QoS concepts and mechanisms. The best known ones are Integrated Services (IntServ) [8], Resource Reservation Protocol (RSVP) [13], and Differentiated Services (DiffServ) [5]. IntServ together with RSVP define a concept based on per flow reservations (signalling) and admission control to be present end-to-end. DiffServ, however, takes another approach assuming no explicit reservation mechanism in the interior network elements. QoS is here realized by giving data packets differentiated treatment relative to the QoS header code information. In the underlying network technologies QoS has to a less extent been emphasized - the key metrics here have mainly been mean throughput and latency. To provide QoS end-to-end, possibly over heterogeneous technologies this means that the lower layers should also have support for predictable transfer including the ability to interoperate with a higher level IETF concept. This issue is being challenged by emerging SAN/LAN standards, such as InfiniBand^{TM} [4] and Gigabit Ethernet [24] providing various QoS mechanisms.

Recently we have also seen several research contributions to this field. Jaspernite et. al. [9, 10] and Skeie et. al [15] discuss different aspects of taking control of the latency through switched Ethernet relative to the IEEE 802.1p standard aiming for traffic priorities. Another body of work is tailored to the InfiniBand^{TM} architecture (IBA) [1, 2, 12]. In [12] Pelissier gives an introduction to the set of QoS mechanisms offered by IBA and the support for DiffServ over IBA. In this approach the presence of admission control is assumed. Alfaro et. al build on this scheme and present a strategy for computing the arbitration tables of IBA networks, moreover a methodology for weighting of virtual layers referring to the dual arbitrator defined by IBA [2]. The concept is evaluated through simulations assuming that only bandwidth sensitive traffic requests QoS. In [1] Alfaro et. al also include time sensitive traffic, besides calculating the worst case latency through various types of switching architectures.

DiffServ is foreseen to be the most prominent concept for providing QoS in the future Internet [11, 17]. DiffServ makes a distinction between boundary nodes and core nodes with respect to support of QoS features. Following the DiffServ philosophy no core switch should hold status information about passing-through traffic. Neither should there be any explicit signalling on a per flow basis to these components. This means that within the DiffServ framework any admission control or policing functionality would have to be implemented by boundary nodes or handled by a dedicated bandwidth broker. The core switches are assumed to perform traffic discrimination only based on service class, which is decided by a QoS tag included in the packet header - all packets carrying the same QoS tag will get equal treatment. From that viewpoint DiffServ is apparently a relative service model having difficulties giving absolute guarantees.

None of the previous debated contributions comply with the DiffServ model. In [12] Pelissier, however, discusses interoperation between DiffServ and IBA on a traffic class and service level basis, but refer to RSVP with respect to admission control. The strategy proposed by Alfaro et. al has to recompute the IBA dual arbitrator every time that a new connection is honored [1, 2]. Such a scheme is not associable with DiffServ. Neither is the admission control scheme presented in [29] by Yum et. al, which use hop by hop bandwidth reservations and requires recomputations of the weighted round robin scheduler at every hop towards the destination. In [25] Reinemo et. al. studied the provision of QoS in cut-through networks by adhering to the DiffServ model. The problem was approached without any explicit admission control mechanism, as a pure relative model. Empirically they examined the sensitivity of different QoS properties under various load and traffic mixture conditions, hereunder assessing the effect of flow-control. This work was further studied in [30] where the concept described in [25] was extended with three different admission control mechanisms. Our contributions showed the feasibility

of doing this at the *class level* (i.e. aggregated flows). One important question that we need to ask with regards to these results is “What happens to QoS at the *flow level*?”. Even if things look good on the class level it might not look good on the flow level. We might have a lot of good and a lot of bad flows making the overall result for the class look good, or we have large differences between all flows with a large variation in QoS. Finally, we might have each flow receiving QoS in according to what we see at the class level, which is what we actually want. Our earlier work doesn’t answer these questions so we intend to study this in detail in this paper.

The object of this paper is to study if we are able to achieve *flow level* QoS in cut-through networks by combining admission control with a class based scheme which is in compliance with the DiffServ paradigm. Spesifically, we will have a QoS concept with flow aware admission control and flow negligent traffic classes. Empirically we will study the *throughput*, *latency* and *jitter* characteristics at the flow level, all in combination with three different admission control mechanisms each with a fundamentally different approach to admission control. Our first scheme assumes pre-knowledge of the network’s performance behavior without admission control and is implemented as a centralised bandwidth broker. Our second scheme is based on endpoint/egress measurements to assess the load situation, and our third scheme makes use of probe packets to assess the load situation.

Our results are important in two ways. Firstly, they are important as a means to achieve QoS in cut-through networks. Secondly, they are important to bridge QoS between the global Internet and a local cluster. If IETF standards such as DiffServ or IntServ are applied for some applications on the Internet we need ways to represent these QoS attributes on our cluster to be able to serve the appliaction request according to their specified QoS.

The rest of this paper is organised as follows. In section 2 we give a description of our QoS architecture, in section 3 our three admission control mechanisms are described, and in section 4 our simulation scenario is described. We discuss our performance results in section 5 , and in section 6 we finish off with some concluding remarks.

2 QoS Architecture

The architecture used in our simulations is inspired by IBA link layer technology [4] and is a flit based virtual cut-through (VCT) switch. The overall design is based on the canonical router architecture described in [6].

In VCT the routing decision is made as soon as the header of the packet is received and if the necessary resources are available the rest of the packet is forwarded directly to the destination link [23]. If the necessary resources are busy the packet is buffered in the switch. In addition we use flow control on all links so all data is organised as flow control digits (flits) at the lowest level.

The switch core consists of a crossbar where each link and VL has dedicated access to the crossbar. Each link supports one or more virtual lanes (VL), where each VL has its own buffer resources which consist of an input buffer large enough to hold a packet and an output buffer large enough to hold two flits to increase performance. Output link arbitration is done in a round robin fashion.

To achieve QoS our switch architecture support QoS mechanisms similar to the ones found in the IBA architecture. IBA supports three mechanisms for QoS which are mapping of service level (SL) to VL, weighting of VLs and prioritising VLs as either low priority (LP) or high priority (HP). A more detailed description of these QoS aspects can be found in [25].

2.1 Layered shortest path routing

LASH is a deterministic routing algorithm that guarantees true shortest path routing and in-order delivery in both regular and irregular networks [16]. It achieves this through the use of virtual layers without any need for more advanced functionality in the switches. It fits well with our simple approach to QoS and we have used it in all our simulations.

The idea is that each virtual layer in the network has a set of source/destination pairs assigned to it, in such a way that all source/destination pairs are assigned to exactly one virtual layer. In addition it makes sure that each virtual layer is deadlock free by ensuring that the channel dependencies stemming from the source/destination pairs of one layer do not generate cycles. An in depth descriptions of LASH is found in [16].

3 Admission Control

In this section we propose three different admission control (AC) mechanisms that we carefully evaluate in section 5.

3.1 Link by Link Based Admission Control

In the Link-by-Link (LL) approach a bandwidth broker (B) knows the load on every link in the network and will consult the availability of bandwidth on every link between source and destination before accepting or rejecting a flow. This solution assumes that both topology and routing information about the network is available.

For the AC decision we adopt the *simple sum* approach as presented in [28]. This algorithm states that a flow may be admitted if its peak rate p plus the peak rate of the already admitted flows s is less than the link bandwidth bw. Thus the requested flow will be admitted if the following inequality is true [28]

$$p + s < bw \tag{1}$$

We deduce the effective bandwidth from the measurements obtained in [25]. Since we are dealing with service levels where each SL has different bandwidth requirements it is natural to introduce some sort of differentiation into equation (1). We achieve this by dividing the link bandwidth into portions relative to the traffic load of the SLs, and include only the bandwidth available to a specific service level bw_{SL} in the equation as follows

$$p + s_{SL} < bw_{SL} \tag{2}$$

where

$$bw_{SL} = bw_{link} * \frac{load_{SL}}{load_{total}} \tag{3}$$

and s_{SL} is the sum of the admitted peak rates for service level SL and bw_{link} is the effective link bandwidth.

3.2 Egress Based Admission Control

The Egress Based (EB) approach is a fully distributed AC scheme where the egress nodes are responsible for conducting the provisions. Basically, we adopt the Internet AC concept presented by Cetikaya and Knightly in [26]. This method does not assume any pre-knowledge of the network behaviour as was the case with the previous solution. Also different from the previous approach is the use of a delay bound as the primary AC parameter. For clarity we give a brief outline of the algorithm below, a more detailed description can be found in [27].

The method is entirely measurement based and relies on that the sending nodes timestamp all packets enabling the egress nodes to make two types of measurements. First, by dividing time into timeslots of length τ and counting the number of arriving packets, the egress nodes can deduce the arrival rate of packets in a specific timeslot. By computing the maximum arrival rate for increasingly longer time intervals we get a peak rate arrival envelope $R(t)$, where $t = 0, ..., T$ timeslots, as described in [26]. Second, by comparing the originating timestamp relative to the arrival time, the egress node can calculate the transfer time of a packet. Having this information available the egress node can furthermore derive the time needed by the infrastructure to service k following packets; i.e. a consecutive stream of packets where the next packet in the service class enters the infrastructure before the previous packet has departed the egress node. By doing this for larger and larger k sequences of packets within a measuring interval of length $T\tau$ and subsequently inverting this function we achieve the service envelope $S(t)$, giving the amount of packets processed by the network in a given time interval t. Now repeating this M times, the mean $\overline{R}(t)$ and the variance $\sigma^2(t)$ of $R(t)$, and the mean $\overline{S}(t)$ and variance $\Psi^2(t)$ of $S(t)$ may be calculated. If a flow request has a peak rate P and a delay bound D it may be accepted if the peak rate P plus the measured arrival rate $R(t)$ is less than the service rate allowing for the delay $D, S(t + D)$.

The EB scheme derives its knowledge of the network from measurements of the traffic passing through the egress nodes. It is therefore difficult for the egress nodes to have a complete picture of the load in the network, moreover the packet latency is used to infer the network load utilising the fact that an increased network load will cause an increase in latency as well. From that viewpoint it seems difficult to give a service class bandwidth guarantees since it has no concrete knowledge of the network load. The algorithm will admit as much traffic as it can without breaking the delay bound. The key instrument of the scheme is the given delay bound for the different flows, and the efficiency of the algorithm is linked to its ability to limit the service levels to operate within the delay bounds.

3.3 Probe Based Admission Control

As an alternative to passively monitoring the network activity in the egress nodes of the network, as was the case with the EB scheme, it is possible for the end nodes in the network to take a more active role in the AC decision. This can be done by actively sending probe packets through the network from source to destination and monitor the arrival of the probes at the egress of the network [36,37]. If the size and rate of the probe packets is designed correctly they should give the egress node the opportunity to calculate how the new flow will be treated by the network. Several probing schemes have been proposed in the litterature, some of which are described in [36,37]. The approach in [37] require either that packets be dropped to indicate congestion or that congested packets be marked in the switches in the network. Packet dropping is implausible in virtual cut-through networks such as Infiniband since packets are not dropped, but flow-controlled. Explicit marking of packets requires intelligence in the switches and partly avoids the whole point of end-point admission control. In [36] however Bianchi et.al. propose a probing

scheme where the load is infered by measuring the jitter for the probe packets. They require that the probe packets are forwarded through the network with the lowest priority of all packets. This ensures that the probe packets will be unable to steal bandwidth from the already existing traffic in the network whilst additionally giving worst-case measurements of the network jitter and thus guaranteeing that the traffic, when admitted, will get at least the service of the probe packets. When applying this in our simulation scenario it is natural to let the probe traffic be forwarded on one of the low priority SLs with a relatively low weight, possibly equal to 1. The AC decision is as follows

$$TransmissionTime_{max} - TransmissionTime_{min} < Jitter_threshold \quad (4)$$

and

$$Packets_rejected = 0 \quad (5)$$

For each probe packet received the receiver registers the packet's transmission time, e.g. the time the packet spends in the network. When an adequate amount of probe packets have been sent and received the receiver calculates the jitter by substracting the minimum packet transmission time from the maximum packet transmission time. This value is compared to the jitter requirements embedded in the probe packets and an admission decision is sent back to the sender. If the percived jitter was less than the requirement the flow is acccepted, otherwise the flow is rejected. Additionally if any of the probe packets are rejected by the sender due to the limited size of the send queue buffer the flow is also rejected.

3.4 Target for Admission Control

The main findings for the work in [25] are that (i) throughput differentiation can be achieved by weighting of VLs and by classifying the VLs as either low or high priority, (ii) the balance between VL weighting and VL load is not crucial when the network is operating below the saturation level. In general this sets the target for the AC, since as long as we can ensure that the load of the various service classes is below saturation level we can also guarantee that each of these classes get the bandwidth they request. The target for admission control is thus the point where the amount of accepted traffic is starting to become less than the traffic offered. The effective bandwidth at this point will be used as a steering vehicle by the LL method. The success of this method is documented in [30].

Another main finding in [25] is that although the latency characteristics below saturation were fairly good, significant jitter was observed. We challenged this problem in [30] with the EB method, unfortunately the EB scheme was unable to give bandwidth guarantees and the latency results was slightly worse than the LL scheme at the class level. We have included the EB scheme for comparison only, and added the PB scheme in order to improve the latency and jitter performance.

4 Simulation Scenario

For all simulations we have used a flit level simulator developed in house at Simula Research Laboratory. In the simulation results that follow all traffic is modelled to display self-similair behaviour (see section 4.1). We have performed simulations on a network with 32 switches, where each switch is connected to 5 end nodes and the maximum number of links per switch is 10 in addition to the end nodes. We have randomly generated 16 irregular topologies and we have run measurements on these topologies

SERVICE LEVELS				
SL	DS[1]	Load %	BW[2]	Pri
1	EF	10	4	high
2	EF	15	6	high
3	AF	20	8	low
4	AF	25	10	low
5	BE	30	1	low

Table 1. The five services levels used in simulation.

at increasing load. We use LASH [16] as our routing algorithm and random pairs as our traffic pattern. In the random pairs scheme each source sends to *one* destination and no destination receives from more that *one* source. The link speed is one flit per cycle, the flit size is one byte and the packet size is 32 bytes for all packets.

The five different end nodes send traffic on one of five different service levels. One service level for each node (Table 1), SL 1 and 2 are considered to be of the expedited forwarding (EF) class in DiffServ terminology,SL 3 and 4 are considered to be of the assured forwarding (AF) class. SL 5 is considered as best effort (BE) traffic and from that viewpoint is not a subject for AC.

For the LL scheme all simulations where run with a target load deduced from our measurements in [25]. In the first part of the simulation the send rate is steadily increased by adding more and more flows until admission is denied by the AC scheme. When this happens the current rate is not changed, but the node will try to add more flows for a fixed number of times before it gives up. For the EB scheme the send rate is increased in the same way, but the AC decision is primarily based on measured latency as described in section 3.2. For the PB scheme the AC decision is based on the measured jitter for the probe packets as described in section 3.3.

4.1 Self-similair Traffic

Analyses of real-life network traffic traces have shown that the arrival of each packet in the network is not totally independent of the arrival of any other packet such as in a Poisson process. Instead the arrival patterns display a degree of self similarity where the traffic is repeated on smaller and larger time scales in accordance with fractal theory [33, 34]. Several papers have been written on how to efficiently simulate long-range dependencies in network traffic. One of the findings is that such traffic can be modelled by a process with a finite mean and infinite variance [35]. In [35] Willinger et.al. show that an aggregation of Pareto distributed on/off sources are within the necessary mathematical criterion to produce self-similar traffic. This is the approach we have adopted in this paper.

5 Performance Results

We will now discuss our results with regards to throughput, latency and jitter, all in that order.

[1] The DiffServ equivalent service class.

[2] The maximum number of flits allow to transmit when scheduled.

5.1 Throughput

As was presented in section 4 our traffic is divided into five different classes. The throughput results for these classes are presented in figure 1, while throughput results for flows are presented in figure 2. Figure 1(a) shows the throughput achieved without any form of admission control (NoAC). We observe that we are unable to give all service classes the requested bandwidth as we enter saturation. Furthermore, the high priority (HP) classes preempts the low priority (LP) classes, i.e. the bandwidth differentiation is no longer according to the percentages in Table 1. This behaviour is reflected at the flow level in figure 2(a) where we see that the throughput per flow is decreased as the number of flows is increased. With this in mind we will evaluate each of our proposed AC schemes.

Our first candidate is the probe based (PB) scheme where jitter is the primary AC parameter. From figure 1(b) we observe that this scheme performs very well. The admission control decision is very precise about when to accept and reject traffic and we see bandwidth differentiation that is relative to the actual requests. In other words we are able to give bandwidth guarantees with this scheme. We are also able to utilise the network resource well since we get close to the saturation point without passing it. Looking at flow level througput we see from figure 2(b) that all flows get the requested bandwidth at the cost of less bandwidth for best effort traffic in SL5.

The next candidate is the EB scheme using latency as the primary AC parameter. It is apparent from figure 1(c) that the EB is unable to give bandwidth guarantees at the class level. The load is allowed to increase beyond the saturation point and admits too much traffic. In addition the differentiation between SLs detoriates as the load increases. The poor performance[3] of the EB scheme can be ascribed to the use of delay as the primary AC parameter, this results in the bandwidth requirements being ignored. At the flow level the EB scheme gives all accepted flows the requested bandwidth (figure 2(c)), but the number of flows accepted in each class is not differentiated in a way relative to the actual requests.

Our final candidate is the LL scheme which use bandwidth as the primary AC parameter. The LL scheme presented in figure 1(d) shows several improvements compared to the mediocre performance of the EB method. It is actually as good as the PB approach. The admission control is precise about when to accept and deny admission. We achieve a bandwidth differentiation which is relative to the actual requests, meaning that we can give bandwidth guarantees. Moreover, we are able to utilise the network resource well since we go close to the saturation point without passing it. The good performance of the LL can be attributed to the fact that it knows the load of every link in the network and is able to make the AC decision based on the load along the actual source/destination path. Comparing figure 1(b) and 1(d) in detail we observe that the PB scheme gives slightly higher throughput to most classes. This is caused by the slightly higher load applied in the PB scheme to balance the lack of normal SL4 traffic. At the flow level the LL approach is as good as the PB scheme. It is able to give all flows the requested bandwidth at the cost of less bandwitdh available to best effort traffic in SL5.

5.2 Latency

In the previous section we saw that we are able to give bandwidth guarantees with both the PB and the LL scheme at both the class and flow level. Now we will study the ability to guarantee latency at the flow level, typically we want to have low latency for flows in the high priority SLs, while we accept

[3]As a sidenote, the performance of the EB scheme is worse when using self-similair compared to the use of a Poisson process as was the case in [30].

higher latency values for the low priority SLs. For the best effort traffic in SL5 we dont't care about latency.

Figure 3(a) shows the average per flow latency for increasing load values without admission control. Comparing this with the results from PB in figure 3(b) shows that there is an improvement in latency of about one order of magnitude for HP flows, furthermore the differentiaton between flows from different SLs is good. For the EB scheme in figure 3(c) the results are similar, but the improved latency is not as low as is the case for the PB method. On the other hand the differentiation between flows in different SLs is better. Still, the EB scheme is unable to achieve latency as low as the other schemes even if its using latency as the primary AC parameter. The reason for this is probably the unpredictable latency characteristics in cut-through networks as observed in [25]. Finally the LL scheme is able to get an improvement in latency on par with the PB scheme. From figure 3(d) we see an improvement in latency of more than one order of magnitude. It is also able to diffferentiate better between flows in different SLs than both the PB and EB scheme. The good performance of the LL scheme can be ascribed to its detailed knowledge about the network.

5.3 Jitter

Our third QoS attribute is the variation in latency, also refered to as jitter. We want our jitter to be as low as possible and to better evaluate this we have plotted the maximum observed jitter for all our AC schemes in figure 4. The plots contain the maximum per flow jitter observed throughout the simulation run.

The NoAC results show that there is substantial increase in jitter for all flows even at very low load. With the introduction of the PB scheme jitter is reduced by one order of magnitude for high priority flows (figure 4(b)). For low priority flows the reduction is slightly less. The primary AC parameter for this method is jitter and thus it performs well. Still, even if we are able to reduce jitter significantly our guarantees are coarse since jittering is still in the order of several hundred cycles.

The EB scheme performs only slightly worse that the PB approach with higher jitter and less difference between SLs. This is understandable as the EB method focus on latency instead of jitter.

Moving on to our last candidate, the LL mechanism in figure 4(d), we see an improvement in overall jitter. But the improvement is worse than what is the case for both the EB and PB scheme. Furthermore, the jitter in flows from SL1 and SL4 shows a large amount of variation compared to both EB and PB approach. This is probably caused by the fact that this scheme ignores latency and jitter properties and only concentrates on throughtput when making the AC decisions.

6 Conclusion

In this paper we set out to achieve flow level QoS with regards to throughput, latency and jitter. Our goal was to achieve this by only using flow aware admission control in combination with a flow negligent DiffServ inspired QoS mechanism. Towards this goal we have evaluated three different admission control schemes for virtual cut-through networks. These three schemes represents three different approaches to admission control. One is a probe based scheme using jitter as the primary AC parameter, another is a measurements based approach using lantecy as the primary AC parameter and yet another is a centralized bandwidth broker approach using pre-knowledge of the network link load without admission control as the primary AC parameter.

Our contributions are as follows. First, flow level bandwidth guarantees are achievable with the use of the Link-by-Link and the Probe based schemes, while the Egress Based method is unable to achieve good guarantees. Second, improved per flow latency and jitter properties are achievable with both the Probe and Egress based methods, but strict guarantees are hard to give since jitter is still high. Overall, the Probe based scheme gives us the best performance with regards to throughput, latency and jitter. The final conclusion is that we are able to achieve flow level QoS with a combination of DiffServ and admission control in cut-through networks.

References

[1] F. J. Alfaro, J. L. Sanchez, J. Duato, and C. R. Das. A strategy to compute the InfiniBand arbitration tables. In *Proceedings of International Parallel and Distributed Processing Symposium*, April 2002.

[2] F. J. Alfaro, J. L. Sanchez, and J. Duato. A strategy to manage time sensitive traffic in InfiniBand. In *Proceedings of Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet – a gigabit-per-second LAN. IEEE MICRO, 1995.

[4] InfiniBand Trade Association. Infiniband architecture specification.

[5] Differentiated Services. RFC 2475.

[6] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks an engineering approach*, IEEE Computer Society, 1997.

[7] R. W. Horst. Tnet: A reliable SAN. *IEEE Micro*, 15(1):37–45, 1995.

[8] Integrated Services. RFC 1633.

[9] J. Jaspernite, and P. Neumann. Switched Ethernet for Factory Communication. In *Proceedings of* 8th *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'01)*, 205–212, October 2001.

[10] J. Jaspernite, P. Neumann, M. Theiss, and K. Watson. Deterministic real-time communication with switched Ethernet. In *Proceedings of* 4th *IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.

[11] K. Kilkki. Differentiated services for the Internet. *Macmillian Tech. Publishing*, 1999.

[12] J. Pelissier. Providing quality of service over InfiniBand^{TM} architecture fabrics. In *Proceedings of Hot Interconnects X*, 2000.

[13] ReSource ReserVation Protocol. RFC 2205.

[14] M. D. Schroder et.al., "Autonet: a high-speed, self-configuring local area network using point-to-point links," SRC Research Report 59, Digital Equipment Corporation, 1990.

[15] T. Skeie, J. Johannessen, and Ø. Holmeide. The road to an end-to-end deterministic Ethernet. In *Proceedings of* 4th *IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.

[16] T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (LASH) routing in irregular system area networks. In *Proceedings of Communication Architecture for Clusters*, 2002.

[17] X. Xiao and L. M. Ni. Internet QoS: A Big Picture In *IEEE Network Magazine*, 8–19, March/April 1999.

[18] J. S. Yang and C. T. King, "Turn-restricted adaptive routing in irregular wormhole-routed networks," in *Proceedings of the 11'th International Symposium on High Performance Computing (HPCS97)*, July 1997.

[19] A. A. Chien and J. H. Kim, "Approaches to Quality of Service in High-Performance Networks," in *Lecture Notes in Computer Science*, vol. 1417, 1998.

[20] J. Duato and S. Yalamanchili and B. Caminero and D. S. Love and F. J. Quiles, "MMR: A High-Performance Multimedia Router - Architecture and Design Trade-Offs," in *HPCA*, pages 300-309, 1999.

[21] B. Caminero, C. Carrion, F. J. Quiles, J. Duato and S. Yalamanchili, " A Solution for Handling Hybrid Traffic in Clustered Environments: The MultiMedia Router MMR," in *Proceedings of IPDPS-03*, April 2003.

[22] M. Gerla and B. Kannan and B. Kwan and E. Leonardi and F. Neri and P. Palnati and S. Walton, "Quality of Service Support in High-Speed, Wormhole Routing Networks," in *International Conference on Network Protocols (ICNP'96)*, 1996.

[23] P. Kermani and L. Kleinrock, "Virtual Cut-through: A New Computer Communication Switching Technique," in *Computer Networks*, no. 4, vol. 3, 1979.

[24] R. Seifert, *Gigabit Ethernet,* Addison Wesley Pub Co., 1998.

[25] S. A. Reinemo and T. Skeie and O. Lysne, "Applying the DiffServ Model in Cut-through Networks," in *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2003.

[26] C. Cetikaya and E. W. Knightly, "Egress admission control," in *INFOCOM (3)*, pages 1471-1480, 2000.

[27] J. Schlembach and A. Skoe and P. Yuan and E. Knightly, "Design and Implementation of Scalable Admission Control," in *QoS-IP*, pages 1-15, 2001.

[28] S. Jamin and S. J. Shenker and P. B. Danzig, "Comparison of Measurement-Based Admission Control Algorithms for Controlled-Load Service," in *INFOCOM (3)*, pages 973-980, 1997.

[29] K. H. Yum, E. J. Kim, C. R. Das, M. Yousif, J. Duato, "Integrated Admission and Congestion Control for QoS Support in Clusters," in *Proceedings of IEEE International Conference on Cluster Computing*, pages 325-332, September 2002.

