UNIVERSITY OF OSLO
Department of Informatics

Hop-by-hop Flow Control in Ethernet Networks; Implementation and Simulationbased Analysis of Performance

Bergfrid Marie Skaara

7th May 2007

Abstract

T he overall goal of this thesis is to evaluate the properties of Protocol **P** in an Ethernet context. The organization, amount and management of buffers in a switch are the most important network resource studied here. The evaluation is done through simulationbased analysis using the J-Sim network simulator. Specifically, this thesis looks at how Protocol **P** behaves compared to IEEE 802.3x and to the no flow control scenario with respect to performance (throughput and latency), backpressure, packetloss, deadlocks and livelocks in the Gigabit Ethernet context. And, we have searched for differences in control message overhead, buffer occupancy and bottleneck link utilization between Protocol **P** and IEEE 802.3x.

U sing a irregular spanning tree with 16 switches and 64 hosts, our findings are very internally coherent, and show a very small or none significant difference between the two flow control schemes. We have shown that protocol **P** in fact exhibits all the promised properties, with the limitation of deadlocks to store-and-forward deadlocks. The surprisingly low variance when comparing the two flow control schemes are mainly attributed to using the same network components, and in particular pause scheme, buffer thresholds and link scheduling. We conclude that in the current Ethernet context, protocol **P** does not give additional performance, and has the drawback of higher buffer management cost.

0.1 Acknowledgments

I would like to thank my mentor Tor Skeie, and his assistant Svein-Arne Reinemo, for guiding me through the work related to this master thesis. Thanks to my fellow students who have inspired me and encouraged me to keep going when times were rough and the days at the computer room endless.

T hanks to Hung-Ying Tyan for advice on the J-Sim in general, and permission to use illustrations from the documentation.

M ark Karol, thank you for being interested in my work on implementing your protocol, for providing me with illustrations from your papers, and for answering my implementation related questions.

I would like to thank my friends and family for being patient, understanding and supportive during my long time involvement with the university. In particular; thank you Linda for being such a great friend, I miss our small-chats in the Katina; thanks to Anne-Helen and Sveinar for pushing me through to the final end, your support has been invaluable; Pedro .. you are one in a million, and the one spot I find peace when everything else is rough, thank you!

Contents

List of Figures

Chapter 1

Introduction

G iven the scenario of a long-term power blackout striking a city, chaos resulting from lack of interaction, frustration of not being able to do a wide variety of the normal tasks, and confusing feeling of isolation might soon become evident. Computers and communication networks connecting them are a vital cornerstone in today's modern society. They come in all sizes and shapes today; devices connecting home PCs to the Internet, corporate and campus networks as well as systems designed to aid high performance processing, to mention a few. In each case there is an endless competition for resources, and handling these disputes is a vital part of most communication networks. Flow control applied in packet networks is one technique aiming to regulate resources and manage congestion prone networks:

> No packets will be dropped inside a packet network, even when congestion builds up, if congested nodes send back/-pressure feedback to neighboring nodes, informing them of unavailability of buffering capacity - stopping them from forwarding more packets until enough buffer becomes available.[43]

1.1 Research questions and focus

O ur overall goal is to evaluate the properties of Protocol **P** in an Ethernet context. The organization, amount and management of buffers in a switch are the most important network resource studied in this thesis. The following list states our specific research questions for this thesis:

1. What is the current state of congestion and flow control approaches in general, and backpressured store and forward packet switched net-

works in particular?

2. How does Protocol **P** behave compared to IEEE 802.3x and to the no flow control scenario with respect to performance (throughput and latency), backpressure, packetloss, deadlocks and livelocks in the Gigabit Ethernet context?
3. Are there any differences in control message overhead, buffer occupancy and bottleneck link utilization between Protocol **P** and IEEE 802.3x, and if so, what characterize these differences.
4. To what extent do Protocol **P** apply to solving deadlocks in general, or is it limited to handling store-and-forward deadlocks?
5. In what extent is Protocol **P** suited for SAN ?

T he first question will be solved through a literature review. Questions 2 and 3 require actual implementation and testing of the protocol. Question 4 is target for a theoretical discussion, in which we address some misconceptions about Protocol **P** . The last research question touches the properties of the interconnection network itself and evaluates the practical usage of Protocol **P** .

T his chapter provides an introduction to the theoretical framework for this thesis, as well as build an initial understanding of our research domain. At the end, you will find an overview of important concepts and abbreviations.

1.2 Theoretical framework: network, switches and protocols

T his section precedes the problem domain of this thesis. Its intention is to establish a theoretical framework for this work by defining key concepts and models. In other words, this material is a necessary requirement for describing and examining our research problems, but it is not part of our research domain in its own right, hence the structural separation.

1.2.1 A simple network model

F or illustrating purposes we adopt a simplified network model that will be further specified and adjusted when needed. In the purest form, a net-

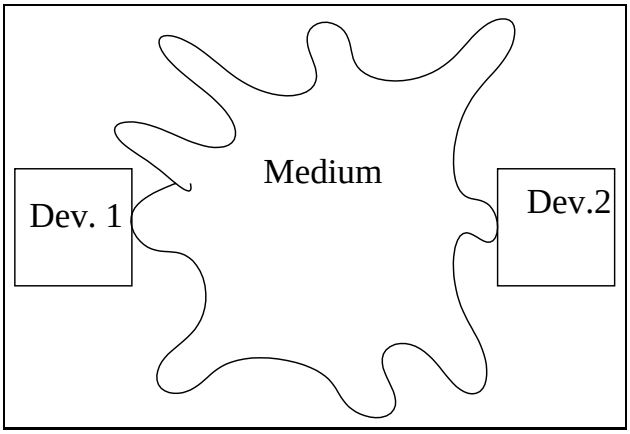

Figure 1.1: Simplistic network model with two devices connected by a medium cloud

work consist of a physical transmission medium and at least two devices capable of communicating over that medium, illustrated by figure 1.1.

A device commonly reside inside a *network node* (node for short), and in the following we use this term as a general concept including the transmission interface device. Likewise, the medium cloud connecting the nodes will be referred to as an *interconnection network* (network for short). This network is either a *broadcast network*, in which a single channel is shared between all the connected nodes, or a *point-to-point network* in which individual pairs of nodes are connected and data from one endpoint to another might have to traverse one or more *intermediate nodes* enroute. The capabilities of these intermediate nodes beyond the *data forwarding* ability, varies greatly, and will be addressed later when applicable. Broadcast networks are alternatively referred to as shared medium network.

N etworks can also be classified according to their scale [72]. For many people scale means recognizing the difference between their office/campus network, usually an Ethernet Local Area Network (LAN), and the worldwide Internet. We will in this thesis in addition look at networks scaled both between and below these two.

T he scale dimension can be summarized by 'X Area Network', where the X denotes the geographical magnitude of the network. *Local Area Networks* (LANs) are well-known, but in order to be specific in our discussion, some aspects must be clarified. Its size is restricted in the order of a few meters to some kilometers [72, pg. 16]. Extending the campus boundaries, the

term LAN is replaced by *Metropolitan Area Network* (MAN) or *Wide Area Network* (WAN) even further out. In the outer edge the Internet resides, connecting computers worldwide. The shorter the range of the cabling, the more common is the use of broadcast technology. Contrary, point-to-point links dominate the layout of the Internet.

A *System Area Network* (SAN) may adopt existing LAN technologies as well as network schemes targeted at SAN specifically. Among these are Myrinet [7], Infiniband [3], Autonet [66], gigabit Ethernet (GE) and Advanced Switching Interconnect (ASI). [63] represent an overview of server I/O demands and fabric types, and a comparison of gigabit Ethernet and Myrinet is reported in [11].

B oth LAN and SAN are interconnection networks, the difference lies in their scale, and partly in the way they are adopted in practical solutions. A SAN tends to be denser with respect to number of nodes, and to have physically shorter channels connecting them. Hence, the operations within the nodes; the switching, routing, queuing and general protocol operations, take up a large part of the end-to-end delay (latency) compared to the interrouter delay determined by bandwidth and channel span.

1.2.2 Network protocols and the OSI reference model

> Network terminology tends to follow the layered architectural model[67, pg. 55]

I n this thesis we follow the OSI Reference Model [70] (shown in figure 1.2 as a theoretical framework for our discussion.

A (network) *protocol* is a set of rules regulating communication between entities that exchange data, typical one per layer, in a system at a given time: "a set of behavioral algorithms, message formats, and message semantics used to support communications between entities across a network" [35, pg. 676]. Together the list of protocols make up a *protocol stack*. The seven-layered monolithic reference model outlines the tasks of each layer, however it is not a *network architecture*; a term that should only be used when denoting both the set of layers and the corresponding protocols [72, pg.28]. In other words, the reference model dictates *what* whereas a protocol dictates *how*.

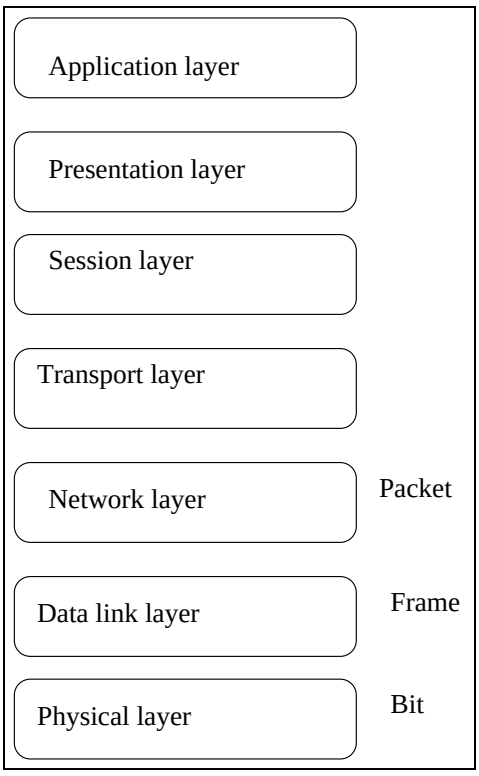

Figure 1.2: The OSI Reference Model

L ogically data is exchanged horizontally between peer A and peer B. Physical the data traverses down the protocol stack of peer A, over some form of medium, and up the protocol stack of peer B. As such, one of the most important properties of a layer is to offer a well-defined interface, defining "which primitive operations and services the lower layer makes available to the upper one"[72, pg. 27].

D ata unit *encapsulation* is performed at each layer; protocol control information is added to guard the data passed by the above layer, over the network to the counterpart layer at the receiving node. The raw data offered is termed *Service Data Unit* (SDU), whereas the result of encapsulation is *Protocol Data Unit* (PDU). The PDUs relevant for this thesis are *frame* (link PDU) and *packet* (network PDU). When the layer is of minor importance, the term 'packet' may be used here in a general way to denote 'data unit'.

Data Link Layer

R esiding directly above the physical layer, the *data link layer* handles direct communication between network entities over some form of physical medium. As such, the linklayer must support framing, addressing and error detection to fulfill the requirement of an interface. Hence it also deals with transmission errors and controls the flow of data, the latter being the main attention of this thesis.

C onceptually and physically, the data link layer is split into *Logical Link Control* (LLC) which interfaces to the above (network) layer in an uniform manner [72], and *Medium Access Control* (MAC) which is specific to the network technology, and exchanges data directly with the physical layer [67].

T he layer specific functions the intermediate nodes of an interconnection network perform, gives them their designations. The data link layer uses the term *bridge* whereas at the network layer it is referred to as *router*. The term *switch* is however more widely used at the link layer. In fact, a switch and a bridge are technically the same thing, only labeled different of marketing reasons [67, pg. 150]. Moreover, a hub is an intermediate node termed repeater if it blindly broadcasts a data unit on all its links, whereas termed a (switching) hub if it performs forwarding based on address lookup.

We will in the following use the term switch for all layer 2 intermediate devices unless the context requires us to be more specific with respect to functionality.

1.2.3 Topologies

A *topology* is the description of the physical (or logical) layout or arrangement of edge and intermediate nodes and channels in a network. In other words, a topology is like a road-map. [35] identifies four network classes using topology as the classification criteria: shared-medium, direct, indirect and hybrid networks. Ethernet is in this scheme classified as a shared-medium LAN, in which the *arbitration strategy* is a key feature [35].

The *shared-medium networks* (1) coincide with the above mentioned broadcast networks. Point-to-point networks are in this scheme split into *direct* (2) and *indirect* (3) networks. The difference lies in that the former connects processing nodes directly, whilst the latter positions switches between those processing nodes.

1.2.4 Generic switch model

The components of a generic switch are commonly buffers, switching unit, routing and arbitration unit, link controllers connected to input/output ports, and processor interface [35] shown in figure 1.3. The type characteristics of the switch, as well as the buffering technique(s) used depend on how the above elements are organized in relation to each other.

The *switching unit*, also known as switch fabric, connects inputs to outputs, and in order to reduce the delay across the unit, a fully connected crossbar is commonly used [35][1].

The *routing and arbitration unit* implements the routing algorithm (explained in chapter 3), chooses an output link for each arriving message. The algorithm used at a sending node selecting the next packet to be sent over the link, also known as *scheduling algorithm*, can base the choice on factors like packets' order of arrival, service priorities, service deadlines, fairness considerations. If a scheduling algorithm is well-behaved, it means

[1] Current status is that the switching unit itself is designed as an interconnection network

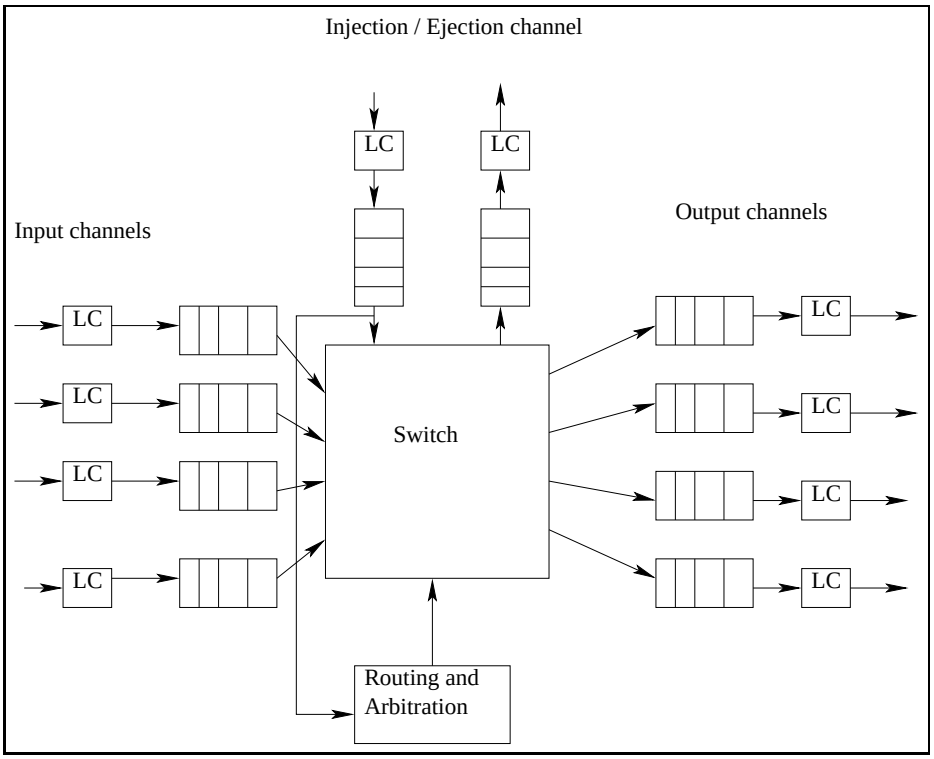

Figure 1.3: Generic Switch Model, here shown with 4 input and 4 output channels in addition to single injection and ejection channel. LC=link controller

that no packet is continually ignored/neglected and hence not continually bypassed[2], as can be the case in a strict priority-based scheduling.

T he *link controller* is in charge of coordinating transmission of flow control units between two adjacent nodes. In this context *flow control* refers to a synchronization protocol for information exchange between two adjacent nodes [35]. The information may be transferred and managed in units of varying size, for example Ethernet frames. However, a packet can be broken down into smaller logical information units, *flits* (flow control units), that denotes the smallest unit dealt with by the request/acknowledge signaling [35]. In contrast, a *phit* is the physical unit encompassing the actual number of bits that can be parallel transferred in a single cycle on the channel.

S witching techniques will be covered in detail in chapter 3. For now we only note the distinction between packet switching, also known as *store-and-forward switching* (SAF) in which an entire packet is received and buffered before the forwarding operation starts, and pipelined *cut-through* based switching in which header processing starts as soon as the header has arrived not waiting for the entire packet to arrive. The majority of this thesis deals with store-and-forward packet switching exclusively, and the switch buffering techniques discussed below reflect this focus.

1.2.5 Performance

I n relation to performance evaluation of a network in general, and switches, routers and links therein in particular, parameters like *bandwidth*, throughput and delay are considered. The first parameter refers to the theoretical maximum amount of traffic it is possible to get through the network per time interval (second). In contrast, *throughput* is the physical measured amount of data flow obtained, a fact that highlights the wish to reduce delay and tune the load in order to keep the throughput as close to the theoretical limit as possible.

T he *delay* parameter can be subdivided based on the location it arises according to [35]. *Routing delay* denotes the lookup time required by the switch to determine output link for an arriving packet (and in some cases also to set the switch). *Intrarouter delay* is the switch internal propagation

[2] a situation referred to as livelock in [43]

delay whilst *interrouter delay*, is the propagation time introduced by physical links. *Latency* refers to the cumulative delay, both intrarouter and interrouter between a source and destination node pair. We use these performance parameters in our simulation studies, but only as a basis for comparing different protocols. The interested reader should refer to [14, 15] for a theoretical network calculus.

I n many ways *Quality of Service* (QoS) is a measure of performance parameters, but while the latter is simply a quantitative description, QoS refers more to what guarantees the network can (and do) make regarding those parameters. We do not explicitely address QoS issues in thesis, except for the cases in which this has been integrated in a PAUSE flow control scheme, partially because priority protocols and flow control protocols conteract each others actions. An overview of the diffserv and intserv schemes can be found in [82].

1.3 Problem domain

I n this thesis we will review the field of data link layer flow control and evaluate a proposed protocol for handling deadlock issues in interconnection / system area networks. The protocol is implemented into a simulation environment, and run in a varying set of scenarios. Simulation results will also be obtained from alternate existing protocols, and the data analyzed and compared.

O ur attention will, in line with our previously stated questions, be focused at three research areas and their intersection. First, the IEEE 802 Ethernet technology and standard, with emphasis on MAC issues. Details of the physical medium and encoding techniques are beyond the scope of this thesis. Second, the phenomenon of (network) congestion will be reviewed, laying the foundation for congestion management and flow control (with emphasis on the latter). Third, we address deadlocks and schemes designed to resolve them. Network topology, packet forwarding technique and routing algorithm are closely related to the above three areas, and are included to enhance the understanding of our focus area. Because the proposed protocol is, as will be explained in detail later, tightly connected to buffer management schemes, buffer layout and queuing techniques in switches may be viewed as a fourth research focus.

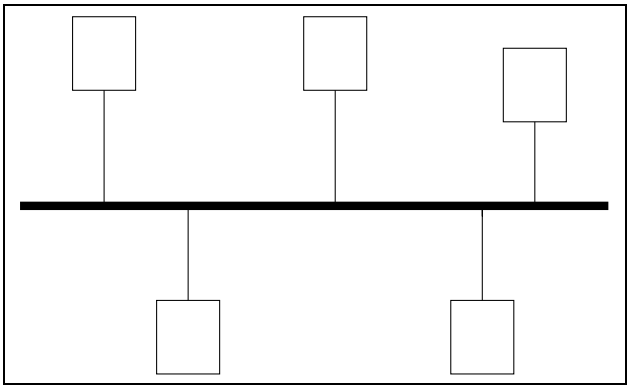

Figure 1.4: Classic Ethernet shared-medium layout. 5 devices are here connected to a single chared channel

T he rest of this section gives an introduction to Ethernet technology, flow control and deadlocks. In the next section, we narrow this domain into a problem specification including Ethernet in the SAN context, buffer layout and queuing, and the IEEE 802.3x and Protocol **P** flow control schemes. The chapter closes with a concept summary and an outline for the rest of this thesis. We start our journey with the history of Ethernet.

1.3.1 Ethernet

T he first local area network designed is credited to Bob Metcalfe. However, the *Ethernet* technology, as he termed it in 1973 in his PhD "Packet communication" [49] at Harvard was not a completely new idea. During his studies at M.I.T. and Harvard, Metcalfe was exposed to the work of Norman Abramson at the university of Hawaii. This ALOHANET used short-range radios with separate frequencies for upstream and downstream transmission [72]. ARPAnet, which later developed into the Internet, also contributed to Ethernet with its packet switching concept.

T ogether with the colleague David Boggs, Metcalfe designed and implemented the Ethernet in 1976 [48], and received patent in the following year [50]. The network connected computers via a thick coaxial cable running at a datarate of 2.94 Mbps [72]. Figure 1.4 shows a classic Ethernet network layout. *Media access control* (MAC) was crucial, as was the ability to confirm successful transmissions in this shared medium technology.

D EC, Intel and Xerox outlined the DIX standard in 1978 for a 10Mbps Ethernet, which was to become the IEEE 802.3 in 1983 with a few minor

changes. Today the two standards coexist and can be told apart by having separate value ranges for a header field in the frame. Ethernet has continued to evolve with higher datarates, improved cabling and more sophisticated media access / utilization techniques, such as the introduction of switching and flow control.

Media Access Control

L et the term *station* denote a network node in an Ethernet. A *Collision (/Access) Domain* is then the stations that compete for the resources in a shared-media environment, i.e. the stations that might attempt to transmit on the media possibly in the same or overlapping time interval(s). The access protocol used in Ethernet is termed *CSMA/CD*: Carrier Sense Multiple Access (with) Collision Detection, and is commonly implemented by the network adapter in hardware [59]. The operation of Ethernet MAC is specified in *IEEE 802.3*, and a descriptive illustration of the transmit and receive processes can be found in in [67, fig 1.8].

I n short, a devise having data ready for transmission checks the channel (carrier sense), starts transmission if the medium is clear or defers if it is busy. After a short time period known as *inter frame gap* the station may start the next transmission unless a collision has been detected. A *collision* is characterized by the bitstream from two or more stations meeting and merging on the channel causing interference. A transmitting station knows its own bit pattern and will hence by sensing the medium know when that pattern is altered. To alert other stations and make sure everyone sees the collision, a detected collision is emphasized by continuing transmission a bit longer (jamming). When a collision has cooled off, the stations ready to transmit uses a *binary exponential backoff algorithm* to randomize waiting time before new access attempts to avoid subsequent collisions. The interested reader should refer to [72] for a description of the algorithm and an analyze of the resulting channel efficiency.

D ue to the collision detection mechanism, the propagation time (in standard Ethernet) of the media places an upper bound on the physical length of a channel. With higher datarates, the length has to be reduced proportionally to preserve the ability to detect interfering signals. Moreover, the backoff algorithm results in statistical randomness, implicating that the Ethernet CSMA/CD is non-deterministic by nature. Both issues restrain the possible applications of Ethernet, and measures have been taken to overcome these limitations, creating new niches for Ethernet.

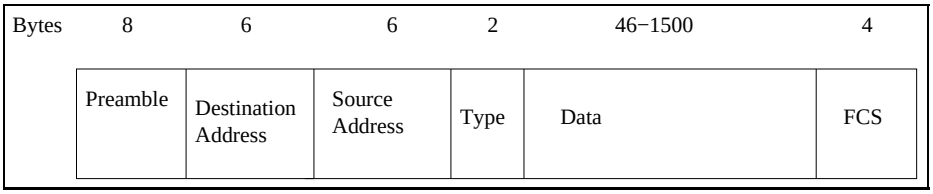

Figure 1.5: Ethernet Frame Format, FCS = Frame Check Sequence

D ata units in an Ethernet are called *frames* and include both the raw data to transmit and some control information. The *frame format* is shown in figure 1.5. The source and destination address fields are 48 bit unicast MAC addresses identifying the communication endpoints.

Full-duplex operation and micro segmentation

I n a shared-medium environment like the CSMA/CD MAC it should be obvious that a station can only transmit or receive at a given time, as doing both would imply more than one frame being on the channel simultaneous, and hence a collision is occurring. Two stations connected to the channel can obtain bi-directional communication, but only by taking turns using the channel. This is known as *half-duplex*. Recall that collisions are associated with an access domain. If the transmit and receive processes were using separate access domains, no collision would occur, and *full-duplex* mode would be enabled. In fact, this is done today by applying dedicated media in the form of point-to-point cabling replacing coaxial cable with twisted pair or optical fiber, and applying switches creating dedicated LANs [67].

I n a *switched Ethernet*, each port of the switch marks the termination point of a collision domain and input buffers are available at the MAC in each port to ensure no send/receive conflicts. *Micro segmentation* as illustrated in figure1.6 denotes the situation in which a single end station resides in a collision domain terminated by a switch port [67]. Applying full duplex links, the need for access control vanishes, as there will be no collisions. One also ensures that each station has dedicated bandwidth toward the switch, Note however that this resource might get restricted downstream in the network if links are aggregated toward a common destination.

Beyond 10Mbps

E thernet was in 1995 standardized for 100Mbps operation, also known as *fast Ethernet*, in the IEEE 802.3u supplement to the existing 802.3 stand-

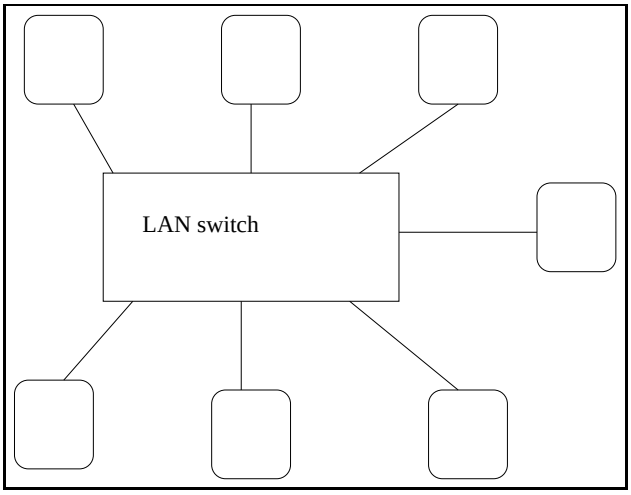

Figure 1.6: LAN micro segmentation. 7 devices are here connected by separate channels to a LAN switch

ard for backward compatibility. In a way, this was just a stepping stone for the *gigabit Ethernet* standard published as IEEE 802.3z three years later. The 3z supports both full-duplex operation as in the switched LAN described above, in which CSMA/CD is disabled, and half-duplex operation still in need of the access algorithm. To overcome the severe physical extension limit required by the MAC operating at 1Gbps datarates, features as *carrier extension* and *frame bursting* has been included for the half-duplex mode[72]. In 2002 yet another factor 10 was added to the Ethernet family by the IEEE 802.3ae specifying 10Gbps operation. Measurement on real HW architecture (Intel) for this latest addition can be found in [31].

D ue to the shared property, Ethernet and other shared-medium networks have an upper bound on legal amount of hosts to prevent bandwidth from becoming a bottleneck. Moreover, Ethernet uses CSMA/CD as the arbitration mechanism and [35] notes the limited bandwidth and span as factors restricting Ethernet from having a reasonably use in multiprocessors, eg. they have serious scalability problem. But, recall our description of switched Ethernet with dedicated LAN and gigabit datarate. With these features changed, Ethernet is virtually no longer a shared-medium network and operates as if it were an indirect irregular network.

1.3.2 Flow control

T he main focus of this thesis is *flow control schemes for managing congestion at the data link layer in a packet switched network*. Congestion, as in 'over-

crowding' and 'clogging', is a well known phenomenon from every day life, bringing images of weekend traffic on the highways and city rush-hours into mind. Whenever "resources are scare and highly in demand"[55], congestion occur.

T he issues of congestion and flow control have been a hot potato in the field of computer communications ever since researchers, lead by M. Schwartz and L. Kleinrock, started working on controlling network packet flow in the mid 1970s[55]. One of the most well-known and frequently cited works is *Congestion Avoidance and Control* [36] by Van Jacobsen in 1988. The advances in processing power, memory, and channel bandwidth increasing the resource pool have not contributed to eliminate congestion phenomena, and several myths about congestion have been discussed by Jain [37, 38].

I n the years proceeding the popularity of the Internet and growth of multimedia traffic, this research field was rather lucid and surveys like [25] and [37] cover the development. However, recent work is more diverse and both improvements to existing strategies and new suggestions are made to accommodate the changing traffic pattern and load in today's network. Multimedia and real time data place quality of service restrictions on the performance. Satellite and other wireless transmission media are also gaining popularity, and call for their own congestion management techniques.

S everal decades ago Pouzin envisioned parts of this development:

> Research on traffic control in packet networks is still much needed, although it may not mature in time to be applied. Indeed, new technologies are likely to obsolete existing designs intended to optimize the use of low bandwidth fixed circuits. High rate broadcast media and fast digital switching will eventually place flow control in a totally different context.[60]

Flow control and the OSI stack

T he concept of flow control can be applied at multiple layers of the OSI stack, but is found in its purest form at the data link layer. The overall task is to prevent data units from arriving at a node faster than they can be processed (and buffered / forwarded) there, eg. avoid swamping a receiver. This requires the source to know the capacity of the intended receiver in advance, or the receiver must inform about its situation [74]. Congestion can be handled by network nodes dropping excessive packets (and relaying on

end-to-end protocols for loss recovery) or exchanging information in order to avoid drops. In fact, the default switch behavior under congestion is to drop packets [67].

Defining congestion and flow control

T he terms *congestion control* and *flow control* are sometimes used interchangeable in the literature, whilst some researchers make a clear distinction between the two. Unfortunately, this distinction is not uniform and hence the same principle can be termed congestion control in one text and flow control in another. The criteria for making this distinction is commonly based on either the layer at which the mechanism is applied, the location in the network where the control is applied, or a combination of the two.

F or now, let us settle on the following definition of congestion related to performance:

Definition 1.1 Congestion *occurs in packet networks when the demand exceeds the availability of network resources leading to lower throughput and higher delays. [43, pg. 923]*

F urther, the main issue is managing congestion scenarios, and hence the term 'congestion control' might be viewed as the key term. On the other hand, controlling the flow of data between (adjacent) network entities at the data link layer is the core of our simulation study, and consequently the term 'flow control' might be more precise. We therefore chose to use the latter as our key term, as well as adopt the former when it is relevant for generality and compatibility with the literature.

F low control generally aim to reduce the flow of excess traffic, and we adopt the following definition by Seifert as a starting point:

Definition 1.2 *Flow control is "a mechanism that prevents a sender of traffic from sending faster than the receiver is capable of receiving".[67]*

This implies some form of dialog between the communicating entities that may allow or stop information propagation in the system [35].

Purpose of flow control

A set of main functions of flow control proposed in [25] match the focus and framework for this thesis well:

1. prevention of throughput degradation and efficiency loss due to overload.
2. deadlock avoidance,
3. fair allocation of resources among competing users, and
4. speed matching between the network and its attached users.

Together, these functions contribute to limiting access of traffic to selected sections of the network.

Approaches to flow control

A taxonomy for flow control will be presented in chapter 4 along with a discussion on the relation between these terms. From an end user perspective, it is more convenient to view the nett perceived performance than the yield of distinct layers. Layers interact, and intervention at one level can affect performance at other levels. I therefore chose to present a broad picture of congestion and flow control, as it in the future might be relevant to study how this work relates to higher layers.

B asic throttling tools available for implementation of flow control include stop-and-go signals, credits (quantifying scheme, e.g. ACK), rate (timeslot allocation), delay (outstanding ACK), and class (traffic segregation) [60].

1.3.3 Deadlocks

D eadlocks are an interesting phenomenon in computing, and can be generalized into involving two roles; the actor and the resource. Whenever an actor requests a resource, there is a potential danger of creating a situation that leads to a deadlock. However it is not the request itself that is the problem. The lock surfaces not until an actor is granted access to a resource but cannot move on until a second resource, which at that point is unavailable, is granted. Hence the actor must wait, and meanwhile it blocks the first resource for subsequent actors. Actors ruled only by 1's and 0's are far less likely to give up waiting when they first have made a claim than humans.

L ivelock situations are related to deadlocks, but differ in that resources are being moved between actors, however only in such a way the needs of one of more actors are not fulfilled. For example, consider the following scenarios: driving your car you get stuck in a traffic jam caused by power blackout making the traffic lights useless , or you enter a roundabout, end up in the inner lanes and find that you cannot reach the exit and have to keep circling for some time. These scenarios correspond to a deadlock and livelock situation respectively, however very simplified.