[30] S. A. Reinemo and Frank Olaf Sem-Jacobsen and T. Skeie and O. Lysne, "Admission Control for DiffServ based Quality of Service in Cut-through Networks," to appear in *Proceedings of the 10th International Conference on High Performance Computing, 2003.*

[31] C.Bell et.al., "An Evaluation of Current High-Performance Networks," in Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.*

[32] R. Seifert, The Switch Book: The Complete Guide to LAN Switching Technology, *John Wiley & Sons, Inc., 2000.*

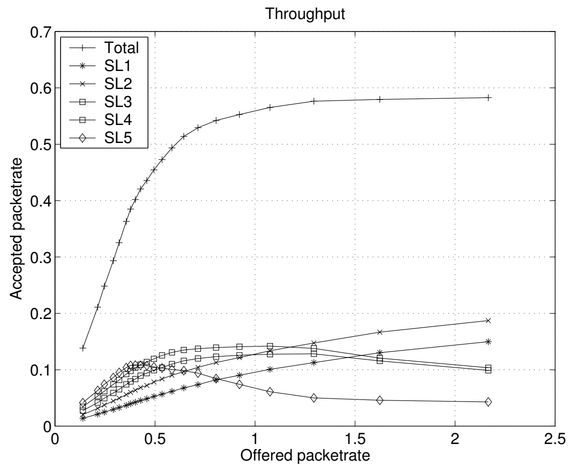
[33] W. E. Leland et.al., "On the self-similar nature of Ethernet traffic," in Proceedings of the ACM Sepcial Interest Group on Data Communications*, 1993.*

[34] V. Paxson and S. Floyd, "Wide area traffic: The failure of Poisson modeling," in IEEE/ACM Transaction on Networking*, 3(3):226-244, 1995.*

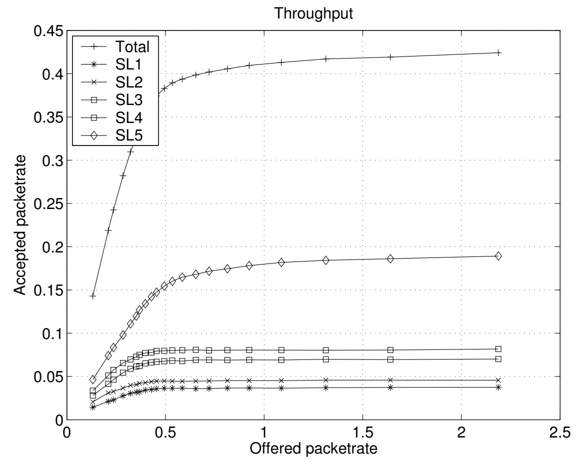
[35] W. Willinger et.al. "Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level," in IEEE/ACM Transaction on Networking*, 5(1):71-86, 1997.*

[36] G. Bianchi and F. Borgonovo and A. Capone and L. Fratta and C. Petrioli "Endpoint ad-mission control with delay variation measurements for qos in ip networks," in ACM SOGCOMM Computer Communications Review*, 32(2):61-69, 2002.*

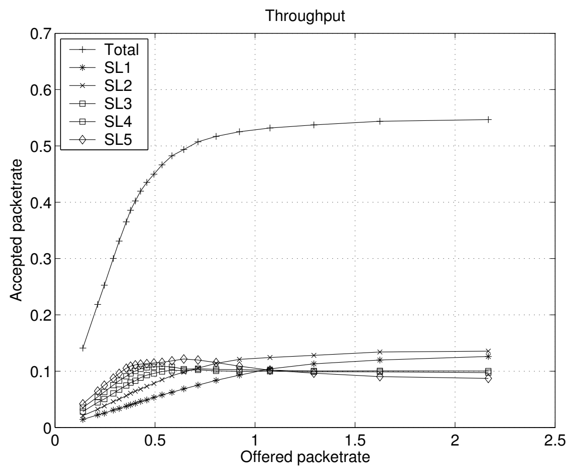
[37] Lee Breslau and Edward W. Knightly and Scott Shenker and Ion Stoica and Hui Zhang "End-point admission control: architectural issues and performance," in Proceedings of the ACM Sepcial Interest Group on Data Communications*, 2000.*



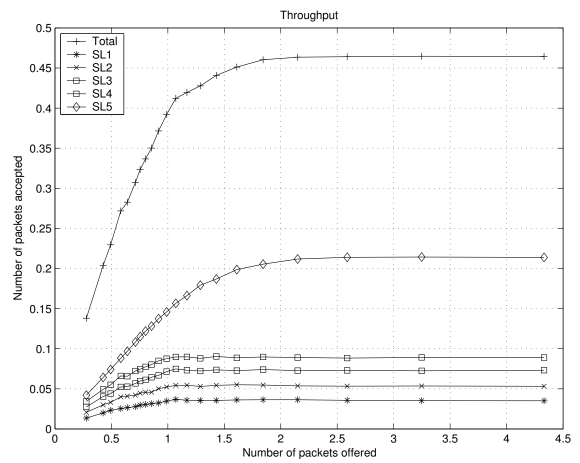
(a) No admission control



(b) Probe

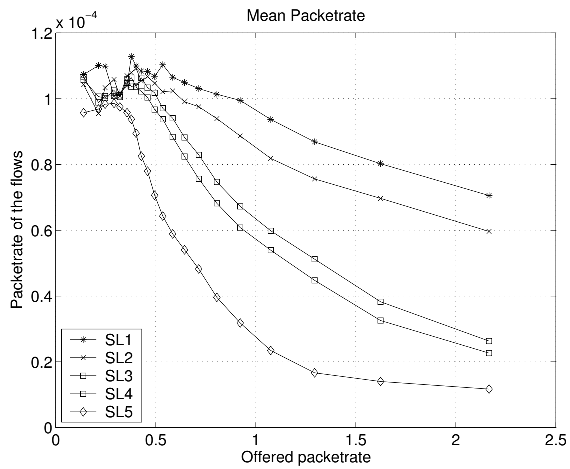


(c) Egress

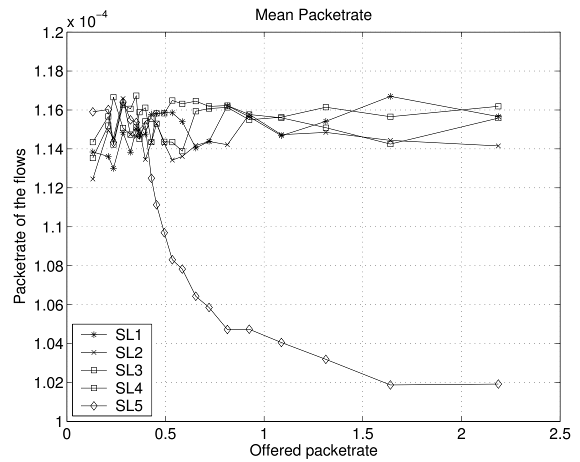


(d) Link by link

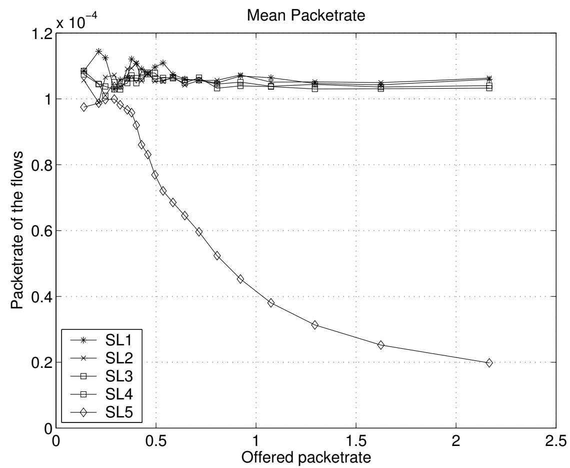
Figure 1. Average class throughput



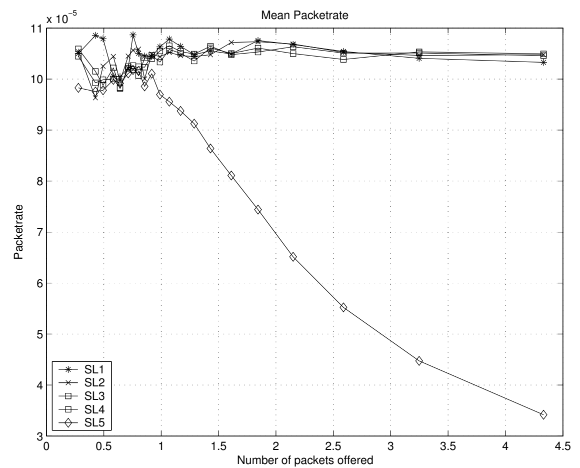
(a) No admission control



(b) Probe

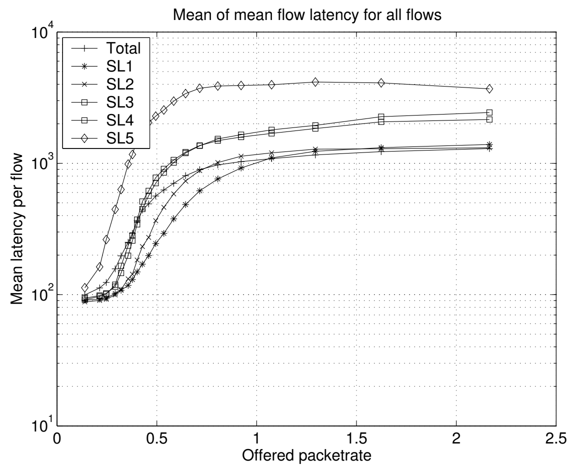


(c) Egress

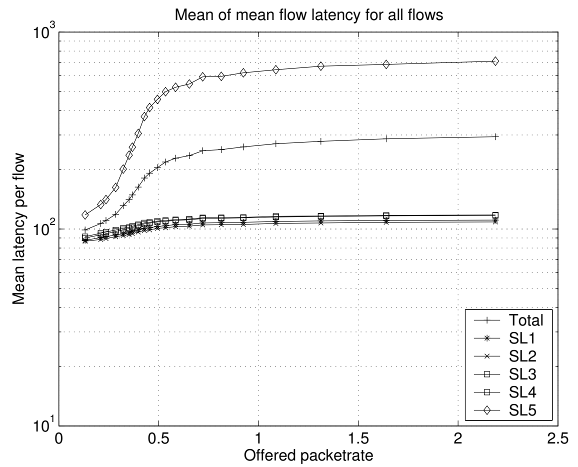


(d) Link by link

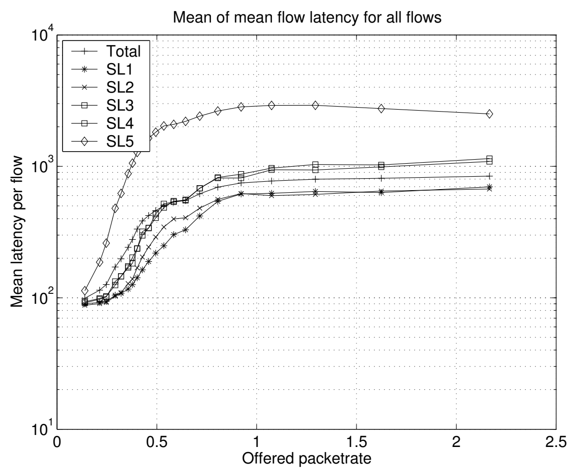
Figure 2. Average flow packetrate



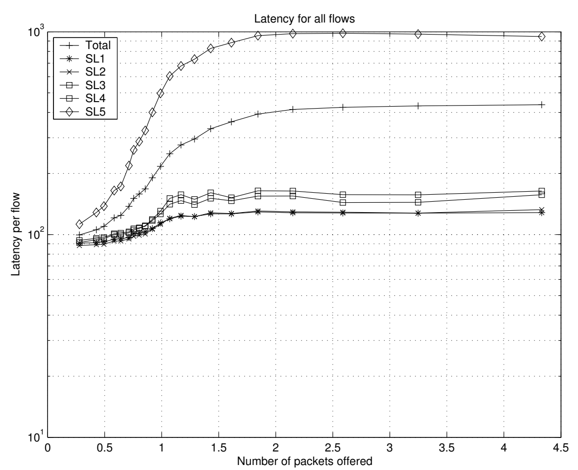
(a) No admission control



(b) Probe

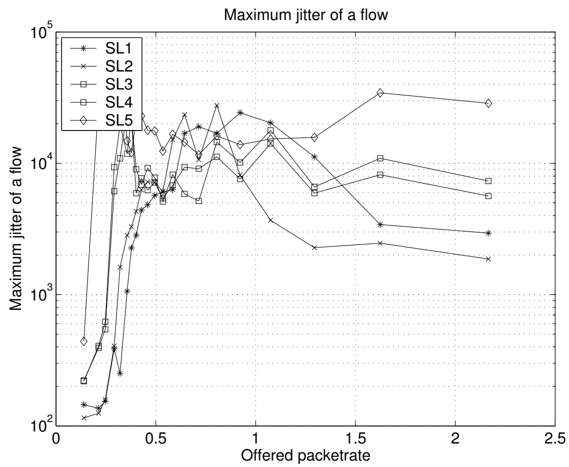


(c) Egress

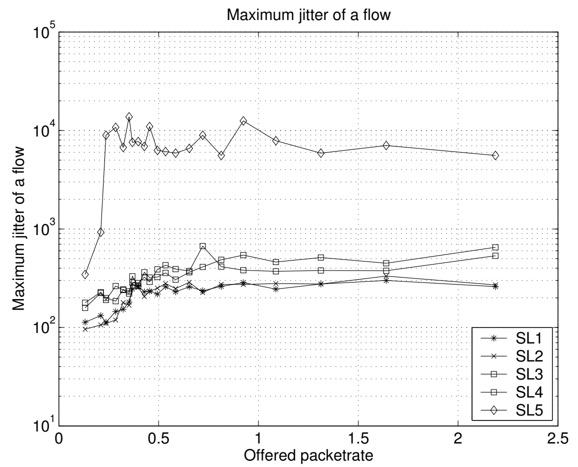


(d) Link by link

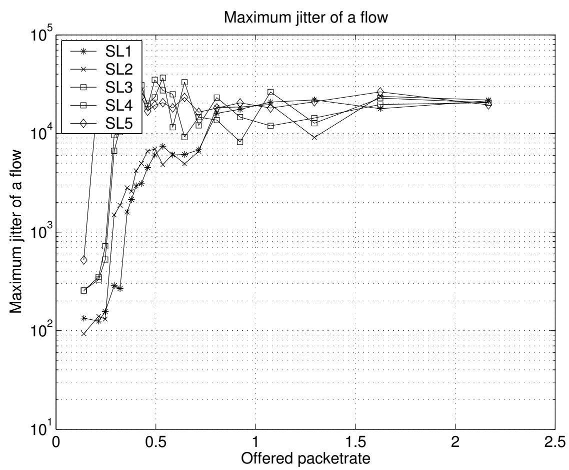
Figure 3. Average flow latency



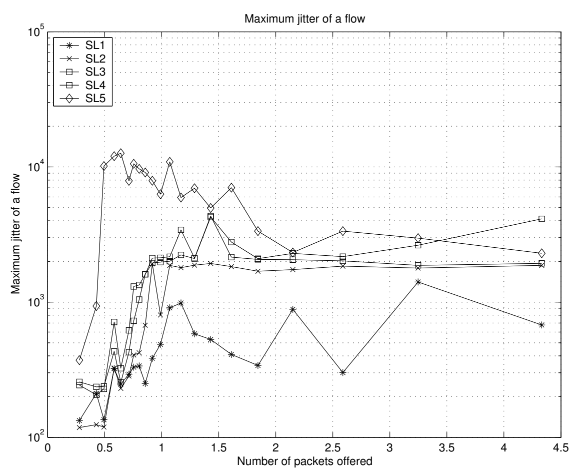
(a) No admission control



(b) Probe



(c) Egress



(d) Link by link

Figure 4. Maximum flow jitter

Appendix C

Simulator Source Code

What follows is the source code which I have produced for the simulator. The simulator itself consists of several thousand lines of code, much of this has therefore been omitted. The source code presented contains all the code relating to admission control and traffic generation. The code presented here is only my additions to the already existing simulator framework. It is included for completeness so that one may verify that the algorithms are implemented as they are described in the thesis. For more information about the simulator and the entire source code, feel free to contact me.