1.4 Problem specification

T his section specifies our research focus with finer granularity by introducing the IEEE 802.3x and Protocol **P** flow control schemes, moving Ethernet into the SAN context and introducing buffer layout and queuing. The subsequent section derive a list of concrete research questions based on this problem specification.

1.4.1 IEEE 802.3x PAUSE flow control

I EEE 802.3x flow control, also known as *MAC Control*, is a generalized architecture and protocol for "real-time control and manipulation of MAC sublayer operation" defined in [69, clause 31]. For backward compatibility, MAC Control is an optional capability in Ethernet, and at the present time, MAC Control PAUSE operation described in [69, Annex 31A and 31B] is the only available control function.

T he key feature is to monitor buffer usage in each switch and send pause control frames to the upstream neighbor if congestion is building up. In other words, IEEE 802.3x is a simple stop/start form of flow control for single full duplex links. It performs control at each node along the path of a data stream, with a sender and a receiver entity separated by a single link as the network subset constituting the control universe. Because the 'at each node'-property these protocols fall into the *hop-by-hop* category for flow control and result in the phenomenon *backpressure*. This approach is presented in detail in chapter 4.

1.4.2 Protocol P flow control

T he overall operation of protocol **P** can be described as a selective backpressure mechanism. It is partially an enhancement of IEEE 802.3x flow

control, and as such it exhibits the same properties of hop-by-hop and backpressure. **P** uses gigabit Ethernet and PAUSE signals as an example of a well-known backpressured packet network. The technique can however be applied to other network technologies, and hence to claim **P** to be "a modified version of PAUSE" is slightly misleading. In addition, this protocol provides selective backpressure and includes an advanced buffer management scheme that is claimed to prevent deadlocks and livelocks. These latter issues targets our specific focus within the area of congestion and flow control. The Protocol **P** approach is presented in detail in chapter 5.

1.4.3 Ethernet in the SAN environment

T he purpose of this section is to restrict the network environment to the focus of this thesis. First, it is an underlying assumption that unless otherwise noted, presentation and discussion herein is based on a connectionless network layer with no form of reservation protocol in use. This framework is consistent with [67].

S econd, we restrict our focus to the 802.3 Ethernet MAC sublayer (of the data link layer). This implies adhering to the IEEE specifications on issues such as frame format and data relaying.

T hird, we direct our focus to the SAN context when it comes to scale and application of interconnection networks. However, we do not impose SAN as a strict limit due to the fact that allot of existing flow control research is targeted at LANs. Transferring that research material uncritically into the SAN context would be a flaw in our work. We therefore take the approach to present relevant research within its original context, conduct our experiments within a SAN context and then analyze and evaluate our findings.

T here is general consensus that *determinism* is an important requirement in the SAN context along with QoS requirements and traffic prioritizing options. The debate on whether Ethernet is suited as a SAN technology primarily focus on if, and to what degree, Ethernet is deterministic. In 2002 this was a hot topic, and the core of dispute was on whether a switched dedicated link configuration, removing the randomness of the access mechanism, was sufficient to make Ethernet deterministic. [83] evaluated buffering delay and suggested a means of estimating the worst-case response time. The IEEE 802.1p priority mechanism were studied in [68], finding that the main delay is inside the network nodes (protocol stack), indicating that it is in the end-nodes that may gain the most by introducing priority.

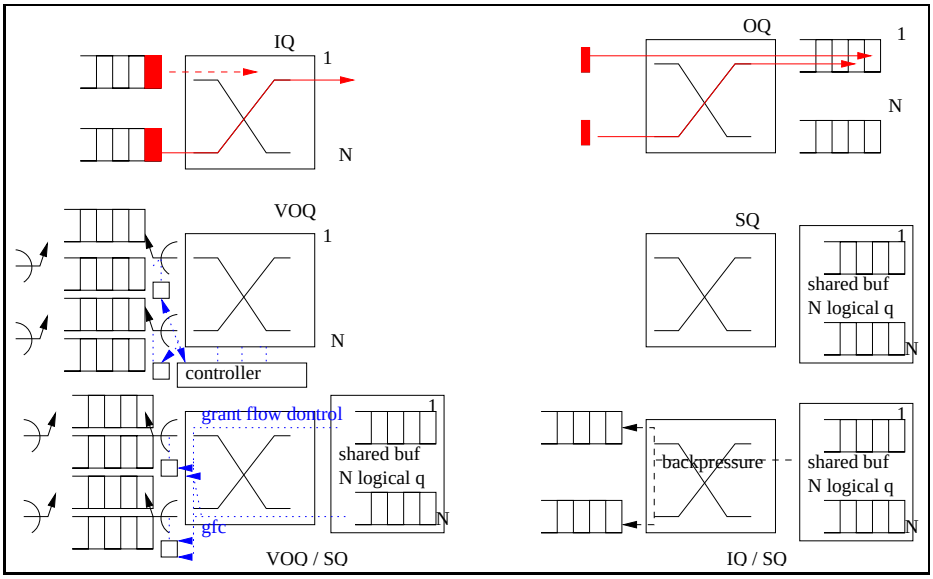

Figure 1.7: Queuing techniques

Arguments were also made based on network calculus ([24]) and OPNET simulations([40]).

1.4.4 Buffer layout and queuing

T his section moves the focus into the network nodes, and we include two issues: organization of buffers (queues) within a switch, and a brief introduction to buffer management.

Classical input and output queuing

A s previously mentioned, it can vary how the buffers are physically organized in a switch. This can be viewed as applying a temporal ordering of switching and queuing for classic architectures [9]. In both cases there is one buffer per input or output. Queuing before switching is called *input queuing* (IQ), and has the weakness of *Head-of-line (HOL) blocking* limiting the throughput. If however switching is done first, before buffering, we have *output queuing* (OQ). In the latter case HOL is avoided, but complexity and cost will limit the buffer capacity so that packet drop becomes a problem. See figure 1.7 for illustration of the different techniques.

Head-of-line blocking

H ead-of-line blocking is illustrated at the top left of figure 1.7. Both input ports have a packet destined for the exact same output port. One of these packets must thus wait to be served. In the case of subsequent packets are waiting behind the one that is temporarily blocked, and these are destined for available output ports, we say that they are blocked by the packet sitting at the head of the line, hence the name. This fenomenon is a result of *many-to-one traffic pattern*[67].

Shared queuing

A range of improvements have been suggested to enhance these two classical switching modes; *Shared queuing* (SQ) reduces the probability of packet loss by utilizing the available buffer capacity better. This is done by not retaining separate buffers for separate queues, rather by using a shared resource until it is collectively exhausted. We can also combine IQ and SQ so that packet loss in the output queue is reduced, just by holding back in the IQ if the SQ is full.

Virtual output queuing

A t each input, *Virtual output queuing* (VOQ) uses a separate queue per exit. This however requires a scheduling algorithm to choose which packet to move across the switch next. Because of this pre-sorting of arriving packets, one packet can be moved across the switch to each exit in parallel, hence the need for scheduling. VOQ can also be combined with SQ for even better performance, by having SQ handle contention, eg. take care of the head of each queue. In this case a decentralized scheduler is required at the input ports. And even as this increases complexity, we have arrived at a very robust switch where HOL is eliminated.

Buffer management

1.5 Terminology

T his section reviews the main concepts introduced here, clarifies the definitions and also add some more concepts that will be central in the upcoming parts of this thesis:

A *switch* is a device used in a computer network to interconnect parts of that network and relay data between those parts. A *Packet (switched) network* is a computer network interconnected by switches where the data units being relayed are *packets* (or *frames*). The first bytes of each packet contains information necessary for the relay operation. A way to ensure that information sent is in fact received by the intended recipient is to add some form of explicit or implicit *acknowledgement* (ack) notifying the sender about successfull reception. In the absence of such an ack, the sender might have to *retransmit* the information.

T he concept of *Performance* refers to a collection of concepts describing both theoretical and actual capabilities of a network. The three most commonly used are bandwidth, throughput and delay: *bandwidth* is the theoretical maximum amount of traffic it is possible to get through the network per time interval (second); *throughput* is the physical measured amount of data flow obtained; *delay* refers to the time spent crossing some part of the network, being it the lookup time inside a switch (routing delay), the switch internal propagation (intrarouter delay) or the propagation time introduced by physical links (interrouter delay). *Latency* refers to the cumulative delay, both intrarouter and interrouter between a source and destination node pair. *Round trip time* (RTT) is the accumulated time for a message to pass from a source to a destination and back again. *Quality of Service* is about placing restrictions and guarantees upon these(and other) performance parameters.

T he phenomenon of *Congestion* "occurs in packet networks when the demand exceeds the availability of network resources leading to lower throughput and higher delays." [43, pg. 923] Flow control is "a mechanism that prevents a sender of traffic from sending faster than the receiver is capable of receiving" [67].

T he *Data Link Layer* is the lowest level of the OSI reference model except for the physical layer itself. This layer deals with transmission of data across single communication links, and there is such only 2 network nodes and the channel connecting them involved.

A *topology* is the description of the physical (or logical) layout or arrangement of edge and intermediate nodes and channels in a network. With respect to nodes along a path within that topology, *downstream* refers to the direction data is flowing from a node, while *upstream* refers to the path

leading to a node. The analogy here is a river where you are located somewhere along its path and observe how the water is moving either towards or away from you.

A n entity is said to be *scalable* (e.g. have good scalability) if there is a reasonable relationship between its proportions. For example if you need to rent only a single extra office room when hiring a new employee, you have a scalable room allocation scheme, in contrast to if you had to get a total new office locale or rent an additional floor in the building.

E thernet (IEEE802.3) is the most widespread LAN technology in usage. Originally designed as a shared medium network, regulating the usage of the medium was crucial. *Collision (/access) domain* refers to the stations that compete for the resources in a shared-media environment, i.e. the stations that might attempt to transmit on the media possibly in the same or overlapping time interval(s). MAC is thus an important part of Ethernet regulating medium usage as well as identifying the communicating entities and managing control information. Two stations connected to the channel can obtain bi-directional communication, but only by taking turns using the channel (half-duplex). By applying dedicated media in the form of point-to-point cabling, and switches dedicated LANs are created enabling full-duplex mode and resulting in switched Ethernet. Each port of the switch marks the termination point of a collision domain and input buffers are available at the MAC in each port to ensure no send/receive conflicts.

T he phenomenon of *deadlock* is an actor / resource situation in which some resource is held by an actor and needed by some other actor, but because these actors directly or indirectly wait on each other to release the resource, neither of them can resume normal operation without some strategy to break the lock. A *livelock* is a situation of unfairness in which some actor is continously neglected (or redirected) while others are being served.

T he flow control strategy Protocol **P** is basically an extension or add-on to the PAUSE scheme defined in IEEE802.3x (z). Both perform control at each node along the path of a data stream, with a sender and a receiver entity separated by a single link as the network subset constituting the control universe. Because the 'at each node'-property these protocols fall into the *hop-by-hop* category for flow control and result in the phenomenon *backpressure*.

1.6 Outline

T his introduction looked at the problem domain and -specification. Next chapter 2 address the method used in this thesis: Network simulation as research method and the specific simulator used (j-Sim). Thereafter we present theoretical material relevant to our research questions, more specifically background and research in the fields of switching, routing algorithms, deadlocks (chapter 3), congestion and flow control including the IEEE 802.3 MAC Control (chapter 4).

B ased on the established foundation Protocol **P** is described in detail in chapter 5, followed by simulation scenarios in chapter 6. We then in chapter 7 give you our performance measurements and present the results of our simulation studies. The thesis closes with a conclusion and topics for future research.

C hapters 4 and 5 will answer our first research question. The remaining questions are addressed in chapters 7 and 8.

Chapter 2

Simulation as Research Method

T his chapter addresses the research methodology and tool used to evaluate the proposed Protocol **P** . The material is presented in a three-step approach beginning with putting our methodology in a context and advocating the selection criteria, followed by describing our chosen simulation tool, and closing with describing how the evaluated Protocol **P** is implemented into that tool.

I n step one we first present our rationale for selecting simulation as our main method, followed by network simulation in particular and approaches to simulator design. Thereafter we discuss selection criteria for a specific simulation tool along with the tools considered for this thesis.

2.1 Introduction

E valuating a network protocol can be done in several ways. Mathematical reasoning and proof was used by the inventors of Protocol **P** in [42, 43] to show the correctness of the protocol. However, to study the behavior and performance of the protocol and to be able to quantify this, a mathematical model approach is not sufficient.

T he ideal situation would be to set up a laboratory or testbed with a real, physical network in which Protocol **P** is an integrated part of the protocol stack and network parameters such as topology, size, link speed, buffering and so on can easily be manipulated. For most research communities this is an utopia due to funding in general and hardware implementation cost and time in particular.

E ven if cost was not a limiting factor, there is a great deal of research in the network domain that is prohibitively complex to be carried out in real life[19, 8]. For example, studying behavior and performance of the worldwide Internet and experimenting with alternative protocols and configuration therein, is something you simply cannot do in a live scenario. Overlooking the consequences, this is still not an ideal research situation because the researcher lacks control over the scenario.

F or these reasons simulation "is a critical tool in developing, testing, and evaluating network protocols and architectures"[8]. Especially student research projects lean toward simulation as being more or less the only feasible option. This is also true for our case. Because Protocol **P** has not to our knowledge, at the time of writing, been implemented in hardware, doing this and manufacturing a sufficient amount of network interfaces with this hardware to carry out the study in real life is not possible.

E mulation, like real life experiments, requires physical equipment where the protocol examined is included [8]. Because we are dealing with a linklayer network protocol usually found within network interfaces, it cannot be implemented in software on top of real hardware since that would both break the order of the protocol stack as well as duplicate linklayer functionality already found in the hardware used.

2.1.1 Research qualities

W hen evaluating a network protocol, we find that determinism and reproducibility are important aspects that needs to be taken into account. *Determinism* here refers to a scenario being limited in extent when it comes to time and space. In other words, determinism guarantees that a task started will finish within a finite amount of time using a finite amount of resources. *Reproducibility* here refers to being able to repeat a scenario in such a way that given the same input, the same output will be generated. Determinism is a prerequisite for reproducibility: if you cannot ensure that a task will finish, you can most certainly not put restrictions on the outcome of that task.

W e will later in section 2.2.5 look at some means of working toward these qualities.

2.2 Network Simulation

> The quality of simulation results is only as good as the methodology used to generate and measure these results.[17]

A fter choosing simulation as research method, we need to review some issues that apply to network simulation in particular. Choosing the abstraction level for the simulation affects subsequent choices like workload parameters, data collection and design issues. It will also propagate into selecting simulator approach as well as the specific tool implementing that approach. These latter issues are discussed in the following sections.

T hree major parts of any network simulation are links, nodes and load [8]. In our case the focus lays at nodes with links only viewed as transport tube exhibiting some properties, and load viewed as the research parameters varied in the experiments carried out. We return to load below in section 2.2.2 after first looking at abstraction levels.

2.2.1 Abstraction level

N etwork simulation does not come without a cost. To carry out a simulation you need computational resources. More specifically, available processing power and memory place an upper limit on how resource demanding your simulation can be [19]. The more details the simulator captures, the more accurate will the results be. But because of limited computational resources and the timeframe researchers have to carry out their work, trade-offs have to be made between accuracy and execution time [8, 19, 17]. In other words, we want to capture the aspects important to our research problem and at the same time leave out unnecessary details.

F ollowing [17], we adopt the four level simulation hierarchy ranging from overall network behavior to hardware implementation details. The list is organized from least to most accuracy (and opposite, the most to least abstraction).

Interface level Behavioral simulation capturing network interface and simple packet delivery

Capacity level Captures resource restraints such as channel bandwidth, buffer capacity and rates.

Flit level Captures detailed resource usage at a flit-by-flit basis and requires detailed modeling of mechanisms such as buffering, switching and arbitration.

Hardware level Captures micro-architecture design.

T he following paragraphs discuss our case in relation to the above hierarchy. Our research problem dictates that the interface level will not provide sufficient details to answer our questions. The hardware level on the other edge provides too much information since we want to evaluate performance in general. As [8] points out, implementation variations "can have a dramatic impact on the protocol's behavior and performance". We therefore do not want to guess on details that would most likely vary across vendors anyway.

T his leaves us with the choice of capacity versus flit level simulation. In line with our research questions, we want to capture detailed resource usage when it comes to buffer organization, management and usage. On the other hand, a flit-by-flit approach is a little bit over the edge in the Ethernet context, since the unit of flow control is Ethernet frames. But, as we shall see, examining individual parts of each frame is important at the link layer, so we cannot treat frames as a black box.

F ollowing from this reasoning, we adopt a modified version of the four level simulation hierarchy using a mild rewrite of the flit level to frame level.

2.2.2 Workload parameters

G iven an interconnection network, protocol behavior can be studied by manipulating aspects such as size and topology, routing function, switch design, parameter tuning (bandwidth, buffer size) and network workload. The latter can be defined as "the pattern of traffic (such as packets) that is applied to the network terminals over time"[17]. A generic switch model was presented in 1.2.4. This section will focus on workload in general. The remaining aspects will be addressed in 6.

Type of workload

I n general there are two kinds of network workload to consider; application-driven and synthetic[17]. Of these two, the former is the most accurate and

ideal, but it is also the one hardest to mimic. While synthetic workload introduces the risk of making wrong estimates about workload, it gives us more control over the research environment and aids us in generating reproducible results.

R eproducibility can also be achieved through application-driven workloads if traces of client execution is recorded and then fed into the simulation[17]. But then again you face the problem of capturing the right traces and ensuring that they are representative. For other researches to be able to reproduce results, trace files must be distributed along with other simulation parameters.

I n our case when not considering behavior of upper layer protocols, nor interaction with those protocols, a simple synthetic workload with controllable parameters is more than sufficient to investigate our research questions.

Traffic pattern

H aving decided on a synthetic workload, we need to consider the length of packets as well as the process of injecting them to the network. For simplicity we choose to use standard frame sizes for Ethernet data and control frames, as specified in section 4.5.1 and chapter 6.

T he injection process can be described as being periodic or not. In its simple form, a packet is injected at a regular and fixed rate. A more advanced injection process for network simulations is the *Bernuolli process*[17], which adds randomness to the rate. To model bursty traffic the *Markov modulated process* (MMP) can be used. For reproducibility we will use a fixed rate in our simulations.

2.2.3 Data collection

D ata collection is an important part of our simulation study. This section therefore addresses the topics of sampling, possible sources of errors and focus points for data collection and analysis.

Sampling

S imulator warm-up refers to the initialization phase of a simulation run. When a simulation is started it is usually done so with all resources unused[17]. In other words, there are no packets in transit, nor sitting in a buffer somewhere or being serviced by scheduler or examined by a routing lookup process. Because of this, packets injected early in the simulation will not have to compete for resources in the same way as packets injected at a later time. They will probably be able to travel the network faster and smoother than the subsequent traffic. For this reason, events in this period should be omitted from the collected data.

T he warm-up phase is over when the simulated system has reached a steady state, and this is when the main sampling begins. Measurements taken here can among others follow the *batch means method* or the *replication method*[17]. In the former the simulation is run over a long time but only once, and the values are chucked together (batched) in order to analyze patterns and variance across the chunks. In the latter approach several shorter runs are done, and each run is then examined and compared, as the batches were in the former method.

W e postpone choosing a sampling method until a specific simulation tool is selected, because we expect the tool to put some restrictions on the available alternatives.

2.2.4 Focus points

T his section identifies our focus points when it comes to evaluating the proposed protocol, **P** . These have already influenced our choice of abstraction level, and will also have implications for design issues and selecting a specific simulation tool later.

T hroughput and latency are two obvious focus points when it comes to measuring performance. These concepts were introduced in section 1.2.5. Further, we want to check that the claims about Protocol **P** being lossless and free of both deadlocks and livelocks hold. This is easily done by observing and comparing the count of packets entering and leaving the network, and by verifying that all simulations terminate properly.

R elated to the flow control itself, we want to measure bottleneck link utilization. And related to the buffer management scheme of Protocol **P** we want to examine queue behavior. For these latter focus points, we need to record detailed trace information during the simulation runs. We expect this to demand great amounts of processing power as well as memory, and for this reason we might have to create smaller scenarios than in the main experiments to obtain this data.

2.2.5 Design issues

R andom number generation plays an important role for non-periodic injection processes used for network workload. It is also used for scheduling of other events inside network nodes. The numbers generated can be either truly random or pseudo-random [17].

P seudo-random numbers are calculated based on a seed value, and this process is deterministic in the way that given a specific seed, the sequence of numbers generated will be identical, and thus deterministic. Network simulators do in fact in most cases benefit from this pseudo-randomness. By controlling the seed, researchers can repeat a particular simulation run and hence obtain reproducibility for the results. Likewise, by varying the seed randomization across runs can also be obtained. In other words, pseudo-random number generators provide the best controlled research environment for network simulations.

A nother design issue we will have to address in this work is modeling source queues for flow control[17]. If flow control slows down a source node injecting packets into the network, the amount of packets waiting to enter the network will pile up inside the node. Those packets will age, meaning accumulate delay, while being held back at the source node. This delay will affect the measured performance and following the analyzed results will be biased by the packet injection strategy.

I n a real life scenario, not only the network interface, but also the higher-level protocols and eventually client applications would be aware of the hold-back and potentially be able to react to it. It is according to [17] desired that this behavior is also mimicked by the synthetic injection process. They suggests to inject a new packet only if it can be serviced without undesired delay. We adopt this approach in our study, and return to the implementation details in chapter 6.

We return to the topics of choosing concrete parameters and generating scenarios in chapter 6 after having examined flow control and Protocol **P** in details. Below we direct our attention to different kinds of simulators.

2.3 Approaches to Simulator Design

This section gives a short overview of two simulation approaches identified by [17], with emphasis on the one selected for our study. The distinguishing feature is how they handle simulation time. Once we have selected a specific tool in section 2.4 we expand the material presented here to show how it maps to the specific tool (in section 2.5).

2.3.1 Cycle-based

The first approach is *cycle-based simulation*. In this approach a global clock is used, the simulation elapses as an alternation of read and write phases accessing global variables, and the critical invariant requires that all actions within a phase can be rearranged without affecting the outcome[17].

We view this approach as too restrictive and inflexible for our needs. In particular, working with buffer management and flow control actions we cannot fulfill the critical invariant. Multiple simultaneous requests for buffer occupancy will (and should) yield different results depending on the order they are serviced.

2.3.2 Event-driven

The second approach to simulation is *event-driven simulation*. In this approach simulation is broken down into individual *events* that are processed from a timestamp sorted queue[17]. At these *event times* system state variables may change[10].

Literature review within our problem domain has revealed that this is the far most applied approach in the simulation tools used. More precisely, the *discrete* event-driven simulation approach is used. In this context, 'discrete' refers to discrete points in time in contrast to continuously with time[21] when it comes to when the model state changes.

D iscrete event-driven simulation has in fact two phases just as the cycle-based approach we turned down. The difference is that while the latter alternated between reading and writing global state variables, the former alternates between processing all scheduled entities at the current discrete timepoint and updating the simulated clock [65]. These phases are referred to as *Entity Movement Phase* (EMP) and *Clock Update Phase* (CUP) respectively.

F ive states in which system variables can be in a discrete event model are identified in [65]. We include this material because it is relevant to the explanation of our concrete protocol implementation later.

States

T he five states are 'active', 'ready', 'time-delayed', 'condition-delayed' and 'dormant'[65]. Each state is associated with a list containing all simulation entities currently being in that state. The 'active' state stands out from the others by having only one entity in the list at any time during a simulation run.

W hile the 'active' state relates to the entity currently being processed, the 'ready' state relates to the list of entities that are waiting to be processed within the current EMP. The simulation will not progress to the CUP phase until the 'ready' list is empty.

T he two delayed states, time-delayed and condition-delayed, both relate to lists with entities that can progress to the ready and active state later. Entities in the former list are delayed for a known amount of time. In other words those entities are scheduled with a future timestamp, and simply waits for the time to elapse before moving to the ready state. Entities in the latter list are delayed with respect to some condition, and not a specific amount of time. For example, a time-delay is the natural choice for implementing interframe gap in a network interface. In the domain of flow control, scheduling the sending the next frame after a network interface has been paused is in contrast usually object to condition-delay.

T he dormant state and list is managed manually and is not associated with automatic triggers. As we shall see for our chosen simulation tool, this state can be used to handle a resource pool.

A further detailed description of discrete event-based simulation is beyond the scope of this thesis. We will however include relevant information below when the chosen simulation tool is described.

2.4 Simulation Tools

A fter looking at network simulation and the discrete event-driven approach in particular, we now turn to the task of selecting a specific simulation tool to use in our research. First we discuss our selection criteria, followed by giving an overview of the main alternatives considered and relate them to the criteria. The next section will present our chosen simulation tool in detail.

2.4.1 Selection criteria

A vailability and implementation cost is perhaps two of the most important criteria for a student researcher[1]. With limited time and resources, several tools will automatically be out of the scope. This holds for most of the proprietary solutions since we do not have a budget to acquire licensed software. Moreover, writing a quality simulator from scratch for the purpose of this thesis is not within the timeframe given for this project. The timeframe also places restrictions to the amount of modifications that can be done to an existing simulator before it is ready to use in our experiments.

T he availability criteria brings us to the portability criteria. Campus computer facilities do not easily support locking computer resources to a specific student. The distributed system call for general resources being provided by the system, and particular resources stored as files in the students home directory. This division also ensures that the simulation environment can be easily transferred to project supervisors as well as other researchers. The latter property also supports reproducibility and enables others to test our results.

T he implementation cost criteria leads to the expandability criteria. The simulation tool must be expandable in order for us to implement and integrate the proposed Protocol **P** . This brings us to the last three criteria, which are partly interleaved: Level of control, level of detail (abstraction)

[1] and in the industry in general

and point of focus. We need to have full control over the parts of the simulator affecting the outcome of our research questions. In particular, the tool must support frame level abstraction as described in section 2.2.1 above. The tool must provide detailed information about what happens inside the data link layer, and enable us to control switch layout, packet scheduling, buffer management, routing and other network interface specific tasks. Protocol stack, topology and other network parameters are also of interest, but we do not put much emphasize on it since these are supported by most network simulators by default.

O ne of the things that separate simulation tools is their point of focus. It can be a particular protocol layer, a specific protocol or implementation of such, multiprotocol interaction, a transmission technology and so on [19], [8]. We need to make sure that the chosen tool in fact supports our link layer focus and Ethernet standards.

L ast but not least is the usability criteria. It is important that the user interface is easy to work with, preferably with a familiar programming language to do the implementation. Development and simulation must also be feasible given the available computer resources. In addition, graphical interfaces should not be of such nature that for example extensive usage of a pointing device leads to health problems.

2.4.2 Considered alternatives

F rom the previous discussion, our selection criteria can be summed up as availability, implementation cost, portability, expandability, level of control, level of detail (abstraction), point of focus and usability. With these criteria in mind we look at the main alternatives we considered for carrying out our experiments; OPNET, in-house alternatives, NS and J-Sim.

OPNET

T he first considered tool was the OPNET Modeler, which is a graphical based hierarchical editor. At different depths in the hierarchy it is possible to specify network components in close to any topology. At the bottom is finite state machines and programmed code, in for example C, that specify how components in the above levels should behave in specific situations. When the model has been defined, simulations can be run and data collected.

T he OPNET Modeler is proprietary software, but free licenses are available for academic usage. This tool comes with several included libraries for different protocols and architectures, but we would be required to implement a switch model suiting our needs as well as in detail implement Protocol **P** . In other words, using the OPNET Modeler in our case requires a tedious and time consuming implementation process. This contradicts our implementation cost criteria.

A ll though the OPNET Modeler meets the availability criteria, it does not score high when it comes to the portability criteria. Moving the work to say a home computer is not trivial due to licensing. Nor is it trivial to publish all parts of the simulation environment.

T esting of the OPNET Modeler revealed that it is hard to work from a remote computer over a network connection, cause in this case the graphical editor becomes intensely slow and unresponsive. The OPNET Modeler may also bring health problems related to the intense usage of the pointing device. However, the compelling reason for turning down the OPNET Modeler in this project was the anticipation of high implementation cost and difficulties with developing models that meet all the requirements of our research problem.

In-house alternatives

A fter turning down the OPNET Modeler, we looked at miscellaneous in-house alternatives. Research communities in the field of interconnection networks commonly have one or more dedicated simulators written locally. The distribution of such simulators are however often restricted to the community it was written, hence the term in-house. This contradicts of the availability and portability criteria, especially when it comes to ensure reproducibility.

W ithin our research community (www.ifi.uio.no and www.simula.no) we have such in-house tools. This alternative would reduce implementation cost compared to doing a implementation from scratch for this project. However, available documentation is scarce and it is anticipated that it would require a non-neglectible amount of time to obtain a full understanding of the tools in order to be able to expand the code to meet the requirements placed by our research topic and questions. We therefore turn down this alternative for the same major reason as given for the OPNET Modeler, and in addition due to the closed nature of the simulation tools.

The Network Simulator (ns-2)

M otivated by our critique of the previous alternatives based on the availability criteria, we turned our focus toward the Network Simulator (ns-2)[75], which is freely available as well as well-known and popular within research communities. This discrete event-based simulator is a part of the VINT project[76] funded by DARPA and run as a collaboration involving USC/ISI, Xerox PARC, LBNL, and UC Berkeley.

I nitial review of ns-2 revealed a good match for our availability and portability criteria. Its widespread usage and popularity also speaks to its advantage. However, its point of focus turned out to be the deciding factor ruling ns-2 unusable for our needs. Specifically, "Ns provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks."[75]. In other words, this simulator does not provide the required data link layer granularity. It can at most simulate a LAN by link layer protocols, MAC protocol and physical channel[2], but this provides at best a interface level abstraction. Due to the diverging points of focus, we do not see a way of incorporating Protocol **P** with and obtaining our desired test results from the Network Simulator.

J-Sim

A fter turning down ns-2 and continuing our search we finally found a simulation tool compatible with our selection criteria: J-Sim[86]. This simulator is freely available, is widely used for network simulation in academic circles, particularly among students, comes with solid documentation, founded on well-known design principles from the autonomous component programming model, written in the Java programming language and is "a truly platform-neutral, extensible, and reusable environment" [86]. It has been shown in comparative experiments that J-Sim has better scalability than the ns-2 simulator[87].

J -Sim easily meets our criteria of availability, portability, expandability and usability. The experimental setting can readily be shared simply by exchanging Java files and setup scripts (written in Tcl). In addition, the component architecture provides the levels of control and abstraction required. Trace mechanisms let us tap into the system and monitor focus points relevant to our research.

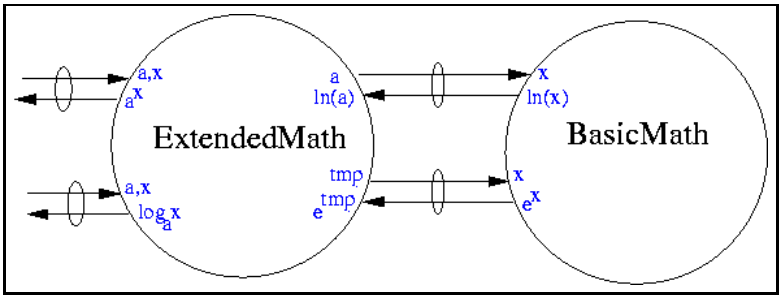

Figure 2.1: OO class relationship

O ur implementation costs are limited to modifying existing components, developing additional components for specific features and setting up concrete simulation scenarios. The component architecture greatly reduce the task of tailoring the link layer of an existing simulator to our needs. As the chosen simulation tool, J-Sim is described in detail below.

2.5 J-Sim

W e start the presentation of J-Sim by describing the autonomous component architecture (ACA) emphasizing the design motivation for J-Sim, its basic concepts and how this is implemented in Java. Second, we look at the abstract network model used, in which the core service layer plays an important part. The entire presentation is based on documentation found at www.j-sim.org, the Ph.D. thesis of the software architecture designer[87] and also own experiences with the tool. Illustrations herein are used with the written consent of Hung-Ying Tyan.

2.5.1 The Autonomous Component Architecture (ACA)

Motivation for ACA

T he technical motivation for Hung-Ying Tyan, the software architecture designer of J-Sim was the fact that software fails to achieve the same modularity as hardware, and the desire to make a network simulator that mimics the principles of integrated circuit (IC) chip design. Such chips interface with the surroundings exclusively by pins, not very different from the way objects interact with each other in the object-oriented programming paradigm.