The basic setup is that Processors are connected to Switches which are connected to other Switches and Processors via HalfLinks which support several VLanes. The admission control for EM is implemented mostly in MSProcessor. Jitter Probing is implemented in PROBEProcessor, while LBL is implemented partly in BBRMSProcessor and BBRMSSwitch. The available bandwidth calculation is performed in VLane and stored in HalfLink. The self-similar traffic generation routines are implemented in ParGen, PSource and Flow. Payload is used for admission control information in the network packets, mostly time stamps for use in EM and Jitter Probing.

```
1   /*
2    * Processor.java:  This class implements the basic processor in the network.
3    * It is responsible for producing and consuming packets.  It is subclassed to
4    * implement various AC algorithms.
5    */
6
7   package base;
8
9   import java.util.Vector;
10
11  public class Processor extends Node {
12
13      public static final boolean DEBUG = false;
14      public static final boolean DEBUG2 = false;
15      public static final boolean DEBUG5 = false;
16      public static final boolean DEBUG9 = false;
17      public static final boolean DEBUG33 = false;
18      public static final boolean DEBUG11 = false;
19
20      // decide the SL for this Processor (modulo number_of_servicelevels)
21      public static int nextSL = 0;
22      public int localSL = 0;
23      public int selfNumToSend = -1;
24      public HalfLink[] drop_links;
25      public Connectable[] drop_connects;
26      public SendQueue send_q;
27      public Vector bind_q;
28      public int service_time;
29      public int receive_buffers;
30      public int bound_receive_buffers;
31      public int in_transmission = 0;
32      // counts the number of packets sent as part of this IBA message
33      public int iba_message_packetcount = 0;
34      public int iba_msg_size = 10;
35      public int iba_last_sl = 0;
36      public Processor iba_last_dest = null;
37      //self-similar flows
38      public double flowBW;
39      public static parGen pareto;
40      public Flow[] flows;
41      public double localOfferedFlows = 0;
42      protected double maxFlows;
43      public static boolean initdone = false;
44      //adctr
45
46      private static int[] procNum =
47          new int[Kernel.CV[Kernel.highpri_service_levels] +
48                  Kernel.CV[Kernel.lowpri_service_levels]];
49      public int adctrlID = -1;
50      private int[] pack_to_drop = new int[16];
51      public Processor (int id) {
52          this (id,
53                Kernel.CV[Kernel.Processor_receive_buffers],
54                Kernel.CV[Kernel.Processor_send_q_size],
55                Kernel.CV[Kernel.Processor_service_time],
56                Kernel.CV[Kernel.Processor_drop_connects],
57                Kernel.CV[Kernel.Node_drop_links]);
58      }
59
60      public Processor (int id,
61                        int receive_buffers,
62                        int send_q_size,
63                        int service_time, int drop_connects, int drop_links) {
64          this.id = id;
65          //self-similar, initiate the pareto generator
66          if (!initdone) {
67              pareto = new parGen ( /*Kernel.rand.nextLong() */ 2341534);
68              initdone = true;
69          }
70
71          this.receive_buffers = receive_buffers;
72          this.bound_receive_buffers = 0;
73          this.service_time = service_time;
74          this.send_q = new SendQueue (send_q_size);
75          this.bind_q = new Vector (send_q.size, 1);
```

```
76          this.drop_links = new HalfLink[drop_links];
77          this.drop_connects = new Connectable[drop_connects];
78          for (int i = 0; i < drop_connects; i++)
79              this.drop_connects[i] = new Connectable (i);
80
81          this.localSL = nextSL % (Kernel.CV[Kernel.highpri_service_levels]
82                                 + Kernel.CV[Kernel.lowpri_service_levels]);
83          //System.out.println(this.toString() + " sends SL " + localSL);
84          nextSL++;
85          //adctrl
86          adctrlID = procNum[this.localSL];
87          procNum[this.localSL]++;
88      }
89
90
91      public String toString () {
92          return " Processor [" + id + "]";
93      }
94
95      // the producer state procedures, see the process model processor_producer
96
97      public Packet createPacket (Processor src, int pid,
98                                  Processor dst, int payload, int sl) {
99          return new Packet (src, pid, dst, payload, sl);
100     }
101
102     public int pid = Kernel.max_switches;
103     public void packet_produced_enter (EasterEgg egg) {
104         Packet p = null;
105         int sl = 0;
106
107         sl = localSL;
108
109         if (DEBUG)
110             System.out.println (toString () + " packet_produced_enter");
111
112         if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null) {
113             // first msg send from this processor or iba_msg not enabled
114             p = createPacket (this,
115                               pid + id,
116                               Kernel.kernel.packetDestination (this),
117                               Kernel.Traffic.payload_size (id), sl);
118             iba_last_dest = p.destination;
119             iba_last_sl = p.sl;
120         } else {
121             if (iba_message_packetcount > iba_msg_size) {
122                 // reset counter
123                 iba_message_packetcount = 0;
124                 // starting a new message with a new destination
125                 p = createPacket (this,
126                                   pid + id,
127                                   Kernel.kernel.packetDestination (this),
128                                   Kernel.Traffic.payload_size (id), sl);
129                 iba_last_dest = p.destination;
130                 iba_last_sl = p.sl;
131             } else {
132                 // continuing and existing message
133                 p = createPacket (this,
134                                   pid + id,
135                                   iba_last_dest,
136                                   Kernel.Traffic.payload_size (id), iba_last_sl);
137
138             }
139         }
140
141         pid += Kernel.max_switches;
142         if (send_q.full ()) {
143             Packet.packets_rejected[p.sl]++;
144             Packet.incRejected (p.hops, p.sl);
145         }
146         send_q.insert (p);
147         if (in_transmission < drop_connects.length) {
148             send_enter (null);
149         }
150
```

```
151             if (!Kernel.stop_packet_generation) {
152                 if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null
153                         || iba_message_packetcount > iba_msg_size) {
154                     //        (new Event(this, PACKET_PRODUCED))
155                     //          .schedule(Kernel.Now + Kernel.Traffic.packet_interarriv
    al_time(id));
156                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now
157                             +
158                             Kernel.Traffic.
159                             packet_interarrival_time
160                             ((int) Kernel.
161                              slLoad[localSL],
162                              Kernel.CV[Kernel.
163                                        std_dev_packet_interarrival_time]));
164                 } else {
165                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now + 10);
166                 }
167             }
168 
169             // keep track of number of packets send in this msg
170             if (Kernel.IBA_ENABLE_MESSAGE)
171                 iba_message_packetcount++;
172         }
173 
174         public void send_enter (EasterEgg egg) {
175             if (DEBUG)
176                 System.out.println (toString () + " send_enter");
177             (new Event (this, CONNECTABLE)).schedule (Kernel.Now, connectable_flag);
178         }
179 
180         // the connecter state procedures, see the process model processor_connecter
181 
182         public void connectable_enter (EasterEgg egg) {
183             if (DEBUG)
184                 System.out.println (toString () + " connectable_enter");
185 
186             Packet p = send_q.firstUnConnectedElement ("Processor");
187 
188             while (p != null && in_transmission < drop_connects.length) {
189 
190                 Connectable connector = null;
191                 // we guess that there are generally not more than two link
192                 // alternatives, and try to optimize size of the vector.
193                 Vector alternative_vlanes =
194                     new Vector (drop_links[localSL].vlinks * 2,
195                                 drop_links[localSL].vlinks);
196 
197                 int ii;
198                 for (ii = 0; ii < drop_connects.length; ii++) {
199                     if (drop_connects[ii].packet == null) {
200                         connector = drop_connects[ii];
201                         break;
202                     }
203                 }
204 
205                 if (Kernel.LOGICAL_ERROR_CHECK) {
206                     if (connector == null) {
207                         System.out.println (toString () + ": No connector found");
208                         System.exit (0);
209                     }
210                 }
211                 // how do we find alternative vlanes? for now, we just use the
212                 // drop link at the same index as drop_connects :D
213 
214                 // select vlanes according to SLs
215                 int nVL = Kernel.CV[Kernel.vlinks_per_sl];
216                 int nSL = Kernel.CV[Kernel.highpri_service_levels]
217                         + Kernel.CV[Kernel.lowpri_service_levels];
218 
219                 nSL = (p.sl * nVL) + nVL;
220 
221                 for (int i = p.sl * nVL; i < nSL; i++) {
222                     alternative_vlanes.addElement (drop_links[ii + localSL].
223                                                    vlanes[i]);
224                 }
```

```
225
226
227                     //ADMISSION CONTROLL
228                     //send reservation- and confirmation- messages on ctrl_vlane to speed up
        the process
229                     /*
230                      if(Kernel.RMS_ADMISSION_CONTROL && p.payloadpointer.resvinit || p.payloa
        dpointer.resvok || p.payloadpointer.resvnotok){
231                      alternative_vlanes.clear();
232                      alternative_vlanes.addElement(drop_links[ii + localSL].ctrl_vlane);
233                      }
234                      */
235
236                 alternative_vlanes = Kernel.kernel.shuffle (alternative_vlanes);
237
238                 for (int i = 0; i < alternative_vlanes.size (); i++) {
239                     VLane outvl;
240
241                     outvl = (VLane) (alternative_vlanes.elementAt (i));
242
243
244
245                     // note: this check for letting a packet onto a link
246                     // implements "late release" of a vlane... that is, no vlane
247                     // may hold two different packets simultaneously
248
249                     // early release is a very complex feature to implement; it
250                     // involves the routing function, and any killing of packets
251                     // and flushing links is much more complex and will require
252                     // much more testing!  early release requires room for (at
253                     // least) one flit in the rx buffer.  early release implements
254                     // that the tail flit of the previous packet has been received
255                     // at the rx buffer.
256
257                     // we implement early release by the code decorated with
258                     // CHANGE
259
260                     if (DEBUG5) {
261                         if (Kernel.Now >= 33850)
262                             System.out.println (Kernel.time () + toString ()
263                                                 + " trying to connect new packet with "
264                                                 + outvl.toString ());
265                     }
266
267                     if (outvl.receiver.no_restrictions (null, outvl)
268                             && outvl.tx_buffer.empty ()
269                             && outvl.new_connection_acceptable ()) {
        // CHANGE
270
271                         if (DEBUG5) {
272                             if (Kernel.Now >= 33850)
273                                 System.out.println ("  ..... succeeded");
274                         }
275
276                         EasterEgg SF_egg = new EasterEgg ();
277                         in_transmission++;
278                         outvl.previous = connector; // "connect"
279                         connector.next = outvl;
280                         connector.packet = p;
281                         p.connected = true;
282                         p.injection_time = Kernel.Now;
283                         Kernel.SourceRoute (p);
284                         if (DEBUG)
285                             System.out.println (toString () + " choosing "
286                                                 + outvl.toString () + " for "
287                                                 + p.toString ());
288                         p.traversed_route.addElement (outvl);
289                         SF_egg.c = connector;
290                         (new Event (this, SCHEDULED_FRAGMENT, SF_egg)).
291                         schedule (Kernel.Now + Kernel.Cycle (1));
292
293                         break;
294                     } else {
295                         if (DEBUG5) {
296                             if (Kernel.Now >= 33850)
```

```
297                             System.out.println ("  ..... failed");
298                         }
299                     }
300 
301                 }                       // for
302                 p = send_q.nextUnConnectedElement ();
303             }                                   // while
304         }
305 
306         // the transmitter state procedures, see the process model processor_tx
307 
308         public void scheduled_fragment_enter (EasterEgg egg) {
309             if (DEBUG)
310                 System.out.println (toString () + " scheduled_fragment_enter");
311             VLane outvl = egg.c.next;
312             EasterEgg cre_egg = new EasterEgg ();
313             cre_egg.vl = outvl;
314             cre_egg.flit = egg.c.packet.nextFlit ();
315             if (DEBUG9) {
316                 if (Kernel.now >= 466660)
317                     System.out.println (toString () + " packet " +
318                                         cre_egg.flit.toString ());
319             }
320             if (DEBUG) {
321                 if (cre_egg.flit == null)
322                     System.out.println (toString () + " gack, produced flit null"
323                                         + " for " + egg.c.packet.toString ()
324                                         + " gotten from out vlane " +
325                                         outvl.toString ());
326             }
327             if (Kernel.TRACE_PACKET) {
328 
329                 if (!cre_egg.flit.isKill ()
330                         && cre_egg.flit.packet.pid == Kernel.PACKET_TRACED
331                         && (cre_egg.flit.isHeader () || cre_egg.flit.isTail ()))
332                     System.out.println (Kernel.time () + toString () +
333                                         " scheduled_fragment " +
334                                         cre_egg.vl.toString () + " for " +
335                                         cre_egg.flit.toString ());
336             }
337             check_rx_enter (cre_egg);
338         }
339 
340         public void check_rx_enter (EasterEgg egg) {
341             if (DEBUG)
342                 System.out.println (toString () + " check_rx_enter to "
343                                     + egg.vl.toString () + " for "
344                                     + egg.flit.toString ());
345             VLane outvl = egg.vl;
346             if (outvl.rx_buffer.full () || !outvl.credits_available ()) {
347                 if (DEBUG)
348                     System.out.println (" (full)");
349                 EasterEgg wre_egg = new EasterEgg ();
350                 wre_egg.vl = egg.vl;
351                 wre_egg.flit = egg.flit;
352                 if (Kernel.DROP_PACKETS && false) {
353                     if (egg.vl.rx_buffer.top ().isHeader () || true) {
354                         System.out.println ("dropping header at RX processor");
355                         pack_to_drop[egg.vl.layer] =
356                             egg.vl.rx_buffer.top ().packet.pid;
357                         egg.vl.rx_buffer.pop ();
358                     } else if (egg.vl.rx_buffer.top ().packet.pid ==
359                                pack_to_drop[egg.vl.layer]) {
360                         egg.vl.rx_buffer.pop ();
361                     } else
362                         wait_rx_enter (wre_egg);
363                 } else
364                     wait_rx_enter (wre_egg);
365                 //wait_rx_enter(wre_egg);
366             } else {
367                 if (DEBUG)
368                     System.out.println (" (ready)");
369                 outvl.rx_buffer.reserve ();
370                 EasterEgg cte_egg = new EasterEgg ();
371                 cte_egg.vl = egg.vl;
```

```
372             cte_egg.flit = egg.flit;
373             check_tx_enter (cte_egg);
374         }
375     }
376
377     public void wait_rx_enter (EasterEgg egg) {
378         if (Kernel.TRACE_PACKET) {
379
380             if (!egg.flit.isKill () && egg.flit.packet.pid == Kernel.PACKET_TRACED
381                     && (egg.flit.isHeader () || egg.flit.isTail ()))
382                 System.out.println (Kernel.time () + toString () +
383                                     " wait_rx_enter on " + egg.vl.toString () +
384                                     " for " + egg.flit.toString ());
385         }
386         egg.vl.NU_wait_rx = true;
387         egg.vl.previous.flitstore = egg.flit;
388     }
389
390     public void wait_rx_exit (EasterEgg egg) {
391         VLane outvl = egg.vl;
392         if (outvl.NU_wait_rx) {
393             if (Kernel.TRACE_PACKET) {
394                 if (!egg.vl.previous.flitstore.isKill ()
395                         && egg.vl.previous.flitstore.packet.pid ==
396                         Kernel.PACKET_TRACED && (egg.vl.previous.flitstore.isHeader (
  )
397                                                  || egg.vl.previous.flitstore.
398                                                  isTail ()))
399                     System.out.println (Kernel.time () + toString () +
400                                         " wait_rx_exit on " + egg.vl.toString () +
401                                         " for " +
402                                         egg.vl.previous.flitstore.toString ());
403             }
404             outvl.NU_wait_rx = false;
405             EasterEgg cte_egg = new EasterEgg ();
406             cte_egg.vl = egg.vl;
407             cte_egg.flit = outvl.previous.flitstore;
408             outvl.previous.flitstore = null;
409             outvl.rx_buffer.reserve ();
410             check_tx_enter (cte_egg);
411         } else {
412             if (DEBUG)
413                 System.out.println (toString () + "on_rx_available "
414                                     + outvl.toString () + " disregarded");
415         }
416     }
417
418     public void check_tx_enter (EasterEgg egg) {
419         if (DEBUG)
420             System.out.println (toString () + " check_tx_enter to "
421                                 + egg.vl.toString () + " for "
422                                 + egg.vl.previous.packet.toString ());
423         VLane outvl = egg.vl;
424         if (outvl.tx_buffer.full ()) {
425             if (DEBUG)
426                 System.out.println (" (tx full)");
427             EasterEgg wte_egg = new EasterEgg ();
428             wte_egg.vl = egg.vl;
429             wte_egg.flit = egg.flit;
430             if (Kernel.LOGICAL_ERROR_CHECK) {
431                 if (egg.flit.isHeader ())
432                     System.out.println (Kernel.time () + toString () +
433                                         " logical error:" + " header " +
434                                         egg.flit.toString () +
435                                         " needs to wait for tx");
436             }
437             if (Kernel.DROP_PACKETS && false) {
438                 if (egg.flit.isHeader () || true) {
439                     System.out.println ("dropping header at TX-procesor");
440                     pack_to_drop[outvl.layer] = egg.flit.packet.pid;
441                     if (egg.flit.isEnd ()) {
442                         EasterEgg cce_egg = new EasterEgg ();
443                         cce_egg.vl = outvl;
444                         close_connection_enter (cce_egg);
445                     } else {
```

```
446                             self_signal_scheduled_fragment (egg.vl.previous);
447                         }
448                     } else if (egg.flit.packet.pid == pack_to_drop[egg.vl.layer]) {
449                         if (egg.flit.isEnd ()) {
450                             EasterEgg cce_egg = new EasterEgg ();
451                             cce_egg.vl = outvl;
452                             close_connection_enter (cce_egg);
453                         }
454                         System.out.println ("Dropping bodyflit at processor");
455                     } else
456                         wait_tx_enter (wte_egg);
457                 } else
458                     wait_tx_enter (wte_egg);
459 
460                 //wait_tx_enter(wte_egg);
461             } else {
462                 if (DEBUG)
463                     System.out.println (" (tx ready)");
464                 EasterEgg fse_egg = new EasterEgg ();
465                 fse_egg.c = egg.vl.previous;
466                 fse_egg.flit = egg.flit;
467                 if (Kernel.DROP_PACKETS && false) {
468                     if (egg.flit.packet.pid == pack_to_drop[egg.vl.layer]) {
469                         if (egg.flit.isEnd ()) {
470                             EasterEgg cce_egg = new EasterEgg ();
471                             cce_egg.vl = outvl;
472                             close_connection_enter (cce_egg);
473                         } else {
474                             self_signal_scheduled_fragment (egg.vl.previous);
475                         }
476                         System.out.println ("Dropping bodyflit at processor");
477                     } else
478                         flit_send_enter (fse_egg);
479                 } else
480                     flit_send_enter (fse_egg);
481             }
482         }
483 
484         public void wait_tx_enter (EasterEgg egg) {
485             VLane outvl = egg.vl;
486             if (Kernel.TRACE_PACKET) {
487                 if (!egg.flit.isKill ()
488                         && egg.flit.packet.pid == Kernel.PACKET_TRACED
489                         && (egg.flit.isHeader () || egg.flit.isTail ()))
490                     System.out.println (Kernel.time () + toString () +
491                                         " wait_tx_enter on " + egg.vl.toString () +
492                                         " for " + egg.flit.toString ());
493             }
494             outvl.NU_wait_tx = true;
495             outvl.previous.flitstore = egg.flit;
496         }
497 
498         public void wait_tx_exit (EasterEgg egg) {
499             VLane outvl = egg.vl;
500 
501             if (outvl.NU_wait_tx) {
502                 if (Kernel.TRACE_PACKET) {
503                     if (egg.vl.previous != null
504                             && !egg.vl.previous.flitstore.isKill ()
505                             && egg.vl.previous.flitstore.packet.pid ==
506                             Kernel.PACKET_TRACED && (egg.vl.previous.flitstore.isHeader (
    )
507                                                  || egg.vl.previous.flitstore.
508                                                  isTail ()))
509 
510                         System.out.println (Kernel.time () + toString () +
511                                             " wait_tx_exit on " + egg.vl.toString () +
512                                             " for " +
513                                             egg.vl.previous.flitstore.toString ());
514                 }
515                 outvl.NU_wait_tx = false;
516                 EasterEgg fse_egg = new EasterEgg ();
517                 fse_egg.c = egg.vl.previous;
518                 fse_egg.flit = outvl.previous.flitstore;
519                 outvl.previous.flitstore = null;
```

```
520                 flit_send_enter (fse_egg);
521             } else {
522                 if (DEBUG)
523                     System.out.println (toString () + "on_tx_available "
524                                         + outvl.toString () + " disregarded");
525             }
526         }
527 
528         public void flit_send_enter (EasterEgg egg) {
529             if (DEBUG)
530                 System.out.println (toString () + "flit_send_enter on "
531                                     + /*egg.vl.toString()+ */ " for " +
532                                     egg.flit.toString ());
533 
534             VLane outvl = egg.c.next;
535             outvl.tx_buffer.reserve ();
536             outvl.tx_buffer.insert (egg.flit);
537             (new Event (outvl, VLane.FLIT_IN_TRANSMITTER)).schedule (Kernel.Now);
538             if (egg.flit.isEnd ()) {
539                 EasterEgg cce_egg = new EasterEgg ();
540                 cce_egg.vl = outvl;
541                 close_connection_enter (cce_egg);
542             } else {
543                 self_signal_scheduled_fragment (egg.c);
544             }
545         }
546 
547         public void self_signal_scheduled_fragment (Connectable c) {
548             if (DEBUG)
549                 System.out.println (toString () + " self_signal_scheduled_fragment");
550             EasterEgg SF_egg = new EasterEgg ();
551             SF_egg.c = c;
552             (new Event (this, SCHEDULED_FRAGMENT, SF_egg)).schedule (Kernel.Now +
553                     Kernel.
554                     Cycle (1));
555         }
556 
557         public void close_connection_enter (EasterEgg egg) {
558             VLane outvl = egg.vl;
559             if (DEBUG)
560                 System.out.println (toString () + " close_connection_enter on "
561                                     + outvl.previous.toString () + " for "
562                                     + outvl.previous.packet.pid);
563             send_q.removeElement (outvl.previous.packet);
564             outvl.previous.packet = null;
565             outvl.previous.next = null;
566             outvl.previous = null;
567             in_transmission--;
568             //System.out.println("close_connection_enter");
569             (new Event (this, CONNECTABLE)).schedule (Kernel.Now);
570         }
571 
572         public void handle_ctrl_flit (EasterEgg egg) {
573             // the basic processors ignores control flits
574             // it will typically be overridden by fault tolerant switches
575 
576 
577         }
578 
579 
580         // the receiver state procedures, see the process model processor_rx
581 
582         public void flit_on_top_enter (EasterEgg egg) {
583             if (DEBUG)
584                 System.out.println (toString () + " flit_on_top_enter on "
585                                     + egg.vl.toString () + " for "
586                                     + egg.vl.rx_buffer.top ().toString ());
587 
588             if (egg.vl.link.ctrl_vlane == egg.vl) {
589                 // if the incoming flit is on a control link!
590                 if (DEBUG)
591                     System.out.println (toString () + " control lane");
592                 EasterEgg hcf_egg = new EasterEgg ();
593                 hcf_egg.vl = egg.vl;
594                 hcf_egg.flit = egg.vl.rx_buffer.pop ();
```

```
595                 handle_ctrl_flit (hcf_egg);
596                 return;
597             }
598 
599             if (egg.vl.NU_header_waiting)
600                 return;
601 
602             //    if (Kernel.TRACE_PACKET) {
603             //        Flit f = egg.vl.rx_buffer.top();
604             //        int pid=0;
605             //        if (egg.vl.packet != null)
606             //      pid = egg.vl.packet.pid;
607             //        else if (f != null)
608             //      System.out.println(toString() + "no packet in FOT");
609             //        else
610             //      pid = f.packet.pid;
611             //        if (pid == Kernel.PACKET_TRACED)
612             //      System.out.println(Kernel.time() + toString() + " FOT "
613             //                       + egg.vl.toString() + f.toString());
614             //    }
615 
616             if (egg.vl.next == null && egg.vl.rx_buffer.top ().isHeader ()) {
617                 EasterEgg be_egg = new EasterEgg ();
618                 be_egg.vl = egg.vl;
619                 egg.vl.NU_header_waiting = true;
620                 bound_enter (be_egg);
621             } else if (egg.vl.next != null) {
622                 EasterEgg ae_egg = new EasterEgg ();
623                 ae_egg.vl = egg.vl;
624                 assemble_enter (ae_egg);
625             } else {
626                 // probably a kill flit.  some garbage, which we should ignore!
627                 Flit f = (Flit) egg.vl.rx_buffer.pop ();
628 
629                 if (DEBUG33) {
630                     System.out.println (toString () + " garbage entered: " +
631                                         f.toString ());
632                 }
633 
634                 (new Event (egg.vl, VLane.PURGED_RX)).schedule (Kernel.Now);
635             }
636         }
637 
638         public void bound_enter (EasterEgg egg) {
639             if (DEBUG)
640                 System.out.println (toString () + " bound_enter");
641             bind_q.addElement (egg.vl);
642             (new Event (this, BINDABLE)).schedule (Kernel.Now, bindable_flag);
643         }
644         public void bound_exit (EasterEgg egg) {
645             if (DEBUG)
646                 System.out.println (toString () + " bound_exit");
647             EasterEgg ae_egg = new EasterEgg ();
648             ae_egg.vl = egg.vl;
649             egg.vl.NU_header_waiting = false;
650             assemble_enter (ae_egg);
651         }
652 
653         public void assemble_enter (EasterEgg egg) {
654             VLane invl = egg.vl;
655             Flit flit = (Flit) invl.rx_buffer.pop ();
656             if (DEBUG)
657                 System.out.println (toString () + " assemble_enter on "
658                                     + invl.toString () + " for " + flit.toString ());
659             if (DEBUG11)
660                 System.out.println (flit.packet.sl);
661             (new Event (invl, VLane.PURGED_RX)).schedule (Kernel.Now);
662 
663             if (!flit.isKill ())
664                 flit.packet.ticked_in ();
665 
666             if (flit.isEnd ()) {
667                 if (DEBUG)
668                     System.out.println (" (was end-flit)");
669                 EasterEgg re_egg = new EasterEgg ();
```

```
670                 re_egg.vl = egg.vl;
671                 re_egg.flit = flit;
672                 receive_enter (re_egg);
673 
674             } else if (invl.rx_buffer.inhabitated ()) {
675                 // any trailing header flits have been self signalled in
676                 // receive_enter
677                 self_signal_flit_on_top (invl);
678             }
679         }
680 
681         public void receive_enter (EasterEgg egg) {
682             if (DEBUG)
683                 System.out.println (toString () + " receive_enter on "
684                                     + egg.vl.toString () + " for "
685                                     + egg.flit.toString ());
686 
687             VLane invl = egg.vl;
688             invl.next = null;
689             invl.packet = null;
690 
691             if (invl.rx_buffer.inhabitated ()) {
692                 self_signal_flit_on_top (invl);
693             }
694 
695             // what about invl's packet?? it is used when receiving so that
696             // it is possible to remember which packet we are currently
697             // assembling. and only there, mesa thinks. it is set again by
698             // bindable, so it should not be moved here, just nulled out.
699             EasterEgg PR_egg = new EasterEgg ();
700             PR_egg.flit = egg.flit;
701             (new Event (this, PACKET_RECEIVED, PR_egg)).schedule (Kernel.Now +
702                     service_time);
703         }
704 
705         public void self_signal_flit_on_top (VLane invl) {
706             if (DEBUG)
707                 System.out.println (toString () + " self_signal_flit_on_top_enter");
708             EasterEgg FOT_egg = new EasterEgg ();
709             FOT_egg.vl = invl;
710             (new Event (this, FLIT_ON_TOP, FOT_egg)).schedule (Kernel.Now +
711                     Kernel.Cycle (1));
712         }
713 
714         // the binder state procedures, see the process model processor_binder
715 
716         public void bindable_enter (EasterEgg egg) {
717             while (bind_q.size () > 0 && bound_receive_buffers < receive_buffers) {
718                 VLane top = (VLane) (bind_q.firstElement ());
719                 bind_q.removeElementAt (0); // pop
720                 top.next = top;             // "bind" (or "connect", if you like)
721                 top.packet = top.rx_buffer.top ().packet;
722                 bound_receive_buffers++;
723                 EasterEgg B_egg = new EasterEgg ();
724                 B_egg.vl = top;
725                 (new Event (this, BOUND, B_egg)).schedule (Kernel.Now +
726                         Kernel.Cycle (1));
727             }
728         }
729 
730         // the consumer state procedures, see the process model processor_consumer
731 
732         public void packet_received_enter (EasterEgg egg) {
733             bound_receive_buffers--;
734             (new Event (this, BINDABLE)).schedule (Kernel.Now, bindable_flag);
735 
736 
737             if (!egg.flit.isKill () && egg.flit.packet.destination != this) {
738                 System.out.println (Kernel.time () + toString ()
739                                     + "Misrouted packet (SL " + egg.flit.packet.sl +
740                                     "): " + egg.flit.toString ());
741                 System.out.println ("    Src: " + egg.flit.packet.source + " " +
742                                     egg.flit.packet.source.hashCode ());
743                 System.out.println ("    Dst: " + egg.flit.packet.destination + " " +
744                                     egg.flit.packet.destination.hashCode ());
```

```
745                 System.out.println ("    Cur: " + this + " " + this.hashCode ());
746             }
747         }
748 
749         // event implementations, remember to update the dispatcher function
750         // if adding more events!!
751 
752         public static final int PACKET_PRODUCED = 10;
753         public void packet_produced (EasterEgg egg) {
754             packet_produced_enter (egg);
755         }
756 
757         public static final int CONNECTABLE = Node.CONNECTABLE;
758         public Flag connectable_flag = new Flag (false, Kernel.Edge);
759         public void connectable (EasterEgg egg) {
760             connectable_enter (egg);
761         }
762 
763         public static final int SCHEDULED_FRAGMENT = 30;
764         public void scheduled_fragment (EasterEgg egg) {
765             scheduled_fragment_enter (egg);
766         }
767 
768         public static final int RX_AVAILABLE = Node.RX_AVAILABLE;
769         public void rx_available (EasterEgg egg) {
770             wait_rx_exit (egg);
771         }
772 
773         public static final int TX_AVAILABLE = Node.TX_AVAILABLE;
774         public void tx_available (EasterEgg egg) {
775             wait_tx_exit (egg);
776         }
777 
778         public static final int FLIT_ON_TOP = Node.FLIT_ON_TOP;
779         public void flit_on_top (EasterEgg egg) {
780             flit_on_top_enter (egg);
781         }
782 
783         public static final int BINDABLE = 60;
784         public Flag bindable_flag = new Flag (false, Kernel.Edge);
785         public void bindable (EasterEgg egg) {
786             bindable_enter (egg);
787         }
788 
789         public static final int BOUND = 70;
790         public void bound (EasterEgg egg) {
791             bound_exit (egg);
792         }
793 
794         public static final int PACKET_RECEIVED = 80;
795         public void packet_received (EasterEgg egg) {
796             packet_received_enter (egg);
797         }
798 
799         //Regular event for refreshing egress statistics, defined in MSProcessor.java
800         public static final int MEASURE_INTERVAL = 90;
801         public void measure_interval (EasterEgg egg) {
802             measure_interval_enter (egg);
803         }
804 
805         public static final int FLOW_INTERVAL = 100;
806         public void executeFlowEnter (EasterEgg egg) {
807             executeFlow (egg);
808         }
809 
810         public void measure_interval_enter (EasterEgg egg) {}
811         public void executeFlow (EasterEgg egg) {}
812 
813         // common for all Processor events
814 
815         public void dispatcher (int dispatcher, EasterEgg egg) {
816 
817             switch (dispatcher) {
818             case PACKET_PRODUCED:
819                 packet_produced (egg);
```

```
820                 break;
821             case CONNECTABLE:
822                 connectable (egg);
823                 break;
824             case SCHEDULED_FRAGMENT:
825                 scheduled_fragment (egg);
826                 break;
827             case RX_AVAILABLE:
828                 rx_available (egg);
829                 break;
830             case TX_AVAILABLE:
831                 tx_available (egg);
832                 break;
833             case FLIT_ON_TOP:
834                 flit_on_top (egg);
835                 break;
836             case BINDABLE:
837                 bindable (egg);
838                 break;
839             case BOUND:
840                 bound (egg);
841                 break;
842             case PACKET_RECEIVED:
843                 packet_received (egg);
844                 break;
845                 //used for admis.ctrl.
846             case MEASURE_INTERVAL:
847                 measure_interval (egg);
848                 break;
849             case FLOW_INTERVAL:
850                 measure_interval (egg);
851                 break;
852             default:
853 
854             }
855         }                                   // end dispatcherEvent
856 
857         public void purgeEvents () {
858             while (scheduled_events.size () > 0) {
859                 Event e = (Event) (scheduled_events.elementAt (0));
860                 scheduled_events.removeElementAt (0);
861                 Kernel.globalHeap.removeEvent (e);
862                 e.dismiss ();
863             }
864         }
865 
866         public Vector scheduled_events = new Vector ();
867         public Vector scheduled_events () {
868             return scheduled_events;
869         }
870 
871         public boolean remote_admit (Packet p) {
872             return true;
873         }
874         public boolean probeDone (Packet p) {
875             return false;
876         }
877         public void probeReset () {}
878         public void stopProbe () {}
879         public static boolean stableDone () {
880             return true;
881         }
882         public void continueAdmitting (int load) {}
883     }                                       // end class Processor
884 
```

```
1    /*
2     * MSProcessor.java:  MSProcessor is a subclass of Processor.  It implements
3     * the Egress Measurements AC scheme by introducing flows, and an AC routine.
4     * The admission descission is performed by having the sending node call the AC
5     * routine at the reciever.  The reciever responds based on the data it has gathered.

6     */
7    package base;
8
9    import java.util.Random;
10   import java.util.Vector;
11   import java.lang.Math;
12
13   public class MSProcessor extends Processor {
14       boolean ADDEBUG = false;
15       boolean DEBUG = false;
16       static boolean NEWDEBUG = false, INTERESTED = true, MUCHOUTPUT = false;
17       private int flowMessages = 0;
18       private boolean wasnotOk = true, statsOk = false;
19       long resvtime, timeout = 10000;
20       private static boolean print_stats = true;
21       private int num_flows = 0;
22       private double peakRate = 2, delay = 500;
23       private double alpha = 1;
24       private int numFlowsInNet = 0;
25       private int measPeriodTau = Kernel.measPeriod;        //Resolution of arrival measu
     rements
26       private static final int timeSlotsT = Kernel.timeSlots;       //Number of measureme
     nt periods
27       private static final int measHistorym = 5;  //How long to retain history
28       private static final int maxBacklog = timeSlotsT;
29       private static final int BHLength = maxBacklog;
30       private int[][] arrivals = new int[16][timeSlotsT + 1];
31       private float[][][] R = new float[16][measHistorym][timeSlotsT];      //arrival env
     elope
32           //in U(i), The time interval for i packets in backlog, index 0 equals 1 packet in
      backlog osv.
33       private int totalSL =
34           Kernel.CV[Kernel.highpri_service_levels] +
35           Kernel.CV[Kernel.lowpri_service_levels];
36       private long[][][] U = new long[totalSL][timeSlotsT][maxBacklog];     //service env
     elope
37       private long[][][] Uhistory = new long[totalSL][measHistorym][maxBacklog];
38       private long[][] BacklogHistory = new long[BHLength][2];
39       private int index;
40       private int flowRetry = 100, retryCounter = 1;
41       private boolean onlyFlows = true;
42       private Random rand = new Random ();
43       private long measInt, measStart = 0;
44       private double sendPacketCounter = -12;
45       private double sendPacketCounterInit = 0;
46       private boolean NEW_FLOW = false;
47       private long lnf = 0;
48       private boolean localAdded = false, localUse1 = true;
49       public static double offeredFlows = 0;
50       //static things for ensuring that all processors add a new flow
51       public static int newFlowCounter = 0;
52       private static boolean stabilise = false;
53       public static boolean adding1 = false, adding2 = false, use1 =
54                                           true, NO_MORE_FLOWS = false;
55       public static int increases = 0, numNoAdditions = 0;
56       private static long lastNewFlow = 0, timeBetweenFlows = 0;
57       private static int lastSent = 1;
58       private int localLastSent = 0;
59       private boolean toosmall = false;
60       private boolean firsttime = true;
61       private static int numOk = 160;
62       private int flowNum = 0;
63       private long outSynk = 0;
64
65       private double flowBW;
66       int[] flowMapping;
67
68
69       /*Method called when the processor is done adding a new flow, handels finishing o
```

```
     f flow-adding period and so on */
70       private static boolean doneNewFlow (boolean admitted) {
71           if (admitted) {
72               if (NEWDEBUG)
73                   System.out.println ("Admitted...");
74               stabilise = true;
75               numNoAdditions = 0;
76           }
77           newFlowCounter++;
78           if (NEWDEBUG)
79               System.out.
80               println ("Numbers of processors who have added their flow: " +
81                         newFlowCounter);
82           if (newFlowCounter ==
83                   Kernel.CV[Kernel.num_switches] *
84                   Kernel.CV[Kernel.Processor_per_switch]) {
85               if (!stabilise)
86                   numNoAdditions++;
87               if (NEWDEBUG)
88                   System.out.println ("Time of stable period: " + numNoAdditions);
89               newFlowCounter = 0;
90               if (NEWDEBUG)
91                   System.out.println ("Time to stabilise if necessary");
92               increases++;
93 
94               if ((numNoAdditions == 10) || increases >= offeredFlows - 1) {
95                   if (NEWDEBUG)
96                       System.out.println("%% 10 times without admittance, time to stabi
     lize and get results %%");
97                   Kernel.unstable = true;
98                   Kernel.admitting = false;
99                   //stableDone();
100                  NO_MORE_FLOWS = true;
101                  stabilise = false;
102                  lastSent++;
103                  if (NEWDEBUG || INTERESTED)
104                      System.out.println ("%Number of times new flows have been added:
     " + increases + "\n%Stabilising network");
105              } else {
106                  Kernel.stabiliseEnd = Kernel.Now;
107                  lastSent++;
108                  if (NEWDEBUG || INTERESTED)
109                      System.out.
110                      println ("%Number of times new flows have been added: " + increas
     es + "\n%No need to restabilise network.");
111 
112              }
113              stabilise = false;
114          }
115          return true;
116      }
117 
118      /*Called when network is stabilised */
119      public static boolean stableDone () {
120          Kernel.stabiliseEnd = Kernel.Now;
121          lastSent++;
122          if (NEWDEBUG)
123              System.out.println ("%Done stabilising...");
124          return true;
125      }
126 
127      private int NumPackInFlow (int min, int max) {
128          return min + rand.nextInt (max);
129 
130      }
131 
132      public MSProcessor (int id) {
133          this (id, Kernel.CV[Kernel.Processor_receive_buffers],
134                Kernel.CV[Kernel.Processor_send_q_size],
135                Kernel.CV[Kernel.Processor_service_time],
136                Kernel.CV[Kernel.Processor_drop_connects],
137                Kernel.CV[Kernel.Node_drop_links]);
138      }
139 
140      /*Constructor initsialising variables for flow handling, packet interrarival time
```

```
     .... */
141      public MSProcessor (int id, int receive_buffers, int send_q_size,
142                          int service_time, int drop_connects, int drop_links) {
143          super (id, receive_buffers, send_q_size, service_time, drop_connects,
144                 drop_links);
145          Kernel.admitting = true;
146          for (int i = 0; i < BHLength; i++)
147              BacklogHistory[i][0] = -1;
148          if (maxBacklog > BHLength)
149              System.out.println ("BHLength er for liten\n");
150          double f = 1000000, h = 0;
151          for (int i = 0; i < totalSL; i++) {
152              f = Math.min (f, Kernel.BWflow[i]);
153              h = Math.max (h, Kernel.slLoad[i]);
154          }
155
156          //calculates number of flows supported at this load-level
157          h = Kernel.minMean * (float) Math.pow (h / 100.0, -1);
158          offeredFlows = 0;
159
160          //doubles flowBW of every other flow
161          if (Kernel.DOUBLE_BW && Math.IEEEremainder (adctrlID, 2) == 0)
162              flowBW = 2 * Kernel.BWflow[localSL];
163          else
164              flowBW = Kernel.BWflow[localSL];
165          Packet.bandwidths[localSL][adctrlID] = flowBW;
166          double temp =
167              (double) 1 / Kernel.CV[Kernel.mean_packet_interarrival_time];
168          localOfferedFlows =
169              (temp * (Kernel.slPercentage[localSL] / 100)) / flowBW;
170          if (localOfferedFlows < 2)
171              localOfferedFlows = 2;
172          Packet.numFlows[localSL] = (int) localOfferedFlows;
173          offeredFlows = Math.max (offeredFlows, localOfferedFlows);
174          maxFlows = localOfferedFlows;
175          timeBetweenFlows = measHistorym * timeSlotsT * measPeriodTau;
176          if (NEWDEBUG)
177              System.out.println ("%Time between flows: " + timeBetweenFlows +
178                                  "\n%Number of flows to be offered: " +
179                                  offeredFlows);
180          h =
181              Kernel.CV[Kernel.mean_packet_interarrival_time] *
182              (float) Math.pow (Kernel.slLoad[localSL] / 100.0, -1);
183          flowNum = (int) localOfferedFlows;
184          Packet.numFlows[localSL] =
185              Math.max (Packet.numFlows[localSL], (int) localOfferedFlows);
186          flowMapping = new int[(int) maxFlows];
187          if (Kernel.admitall) {
188              timeBetweenFlows = 0;
189          }
190          //self-similar
191          flows = new Flow[(int) localOfferedFlows + 3];
192
193          //clocks out of synk with 10% of min latency.
194          if (Kernel.MS_OUT_OF_SYNK)
195              outSynk = (long) (0.1 * (rand.nextInt (2 * 65) - 65));
196
197      }
198
199      /* Called when it is time to produce a new packet */
200      public void packet_produced_enter (EasterEgg egg) {
201          Packet p = null;
202          int sl = 0;
203          int hops = 0;
204          sl = localSL;
205          if (numOk == 0) {
206              Packet.statsOk = true;
207          }
208
209
210          if (DEBUG)
211              System.out.println (toString () + " packet_produced_enter");
212          //send dummy packets when waiting for reservation confirm, to create correct
     statistics.
213          Packet.dummy_send ();
```

```
214
215
216            long e = Kernel.Now - Kernel.stabiliseEnd;
217            if (NEWDEBUG)
218                if (!Kernel.unstable && !localAdded)
219                    System.out.println ("tid siden sist nye flyt: " + e);
220            /*determines if it is time to add a new flow */
221            if (!NO_MORE_FLOWS && !Kernel.unstable && (lastSent != localLastSent)
222                    && (Kernel.Now - Kernel.stabiliseEnd >=
223                        Kernel.Cycle (timeBetweenFlows))) {
224                localLastSent = lastSent;
225                if (NEWDEBUG)
226                    System.out.println ("Time to add new flow: " + id);
227                double f =
228                    1 / (Kernel.BWflow[localSL] * Kernel.slLoad[localSL] *
229                         (num_flows + 1));
230                if (num_flows + 1 <= (int) localOfferedFlows) {
231                    NEW_FLOW = true;
232                    if (Kernel.SELF_SIMILAR) {
233                        //make a pessimistic assumption about peakrate
234                        if (Kernel.SELF_TIGHT) {
235                            peakRate =
236                                ((double) 1 /
237                                 Kernel.CV[Kernel.mean_packet_interarrival_time]);
238                            peakRate *= (Kernel.Traffic.payload_size (id) + 2);
239                        } else {
240                            peakRate = (double) (1 / (flowBW * 2));
241                            peakRate -=
242                                2 * Kernel.CV[Kernel.std_dev_packet_interarrival_time];
243                            peakRate = 1 / peakRate;
244                            peakRate *= (Kernel.Traffic.payload_size (id) + 2);
245                        }
246                    } else {
247                        peakRate = flowBW;
248                        peakRate *= (Kernel.Traffic.payload_size (id) + 2);
249                        peakRate *= (localOfferedFlows * Kernel.slLoad[localSL]);
250                        peakRate /= (localOfferedFlows * Kernel.slLoad[localSL] -
251                                     2 *
252                                     Kernel.CV[Kernel.
253                                               std_dev_packet_interarrival_time]);
254                    }
255                } else {
256                    //handle special case of flowbw being larger than packet sending rate
     ,
257                    //reduces flowbw to sending rate.
258                    if (firsttime) {
259                        peakRate =
260                            (Kernel.Traffic.payload_size (id) +
261                             2) / (Kernel.slLoad[localSL]);
262                        NEW_FLOW = true;
263                        toosmall = true;
264                    } else {
265                        if (NEWDEBUG)
266                            System.out.
267                            println
268                            ("Not room for another flow produced between flows = " +
269                             f);
270                        Packet.rejFlowsPreOk[localSL]++;
271                        this.doneNewFlow (false);
272                    }
273                }
274            }
275            firsttime = false;
276
277            if (NEW_FLOW)              //It is time to add new flow..
278            {
279                NEW_FLOW = false;
280                //if new flow, do not send new packet, just perform reservations
281                flowMessages =
282                    NumPackInFlow (Kernel.min_flow_length,
283                                   Kernel.max_flow_length - Kernel.min_flow_length);
284                resvtime = Kernel.Now;
285                if (DEBUG)
286                    System.out.println ("Sending new reservation\n");
287
```

```
288             p = createPacket (this,
289                               pid + id,
290                               Kernel.kernel.packetDestination (this), 20, sl);
291
292             hops = Kernel.kernel.numHops (this, p.destination, p.sl);
293             delay = Kernel.delayBound[p.sl] + ((hops - 1) * 25);
294             p.payloadpointer = new Payload (Kernel.Now, peakRate, delay);
295             p.payloadpointer.resvinit = true;
296             iba_last_dest = p.destination;
297             iba_last_sl = p.sl;
298
299
300             // Admission is done using procedure-calls between processors and switche
    s, no packets are involved, takes too much time.
301             //send_pack(p);
302             if (p.destination.remote_admit (p)) {
303                 num_flows++;
304                 //self-similar
305                 flows[num_flows] = new Flow (this, num_flows);
306                 double ratio = maxFlows / num_flows;
307                 if (ratio < 1)
308                     ratio = 1;
309                 double findex = 0;
310                 for (int fn = 0; fn < (int) maxFlows; fn++)
311                     flowMapping[fn] = 0;
312                 for (int fn = 0; fn < (int) num_flows; fn++) {
313                     findex = fn * ratio;
314                     if (findex >= (int) maxFlows)
315                         findex = maxFlows - 1;
316                     if (findex < maxFlows && Kernel.EVEN_FLOW)
317                         flowMapping[(int) (findex)] = fn + 1;
318                     else
319                         flowMapping[fn] = fn + 1;
320                 }
321                 sendPacketCounterInit = 1 / (Kernel.BWflow[localSL] * Kernel.slLoad[l
    ocalSL] * num_flows);
322                 sendPacketCounter = sendPacketCounterInit;
323                 if (toosmall)
324                     sendPacketCounterInit = sendPacketCounter = 1;
325                 this.doneNewFlow (true);
326             } else
327                 this.doneNewFlow (false);
328         }
329         flowNum--;
330         //Sends a packet belonging to the correct flow.
331         if (flowMapping[flowNum] != 0) {
332             //self-similar
333             if ((flows[flowMapping[flowNum]].update_selfsim_flow (this))
334                     || !Kernel.SELF_SIMILAR) {
335                 if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null) {
336                     // first msg send from this processor or iba_msg not enabled
337                     p = createPacket (this,
338                                       pid + id,
339                                       Kernel.kernel.packetDestination (this),
340                                       Kernel.Traffic.payload_size (id), sl);
341                     p.payloadpointer = new Payload (Kernel.Now);
342                     p.flowNum = flowMapping[flowNum];
343                     iba_last_dest = p.destination;
344                     iba_last_sl = p.sl;
345                     send_pack (p);
346                 } else {
347                     if ((iba_message_packetcount > iba_msg_size)
348                             || !Kernel.IBA_ENABLE_MESSAGE) {
349                         // reset counter
350                         iba_message_packetcount = 0;
351
352                         p = createPacket (this,
353                                           pid + id,
354                                           iba_last_dest,
355                                           Kernel.Traffic.payload_size (id),
356                                           iba_last_sl);
357                         p.payloadpointer = new Payload (Kernel.Now);
358                         p.flowNum = flowMapping[flowNum];
359                         if (DEBUG)
360                             System.out.println ("Decreasing flowMessages\n");
```

```
361                             send_pack (p);
362                         } else {
363                             // continuing and existing message
364                             if (DEBUG)
365                                 System.out.println ("Sending continued message\n");
366                             p = createPacket (this,
367                                               pid + id,
368                                               iba_last_dest,
369                                               Kernel.Traffic.payload_size (id),
370                                               iba_last_sl);
371                             p.payloadpointer = new Payload (Kernel.Now);
372                             p.flowNum = flowMapping[flowNum];
373                             send_pack (p);
374                         }
375                     }
376                 }
377             }
378             if (flowNum <= 0) {
379                 flowNum = (int) maxFlows;
380             }
381 
382 
383             if (!Kernel.stop_packet_generation) {
384                 //Only send 1 packet in FLOW_RESV state, not whole message
385                 // comment out first if to continue sending packets when waiting for rese
    rvation confirm.
386                 //if(flowState!=FLOW_RESV){
387                 if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null
388                         || iba_message_packetcount > iba_msg_size) {
389 
390                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now
391                             +
392                             Kernel.Traffic.
393                             packet_interarrival_time
394                             ((int) Kernel.
395                              slLoad[localSL],
396                              Kernel.CV[Kernel.
397                                       std_dev_packet_interarrival_time],
398                              this));
399 
400                 } else {
401                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now + 10);
402                 }
403             }
404 
405             // keep track of number of packets send in this msg
406             if (Kernel.IBA_ENABLE_MESSAGE) {
407                 if (DEBUG)
408                     System.out.println ("Increasing message_packetcount\n");
409                 iba_message_packetcount++;
410             }
411         }
412         private static int teller = -1;
413 
414         //Determines admission using egress admission control
415         private boolean admit_flow (Packet p) {
416             boolean ok = true;
417             double U;
418             boolean allzero1 = true, allzero2 = true;
419             boolean highpri;
420             int j = 0, k = 0;
421             Packet packettemp = createPacket (this,
422                                               pid + id,
423                                               p.source,
424                                               20, p.sl);
425             packettemp.payloadpointer = new Payload (Kernel.Now);
426             double[] mR = meanR (R, p.sl);
427             double[] s2 = sigma2 (mR, R, p.sl);
428             double[] mU = meanU (Uhistory, p.sl, p.payloadpointer.delay);
429             double[] p2 = psi2 (mU, Uhistory, p.sl, p.payloadpointer.delay);
430             int antsl;
431             teller++;
432 
433             if (p.sl < Kernel.CV[Kernel.highpri_service_levels]) {
434                 antsl = Kernel.CV[Kernel.highpri_service_levels];
```

```
435                 highpri = true;
436             } else {
437                 antsl = Kernel.CV[Kernel.lowpri_service_levels];
438                 highpri = false;
439             }
440             double[][] mRall = new double[totalSL][timeSlotsT];
441 
442             for (int u = 0; u < totalSL; u++) {
443                 mRall[u] = meanR (R, u);
444             }
445 
446             //print debug
447             if (Math.IEEEremainder (teller, 160) == 0 && MUCHOUTPUT) {
448                 long in =
449                     Packet.numFlowHist * (Kernel.Now) /
450                     (Kernel.Cycle (Kernel.CV[Kernel.simulation_cycles]) +
451                      Kernel.stableTime);
452                 System.out.println ("Servicelevel = " + p.sl);
453 
454                 System.out.println ("mRplot = [");
455                 for (int g = 0; g < timeSlotsT; g++) {
456                     System.out.print (g * measPeriodTau * mR[g] + ",");
457                 }
458                 System.out.println ("]");
459                 System.out.println ("");
460 
461                 System.out.println ("mUplot = [");
462                 for (int g = 0; g < maxBacklog; g++) {
463                     System.out.print (mU[g] + ",");
464                 }
465                 System.out.println ("]");
466                 System.out.println ("");
467 
468                 System.out.println ("peakrate = " + p.payloadpointer.peak);
469 
470 
471             }
472 
473             if (ADDEBUG)
474                 System.out.println ("admitting...");
475 
476             //the actual AC calculation
477             for (int i = 0; i < timeSlotsT; i++) {
478                 if (Kernel.delaytoo) {
479                     double tot =
480                         ((i) * measPeriodTau) * mR[i] +
481                         ((i) * measPeriodTau) * p.payloadpointer.peak - mU[i];
482                     if (i * measPeriodTau * mR[i] +
483                             i * measPeriodTau * p.payloadpointer.peak - mU[i] +
484                             alpha *
485                             Math.
486                             sqrt ((double)
487                                   ((i) * (i) * measPeriodTau * measPeriodTau * s2[i]) +
488                                   (p2[i])) >= 0)
489                         ok = false;
490                     if (Math.IEEEremainder (teller, 160) == 0 && MUCHOUTPUT)
491                         System.out.print (tot + ",");
492                 }
493                 j = 0;
494                 U = 0;
495                 //inequality 2:
496 
497                 double abwSL = 0;
498                 int totw = 0;
499                 int LHP = Kernel.CV[Kernel.iba_limit_of_highpri];
500                 double tbw = 1;
501                 double abw;
502                 if (LHP >= 0 && LHP <= 1)
503                     abw = 0.5 * tbw;
504                 else {
505                     abw = tbw / (LHP + 1);
506                 }
507                 if (p.sl < Kernel.CV[Kernel.highpri_service_levels]) {
508                     abw = tbw - abw;
509                     if (abw == 0)
```

```
510                     abw += 0.00000001;
511                 for (int v = 0; v < Kernel.CV[Kernel.highpri_service_levels]; v++) {
512                     totw += Kernel.vlWeight[v];
513                 }
514                 abwSL = abw * Kernel.vlWeight[p.sl] / totw;
515             } else {
516                 if (abw == 0)
517                     abw += 0.00000001;
518                 for (int v = Kernel.CV[Kernel.highpri_service_levels];
519                         v <
520                         Kernel.CV[Kernel.lowpri_service_levels] +
521                         Kernel.CV[Kernel.highpri_service_levels]; v++) {
522                     totw += Kernel.vlWeight[v];
523                 }
524                 abwSL = abw * Kernel.vlWeight[p.sl] / totw;
525             }
526
527
528             if (highpri) {
529                 for (int u = 0; u < antsl; u++) {
530                     if (u != p.sl)
531                         abw -= mRall[u][i];
532                 }
533             } else {
534                 for (int u = Kernel.CV[Kernel.highpri_service_levels];
535                         u < Kernel.CV[Kernel.highpri_service_levels] + antsl; u++) {
536                     if (u != p.sl)
537                         abw -= mRall[u][i];
538                 }
539             }
540
541             //usually use nobandwidth, inequality 2 is therefore not used.
542             if (Kernel.rembandwith) {
543                 if (mR[i] + p.payloadpointer.peak > abw)
544                     ok = false;
545             } else if (Kernel.nobandwidth) {}