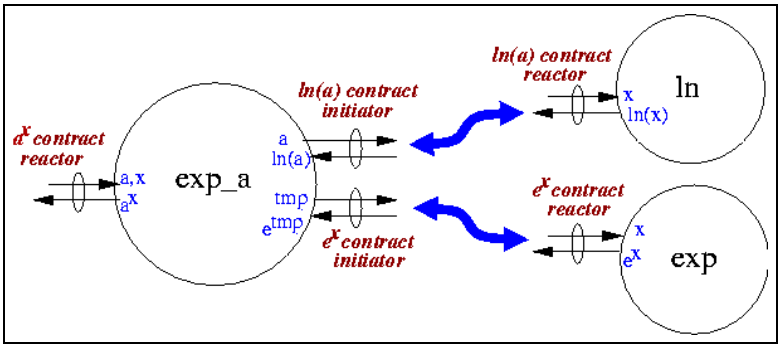

Figure 2.2: Three components and the contracts they are bound to. Thick lines between the contracts indicate the contracts are matched to each other.

T he relationship between two object-oriented classes, ExtendedMath and BasicMath is shown in figure 2.1. The claim of J-Sim's inventors is that "software design cannot achieve the same level of modularity as IC design [..] because the object-oriented programming paradigm is fundamentally different from hardware design in component binding"[84]. The desired relationship between the components of figure 2.1 above is obtained by separating contract binding from component binding, as illustrated in figure 2.2.

T o understand the illustration of figure 2.2 we need to examine the key concepts of ACA, including component and contract.

ACA basic concepts

C omponents and ports are the main concepts of ACA. Their relationship, and the parallel to IC chips are shown in figure 2.3. Ports reside within components and are connected through wiring. Causality of information exchange at port or component level is regulated by contracts. We elaborate on these concepts in the following paragraphs.

C omponent is the basic entity of ACA. As figure 2.3 shows, endpoints are referred to as ports, corresponding to pins on the IC chip. Through encapsulation *composite components* can be built. We have found that viewing the system in different degrees of encapsulation can be directly transferred to the abstraction levels previously discussed. Figure 2.4 illustrates encapsulation. Parent component denotes the enclosing component, whereas child component(s) denotes the components that are encapsulated.

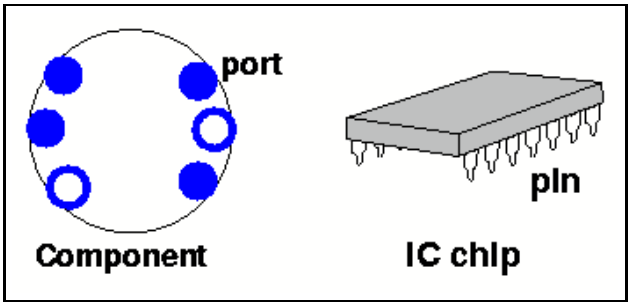

Figure 2.3: Analogy between an IC chip and a component

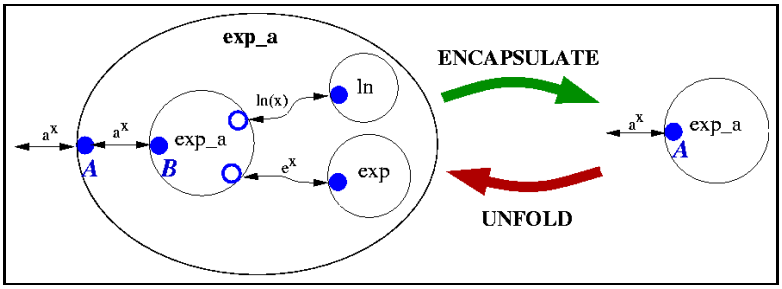

Figure 2.4: Encapsulation of the three-component system in 2.2

A s the name 'autonomous component architecture' suggests, components form a hierarchical system. The autonomous part of ACA comes from component and contract binding. Getting there requires us to look a bit closer at ports and wiring.

C omponents are connected to each other through their endpoints (ports) more or less in the same way as cabling physically connects stations and switches in an interconnection network. A port has both an input and an output wire. Whether the other end of those wires are bound to the wires of one or more other ports dictate if the port is able to send and / or receive data.

W iring is done by connecting the output wire of one port to the input wire of one of more other ports. Whether or not this joining is mutual dictates if the connection is simplex or duplex. By combining different wiring scenarios it is possible to get one-to-one, one-to-many or many-to-many connections. In our concrete protocol implementation we will exploit this to obtain the link layer architecture desired.

As noted above, contracts regulate the causality of information exchange at port or component level. Figure 2.2 showed contract initiators and reactors. A contract specifies how these fulfill a given task. A further detailed explanation of ACA concepts is out of the scope of this thesis. Interested readers should refer to [87] for a detailed description.

ACA implementation in Java

The ACA was implemented in Java by the creators originally under the name *JavaSim*, and later renamed to *J-Sim* due to trademark restrictions. This section gives an overview of this implementation emphasizing on the simulator engine. We also include showing how J-Sim fits with our selected discrete event-driven approach.

J-Sim is "a real-time process-driven simulation technique that fits naturally in ACA"[87, pg. 1]. Through describing the execution model of ACA we show how this technique was designed extending the discrete-time event-driven simulation approach.

In the independence execution model data are handled in independent execution contexts, giving the ACA its autonomous property[87]. In other words, simultaneously-arrived data are also processed simultaneously and independent. This calls for synchronization when accessing shared data. In addition, J-Sim also provides a function-call execution model used for send-receive operations among components. How we have complied to these execution models is documented in 6.

J-Sim makes use of *JVM Java Threads* to provide independent execution contexts[85]. For performance a background thread manager called *Runtime* is used in the ACA implementation. Figure 2.5 illustrates how Runtime is involved in delivering data between two components C1 and C2. Each component represents a separate execution context. Consequently, it is the responsibility of Runtime to create a new context for C2 when it hands of the data.

Runtime is the composite of two component classes, `WorkerThread` and `ACARuntime`[84]. The former is a wrapper for the Java Thread class adding features needed for the execution context models. The latter is used for managing the `WorkerThreads` for the purpose of boosting performance. Thread scheduling will not be an direct part of our work, but knowledge of

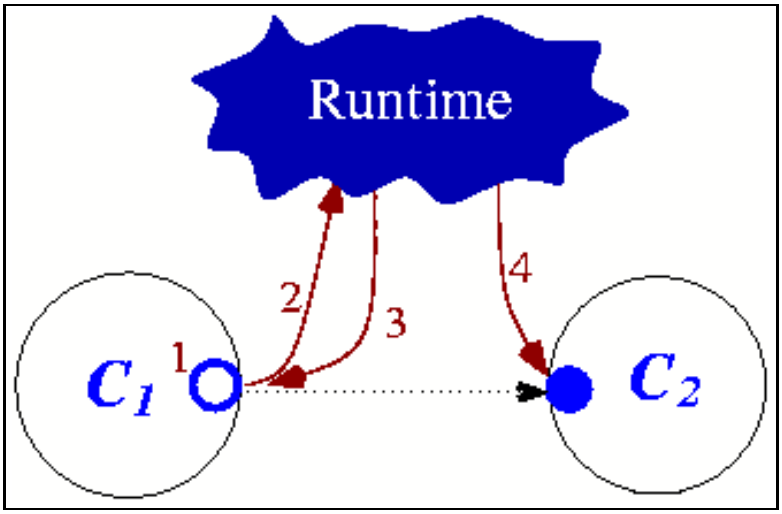

Figure 2.5: How the runtime handles data delivery

execution contexts and thread scheduling is important to ensure the integrity of shared data such as buffers and flow control parameters.

> Discrete-time event-driven simulation is actually a special case of real-time process-driven simulation.[87, pg. 28]

C ompared to the discrete event-based approach, in which events happen in a sequence at discrete time points, the event execution of J-Sim substitutes the discrete time points with real time. This makes the network simulator mimic real network scenarios closer[87].

T o accomplish this, the following three variables are used within ACARuntime: `last_time_updated`, `time_scale` and `time_advances`. These variables maps the simulation time to real wall time and will be important for us when specifying simulation parameters for the experiments.

T he current simulation time is calculated as the difference between current wall time and `last_time_updated` divided at the sum of `time_scale` and `time_advances`. [87] claim that by setting the ratio between real and simulated time (`time_scale`) to infinity and limiting the total amount of execution contexts to one, the simulation becomes "sequential [..] discrete-time event-driven simulation".

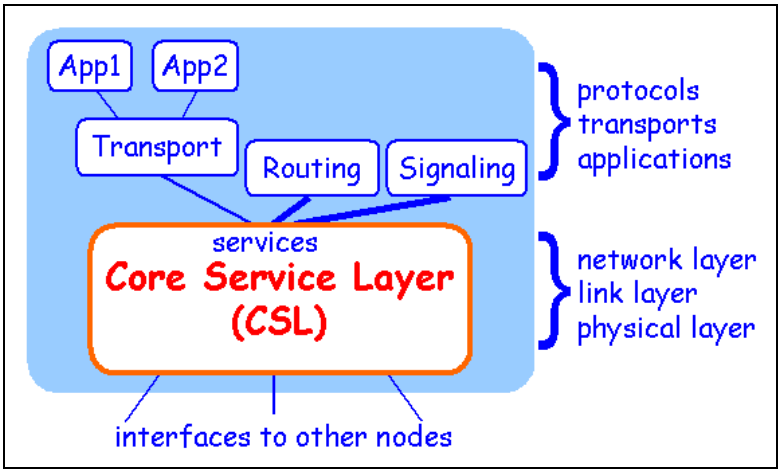

Figure 2.6: The internal structure of an INET node

2.5.2 Abstract Network Model

T his section introduces the abstract network model built on ACA. The building blocks of this model was abstracted from the Internet, and this background gave the model its name: *Internetworking Simulation Platform* (INET)[85].

W e start by examining the core service layer, which will play an important role in our study. Thereafter we look at how INET is implemented in Java, focusing on component classes and the class pyramid relevant for our usage of this simulator.

Core Service Level (CSL)

T he *core service layer* (CSL) includes only the most fundamental services like data forwarding/delivery, identity lookup and routing, and packet filter configuration. These services are defined as contracts. On top of this layer protocols and applications are put, like illustrated in figure 2.6

T he data forwarding/delivery service handles exchanging data between the CSL and the upper layer protocols. In other words this service dictates how packets should move up and down the protocol stack. The identity service maintains a list of addresses of other nodes in the network, and its lookup and configuration contracts makes the identity service related to the routing service.

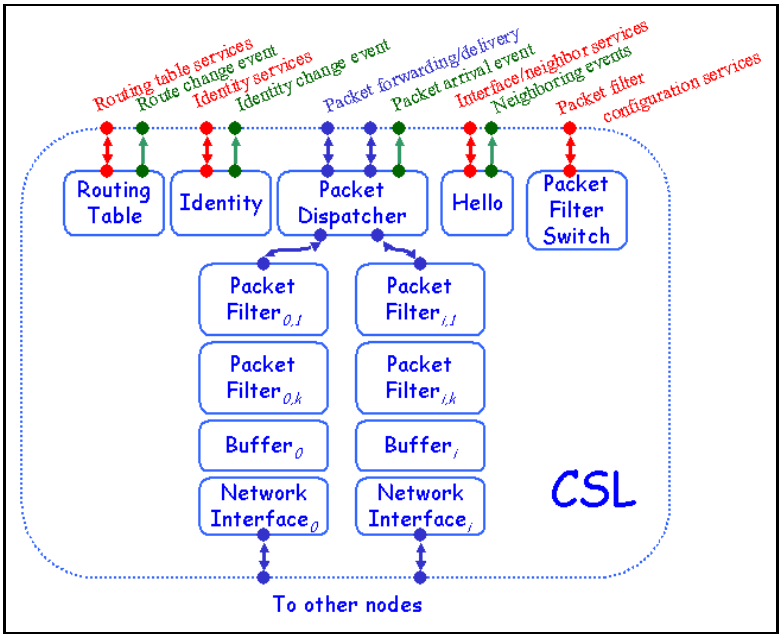

Figure 2.7: The decomposition of the core service layer

T he route lookup contract of the routing service uses both source and destination address as well as incoming interface to select the appropriate outgoing interface. There is one routing table per node. We have not found support for finer granularity route lookup in the INET documentation[85].

I n each node a collection of packet filters are available. These filters are the extensible part of CSL[85]. Serially connected, a subset of the packet filter pool act as outgoing and incoming interfaces for the node. In other words, when a packet arrives at an incoming port on a node, it will pass through one or more packet filters before it is handed off to the dispatcher. This is illustrated in figure 2.7.

T he CSL also have an interface/neighboring service handling information about the interfaces of the node and its adjacent neighbors. We will treat this service as a black box, simply a feature of the simulation tool.

T he CSL is decomposed into components related to the services, as illustrated in figure 2.7. Note the packet dispatcher component and the series of packet filters. These components, and modifications to them, will be our main focus for this thesis.

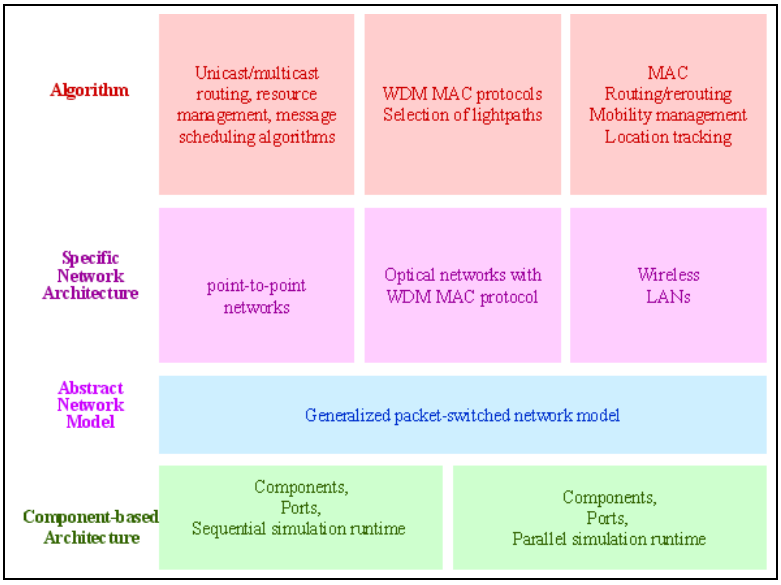

Figure 2.8: A possible module stack using the abstract network model

I t should be noted that the sequence of packet filters "not necessarily mirror the way in which the Internet protocol stack is layered"[85]. This gives us the flexibility and tool to adopt the desired frame level abstraction as described in 2.2.1 above.

T he abstract network model is laid out on top of ACA, and can be part of a module stack as illustrated in figure 2.8. How components in this model are implemented and organized is described in the following section.

INET implementation in Java

J -Sim includes an INET implementation in Java in which each component corresponds to a Java class. These component classes together with other J-Sim components form a class pyramid as illustrated in figure 2.9. It should be noted that the internal structure of INET follows a client-server model in which CSL is the server and the upper layer protocols act as clients[87].

O ur focus lays within the NET and INET layer. After we have presented the theoretical foundation leading up to our implementation of Protocol **P** we return to this component class hierarchy to show exactly how our work integrates into J-Sim.

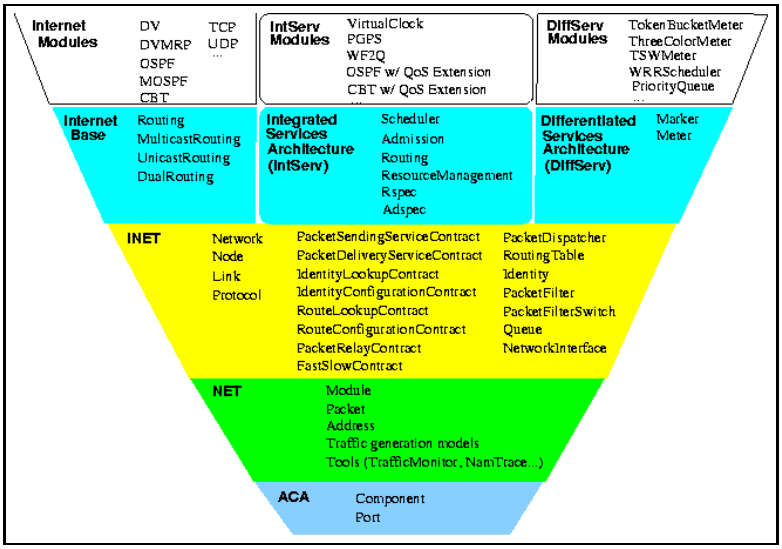

Figure 2.9: The class pyramid in J-Sim

2.6 Terminology summary

T his section shortly recaptures the most important concepts from this chapter. Frame level simulation abstraction captures detailed resource usage at a frame-by-frame basis and requires detailed modeling of mechanisms such as buffers, switching and arbitration. Injection process dictates the traffic pattern and in general the workload that is put on the system.

S imulator design can be either cycle-based or event-driven. Based on our selection criteria we arrived at J-Sim as the simulation tool to be used for our experiments. This is an extended version of discrete event-driven simulation referred to as real-time process-driven simulation. J-Sim uses independent execution contexts and a real-time to simulation-time ratio to accomplish this.

T he Autonomous Component Architecture makes it easy to modify and extend J-Sim. The Core Service Layer is an important part of the abstract network model and acts as an server to upper layer protocols and applications (clients).

Chapter 3

Switching, topologies, deadlocks and routing algorithms

D eadlocks are the main focus of this chapter. We aim to give a thorough review on the issues relevant for our study, and in particular store and forward deadlocks. These are related to both the switching technology and the packet forwarding (routing algorithm) used, as well as the network topology. The overall purpose is to establish a foundation for exploring one of our research questions: To what extent do Protocol **P** apply to solving deadlocks in general, or is it limited to handling store-and-forward deadlocks? Detailed knowledge of Protocol **P** itself is also needed, and for that reason we postpone conclusions to chapter 7.

R ecall from chapter 1 that a *topology* is the description of the physical (or logical) layout or arrangement of edge and intermediate nodes and channels in a network. Moreover, the phenomenon of *deadlock* is an actor / resource situation in which some resource is held by an actor and needed by some other actor, and either direct or indirect these are connected in a cycle in such a way that external intervention is needed to resolve it.

W e start with a quick overview of different switching techniques, emphasizing the store and forward approach, followed by elaborating on the interconnection network classification introduced in 1.2.3, and considering some issues related to bridge operation. With this necessary background established, we turn our attention to deadlocks. In particular we will distinguish between routing deadlocks and store and forward deadlocks. A

small section on livelocks is also included because it applies to one of the properties of Protocol **P** to be studied.

T he last part of this chapter deals with routing algorithms, with emphasis on the spanning tree protocol and strategies that adopt some form of that protocol. Our approach is to identify the key idea as well as critique voiced in the literature, applicable to these strategies.

3.1 Switching

O ur first step in examining deadlocks is to provide a brief, and by no means exhaustive, background on switching methods. Deadlock resolving approaches defined for one swathing method is not necessarily applicable to others. The presentation is based on [35].

3.1.1 Circuit Switching

I n circuit switching a physical path through the network from sender to receiver is reserved by sending of a setup message containing destination address and some control information. After this setup phase where the source decided the complete path, data can be transmitted, and will always follow the same route through the network from source to destination until the line is removed either by break or explicit termination. Even if the setup needs to be buffered at the nodes until the entire message has arrived, this does not hold for the data flow, and hence this method does not require as much buffer capacity at each node compared to packet switching. In short, circuit switching is a fixed path, reservation based data flow approach. It corresponds to opening a pipeline from location A to B with a reasonable assurance that whatever you insert in one end will show up in the other end, for example hooking up oil supply pipe from the well to the refinery.

3.1.2 Packet switching

I n contrast to the circuit switching approach, the first bytes of each packet will in the packet switching method contain routing and control information, eg. involves some overhead per packet transmitted in the network compared to the pure payload. Each packet must here be stored completely at each intermediate node before forwarded to the next. The buffering will introduce delay proportional to the distance (number of hops) between sender and receiver. Because of this trait, this method has been named *Store*

and Forward (SAF) switching. In line with our above oil analogy, this corresponds to pumping the oil into trucks labeled with the destination refinery and providing the driver with a road map and route descriptions.

T he communication links will be fully utilized as long as there is data to send, because multiple packets can be in transit at the same time. The total buffer requirement can be reduced by using shared queuing (SQ) instead of pure input queuing (IQ), pure output queuing (OQ) or input/output queuing (IQ/OQ), but in either case the requirements will commonly exceed that of circuit switching. The store and forward approach consequently tab into the field of buffer management. Solving problems like head-of-line (HOL) blocking is an important part of this management. Another problem is store and forward deadlocks, which will be addressed later in this chapter.

3.1.3 Virtual cut-through and wormhole switching

R ather than waiting until the whole packet has arrived before examination and forwarding, the packet header can be analyzed immediately at reception so that forwarding can start as soon as a route has been selected and the desired output port is free. *Virtual cut-through* (VCT) switching will reduce the packet delay provided that there is enough capacity to handle the arriving traffic flow. In cases of packet contention in the nodes, there is however a need to buffer a few complete packets, as the transmission of these units must be atomic[1].

W ith *wormhole switching* packets are pipelined through the network, where these are broken into smaller flow control units to reduce buffer requirements. In other words, wormhole switching also follows the cut-through principle, but with a finer granularity than the VCT approach. A message can hence stretch over multiple nodes, and complicates the mechanisms to guarantee that deadlocks does not occur. In addition, different messages cannot be intertwined over the same link at the same time, an ability that calls for *virtual channels* to work. Since buffers usually are handled as FIFO queues, there is a danger of HOL blocking.

B y letting a physical channel supporting multiple logical channels, or virtual channels, it is still possible to avoid blocking by disconnecting physical channels from buffers and letting multiple messages share the same

[1] all-or-nothing approach to task

link at the same time. In case of two such virtual channels multiplexed on the same physical link, each of them will experience the situation as if they were alone at the link, but at the cost of half the bandwidth available. Due to this degrading of speed for each split into even more virtual channels, the sum obtained increase in performance will be smaller. In addition, the delay experienced by each packet increases, and, virtual links need dedicated buffers which increases the cost and complexity of the switch.

3.2 Topologies

I nterconnection networks can, as we saw in 1.2.3, be classification as shared-medium, direct, indirect and hybrid networks[35]. This section, as does the original source, uses graph theory to further describe these networks.

L et us start by establishing some key graph theory concepts; *node degree*: number of connections between a node and adjacent nodes, *diameter*: the highest number of hops (or some other physical metrics) between a random pair of nodes in the network. If all nodes are of identical degree, the network is said to be *regular*. Equally, seen from a random node, if the network resembles the same picture, the property symmetry holds.

T he perfect network would be a single crossbar switch with NxN ports connection all N processing nodes of the network. There will always be a balance between performance and cost, as to have all nodes being strictly adjacent to every other node is a tad unfeasible to do in real life. Direct and indirect networks share the point-to-point connectivity aspect. In the latter switches act as mediators between communication nodes. Network adapters interface with the network, connecting nodes to *ports* at the switches. Because processing and switching are done in separate nodes, the distance/diameter parameter will be +2 in these networks, compared to direct networks.

D irect networks include well-known topologies as mesh, torus and hypercube, but also various tree topologies fall herein. Indirect networks consist of regular topologies, such as crossbar and Multistage Interconnections Networks (MIN), and irregular topologies.

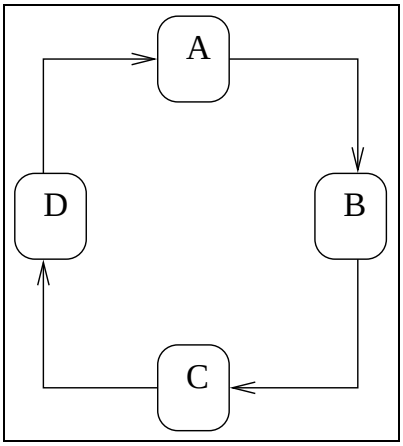

Figure 3.1: Network loop scenario, nodes A-D form a cycle

T he last of the four classes in [35] is *Hybrid Networks*, and hold network topologies that deviate from or combine the schemes above. One such network is the bridged shared LAN in which segments of shared-medium LAN are connected in a hierarchical way. Further structuring into a switched dedicated LAN places such a network in the indirect irregular class. That is, if one does not argue that Ethernet should remain classified as shared-medium network despite that the arbitration mechanism is disabled.

F or this thesis we adopt the view of micro-segmented switched Ethernet belonging to the indirect irregular class of interconnection networks. The following section on bridge operation builds on this view.

3.3 Bridge operation

A bridge, better known as a layer 2 switch, relays frames between its ports [67]. This relay is based on data link layer information, like the 48 bit unique network device address. In the following, we assume Ethernet Store-and-Forward operation unless otherwise noted. Classical Ethernet with a true shared broadcast medium employed *filtering* operation on the destination address contained in arriving frames to prevent broadcasting duplicates back in the shared medium it came from. Point-to-point switched LANs are not prone to this duplicate creation, however without proper filtering a frame can end up being ping-ponged back and forth, generating extra load, consuming resources and not getting closer to its destination.

A s long as end nodes do not need to be aware of the intermediate switches, those switches are said to be *transparent* [67]. Unicast frames are much less dangerous in this scenario than multicast frames, as the latter tend to get forwarded around forever if filtering /forwarding is malfunctioning. So what happens if filtering and forwarding is ok, and the tables are sound as well? Add a few links and you have a network loop scenario at your hands, as shown in figure 3.1.

3.4 Deadlocks

> buffer capacity is finite [..] A deadlock occurs when some packets cannot advance toward their destination because the buffers requested by them are full[35, pg. 83].

A s stated in the chapter introduction, deadlock is an actor / resource problem of direct or indirect mutual dependency. We have come to the conclusion that the literature examined on deadlocks in interconnection networks follow one of two approaches; the *connectivity* approach or the *buffer management* approach. The former is characterized by emphasize on routing algorithms and cycle breaking strategies [66, 64, 18, 58], while the latter is characterized by store and forward scenarios, smart buffer allocation and flow control [6, 43, 28, 30, 47]. In other words, the connectivity approach deals with *routing deadlocks* whereas the buffer management approach deals with *store and forward deadlocks*.

U nfortunately, there is also a gray area between these. For example [43, pg. 923] is using performance as criteria and state that "backpressured networks that do not allow packet dropping [..] are susceptible to a condition known as deadlock in which througput of the network or part of the network goes to zero (i.e., no packets are transmitted)".

R esearch reports and other sources do not necessarily explicitly state which kind of deadlock they describe, and this constitutes a problem when it comes to interpreting and using the material. We believe this has contributed to some misconceptions about the proposed Protocol **P** , a claim we return to later.

W e start by outlining three well-known strategies for deadlock management: prevention, recovery and avoidance [35]. Thereafter we exam-

ine routing deadlocks, and finally we explore store and forward deadlocks. Livelocks are examined in the following section. Specific routing algorithms are covered in section 3.6.

3.4.1 Prevention, recovery and avoidance

G enerally there are three ways of dealing with deadlocks, namely prevention, recovery and avoidance. The distinguishing feature is how resources are allocated. Prevention involves some form of resource allocation prior to transmission so that a packet never get stranded enroute unable to get served. This dedication however can lower overall performance, as the requests rarely utilizes what s granted fully and for the whole duration of the reservation[35]. It is like setting up roadblocks shielding the route from downtown Manhattan (NY) to the airport Friday afternoon rush hour to make sure a single car doesn't get caught in traffic jam along the path.

A voidance on the other hand requests resources enroute, advancing one step at the tine making sure not to make choices leading into dead ends. In many ways this corresponds to not driving into a traffic light regulated road junction until you are positive that you can clear the junction before the lights turn red.

T he optimistic alternative is recovery, in which the behavior is to comply to any request, keep an open eye and if (and when) a deadlock occurs, reallocated resources so that the problem is solved. As this often include dropping one of the offending packets, a property we intend to avoid in our target system, recovery will not be pursued as an option.

O f the above three approaches, only the avoidance option is suitable for our Ethernet context. The rationale for this is the fact that we are dealing with data link layer protocols with only point-to-point connections, and consequently the protocol cannot rely on end-to-end reservation schemes. It should however be noted that [43] uses the term 'prevention' although Protocol **P** is designed to work on a hop-by-hop basis. While this initially seems to contrast with the conceptual difference between 'prevention' and 'avoidance', it will be evident from the buffer management strategy of **P** described in section 5.4 that classifying it as a preventive strategy is justified. We now move on to routing deadlocks.

3.4.2 Routing deadlocks

Definition 3.1 *Deadlocks may appear if the routing algorithms are not carefully designed.[18, 842].*

B reaking channel dependencies seems to be the most used approach to routing deadlocks in the literature: "When a packet is holding a channel, and then it requests the use of another channel, there is a dependency between those channels"[35, 90]. A condition for deadlock free routing was presented in [18] by stating properties for the routing function including the ability to deliver all packets and the non-existence of cyclic dependencies.

T he Spanning Tree Protocol (STP) specified in IEEE 802.1d is one of the most well-known algorithms for cycle-breaking. STP and successors such as *up*/down** routing[66] and *Tree-Based Turn-Prohibition*(TBTP) [58] are described in section 3.6 below.

3.4.3 Store and forward deadlocks

> Store-and-forward deadlock refers to the situation in which there is a set of buffers, all of which hold messages waiting to be forwarded, and these messages can be forwarded *only* to other buffers of the set.[47]

T his mutual buffer dependency is inherent to the store-and-forward packet switching scheme introduced earlier in this chapter. The surveyed material is surprisingly coherent on this type of deadlocks, for example [28], [6] and [30] use definitions very similar to the one quoted above. There is also consensus in that buffer management is a reasonable approach to handling the problem.

S tore and forward deadlocks can be direct or indirect[30]. The former case is also known as head-on-collision due to the fact that it involves packets stuck in buffers of adjacent nodes. Indirect SFD involve at least three nodes that form a cycle. For the latter to occur, it is our view that the network topology must contain cycle(s) and following that the routing algorithm used is not deadlock-free. In other words, we choose to view direct SFD as true store and forward deadlocks, and indirect SFD as routing deadlocks requiring dependency breaking measures.

T o sum up some of the buffer strategies preceding Protocol **P** the journey start with the work of K. D. Günther and the GMD-net protocol[62, 26, 47, 30]:

> The problems of deadlocks and flow control are handled by separate but cooperating means: A structured buffer pool is used against deadlocks whereas a two level dynamic window mechanism provides flow control.[62]

F or decades this strategy has matured and offspring can be found in for example [13] and [29]. We refer interested readers to the literature, as a full survey is out of the scope for this thesis. Protocol **P** and its advanced buffer management scheme is presented in chapter 5.

A s we shall see when we examine flow control in detail in the next chapter, there is also a potential risk of *flow control deadlock* in which a set of nodes mutually halt each other. The specific flow control scheme will decide whether this halt is permanent or temporal.

3.5 Livelocks

L ivelocks are a phenomenon related to deadlocks. Both describe a situation where some packets never reach their destination, however in the livelock case, they are not solidly blocked, rather stuck on a trail in a roundabout without being allowed to get to an exit[35]. This can be the result of unfair scheduling, or even the bi-product of a deadlock prevention strategy[43]. Observing this is typically done by looking at the throughput and comparing the amount of injected traffic to the amount of drained traffic (possibly accounting for packet drops).

W e sum up the topic of deadlocks and livelocks with the following definitions from [43]:

Definition 3.2 *A network is defined to be* deadlock-free *if, given an arbitrary combination of packets sitting in its buffers, the delivery of each packet to its destination is guaranteed within a finite time, provided that there are no new packet arrivals to the network.*

Definition 3.3 *A network is defined to be* livelock-free *if, given an arbitrary combination of packets sitting in its buffers and an arbitrary pattern of new arrivals into the network, the delivery of each packet to its destination is guaranteed within a finite time.*

3.6 Routing algorithms and packet forwarding scheme

A great portion of the deadlock handling mechanisms relate to routing algorithms in some way, as indicated in section 3.4 above. A taxonomy using number of destinations, routing decisions, implementation and adaptivity (in that order) is presented in [35]. Routing algorithms are of secondary interest in this thesis, and we therefore limit our focus to the classical spanning tree protocol, up*/down* routing and Tree-Based Turn-Prohibition. We start by looking into the spanning tree protocol below.

3.6.1 Spanning Tree

I n contrast to the difficult task of managing a loop-free topology manually, a protocol is a better choice. LANs widely adopt a vendor independent loop resolution protocol, known as the *Spanning Tree Protocol*(STP). Originating at DEC [82], this protocol was later standardized as IEEE802.1D [2]. The desired goal of a spanning tree is the properties of a natural biological tree: a root sprouting into branches that divide into smaller branches and finally end up in leaves. All arcs of the tree are connected direct or indirect to the root, and no branch ever grows (joins) another branch, it just splits.