546             else {
547                 if (mR[i] + p.payloadpointer.peak > abwSL)
548                     ok = false;
549             }
550         }
551
552
553         for (int i = 0; i < timeSlotsT; i++) {
554             if (mR[i] != 0)
555                 allzero1 = false;
556         }
557
558         for (int i = 0; i < maxBacklog; i++) {
559             if (mU[i] < 10000000)
560                 allzero2 = false;
561         }
562
563         //always admitt if not enough data.
564         if ((allzero1) && !statsOk) {
565             if (Math.IEEEremainder (teller, 160) == 0 && MUCHOUTPUT)
566                 System.out.println ("Chicken");
567             ok = true;
568         } else
569             statsOk = true;
570
571         if (!ok) {
572             packettemp.payloadpointer.resvnotok = true;
573             if (ADDEBUG)
574                 System.out.println ("Rejecting reservation");
575         } else {
576             packettemp.payloadpointer.resvok = true;
577             if (ADDEBUG)
578                 System.out.println ("Accepting reservation");
579         }
580
581         //debug
582         if (DEBUG) {
583             System.out.println ("mR[i]\n");
584             for (int i = 0; i < timeSlotsT; i++) {
```

```
585                     System.out.println (mR[i] + "\n");
586                 }
587                 System.out.println ("mU[i]\n");
588                 for (int i = 0; i < maxBacklog; i++) {
589                     System.out.println (mU[i] + "\n");
590                 }
591             }
592 
593             //debug
594             if (Math.IEEEremainder (teller, 160) == 0 && MUCHOUTPUT) {
595                 if (ok)
596                     System.out.println ("%flow was accepted");
597                 else
598                     System.out.println ("%flow was not accepted");
599                 print_stats = false;
600             }
601 
602             if (Kernel.admitall)
603                 ok = true;
604             return ok;
605 
606         }
607 
608         //gathers statistics from revceived data.
609         public void packet_received_enter (EasterEgg egg) {
610             bound_receive_buffers--;
611             (new Event (this, BINDABLE)).schedule (Kernel.Now, bindable_flag);
612 
613 
614 
615             if (!egg.flit.isKill () && egg.flit.packet.destination != this) {
616                 Packet.misRouted++;
617                 System.out.println (Kernel.time () + toString ()
618                                     + "Misrouted packet (SL " + egg.flit.packet.sl +
619                                     "): " + egg.flit.toString ());
620                 System.out.println ("   Src: " + egg.flit.packet.source + " " +
621                                     egg.flit.packet.source.hashCode ());
622                 System.out.println ("   Dst: " + egg.flit.packet.destination + " " +
623                                     egg.flit.packet.destination.hashCode ());
624                 System.out.println ("   Cur: " + this + " " + this.hashCode ());
625             } else {
626                 //do not count resv messages
627                 //gather statistics
628                 //aggregate maximal rate envelope
629                 if (measStart == 0) {
630                     measStart = Kernel.Now;
631                     (new Event (this, MEASURE_INTERVAL)).
632                     schedule (Kernel.Now + Kernel.Cycle(measPeriodTau) * timeSlotsT);
633                 }
634                 index = (int) ((Kernel.Now - measStart) / Kernel.Cycle (measPeriodTau));
635                 if (index > timeSlotsT)
636                     System.out.
637                     println ("Index in Packet_recieved_enter is too large: " + index);
638                 else
639                     arrivals[egg.flit.packet.sl][index] += egg.flit.packet.size;
640 
641                 //Service envelope statistics
642                 for (int i = 0; i < BHLength - 1; i++) {
643                     BacklogHistory[i][0] = BacklogHistory[i + 1][0];
644                     BacklogHistory[i][1] = BacklogHistory[i + 1][1];
645                 }
646                 BacklogHistory[BHLength - 1][0] =
647                     (egg.flit.packet.payloadpointer.timestamp / 10) + outSynk;
648                 BacklogHistory[BHLength - 1][1] = Kernel.Now / 10;
649                 int i = 0;
650                 while ((i < maxBacklog) && (BacklogHistory[i][0] == -1))
651                     i++;
652                 int start = i;
653 
654                 if (i == 0) {
655                     while (i < BHLength - 1
656                             && (BacklogHistory[i][1] > BacklogHistory[i + 1][0]))
657                         i++;
658                     for (int p = 0; p <= i; p++) {
659                         int j = 0;
```

```
660                     if (p < maxBacklog)
661                         U[egg.flit.packet.sl][j][p] =
662                             Math.max (U[egg.flit.packet.sl][j][p],
663                                     (BacklogHistory[p][1] - BacklogHistory[0][0]));
664                 }
665             }
666         }
667     }
668
669
670
671
672
673     private void send_pack (Packet pp) {
674         pid += Kernel.max_switches;
675         if (send_q.full ()) {
676             flows[pp.flowNum].sentPackets++;
677             Packet.packets_rejected[pp.sl]++;
678             Packet.incRejected (pp.hops, pp.sl);
679             if (pp.payloadpointer.resvok) {
680                 if (Packet.statsOk)
681                     Packet.admitFlows[pp.sl]--;
682                 else
683                     Packet.admitFlowsPreOk[pp.sl]--;
684             } else if (pp.payloadpointer.resvnotok) {
685                 if (Packet.statsOk)
686                     Packet.rejFlows[pp.sl]--;
687                 else
688                     Packet.rejFlowsPreOk[pp.sl]--;
689             }
690         }
691         send_q.insert (pp);
692         if (in_transmission < drop_connects.length) {
693             send_enter (null);
694         }
695     }
696
697     /*All functions below are helperfunctions used in calculating
698      the variables used in the AC descission
699      */
700
701
702     //starts a new measurement interval, moves current data to history.
703     public void measure_interval_enter (EasterEgg egg) {
704         measStart = 0;
705         float temp = 0, temp2 = 0;
706         //arrival
707         for (int sltemp = 0; sltemp < totalSL; sltemp++) {
708             for (int i = 0; i < measHistorym - 1; i++) {
709                 for (int j = 0; j < timeSlotsT; j++) {
710                     R[sltemp][i][j] = R[sltemp][i + 1][j];
711                 }
712             }
713         }
714         for (int sltemp = 0; sltemp < totalSL; sltemp++) {
715             for (int i = 0; i < timeSlotsT - 1; i++)
716                 R[sltemp][measHistorym - 1][i] = 0;
717         }
718
719         for (int sltemp = 0; sltemp < totalSL; sltemp++) {
720             R[sltemp][measHistorym - 1][0] = 0;
721             for (int k = 0; k < timeSlotsT - 1; k++) {
722                 temp2 = 0;
723                 for (int s = 0; s < timeSlotsT - k; s++) {
724                     temp = 0;
725                     for (int j = 0; j <= k; j++) {
726                         temp += arrivals[sltemp][s + j];
727                     }
728                     temp2 = Math.max (temp, temp2);
729                 }
730                 R[sltemp][measHistorym - 1][k + 1] =
731                     temp2 / ((k + 1) * measPeriodTau);
732
733             }
734         }
```

```
735
736
737
738
739         //service
740         //retain history
741         for (int sltemp = 0; sltemp < totalSL; sltemp++) {
742             for (int i = 0; i < measHistorym - 1; i++) {
743                 for (int j = 0; j < maxBacklog; j++) {
744                     Uhistory[sltemp][i][j] = Uhistory[sltemp][i + 1][j];
745                 }
746             }
747         }
748
749         for (int sltemp = 0; sltemp < totalSL; sltemp++) {
750             Uhistory[sltemp][measHistorym - 1][0] = U[sltemp][0][0];
751             for (int j = 1; j < maxBacklog; j++) {
752                 Uhistory[sltemp][measHistorym - 1][j] =
753                     Math.max (U[sltemp][0][j],
754                               Uhistory[sltemp][measHistorym - 1][j - 1]);
755             }
756         }
757
758         for (int sltemp = 0; sltemp < totalSL; sltemp++) {
759             for (int j = 0; j < maxBacklog; j++) {
760                 U[sltemp][0][j] = 0;
761             }
762         }
763
764         for (int sltemp = 0; sltemp < totalSL; sltemp++) {
765             for (int j = 0; j < timeSlotsT; j++) {
766                 arrivals[sltemp][j] = 0;
767             }
768         }
769
770         if (!Packet.statsOk) {
771             double[] mR;
772             double[] mU;
773             boolean allzero1 = true, allzero2 = false;
774             for (int sl = 0; sl < totalSL; sl++) {
775                 mR = meanR (R, sl);
776                 mU = meanU (Uhistory, sl, 0);
777                 for (int i = 0; i < timeSlotsT; i++) {
778                     if (mR[i] != 0)
779                         allzero1 = false;
780                 }
781                 for (int i = 0; i < maxBacklog; i++) {
782                     if (mU[i] > 10000000)
783                         allzero2 = true;
784                 }
785                 //always admitt if not enough data.
786                 if (allzero1 || allzero2) {
787                     if (ADDEBUG)
788                         System.out.println ("Chicken");
789                 } else {
790                     numOk--;
791                 }
792
793             }
794         }
795
796     }
797
798     private double[] meanR (float[][][]Rtemp, int servicel) {
799         double[] temp = new double[timeSlotsT];
800         int m = measHistorym;
801         boolean z = true;
802         for (int j = 0; j < measHistorym; j++) {
803             z = true;
804             for (int i = 0; i < timeSlotsT; i++) {
805                 if (Rtemp[servicel][j][i] != 0)
806                     z = false;
807             }
808             if (z)
809                 m--;
```

```
810            }
811        if (m < 1)
812            m = 1;
813        for (int i = 0; i < timeSlotsT; i++) {
814            for (int j = 0; j < measHistorym; j++) {
815                temp[i] += Rtemp[servicel][j][i];
816            }
817            temp[i] /= m;
818        }
819        return temp;
820    }
821
822
823    private double[] sigma2 (double[]meanRtemp, float[][][]Rtemp, int servicel) {
824        double[] res = new double[timeSlotsT];
825        int m = measHistorym;
826        boolean z = true;
827        for (int j = 0; j < measHistorym; j++) {
828            z = true;
829            for (int i = 0; i < timeSlotsT; i++) {
830                if (Rtemp[servicel][j][i] != 0)
831                    z = false;
832            }
833            if (z)
834                m--;
835        }
836
837        if (m < 2)
838            m = 2;
839        for (int i = 0; i < timeSlotsT; i++) {
840            for (int j = 0; j < measHistorym; j++) {
841                res[i] +=
842                    ((Rtemp[servicel][j][i] -
843                      meanRtemp[i]) * (Rtemp[servicel][j][i] - meanRtemp[i]));
844            }
845            res[i] /= (m - 1);
846        }
847        return res;
848    }
849
850    private double[] inverseU (long[]Utemp, double delay) {
851        double[] UU = new double[maxBacklog];
852        int j;
853        double U;
854        for (int i = 0; i < timeSlotsT; i++) {
855            j = 0;
856            U = 0;
857            //inequality 1:
858
859            while (j < maxBacklog && (Utemp[j] < ((i * measPeriodTau) + delay)))
860                j++;
861            if (j > 0 && j < maxBacklog) {
862                if (Utemp[j] - Utemp[j - 1] != 0)
863                    U =
864                        j +
865                        (((i * measPeriodTau) + delay - Utemp[j - 1]) / (Utemp[j] -
866                                Utemp[j -
867                                      1]));
868                else
869                    U = j;
870                //U=j;
871            } else if (j >= maxBacklog) {
872                U = 10000000;
873                j = maxBacklog - 1;
874            } else {
875                U = 0;
876            }
877            U *= Kernel.CV[Kernel.payload_size] + 2;
878            UU[i] = U;
879        }
880        return UU;
881    }
882
883    private double[] meanU (long[][][]Uhtemp, int servicel, double delay) {
884        int j;
```

```
885            double U;
886            double[] temp = new double[maxBacklog];
887            double[][] temp2 = new double[maxBacklog][timeSlotsT];
888            int m = measHistorym;
889            boolean z = true;
890            for (int y = 0; y < measHistorym; y++) {
891                z = true;
892                for (int i = 0; i < maxBacklog; i++) {
893                    if (Uhtemp[servicel][y][i] != 0)
894                        z = false;
895                }
896                if (z)
897                    m--;
898            }
899            if (m < 1)
900                m = 1;
901            m = measHistorym;
902            for (int g = 0; g < measHistorym; g++) {
903                temp2[g] = inverseU (Uhtemp[servicel][g], delay);
904            }
905
906            for (int i = 0; i < maxBacklog; i++) {
907                for (int y = 0; y < measHistorym; y++) {
908                    temp[i] += temp2[y][i];
909                }
910                temp[i] = temp[i] / m;
911            }
912            return temp;
913        }
914
915        private double[] psi2 (double[]meanUtemp, long[][][]Uhtemp, int servicel,
916                               double delay) {
917            int j;
918            double U;
919            double[] res = new double[maxBacklog];
920            double[][] temp2 = new double[measHistorym][timeSlotsT];
921            int m = measHistorym;
922            boolean z = true;
923            for (int y = 0; y < measHistorym; y++) {
924                z = true;
925                for (int i = 0; i < maxBacklog; i++) {
926                    if (Uhtemp[servicel][y][i] != 0)
927                        z = false;
928                }
929                if (z)
930                    m--;
931            }
932            if (m < 2)
933                m = 2;
934
935            for (int mm = measHistorym - 1; mm >= measHistorym - m; mm--) {
936                temp2[mm] = inverseU (Uhtemp[servicel][mm], delay);
937            }
938
939            for (int i = 0; i < timeSlotsT; i++) {
940                for (int y = measHistorym - 1; y > measHistorym - m; y--) {
941                    res[i] +=
942                        ((temp2[y][i] - meanUtemp[i]) * (temp2[y][i] - meanUtemp[i]));
943                }
944                res[i] /= (m - 1);
945            }
946            return res;
947        }
948
949
950
951        //called by requesting processor, uses AC descission implemented above
952        public boolean remote_admit (Packet p) {
953            boolean ok = admit_flow (p);
954            if (ok) {
955                if (Packet.statsOk || true)
956                    Packet.admitFlows[p.sl]++;
957                else
958                    Packet.admitFlowsPreOk[p.sl]++;
959            } else {
```

```
960                 if (Packet.statsOk || true)
961                     Packet.rejFlows[p.sl]++;
962                 else
963                     Packet.rejFlowsPreOk[p.sl]++;
964             }
965             return ok;
966         }
967
968     }
969
```

```
1   /*
2    * BBRMSProcessor.java: BBRMSProcessor is a subclass of Processor. Together with BBRM
    SSwitch
3    * it implements the link-by-link AC scheme. The sending node calls a AC method in th
    e first
4    * switch, which calls in next switch etc. until the destination is reached.
5    */
6
7   package base;
8
9   import java.util.Random;
10  import java.util.Vector;
11  import java.lang.Math;
12
13  public class BBRMSProcessor extends Processor {
14      boolean ADDEBUG = false;
15      boolean DEBUG = false;
16      boolean DEBUGAC = false;
17      static boolean NEWDEBUG = false, INTERESTED = true;
18      private int flowMessages = 0;
19      long resvtime, timeout = 10000;
20      private int num_flows = 0;
21      private double peakRate = 2, delay = 500;
22      private double alpha = 0;
23      private int totalSL =
24          Kernel.CV[Kernel.highpri_service_levels] +
25          Kernel.CV[Kernel.lowpri_service_levels];
26      private int index;
27      private Random rand = new Random ();
28
29      private double sendPacketCounter = -12;
30      private double sendPacketCounterInit = 0;
31      private boolean NEW_FLOW = false;
32      private long lnf = 0;
33      private boolean localAdded = false, localUse1 = true;
34      public static double offeredFlows = 0;
35
36      //static things for ensuring that all processors add a new flow
37      public static int newFlowCounter = 0;
38      private static boolean stabilise = false;
39      public static boolean NO_MORE_FLOWS = false;
40      public static int increases = 0, numNoAdditions = 0;
41      private static long lastNewFlow = 0, timeBetweenFlows = 0;
42      private static int lastSent = 1;
43      private int localLastSent = 0;
44      private Vector retry = new Vector ();
45      private boolean toosmall = false;
46      private boolean firsttime = true;
47      private int flowNum = 1;
48      public double flowBW;
49      int[] flowMapping;
50      //self-similar
51      boolean print = true;
52
53      /*called when a new flow is accepted or rejected, handles stabilising of network
    when done adding flows.*/
54      private static boolean doneNewFlow (boolean admitted) {
55          if (admitted) {
56              stabilise = true;
57              numNoAdditions = 0;
58          }
59
60          newFlowCounter++;
61          if (NEWDEBUG)
62              System.out.println ("Number of processors who have added their flow: " +
63                                  newFlowCounter);
64          if (newFlowCounter ==
65                  Kernel.CV[Kernel.num_switches] *
66                  Kernel.CV[Kernel.Processor_per_switch]) {
67              if (!stabilise) {
68                  numNoAdditions++;
69                  if (INTERESTED)
70                      System.out.println ("No new additions");
71              }
72              if (NEWDEBUG)
```

```
73                  System.out.println ("Time of stable period: " + numNoAdditions);
74              newFlowCounter = 0;
75              if (NEWDEBUG)
76                  System.out.println ("Time to stabilise if necessary");
77              increases++;
78              // Done adding flows, time to stabilize
79              if ((numNoAdditions == 10) || increases >= offeredFlows - 1) {
80                  if (NEWDEBUG)
81                      System.out.
82                      println
83                      ("%% 10 times without admittance, time to stabilize and get resul
    ts %%");
84                  Kernel.unstable = true;
85                  NO_MORE_FLOWS = true;
86                  Kernel.admitting = false;
87                  stabilise = false;
88                  lastSent++;
89                  if (NEWDEBUG || INTERESTED)
90                      System.out.
91                      println ("%Number of times new flows have been added: " +
92                               increases + "\n%Stabilising network");
93              } else {
94                  Kernel.stabiliseEnd = Kernel.Now;
95                  lastSent++;
96                  if (NEWDEBUG || INTERESTED)
97                      System.out.
98                      println ("%Number of times new flows have been added: " +
99                               increases + "\n%No need to restabilise network.");
100
101             }
102             stabilise = false;
103         }
104         return true;
105     }
106     /* Called from simulate() when done stabilising */
107     public static boolean stableDone () {
108         Kernel.stabiliseEnd = Kernel.Now;
109         lastSent++;
110         System.out.println ("%Done stabilising...");
111         return true;
112     }
113
114     private int NumPackInFlow (int min, int max) {
115         return min + rand.nextInt (max);
116
117     }
118
119     //Dummy constructor
120     public BBRMSProcessor (int id) {
121         this (id, Kernel.CV[Kernel.Processor_receive_buffers],
122               Kernel.CV[Kernel.Processor_send_q_size],
123               Kernel.CV[Kernel.Processor_service_time],
124               Kernel.CV[Kernel.Processor_drop_connects],
125               Kernel.CV[Kernel.Node_drop_links]);
126     }
127
128
129     public BBRMSProcessor (int id, int receive_buffers, int send_q_size,
130                            int service_time, int drop_connects, int drop_links) {
131         super (id, receive_buffers, send_q_size, service_time, drop_connects,
132                drop_links);
133
134         Kernel.admitting = true;
135         double f = 1000000, h = 0;
136         for (int i = 0; i < totalSL; i++) {
137             f = Math.min (f, Kernel.BWflow[i]);
138             h = Math.max (h, Kernel.slPercentage[i]);
139         }
140         h = Kernel.minMean * (float) Math.pow (h / 100.0, -1);
141         //double the flowbandwidth of every other flow.
142         if (Kernel.DOUBLE_BW && Math.IEEEremainder (adctrlID, 2) == 0)
143             flowBW = 2 * Kernel.BWflow[localSL];
144         else
145             flowBW = Kernel.BWflow[localSL];
146         Packet.bandwidths[localSL][adctrlID] = flowBW;
```

```
147             //calculates the number of flows that may be added.
148             h = Kernel.CV[Kernel.mean_packet_interarrival_time] * (float) Math.pow (Kerne
    l.slLoad[localSL] / 100.0, -1);
149             double temp = (double) 1 / Kernel.CV[Kernel.mean_packet_interarrival_time];
150             localOfferedFlows = (temp * (Kernel.slPercentage[localSL] / 100)) / flowBW;
151             if (localOfferedFlows < 2)
152                 localOfferedFlows = 2;
153             Packet.numFlows[localSL] = Math.max (Packet.numFlows[localSL], (int) localOff
    eredFlows);
154             offeredFlows = Math.max (offeredFlows, localOfferedFlows);
155             maxFlows = localOfferedFlows;
156             flowNum = (int) localOfferedFlows - 1;
157             //debug
158             if (INTERESTED)
159                 System.out.println ("Flows to be offered sl" + localSL + ": " +
160                                     localOfferedFlows);
161             timeBetweenFlows =
162                 (long) (Kernel.CV[Kernel.simulation_cycles] / offeredFlows);
163             timeBetweenFlows = 1;
164             if (NEWDEBUG)
165                 System.out.println ("%Time between flows: " + timeBetweenFlows +
166                                     "\n%Number of flows to be offered: " +
167                                     offeredFlows);
168             flowMapping = new int[(int) maxFlows + 1];
169             //self-similar
170             flows = new Flow[(int) localOfferedFlows + 3];
171         }
172 
173         public void packet_produced_enter (EasterEgg egg) {
174             Packet p = null;
175             int sl = 0;
176             sl = localSL;
177             if (DEBUG)
178                 System.out.println (toString () + " packet_produced_enter");
179             //Increase send_counter no matter if a packet is sent, creates correct statis
    tics
180             Packet.dummy_send ();
181 
182             //A local retry queue, not used
183             if (!retry.isEmpty ()) {
184                 Packet t = (Packet) retry.firstElement ();
185                 send_pack (t);
186                 retry.remove (0);
187                 Packet.packets_resent++;
188             }
189 
190             //set up necessary variables for adding a new flow, determine if new flow sho
    uld be added.
191             long e = Kernel.Now - Kernel.stabiliseEnd;
192             if (NEWDEBUG)
193                 if (!Kernel.unstable && !localAdded)
194                     System.out.println ("tid siden sist nye flyt: " + e);
195             if (!NO_MORE_FLOWS && !Kernel.unstable && (lastSent != localLastSent) && (Ker
    nel.Now - Kernel.stabiliseEnd >= Kernel.Cycle (timeBetweenFlows))) {
196                 localLastSent = lastSent;
197                 if (NEWDEBUG)
198                     System.out.println ("Time to add new flow: " + id);
199                 //f is the number of packets to be produced between each time a packet sh
    ould be sent according to the number of flow sending.
200                 double f = 1 / (flowBW * Kernel.slLoad[localSL] * (num_flows + 1));
201                 if (num_flows + 1 < (int) localOfferedFlows) {
202                     NEW_FLOW = true;
203                     if (Kernel.SELF_SIMILAR || Kernel.SELF_SIM_NEW) {
204                         //make a pessimistic assumption about peakrate
205                         //Kernel.SELF_TIGHT is a variation on self_similar, not used
206                         if (Kernel.SELF_TIGHT) {
207                             peakRate =
208                                 ((double) 1 /
209                                  Kernel.CV[Kernel.mean_packet_interarrival_time]);
210                             peakRate *= (Kernel.Traffic.payload_size (id) + 2);
211                         } else {
212                             peakRate =
213                                 ((double) flowBW * 1.5) * (localOfferedFlows *
214                                                            Kernel.slLoad[localSL]);
215                             peakRate /= (localOfferedFlows * Kernel.slLoad[localSL] -
```

```
216                                         2 *
217                                         Kernel.CV[Kernel.
218                                                   std_dev_packet_interarrival_time]);
219                             peakRate *= (Kernel.Traffic.payload_size (id) + 2);
220                         }
221                 } else {
222                     peakRate = flowBW;
223                     peakRate *= (Kernel.Traffic.payload_size (id) + 2);
224                 }
225
226             } else {
227                 //handle special case of flowbw being larger than packet sending rate
    ,
228                 //reduces flowbw to sending rate.
229                 if (firsttime) {
230                     peakRate =
231                         (Kernel.Traffic.payload_size (id) +
232                          2) / (Kernel.slLoad[localSL]);
233                     if (NEWDEBUG)
234                         System.out.println ("Fisttime with peakrate: " + peakRate);
235                     NEW_FLOW = true;
236                     toosmall = true;
237                     firsttime = false;
238                 } else {
239                     if (NEWDEBUG)
240                         System.out.
241                         println
242                         ("Not room for another flow produced between flows = " +
243                          f);
244                     Packet.rejFlowsPreOk[localSL]++;
245                     this.doneNewFlow (false);
246                 }
247             }
248         }
249
250         //time to add new flow
251         if (NEW_FLOW) {
252             NEW_FLOW = false;
253
254             flowMessages =
255                 NumPackInFlow (Kernel.min_flow_length,
256                                Kernel.max_flow_length - Kernel.min_flow_length);
257             resvtime = Kernel.Now;
258             if (DEBUG)
259                 System.out.println ("Sending new reservation\n");
260
261             p = createPacket (this,
262                               pid + id,
263                               Kernel.kernel.packetDestination (this), 20, sl);
264
265
266             if (DEBUGAC) {
267                 System.out.println ("peakRate: " + peakRate);
268                 System.out.println ("Payload size = " +
269                                     Kernel.Traffic.payload_size (id));
270                 System.out.println ("Interarrival time = " +
271                                     Kernel.Traffic.
272                                     packet_interarrival_time ((int) Kernel.
273                                                               slLoad[localSL],
274                                                               Kernel.CV[Kernel.
275                                                                         std_dev_packe
    t_interarrival_time]));
276             }
277             delay = Kernel.delayBound[p.sl];
278
279             p.payloadpointer = new Payload (Kernel.Now, peakRate, delay);
280             p.payloadpointer.resvinit = true;
281             p.payloadpointer.resvok = true;
282             p.payloadpointer.resvnotok = false;
283             iba_last_dest = p.destination;
284             iba_last_sl = p.sl;
285             if (Kernel.Now < Kernel.Cycle (Kernel.CV[Kernel.simulation_cycles])
286                     && Kernel.Now > 0) {
287                 long indextest =
288                     Packet.numFlowHist * ((Kernel.Now)) /
```

```
289                     Kernel.Cycle (Kernel.CV[Kernel.simulation_cycles]);
290                 }
291
292                 //don't perform admission control on SL5 if we have BEST_EFFORT traffic
293                 //calculate time between sending packest belonging to which flow.
294                 if ((Kernel.BEST_EFFORT && localSL == 4) || admit_flow (p)
295                         || Kernel.admitall) {
296                     Packet.admitFlows[p.sl]++;
297                     num_flows++;
298                     //self-similar
299                     if (flows[num_flows] != null)
300                         System.out.println ("Noe er galt");
301                     flows[num_flows] = new Flow (this, num_flows);
302                     double ratio = maxFlows / num_flows;
303                     double findex = 0;
304                     for (int fn = 0; fn < (int) maxFlows; fn++)
305                         flowMapping[fn] = 0;
306                     for (int fn = 0; fn < (int) num_flows; fn++) {
307
308                         if (fn * ratio < maxFlows && Kernel.EVEN_FLOW)
309                             flowMapping[(int) (fn * ratio)] = fn + 1;
310                         else if (!Kernel.EVEN_FLOW)
311                             flowMapping[fn] = fn + 1;
312                         else
313                             System.out.println ("mattefeil!!!!");
314                     }
315                     if (NEWDEBUG)
316                         System.out.println ("Admitted");
317                     this.doneNewFlow (true);
318                 } else {
319                     Packet.rejFlows[p.sl]++;
320                     if (NEWDEBUG)
321                         System.out.println ("Rejected");
322                     this.doneNewFlow (false);
323                 }
324                 EasterEgg egget = new EasterEgg ();
325                 egget.flyt = flows[num_flows];
326                 if (Kernel.SELF_SIM_NEW)
327                     executeFlow (egget);
328             }
329
330             //send packet
331             if (!Kernel.SELF_SIM_NEW)
332                 executeFlow (null);
333
334             if (!Kernel.stop_packet_generation) {
335                 //Only send 1 packet in FLOW_RESV state, not whole message
336                 // comment out first if to continue sending packets when waiting for rese
    rvation confirm.
337                 //if(flowState!=FLOW_RESV){
338                 if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null
339                         || iba_message_packetcount > iba_msg_size) {
340
341                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now +
342                             Kernel.Traffic.
343                             packet_interarrival_time
344                             ((int) Kernel.
345                              slLoad[localSL],
346                              Kernel.CV[Kernel.
347                                        std_dev_packet_interarrival_time],
348                              this));
349
350                 } else {
351                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now + 10);
352                 }
353             }
354
355             // keep track of number of packets send in this msg
356             if (Kernel.IBA_ENABLE_MESSAGE) {
357                 if (DEBUG)
358                     System.out.println ("Increasing message_packetcount\n");
359                 iba_message_packetcount++;
360             }
361         }
362
```

```
363         /* Sends a packet in normal opperation, without SELF_TIGHT */
364         public void executeFlow (EasterEgg egg) {