A LAN has exactly one root node, however it can be logical rather than physical. Bridges are identified by a 64 bit unique identifier, which is the concatenation of the MAC address of one of the bridge ports and a 16 bit priority field enabling tree management. Similarly, ports are identified with 8bit port number plus a 8bit priority field. The spanning tree protocol use these identifiers to determine the root and mark bridges and ports as designated[3]. Determining the root is done by an election algorithm, which selects the lowest identifier.

I t is assumed that the network does not change more frequently than the algorithm can stabilize the tree, and that it will not be constantly performing reconfigurations. The algorithm is simple, but has some severe limitations; Because all active redundancy is prohibited, no load-sharing can occur and hence, frequently visited parts of the tree can become bottlenecks. Sequential delivery and non-duplication are guaranteed by a standalone LAN, and the single path , non-loop property of the STP maintains this invariant. The physical topology may offer more optimal paths between a given source-destination pair sometimes, but the non-optimality is a trait

[2]The standard includes a definition of STP in C language

[3]responsible for forwarding data toward the root

we must accept to avoid loops. Being a natural *congregation point*, the root is a severe bottleneck, unless topology is configured and resources allocated to balance this congregation. In conclusion, the spanning tree protocol is a *link prohibition algorithm*[58] meaning that entire links are pruned off with respect to data transmission.

3.6.2 up*/down*

T he IEEE802.1d spanning tree has for the reasons given above frequently been criticized in the literature, and more sophisticated versions have been suggested[66, 64, 58].

C lassical up*/down* routing was presented in [66] for Autonet. The key defining property is the assignment of direction to links based on a spanning tree over the given topology in such a way that channel dependencies are broken. 'Up' denotes the direction toward the root. Allowing for all links to carry traffic, a simple routing rule has to be followed: "a legal route must traverse zero or more links in 'up' direction followed by zero or more links in 'down' direction"[64].

A lthough the up*/down* routing scheme has less restrictions than the classical spanning tree, it is not optimal and excludes some of the minimal paths[43, 64]. Measuring performance, [5] has found that throughput depends on assigned direction of links.

C ycles are still broken in the up*/down* scheme, but in contrast to classical STP using link prohibition, this scheme is an (early) *turn-prohibition*[58] scheme through its direction assignment and traversing rule. Especially J. Duato has published allot of work on enhancements to the up*/down* scheme. Recently, a more sophisticated turn-prohibition algorithm has been introduced, and we close this chapter with an introduction to this.

3.6.3 TBTP

T he paper [58] describes the Tree-Based Turn-Prohiqbition algorithm. In essence it exhibits the same characteristics as the up*/down* scheme. The algorithms differ on which (and how many) turns to prohibit. TBTP is more scalable, introduces fewer routing restrictions and guarantees that less than 1/2 of the turns in the network will be prohibited.

Chapter 4

Managing congestion - the art of flow control mechanisms

R eturning to the classic analogy of cars, roads and traffic jam[25, 17], flow control refers to traffic lights regulating road intersections. Moreover, assuming an input queue based network node, buffer layout will then refer to the number of lanes dedicated to traffic destined for the various exits of the intersection. In its simplest form, an intersection has at least one lane for all arriving traffic. And, upgrading our road to a highway, it is common to differentiate between transit and on(off)-ramp traffic meaning that if you have succeeded in getting *in* then you have some form of advantage over those that have not.

T his chapter explores the field of congestion and flow control in terms of a literature review. Recall the controversy from chapter 1 on the meaning of these terms. The main purpose of this chapter is to address one of our research questions, namely *what is the current state of congestion and flow control approaches in general, and backpressured store and forward packet switched networks in particular?*

W e start off looking at congestion as a resource sharing problem, and move on to examining a well-know taxonomy for congestion control algorithms, followed by identifying different properties control schemes exhibit. Unless specifically noted, this first part uses 'congestion' as a general term including flow control, and it is not coupled to any specific network technology. In particular, the role of feedback (explicit or implicit information sharing) is of special interest to us, as this plays an important role in Protocol **P** .

T he second part of this chapter is restricted to the data link layer, leading up to and exploring the IEEE 802.3 MAC Control in detail, including performance, alternatives and improvements reported on this protocol. In addition to answering our research question, this material lays the foundation for the in-depth exploration of Protocol **P** in chapter (5).

4.1 Congestion - a resource sharing problem

C ongestion in computer networks is in essence a classic demand versus supply dilemma. Controlling congestion dictates that these two entities are kept in balance. In addition, a congestion control scheme must have low overhead, be fair and responsive, work in bad environments and maximize the overall performance [37].

> Congestion control is concerned with allocation the resources in a network such that the network can operate at an acceptable performance level when the demand exceeds or is near the capacity of the network resources. [37, pg. 24]

T he resources referred to here by Jain are channel bandwidth, buffer space and processing power. A similar view on congestion is held by [16], and emphasizes that it is the saturation of these network resources that leads to the congestion state with performance degradation. Moreover, [35] defines congestion as "the state where the offered network load approaches or exceeds locally available resources designed to handle that load".

N ot all communication networks are buffered, and depending on the switching technique (see section 3.1) adopted, these three resources are more or less relevant. Because our focus is on backpressured store-and-forward packet networks, attention will mainly be given to buffer management and hence the size and organization of queues are an important issue.

J ain [37] recognizes congestion as a *dynamic problem* which can be solved by either increasing the resources or reducing the demand to balance the equation. The algorithms relevant for our study fall into the latter category, as the majority of the ones applied in packet switched store-and-forward networks. Demand reduction schemes are basically service denial, service degradation or scheduling based[37], and examples of these will be given throughout this chapter.

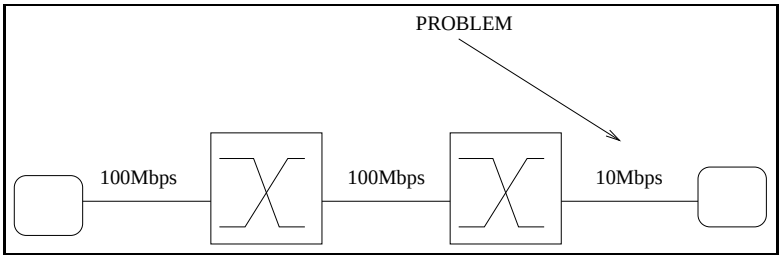

Figure 4.1: Rate mismatch

4.1.1 Rate-mismatch and traffic aggregation

B andwidth itself is not a critical issue, but, when network links of different capacities are combined in a topology creating *rate-mismatch* between two subsequent links on a data path, or when the load and topology result in traffic *aggregation* at some point in the network, bandwidth becomes an important factor of the resource puzzle and do often lead to congestion. [72] points out that "the real problem is often a mismatch between parts of the system"[pg. 385]

4.1.2 The relevance of buffer space

T he relevance and size of buffer space is intriguing and even frustrating. Memory can be either too scarce or too plentiful. As seen in figure 4.3, having to drop a packet before it enters memory due to lack of space, and having to discard it after it has left memory because it has waited too long, both result in the loss of a packet. As pointed out in [37], the latter can be more harmful because resources in that case are consumed and then wasted.

4.1.3 Processing power

P rocessing power being the third congestion related resource, might create bottlenecks, and hence contribute to congestion, if not able to keep up with packet arrivals. We will not pursue this parameter in this thesis.

4.1.4 Policies that affect congestion

V iewing congestion control as a resource sharing problem it is clear that design choices affecting either the demand or supply of resources might

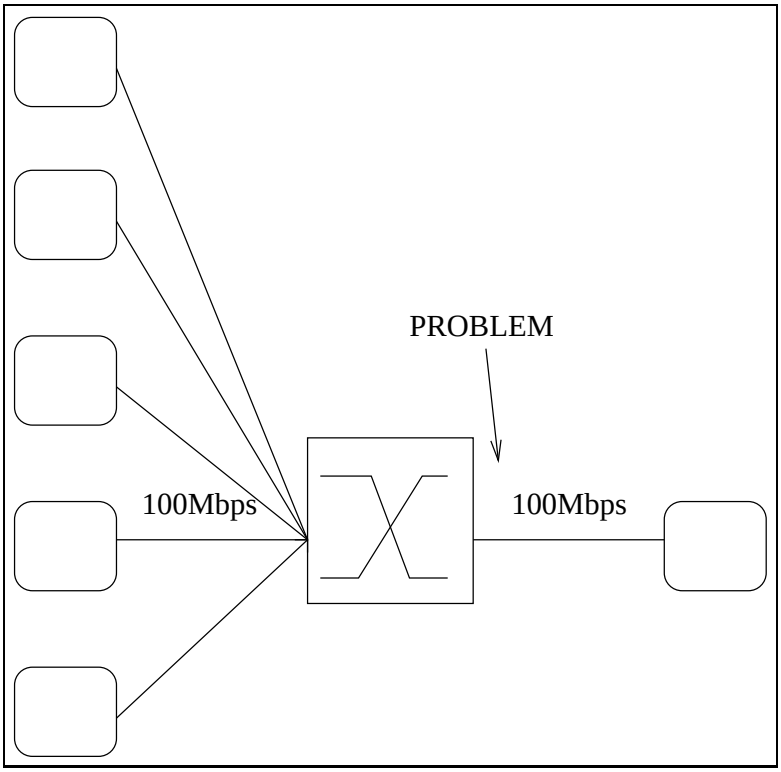

Figure 4.2: Link aggregation

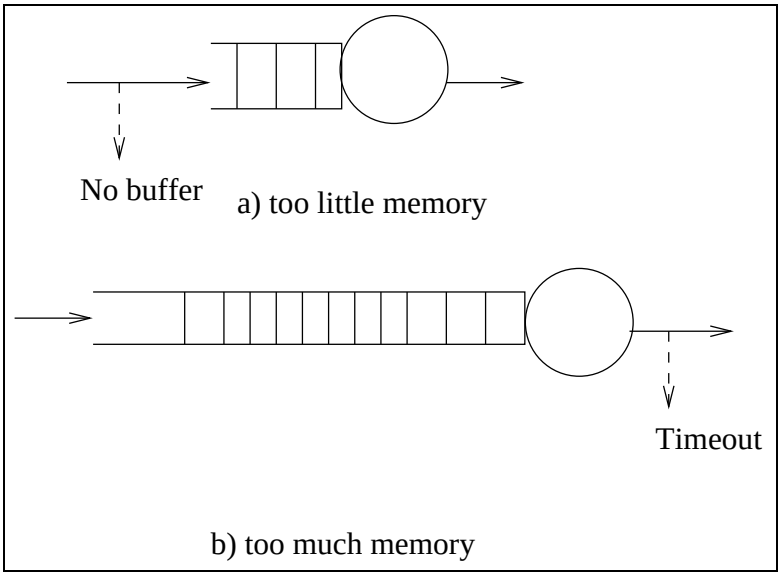

Figure 4.3: Memory problems leading to packet discarding in the cases to little and too much memory

contribute to tip the weight scale toward or away from congestion. Table 4.1.4 presented in [37] and adopted by [72] summarize such contributing policies.

1. Network layer:
    - Connection mechanism
    - Packet queuing and service policy
    - Packet drop policy
    - Packet routing policy
    - Lifetime control policy
2. Transport layer:
    - Round-trip delay estimation algorithm
    - Timeout algorithm
    - Retransmission policy
    - Out-of-order packet caching policy
    - Acknowledgement policy
    - Flow control policy
    - Buffer management policy
3. Data link layer:
    - Data link level retransmission policy
    - Data link level queuing and service policy
    - Data link level packet drop policy
    - Data link level acknowledgement policy
    - Data link level flow control policy

Policies related to switching and routing

T he distinction between connection-oriented and connectionless networks have already been made in chapter 1, and the presence of a reservation protocol to limit access to the network and enforce that accepted data can and will consume at most its allocated share of resources, is the key difference. In Virtual Circuit (VC) networks it is common to adopt a admission policy related to the reservation protocol [72]. Route selection and path splitting were covered in chapter 3.

Buffers and packet drop policy - milk or wine approach

P acket queuing and serving policies are related to buffer layout and scheduling algorithm, and are covered later in section 5.4. Packet drop policy follow buffer layout and is commonly adopted at the receiving entity in response to overflow. Dropping a packet results in immediate relief and can be used to clear buffers in addition to provide implicit feedback. When using packet drops to provide load shedding, the *milk or wine - policy* [72] applies.

Timing and delay

L ifetime control, network round-trip delay (RTT) and timeout intervals affect the reaction time of a scheme and are themselves affected by queuing policies at intermediate nodes that can make a packet grow severely old while sitting in a queue at a (congested) node. Packet retransmission and acknowledgment strategy dictate the consequences of a packet drop and the feedback delay cycle respectively.

Flow control policy

F inally, *flow control policy* (i.e. window-based or rate-based) at the transport layer depending on "the bottleneck resource at the destination"[37, pg. 28] are also considered when designing a congestion control scheme.

4.2 A taxonomy for congestion control algorithms

A now well-known theoretical framework for congestion control algorithms has been offered in [16]. The taxonomy in figure 4.4 is based on control theory and differentiates between open loop and closed loop control. At the leaf nodes of the classification tree, existing control schemes can and have been placed. We refer to [16] for classification of well-known existing algorithms not covered in this thesis. Note that this taxonomy is a framework for classification, and a specific algorithm may fit into more than one category.

4.2.1 Open loop control schemes

O pen loop control schemes solve congestion mainly by good design and are not dependent on network state[72]. Moreover, because of the de-

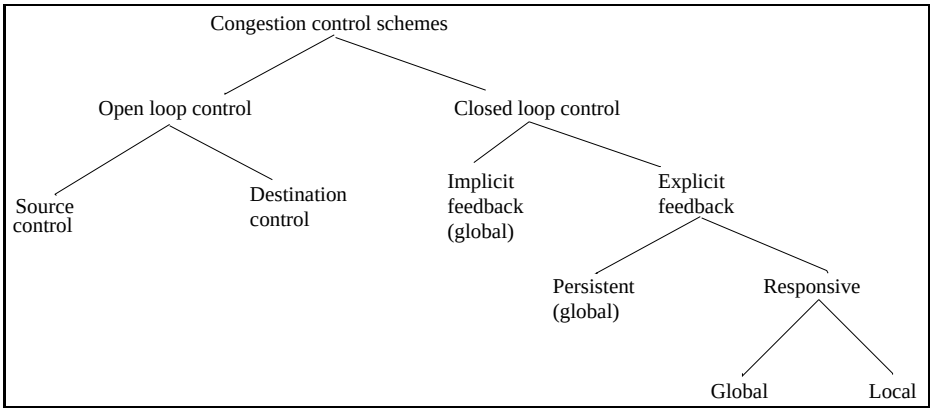

Figure 4.4: Taxonomy for congestion control algorithms

coupling from network state, which imply lack of feedback, open loop algorithms are inherently rate-based[56, 38]. This decoupling leads to the following attributes: First, the control decision is not dependent on feedback from congested spots. Second, no dynamic monitoring of network is required. And third, control agent, either at source or destination, uses local knowledge of network in the decision making process.

T he open loop category is subdivided based on whether it is the source or the destination node that performs the control. *Source control* implements some form of admission policy "that stabilize the traffic arrival process" [16, pg. 39]. Algorithms in this category include the input buffer limit model [45] and stop-and-go policy [27]. *Destination control* is mainly some form of (selective) packet discarding, because once a packet has entered the network, a destination node has few other options, just like a glass of water being filled until it eventually overflows.

4.2.2 Closed loop control schemes

C losed loop control schemes solve congestion by dynamic monitoring of the network and issuing either implicit or explicit feedback to the source. This category has also been termed *credit-based* control [56] and *window-based* control [38]. A range of feedback mechanisms exist, and a summary can be found in [37], including explicit feedback messages, feedback as part of routing messages, rejection of excess traffic, probe packets and feedback fields in packets.

Implicit feedback

I mplicit feedback is inherently global in nature, since the whole network with intermediate nodes between the node pair in question is involved, in contrast to local information only involving adjacent nodes. Because the feedback information is not transmitted, network state is deduced based on local observation on factors like delayed acknowledgment on packets, arrival rates and timeouts. Most schemes here are window-based, as is the *slow-start* [71] algorithm deployed as part of the TCP congestion control scheme.

Explicit feedback

E xplicit feedback can be sent either as separate control messages or by piggybacking data packets. This category is divided further based on whether the feedback is available constant or it is triggered by certain events, termed *persistent* and *responsive* respectively. The subdivision into global/local, as described for implicit feedback, applies here to the nature of the feedback. Persistent explicit feedback is refreshed periodically; if done on a hop-by-hop basis the algorithm adopts local control. Responsive explicit feedback commonly involve some threshold parameters related to queue-length that trigger feedback in response of traffic conditions. Global algorithms matching this classification are *source quench* [61], *choke packet* and *rate-based congestion control*. The source quench scheme has in addition a variation placed in the local responsive category.

B oth the DECbit and Qbit scheme are based on a warning bit set in the header of data packets passing a congested node. The DECbit method monitors the percentage of ACK with the warning bit set, and adjust the transmission rate accordingly. Choke packets are used in datagram networks reducing the rate of traffic entering the network. A control packet is generated at a congested node and travels against the flow toward the source. The choke principle can alternatively be applied by tagging data packets and let that info return to the source via ACK form the destination. If relying on end-to-end response is too slow to be useful by the scheme, a hop-by-hop version is available. This tend to give a quick relief at the congested node but consume more buffers upstream.

4.3 Control scheme properties

C ongestion control and resource allocation are in [59] described along tree dimensions: router centric versus host centric, which are covered below in section 4.3.4, reservation-based versus feedback-based and window-based versus rate-based. This is a slightly different approach than the taxonomy given in the previous section. In order to work with a broader perspective, we acknowledge these dimensions and outline several control scheme properties below.

4.3.1 Credit vs rate based schemes

W hether a scheme controls rate or credits/window size is mainly connected to the open loop - closed loop classification in the taxonomy, but exceptions exist. Credit-based control depend on some form of feedback and works on a per link per window basis with the receiver issuing credits to the sender. In this manner, storage of excess traffic is distributed in the network, and more packets are stored inside the net during congestion compared to rate-based schemes. However, the latter tends to center the storage at a single point whereas the former distributes it over more switches [55].

R ate-based schemes suffer from large overhead when used in short and frequent congestion situations, and function more optimal during infrequent and longterm congestion [55]. In general, schemes like leaky bucket and the *additive increase, multiplicative decrease* algorithm controlling the rate are limited to admission control[38], reduce input traffic to the (sub)network , and are never used at the link layer[72].

4.3.2 Active vs passive schemes

T here are two ways to react when encountering a situation, the passive and the active way. This is true for congestion as well. Passive schemes are preventive by nature and commonly implemented during the design phase. Active schemes on the other hand are reactive and are triggered in the response of congestion indications [55]. The latter method involves estimating the network state and informing the sources to reduce traffic. This avoidance vs. recovery classification criteria is quite common in the literature [16].

T he Transmission Control Protocol (TCP) for instance uses a combination avoidance and recovery mechanisms, implemented by the DECbit scheme and the slow-start algorithm respectively [59] [1].

4.3.3 Feedback

A simple classification criteria for congestion control algorithms is whether a scheme uses feedback information or not, resulting in *feedback-based* and *rate-based* flow control respectively [72, pg.192]. As the terms indicate, the former method transmits feedback information to the sender, containing some form of status indicator or a permission to transmit data. The latter method has mechanisms built into the protocol / network interfaces that put a restriction on the data rate without the need to exchange feedback information.

R ate-based flow control is rarely used at the link layer [72]. In order to provide feedback, the network is required to monitor load and then take remedial action providing state information to some control point, and the feedback frequency should match the control frequency [37]. The picture is somewhat more complex than this, and the described taxonomy gives a better and more complete view.

4.3.4 Control point

T he noted controversy of the terms 'congestion control' and 'flow control' is best addressed by looking at the points where control is implemented and applied. We have already seen in the preceding section on taxonomy that some algorithms are global whilst others are local, some apply control at the source or the destination, others at the intermediate nodes. In the current section this issue is related to location in the OSI protocol stack as well as to whether control is limited to a single point or involve the entire network, or somewhere in between.

Scheme location in the protocol stack

D ifferent policies affect design of control mechanisms at different layers, as seen in table **??**. Data traffic originates in the end-systems of a network, and with the packet conservation principle described in section 4.3.2

[1] additional algorithms contribute to the total TCP congestion management package

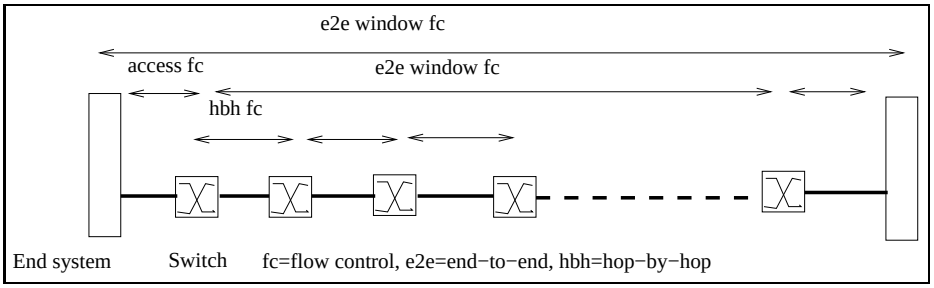

Figure 4.5: Control points, difference between end-to-end, hop-by-hop and access flow control

in mind, these hosts are in the best position to regulate network load, and commonly do so by adopting dynamic window schemes [37].

T he network layer has two types of control point, and may apply different schemes at these locations. First, *network access* exercises some form of admission policy, exemplified by the input buffer limit scheme [45]. Second, intermediate routers (and gateways) can act upon congestion by service degradation to greedy sources. Schemes like fair queuing, buffer class and leaky bucket are well suited to enforce these restrictions.

F inally, at the lowest layer capable of handling congestion, data link level flow control applied at each hop in a backpressure manner can be an efficient mechanism to control congestion. Different network and congestion scenarios call for different solutions, and despite strengths and shortcomings in the control schemes discussed, one has to consider the intended use in order to pick the one best suited. Moreover, multiple schemes often coexist in a system to handle a variety of situations at different control points. Some rules-of-thumb have been made, and are described below after a brief review of the network subsets involved.

End-to-end or hop-by-hop scheme

F igure 4.5 illustrates the network subsystems involved in the different control schemes, and as can be seen, the subdivisions follow the structure imposed by layers in the protocol stack described above. The figure is inspired by [55], but the idea of using control point level as an analytic measurement dates at least back to [25]. Both source-to-destination and entry-to-exit control are termed *end-to-end (flow) control*(E2E) [55] despite that they are applied at different layers. End-to-end flow control cannot guarantee that resources are available at intermediate nodes, only at the

destination [67]. *Hop-by-hop (flow) control* denotes node-to-node (switch-to-switch) control and are applied at layer 2.

M oreover, layer 2 flow control can in addition be viewed as not targeted at congestion primarily, rather as a technique to ensure reliable transmission over a single link with acknowledgment, retransmissions and timeout as tools. *Positive Acknowledgment and Retransmission* (PAR) algorithms [67], also known as *Automatic Repeat Request* (ARQ) algorithms [59], implement some form of sliding window scheme that ensure reliable delivery, preserve packet order and yield flow control.

T he *Sliding window protocol* is a simple form of feedback-based flow control in which ACK and timeout serve as feedback signals [72]. The *stop-and-wait* protocol is a minimal version of sliding window using a window size of only 1. To prevent lost or delayed packets from interfering with normal operation , some form of sequencing indicator is needed in the packets. Lack of a timely ACK will trigger halt of the source (and maybe retransmission). These algorithms mainly address allocation of the buffer resource part of the congestion problem, and this is why the data link layer in packet switched store-and-forward networks only uses feedback based flow control, where the sender is given permission to send more data, and rate-based control is rarely seen[72].

Source or router centric scheme

S ource-centric or router-centric control, represented by schemes as slowstart, CUTE, DECbit on one side, and on the other side Qbit and random drop, fair queuing, or backpressure respectively[38], is a question of protocol layer and the type of congestion it aims to handle. Source-centric control use network layer for feedback and transport layer for rate/traffic reduction.

4.3.5 Conservation of packet principle

J acobson introduce the *conservation of packet - principle*: "for a [TCP] connection 'in equilibrium' [...] the packet flow is what a physicist would call 'conservative': A new packet isn't put into the network until an old packet leaves"[36, pg. 1]. Congestion is hence controlled by identifying and managing points in the network where this principle is violated. Variations

of this principle can be found at the basis of most congestion control algorithms, modified by the degree the *connection in equilibrium* can be applied.

4.3.6 Protocol interaction

A s part of the TCP/IP suite used in the Internet, TCP has received a great amount of research interest in the field of congestion control. One of the lighthouses along the road have been the work of Van Jacobsen ([36]). S. Floyd and R. Jain have put in remarkable contributions as well. The *Random Early Detection* (RED) algorithm [22] and the DECbit scheme [39] have both been proposed for use in TCP/IP networks. Both calculate average queue length, and then mark (or drop) one or more packets to signal the congesting source. The difference lies in that the former require only one marked packet to trigger control action, where as the latter relies on a fraction of packets being marked.

T CP performance has been studied in detail in the IP/Ethernet context [54, 53] as well as in the ATM context [57], hot-spots of TCP processing identified [23], and interaction with lower-layered backpressure mechanisms examined [57, 52, 79]. Moreover, inter-working of switched Ethernet and ATM flow control operating below TCP has been studied, concluding that these flow control mechanisms are complementary [4]. And, a hop-by-hop rate-based scheme have been proposed as an alternative to TCP at the transport layer [51].

4.4 Flow Control

A fter our exploration of congestion as a resource sharing problem, the well-know taxonomy and different properties of control schemes we are now done with the general survey of congestion and move on to data link layer flow control in particular. This section addresses general concepts, followed by IEEE 802.3 MAC Control in the subsequent section. We start by strengthening the conceptual difference between 'congestion control' and 'flow control'. Second we examine flow control symmetry and some buffer management schemes. Finally, we look at the concepts of 'on/off', 'hop-by-hop' and 'backpressure' in the link layer flow control context.

4.4.1 Congestion control or flow control?

As a rule of thumb, "the longer the duration, the higher the layer at which control should be exercised" [38, pg. 17]. It follows that router-based flow control is best up to handling short-term congestion, whereas in order to cope with long-term congestion source-based control schemes should be applied to reduce the overall load entering the network.

It might seem like the current dominant answer to the congestion or flow control question can be put like in [72]: Congestion control is a global issue involving all nodes whereas flow control relates to point to point traffic between a specific source-destination pair. This distinction gets a bit clouded however when hop-by-hop flow control are applied throughout the entire network reaching all the way back to the sources by backpressure giving a total effect much like the global end-to-end scheme. Moreover, "some congestion control algorithms operate by sending messages back to the various sources telling them to slow down when the network gets in trouble. Thus, a host can get a 'slow down' message either because the receiver cannot handle the load or because the network cannot handle it" [72, pg. 386], and this situation is a major reason to confusion.

As we have seen, if attention lays within the data link layer, there is a tendency to reserve the term 'flow control' for reliable transmission over a single link, as in [59], and use the term 'congestion control' for all higher-layer control schemes. Flow control in interconnection networks "dictates which messages get access to particular network resources over time" [17]. Moreover, in the field of interconnection networks [35] do not even mention congestion control and only briefly address flow control as an issue closely connected to buffer management algorithms. Designing LAN switches, the use of (backpressured) hop-by-hop flow control to manage short-term buffer congestion, exceeding the reliable transmission limited view, gets more relevant, as seen in [67].

4.4.2 Flow control symmetry

Flow control can be applied in either one or both directions of a full duplex link. We adopt the following distinction between symmetric and asymmetric flow control from [67].

Symmetric flow control is commonly seen on switch-to-switch links, characterized by a relative uniform traffic pattern, nodes having similar buffer

memory constraint and neither is source/sink of much of the traffic.

Asymmetric flow control results in a scenario where one of the link partners can throttle the other, but not vice-verse. A switch can control an end station reducing the total offered load by pushing back the source of the frames entering the network. The opposite control direction is useful whenever a destination host cannot operate at linkspeed (wirespeed) and needs to borrow buffers at the switch in order to avoid being swamped.

4.4.3 The effect of frame loss

A ccording to [67], higher-level PAR-protocols have a performance penalty on layer 2 packet loss. This is a strong motivating factor driving research on preventing, or at least reducing, packet drops at the data link layer. And it has has been studied over and over again how the TCP congestion control interact with and can contribute from, flow control algorithms applied at lower layers. This is addressed below in section 4.5.4.

4.4.4 Schemes that address buffer management

F low control at the data link layer can be divided into *bufferless* and *buffered* flow control [17], a division that emphasizes the strong connection between buffer management, switching technique and control scheme. By *decoupling* allocation of resources used in sequence, buffered flow control can yield a performance boost.

D ifferent families of buffer management schemes were identified in [25] and include *channel queue limit* (CQL) and *buffer class* schemes. *virtual circuit hop level schemes* constitute a third family, but fall out of the scope of this thesis. The interested reader should consult [25] for a review. The difference between the former two schemes is that in the CQL scheme, arriving packets are distinguished based on the output queue they are destined for, while in the buffer class schemes a *hop-count* is used as the distinguishing criteria. We will later see that the proposed Protocol **P** has traits from both families, and can thus be viewed as a hybrid.

B uffer management can take three forms in relation to backpressure, namely credit-based, on/off and ack/nack[17]. Below we target on/off backpressure, as this is the only one that apply to our simulation study.

4.4.5 On/off hop-by-hop backpressured flow control

T he principle of on/off flow control can be found in Sirpent [12] and Autonet [66]. The latter notes explicit that on/off flow control is not intended or suited to handle long term congestion at the switches.

A control scheme is *source-blind* if it gives the same service to a packet independent of its origin. The counterpart is *selective* schemes. A source-blind on/off scheme can be unfair, because when issuing stop-signal, it affects all traffic from the upstream node, including traffic not contributing to the congestion. Moreover, [51] claims that it might spread congestion and cause oscillating buffers throughout the network.

H op-by-hop flow control have been described in several contexts above. It has a shorter delay in the feedback cycle compared to end-to-end schemes, and is therefore more responsive during short-term congestion and has an advantage in networks with a high bandwidth-delay product [51]. Early hop-by-hop flow control struggled with deadlock problems and unfairness. The former was solved in Autonet, but the latter issue still remain[56] and is coupled to the source-blind versus selective approach.

W hen a hop-by-hop scheme is applied to a chain of adjacent nodes (or links) a *backpressure* effect emerges. The advantage of backpressure is two-fold; first, buffer sharing distributing storage of excess traffic over the upstream nodes[56, 55], and second, feedback may eventually propagate to the edges of the network where sources get notified and can act to reduce the traffic load. According to [53, 3.4] a *back-pressure scheme* is based on the three main components; 'congestion detection' which should be simple, effective and instantaneous; 'notification', which originally do not distinguish at input ports; and 'control actions':

Congestion detection Even if several resources can be oversubscribed, the fact that congestion leads to longer queues for some switch ports is exploited to an easy detection mechanism. High/low thresholds are set on output buffers.

Notification is to ask for control actions to be performed or cancelled. One assumes that output buffered switches does not distinguish between input links. Different schemes for what info to include in the messages sent is a) simple (all), b) CoS-based, c) Destination address based choosing some flows random or all.

Control actions are (un)blocking the traffic vs controlling the transmit rate. Time period to pause vs indefinite time with explicit cancel can be used. They differ in the number of control messages that has to be sent.

W e return to identifying these three components in the specific flow control schemes studied, in their respective descriptions.

4.5 IEEE802.3 MAC Control

I n the following section, familiarity with Ethernet and the IEEE 802.3 standard is assumed.[2]. As we have seen, there is a need for flow control at the Ethernet data link layer which is connectionless, operating on a best effort basis. Because of the low worst case bit error rate (BER) of 10^{-8} [69, Clause 16] [3], recovery from frame corruption is ignored in the data link layer and handled at higher layers [67].

I t has been pointed out that transport protocols adopting end-to-end flow control can only guard resource management at the sender and the receiver, and that (temporary) shortage of buffer at some intermediate node is out of the scope of such a scheme. Hence, in the words of [67, pg. 325]: "If we need to solve a link buffer overflow problem, we must solve it within the link layer".

MAC Control is a generalized architecture and protocol for "real-time control and manipulation of MAC sublayer operation" defined in [69, clause 31]. For backward compatibility, MAC Control is an optional capability in Ethernet, and at the present time, MAC Control PAUSE operation described in [69, Annex 31A and 31B] is the only available control function.