365             Packet p = null;
366             int sl = 0;
367             sl = localSL;
368             do {
369                 flowNum--;
370                 if (flowMapping[flowNum] != 0 || Kernel.SELF_SIM_NEW) {
371                     //self-similar
372                     if ((Kernel.SELF_SIM_NEW || flows[flowMapping[flowNum]].update_selfsi
    m_flow (this))) {
373                         if (Kernel.SELF_SIM_NEW)
374                             egg.flyt.update_selfsim_flow (this);
375                         if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null) {
376                             // first msg send from this processor or iba_msg not enabled
377                             if (print) {
378                                 //System.out.println("destinasjon: " + Kernel.kernel.pack
    etDestination(this).id);
379                                 print = false;
380                             }
381                             p = createPacket (this,
382                                               pid + id,
383                                               Kernel.kernel.packetDestination (this),
384                                               Kernel.Traffic.payload_size (id), sl);
385 
386                             //set the correct flow number of the packet.  Special case if
     SELF_SIM_NEW, not used/not working
387                             if (Kernel.SELF_SIM_NEW)
388                                 p.flowNum = egg.flyt.flowNum;
389                             else
390                                 p.flowNum = flowMapping[flowNum];
391 
392                             iba_last_dest = p.destination;
393                             iba_last_sl = p.sl;
394                             send_pack (p);
395                         } else {
396                             if ((iba_message_packetcount > iba_msg_size)
397                                     || !Kernel.IBA_ENABLE_MESSAGE) {
398                                 // reset counter
399                                 iba_message_packetcount = 0;
400 
401                                 p = createPacket (this,
402                                                   pid + id,
403                                                   iba_last_dest,
404                                                   Kernel.Traffic.payload_size (id),
405                                                   iba_last_sl);
406                                 if (Kernel.SELF_SIM_NEW)
407                                     p.flowNum = egg.flyt.flowNum;
408                                 else
409                                     p.flowNum = flowMapping[flowNum];
410                                 if (DEBUG)
411                                     System.out.println ("Decreasing flowMessages\n");
412                                 send_pack (p);
413                             } else {
414                                 // continuing and existing message
415                                 if (DEBUG)
416                                     System.out.println ("Sending continued message\n");
417                                 p = createPacket (this,
418                                                   pid + id,
419                                                   iba_last_dest,
420                                                   Kernel.Traffic.payload_size (id),
421                                                   iba_last_sl);
422                                 if (Kernel.SELF_SIM_NEW)
423                                     p.flowNum = egg.flyt.flowNum;
424                                 else
425                                     p.flowNum = flowMapping[flowNum];
426                                 send_pack (p);
427                             }
428                         }
429                     }
430                 }
431             } while (flowNum > 0 && Kernel.SELF_TIGHT && Kernel.SELF_SIMILAR);
432 
433             if (flowNum <= 0) {
434                 flowNum = (int) maxFlows;
```

```
435             }
436         }
437
438         public void packet_received_enter (EasterEgg egg) {
439             Packet p = null;
440             bound_receive_buffers--;
441             (new Event (this, BINDABLE)).schedule (Kernel.Now, bindable_flag);
442
443
444             if (!egg.flit.isKill () && egg.flit.packet.destination != this) {
445                 System.out.println (Kernel.time () + toString ()
446                                     + "Misrouted packet (SL " + egg.flit.packet.sl +
447                                     "): " + egg.flit.toString ());
448                 System.out.println ("   Src: " + egg.flit.packet.source + " " +
449                                     egg.flit.packet.source.hashCode ());
450                 System.out.println ("   Dst: " + egg.flit.packet.destination + " " +
451                                     egg.flit.packet.destination.hashCode ());
452                 System.out.println ("   Cur: " + this + " " + this.hashCode ());
453             }
454
455         }
456
457         private void send_pack (Packet pp) {
458             pid += Kernel.max_switches;
459             Packet.sentPackets[pp.sl]++;
460
461             //normally used for internal send_q, not used.
462             if (send_q.full ()) {
463                 //if(pp.payloadpointer!=null) enqueue(pp);
464                 Packet.packets_rejected[pp.sl]++;
465                 Packet.incRejected (pp.hops, pp.sl);
466             }
467             send_q.insert (pp);
468             if (in_transmission < drop_connects.length) {
469                 send_enter (null);
470
471             }
472         }
473
474         private void enqueue (Packet p) {
475             retry.addElement (p);
476         }
477
478         //Admition routine used in link-by-link.  Reserves resources here in processor an
      d likewise in BBRMSSwitch.java
479         private boolean admit_flow (Packet p) {
480             VLane outvl = drop_links[p.sl].vlanes[p.sl];
481             BBRMSSwitch s = (BBRMSSwitch) outvl.receiver;
482             if (outvl.link.bw - p.payloadpointer.peak >= 0) {
      //continue if admitted
483                 outvl.link.bw -= p.payloadpointer.peak;
484                 if (!s.next_admit_flow (p, outvl)) {
485                     outvl.link.bw += p.payloadpointer.peak;
486                     return false;
487                 }
488                 return true;
489             }
490             return false;
491         }
492
493         //Processor has no link to be congested, allways accept new flow.
494         public boolean next_admit_flow (Packet p, VLane vl) {
495             if (p.destination != this)
496                 System.out.
497                 println
498                 ("!!!!Wrong destination, something is wrong with the admission-routing.\n
      "
499                  + p.destination.toString () + " is not " + toString () + "!");
500             else if (NEWDEBUG)
501                 System.out.println ("Correct destination");
502             return true;
503         }
504
505
506
```

```
1   /*
2    * PROBEProcessor.java:  PROBEProcessor subclasses Processor. It implements the
3    * "Jitter Probing" AC scheme.  The processor sends probePacketsInit (=6) probe
4    * packets through the network to the reciever which monitors the transmission time.
5    * The probe packets are sent on probeSL if the parameter "probelow" is given,
6    * which might be a separate SL. Without "probelow", the probes are sent on localSL.
7    */
8
9   package base;
10
11  import java.util.Random;
12  import java.util.Vector;
13  import java.lang.Math;
14
15  public class PROBEProcessor extends Processor {
16      boolean ADDEBUG = false;
17      boolean DEBUG = false;
18      boolean DEBUGAC = false;
19      static boolean NEWDEBUG = false, INTERESTED = true;
20      private int flowMessages = 0;
21      private boolean wasnotOk = true;
22      long resvtime, timeout = 10000;
23      private int num_flows = 0;
24      private double peakRate = 2, delay = 500;
25      private double alpha = 0;
26      private int numFlowsInNet = 0;
27      private int totalSL =
28          Kernel.CV[Kernel.highpri_service_levels] +
29          Kernel.CV[Kernel.lowpri_service_levels];
30      private int index;
31      private int flowRetry = 100, retryCounter = 1;
32      private boolean onlyFlows = true;
33      private Random rand = new Random ();
34
35      private double sendPacketCounter = -12;
36      private double sendPacketCounterInit = 0;
37      private boolean NEW_FLOW = false;
38      private long lnf = 0;
39      private boolean localAdded = false, localUse1 = true;
40      public static double offeredFlows = 0;
41
42      //static things for ensuring that all processors add a new flow
43      public static int newFlowCounter = 0;
44      private static boolean stabilise = false;
45      public static boolean adding1 = false, adding2 = false, use1 =
46                                      true, NO_MORE_FLOWS = false;
47      public static int increases = 0, numNoAdditions = 0;
48      private static long lastNewFlow = 0, timeBetweenFlows = 0;
49      private static int lastSent = 1;
50      private int localLastSent = 0;
51      private int probePackets = -1, probePacketsInit = 6, probeRec =
52                                 0, probeSent = 0, flowsSent = 0, rejectedCounter =
    0, probeDropped;
53      private long probeMin = 100000000, probeMax = 0, probeSessionNum =
54                             0, probeSessionExpected = 1, probeLast = 0;
55      private Vector retry = new Vector ();
56      private boolean toosmall = false, sendProbe = false;
57      private boolean firsttime = true;
58      private Packet probePack;
59      public int recProbe = 0;
60      private static int admittedthistime = 0;
61      private long lastprobe = 0;
62      private int randomSendDelay = 0;
63      private int probeSL = 5;
64      private double flowBW;
65      private int newLoad;
66      int[] flowMapping;
67      int lastBufferFill = 0, maxBufferFill = 0;
68
69
70      //used for long run with all loads, not working, not used.
71      public void continueAdmitting (int load) {
72          newLoad = load;
73          Kernel.admitting = true;
74          NO_MORE_FLOWS = false;
```

```
75            NEW_FLOW = false;
76            localAdded = false;
77            localUse1 = true;
78            stabilise = false;
79            adding1 = false;
80            adding2 = false;
81            use1 = true;
82            NO_MORE_FLOWS = false;
83            sendProbe = false;
84            double temp = (double) 1 / load;
85            localOfferedFlows =
86                (temp * (Kernel.slPercentage[localSL] / 100)) / flowBW;
87            Packet.numFlows[localSL] =
88                Math.max (Packet.numFlows[localSL], (int) localOfferedFlows);
89            offeredFlows = Math.max (offeredFlows, localOfferedFlows);
90
91        }
92
93        /*Called when new flow has been added un/succesfully*/
94        private static boolean doneNewFlow (boolean admitted) {
95            if (admitted) {
96                stabilise = true;
97                numNoAdditions = 0;
98                admittedthistime++;
99            }
100
101           newFlowCounter++;
102           if (NEWDEBUG)
103               System.out.println ("Number of processors who have added their flow: " +
104                                   newFlowCounter);
105           if (newFlowCounter ==
106                   Kernel.CV[Kernel.num_switches] *
107                   Kernel.CV[Kernel.Processor_per_switch]) {
108               if (INTERESTED)
109                   System.out.println ("Kernel time now (/10): " + Kernel.Now / 10);
110               if (INTERESTED)
111                   System.out.println ("Flows admitted: " + admittedthistime);
112               admittedthistime = 0;
113               if (!stabilise)
114                   numNoAdditions++;
115               if (NEWDEBUG)
116                   System.out.println ("Time of stable period: " + numNoAdditions);
117               newFlowCounter = 0;
118               if (NEWDEBUG)
119                   System.out.println ("Time to stabilise if necessary");
120               increases++;
121               if (INTERESTED)
122                   System.out.println ("Number of flows added: " + increases);
123               if ((numNoAdditions == 1) || increases >= offeredFlows - 1) {
124                   if (NEWDEBUG || INTERESTED)
125                       System.out.
126                       println
127                       ("%% 10 times without admittance, time to stabilize and get resul
    ts %%");
128                   Kernel.unstable = true;
129                   NO_MORE_FLOWS = true;
130                   Kernel.admitting = false;
131                   stabilise = false;
132                   lastSent++;
133                   if (NEWDEBUG || INTERESTED)
134                       System.out.
135                       println ("%Number of times new flows have been added: " +
136                                increases + "\n%Stabilising network");
137               } else {
138                   Kernel.stabiliseEnd = Kernel.Now;
139                   lastSent++;
140                   if (NEWDEBUG || INTERESTED)
141                       System.out.
142                       println ("%Number of times new flows have been added: " +
143                                increases + "\n%No need to restabilise network.");
144
145               }
146               stabilise = false;
147           }
148           return true;
```

```
149     }
150
151     public static boolean stableDone () {
152         Kernel.stabiliseEnd = Kernel.Now;
153         lastSent++;
154         if (NEWDEBUG || INTERESTED)
155             System.out.println ("%Done stabilising...");
156         return true;
157     }
158
159     private int NumPackInFlow (int min, int max) {
160         return min + rand.nextInt (max);
161
162     }
163
164     public PROBEProcessor (int id) {
165         this (id, Kernel.CV[Kernel.Processor_receive_buffers],
166               Kernel.CV[Kernel.Processor_send_q_size],
167               Kernel.CV[Kernel.Processor_service_time],
168               Kernel.CV[Kernel.Processor_drop_connects],
169               Kernel.CV[Kernel.Node_drop_links]);
170     }
171
172
173     public PROBEProcessor (int id, int receive_buffers, int send_q_size,
174                            int service_time, int drop_connects, int drop_links) {
175         super (id, receive_buffers, send_q_size, service_time, drop_connects,
176                drop_links);
177         Kernel.admitting = true;
178         double f = 1000000, h = 0;
179         for (int i = 0; i < totalSL; i++) {
180             f = Math.min (f, Kernel.BWflow[i]);
181             h = Math.max (h, Kernel.slLoad[i]);
182         }
183         h = Kernel.minMean * (float) Math.pow (h / 100.0, -1);
184         //doubles flowBW of every other flow
185         if (Kernel.DOUBLE_BW && Math.IEEEremainder (adctrlID, 2) == 0)
186             flowBW = 2 * Kernel.BWflow[localSL];
187         else
188             flowBW = Kernel.BWflow[localSL];
189         Packet.bandwidths[localSL][adctrlID] = flowBW;
190         double temp =
191             (double) 1 / Kernel.CV[Kernel.mean_packet_interarrival_time];
192         newLoad = Kernel.CV[Kernel.mean_packet_interarrival_time];
193         localOfferedFlows =
194             (temp * (Kernel.slPercentage[localSL] / 100)) / flowBW;
195         if (localOfferedFlows < 2)
196             localOfferedFlows = 2;
197         maxFlows = localOfferedFlows;
198         offeredFlows = Math.max (offeredFlows, localOfferedFlows);
199         timeBetweenFlows = 1;
200         System.out.println ("%Time between flows: " + timeBetweenFlows +
201                             "\n%Number of flows to be offered: " +
202                             localOfferedFlows);
203         randomSendDelay = NumPackInFlow (0, 1000);
204         flowMapping = new int[(int) maxFlows];
205         flowsSent = (int) localOfferedFlows;
206         flows = new Flow[(int) localOfferedFlows + 3];
207     }
208
209     public void packet_produced_enter (EasterEgg egg) {
210         Packet p = null;
211         int sl = 0;
212         sl = localSL;
213
214         if (DEBUG)
215             System.out.println (toString () + " packet_produced_enter");
216
217         long e = Kernel.Now - Kernel.stabiliseEnd;
218         if (NEWDEBUG)
219             if (!Kernel.unstable && !localAdded)
220                 System.out.println ("tid siden sist nye flyt: " + e);
221         //determines if it is time to add another flow
222         if (!NO_MORE_FLOWS && !Kernel.unstable && (lastSent != localLastSent)
223                 && (Kernel.Now - Kernel.stabiliseEnd >=
```

```
224                     Kernel.Cycle (timeBetweenFlows) + randomSendDelay)) {
225             localLastSent = lastSent;
226             if (NEWDEBUG)
227                 System.out.println ("Time to add new flow: " + id);
228             //f is the number of packets to be produced between each time a packet sh
ould be sent according to the number of flow sending.
229             double f = 1 / (flowBW * Kernel.slLoad[localSL] * (num_flows + 1));
230             if (num_flows + 1 <= (int) localOfferedFlows) {
231                 NEW_FLOW = true;
232                 //make a pessimistic assumption about peakrate
233                 if (Kernel.SELF_SIMILAR) {
234                     if (Kernel.SELF_TIGHT) {
235                         peakRate =
236                             ((double) 1 /
237                              Kernel.CV[Kernel.mean_packet_interarrival_time]);
238                         peakRate *= (Kernel.Traffic.payload_size (id) + 2);
239                     } else {
240                         peakRate = (double) (1 / (flowBW * 2));
241                         peakRate -=
242                             2 * Kernel.CV[Kernel.std_dev_packet_interarrival_time];
243                         peakRate = 1 / peakRate;
244                         peakRate *= (Kernel.Traffic.payload_size (id) + 2);
245                     }
246                 } else {
247                     peakRate = flowBW;
248                     peakRate *= (Kernel.Traffic.payload_size (id) + 2);
249                     peakRate *= ((localOfferedFlows * Kernel.slLoad[localSL]));
250                     peakRate /=
251                         ((localOfferedFlows * Kernel.slLoad[localSL] -
252                           2 * Kernel.CV[Kernel.std_dev_packet_interarrival_time]));
253                 }
254             } else {
255                 if (NEWDEBUG || INTERESTED)
256                     System.out.
257                     println ("Not room for another flow produced between flows = "
258                              + ((int) (localOfferedFlows) - num_flows));
259                 Packet.rejFlowsPreOk[localSL]++;
260                 this.doneNewFlow (false);
261 
262             }
263         }
264         //It is time to add another flow
265         if (NEW_FLOW) {
266             NEW_FLOW = false;
267             probeDropped = 0;
268             probeSessionNum++;
269             flowMessages = NumPackInFlow (Kernel.min_flow_length,
270                                           Kernel.max_flow_length - Kernel.min_flow_le
ngth);
271             resvtime = Kernel.Now;
272             if (DEBUG)
273                 System.out.println ("Sending new reservation\n");
274 
275             p = createPacket (this,
276                               pid + id,
277                               Kernel.kernel.packetDestination (this), 20, sl);
278 
279 
280             if (DEBUGAC) {
281                 System.out.println ("peakRate: " + peakRate);
282                 System.out.println ("Payload size = " +
283                                     Kernel.Traffic.payload_size (id));
284                 System.out.println ("Interarrival time = " +
285                                     Kernel.Traffic.
286                                     packet_interarrival_time ((int) Kernel.
287                                                               slLoad[localSL],
288                                                               Kernel.CV[Kernel.
289                                                                         std_dev_packe
t_interarrival_time]));
290             }
291             delay = Kernel.delayBound[p.sl];
292             p.payloadpointer = new Payload (Kernel.Now, peakRate, delay);
293             iba_last_dest = p.destination;
294             iba_last_sl = p.sl;
295             if (Kernel.Now < Kernel.Cycle (Kernel.CV[Kernel.simulation_cycles])
```

```
296                         && Kernel.Now > 0) {
297                     long indextest =
298                         Packet.numFlowHist * ((Kernel.Now)) /
299                         Kernel.Cycle (Kernel.CV[Kernel.simulation_cycles]);
300                     //System.out.println(index);
301                 }
302
303
304                 num_flows++;
305                 //self-similar
306                 flows[num_flows] = new Flow (this, num_flows);
307
308                 //this SL shall send probes on probeSL
309                 if ((localSL != 4 && !Kernel.PROBE_LOWPRI) || (Kernel.PROBE_LOWPRI && loc
    alSL != probeSL && localSL != 4)) {
310                     if (localOfferedFlows <= num_flows) {
311                         this.doneNewFlow (false);
312                         num_flows--;
313                         Packet.rejFlows[localSL]++;
314                     } else {
315                         if (localSL == 2) //this is not used
316                             lastBufferFill = send_q.size ();
317                         probePack = p;
318                         probePackets = probePacketsInit;
319                         double ratio = maxFlows / num_flows;
320                         double findex = 0;
321                         for (int fn = 0; fn < (int) maxFlows; fn++)
322                             flowMapping[fn] = 0;
323                         for (int fn = 0; fn < (int) num_flows; fn++) {
324                             if (fn * ratio < maxFlows && Kernel.EVEN_FLOW)
325                                 flowMapping[(int) (fn * ratio)] = fn + 1;
326                             else
327                                 flowMapping[fn] = fn + 1;
328                         }
329                     }
330                 }
331
332                 //This is probeSL, should not send or admitt packets
333                 else if (Kernel.PROBE_LOWPRI && localSL == probeSL) {
334                     this.doneNewFlow (false);
335                     num_flows--;
336                 }
337                 //this is SL5, not subject to AC, admit all.
338                 else if (localSL == 4) {
339                     Packet.admitFlows[localSL]++;
340                     this.doneNewFlow (true);
341                     double ratio = maxFlows / num_flows;
342                     double findex = 0;
343                     for (int fn = 0; fn < (int) maxFlows; fn++)
344                         flowMapping[fn] = 0;
345                     for (int fn = 0; fn < (int) num_flows; fn++) {
346                         if (fn * ratio < maxFlows && Kernel.EVEN_FLOW)
347                             flowMapping[(int) (fn * ratio)] = fn + 1;
348                         else
349                             flowMapping[fn] = fn + 1;
350
351                     }
352
353
354                 }
355             }
356             flowsSent--;
357             if (flowMapping[flowsSent] != 0) {
358                 //self-similar, if this flow should send a packet, determine if probeing
    is over..
359                 if ((flows[flowMapping[flowsSent]].update_selfsim_flow (this)) || !Kernel
    .SELF_SIMILAR) {
360                     //Determines if probe period is over, is over if sent probes == recie
    ved + rejected, and have sent all.
361                     if (probePackets == -2 || (probePackets == 0 && localSL != 4 && (((PR
    OBEProcessor) (probePack.destination)).recProbe + probeDropped >= probePacketsInit)))
    {
362                         probePackets = -1;
363
364                         if (((PROBEProcessor) (probePack.destination)).recProbe < probePa
```

```
cketsInit || !probePack.destination.probeDone (probePack)) {
365                     //Flow is rejected
366                     probePack.destination.probeReset ();
367                     num_flows--;
368                     double ratio = maxFlows / num_flows;
369                     double findex = 0;
370                     for (int fn = 0; fn < (int) maxFlows; fn++)
371                         flowMapping[fn] = 0;
372                     for (int fn = 0; fn < (int) num_flows; fn++) {
373                         if (fn * ratio < maxFlows && Kernel.EVEN_FLOW)
374                             flowMapping[(int) (fn * ratio)] = fn + 1;
375                         else
376                             flowMapping[fn] = fn + 1;
377 
378                     }
379                     Packet.rejFlows[localSL]++;
380                     rejectedCounter++;
381                     this.doneNewFlow (false);
382                 } else {
383                     //flow is admitted
384                     probePack.destination.probeReset ();
385                     Packet.admitFlows[localSL]++;
386                     rejectedCounter = 0;
387                     this.doneNewFlow (true);
388                 }
389                 probeSent = 0;
390                 probeDropped = 0;
391             } else if (probePackets == 0 && localSL != 4) {
392                 probePackets = -2;
393             }
394 
395             //keep track of the number of flows being sent.
396             if (num_flows > localOfferedFlows)
397                 System.out.println ("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
398             //Send probepacket or conclude probing.
399             if (flowMapping[flowsSent] == num_flows && probePackets > 0) {
400                 sendProbe = true;
401                 probePackets--;
402             }
403 
404             //Probes ar typically done on SL6 -> localSL=5, with a weight of 1 an
d no traffic
405             if ((Kernel.PROBE_LOWPRI && localSL != probeSL)
406                     || !Kernel.PROBE_LOWPRI) {
407                 if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null) {
408                     // first msg send from this processor or iba_msg not enabled
409                     p = createPacket (this,
410                                       pid + id,
411                                       Kernel.kernel.packetDestination (this),
412                                       Kernel.Traffic.payload_size (id), sl);
413                     p.flowNum = flowMapping[flowsSent];
414                     iba_last_dest = p.destination;
415 
416                     iba_last_sl = p.sl;
417 
418                     if (sendProbe && localSL != 4) {
419                         //send on other SL if low priority probing is used
420                         if (Kernel.PROBE_LOWPRI) {
421                             p.sl = probeSL;
422                         }
423                         p.processorSL = localSL;
424 
425                         sendProbe = false;
426                         probeSent++;
427                         if (NEWDEBUG)
428                             System.out.
429                             println ("Sending probepacket, probePackets=" +
430                                      probePackets + "\nHave sent " +
431                                      probeSent + " probepackets.");
432                         p.payloadpointer = new Payload (Kernel.Now);
433                         p.payloadpointer.probe = true;
434                         p.payloadpointer.probeSessionNum = probeSessionNum;
435                         //used for sendbuffer monitoring by localSL=2, alternativ
 AC, not used
436                         maxBufferFill =
```

```
437                                 Math.max (maxBufferFill, send_q.size ());
438                         }
439                         send_pack (p);
440                     } else {
441                         if ((iba_message_packetcount > iba_msg_size)
442                                 || !Kernel.IBA_ENABLE_MESSAGE) {
443                             // reset counter
444                             iba_message_packetcount = 0;
445                             p = createPacket (this,
446                                               pid + id,
447                                               iba_last_dest,
448                                               Kernel.Traffic.payload_size (id),
449                                               iba_last_sl);
450
451                             p.flowNum = flowMapping[flowsSent];
452                             if (DEBUG)
453                                 System.out.println ("Decreasing flowMessages\n");
454                             if (sendProbe && localSL != 4) {
455                                 if (Kernel.PROBE_LOWPRI) {
456                                     p.sl = probeSL;
457                                 }
458                                 p.processorSL = localSL;
459
460                                 sendProbe = false;
461                                 p.payloadpointer = new Payload (Kernel.Now);
462                                 probeSent++;
463                                 p.payloadpointer.probe = true;
464                                 p.payloadpointer.probeSessionNum =
465                                     probeSessionNum;
466                                 maxBufferFill =
467                                     Math.max (maxBufferFill, send_q.size ());
468                             }
469                             send_pack (p);
470                         } else {
471                             // continuing and existing message
472                             if (DEBUG)
473                                 System.out.println ("Sending continued message\n");
474                             p = createPacket (this,
475                                               pid + id,
476                                               iba_last_dest,
477                                               Kernel.Traffic.payload_size (id),
478                                               iba_last_sl);
479
480                             p.flowNum = flowMapping[flowsSent];
481                             if (sendProbe && localSL != 4) {
482                                 if (Kernel.PROBE_LOWPRI) {
483                                     p.sl = probeSL;
484                                 }
485
486                                 p.processorSL = localSL;
487
488                                 sendProbe = false;
489                                 probeSent++;
490                                 p.payloadpointer = new Payload (Kernel.Now);
491                                 p.payloadpointer.probe = true;
492                                 p.payloadpointer.probeSessionNum =
493                                     probeSessionNum;
494                                 maxBufferFill =
495                                     Math.max (maxBufferFill, send_q.size ());
496                             }
497                             send_pack (p);
498                         }
499                     }
500                 }
501             }
502         }
503
504         if (flowsSent <= 0) {
505             flowsSent = (int) maxFlows;
506             Packet.offered += localOfferedFlows;
507         }
508
509
510         if (!Kernel.stop_packet_generation) {
511             //Only send 1 packet in FLOW_RESV state, not whole message
```

```
512                 // comment out first if to continue sending packets when waiting for rese
    rvation confirm.
513                 //if(flowState!=FLOW_RESV){
514                 if (!Kernel.IBA_ENABLE_MESSAGE || iba_last_dest == null
515                         || iba_message_packetcount > iba_msg_size) {
516
517                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now + Kernel.Tra
    ffic.packet_interarrival_time((int) Kernel.slLoad[localSL],Kernel.CV[Kernel.std_dev_pa
    cket_interarrival_time], this));
518                 } else {
519                     (new Event (this, PACKET_PRODUCED)).schedule (Kernel.Now + 10);
520                 }
521             }
522
523             // keep track of number of packets send in this msg
524             if (Kernel.IBA_ENABLE_MESSAGE) {
525                 if (DEBUG)
526                     System.out.println ("Increasing message_packetcount\n");
527                 iba_message_packetcount++;
528             }
529         }
530
531
532         public void packet_received_enter (EasterEgg egg) {
533             Packet p = null;
534             bound_receive_buffers--;
535             (new Event (this, BINDABLE)).schedule (Kernel.Now, bindable_flag);
536
537
538             if (!egg.flit.isKill () && egg.flit.packet.destination != this) {
539                 System.out.println (Kernel.time () + toString ()
540                                     + "Misrouted packet (SL " + egg.flit.packet.sl +
541                                     "): " + egg.flit.toString ());
542                 System.out.println ("   Src: " + egg.flit.packet.source + " " +
543                                     egg.flit.packet.source.hashCode ());
544                 System.out.println ("   Dst: " + egg.flit.packet.destination + " " +
545                                     egg.flit.packet.destination.hashCode ());
546                 System.out.println ("   Cur: " + this + " " + this.hashCode ());
547             }
548
549             if (egg.flit.packet.destination != this)
550                 System.out.println ("Wrong destination");
551             if (egg.flit.packet.payloadpointer != null
552                     && egg.flit.packet.payloadpointer.probe
553                     && egg.flit.packet.processorSL != localSL)
554                 System.out.println ("Wrong processorSL");
555             //handles the reception of probe packets
556             if (egg.flit.packet.payloadpointer != null
557                     && egg.flit.packet.payloadpointer.probe
558                     && egg.flit.packet.payloadpointer.probeSessionNum ==
559                     probeSessionExpected) {
560                 probeMin =
561                     Math.min ((Kernel.Now - egg.flit.packet.injection_time) / 10,
562                               probeMin);
563                 probeMax =
564                     Math.max ((Kernel.Now - egg.flit.packet.injection_time) / 10,
565                               probeMax);
566                 recProbe++;
567             }
568         }
569
570         private void send_pack (Packet pp) {
571             sendProbe = false;
572             pid += Kernel.max_switches;
573             if (send_q.full ()) {
574                 //internal send_q, not used
575                 //if(pp.payloadpointer!=null) enqueue(pp);
576                 Packet.packets_rejected[pp.sl]++;
577                 Packet.incRejected (pp.hops, pp.sl);
578                 probeDropped++;
579             }
580             send_q.insert (pp);
581             if (in_transmission < drop_connects.length) {
582                 send_enter (null);
583
```

```
584             }
585         }
586
587         private void enqueue (Packet p) {
588             retry.addElement (p);
589         }
590
591         //Remnant from BBRMSProcessor, not used here
592         private boolean admit_flow (Packet p) {
593             VLane outvl = drop_links[p.sl].vlanes[p.sl];
594             BBRMSSwitch s = (BBRMSSwitch) outvl.receiver;
595             if (outvl.link.bw - p.payloadpointer.peak >= 0) {//continue if admitted
596                 outvl.link.bw -= p.payloadpointer.peak;
597                 if (!s.next_admit_flow (p, outvl)) {
598                     outvl.link.bw += p.payloadpointer.peak;
599                     return false;
600                 }
601                 return true;
602             }
603             return false;
604         }
605         //Remnant from BBRMSProcessor, not used here
606         public boolean next_admit_flow (Packet p, VLane vl) {
607             if (p.destination != this)
608                 System.out.
609                 println
610                 ("!!!!Wrong destination, something is wrong with the admission-routing.\n
    "
611                  + p.destination.toString () + " is not " + toString () + "!");
612             else if (NEWDEBUG)
613                 System.out.println ("Correct destination");
614             return true;
615         }
616
617         //reset probe data
618         public void probeReset () {
619             probeLast = 0;
620             probeSessionExpected++;
621             recProbe = 0;
622             probeMax = 0;
623             probeMin = 100000000;
624         }
625
626         //returns result of probing.
627         public boolean probeDone (Packet p) {
628             System.out.println ("Max Probe Jitter: " + (probeMax - probeMin));
629             System.out.println ("Recieved probepackets: " + recProbe);
630             if (probeMax - probeMin > 0
631                     && probeMax - probeMin < p.payloadpointer.delay
632                     && recProbe >= probePacketsInit) {
633                 return true;
634             } else {
635                 return false;
636             }
637         }
638     }
639
```

```
1     /*
2      * BBRMSSwitch.java: BBRMSSwitch subclasses switch.  It is part of the link-by-link
3      * scheme. Uses the avaliable bandwidth for each SL calculated in HalfLink.
4      */
5
6     package base;
7     public class BBRMSSwitch extends Switch {
8         boolean DEBUGAD = false;
9
10        public BBRMSSwitch (int id) {
11            super (id,
12                   Kernel.CV[Kernel.Switch_routetime],
13                   Kernel.CV[Kernel.Switch_degree],
14                   Kernel.CV[Kernel.Node_drop_links]);
15        }
16
17        public BBRMSSwitch (int id, int routetime, int degree, int drop_links) {
18            super (id, routetime, degree, drop_links);
19        }
20
21
22
23
24        //The main AC routine for RMS. It checks capacity and forwards the request to the
      next switch.
25        public boolean next_admit_flow (Packet p, VLane last) {
26            VLane outvl = (VLane) Kernel.ExtendedRoute (last, p, p.source, this,
27                          p.destination).elementAt (0);
28
29            if (Kernel.SMS_ADMISSION_CONTROL) {
30                //used for RMS version with measured sum instread of static sum, not used
      in results.
31                int a = 0;
32                while (outvl.link.mean[p.sl][a] != 0 && a < outvl.link.history)
33                    a++;
34                if ((a == 0 && p.payloadpointer.peak < outvl.link.abwSL[p.sl])
35                        || 10 * (a + 1) / (Kernel.Now - outvl.link.mean[p.sl][a]) +
36                        p.payloadpointer.peak < outvl.link.abwSL[p.sl]) {
37                    return true;
38                } else {
39                    return false;
40                }
41            } else {
42                //admits flow if available bw here, and in every following switch.
43                //Kernel.slLoad
44                if (DEBUGAD)
45                    System.out.println ("Peakrate: " + p.payloadpointer.peak);
46                if (outvl.link.abwSL[p.sl] - p.payloadpointer.peak > 0) {
47                    outvl.link.abwSL[p.sl] -= p.payloadpointer.peak;
48                    if (!outvl.receiver.next_admit_flow (p, outvl)) {
49                        outvl.link.abwSL[p.sl] += p.payloadpointer.peak;
50                        if (DEBUGAD)
51                            System.out.println ("Rejected by a later switch");
52                        return false;
53                    }
54                    if (DEBUGAD)
55                        System.out.println ("Admitting: " + outvl.link.abwSL[p.sl]);
56                    return true;
57                }
58                if (DEBUGAD)
59                    System.out.println ("Rejected by this switch");
60                return false;
61            }
62            /*
63             //abwHP/LP
64             //Base AC on high-pri/low-pri available BW instead of each SL.
65             if(DEBUGAD)System.out.println("Peakrate: "+p.payloadpointer.peak);
66             if(outvl.ibaHighPri){
67             if(outvl.link.abwHP-p.payloadpointer.peak>0){
68             outvl.link.abwHP-=p.payloadpointer.peak;
69             if(!outvl.receiver.next_admit_flow(p,outvl)){
70             outvl.link.abwHP+=p.payloadpointer.peak;
71             if(DEBUGAD)System.out.println("Rejected by a later switch");
72             return false;
73             }
```

```
74                      if(DEBUGAD)System.out.println("Admitting: " + outvl.link.abwHP);
75                      return true;
76                      }
77                      if(DEBUGAD)System.out.println("Rejected by this switch");
78                      return false;
79                      }else {
80                      if(outvl.link.abwLP-p.payloadpointer.peak>0){
81                      outvl.link.abwLP-=p.payloadpointer.peak;
82                      if(!outvl.receiver.next_admit_flow(p,outvl)){
83                      outvl.link.abwLP+=p.payloadpointer.peak;
84                      if(DEBUGAD)System.out.println("Rejected by a later switch");
85                      return false;
86                      }
87                      if(DEBUGAD)System.out.println("Admitting: " + outvl.link.abwLP);
88                      return true;
89                      }
90                      if(DEBUGAD)System.out.println("Rejected by this switch");
91                      return false;
92                      }
93                      */
94              }
95      }
96
```

```
1   package base;
2
3   /*
4    PSource.java:  PSource is the class generating the pareto sequence,
5    use 10 for each flow in the simulator.
6    */
7   public class PSource {
8       int ontime = 0, offtime = 0;
9       Flow f;
10      Processor p;
11      int turnsToSend = 0;
12      int turnsToWait = 0;
13      boolean sending = false;
14
15      public PSource (Flow fl, Processor pr) {
16          f = fl;
17          p = pr;
18      }
19
20      //Returns the length of an on-period, called each time a packet is to be sent.
21      //returns turnsToSend only when current on and following off period is over, 0 ot
    herwise
22      public int update () {
23          if (sending && turnsToSend > 0) {
24              turnsToSend--;
25              return 0;
26          } else if (sending && turnsToSend == 0) {
27              sending = false;
28              turnsToWait =
29                  (int) ((p.pareto.getNext (f.alfa_off, f.b_off) + f.b_on) / f.b_on);
30              return 0;
31          } else if (!sending && turnsToWait > 0) {
32              turnsToWait--;
33              return 0;
34          } else if (!sending && turnsToWait == 0) {
35              sending = true;
36              turnsToSend =
37                  (int) ((p.pareto.getNext (f.alfa_on, f.b_on) + f.b_on) / f.b_on);
38              return turnsToSend;
39          }
40          return 0;
41      }
42
43
44      public int update_new () {
45          return (int) ((p.pareto.getNext (f.alfa_off, f.b_off) +
46                         f.b_on) / f.b_on) + (int) ((p.pareto.getNext (f.alfa_on,
47                                                     f.b_on) +
48                                                     f.b_on) / f.b_on);
49      }
50  }
51
```

```
1    package base;
2
3    import java.util.Random;
4    /*
5     * parGen.java: parGen Generates a pareto distributed number
6     */
7
8    public class parGen {
9        protected Random randNum;
10       private double tall;
11       public parGen (long seed) {
12           randNum = new Random (seed);
13       }
14
15
16       public double getNext (double a, double b) {
17           tall = b / Math.pow (1 - randNum.nextDouble (), 1.0 / a);
18           return (tall);
19       }
20   }
21
```

```
1    /*
2     * Flow.java: Flow is the class containing the pareto send statistics of a flow
3     */
4
5    package base;
6
7    public class Flow {
8        //int turnsToSend=0;
9        //int turnsToWait=0;
10       //boolean sending=false;
11       int sentPackets = 0;
12       public int flowNum;
13       int waitPackets = 0;
14       int recievedPackets = 0;
15       double ratio;
16       private boolean paretoInit = true;
17       private int initTime = 500000;
18       public int sendQueue;
19       public final int numPareto = 10;
20       PSource[] sources;
21       public double b_on, b_off = 1;
22       public double alfa_on = 1.5;
23       public double alfa_off = 1.5;
24
25       public Flow (Processor p) {
26           b_on = 5000;
27           double C_on = Math.pow (1.19 * alfa_on - 1.166, -0.027);
28           double C_off = Math.pow (1.19 * alfa_off - 1.166, -0.027);
29           double S = Math.pow (2, -53);
30           double T_on = (alfa_on - 1) / alfa_on;
31           double T_off = (alfa_off - 1) / alfa_off;
32           b_off = b_on * ((T_off / T_on) * ((1 - Math.pow (S, (T_on))) / (1 - Math.pow
     (S, T_off))) * ((1.0 / 0.05) - 1));
33           sources = new PSource[numPareto];
34           for (int g = 0; g < numPareto; g++) {
35               sources[g] = new PSource (this, p);
36           }
37       }
38
39
40       //This is the constructor normally used, it is the one working.
41       public Flow (Processor p, int fnum) {
42           flowNum = fnum;
43           if (Kernel.SELF_SIM_NEW)
44               b_on = (1 / p.flowBW) / 2;
45           else
46               b_on = 5000;
47           double C_on = Math.pow (1.19 * alfa_on - 1.166, -0.027);
48           double C_off = Math.pow (1.19 * alfa_off - 1.166, -0.027);
49           double S = Math.pow (2, -53);
50           double T_on = (alfa_on - 1) / alfa_on;
51           double T_off = (alfa_off - 1) / alfa_off;
52           //This works at least as long as T_on==T_off.
53           if (Kernel.SELF_SIM_NEW)
54               b_off = b_on * ((T_off / T_on) * ((1 - Math.pow (S, (T_on))) / (1 - Math.
     pow (S, T_off))) * ((1.0 / p.flowBW / 10) - 1));
55           else
56               b_off = b_on * ((T_off / T_on) * ((1 - Math.pow (S, (T_on))) / (1 - Math.
     pow (S, T_off))) * ((1.0 / 0.05) - 1));
57           sources = new PSource[numPareto];
58           for (int g = 0; g < numPareto; g++) {
59               sources[g] = new PSource (this, p);
60           }
61       }
62
63       public double report () {
64           ratio = (double) sentPackets / (sentPackets + waitPackets);
65           return ratio;
66       }
67
68
69       public boolean update_selfsim_flow (Processor p) {
70
71
72           if (!Kernel.SELF_SIMILAR && !Kernel.SELF_SIM_NEW) {
```

```
73                  //We are not using self-similar traffic, always send a packet.
74                  sentPackets++;
75                  return true;
76              }
77
78
79              //alternative way of generating self-similar traffic, generates new events ir
        regurarely instead of what is normal, not used.
80              if (Kernel.SELF_SIM_NEW) {
81                  if (paretoInit) {
82                      for (int h = 0; h < initTime; h++)
83                          for (int g = 0; g < numPareto; g++)
84                              sources[g].update ();
85                      paretoInit = false;
86                  }
87
88                  for (int g = 0; g < numPareto; g++) {
89                      EasterEgg egget = new EasterEgg ();
90                      egget.flyt = this;
91                      (new Event (p, Processor.FLOW_INTERVAL, egget)).schedule (Kernel.
92                              Now +
93                              Kernel.
94                              Cycle
95                              (sources
96                               [g].
97                               update_new
98                               ()));
99                  }
100                 return true;
101             } else {
102                 //normal way
103                 if (paretoInit) {
104                     //init the pareto sources of the flow
105                     for (int h = 0; h < initTime; h++)
106                         for (int g = 0; g < numPareto; g++)
107                             sources[g].update ();
108                     paretoInit = false;
109                 }
110
111                 //gather length of on period from PSources
112                 for (int g = 0; g < numPareto; g++) {
113                     sendQueue += sources[g].update ();
114
115                 }
116                 //debug
117                 //if (p.adctrlID == 0 && p.localSL == 0) System.out.println ("recievedPac
        kets proc " + p.adctrlID + " " + flowNum + ": " + recievedPackets);
118                 if (sendQueue > 0) {
119                     //We are allowed to send a packet
120                     sentPackets++;
121                     sendQueue--;
122                     //if (p.adctrlID == 0 && p.localSL == 0) System.out.println ("sentPac
        kets proc " + p.adctrlID + " " + flowNum + ": " + sentPackets);
123                     return true;
124                 } else {
125                     //may not send packet
126                     waitPackets++;
127                     //if (p.adctrlID == 0 && p.localSL == 0) System.out.println ("waitPac
        kets proc " + p.adctrlID + " " + flowNum + ": " + waitPackets);
128                     return false;
129                 }
130             }
131         }
132     }
133
```

```
1   /*
2    * Payload.java: Payload is a class for carrying AC specific data in network packets,
3    * like timestamps and AC requirements.
4    */
5   package base;
6   import java.util.Random;
7
8   public class Payload {
9       public SwitchList sList = new SwitchList ();
10      public long timestamp, probeSessionNum;
11      public double delay, peak;
12      public boolean resvinit = false, resvok = false, resvnotok =
13                                  false, resvteardown = false, probe = false;
14      public int sl;
15      public int packetNum = 0;
16      private static Random rand = new Random ();
17
18      public Payload (long ts) {
19          /*if(Kernel.MS_OUT_OF_SYNK)
20           timestamp=ts+(0.1*(rand.nextInt(2*Kernel.CV[Kernel.mean_packet_interarrival_
    time])-Kernel.CV[Kernel.mean_packet_interarrival_time]));
21           else
22           */
23          timestamp = ts;
24      }
25      public Payload (long ts, double p, double d) {
26          /*if(Kernel.MS_OUT_OF_SYNK)
27           timestamp=ts+(rand.nextInt(2*Kernel.CV[Kernel.mean_packet_interarrival_time]
    )-Kernel.CV[Kernel.mean_packet_interarrival_time]);
28           else
29           */
30          timestamp = ts;
31          delay = d;
32          peak = p;
33      }
34  }
35
```

```
1     package base;
2
3     import java.util.*;
4     import java.math.BigInteger;
5
6     /*
7      * HalfLink - A class implementing the basic link.  Subclassing this class
8      * will typically be used for gathering more statistics than already
9      * offered.
10     *
11     * @author I. Theiss 20010327
12     **/
13
14    public class HalfLink implements Schedulable {
15
16        public static final boolean DEBUG = false;
17        public static final boolean DEBUG2 = false;
18        public static final boolean DEBUG3 = false;
19        public static final boolean DEBUG4 = false;
20        public static final boolean DEBUG_FC = false;
21        //  public static final boolean IBA_FC_ENABLED = false;
22
23
24        //BBRMSSwitch uses this, calculated in VLane:
25        public double abwLP, abwHP, bw = Kernel.linkbw;
26        public double[] abwSL =
27            new double[Kernel.CV[Kernel.highpri_service_levels] +
28                       Kernel.CV[Kernel.lowpri_service_levels]];
29        public int history = 100;
30        public long mean[][] =
31            new long[Kernel.CV[Kernel.highpri_service_levels] +
32                     Kernel.CV[Kernel.lowpri_service_levels]][history];
33        private boolean debugprinted = false;
34
35
36
37        public BiLink bilink;
38        public HalfLink opposite = null;
39        public VLane vlanes[];
40        public VLane ctrl_vlane;
41        public int vlinks;
42        public int length;            // cycles for one flit to cross the cable
43        public Node transmitter, receiver;
44        // Hack! Used when called from Tor's C++ simulator to create
45        // updown routes
46        public int number = -1;
47        //   public VLane token = null;
48        // High pri VLane Limit of High Priority
49        public int ibaLHP;
50        // High priority counter, how much high pri data has been sent since
51        // scheduled reduced for each flit sent on hp vlane, reset when
52        // sending on lp
53        public int ibaHPC;
54        // number of higpri/lowpri vlanes, must be read from routingtable
55        public int nHighPriVLanes =
56            Kernel.CV[Kernel.highpri_service_levels] *
57            Kernel.CV[Kernel.vlinks_per_sl];
58        public int nLowPriVLanes =
59            Kernel.CV[Kernel.lowpri_service_levels] * Kernel.CV[Kernel.vlinks_per_sl];
60        // split VLanes in groups of high and low priority
61        public VLane highpri_token[];
62        public VLane lowpri_token[];
63        public VLane token;
64        // VLane currently active
65        public int curHighPriVLane = 0;
66        public int curLowPriVLane = 0;
67        //public VLane highpri_token = null;
68        //public VLane lowpri_token = null;
69        boolean ibaHPActive = true;
70        int ibaLPFlitCount = 0;
71        public int flits_waiting = 0;
72        public int direction;
73
74        public HalfLink (BiLink bilink,
75                         int length,
```

```
76                         int vlinks,
77                         int rx_buffer_size,
78                         int tx_buffer_size,
79                         Node transmitter, Node receiver, int direction) {
80
81         this.bilink = bilink;
82         this.length = length;
83         if (length < 1) {
84             System.out.println ("Fatal error: links cannot have zero length");
85             System.exit (0);
86         }
87         this.vlinks = vlinks;
88         this.vlanes = new VLane[vlinks];
89         this.transmitter = transmitter;
90         this.receiver = receiver;
91         this.direction = direction;
92
93         // iba arbitration variables initialisation
94         ibaLHP = Kernel.CV[Kernel.iba_limit_of_highpri];
95         //System.out.println("LHP is: " +ibaLHP);
96         //Kolaf: Should this be 4000 * ibaLHP?
97         // SAR: nope, we set this directly in the cfg file
98         ibaHPC = ibaLHP;
99
100        // token initialisation
101        highpri_token = new VLane[nHighPriVLanes];
102        lowpri_token = new VLane[nLowPriVLanes];
103
104        for (int i = 0; i < nHighPriVLanes; i++)
105            highpri_token[i] = null;
106        for (int i = 0; i < nLowPriVLanes; i++)
107            lowpri_token[i] = null;
108
109        //      System.out.println("Number of vlinks: "+vlinks);
110        for (int i = 0; i < vlinks; i++)
111            this.vlanes[i] = createVLane (this, transmitter, receiver,
112                                          direction,
113                                          tx_buffer_size, rx_buffer_size, i);
114
115        this.ctrl_vlane = createVLane (this, transmitter, receiver,
116                                       direction,
117                                       tx_buffer_size, rx_buffer_size, vlinks);
118
119    }
120
121    public VLane createVLane (HalfLink hl, Node transmitter, Node receiver,
122                              int direction,
123                              int tx_buffer_size, int rx_buffer_size, int layer) {
124        return new VLane (hl, transmitter, receiver,
125                          direction, tx_buffer_size, rx_buffer_size, layer);
126    }
127
128
129    public void dump () {
130        System.out.println (toString ());
131        System.out.println ("  highpri flits_waiting: " + flits_waiting);
132        for (int i = 0; i < nHighPriVLanes; i++) {
133            if (highpri_token[i] != null) {
134                VLane t = highpri_token[i];
135                do {
136                    t.tx_buffer.dump ();
137                    t = t.rrrnext;
138                } while (t != highpri_token[i]);
139            }
140        }
141        System.out.println ("  lowpri flits_waiting: " + flits_waiting);
142        for (int i = 0; i < nLowPriVLanes; i++) {
143            if (lowpri_token[i] != null) {
144                VLane t = lowpri_token[i];
145                do {
146                    t.tx_buffer.dump ();
147                    t = t.rrrnext;
148                } while (t != lowpri_token[i]);
149            }
150        }
```

```
151
152             System.out.println ("  Events:");
153             for (int i = 0; i < scheduled_events.size (); i++) {
154                 Event e = (Event) (scheduled_events.elementAt (i));
155                 System.out.println ("    FLIT_AT_ARBITRATOR (10): "
156                                     + e.dispatcher + " time: " + e.time);
157             }
158         }
159
160
161
162         public String toString () {
163             if (transmitter == null && receiver == null)
164                 return "HalfLink [null,null]";
165             else if (transmitter == null)
166                 return "HalfLink  [null," + receiver.id + "]";
167             else if (receiver == null)
168                 return "HalfLink  [" + transmitter.id + ",null]";
169             else
170                 return "HalfLink  [" + transmitter.id + "," + receiver.id + "]";
171         }
172
173         // update hpc per flit
174         public void updateHPC () {
175             if (ibaLHP < 255 && ibaHPC > 0)
176                 ibaHPC--;
177         }
178
179         // iba update High Priority Counter (HPC)
180         //     public void updateHPC(Packet p)
181         //     {
182         //      //BigInteger threshold = new BigInteger("255");
183
184         //      if (ibaLHP != 255)
185         //          ibaHPC -= Math.ceil(p.size / 4);
186
187         //      if (ibaHPC <= 0 && lowpriWaiting() == null)
188         //           resetHPC();
189
190         //      //System.out.println("*** Updated HPC = " + ibaHPC);
191         //     }
192
193         // iba update High Priority Counter (HPC)
194         public void resetHPC () {
195             ibaHPC = ibaLHP;
196
197             //System.out.println("*** Reset HPC = " + ibaHPC);
198         }
199
200         // number of lowpri filts waiting
201         public VLane lowpriWaiting () {
202             int lowpriwaiting = 0;
203
204             for (int i = curLowPriVLane; i < nLowPriVLanes; i++) {
205                 if (lowpri_token[i] != null) {
206                     VLane t = lowpri_token[i];
207
208                     do {
209                         if (t.tx_buffer.flits_to_transfer () && t.ibaWeightCount > 0) {
210                             lowpriwaiting++;
211                             curLowPriVLane = i;
212                             return t;
213                         }
214                         t = t.rrrnext;
215                     } while (t != lowpri_token[i]);
216                 }
217             }
218
219             // if we are here no lowpri was found
220             // start on top next time and reset weight
221             curLowPriVLane = 0;
222             resetLowPriVLWeight ();
223             return null;
224         }
225
```

```
226        // number of highpri filts waiting
227        public VLane highpriWaiting () {
228            int highpriwaiting = 0;
229
230            for (int i = curHighPriVLane; i < nHighPriVLanes; i++) {
231                if (highpri_token[i] != null) {
232                    VLane t = highpri_token[i];
233
234                    do {
235                        //              if (t.ibaWeightCount <= 0)
236                        //                  System.out.println("*** " + t);
237                        if (t.tx_buffer.flits_to_transfer () && t.ibaWeightCount > 0) {
238                            highpriwaiting++;
239                            curHighPriVLane = i;
240                            return t;
241                        }
242                        t = t.rrrnext;
243                    } while (t != highpri_token[i]);
244                }
245            }
246
247            // if we are here no highpri was found
248            // start on top next time and reset weight
249            curHighPriVLane = 0;
250            resetHighPriVLWeight ();
251            return null;
252        }
253
254
255        // iba reset lowpri vl weight counter
256        public void resetLowPriVLWeight () {
257            for (int i = 0; i < nLowPriVLanes; i++) {
258                if (lowpri_token[i] != null)
259                    lowpri_token[i].ibaWeightCount = lowpri_token[i].ibaWeight;
260            }
261        }
262
263        // iba reset highpri vl weight counter
264        public void resetHighPriVLWeight () {
265            for (int i = 0; i < nHighPriVLanes; i++) {
266                if (highpri_token[i] != null)
267                    highpri_token[i].ibaWeightCount = highpri_token[i].ibaWeight;
268            }
269        }
270
271        // the arbitrator state procedures
272
273        public int direction () {
274            return direction (0, vlinks);
275        }
276
277        public int direction (int start, int layers) {
278            int dir = vlanes[start].direction;
279            for (int i = start + 1; i < start + layers; i++) {
280                if (dir != vlanes[i].direction)
281                    return BiLink.UNDEFINED;
282            }
283            return dir;
284        }
285
286        public void set_direction (int dir) {
287
288            set_direction (0, vlinks, dir);
289
290        }
291
292        public void set_direction (int start, int layers, int dir) {
293
294            if (start == 0 && layers == vlinks) {
295
296                direction = dir;
297                opposite.direction = dir * (-1);
298
299                if (DEBUG4) {
300                    String s;
```

```
301                     if (dir == BiLink.UNDEFINED)
302                         s = "UNDEFINED";
303                     else if (dir == BiLink.UP)
304                         s = "UP";
305                     else
306                         s = "DOWN";
307                     System.out.println (toString () + " set to " + s);
308                 }
309             }
310 
311             for (int i = start; i < start + layers; i++) {
312                 vlanes[i].direction = dir;
313                 opposite.vlanes[i].direction = dir * (-1);
314             }
315 
316         }
317 
318         public void set_half_direction (int dir) {
319 
320             set_half_direction (0, vlinks, dir);
321 
322         }
323 
324         public void set_half_direction (int start, int layers, int dir) {
325 
326             if (start == 0 && layers == vlinks) {
327                 if (DEBUG4) {
328                     String s;
329                     if (dir == BiLink.UNDEFINED)
330                         s = "UNDEFINED";
331                     else if (dir == BiLink.UP)
332                         s = "UP";
333                     else
334                         s = "DOWN";
335                     System.out.println (toString () + " set to " + s);
336                 }
337                 direction = dir;
338             }
339 
340             for (int i = start; i < start + layers; i++) {
341                 vlanes[i].direction = dir;
342             }
343 
344         }
345 
346         public void register_transmission (VLane vl) {
347             //     if (vl == ctrl_vlane) return;
348             //     if (token == null) {
349             //       token = vl;
350             //       token.rrrnext = token;
351             //     } else {
352             //       vl.rrrnext = token.rrrnext;
353             //       token.rrrnext = vl;
354             //       token = vl;
355             //     }
356 
357             // IBA link arb.
358             // if VL is highpri then put it in highpri_token otherwise in lowpri_token
359             //System.out.println("VL.layer = " + vl.layer + " highpri = " + vl.ibaHighPri
    );
360             if (vl == ctrl_vlane)
361                 return;
362             if (vl.ibaHighPri) {
363                 //System.out.println("highp registered " + vl.layer);
364                 //      dump();
365                 if (highpri_token[vl.layer] == null) {
366                     highpri_token[vl.layer] = vl;
367                     highpri_token[vl.layer].rrrnext = highpri_token[vl.layer];
368                 } else {
369                     vl.rrrnext = highpri_token[vl.layer].rrrnext;
370                     highpri_token[vl.layer].rrrnext = vl;
371                     highpri_token[vl.layer] = vl;
372                 }
373             } else {
374                 //System.out.println("lowp registered " + vl.layer);
```

```
375                 //      dump();
376                 if (lowpri_token[vl.layer - nHighPriVLanes] == null) {
377                     lowpri_token[vl.layer - nHighPriVLanes] = vl;
378                     lowpri_token[vl.layer - nHighPriVLanes].rrrnext =
379                         lowpri_token[vl.layer - nHighPriVLanes];