4.5.1 Architecture

M AC Control is a transparent data link sublayer residing between the MAC sublayer and the MAC Control client (like LLC) as illustrated in figure 4.6

[2] a brief introduction were given in chapter 1

[3] This is valid for 10Mbps copper media, while higher data rates and optical media benefit from a BER orders of magnitude better

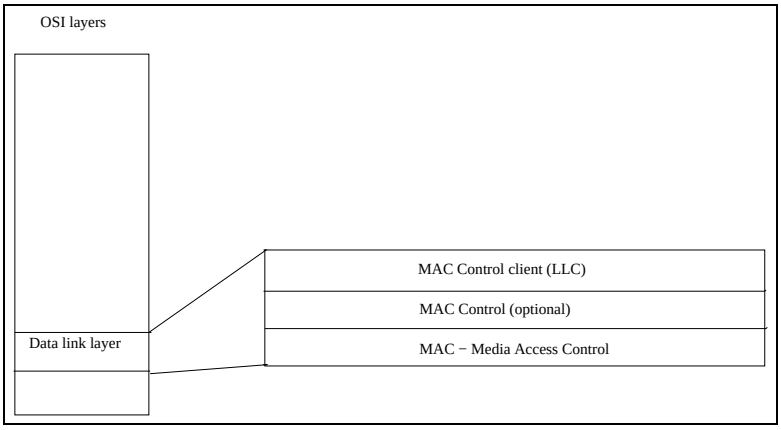

Figure 4.6: MAC Control architecture

MAC Control hence provide additional service to its clients extending the traditional Ethernet MAC. Upon request, the sublayer generates *control frames* transmitted to the link partner by the underlying MAC, much like ordinary data frames. The difference is that these control frames are both sourced and sunk within the data link layer, and never forwarded by the receiving entity.

4.5.2 Frame format

M AC Control frames conform to the format of standard, valid Ethernet frames as described in chapter 1 and in [69, Clause 3], with only the Length/Type field identifier to distinguish them from other MAC frames. This field should have the hexadecimal value 0x8808 to indicate MAC Control. Excluding Preamble and Start-of-Frame Delimiter, a MAC Control frame is exactly 64 byte long, the length of a well-formed minimal Ethernet frame. The first 2 bytes of the data field identify MAC control opcode, with 0x0001 indicating PAUSE operation., followed by opcode-specific parameter(s).

4.5.3 PAUSE function

A ccording to Seifert, chair of the IEEE 802.3x Task Force at the time of writing, the PAUSE function

> is specifically designed to prevent switches (or end stations) from unnecessarily discarding frames due to buffer input overflow under short-term transient overload conditions.[67, pg. 336]

More specific it "is used to inhibit transmission of data frames for a specific period of time" [69, Annex 31B]. This provides a simple stop/start form of flow control for single full duplex links. It is important to notice that PAUSE operation only affect data frames, and that transmission of MAC control frames cannot be inhibited. Moreover, some words of caution of the limits of this scheme might be in place: like other hop-by-hop backpressure schemes is does not solve the problem of steady-state network congestion, neither does it provide end-to-end flow control and does not provide any complexity beyond a simple start-stop mechanism [67].

PAUSE frame semantics

T he parameter list for the PAUSE opcode is short; it contains only the *pause_time* operand which is an 2 byte unsigned int indicating the length of time transmission of data frames should be suppressed. The value of the pause_time parameter is not the absolute time interval, rather it is "measured in units of pause_quanta, equal to 512 bit times of the particular implementation" [69, Annex 31B].

B eside the pause_time and the described type field and opcode, the destination field in the MAC control PAUSE frames have a unique 48-bit multicast address [4].

PAUSE frame - transmit and receive operations

F or an exhausting and detailed specification of the PAUSE operation, we refer to the Transmit and Receive state diagrams in [69, Annex 31B]. Below is an overview of the operation of the receiving and sending side of the MAC Control sublayer respectively.

Parsing received PAUSE frame is done by first checking that it is a well-formed PAUSE MAC Control frame of correct length and valid opcode. Data frames are silently passed on to the next sublayer, while control frame are handled within the MAC Control sublayer. Frames containing unsupported opcodes are discarded. Next, the pause_time parameter is extracted, and the PAUSE function starts a pause_timer of the length (pause_time $*$ pause_quanta). The transmit side of the NI has to be informed (by state variables) to act on the current value of the timer.

[4] the MAC address of the intended recipient of the control_frame may be used as stated in [69, Clause 31]

P AUSE operation always operate on the most recent value of pause_time, meaning that a NI in the not-paused state receiving a non-zero pause_time will enter the paused state. Herein, three events might affect the state. First, the pause_timer *expires* with no new control frame seen leading to the transmit restriction being lifted and the normal operation is resumed in response to the change to not-paused state. Second, the situation at the link partner eases off causing a subsequent control frame with a zero valued pause_time. This works as a *cancel* message, with the same result as the timeout just described. Third, a subsequent control frame with a non-zero pause_time *reset* the timer, with the NI remaining in the paused state with the now described options available.

Transmit PAUSE operation is not required by a IEEE 802.3 NI, but if it does send PAUSE frames, it must implement the above referred to state machine. Within the MAC Control sublayer the TransmitFrame function is called in response of a control request from the client, which may be the housekeeping processor or the queue buffer manager. This function prepares a control frame, which mainly consist of constants: the *pause_command* opcode, the *reserved_multicast_address*, the *802.3_MAC_Control* type indicator and the *phys_Address* of the local MAC. The only variable needed to be inserted is the *n_quanta_tx* specifying the amount of pause_quanta requested. For this reason, [67] suggests that a well-formed PAUSE control frame is kept and transmitted upon request.

A PAUSE control request should be served as soon as the boolean status indicator *transmission_in_progress* is set to false resulting in the transmission of a control frame. This will preempt pending data requests, however not interrupt an ongoing transmission. Hence, control frames are given priority in order to reduce the feedback delay to a minimum. For illustration refer to [67, Fig 8.8]. The IEEE standard does not specify the conditions required to assert and turn off PAUSE flow control, it simply states what the response to such actions shall be for a MAC Control-aware NI. However, some timing considerations are included in [69, Annex 31B] and will be addressed in the following section along with other implementation issues.

4.5.4 Performance studies

M any years have passed since the IEEE 802.3x was proposed, and in this period the MAC PAUSE Control has been analyzed and the performance studied, with emphasis on interaction with, and impact on, TCP [41] and fairness issues [20]. A general trend in the research is the use of relative

small networks just big enough to illustrate some point. This section introduces some of this work focusing on the comments and critique the PAUSE scheme has received. Details on TCP parameter tuning and observed window size are excluded for the clarity of presentation, and the interested reader should refer to the cited papers for a complete description. Modifications [46, 52] and suggested alternatives [88] are then described in the next section.

I n 1997 [41] presented a simulation study of network performance with and without this explicit congestion notification. Simulations were carried out with OPNET and TCP persistent source based on RFC 793 and 1122. The switches used were non-blocking output buffered with one FIFO queue of 64kB per output port, and network links operated at 10Mbps. A asymmetric flow control is applied, with only switches generating PAUSE frames. The switch operates with buffer threshold (400kb) per output port, the authors chose to broadcast pause frames to all upstream neighbors. Observed buffer occupancy was found to have a periodic behavior produced by ACK and RTT. In addition PAUSE created a similar periodic behavior an order of magnitude less than the former. It is concluded that explicit feedback gives a more fair distribution of bandwidth and that the interaction of the two mechanisms results in high utilization of bottleneck links and low loss rates. It should be noted that this work only examined traffic aggregation, excluding link speed mismatch. Moreover, no non-conforming sources were present and only a high water mark for activating flow control were used.

Work by Wechta, Eberlain, Halsall et.al.

T he authors of [79] have published a series of papers on switched LANs, and therein looked at the interaction between flow control at link layer versus higher protocol layers. They claim that the interaction between TCP and hop-by-hop switch flow control is too complex to be solved using theoretical analysis and hence simulation modeling has been used. Because of the impact this work has had on the research field, we will go into some more detail here.

Network model and design is characterized by switches having both input- output- and shared-memory. A high/low threshold is set per input port FIFO queue (16kB). This buffer is filled after buffers at output port and shared memory has been filled. Control XOFF frames are sent with indefinite pause time and explicit XON is required to resume sending in

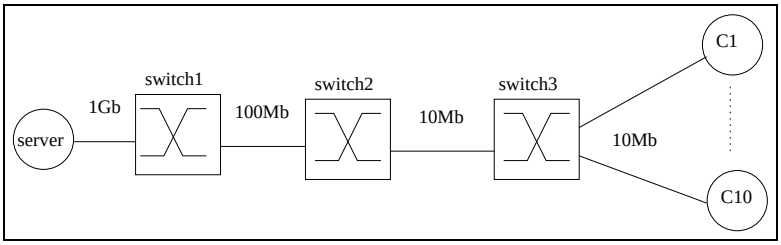

Figure 4.7: Interaction, 3-stage topology

the paused node[5]. Data in transition equals the product of TCP max window size and number of connections. Topology has one server attending to 1-10 clients on the other side of a strait switch chain with one order of magnitude lower bandwidth for each step, see figure 4.7. The server transmits an infinite file over TCP to each of its clients.

Simulation results , e.g. performance, is measured by throughput per link, bandwidth used by a single source-destination pair, and number of packets lost. With flow control on all links, packet drop is avoided and the typical growth curve for TCP window is observed. Buffers are filled faster close to the bottleneck link (L3). Restraining the flow control to only inter-switch links packet drops do occur at the network edges and the connections that avoid drops get better performance. Backpressure cannot propagate information the last step toward the node(s) causing congestion. Because packet drop is random, the different connections do not have identical drop points for the TCP window size, and hence the throughput differs. The harm is however not as critical as we shall see in the next scenario, because all sending processes were in the slow increase phase when the drops occurred. With flow control only on the access links packet loss occurs while the senders are still in the rapid increase, all TCP senders are affected by the drops, and because of bursts in this phase, the probability of drops affecting the same source multiple times is higher. This behavior is identical to the case of applying no flow control at all.

P receding this paper [78] presented a simulation based analysis with a 2-stage topology shown in figure 4.8 designed intended to quantify the effect of head-of-line blocking. The number of clients was varied from 2-10, with client 1 connected by 10Mbps link, the rest of the clients by 100Mbps links, to create a scenario with link speed mismatch. Simulation is done with link layer flow control constant turned on (XOFF triggered and sent to S1), and TCP end-to-end alternatively on/off. With TCP flow control on, traffic

[5]It should be noted that XOFF is non-selective (source-blind)

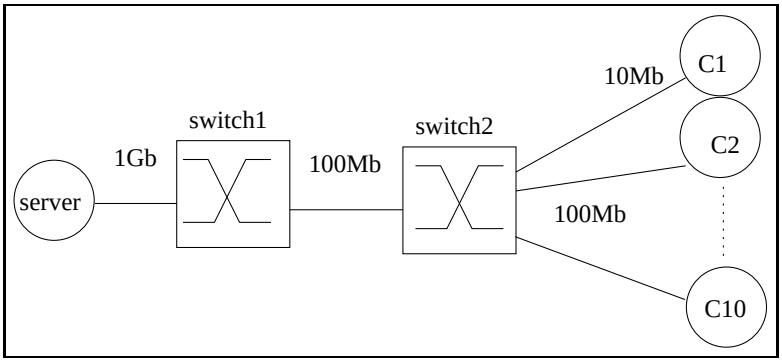

Figure 4.8: Interaction, 2-stage topology

destined for the slow client is limited to its low rate letting the remaining clients share the rest of the bandwidth. Turning TCP control off, HOLB occurs and all connections are throttled down to the capacity of the slowest link. The researchers note that the larger the ratio storage/window size, the lesser the probability of HOLB. They also point to the problems of control frames not being sent directly to the sender. This work has later received criticism for being non-selective (source-blind). However, because of the topology used and the fact that PAUSE flow control is designed for input buffered switches, this should be seen as a critique of the PAUSE control scheme and not the simulation study of it.

L ater that year work on how the choice of topology and transport layer protocol affects performance in a switched LAN, both theoretical and by simulation, were presented in [80]. Topology influence on performance, and the key idea is to locate bottlenecks and replace them. Micro segmentation reduces traffic for each segment and increases the total performance, as seen in figure 4.9.

T he goal of TCP is to reach equilibrium where the window is fully open and the data rate of transmissions is only limited by the presence of bottleneck link(s). However, a TCP window is rarely at its max in LAN context because either the files are too small or buffer overflow leading to packet loss reduces the window. A large window increases the end-to-end delay, but it should not be smaller than allowing maximum utilization of the link. The authors hence suggest using a small TCP window for local connections (IP address) and a larger window for remote traffic. Moreover, regarding the interaction of TCP with PAUSE flow control it is referred to the two previous papers, however, hop-by-hop flow control is only presented as an option with large TCP windows.

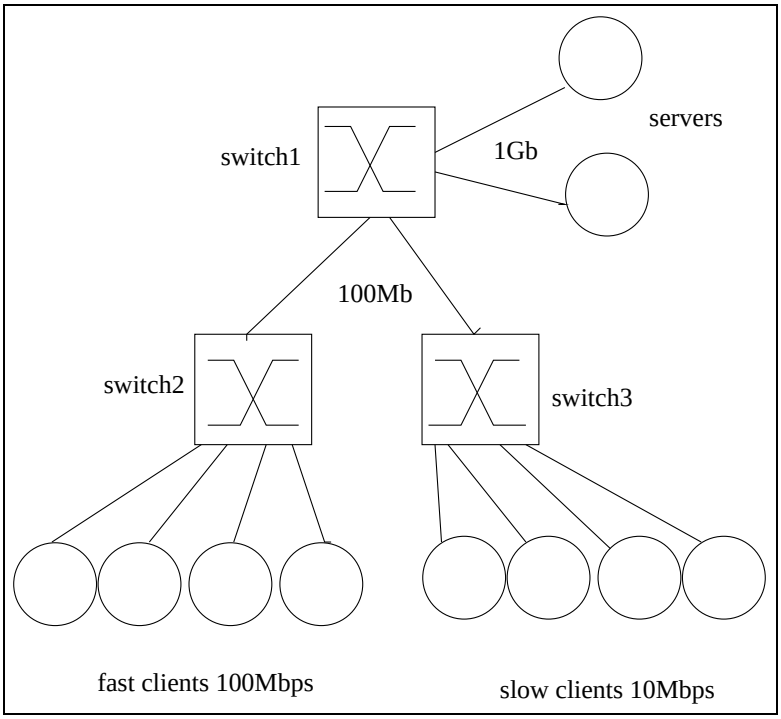

Figure 4.9: LAN configuration with micro segmentation

T he following year another paper ([77]) came continuing the topologies and design from[80] with only minor modifications. For all topologies studied results show that throughput and transfer time are constant, while end-to-end delay increases when TCP window increases, and hence power decreases. Pointing to [78], the authors (of [77]) emphasize that hop-by-hop flow control is responsible for buffer management, whilst bandwidth is controlled by end-to-end flow control (TCP) , and that the two complement each other. Results show that fast links do not get optimal bandwidth because of head-of-line blocking, as the slower links dictate how fast the buffers at the bottleneck empties. As before, a smaller TCP window is suggested to avoid head-of-line blocking and thereby also the throughput degradation for the fast clients. The authors suggest to fine tune the TCP parameters, and have underutilized links for QoS purposes.

T hen focus shifted more toward quality of service with a proposal submitted to the ETT Journal and the paper [81] presenting a modification to 802.3x based on ECN / RED which will presented in the next section.

Addressing the link speed mismatch

In 1999 [20] conducted a simulation study with OOSIM addressing the link speed mismatch problem on the TCP with/without PAUSE scenario as part of an ongoing study. The intention of the study was to compare the loss of bandwidth caused by packet loss to the loss caused by flow control and head-of-line blocking. Referring to [78] and [79] this study claims that the former does not look into the fact that TCP stream suffer from discarded packets. Still, the topology used by [20] is very similar to the one in [78], the only difference being that the former uses multiple servers connected to the first stage switch by links of identical bandwidth to the inter-switch link. Layout and size of switch buffers, threshold levels etc, are not given, however, input buffering can be assumed because of the following sentence: "The flow control scheme monitors the amount of data in the input buffers and [...]" [20] The study concludes that flow control can help performance in homogeneous networks, but link speed mismatch causing head-of-line blocking reduces the throughput more than packet loss. More specifically, fast clients are throttled down to match the speed of the slowest client.

4.6 Suggested alternatives/improvements

4.6.1 Controlling rate on a hop-by-hop basis

The paper [51] presents a selective solution and uses periodic feedback from neighbor switches of buffer occupancy. This congestion control scheme has been frequently cited over the years, and is by several, including [16], viewed as a classic definition of the hop-by-hop scheme. However this takes a lot of calculation overhead, buffers, and extra messages.

4.6.2 QoS extension to IEEE 802.3x

It is important that real-time traffic do not misuse the responsiveness of elastic traffic that have flow control mechanisms (TCP senders), and [81] presents an approach to grant high QoS to real time UDP-based flows in the 802.3 LAN context. It is assumed that the number of real-time flows with guaranteed QoS can be restricted and that these are the last to experience packet loss in times of congestion. The key idea is to create *virtual pipes* through the network for the high priority flows.

The modified scheme is based on the idea to offer two virtually separated pipes in the network; data packets are transferred over the same path, but

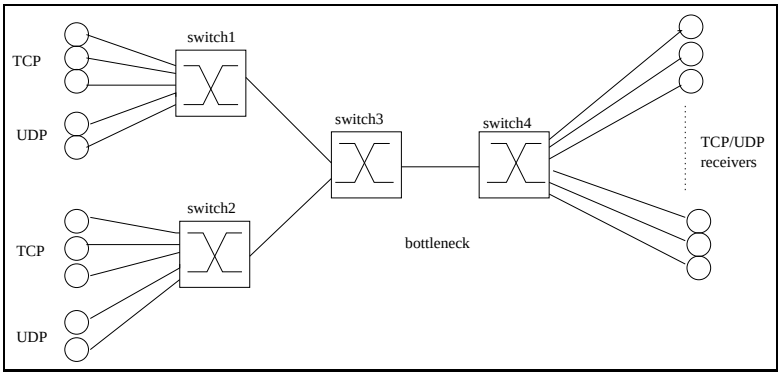

Figure 4.10: Topology used on QoS extension to PAUSE flow control

stored in different queues with high/low priority (2 classes). End-to-end control level uses source quench messages in ICMP packets to the sender, or set the ECN bit in packets chosen by RED method. In relation to bandwidth the latter option is better. XOFF frames are sent only to low priority traffic, and provide short-term immediate relief.

T he four scenarios studied are no flow control, normal flow control to all, modified flow control to low priority traffic, and modified flow control together with ECN/RED respectively, all using the topology shown in figure 4.10. The authors conclude that the hybrid scheme fulfills the same task as no flow control at all in making the source aware of the congestion, and keeping the queuing delays low, but the difference is that is does so without packet loss and without wasting bandwidth. Predicted throughout for UDP were obtained and the inter switch link was fully utilized the whole time. The case with standard 802.3x results in the highest end-to-end delay and significant jitter variations. Still, the case with no flow control is worse for UDP due to the packet losses. The weakness with the third case is that TCP packets are allowed to flood the network unnecessarily so that end-to-end delay gets high. Real-time traffic is no matter still exposed to burst for other real-time flows. ECN demands that the LAN switches are TCP aware, and so doing breaks the layering and demands extra processing.

4.6.3 FLORAX Flow-Rate Based Hop by Hop Back-pressure Control for IEEE 802.3x

T he paper [46] presents modifications to 802.3x based on flow[6] rate to fully utilize the performance in large scale LANs in a fair[7] manner. Addressing shortcomings of the PAUSE flow control scheme found in [81, 52, 53, 41, 20, 78], FLORAX differ from PAUSE flow control by the shift to rate control and the adoption of selectivity in identifying and throttling contesting flows. Hence it leaves the other flows unaffected, and by so doing, aims to evenly distribute bandwidth.

I t is assumed that sender processes in upper layers are told to reduce its TCP window on the reception of XOFF frames. First, the scheme has to identify the congesting flows to avoid unnecessary back-pressure. Second, SLA provisions can be enabled for flows, giving preference to flows that confirms to their bandwidth agreement, and not blindly stop the flow using the largest share of the bandwidth. Third, there is a vulnerability to non-conforming LAN devices in the schemes of standard 802.3x and without source discrimination they might degrade performance for all flows. By selective dropping packets originating from non-responding devices, some resilience can be built into the network.

F or each outgoing buffer, the FLORAX requires the following set of elements. A *Flow Table/List* contains rate estimation and burst related info per flow. Second, a *XOFF Table/List* records the flows currently under XOFF control in order to restore them. Third, *XOFF Control Messages* are sent to invoke control identifying a flow (source-destination pair) and its fair bandwidth associated with an expiration time, whereas *XON Control Messages* restores flows by canceling control action. These messages are triggered by thresholds indicating the need to throttle all, the congesting or none of the flows. Buffer occupancy has to be checked both at frame arrivals and departures, in addition to the calculation of transmission rates for each received frame. In contrast to the original PAUSE scheme, an XOFF message triggers a modification of transmission rate at the upstream node for some flow, not an absolute stop of that NI.

W ith respect to performance, FLORAX and IEEE 802.3x were both found to distribute bandwidth equally between UDP and TCP avoiding the scenario of UDP taking advantage of the responsiveness of TCP congestion win-

[6] defined as a source-destination MAC address pair

[7] Fairness is defined in terms of bandwidth distribution in times of congestion

dow seen when drop-tail buffering is applied [46]. However, FLORAX was measured to have a shorter completion time for file transfers. It should be taken into account that this modified scheme requires additional housekeeping and processing as well as the additional information needed to be included in the control frame.

4.6.4 RATE Control in IEEE 802.3

T he proposed RATE flow control scheme [88] was motivated by the desire for bandwidth allocation enabled in Ethernet in the First Mile (EFM) subscriber services. A simple byte-based leaky bucket at the end node (source) is used to implement the scheme where flows are isolated. Compared to a scenario without RATE control or one with PAUSE control for mis-behaving flows, lesser loss and delay can be seen. It is also interesting to note that this work looks into how RATE and PAUSE can complement each other in a scenario where a switch is RATE controlled and the sending source PAUSE controlled.

I n [88] the authors agree with [52] in that PAUSE flow control should be selective for traffic classes / MAC address and [81] in that sending of PAUSE signals for high priority flows should be avoided. However, both modifications are said to be hard to implement, and because of the need to isolate individual flows in EFM, RATE is seen as a necessary replacement for standard Ethernet in this particular setting.

4.6.5 Selective backpressure

O ne of the perhaps most cited papers on IEEE 802.3 MAC flow control and the call for a selective backpressure mechanism is [52]. This work of Noureddine, occupied with TCP efficiency and the use of link layer flow control to shield TCP from short term congestion packet drops, is found in even more detail in his PhD thesis ([53]). Because of the impact this work has had on the research field, we will go into some detail presenting it below[8].

> Although switched LANs are usually over-provisioned, their characteristics (short RTT, link speed mismatches) lead to increased burstiness, and thus to the occurrence of transient congestion. In order to fully utilize the potential of large switched

[8] However, the presentation will focus on the issues relevant to the link layer, glossing over the details of TCP

> LANs, a link layer back-pressure mechanism may be used to complement end-to-end flow control by handling the short term congestion.

A s seen in the above quote from [52], switched LANs pose a challenge to TCP with respect to enhancing performance. We have already seen in section 4.5.4 that link layer flow control represented by the PAUSE scheme is non-selective backpressure ([41], [78], [80], [81]). Noureddine takes the analyze and simulation studies a step deeper showing that MAC PAUSE Control can result in performance degradation as well as improvement. *short term congestion* is conceptually distinguished from *congestion* in [53, chap. 3] with the former being a temporary condition stemming from link-speed mismatch, traffic aggregation, TCP bursts and multimedia inherent variability, and the latter *chronic long term network overload*[53, pg. 130].

Network model and design used to highlight these benefit and drawback scenarios include full duplex Ethernet links with different data-rate, BSD Reno version of TCP with max congestion window of 64kB and fixed file sizes. Switches are non-blocking output buffered with threshold values at 80% and 70% respectively for sending XOFF[9] and XON frames[10]. For link speeds of 1Gbps, 100Mbps and 10Mbps switch buffers are 1MB, 500kB and 70kB respectively.

Simple back-pressure scheme

W e now return to to some resource management issues introduced at the beginning of this chapter, namely link-speed mismatch and traffic merging. Recall the self-clocking property of TCP running in a steady state. Bursts occurring outside the steady state, as in the slow-start phase, might however cause a temporary over-subscription of buffers at an intermediate node, and [53] showed by simulation that merging of such bursts is harmful and arguments that it should be sought to avoid packet drops due to short-term buffer overflow in this phase. In fact, about 50% of the available throughput was lost when subsequent connections were added to the *link-speed mismatch* scenario shown in figure 4.11. Adding backpressure flow control, the before experienced drop in throughput did not occur keeping performance on a smooth maximum even as more connections were submitted.

[9] pause_time here interpreted as indefinite stop requiring explicit XON for resume

[10] Moreover, it is assumed that the switches do not distinguish between input ports (since they are output buffered)

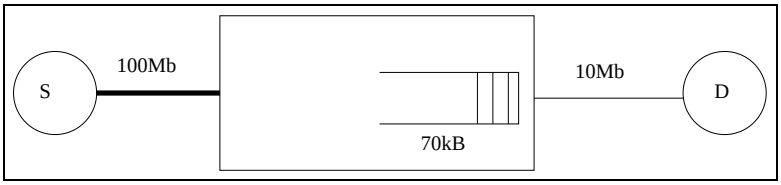

Figure 4.11: Link Speed mismatch

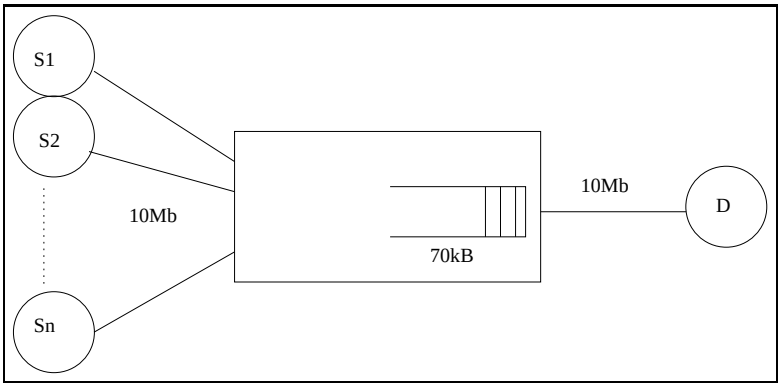

Figure 4.12: Noureddine Topology3

E ven in a scenario without link-speed mismatch shown in figure 4.12 *traffic merging* and the merged bursts of the TCP connections from the different sources contribute to temporary buffer overflow and packet drop at the bottleneck node. The preserving of throughput seen when adding backpressure flow control in the previous case, holds here as well.

T he two above described scenarios can be found in most of the papers by Wechta et.al. (), however Noureddine identifies a third topology, shown in figure 4.13, involving *link sharing* where simple flow control can be beneficial. Because of the big share S1 gets, packet loss can get frequent and hence S1 is heavily throttled without backpressure flow control applied. A variation of this scenario involving a single destination is used for both sources, addresses the issue of drop tail queues showing bias toward bursty sources. In this case backpressure control may be used to handle *fairness issues*. More over, "[by] using back-pressure, congestion can be moved out toward the boundaries of the LAN where it can be dealt with more efficiently"[52, pg. 6].

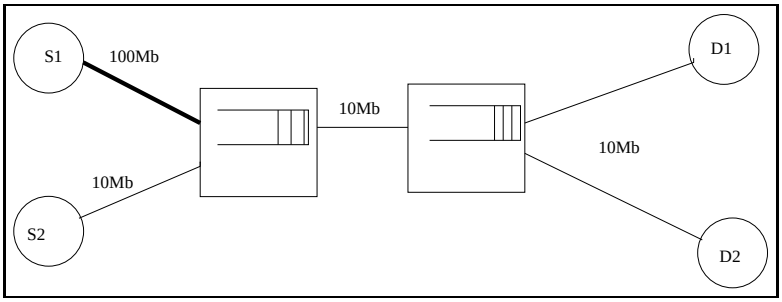

Figure 4.13: Noureddine Topology2

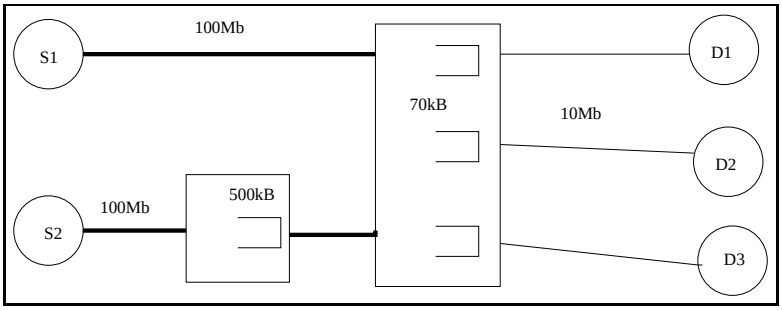

Figure 4.14: Noureddine Topology5

MAC Address Back-Pressure

A s noted, link layer backpressure flow control do in some cases result in performance degrading instead of increase. One such case involves *unnecessary control issues* that might occur in the topology setting shown in figure 4.14. Control actions obtain good performance on the most congested path, but degrades the other paths. The problem lays in the fact that the sources are connected with different link speeds[11], and that the control notification does not differentiate between input links. If one had differentiated on input links the non-selective flow control would not cause harm to the others even in the presence of link speed mismatch.

S econd, *sharing of upstream resources* as illustrated by figure 4.15 may also lead to performance reduction because "when non-selective control is performed, the most congested path [...] dictates the performance of the others"[53, pg. 164].

[11] S2 and S3 share the 100Mbps, while S1 is connected directly to the rightmost switch by a 100Mbps link

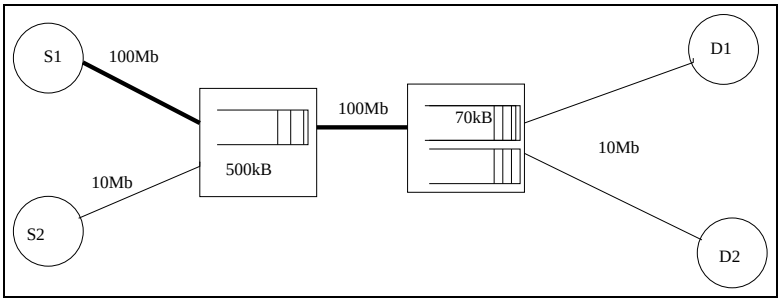

Figure 4.15: Noureddine Topology6

I t is suggested in [53] to make modifications to the IEEE802.3x standard to insert a Type length field of 4 bit for a CoS field and a Adr field, these four fields being inserted after the pause time in slots field in the present header. Moreover, it is remarked that "the implementation of switches which provide selective forwarding of frames based on destination MAC address [...] may be challenging"[53]. The buffer management is significantly more complex that FIFO.

4.7 Summary - congestion and flow control terms

B ecause of the global/local dimension of the taxonomy we claim that the flow control mechanisms discussed in this thesis for the link layer, fits well into the taxonomy. An important point regardless of whether our focus is flow control or congestion control, is that the control mechanisms are in one way or the other closely connected to buffer layout and -management. Understanding and handling the queues are crucial to understanding how the control mechanisms work.

Congestion is a state in which sharing of network resources, such as communication links, processing power and buffer capacity, fail because the total demand (load) exceeds capacity, resulting in performance degradation of the system.

Flow Control acts on a specific sender-receiver node pair and prevent the sender from feeding traffic faster than it can be handled by the receiver.

Congestion Control is the task of managing networks susceptible to congestion in order to preserve performance, and is a generalized form of flow control that is not limited to a specific sender-receiver node pair and might involve network-wide status in the decision making process.

Hop-by-hop flow control is a scheme exercised at each node along the path between two communicating entities.

Chapter 5

Protocol P

T he work on Protocol **P** was first introduced by *Mark Karol, S. Jamaloddin Golestani and David Lee* at ISCOM'99 [44], followed by a more generalized version including theoretical proof at the IEEE INFOCOM 2000 [42], and subsequently revised and printed in IEEE/ACM Transactions of Networking 2003 [43] titled *Prevention of deadlocks and livelocks in lossless backpressured packet networks*.

Protocol **P**

> prevents deadlocks and livelocks in backpressured networks without introducing any packet losses, without corrupting packet sequence, without relying on elaborate network-wide coordination requiring multihop control messages, and without requiring any change to packet headers. [43]

5.1 Motivation for a new protocol

I n the words of [43], here is the desired result of the proposed protocol:

> No packets will be dropped inside a packet network, even when congestion builds up, if congested nodes send backpressure feedback to neighboring nodes, informing them of unavailability of buffering capacity - stopping them from forwarding more packets until enough buffer becomes available.