380                 } else {
381                     vl.rrrnext = lowpri_token[vl.layer - nHighPriVLanes].rrrnext;
382                     lowpri_token[vl.layer - nHighPriVLanes].rrrnext = vl;
383                     lowpri_token[vl.layer - nHighPriVLanes] = vl;
384                 }
385             }
386 
387         }
388 
389 
390         public void flit_at_arbitrator_enter (EasterEgg egg) {
391 
392             // this is not really a round robin, since the order of chosen
393             // vlanes is kinda random and...
394 
395             // the control lanes are always prioritized
396 
397             boolean yowsa = false;
398             if (DEBUG3) {
399                 System.out.println (toString () + " popping");
400                 dump ();
401                 System.out.println (toString () + " end popping");
402                 if (transmitter.id == 12 && receiver.id == 5 && Kernel.now >= 459125) {
403                     System.out.println (toString () + " popping");
404                     dump ();
405                     System.out.println (toString () + " end popping");
406                     yowsa = true;
407                 }
408             }
409 
410             // do a logical error check if enabled
411             //      if (Kernel.LOGICAL_ERROR_CHECK && transmitter.id == 12 && receiver.id
    == 5) {
412             if (Kernel.LOGICAL_ERROR_CHECK) {
413                 int testing = flits_waiting;
414                 if (ctrl_vlane.tx_buffer.flits_to_transfer ()) {
415                     testing -= ctrl_vlane.tx_buffer.flits_contained;
416                 }
417 
418                 for (int i = 0; i < nHighPriVLanes; i++) {
419                     if (highpri_token[i] != null) {
420                         VLane t = highpri_token[i];
421 
422                         do {
423                             if (t.tx_buffer.flits_to_transfer ())
424                                 testing -= t.tx_buffer.flits_contained;
425                             t = t.rrrnext;
426                         } while (t != highpri_token[i]);
427                     }
428                 }
429 
430                 for (int i = 0; i < nLowPriVLanes; i++) {
431                     if (lowpri_token[i] != null) {
432                         VLane t = lowpri_token[i];
433 
434                         do {
435                             if (t.tx_buffer.flits_to_transfer ())
436                                 testing -= t.tx_buffer.flits_contained;
437                             t = t.rrrnext;
438                         } while (t != lowpri_token[i]);
439                     }
440                 }
441 
442                 if (testing != 0)
443                     System.out.println (Kernel.time () + toString () + "logical error: "
444                                         + " flits_waiting doesn't seem to match actual "
445                                         + "number of flits waiting:" + testing);
446 
447             }
448 
```

```
449
450             // pick active table
451             //      if (highpriWaiting() != null) {
452             //          ibaHPActive = true;
453             //      } else {
454             //          ibaHPActive = false;
455             //      }
456
457
458             // pick active table
459             token = null;
460             if (ibaHPC > 0) {
461                 if (highpriWaiting () != null) {
462                     updateHPC ();
463                     token = highpriWaiting ();
464                     token.ibaWeightCount--;
465                     //              if (token.uID == 16)
466                     //                  System.out.println("ibaWeightCount = " + token.ib
        aWeightCount);
467                 } else {
468                     //resetHPC();
469                     token = lowpriWaiting ();
470                     if (token != null) {
471                         token.ibaWeightCount--;
472                         //System.out.println("ibaWeightCount = " + token.ibaWeightCount);
473                     }
474                 }
475             } else {
476                 if (lowpriWaiting () != null) {
477                     resetHPC ();
478                     token = lowpriWaiting ();
479                     token.ibaWeightCount--;
480                 } else {
481                     resetHPC ();
482                     updateHPC ();
483                     token = highpriWaiting ();
484                     if (token != null)
485                         token.ibaWeightCount--;
486                 }
487             }
488
489             //      if (highpriWaiting() != null && lowpriWaiting() == null && ibaHPC <=
        0) {
490             //          // no credit for hp, but no lp waiting so we send a hp
491             //          // without reseting hpc
492             //          token = highpriWaiting();
493             // //       if (token == null)
494             // //           System.out.println("hp ative active on null");
495             //      } else if (lowpriWaiting() != null && highpriWaiting() == null) {
496             //          token = lowpriWaiting();
497             //          resetHPC();
498             //      } else if (lowpriWaiting() != null && highpriWaiting() != null && iba
        HPC <= 0) {
499             //          token = lowpriWaiting();
500             //          resetHPC();
501             //      } else if (highpriWaiting() != null && ibaHPC > 0) {
502             //          token = highpriWaiting();
503             //          updateHPC();
504             //      }
505
506             //      if (highpriWaiting() != null) {
507             //          token = highpriWaiting();
508             //      } else {
509             //          token = lowpriWaiting();
510             //      }
511
512
513
514             // handle ctrl flits first, then highpri or finally lowpri
515             if (ctrl_vlane.tx_buffer.flits_to_transfer ()) {
516                 if (yowsa)
517                     System.out.println ("ctrl_vlane");
518                 ctrl_vlane.tx_buffer.set_next_in_transfer ();
519                 (new Event (ctrl_vlane, VLane.FLIT_TRANSMITTED)).schedule (Kernel.
520                         Now +
```

```
521                     Kernel.
522                     Cycle
523                     (length));
524             flits_waiting--;
525             if (DEBUG3) {
526                 if (transmitter.id == 21 && receiver.id == 11)
527                     System.out.println (Kernel.time () + toString () + " lowering"
528                                         + " flits_waiting (" + flits_waiting + ")");
529             }
530 
531             if (Kernel.LOGICAL_ERROR_CHECK) {
532                 if (flits_waiting < 0)
533                     System.out.println (Kernel.time () + toString () +
534                                         " logical error:" +
535                                         " flit_at_arbitrator, flits_waiting below "
536                                         + "zero on ctrl link");
537             }
538             if (DEBUG)
539                 System.out.println (toString () + " ctrl flit");
540         } else if (token != null) {
541             token.tx_buffer.set_next_in_transfer ();
542             (new Event (token, VLane.FLIT_TRANSMITTED)).schedule (Kernel.Now +
543                     Kernel.
544                     Cycle (length));
545             flits_waiting--;
546         }
547 
548         if (flits_waiting > 0) {
549             self_signal_flit_at_arbitrator_enter (null);
550         }
551     }
552 
553     public void self_signal_flit_at_arbitrator_enter (EasterEgg egg) {
554         (new Event (this, FLIT_AT_ARBITRATOR)).schedule (Kernel.Now +
555                 Kernel.Cycle (1),
556                 flit_at_arbitrator_flag);
557     }
558 
559     // event implementations, remember to update the dispatcher function
560     // if adding more events!!
561 
562     public static final int FLIT_AT_ARBITRATOR = 10;
563     public Flag flit_at_arbitrator_flag = new Flag (false, Kernel.Edge);
564 
565     public void flit_at_arbitrator (EasterEgg egg) {
566         flit_at_arbitrator_enter (egg);
567     }
568 
569     // common for all HalfLink events
570 
571     public void dispatcher (int dispatcher, EasterEgg egg) {
572 
573         switch (dispatcher) {
574         case FLIT_AT_ARBITRATOR:
575             flit_at_arbitrator (egg);
576             break;
577         default:
578             System.out.println ("HalfLink: no such event " + dispatcher);
579 
580         }
581     }                               // end dispatcherEvent
582 
583     public void purgeEvents () {
584         while (scheduled_events.size () > 0) {
585             Event e = (Event) (scheduled_events.elementAt (0));
586             scheduled_events.removeElementAt (0);
587             Kernel.globalHeap.removeEvent (e);
588             e.dismiss ();
589         }
590     }
591 
592     public Vector scheduled_events = new Vector ();
593     public Vector scheduled_events () {
594         return scheduled_events;
595     }
```

```
596
597      public void gatherStats (EasterEgg egg) {
598          for (int s = 0;
599                  s <
600                  Kernel.CV[Kernel.highpri_service_levels] +
601                  Kernel.CV[Kernel.lowpri_service_levels]; s++) {
602              for (int h = history - 2; h >= 0; h--) {
603                  mean[s][h + 1] = mean[s][h];
604              }
605          }
606          if (egg.vl.rx_buffer.top () != null) {
607              mean[((Flit) (egg.vl.rx_buffer.top ())).packet.sl][0] = Kernel.Now;
608
609          }
610      }
611
612  }                                              // end class HalfLink
613
```

```
1   package base;
2
3   import java.util.Vector;
4   import java.math.BigInteger;
5   /*
6    * VLane - A class implementing the basic virtual lane. Subclassing this
7    * class will typically be used for gathering more statistics than
8    * already offered.
9    *
10   * @author I. Theiss 20010329
11   *
12   * Constructor is modified by Frank Olaf Sem-Jacobsen to calculate the available
13   * bandwidth of the VL based on VL weight (commented out) and offered load on
14   * corresponding SL.
15   *
16   **/
17
18  public class VLane extends Connectable implements Schedulable {
19
20      public static final boolean DEBUG = false;
21      public static final boolean DEBUG2 = false;
22      public static final boolean DEBUG3 = false;
23      public static final boolean DEBUG_FC = false;
24
25      public static int uniqueID = 0;
26      public int uID = (-1);
27
28      public boolean NU_wait_rx = false;   // used only by the nodes!
29      public boolean NU_wait_tx = false;   //      -"-
30      public boolean NU_header_waiting = false;    //      -"-
31      public Flit NU_flit_switched = null;         //      -"-
32      public Vector NU_alternative_nexts; //      -"-
33
34      public HalfLink link;
35      public Node transmitter, receiver;
36      public int layer;
37      public int restrictions = 0;
38      public boolean rx_locked = false;
39      //admission control stuff, BBRMSSwitch.
40
41      //RMS admission control
42      int availableWeightLowPri = 64 * 255;
43      int availableWeightHighPri = 64 * 255;
44      final int maxWeightLowPri = 64 * 255, maxWeightHighPri = 64 * 255;
45
46
47      // IBA stuff
48      // is flowcontrol initialised
49      public boolean IBA_FC_UNINITIALISED = true;
50      // VLane priority, can be high or low)
51      public boolean ibaHighPri = false;
52      // VLane service level
53      public int sl;
54      // VLane weight, number of 64 byte blocks allowed to send when scheduled
55      public int ibaWeight = 64;
56      public int ibaWeightCount = ibaWeight;
57      public static int nVLperSL = Kernel.CV[Kernel.vlinks_per_sl];
58      public static int nSL = Kernel.CV[Kernel.highpri_service_levels]
59                              + Kernel.CV[Kernel.lowpri_service_levels];
60      public static int weightIndex = 0;
61
62      // delay between flowcontrol packet events in cycles
63      int ibaFC_TX_DELAY = 500;
64      int ibaFC_RX_DELAY = 300;
65
66      public int direction = BiLink.UNDEFINED;
67
68      public int direction () {
69          return direction;
70      }
71
72      public txBuffer tx_buffer;
73      public rxBuffer rx_buffer;
74
75      public Connectable previous;
```

```
76
77      public VLane rrrnext = null;
78
79      public String toString () {
80          String s;
81          if (link.ctrl_vlane == this)
82              s = " " + uID + " ctrl";
83          else
84              s = " " + uID + " " + layer;
85          if (transmitter == null && receiver == null)
86              return "VLane" + s + "[null,null]";
87          else if (transmitter == null)
88              return "VLane" + s + " [null," + receiver.id + "]";
89          else if (receiver == null)
90              return "VLane" + s + " [" + transmitter.id + ",null]";
91          else
92              return "VLane" + s + " [" + transmitter.id + "," + receiver.id + "]";
93      }
94
95      public VLane (HalfLink link, Node transmitter, Node receiver, int direction,
96                    int tx_buffer_size, int rx_buffer_size, int layer) {
97          this.link = link;
98          this.transmitter = transmitter;
99          this.receiver = receiver;
100         this.direction = direction;
101         this.layer = layer;
102         this.uID = VLane.uniqueID;
103         VLane.uniqueID++;
104
105         tx_buffer = new txBuffer (this, tx_buffer_size);
106         rx_buffer = new rxBuffer (this, rx_buffer_size);
107
108         // highpri vlanes are always lower layers, default is only one
109         // highpri SL
110         if (layer <
111                 (Kernel.CV[Kernel.vlinks_per_sl] *
112                  Kernel.CV[Kernel.highpri_service_levels])) {
113             //System.out.println("set highpri " + layer);
114             this.ibaHighPri = true;
115         }
116
117         // use the same weightIndex for all VLs in an SL
118         weightIndex = layer / nVLperSL;
119
120         //for RMSSwitch
121         if (this.ibaHighPri)
122             this.availableWeightHighPri -= this.ibaWeight;
123         else
124             this.availableWeightLowPri -= this.ibaWeight;
125
126         // skip control vlanes
127         if (layer != this.link.vlinks) {
128             this.ibaWeight = Kernel.vlWeight[weightIndex];
129             this.ibaWeightCount = this.ibaWeight;
130         }
131         //System.out.println("Layer " + layer + "  " + this.toString() + " " + this.i
    baWeightCount + " vlWeight = " + Kernel.vlWeight[layer]);
132         //Trenger ikke ta hensyn til nSLperVL dersom alle VL'ene til en SL har samme
    vekt. nSLperVL vil forsvinne fra abwSL uttrykket.
133         //BBRMSSwitch uses this:
134
135         /* //Used for available bw sl.
136          int totw=0;
137
138          int LHP = Kernel.CV[Kernel.iba_limit_of_highpri];
139          double tbw = link.bw;
140          double abw, abwSL;
141          if(LHP>=0 && LHP <= 1) abw=0.5*tbw;
142          else{
143          abw=tbw/(LHP+1);
144          }
145          if(this.ibaHighPri){
146          abw=tbw-abw;//+p.payloadpointer.peak;
147          link.abwHP=abw;
148          if(abw==0) abw+=0.00000001;
```

```
149                for(int v=0;v<Kernel.CV[Kernel.highpri_service_levels];v++){
150                totw+=Kernel.vlWeight[v];
151                }
152                abwSL=abw*this.ibaWeight/totw;
153                //System.out.println("highpri_sl_" + p.sl+" = " + abwSL);
154                }else{
155                link.abwLP=abw;
156                if(abw==0) abw+=0.00000001;
157                for(int v=Kernel.CV[Kernel.highpri_service_levels];v<Kernel.CV[Kernel.lowpri
_service_levels]+Kernel.CV[Kernel.highpri_service_levels];v++){
158                totw+=Kernel.vlWeight[v];
159                }
160                abwSL=abw*this.ibaWeight/totw;
161                //System.out.println("lowpri_sl_" + p.sl+" = " + abwSL);
162                }
163                if(weightIndex<nSL)
164                link.abwSL[weightIndex]=abwSL;
165                //System.out.println("abwSL for SL " + weightIndex +" is " + abwSL);
166                */
167
168            //use kernel.slload
169
170            int totalload = 0;
171            for (int g = 0; g < nSL - 1; g++) {
172                totalload += Kernel.slLoad[g];
173            }
174            if (weightIndex < nSL) {
175                link.abwSL[weightIndex] =
176                    link.bw * Kernel.slLoad[weightIndex] / totalload;
177            }
178
179        }
180
181        // iba credit check, returns true if credits are available
182        public boolean credits_available () {
183            // using BigInteger to get unsigned modulo arithmetic
184            BigInteger CL, CR, modulo, creditValue, threshold;
185
186            // if disabled we always have iba credits
187            if (!Kernel.IBA_ENABLE_FLOW_CONTROL) {
188                //System.out.println("no fc");
189                return true;
190            }
191            //     if (this.uID == 0)
192            //       System.out.println(toString() + " AvailableBlocks " + rx_buffer.avai
lableBlocks);
193
194            // quick hack to get packet size
195            long psize = Kernel.CV[1] + 2;
196            threshold = new BigInteger ("2048");
197            modulo = new BigInteger ("4096");
198            CL = new BigInteger ("" + rx_buffer.ibaFCCL);
199            CR = new BigInteger ("" + (tx_buffer.ibaFCTBS + (int)
200                                       Math.ceil (psize / 64.0)) % 4096);
201
202            creditValue = CL.subtract (CR).mod (modulo);
203            if (creditValue.compareTo (threshold) <= 0)
204                return true;
205
206            return false;
207        }
208
209        // iba send flowcontrol packet to receiver
210        public void send_flowcontrol_rx_enter (EasterEgg egg) {
211            // we fake flowcontrol by directly updating the rx and tx buffers
212            // no packets are sent/received
213
214            if (DEBUG_FC) {
215                System.out.println ("Update flowcontrol rx: " + toString ());
216                System.out.println ("       FCTBS = " + tx_buffer.ibaFCTBS);
217                System.out.print ("  Old ABR   = " + rx_buffer.ibaABR);
218            }
219            // fake sending of flowctrl by updating receive buffer ABR
220            rx_buffer.incABR (0, 1);
221            if (DEBUG_FC)
```

```
222                 System.out.println ("  New ABR   = " + rx_buffer.ibaABR);
223
224             // schedule the next flowcontrol packet
225             (new Event (this, VLane.SEND_FLOWCONTROL_RX)).schedule (Kernel.Now +
226                     Kernel.
227                     Cycle
228                     (ibaFC_RX_DELAY));
229         }
230
231         // iba send flowcontrol packet to transmitter
232         public void send_flowcontrol_tx_enter (EasterEgg egg) {
233             // we fake flowcontrol by directly updating the rx and tx buffers
234             // no packets are sent/received
235             if (DEBUG_FC)
236                 System.out.println ("Update flowcontrol tx: " + toString ());
237             // before transmission of flowctrl packet update FCCL
238             rx_buffer.incFCCL ();
239             // fake sending of flwoctrl by updating transmit buffer Last Known FCCL
240             if (DEBUG_FC)
241                 System.out.print ("  Old FCCL  = " + tx_buffer.ibaLastFCCL);
242             tx_buffer.ibaLastFCCL = rx_buffer.ibaFCCL;
243             if (DEBUG_FC)
244                 System.out.println ("  New FCCL  = " + tx_buffer.ibaLastFCCL);
245
246             // schedule the next flowcontrol packet
247             (new Event (this, VLane.SEND_FLOWCONTROL_TX)).schedule (Kernel.Now +
248                     Kernel.
249                     Cycle
250                     (ibaFC_TX_DELAY));
251         }
252
253
254         // the tranceiver state procedures, see the process model vlane_trx
255
256         public void flit_in_transmitter_enter (EasterEgg egg) {
257             if (DEBUG)
258                 System.out.println (toString () + " flit_in_transmitter_enter");
259             tx_buffer.set_next_ready_to_transfer ();
260             link.flits_waiting++;
261             if (DEBUG3) {
262                 if (transmitter.id == 21 && receiver.id == 11)
263                     System.out.println (Kernel.time () + toString () + " increasing"
264                                         + " flits_waiting (" + link.flits_waiting +
265                                         ") for flit " +
266                                         tx_buffer.latest_inserted ().toString ());
267             }
268             if (rrrnext == null) {
269                 //     if (layer > 1) {
270                 //         System.out.print("vl " + layer);
271                 //         System.out.print(" " +tx_buffer.latest_inserted().toString())
    ;
272                 //     }
273                 link.register_transmission (this);
274             }
275
276             (new Event (link, HalfLink.FLIT_AT_ARBITRATOR)).schedule (Kernel.Now,
277                     link.
278                     flit_at_arbitrator_flag);
279
280             if (Kernel.TRACE_VLANE_ACCESS) {
281                 if (uID == Kernel.VLANE_TRACE_UID) {
282                     Flit f = tx_buffer.latest_inserted ();
283                     if (f != null && f.packet != null)
284                         System.out.println (Kernel.time () + toString ()
285                                             + " flit " + f.toString ()
286                                             + " in transmitter "
287                                             + Kernel.VLANE_TRACE_UID);
288                 }
289
290             }
291
292             // schedule initial flowcontrol events
293             if (IBA_FC_UNINITIALISED && Kernel.IBA_ENABLE_FLOW_CONTROL) {
294                 IBA_FC_UNINITIALISED = false;
295                 (new Event (this, VLane.SEND_FLOWCONTROL_TX)).schedule (Kernel.Now +
```

```
296                         Kernel.
297                         Cycle
298                         (ibaFC_TX_DELAY));
299                 (new Event (this, VLane.SEND_FLOWCONTROL_RX)).schedule (Kernel.Now +
300                         Kernel.
301                         Cycle
302                         (ibaFC_RX_DELAY));
303             }
304 
305             if (Kernel.TRACE_PACKET) {
306                 Flit f = tx_buffer.latest_inserted ();
307                 //      if (f != null) System.out.println("trace test " + f.packet.pid +
     " : " + Kernel.PACKET_TRACED);
308                 if (f != null && f.packet != null) {
309                     if (f.packet.pid == Kernel.PACKET_TRACED
310                             //       && uID == 136) {
311                             && (f.isHeader () || f.isTail ())) {
312                         System.out.println (Kernel.time () + toString () +
313                                             " fite: " + f.toString ());
314                         tx_buffer.dump ();
315                         rx_buffer.dump ();
316                         if (transmitter.id < Kernel.max_switches && previous != null) {
317                             if (((VLane) previous).NU_flit_switched != null)
318                                 System.out.println (" another coming: "
319                                                     +
320                                                     ((VLane) previous).NU_flit_switched.
321                                                     toString ());
322                             else if (NU_wait_tx)
323                                 System.out.
324                                 println
325                                 (" another coming, reserved rx, waiting for tx: ");
326                         }
327 
328                         link.dump ();
329                         System.out.println (Kernel.time () + toString ()
330                                             + " end fite: " + f.toString ());
331                     }
332                 }
333             }
334         }
335 
336         // on event FLIT_TRANSMITTED scheduled by HalfLink (arbitrator)
337 
338         public void flit_transmitted_enter (EasterEgg egg) {
339             if (DEBUG)
340                 System.out.println (toString () + " flit_transmitted_enter");
341             // a possible optimization could be to only schedule if
342             // the tx_buffer was full before this clearance
343             if (rx_buffer.flits_contained == rx_buffer.size || rx_locked)
344                 return;
345 
346             if (Kernel.TRACE_VLANE_ACCESS) {
347                 if (uID == Kernel.VLANE_TRACE_UID) {
348                     Flit f = tx_buffer.top ();
349                     if (f != null && f.packet != null)
350                         System.out.println (Kernel.time () + toString ()
351                                             + " flit " + f.toString ()
352                                             + " transmitted " + Kernel.VLANE_TRACE_UID);
353                 }
354 
355             }
356 
357             if (Kernel.TRACE_PACKET) {
358                 Flit f = tx_buffer.top ();
359                 if (f.packet != null && f.packet.pid == Kernel.PACKET_TRACED
360                         //      && uID == 93) {
361                         && (f.isHeader () || f.isTail ())) {
362                     System.out.println (Kernel.time () + toString ()
363                                         + " fte: " + f.toString ());
364                     System.out.println ("rx_buffer considered empty: " +
365                                         rx_buffer.empty ());
366                     System.out.println ("rx_buffer.flits_contained: " +
367                                         rx_buffer.flits_contained);
368                     System.out.println ("rx_buffer.reserved_slots: " +
369                                         rx_buffer.reserved_slots);
```

```
370                     //      System.out.println("rx_buffer.top(): " + rx_buffer.top().toSt
ring());
371
372                 }
373             }
374             if (NU_wait_tx) {
375                 EasterEgg TA_egg = new EasterEgg ();
376                 TA_egg.vl = this;
377                 (new Event (transmitter, Node.TX_AVAILABLE, TA_egg)).schedule (Kernel.
378                         Now);
379             }
380
381             if (rx_buffer.empty ()) {
382                 notify_node_enter (null);
383             }
384
385             // iba update HPC when we see a tailflit (i.e. a packet has
386             // finished transmission)
387             //     Flit f = tx_buffer.top();
388             //     if (f != null && f.packet != null && f.isTail() && f.packet.sl == 0)
389             //      link.updateHPC(f.packet);
390             //else if (f != null && f.packet != null && f.isTail() && f.packet.sl == 1)
391             //  link.resetHPC();
392
393             rx_buffer.insert (tx_buffer.pop ());
394         }
395
396         public void notify_node_enter (EasterEgg egg) {
397             if (DEBUG)
398                 System.out.println (toString () + " notify_node_enter");
399             if (Kernel.TRACE_PACKET) {
400                 Flit f = tx_buffer.top ();
401                 if (f != null && f.packet != null
402                         && f.packet.pid == Kernel.PACKET_TRACED
403                         //       && uID == 136) {
404                         && (f.isHeader () || f.isTail ())) {
405                     System.out.println (Kernel.time () + toString ()
406                                         + " nne: " + f.toString ());
407                 }
408             }
409             EasterEgg FOT_egg = new EasterEgg ();
410             FOT_egg.vl = this;
411             FOT_egg.caller = this;
412             if (DEBUG)
413                 System.out.println (toString () + " call FLIT_ON_TOP");
414             (new Event (receiver, Node.FLIT_ON_TOP, FOT_egg)).schedule (Kernel.Now);
415         }
416
417         public void purged_rx_enter (EasterEgg egg) {
418             if (DEBUG)
419                 System.out.println (toString () + " purged_rx_enter");
420             // possible optimization; only send when it moved from full to
421             // non-full
422             if (NU_wait_rx) {
423                 EasterEgg RA_egg = new EasterEgg ();
424                 RA_egg.vl = this;
425                 (new Event (transmitter, Node.RX_AVAILABLE, RA_egg)).schedule (Kernel.
426                         Now);
427             }
428             // CHANGE
429             if (new_connection_acceptable ()) {
430                 notify_connect_enter (null);
431             }
432             //      if (rx_buffer.non_reserved() && previous == null) {
433             //        notify_connect_enter(null);
434             //      }
435         }
436
437         // CHANGE
438         public boolean new_connection_acceptable () {
439             // if there is no current connection, and either the rx buffer
440             // is empty or the last flit in the rx buffer is a tail flit, it
441             // should be possible to accept a new connection on this vlane.
442             // of course, only if there is still room in the rx buffer!
443
```

```
444             if (previous == null) {
445                 Flit f = rx_buffer.latest_inserted ();
446                 return (rx_buffer.non_reserved () ||
447                         (f != null && f.isEnd () && !rx_buffer.full ()));
448             } else {
449                 return false;
450             }
451
452
453         }
454
455         public void notify_connect_enter (EasterEgg egg) {
456             if (DEBUG)
457                 System.out.println (toString () + " notify_connect_enter");
458             EasterEgg C_egg = new EasterEgg ();
459             C_egg.caller = this;
460             C_egg.info = "called from notify_connect_enter";
461             (new Event (transmitter, Node.CONNECTABLE, C_egg)).schedule (Kernel.Now +
462                     Kernel.
463                     Cycle (1),
464                     transmitter.
465                     connectable_flag);
466         }
467
468         // event implementations, remember to update the dispatcher function
469         // if adding more events!!
470
471         public static final int FLIT_IN_TRANSMITTER = 10;
472         public void flit_in_transmitter (EasterEgg egg) {
473             flit_in_transmitter_enter (egg);
474         }
475
476         public static final int FLIT_TRANSMITTED = 20;
477         public void flit_transmitted (EasterEgg egg) {
478             flit_transmitted_enter (egg);
479         }
480
481         public static final int PURGED_RX = 30;
482         public void purged_rx (EasterEgg egg) {
483             purged_rx_enter (egg);
484         }
485
486         public static final int SEND_FLOWCONTROL_RX = 40;
487         public void send_flowcontrol_rx (EasterEgg egg) {
488             send_flowcontrol_rx_enter (egg);
489         }
490
491         public static final int SEND_FLOWCONTROL_TX = 50;
492         public void send_flowcontrol_tx (EasterEgg egg) {
493             send_flowcontrol_tx_enter (egg);
494         }
495
496         // common for all VLane events
497
498         public void dispatcher (int dispatcher, EasterEgg egg) {
499
500             switch (dispatcher) {
501             case FLIT_IN_TRANSMITTER:
502                 flit_in_transmitter (egg);
503                 break;
504             case FLIT_TRANSMITTED:
505                 flit_transmitted (egg);
506                 break;
507             case PURGED_RX:
508                 purged_rx (egg);
509                 break;
510             case SEND_FLOWCONTROL_TX:
511                 send_flowcontrol_tx (null);
512                 break;
513             case SEND_FLOWCONTROL_RX:
514                 send_flowcontrol_rx (null);
515                 break;
516             default:
517                 System.out.println ("VLane: no such event " + dispatcher);
518             }
```

```
519        }                                   // end dispatcherEvent
520
521        public void purgeEvents () {
522            while (scheduled_events.size () > 0) {
523                Event e = (Event) (scheduled_events.elementAt (0));
524                scheduled_events.removeElementAt (0);
525                Kernel.globalHeap.removeEvent (e);
526                e.dismiss ();
527            }
528        }
529
530        public Vector scheduled_events = new Vector ();
531        public Vector scheduled_events () {
532            return scheduled_events;
533        }
534
535    }                                       // end class VLane
536
```