T he authors point to Noureddine [52] for arguments in favor of a backpressure flow control mechanism. Arguments are made to favor link layer

flow control instead of letting TCP handle it (LAN context), but no simulation is done do measure how in fact the link layer affects upper layers

I n chapter 3 we pointed out that deadlock prevention strategies using some kind of distance information like hop counters in the packet headers are problematic because of non-compatibility with the IEEE802.3ae (z) standard. Protocol **P** aims to only use existing fields in this frame format.

5.2 Overview of the protocol

T he overall operation of protocol **P** can be described as a selective backpressure mechanism. Whenever the network, or a node, is in a non-congested state, all packets pending at a given node X are judged *eligible* for transmission. Eligibility is the qualification to be chosen, that is, making a packet a selectable candidate to the scheduling algorithm. As the condition at the downstream node of X worsens and congestion builds up, **P** gradually restricts the amount of eligible packets to avoid buffer overflow, by sending feedback to node X . Whenever congestion eases off, new feedback will relax the restrictions laid on eligibility. In other words, protocol **P** is about managing eligibility in terms of when, where and how.

5.2.1 Assumptions

I t is an underlying design requirement that no change to the standard Ethernet control frame header will be needed. Moreover, three assumptions on network behavior are made to highlight the properties of protocol **P** . First, we assume that the routing is static, see chapter 3. This implies that topology changes and varying traffic conditions in the network, that ordinary would trigger routing updates, are ignored. Later in section 5.6.3 we will show how the protocol can be modified to work with adaptive routing.

N ext we assume that destination-based packet forwarding is used. As we saw in section 3.1, the forwarding choice at a network node is based only on the destination address contained within the packet header. This leads to a network property where all traffic, both original and transit, having the same destination, will follow exactly the same path from a given node. Put in other words, data flows of a given source-destination pair always pass the same intermediate nodes, in the same order, and unless the scheduling algorithm at one of those nodes is ill-behaved (see below), the

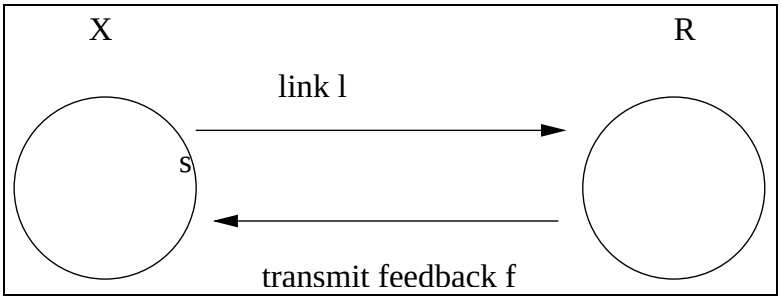

Figure 5.1: Protocol P: link from X to R

internal packet order of each data flow will be preserved. In section 5.6.3 we will explore how packet forwarding can be based on other information like ATM-VCI while still preserving the per flow packet sequence.

F inally, we assume all network links to have zero propagation delays. This is an important simplification when dealing with feedback based flow-control, because it implies that all control information is delivered instantly, and control action can take effect almost immediately eliminating the need for worst-case margins. Margins that will depend on factors like bandwidth, traffic patterns, network architecture and switch design. However, this assumption will be relaxed in section 5.6.2, and as we shall see both when discussing the implementation of **P** 6.4 and our simulation scenarios 6, link propagation delay is emulated.

5.2.2 Overview

We will now turn our attention to some building blocks and concepts for protocol **P** .

Links

L et us break down our packet network to the smallest unit still large enough to discuss all aspects of protocol **P** ; one-way communication links l connecting two adjacent nodes, see figure 5.1. In this model we name the sending side X_l and receiving side R_l respectively, and the reverse link l '. Note that for most switch architectures a port play both the role as input and output access point, and hence X_l and R_l for opposite directions of its link at the same time.

Scheduling algorithm S_l

A s we saw in section 1.2.4 the *scheduling algorithm* S_l can base its choice of the next eligible packet on a range of factors, including the arrival order of packets, and we will here adopt a *first in first out* (FIFO) basis for S_l in order to keep examples as pure as possible. When applying protocol **P** and its eligibility restrictions, this however imposes a semi-priority property to S_l on top of FIFO. The algorithm S_l is also assumed to be *well-behaved*, a necessary condition in order to avoid livelocks. We will return to eligibility issues in detail later.

Avoiding packet drops

I n chapter 3 we saw that dropping packets is a simple way to avoid deadlocks. Protocol **P** however was introduced as a backpressure congestion control mechanism that when applied per link leads to avoided packet drops while still avoiding deadlocks. Like other backpressure control mechanisms it sends a stop signal over the link l ′ to node X before buffer at R overflows. This mechanism is selective with regard to the destination MAC address. Because of assumption of zero propagation delay, the signal reaches X instantly.

Maximum number of hops D

T he maximum number of hops D that a packet has to traverse enroute, depends on the network topology and routing protocol applied. A minimal number of hops is for instance seen when shortest path routing is used, giving D equal the network diameter. Worst case this number will equal the number of nodes in the network: host 1 and host 2 connected to a serial line of N intermediate switches, gives a D value of N if 'hop' is interpreted as the number of switches traversed, D = N+1 if interpreted as the number of links crossed, or D = N+2 if all nodes involved are counted. In any legitimate network route, regardless of how 'hop' is defined, D represents an upper bound that has implications on logical buffer management and layout, which we will return to shortly.

Transmit Feedback

T ransmit Feedback f_l is sent as standard control messages in the reverse direction of the link, from R to X , restricting the set of eligible packets when congestion builds up. Values have to be chosen so that subsequent

packets can be stored at the receiving node, and are integers between 0 and D . The f_l parameter tends to be higher the more severe the congestion gets. Likewise, as congestion eases off f_l decreases resulting in gradually more eligible packets.

Packet Levels

F or each packet buffered at a given node a *packet level*λ_p :, i.e. an integer level varying between 0 and D is determined. This value is local to the nodes and is not transmitted, but λ^d can be predicted at the neighboring nodes based on the previous seen transmit feedback parameter. Including packet levels in headers would compromise the goal of not modifying the existing Ethernet header. Level assignment process will be described below.

I n all nodes a *level table* is kept for destination - λ^d pairs. Entries are only kept for current destinations of buffered packets, unlisted destinations have a λ^d of zero. Levels are stored per destination currently having packets in the buffer, and not pr packet.

Interaction of the elements of P

B efore delving deeper into the inner workings of the protocol, let us take a look at the superficial mechanisms of **P** . Monitoring buffer occupancies, **P** gradually restricts the amount of packets in a node that are selectable by S_l for transmission by sending transmit feedback f_l to the upstream node X . Given the last sent f_l , the packet p arriving at R is assigned a level f_l +1 to prevent deadlock, as the packet must have had a level of at least f_l when it was selected at X .

A ll packets destined for the same endpoint must have identical levels to prevent reordering of packets. Hence, if there are packets pending at R for the same destination as arriving packet p , these should all have their level reset if p is assigned a level higher than the previous. In this way there is only one valid level per destination d at a given time at a given node.

I n the following sections we present protocol **P** in detail. We have chosen to keep close to the IEEE/ACM paper ([43]) for the exactness of the basic requirements. These will however be elaborated and illustrated including material and issues discovered during the implementation phase.

5.3 Eligibility and level assignment

T he concept of eligibility and the way it is embedded in this protocol, can be viewed as one of the properties that differentiate **P** from the standard IEEE802.3x, and we will now have a formal look on this concept. As we shall see, the two parameters involved are transmit feedback parameter f_l and packet level λ^d , hence the level assignment process is also covered in this section.

5.3.1 The Transmit Eligibility Rule

A s we have seen, both transmit feedback f_l and packet level λ_p are integers between 0 and D , inclusive. Eligibility of packets waiting for transmission over a link l at node X_l having received transmit feedback f_l is determined by applying the following rule.

> *Transmit Eligibility Rule:* A packet p waiting at node X_l is *eligible* to be picked up by the scheduler of link l for transmission over l , if its current level λ_p satisfies
>
> $$\lambda_p \geq f_l \tag{5.1}$$
>
> where f_l is the most recent transmit feedback received by X_l from the receiving node R_l .

H ence, a packet remains eligible as long as its current level is equal to, or higher than, the most recent transmit feedback received. In the case of $f_l = 0$, it follows from the rule that any packet is eligible. The higher the f_l parameter, the more restrictive **P** behaves.

A t some point when the network/node is experiencing congestion, it might be that no packet is eligible at all in X_l . In this scenario, the congested node/interface/queue at R_l is shielded from arriving traffic while packets are allowed to proceed toward their destinations, freeing up buffer resources until once again X_l is given a feedback f_l resulting in eligible packets.

D ue to the assumption of zero propagation delay, we have instant delivery of transmit feedback. In real network scenarios this is an utopia, and in section 5.6.2 modifications to protocol **P** relaxing this assumption are introduced.

5.3.2 The Level Assignment Rules

I t has been showed how packet level λ_p affects the outcome of eligibility evaluation, and below we describe the determination process and rules for level assignment.

> Let packet p arrive to node R_l over link l and assume that f_l is the most recent transmit feedback sent to X_l . It follows that the level of p prior to transmission from the previous node (X_l) must have been f_l or larger. To guarantee freedom of deadlocks in the network, it suffices to assign level 1 + f_l to packet p [43, pg. 925].

T his simple scheme has the advantage of being easy to comprehend and straightforward to implement. However, because feedback parameter f_l can and will vary over time in response of traffic pattern and load, we face the danger of assigning different levels to packets belonging to a single session. If this happens, those packets might get misordered when forwarded onto the downstream neighbor. Remember that scheduling is FIFO based, but that eligibility evaluation lies on top of FIFO principles so that the first packet from the head *that satisfies the eligibility criteria* is selected for transmission. Hence, packet p 1 of a session might be skipped and packet p 2 selected if they have non-equal levels λ_p .

B ecause of the assumption of destination-based packet forwarding (see section 5.2.1), and the implications that has on the path taken by packets belonging to a specific session, measures have been taken in the design of protocol **P** to avoid the above mentioned misordering:

> at each node and at each point of time, all buffered packets that have a common destination should have the same level so that all will be eligible/ineligible at the same time, and therefore selected for transmission in the correct order.[43, pg. 926]

T o accomplish this *re-leveling*, protocol **P** monitors the packet level λ_p for all packets having an identical destination address at a given node, and for all those packets lift their levels to the highest λ_p among them. A potential advantage following directly from re-leveling is that those packets are more likely to evaluate as eligible for transmission.

T he above described design principle for **P** in fact simplifies housekeeping involved in level management. Instead of storing a level λ_p per packet,

a level λ^d can be kept per destination in a list referred to as the *level table*. To determine the level of a packet p at a given time, the protocol has to check the level table and apply the following rules.

Level Assignment Rules:

1. At each node, initially set $\lambda^d = 0$ for all destinations d .
2. When a packet p with destination d arrives from another network node over some link l , the level associated with d is updated as
$$\lambda^d \longleftarrow max(\lambda^d, 1 + f_l) \tag{5.2}$$
where f_l is the value of the most recent transmit feedback sent over the reverse link l '.
3. When a packet p with destination d enters the network at node n (over some network access link) the destination level λ^d does not change.

W hen a **P** enhanced node is brought online, the level table begins as an empty list. Any destination d not found in the table, is assigned $\lambda^d = 0$ as default. Remember that all packets with a zero λ_p are eligible. The protocol works by putting restrictions on eligibility, and hence it makes sense only to keep records for those packets/destinations currently under some form of control action. When the last of the packets belonging to a session, or more precise belonging to a destination, as sessions toward a common destination d are treated identical by **P** , has left the node, the entry d - λ^d can be removed from the table in order to keep table sizes small. Another way to view this removal, is to reset λ^d to 0 for all d not matching any buffered packet at a node.

G iven an arbitrary packet p destined for destination d at an arbitrary node n at time t, let $j_n^d(t)$ denote the associated level. As seen above, a destination d not listed in the level table leads to $j_n^d(t) = 0$. Moreover, $j_n^d(t)$ was necessary updated or set to its current value in response of a previously arriving packet p_1 (for d), which was transmitted in response of a transmit feedback f_l 1 less than $j_n^d(t)$ sent from the current node. Hence, the λ^d must have been $j_n^d(t) - 1$ at some point of time at the upstream node. This argument can be applied on each step of the reverse path[1], for a total of h steps indicating that the source of packet p has been reached. Since h must be less or equal to parameter D , and λ^d at the network enter point is 0, the expression $j_n^d(t) \leq h \leq D$ holds, and will be true for all p 's , n 's and d 's.

[1]The upstream subset of the path taken by packets belonging to the session including packet p

Protocol designers' observations regarding the rules

T he following section presents a list of observations as reported by [43] regarding the Level Assignment Rules. The implications of these issues will be summarized thereafter.

- If all packets encountered by node n and destined for destination d enter the network at n , then λ^d is always equal to zero (at n since it is never subjected to the update in 5.2. [...]
- When a packet arrives at node n from another network node, it will be assigned a level of at least 1, since the level associated with its destination will undergo the update in 5.2.
- Updating according to the above rules will never result in a level larger than D . [...] by the time the level associated with a packet reaches D , it must have reached its destination [...]

T raffic entering the network at node n have $\lambda^d = 0$ unless other traffic for d passes through that node. Lowest level for packets being forwarded is 1, and always no greater than D .

5.4 Switch Model and Buffer Management

5.4.1 Switch Model

P rotocol **P** was designed as a technique to be used with any switch configurations, and this leads us to a general switch model used to illustrate the properties of **P** (see figure 5.2). It will later be shown that this model has some implications for implementation in our simulation environment, as well as pointed to that this model is not all but unproblematic.

Virtual input-output queue $Q_{i,j}$ associated with input-output pairs (i, j)

Virtual receiving queue for each incoming network link, used for monitoring buffer occupancies and choosing the next transmit feedback f_l based on the current situation

Virtual sending queue for each outgoing network link, used by the scheduling algorithm S_l selecting the next eligible packet for transmission

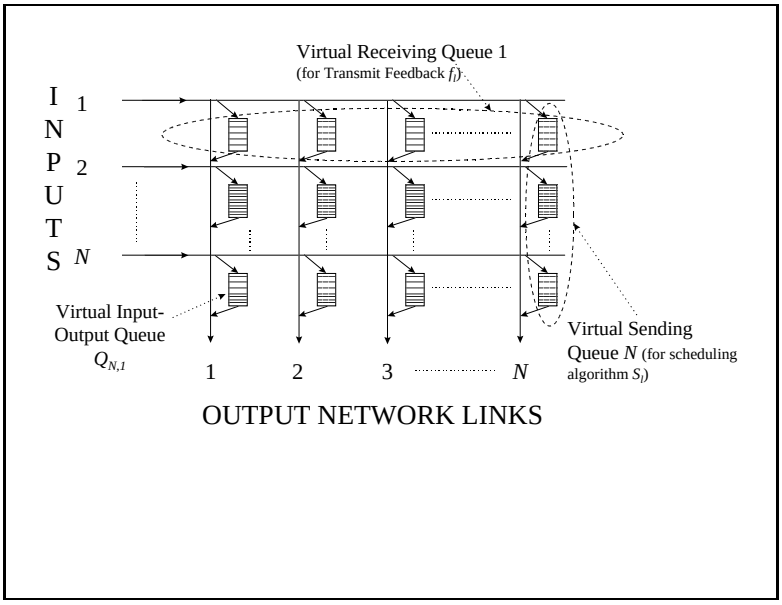

Figure 5.2: Protocol P: generic switch model

A packet p belongs to both the latter queues at the same time in this model. In fact, the layout of figure 5.2 can be viewed as a plain two dimensional matrix where $Q_{i,j}$ is represented by a single square, and the receiving and sending queues are represented by a single row and single column respectively.

N ote that this switch model only includes interfaces for inter-switch communication in the network. *Drop-links* (access-links) connecting a switch to endnodes (data sources and/or destinations) are not shown. Furthermore, protocol **P** does not specify anything regarding these drop-links, not even on admission policies, flow control or buffer requirements.

P hysical (hardware) implementation of this switch model is by [43] claimed to be possible "in many ways" [43, pg. 927]:

> in a completely-shared-memory switch , all input-output , receiving, and sending queues are maintained in lists as packets arrive on various incoming links and depart on various outgoing links. In an input-buffered (output-buffered) switch, however , the receiving (sending) queue could be a physical buffer and the other queues would still be virtual entities that are maintained

W e will return to this claim in section 6.4 when discussing how protocol **P** is implemented and fitted into our simulation environment. Until then, we will keep to this general switch model, and ignoring the issue of drop-links. Moreover, the fact that some switch configurations can have variable size queues (space allocated to those queues) is ignored for the clearness of the presentation. Meanwhile attention is turned toward layout and management of (virtual) buffers.

5.4.2 Buffer Layout

B uilding on the knowledge of protocol **P** that we have established so far, it is now to fill in the last pieces with rules for buffer management and transmit feedback. Let us return to the scenario of figure 5.1 with a link l connecting a sending node X_l and a receiving node R_l , and let size of the receiving queue at R_l be denoted by b^l . More, let γ_{max} be the value of the networks MTU. This parameter will be used as a *chunk unit* for the buffer layout, as it is of little use to have buffer space not capable of holding an entire packet.

M anaging the receiving queue and, in relation to that, set the f_l parameter accordingly will be our main focus for the rest of this section. For the receiving queue initialize a threshold B_i guarding the upper limit of buffer space available to packets of levels *leq i*. The f_l sent by R_l to X_l is the lowest level of packets X_l can transmit and R_l receive without violating the B_i thresholds.

T he buffer space b^l is divided as shown in figure 5.3 into D parts b_i, $i = 1, 2, \ldots, D$ where

$$b_i \geq \gamma_{max} \tag{5.3}$$

and

$$b^l = \sum_{j=1}^{D} b_j \geq D \times \gamma_{max} \tag{5.4}$$

> [...] We refer to b_i as the *buffer budget* of level i and require that a packet of level i be accepted into the buffer only if there is enough budget available for it at levels i below. Let n_i, $i = 1, 2, \ldots, D$, denote the combined size of packets of level i that are stored in the receiving queue of link l . The above requirement may be stated as [...]

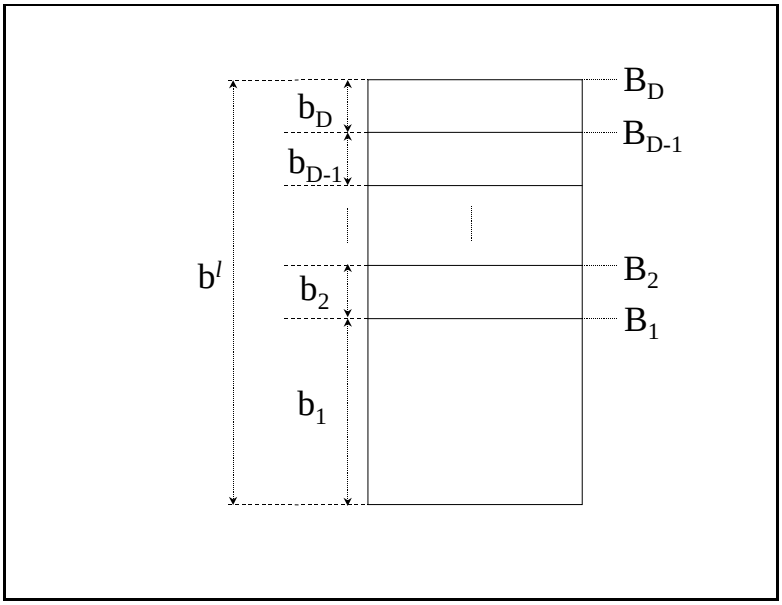

Figure 5.3: Budget allocations of link l 's receiving queue

$$\sum_{j=1}^{i} n_j \leq \sum_{j=1}^{i} b_j, i = 1, 2, \ldots, D. \tag{5.5}$$

In other words, the combined packet sizes n_j don't exceed the combined buffer sizes.

T he buffer budget of level i is hence established as b_i. Moreover, we need an upper threshold on buffer used by packets of level $\leq i$:

$$B_i = \sum_{j=1}^{j=i} b_j \tag{5.6}$$

It is crucial that this constraint is satisfied for all i at all times. Following from equation 5.6 and figure 5.3, we have that $B_D = b^l$.

V irtually, these requirements can be implemented as seen in figure 5.4 using a set of buffer management parameters m_i, where m_i refers to the part of combined buffer budget of levels $j \leq i$ not yet allocated to packets of these levels respectively:

$$m_i \triangleq \sum_{j=1}^{j=i} (b_j - n_j), i = 1, 2, \ldots, D. \tag{5.7}$$

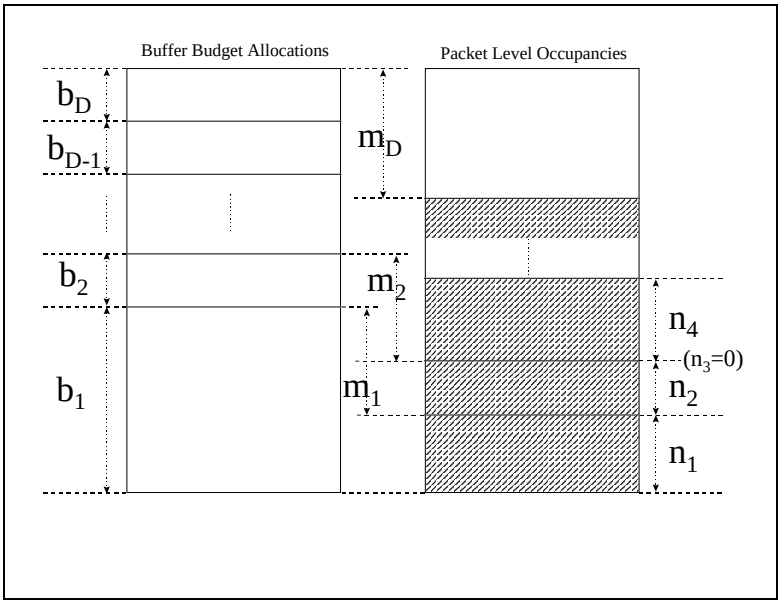

Figure 5.4: Buffer management parameters - *l* 's receiving queue

F rom these definitions, [43] point to some relations between buffer management parameters and packet level occupancies. First, "out of the combined buffer budget of levels $j \leq i$, a budget m_i is either allocated to packets of levels $j \geq i$ or not allocated to any packets at all". Second, m_D denotes the sum of non-allocated buffer space in the receiving queue. Third, "since packets of level j can use the buffer budget of any level $k \leq j$, the term $b_j - n_j$ [... in equation 5.7] can be negative, for some j. However, m_i cannot be negative for any i since packets of levels $j \leq i$ cannot use the buffer budget of a level higher than i"[43, pg. 928].

Buffer Management Rules:

1. When the receiving queue of link l is empty, initiate

$$m_i = \sum_{j=1}^{j=i} b_j, i = 1, 2, \ldots, D. \tag{5.8}$$

2. If a packet of length γ arrives and is buffered at level j, decrease m_i by γ, for $i \geq j$.
3. If a packet of length γ and level j leaves the buffer, increase m_i by γ, for $i \geq j$.
4. If a packet of length γ is lifted from level j to level $k \geq j$, increase m_i by *gamma* for all i such that $k \geq i \geq j$.

N ow that we have a set of buffer management parameters, we also have the means of choosing the transmit feedback parameter f_l to enforce the above declared rules. m_i should never be negative, as that would indicate a serious error where more budget than available has been allocated. Hence, as long as $m_i \geq \gamma_{max}$ for $i \geq j$, an arriving packet designated level j will be allowed into the queue. [43] conclude that "f_l should be set to a level j such that $m_i \geq \gamma_{max}$, for all $i \geq j+1$, and $m_j \leq \gamma max$ " . If there is at least γmax space available at all levels, there is no need for control action as all packets will be eligible, and hence $f_l = 0$. In the opposite case where $m_D \leq \gamma max$, f_l must be set to D , resulting in zero eligible packets at X_l . Remember that in the case a node n encounters a packet of level D , that packet is destined for n (or an endnode connected to n), and consequently that packet will under no circumstances be forwarded on to the link of an downstream switch.

Transmit Feedback Rules:

1. At the receiving end of R_l of each link l , set the corresponding transmit feedback $f_l = j$, where j is the *largest* level for which $m_i \leq \gamma_{max}$. If no such level exists, set $f_l = 0$.
2. Whenever f_l changes, send an immediate feedback with the new value of f_l to the transmitting node X_l .

F inally, a comment on the scheduling algorithm S_l regarding eligibility. Contrary to plain backpressure mechanisms, like IEEE802.3x described in chapter 4, which either mark all or none packets eligible at a given time at a given node, **P** might, and is likely to, mark a subset of the packets pending at n eligible whilst the rest is in-eligible. In this way protocol **P** interferes with the normal operation of S_l , and may change the order of packet transmissions. However, because of the common level table in the node, the order is not changed for each source-destination pair, which internally are treated in FIFO manner. As promised, we will return to implementation issues on buffer layout for protocol **P** in section 6.4, and see how D affect b^l .

5.5 Theoretical Proof

P rotocol **P** was introduced as a *proposed technique*, a technique whose properties are formally stated in the form of a theorem that is proved by reasoning and induction. Up to today, to the knowledge of the designers of **P** and the author of this thesis, there has been no implementation of the

protocol, neither software simulation nor in hardware. As described in the problem domain of this thesis 1.3, we here represent a software based simulation focusing on the performance measurements. Hence, the detailed mathematical reasoning of [43] will only be summarized in this section, and we refer to the paper for a complete description. The definitions, theorems and lemmas below in this section are all cited directly from [43, pg. 929-931].

R ecall the definitions of a deadlock and livelock free network (3.23.3) along with the criteria for a deadlock or livelock (1). It has been demonstrated in the above sections that protocol **P** is sufficient to avoid deadlocks based on the level assignment rules and eligibility rule. To eliminate the possibility of livelock in a network, the following conditions must hold for each link in the network. [2].

Definition 5.1 *The* eligibility age *of a packet waiting for transmission over a link* l *is the combined duration of all periods of time during which the packet has been waiting and has been eligible for transmission over* l *.*

Definition 5.2 *The scheduling algorithm of a link* l *is defined to be* livelock-free *if the eligibility age of no packet waiting for transmission over* l *can grow indefinitely.*

T he properties of protocol **P** are formally stated below

Theorem 5.1 *Consider a packet network using the selective backpressure protocol* ***P*** *. Assume that no packet in the network travels more than* D *hops. Furthermore, assume that the propagation delays of all links are zero, the network routing is static and packet forwarding in the network is destination-based. In the absence of transmission or processing errors, the following properties hold.*

1. *Packet transmission in the network is loss free.*
2. *The order of packets belonging to the same session is maintained as they pass through the network provided that their order would be maintained by the scheduling algorithm S_l of each traversed link* l *when operating in the absence of protocol* ***P*** *.*
3. *The network is free of deadlock.*
4. *The network is free of livelock provided that the scheduling algorithm S_l of each network link is livelock-free*

[2]The definition 5.2 below is not satisfied by all scheduling algorithms, e.g. strict priority

To support this theorem, three lemmas are needed

Lemma 5.1 *The level λ_p of a packet* p *buffered at a given node* n *always satisfies*

$$\lambda_p \leq D \tag{5.9}$$

with equality only if n *is the destination node for* p .

Lemma 5.2 *At the receiving queue associated with any link* l , *parameters m_i always satisfy*

$$m_i = m_{i-1} + b_i - n_i, i = 2, 3, \ldots, D \tag{5.10}$$

and

$$m_i \geq 0, i = 1, 2, \ldots, D \tag{5.11}$$

Lemma 5.3 *Consider the receiving queue associated with a network link* l *and an arbitrary level k, $1 \leq k \leq D$. Assume that each packet in the buffer has a level k or higher, will leave the buffer within some finite time. It follows that for any arbitrary time t_0, there is a finite time $t_1 \geq t_0$ at which $f_l \leq k$.*

The proof of theorem 5.1 can be summarized as follows. At a node n (R_l) there will always be available space to buffer packet p because the upstream node X_l would not send p unless a feedback $f_l \leq D-1$ has been sent it from R_l indicating that $m_D \geq \gamma max$ (at least 1 MTU worth of free buffer total). Second, since all packets destined for d stored at n at a given time t are assigned the same level, they will either all be eligible or all in-eligible with respect to transmission over l . Consequently, since S_l is assumed to preserve the relative order of packets, the same would hold in the presence of P. Third, it can be shown by induction (applying lemma 5.1 and 5.3) that a packet of arbitrary level $1 - D$ enqueued at node n will leave n in finite time[3]. Finally, "since each packet may travel a bounded number of hops in the network, and since the waiting time of each packet at each node is bounded, each packet will leave the network in finite time." [43, pg. 31].

5.6 Extensions and variations to the protocol

In this section we will present alternatives that has been suggested on protocol **P** in [42] and [43]. Because of the focus of this thesis, emphasis will be put on demonstrating how protocol **P** can coexist in a network with IEEE802.3x capable nodes, and in addition the issue of non-zero propagation delays. The other assumptions made in section 5.2.1 are covered more lightly along with some variations of minor importance for our study.

[3]It is assumed that after some point of time, there are no new packet arrivals to the network

5.6.1 Protocol P coexisting IEEE802.3x

H ow protocol **P** transmits its feedback parameter f_l is a bit glossed over in the 2003-paper ([43]), where "PAUSE signals of gigabit Ethernet (IEEE802.3z) technology" (pg. 923) is used as an example of backpressure based networking, and it is pointed out that the proposed protocol should be compatible with this frame format. It is however not required that the underlying technology is gigabit Ethernet, nor that the f_l is exchanged between nodes using PAUSE frames. But, for the rest of this thesis we will assume these requirements fulfilled unless otherwise noted.

W e have seen in chapter 4 how the PAUSE signals of gigabit Ethernet works, but let us brief review the important traits; Congestion is handled with a stop-start mechanism using XOFF and XON units transmitted form the node experiencing congestion to its troublesome upstream neighbor. These units are carried inside the standard control frame as an integer in the headers pause_value field, and the network interface receiving this unit as a number of 512 bit time slots that it is required to refrain from transmission of any new data frames. It varies between vendors whether this activates a deactivation timer, or an explicit XON has to be received before normal operation can be restored.

L et us refer to nodes implementing XOFF PAUSE signals and transmit feedback parameter as *regular* and *enhanced* nodes respectively. Upon receiving a control frame, a regular node will interpret the pause_field in accordance with PAUSE rules and stop(start) the interface, whereas enhanced nodes will interpret it as a f_l value adjusting the basis for eligibility criteria.

I ntroducing networks with a mix of regular and enhanced nodes, a potential source of misinterpreting arises. The regular nodes have no means to know anything about protocol **P** nor how it reinterprets the pause_field in the control frame header. Hence, regular nodes will always operate in compliance of the standard protocol. Enhanced nodes on the other side have the potential advantage of knowing that there might be regular nodes somewhere in the network, but without adding a signaling sub-protocol to probe the network[4] they can only infer from the response seen on transmit feedback sent, the nature of their adjacent link partners. However, not

[4] or a network administrator explicit setting this parameter in the switches; an intervention that will prohibit (proper) reaction to dynamic topology changes during system operations

knowing for sure if R_l is enhanced or not, X_l has to react to a control frame as if R_l were a regular node not to violate the IEEE standard.

T herefore,to reduce the danger of misunderstanding,

> When a PAUSE frame is sent form an "enhanced" node to signify a Transmit Feedback value, then the PAUSE frame should be followed immediately by a second PAUSE frame with its parameter set equal to zero (to continue the immediate transmission of data frames).

We will refer to the second PAUSE frame containing XON for protocol **P** as *confirm frame*. The list below represents the four different scenarios that a mix of regular and enhanced nodes can create.

X_l **regular** R_l **regular :** Link partners in this scenario both understand, use and comply to standard PAUSE signaling, and the resulting performance has been studied in detail in papers like [52], [81] etc. Both nodes interpret the pause_field as bit times, and no extra XON is ever generated.

X_l **regular** R_l **enhanced :** In this case, R_l sends a first control frame containing the f_l parameter, followed by a confirm frame. Upon reception, X_l will assume that the first control frame with a positive non-zero value was sent it to halt transmission for the given time period. The value 0 in the subsequent frame will cancel the previous seen stop signal. Consequently, X_l will continue sending at full rate in spite of R_l 's attempts to throttle it, and likely cause packet drops at R_l as buffers overflow. This is clearly the most dangerous combination of the two protocols in terms of potential unwanted packet loss.

X_l **enhanced** R_l **regular :** In the third scenario the enhanced node will have to assume that the first control frame from R_l tells it to pause for the specified time period. In the absence of the expected control frame, this case is identical to the both X_l and R_l regular combination in that a pause in fact occurs. But, when R_l sends the XON message, X_l will interpret it as confirmation and resume sending with a eligibility criteria dictated by the previous seen XOFF value (which most likely will be an integer way out of the range of normal f_l and even exceeding D . Despite [42] does not mention it explicit, we find that this scenario in particular has the potential of indefinitely halting a network interface creating a serious deadlock. Whether this is worse than the previous scenario leading to packet loss, depends on the higher-level protocols and applications used. In case of zero-tolerance

on packet loss, a deadlock might be better, but unless the network has some form of fault-tolerance and capability to adapt to topology changes in response of a link failure, we are facing a growing network-wide breakdown.

X_l **enhanced** R_l **enhanced :** Last we have the case of a pure enhanced network, or at least link partner pair, and this is the only scenario where guarantees against deadlock, livelock and packet loss are preserved by protocol **P** . All nodes interpret the pause time as f_l , and adjust the amount of packets being eligible at X_l while continuing transmissions upon receiving the confirm frame. The latter frame can be removed to increase performance if the two nodes learn that they both are enhanced.

Conclusions on inter-working of regular and enhanced nodes Protocol P needs to use confirm frames for each control frame with transmit feedback f_l sent, to discriminate operation from standard gigabit Ethernet flow control. We have pointed to different harmful effect that some of the combinations of these two mechanisms can lead to. The authors of [42] conclude with that

> when a network is built with a mixture of both "enhanced" and "regular" IEEE802.3 equipment, there are no guarantees against the possibility of network deadlocks and livelocks, and packet loss.

5.6.2 Non-zero propagation delays

A non-existing propagation delay on inter-switch links is a networking utopia, but despite the hard facts of reality, many network administrators have probably wished for just that on some occasion. It may also be convenient to ignore this parameter when designing, proving and testing new protocols. However, it is now time to relax this assumption and look at how the effects of non-zero propagation delays can be incorporated into protocol **P** .

L et T denote the round-trip-time (RTT) of the link l in figure 5.1. According to [43], "we simply need to redefine (re-interpret) the n_i occupancy parameters to include (worst-case) potential packets of level i that *might be* received at R_l during the next T time units" [43, pg. 932]. In other words, we have to expand n_i to include a margin of x packets ($x * \gamma max$), where x depends on the underlying technology and RTT.

T he idea is to make sure that the f_l signals are transmitted early enough to reach the upstream node X_l and take effect in time to alter the eligibility criteria before the buffers at R_l overflow. For action wanted at time t_0 the control frame has to be sent at time $t_0 - T$. The two most recently published revisions of protocol **P** ([42] and [43]) have a slightly different view on how and and which rules to modify. The latter paper seems to have the theoretical most sound suggestion. However, despite that it only requires the above cited re-interpretation of n_i, along with altering the level assignment rule 5.2, it requires R_l to keep a history record of all transmit feedbacks f_l sent during the previous T period. This will require additional book-keeping to a protocol already burdened with extensive virtual buffer management.

O n the other hand, the former paper mentioned above requires changes both to the transmit feedback rule and the level assignment rule. But, instead of emulating filling and shrinking of queues (buffer occupancies) based on the current f_l , this paper suggests to calculate the worst-case amount of packets that might arrive from X_l to R_l during time T. The value rT arrived at will be a function of the datarate r and the RTT (T) of link l , and indicate how much the 'water' on the receiving queue bar will rise. In other words, rT is the *margin* reffered to a few paragraphs above. Consequently, the described strategy leads to the following modified rules[5]:

Transmit Feedback Rule: This rule replaces step 1 of the old transmit feedback rule on page 104, preserving step 2 as it is.

> Set link l 's Transmit Feedback $f_l = j$, where j is the *largest* Level for which $m_j - rT \leq \gamma max$. If no such Level exists, set $f_l = 0$.

Level Assignment Rule: This rule replaces step 2 of the old level assignment rules on page 98 (i.e. equation 5.2), preserving steps 1 and 3.

> When a packet p with destination d arrives from another network node over somelikn l , the Level associated with d is updated as
>
> $$\lambda^d \longleftarrow max(\lambda^d, 1 + j) \tag{5.12}$$
>
> where j is *largest* Level for which currently $m_j \leq \gamma max$ (if no such Level exists, set $j = 0$).

[5] note that both transmit eligibility rule and the buffer management rules are unchanged

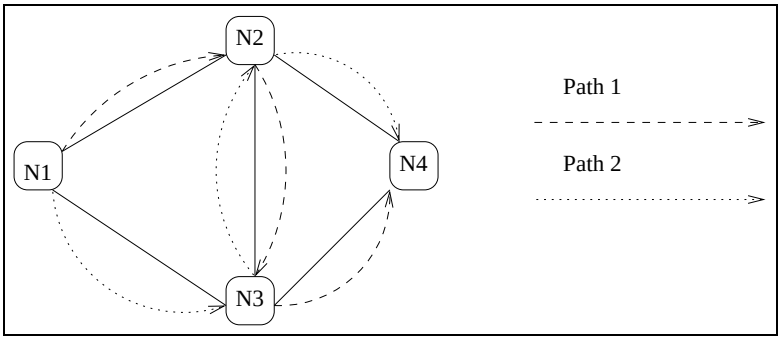

Figure 5.5: Topology and alternative paths N1-N4 using adaptive routing

I t is noted in [42] that R_l should have correct estimate of the RTT T. If this is not the case and the RTT is greater than the estimate, buffer overflow and packet loss might occur. On the other hand, a too generous estimate will waste valuable buffer resources. Because of this, an *upper bound* for the T is suggested applied to all nodes, in the same way as the parameter D . Moreover, to ensure that packet burst during a non-congested time period do not lead to buffer overflow at R_l , the following expression should hold:

$$b_1 \geq \gamma max + rT \tag{5.13}$$

W e have now seen how non-zero propagation delays and their effect can be incorporated onto protocol **P** . Next, we present some other variations that are suggested in the papers describing **P** , but of lesser importance for this thesis.

5.6.3 Variations on forwarding and routing

Compatibility with adaptive routing

R elaxing the assumption of static routing made in 5.2.1, there is a potential danger of entering an endless loop of level table updates that would lead to violation of lemma 5.1. Let us illustrate this scenario with an example using the topology and paths shown in figure 5.5. Assume that all traffic originates at node N1 and that all packets p_n have d = N4 and follow Path 1 (N1-N2-N3-N4). Assume further that D is set to 3, as no route in this four-node topology can span more than 3 hops. While packets for d still are routed along Path 1, the entry for d in level table at N3 reaches its maximum value 2 [6].

[6]Refer to section 5.3 for review of level assignment rules and explanation of why λ^d caps at D -1 for a packet waiting to be forwarded ont the next link

A t this point of time a routing update occurs in the network, where the internal order of the two intermediate nodes change so that packets follow Path2. Packet p arriving at N3 will according to the rules be assigned λ_p = 2 despite that it have only traversed one hop. If the most recent feedback level f_l sent N3 form N2 is 2, then p will be assigned level 3 (i.e. $\lambda_d = D$) upon arriving node N2, despite that one link remains to be traversed before p reaches its destination. This behavior will repeat if the routing continues to alter between Path 1 and Path 2, and as a result λ^d may grow indefinitely.

T he source of this problem is the preserving nature of the level table combined with the principle of assigning all packets with a common destination identical levels, as described in section 5.3.2 (equation 5.2), in order to not corrupt the packet sequence. [43] therefor suggest a new set of level assignment rules that will permit packets with a common destination d at node n to have dissimilar packet levels, provided that for two arbitrary packets p_1 and p_2 with destination d , the oldest packet should never have λ_p lower than the newer one. Note that this alternative involves storing levels per packet in stead of per destination.

Level Assignment Rules -revised:

At each node n , when a packet p with destination d arrives from another network node over some link l :

1. assign p with the level

$$\lambda_p = 1 + f_l \tag{5.14}$$

   where f_l is the value of the most recent transmit feedback sent over the reverse link l ';

2. lift the level of all packets p ' that have the same destination d and which are currently buffered at n as

$$\lambda_{p'} \longleftarrow max(\lambda_p, \lambda_{p'}). \tag{5.15}$$

Packet forwarding considerations

I n most network scenarions it is desirable to preserv the relative ordering of packets as they are being forwarded. Different technologies use different header fields to identify the subset of packets that belong to a session and hence should remain in order. So far we have assumed destination-based forwarding where destination address is mapped to packet level λ^d . However, the underlying principle can easily be fitted into other technologies

by redefining the mapping criteria. In practice, this would mean mapping the virtual circuit identifier (VCI) in ATM networks, or the label in MPLS networks, to vc-levels or label-levels respectively.

I f maintaining the packet sequence is irrelevant, the level assignment can be done individually for each packet, with $\lambda_p = 0$ for packets arriving the network and $\lambda_p = f_l + 1$ for the rest.

5.6.4 Other variations

W e have now seen how protocol can be modified to coexist with standard IEEE802.3 equipment in a network. Further, the assumptions on zero-propagation delays, static routing and destination-based packet forwarding have been relaxed to make protocol **P** more fit for real networking scenarios. Before we turn to how **P** was implemented in our simulation environment, a few more alternatives have been addressed in [42] and [43], and are presented below. In short, these alternatives regard fairness issues and situations in which **P** relates to other network properties (beside deadlock, livelock and lossless transmission) and higher-level protocols .

N etworking issues other than the ones addressed by protocol **P** might rely on packet dropping for proper operation. This is the case when a packet gets corrupted, or its lifetime expires, and hence as the packet is no longer deliverable it should be allowed removed (i.e. dropped).

M oreover, during congestion some input-output port pair in a given node can become a bottleneck with potential unfair memory-sharing in the virtual receiving queue of the input port in question. Recall the layout of these virtual queues, see figure 5.3. Assume that most of the packets residing in queue Q_0, as well as the majority of packets arriving on that interface (Ni_0), are destined for output link l_2 and that Ni_2 is not capable of transmitting packets onto the link at the speed packets for that destination port arrives the switch.

I f all ports of a switch are link speed capable, and traffic arriving at port A (at linkspeed) exclusively is forwarded onto port B with no other traffic crossing the switch toward port B, then this traffic will not cause problems. However, the excess traffic arriving from other input ports will contribute to a fan-in effect toward port B, and hence these packets will gradually fill

up the virtual input queues of the switch. This might cause protocol **P** control action sending transmit feedback to the upstream node. For the packets entering at port A but exiting on a different output port being free (underutilization), this behavior implies that some packets will be unnessesary delayed. To resolve this issue, [43] suggest to drop some of the offending packets, and notes that:

> performance of the protocol **P** (which reduses end-to-end delay when network resources are used in a "fair" fashin) coubled with selective dropping of packets that take "too much" of a node's memory is a topic for future research [43, pg. 933].

A s established in chapter 1 flow control at the link layer mainly addresses short-term congestion, whereas long-term (sustained) congestion more often is handled at higher levels. Protocol **P** does not address fairness in providing service to different users, as it only inspects MAC destination addresses and has no concept of users[7]. Therefore, it might be relevant "to couple this technique with end-to-end congestion control schemes that handle congestion problems on a quasi-static basis while providing the desired fairness and/oir priorities in the amount of services given to different users in the long run" [43, pg. 933].

T he Transmission Control Protocl (TCP) relies on packet dropps to rate control its sources for end-to-end congestion management. To internetwork a lossless network applying protocol **P** with TCP, gateways at the network edges might be inserted to handle losses and signal TCP while preserving a lossless nature inside the **P** enhanced network.

F inally, the question of Quality of Service (QoS) in networks utilizing the proposed protocol need attention. The current buffer layout and management does not allow priorities nor differentiated service at a node. However, some modifications have been suggested to accomodate thisl. If dedicated buffers are avaialble to prioritized traffic, this subset of packets might be permitted to ignore the transmit feedback parameter f_l and be evaluated as eligible at all times. Alternatively, a minimum threshold can be set on the packet level λ_p assignable to this traffic.

A ll the above alternatives require additional book-keeping, processing, buffers and/or devices to funcion. The total cost of these factors must be

[7] However, like described in 5.6.3 above, replacing the MAC address with VCI in ATM networks might give a stronger bond to users

taken into account and weighed agains the resulting performance effects in order to answer whether the modifications is worth to pursue.

Chapter 6

Implementation and Simulation Scenarios

This chapter explains how we have implemented and carried out simulations of the flow control schemes. We start by showing the overall network interface operation in terms of how the host and switch nodes are composed, and the initial processing of arriving frames, followed by making our contributions to the J-Sim architecture explicit. Subsequently we move on to specific implementation issues for IEEE 802.3x and protocol **P** respectively. Sections 6.5 and 6.6 state and explain topologies, routing and workload parameters used. The chapter closes with a section on how we have set up the simulations using J-Sim and Tcl.

6.1 Network interface operation

In the following we will make extensive usage of flowcharts to explain behavior of network interface operation. Figure 6.1 shows the most commonly used symbols, arrows excluded. When simply showing the relationship between elements, for example the main components of a node, we resort to a simpler scheme using rectangles for network interface and queue elements, and circles for the remaining.

Further, we use a shorthand notation and also abbreviations related to the specific protocol implementation. Abbreviations are introduced in the context they appear, whereas the shorthand notation is listed below.

- *Q*, queue
- *NI*, network interface

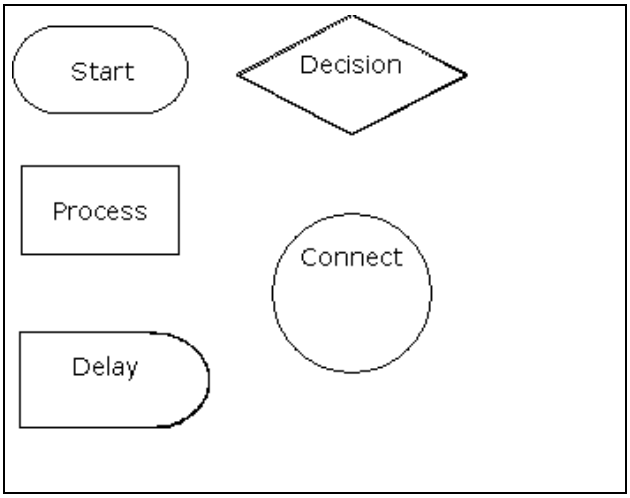

Figure 6.1: Description of flow chart elements

- *RT*, routing/ routing table
- *BB*, buffer budget (BBQ = bb queue, BBC = bb counter)
- *FC*, flow control
- *N/Y*, no/yes (for choices)
- *P*, Protocol **P**
- *X*, IEEE 802.3x
- two letter combinations ending with *F*, various timers named by the fork process it is associated with
- *flags*, are used within, and in relation to, the flowcharts. Names are based on what each flag is used for, and not directly related to a specific protocol or standard.

W e define integer constants for flow control types as `FC_TYPE_NONE` = 0, `FC_TYPE_IEEE` = 1 and `FC_TYPE_P` = 2.

T o relate the following models of network nodes to the J-Sim environment, recall the illustrations of the Core Service Layer (CSL) from figures 2.6 and 2.7.

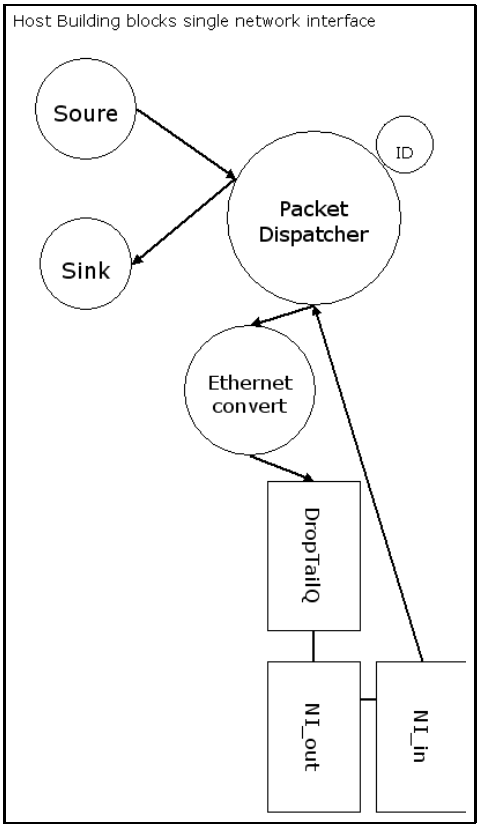

Figure 6.2: Host node with single network interface, droptail queue, packet dispatcher with identification service, and components related to workload and framing.

6.1.1 Host node

F igure 6.2 shows a host node (end station) with a single network interface. Host nodes act in our study only as sources/sinks. Traffic is injected by using the built-in traffic generation tool of J-Sim, then converting these packets to well-formed Ethernet frames, inserting them to a droptail queue, and within turn handing them off to the the link scheduler. This queue ensures that a host node will always have data to inject to the network if it is permitted to send. The droptail property ensures that in the absence of an upper layer protocol, outgoing frames do not pile up and grow out of age. This behavior is in line with [17] and our remarks in section 2.2.5 earlier. Incoming frames destined for a host node are delivered directly to the dispatcher and drained.

6.1.2 Switch node

T he scenario gets more complicated within the switch nodes, as shown in figure 6.3. In contrast to the host node, switches have multiple network interfaces, a level table and RT component, but no source/sink or Ethernet converter. Each enclosed group of rectangles corresponds to one network interface. Each element of the NI has a suffix corresponding to the local id of that interface. As for the host node in figure 6.2, there is a NI_in and a NI_out as well as a queue. The simple droptail queue is here replaced with a *buffer budget queue* associated with a local counter (bbc) and global level table, and a FIFO queue of variable size. Queue functionality and implementation is given below in section 6.4.3.

6.1.3 Queuing and processing of arriving frames

I n general we use the input/output queue scheme shown in **??**. The location within the actual simulation environment uses physical output queues, one per NI_out. However, reservations are done through the buffer budget counter based on NI_in id and the destination address of the frame.

W ithout flowcontrol enabled, control frames are ignored and droptail queues are assumed to cause frame loss when congestion builds up. The processing of an arriving frame at some NI_in is illustrated in figure 6.4. This processing is consistent with the *MAC Control Receive state diagram* given in [69, clause 31].

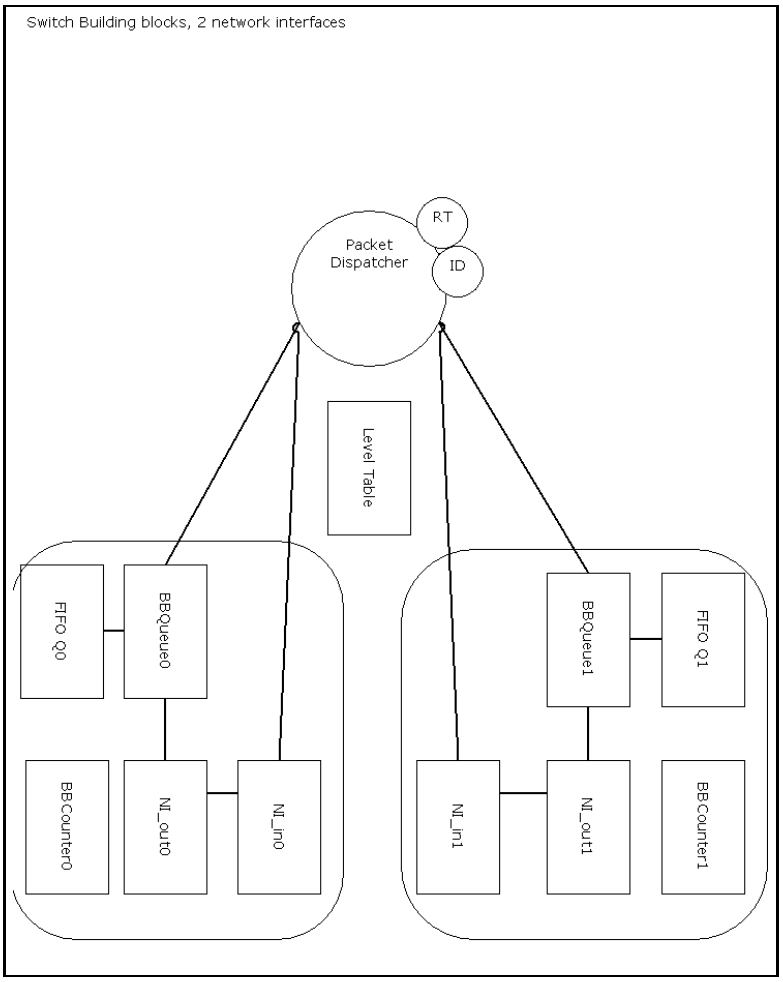

Figure 6.3: Switch node with 2 network interfaces, level table and packet dispatcher with identification and routing services

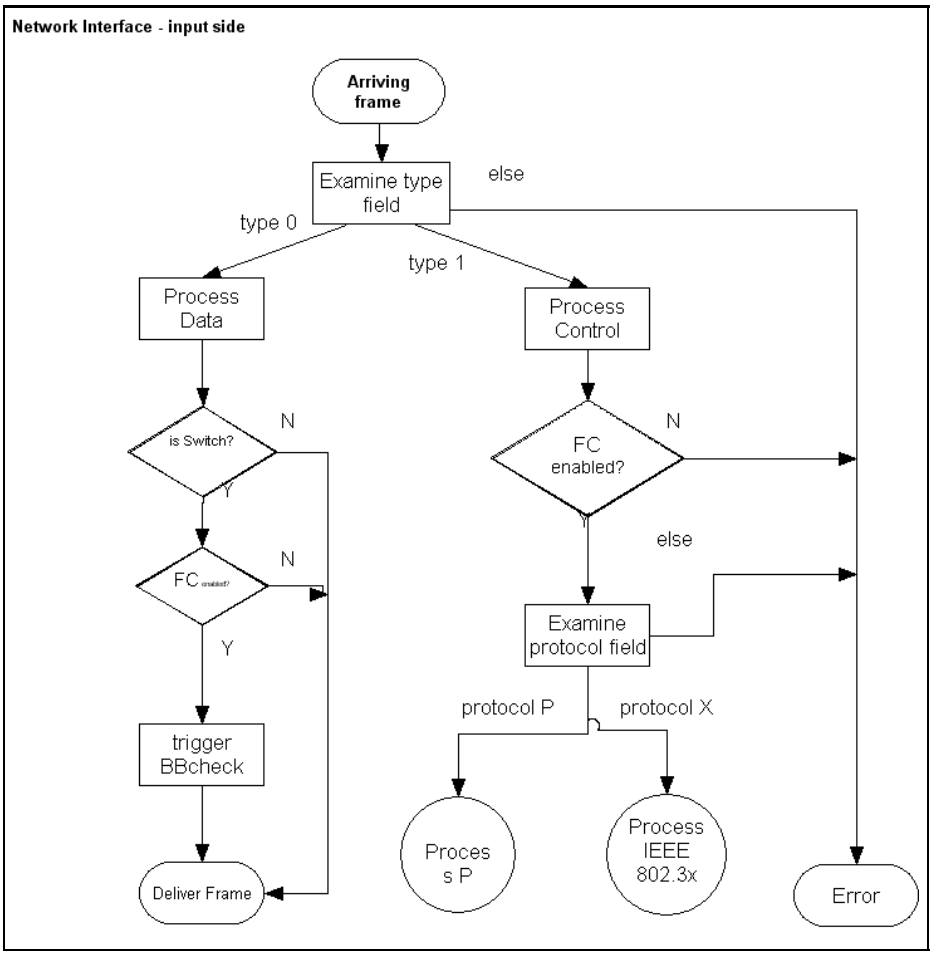

Figure 6.4: Processing of an arriving frame at the in-port of a network interface (host or switch)

D ifferent tests are applied to the received data, starting with determining if it is data or control frame (type field check). Since data frames are to be sunk within host nodes, and enqueued to be forwarded in switch nodes, the flowcontrol and buffer check only apply to switches. On the other hand, control frames are only valid if flow control is enabled, the received frame is well-formed and contain one of the correct protocol field values. The control frame is then further processed depending on the protocol, either IEEE 802.3x or protocol **P** , described below in the implementation sections.

6.2 Our contributions to the J-Sim component hierarchy

I mplementation has been an important part of our work, and we would like to specify our J-Sim contributions. The following Java classes have been developed for this project:

- EthConstants
- EthFrame
- PktModifyer
- P2PNIPIn
- P2PNIPOut
- BufferBudgetDropTailQueue
- BufferBudgetConstants
- BufferBudgetCounter
- LevelQElement
- VSFIFOLevelQueue
- PNodeBuilder
- ProtocolPBuilder
- AddressModifier
- PStatCollector

6.2.1 Ethernet frame and packet modifier

T he Ethernet frame class, `EthFrame`, is a subclass of the built-in `Packet` class of J-Sim. We have included `framesize`, `bodysize`, `seqnr`(frame se-

quence number), `type` and `pauseTime` to achieve desired Ethernet properties.

T he type field corresponds to the protocol opcode field for MAC control. We have in our implementation added an opcode for protocol **P** in order to be able to distinguish it from standard MAC PAUSE control. In a real hardware implementation, this simplification would cause problems for a network consisting of a mix of **P** enhanced and regular switches. The challenge is that the regular switches would not recognize the new opcode and discard all those control frames, leading to flow control effectively being prohibited between heterogeneous switches, at least for the duration it takes the **P** enhanced node to detect the lack of response and resort to standard PAUSE behavior. Such adaptivity would require additional control logic in the interface.

T he packet modificator, `PktModifyer` ,extends `drcl.net.Module` of J-Sim and is used for the Ethernet converter (shown in figure 6.2), with the single purpose of creating a new well-formed Ethernet data frame (`EthFrame`), and inserting it to the body of the `InetPacket` that is to be injected to the network. This breaks with the well-established scheme of encapsulation, by wrapping the lower-level protocol data inside, and not around, the received SDU.

T his behavior is the result of an implementation choice related to J-Sim using InetPackets as the data unit in the simulation. In order to preserve proper encapsulation, we would have to rewrite extensive parts of the simulator. By ensuring that our special-purpose components developed for network interfaces and buffer management uses the info stored within the Ethernet frames, we obtain a stronger degree of code reuse with respect to well-proven J-Sim components, and a lower implementation cost.

C omponents for network interfaces and buffer management are described in the implementation sections below. The builders are explained in section 6.7, the address modifier in section 6.6 and the statistical component is introduced in section 7.2 of the following chapter.

6.3 IEEE 802.3x implementation issues

T he basics of IEEE 802.3x was explained in section 4.5. This section deals with how MAC Control is handled in our specific implementation. We have in our Java code, as well as in the following text, adopted some general implementation tips from [67]. These tips are given high confidence and weight due to Seiferts strong involvement with the IEEE related to the topic domain.

6.3.1 PAUSE timing

P AUSE timing is crucial in order to ensure effective flow control. When the MAC Control sublayer receives a valid PAUSE control frame with a non-zero pause_time, an upper bound has been set on the length of time that interface is permitted to transmit data frames before it must halt [69, annex 31B]. This upper bound serves a twofold purpose: it ensures that PAUSE control requests are served in a timely manner, and second, it allows completion of an already submitted transmission. Moreover, this response time is measured in *pause_quantum*: a NI operating at 100Mbps or less, is restricted from starting transmission of a data frame 1 `pause_quanta` after reception of the control frame. Operating at speeds beyond 100Mbps, the upper bound is extended to 2 `pause_quantum` [69, annex 31B], [67].

6.3.2 Selecting values for pause_time

S election of pause times is left to the implementer/vendor and values between 0x0000 and 0xFFFF are available ("2-octet unsigned integer" [69, annex 31B]. A simple choice of `pause_time` is to use the latter as a *XON* flow control assertion signal, and the former as a *XOFF* cancel signal in line with the standard. In this case, when a congested buffer that empties more rapid than the `pause_time` indicated and drops below the low water mark, action should be taken to issue an explicit cancel message to prevent unnecessary idle operation that would otherwise follow from waiting for the timer to expire.

F or such short-term scenarios, each triggering of flowcontrol has the overhead cost of an additional control frame to lift it. On the other hand, a switch that is part of a heavily congested subset of the network, might need several re-invocations of XON, and the longer the `pause_time` in this latter scenario, the fewer control frames need to be sent and processed, reducing the overhead compared to shorter time intervals. Hence, selecting

the non-zero `pause_time` can be done based on simplicity, desire for more control granularity, and/or desire to minimize control overhead.

M AC PAUSE Control is commonly implemented in hardware due to the tight time constraints[67]. If keeping a well-formed control frame like described in paragraph 4.5.3, keeping to the minimum and maximum values of `pause_time` would add to implementation simplicity.

P lacing an upper bound on the `pause_time` and the use of a timer in the MAC Control sublayer, ensures that permanent disabling of a NI cannot occur, even if the XON control frame gets lost or corrupted. Despite this safeguard, some has advocated to eliminate the `pause_timer` and use the XON/XOFF as a simple light switch that remains in a position until flipped. This may simplify implementation on some areas like the need for timer and periodic testing for whether an issued XON has to be prolonged by sending a subsequent XON message. However, caution has to be made to prevent flow control deadlocks, in which a NI is constantly prevented from transmitting.

6.3.3 Flow control responsiveness and buffer requirements

B uffering additional data is required when using the PAUSE flow control because link propagation delay and MAC Control response time prevent instant reaction to a control frame. A list of worst case amount of additional data a NI might receive after issuing a PAUSE request before the link-partner halts transmission, is given in [67]:

- 1 MTU frame on transit
- 1 PAUSE frame time
- PAUSE frame decode time (1 or 2 pause_quanta)
- 1 MTU frame on receive
- 1 link RTT

E xcluding the RTT measured in bytes, this totals approximately 3.2KB. A 100m 1Gbps UTP cable has a RTT of 143 byte. 10Mbps and 100Mbps links have RTT in corresponding orders of magnitude less. Following, in the LAN/SAN context a *buffer margin* of 3 MTU for triggering flow control should be sufficient to prevent buffers from overflowing. Shortening the links yields lower RTT. For other link types, like the 1000BASE-LX, a

larger margin is required. The exact design of buffer organization and -management within the switch also influence the buffer margin choice. For example, a pure output queued switch may have to allow for this amount of data to arrive per incoming interface in order to preserve lossless operation.

I n the host nodes a simple droptail queue is used with a capacity of 2 MTU (3044B) without any buffer margin, since these queues handle only frames from the traffic generator, not from the network. In other words, host queues do not need margins since no traffic from within the network is buffered there.

6.3.4 Selecting threshold values for PAUSE actions

T hreshold selection for asserting and canceling flow control is tied to the above described buffering requirements of input queued switches. The buffer margin arrived at can be used to avoid buffer starvation as well as accommodating additional traffic, as illustrated in figure 6.5.

B y issuing PAUSE control when the buffer occupancy exceeds a preset high water mark, at least buffer margin less than the total available buffer, and canceling PAUSE when buffer occupancy drops below a preset low water mark, with least a buffer margin worth of data still in the buffer, the NI will always have enough, but not too many, frames in order to operate continuously (and hence optimal). Buffer management for PAUSE control uses partially the same components as protocol **P** , and is described in section 6.4.3 below.

6.3.5 Parsing IEEE 802.3x

F igure 6.6 illustrates how standard IEEE 802.3x control frames are parsed. This scheme connects with figure 6.4. The on/off state variable for flow control is set in the outgoing interface (of the current link) to 1 or 0 depending on the received value of pause field. This corresponds to a simple XON/XOFF scheme. Utility methods are called to ensure scheduling reacts to the control message.

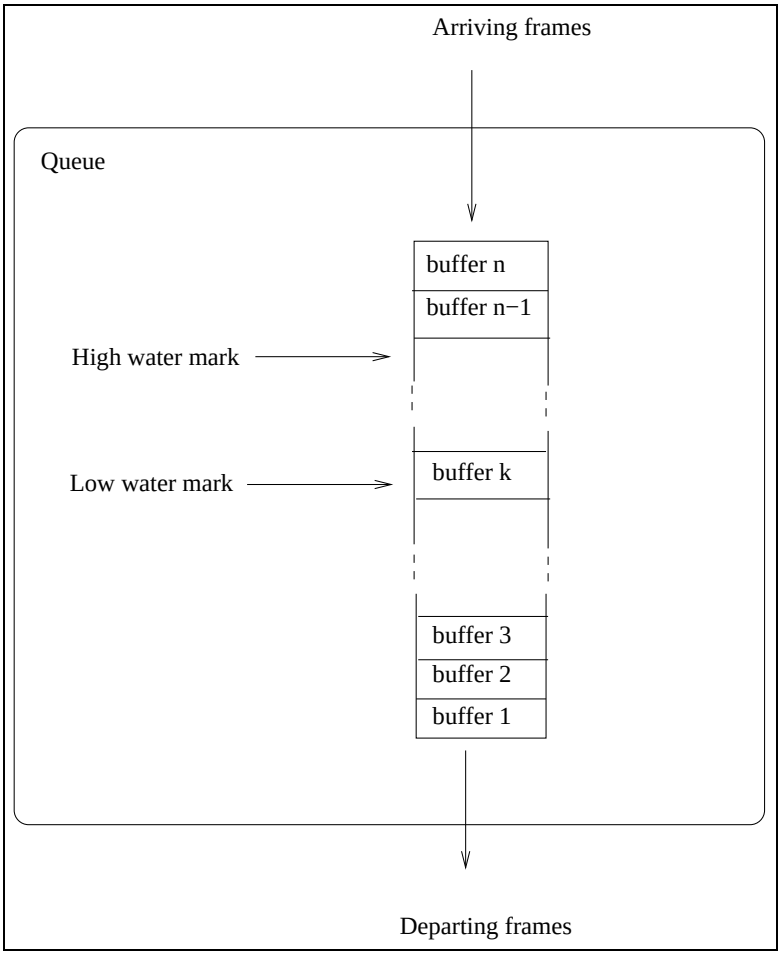

Figure 6.5: Buffer thresholds with high/low water mark

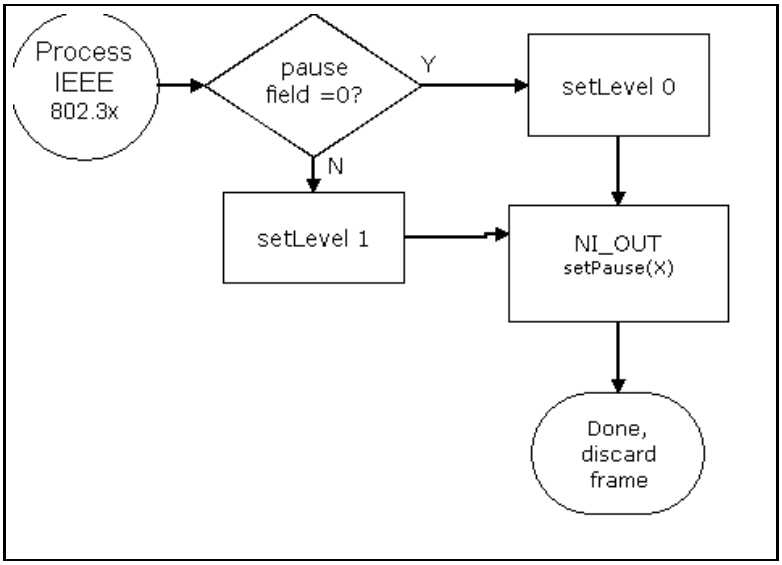

Figure 6.6: Parsing IEEE 802.3x: the on/off state in the interface is toggled based on pause time in the received control frame.

6.4 Protocol P implementation

T his section addresses the implementation issues of protocol **P** in our simulation environment, focusing on processing and the components needed in the simulator to administrate the protocol.

6.4.1 Parsing protocol P (received control frame)

P arsing protocol **P** control frames at the receiving NI_in contrasts to handling arriving data frames and monitoring buffer occupancy, and we therefore handle these issues separately. Within the implementation, these issues are handled by dedicated methods in the components.

I n order to preserve backward compatibility with IEEE 802.3x, protocol **P** needs, as described in 5.6.1, to use control confirmation frames when sending transmit feedback. Consequently, housekeeping must, to ensure proper operation, take not only the current frame, but possibly also the previous frame into account. In other words, the first sequence of checks for a valid, well-formed control frame is done to establish whether the current frame is the main or confirming frame of **P** , and also if there is a chance that the control frame originated from a node that is not **P** enhanced, in which

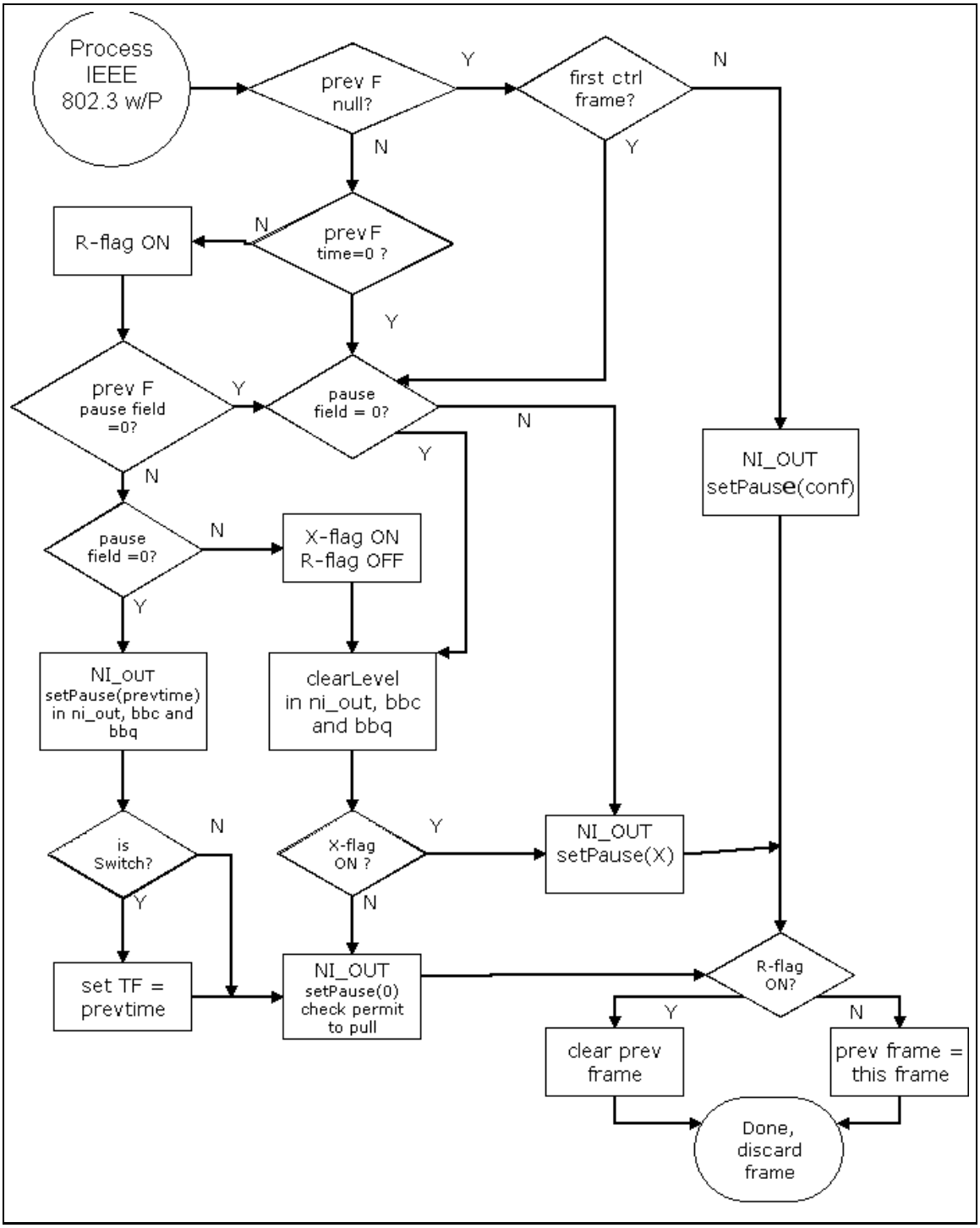

Figure 6.7: Parsing control frames of protocol **P** . F = frame, TF = transmit feedback

case the NI must fall back on standard PAUSE control operation. Basically we distinguish between four different cases:

- **P** control frame, specifying transmit feedback
- **P** control frame, confirmation
- standard IEEE 802.3 PAUSE control frame
- error

T he *R-flag* is used to decide if the current frame should be stored for reference after it is parsed, or the old reference should be cleared. The *X-flag* indicates whether flow control should be activated according to the simple on/off scheme. The actions in the lower part of the figure are related to adjusting transmit feedback level and pause status according to the received control frame. We now move on to examining the reception of data frames.

6.4.2 Incoming interface operation (received data)

I ncoming data frames are processed according to the scheme shown in figure **??**. A flow control enabled switch is the only non-trivial case here, and it is directly related to buffer management. The *buffer budget check* is preformed regardless of the type of flow control used, but the action outcome of that check is protocol dependent, leading us to buffer partitioning and queuing.

6.4.3 Partitioning the buffer pool

T his section explores the buffer budget check and the engueue / dequeue procedures. We start with layout of how we have organized the buffers. Default switch memory is calculated based on an estimated hop limit, MTU size, amount set of for b_1 and the port count of the switch.

> The buffer size b^l is only weakly dependent on the maximum route length D . Most of the buffer space is in b_1 - i.e., b_j is very small for j > 1 - and the partitioning is ”virtual”. The buffer space set aside solely for higher level packets is only used when congestion occurs and a small amount of space is needed to prevent deadlocks/livelocks.[43, pg. 928]

I n our implementation the main queue type for switches is a `BufferBudgetDropTailQueue` (BBQ) extending `drcl.inet.core.Queue` of J-Sim. This is associated with an array of `BufferBudgetCounter`s, one for each incoming interface of the switch, as well as a `LevelTable`, a `VSFIFOLevelQueue` and a current value of transmit feedback. A BBQ contains enqueue and dequeue procedures, along with utility tools for managing the level table and enforcing the eligibility rule of protocol **P** , as explained in section 5.3.

W e have defined default buffer budget constants in a Java interface class. Exact used values are set during build for each buffer budget counter (BBC). The amount of memory required for a switch that is protocol **P** enhanced, using virtual input-output queues with reservation regulated per incoming link, totals to the allowed memory per NI_in multiplied with the number of interfaces at that switch. This can further be specified per interface as the size of the shared buffer pool b_1 added to the product of the hop limit D and dedicated buffer b_i. Using 1 MTU at for all i > 1, we see that the additional memory requirement by **P** to solve store-and-forward deadlocks, is a direct consequence of network topology, while standard PAUSE operation does not come with this dependence.

6.4.4 Managing queues

A buffer budget check is triggered in a flow controlled switch by an arriving data frame, or a deactivation timer (DF). The outcome is, as shown in Figure 6.8, setting flow control to on / off, unless an error occurred. The *A-flag* indicates if the outgoing interface is, or should be set to be, in an active state capable of sending frames.

I n the case of timeout of the deactivation timer, the queue is examined if the flow control is still on. The queue is also automatically examined when new data frames arrive. We have used the following queue states: above high threshold, below low threshold, or between marks. The second case unconditionally leads to flow control being deactivated, as there is no need for throttling the sender any longer. In the case of buffer occupancy being somewhere between the marks, status qua is preserved. This implies that if the flow control was on, it will be kept on and timers possibly reset. If the flow control was off, it will not yet be asserted, but arrival of subsequent data frames might trigger its onset.

T he most complicated issue is when the queue is filled beyond the high threshold level, and this is directly connected to protocol differences. For

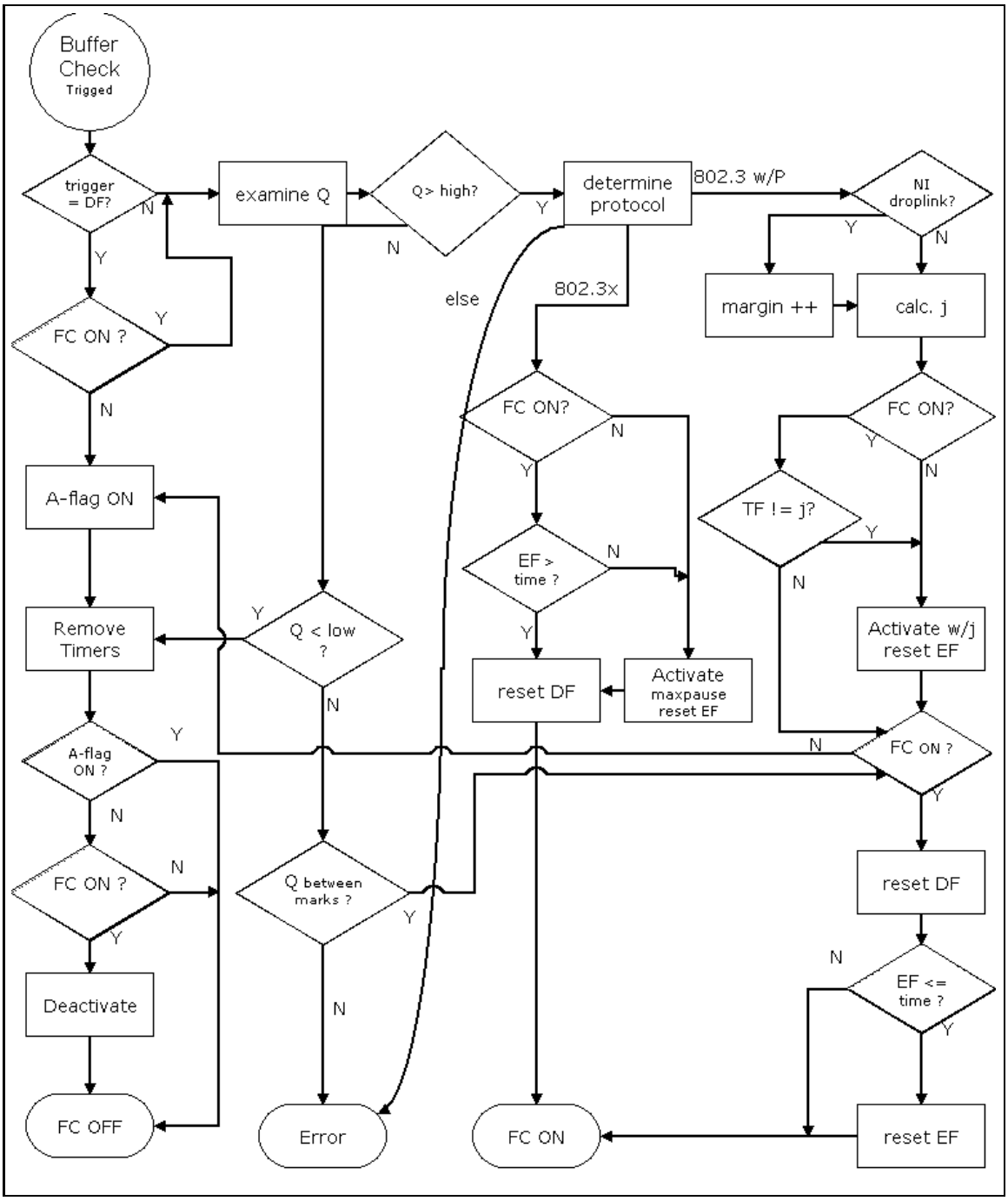

Figure 6.8: BB check

standard PAUSE operation, existing pause state is maintained with refreshed timers, or a beginning pause is triggered. In the case of **P** , transmit feedback (j) is calculated and compared to the previous value. If flow control was currently off, or calculation arrived at a different transmit feedback than previously transmitted, a notification is sent to the upstream node. As for the PAUSE case, timers are updated to ensure proper expiration and deactivation of ongoing flow control action to avoid flow control deadlocks.

Enque and deque behavior

E nqueuing a frame is done at the outgoing side of an interface (OQ), but based on info about destination address and incoming interface (IQ), and set for the BBC of that NI_in. IEEE802.3x uses `plevel` 0 and `qlevel` 1. Protocol **P** assigns these values based on info from the level table and transmit feedback j. Frames are accepted to the queue if and only if the queue is not full and there is free space at `qlevel`. All frames exceeding available buffer space are thrown away unconditionally (droptail behavior).

D equeueing is done by obtaining a level-key, finding an eligible frame in a FIFO manner, and afterwords updating the bbc and availability parameters. In other words this is more or less straight forward given the key. Protocol **P** 's eligibility is expected to impose lookup costs on long queues. This issue will therefore be included in our analysis. For IEEE 802.3x all packets have the same level and thus only FIFO principles apply, reducing the frame retrieval cost.

6.4.5 Outgoing interface operation

T he complexity of the buffer management imposed by protocol **P** is mostly hidden from the outgoing interface. In other words, it operates as if it were a normal flow control enabled network interface capable of scheduling frames from a queue and adhering to PAUSE actions. Eligibility is managed within the queue, upon a dequeue request. In our opinion, when a NI_out issues a dequeue request, it is at that point committed to sending the received frame without unnecessary delay.

6.5 Topologies and routing algorithms

W e have used 16 different irregular topologies, each consisting of 16 switches and 64 hosts. Each switch has 8 ports, four of them for drop-links

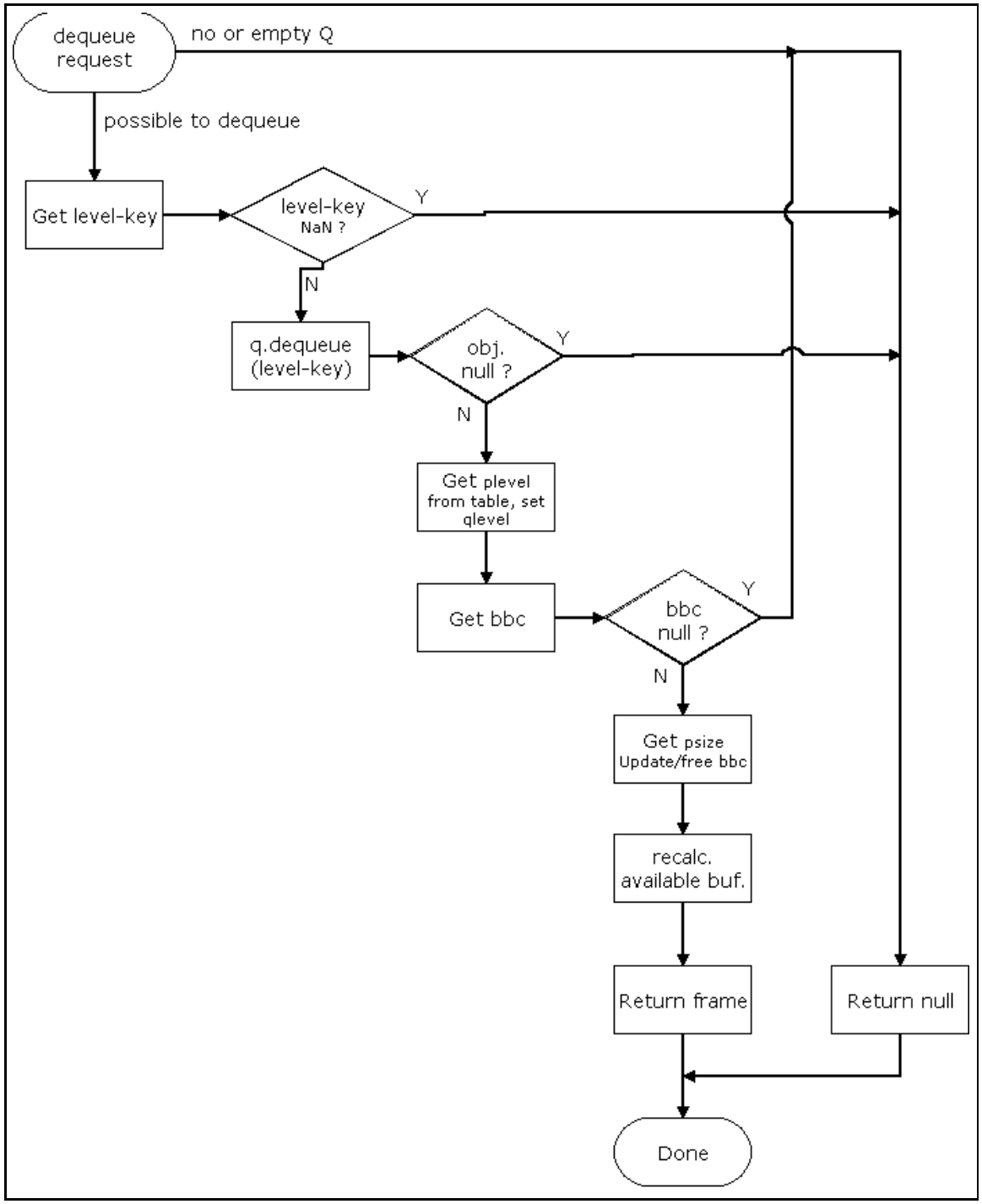

Figure 6.9: Dequeue

connecting to hosts, and the remaining for inter switch connections. A connection table is used as input to create an adjacency matrix by built-in J-Sim tools. `drcl.inet.InetUtil.createTopology` is then utilized to build a spanning tree.

A s seen in B, the `buildSpanningTree Topology` procedure used `drcl.comp.lib.MSTKruskal` to prepare the adjacency matrix. After the development phase and simulation runs with the 16 irregular topologies, we learned that this feature has been become deprecated / unavailable. Attempts to reproduce our old scenarios have failed, and consequently we have to admit that without time to conduct a second implementation phase, we are stranded with the data so far collected. Additional scenarios of interest include relaxing the spanning tree with up*/down* routing and the TBTP scheme. We have included these in the section about future work at the end of this thesis.

T he routing tables, one within the RT component of each switch node, are populated by the `setupRoutes` method in `drcl.inet.InetUtil` in a bi-direction manner between all possible switch pairs. Source-destination pairs are handled by the `AddressModifyer` component described later.

6.6 Workload parameters

6.6.1 Protocol stack issues

D ealing with an unreliable data link layer should be addressed when transmitting PAUSE frames[67]. Despite the low BER, a control frame might be corrupted, and because they are sourced and sunk within the MAC, no higher level protocol or application is available to deal with the loss. Adding this to our simulation environment would add complexity to the network interfaces implemented and buffering used, without providing significantly more information in the results. Hence, in line with our considerations around abstraction level in section 2.2.1, we choose to assume reliable links in our implementation.

T he protocols normally impose a limit on the size of a SDU, and hence mapping a SDU to smaller or larger data unit blocks commonly done concurrently to encapsulation. This is not of importance to our study, and we use a 1:1 mapping on all layers and in all scenarios as it does not contribute

to any variations in behavior nor performance between the protocols studied. Further, MTU is set to 1522, a maximum well-formed valid Ethernet frame.

6.6.2 Traffic generator

W orkload in our simulations is overall handled by `drcl.net.traffic.traffic_PeakRate` of J-Sim. This class generates traffic based on packet size and inter arrival time of those packets. Separate maximum and minimum values can be specified, however for strict reproducibility concerns we have chosen to use a fixed value in stead of a range.

6.6.3 Link speed and injection rate

W e have used a fixed wire speed of 1Gbps (1000Mbps). Both hosts and switches operate at wirespeed. Using a inter frame gap of 96ns, the resulting injection and sending datarate of the network interfaces is 0.000012. All links are set to 10m UTP resulting in an interrouter propagation delay of 1.14e-6 (cable holds 120 bits).

6.6.4 Source-destination pairs and address distribution

R outes are set up by using J-Sim functionality through Tcl script functions as previously described. Addressing is handled by the address modifying component, `AddressModifyer` (AM). This component is located between the source and the packet dispatcher of the host nodes. It was originally written by Svein-Arne Reinemo as an university in-house tool.

T he AM has three different but related tasks. It wraps the raw packet from the traffic generator within an `InetPacket`, it sets the source and destination address within the packet, and it sets the time to live (TTL) field. The source address is the id of the current host node. We have used a uniform address distribution in our final runs. The destination address is consequently calculated with the nextInt feature of Javas random number generator, `Random`, limited to the range of available hosts. The TTL is set to one more than the number of switches in the network, in our case 17. The packet dispatcher of J-Sim decrements this value when it relays packets from an incoming interface. The worst case scenario is a topology with all switches serially connected. The +1 assures that the packet is not discarded in the packet dispatcher of the sink host in such a worst case scenario.

6.7 Setting up simulation with J-Sim and Tcl

S imulations are with J-Sim run through the built-in *RUntime Virtual system* (RUV) on top of Tcl/Java:

```
java drcl.ruv.System ?<script>? ?<argument>...?
```

Information about the special-purpose scripting environment can be found at the J-Sim webpage [86].

W e have run the simulations with a duration of packet injection set to 2, and a finishing time set to 1000. The rationale for choosing these exact values were based on memory restrictions and observed behavior in test simulations. The finishing time needs to be fine-tuned to the duration of packet injection phase so that we ensure all packets have been drained before halting the run.

6.7.1 Datarate parameters

D atarate constants for use within Java code are defined in EthConstants.java, and can be overridden by scripts:

Real Time Datarate : 1.0e9 (1Gbps) as default, with 100 and 10 Mbps options available

Time Scale : proportion simulation vs real time (J-Sim), 1.0e3 as default

6.7.2 Ethernet constants

W e have used standard compliant values for our Ethernet constants

MTU : maximum 1522, minimum 64 byte

BODY : 1500 / 42 byte

HEADER : 18 byte (InetPacket adds 4B overhead, yielding 22 byte in total)

BIT_TIME : 512 (multiplication factor)

MAX_PAUSE : 255 (8bit value)

Inter Frame Gap(IFG) : 96 bit times

Propagation Delay : 1140 bit times

6.7.3 Automatic builders

To build the network topology and its nodes, J-Sim provides automatic node builder tools. We have implemented our own version, the `PNodeBuilder` extending `drcl.inet.NodeBuilder` of J-Sim, and using our own CSLBuilder `ProtocolPBuilder` in the build method override. This gives us full control of the components and their relation within the CSL in each host and switch node.

6.7.4 Bash and Tcl scripts

S etting up and tuning parameters of simulation scenario is best illustrated by exploring the scripts we used for the simulations. A complete and general version of these can be found in appendix B. We here only include selected features significant to our scenarios. The general simulation setup script simply creates all needed output files, and stitches together all possible combinations of the topologies, flow control options and different data rates, and enters these parameters into the master scenario script listed in B.1. The scenario template also uses various utility scripts that contain subroutines and constants, some of them already described.

I n order to enhance customization, several Bash (bin/sh) scripts were developed to automatically create log fies, per simulation scenario Tcl scripts and submit files for Condor. *Condor* is a "a specialized workload management system for compute-intensive jobs"[73]. This system was available and used at campus for our main simulation runs. The author can be contacted for further information and re-purposing of scripts.

Chapter 7

Analysis

W ith the background so far given in this thesis, we now report on our data collection process and the outcome of that. The research goals stated in the introduction, play an important role in our analysis. We compare the different flow control scenarios, re-visit issues related to deadlocks and routing, and discuss buffer management complexity. The chapter closes with a critical view on our own research, sources of errors and simulation related memory challenges.

7.1 Performance measurements

A ccording to our research goals we have measured latency, throughput and packet drop, as well as observed deadlock and livelock symptoms.

I n a simulation environment researchers have control over both switch based and external variables, whereas in a real life scenario with vendor produced hardware components, we have influence only over the latter. Traffic pattern and injection rate is consequently the easiest factor for a researcher to manipulate, and this was described in section 6.6. In summary, our scenarios uses a fixed high data injection rate, with sources injecting packets of 1 MTU (1522B) to the network continuously with only a constant inter frame gap (and the time it takes to shuffle the bits onto the link), only halted or throttled by flow control actions.

T he switch based parameters include switching method, flow control protocol, network connectivity, routing algorithm and buffer layout and

management, all of them previously examined. For the clarity of the following analysis, we briefly list the selected parameters:

- Store-and-forward packet switching
- IEEE 802.3x PAUSE flow control / protocol **P** enhancement to PAUSE
- Irregular topologies with 16 switches (4 inter switch links and 4 drop links) limited to a spanning tree, 64 hosts
- Destination based Ethernet routing tables, ensuring that all frames between a source-destination pair always follow the same path
- static routing, no dynamic updates occur
- single droptail output queue in hosts, advanced switch buffer management with flow control actions taken based on input queue while actual queuing happens in an output scheme. Packets are associated with a packet level.

7.2 Data collection

D ata collection is in our studies done by running the simulations with an associated component for collecting results, and dumping information from that component to log files when the simulation terminates. These log files can then be used as input to plot generation tools.

7.2.1 Running the simulations

F ollowing a long implementation phase, simulations were set up and submitted batch-wise to the Condor system at campus (Ifi). The local system administration group has noted that Condor has problems related to estimating memory requirements of a Java job prior to running it, especially in the presence of multithreading[34]. The solution is to specify memory requirements in the submit file. From trial and error we arrived at 256MB as a minimum. Observations were made of extreme memory consumptions of our jobs. We have therefore dedicated a section in our analysis to memory related issues.

7.2.2 PStatCollector

T he `PStatCollector` extends the general built-in component class of J-Sim, and is a customized version of an earlier in-house component, the

`drcl.net.StatCollector`. A single instance of this component is hooked up to all network nodes, and receives data about all frames sent (injected), received (drained) and dropped. Results are recorded continuously during a simulation run building a statistic per node as well as in total.

For measuring throughput we simply count packets in a two dimensional array. Latency is recorded by adding up the latency value of each drained InetPacket, and keeping a count of packets that have contributed to this accumulated value.

7.2.3 Dumping results

At the termination of each simulation run, data on throughput, packetloss and accumulated latency are dumped from the `PStatCollector` to log files. The dump is triggered form the main Tcl script for each simulation scenario using the `dumpLatency` and `dumpThroughput` procedures listed in B.2.

Due to the high number of simulation runs arrived through all possible combinations of topology, datarate and flow control type, we developed bash scripts to parse these log files and generate data input files for Gnuplot [1].

7.2.4 Introduction to plots

We have in general two different types of graphs; *Throughput graphs* and *latency graphs*. Other figures are special purpose variants of these used to high-light a point of interest.

To get the best possible picture of each combination of topology and flow control type, we performed some initial simulation runs using data rates of coarse granularity to learn where to invest our computational resources. The data rates range from 10 to 1000 Mbps, with measures taken at 10Mbps intervals between 10 and 100, 20Mbps intervals between 100 and 200, and subsequently 100Mbps intervals up to the 1000Mbps limit.

In all result plots, we use a scale of 100Mbps at the horizontal axis unless specifically stated. Vertical axis for throughput graphs use number of frames as measuring unit with intervals of 100000. Latency graphs use seconds at the left axis, with intervals adjusted to the granularity of the

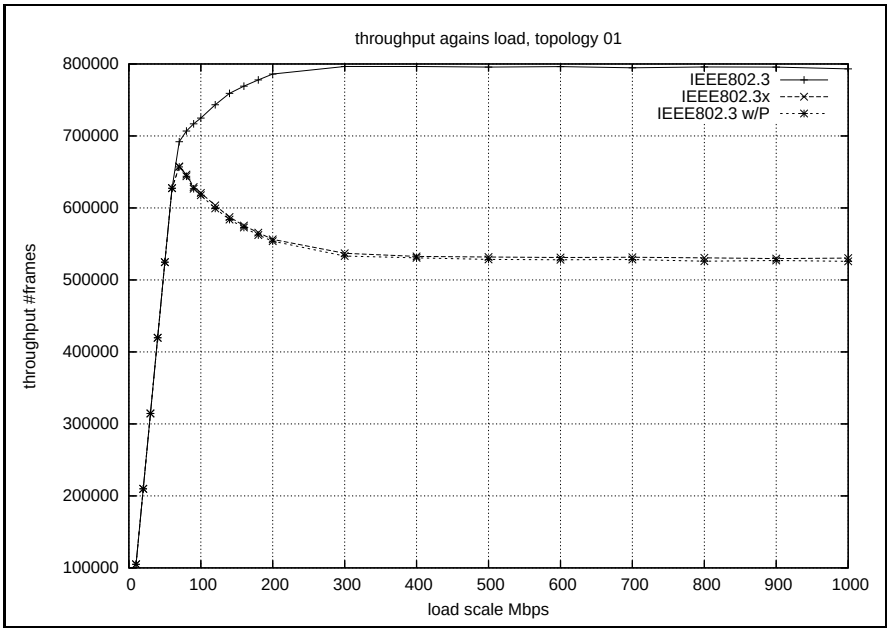

Figure 7.1: Throughput for a single topology for each flow control mode

results plotted (ranging from 0 to the highest latency observed in those results). Description of the different line styles is included in the upper right corner of each plot.

7.3 Presentation of results

F igures 7.1 and 7.2 shows graphs of throughput and latency for a selected topology. For the clarity of presentation, packet drop has been deliberately omitted in these figures. We start by noting that there is a striking difference between standard Ethernet with no flow control on one side, and the two flow control schemes on the other side. In fact, IEEE 802.3x and protocol **P** is so close in this plot that it is hard to distinguish the one from the other.

T his topology was randomly selected out of the 16 used. Our first question, before examining it in detail, is whether or not it is representative. To answer this, we have compiled plots illustrating the variation in both throughput and latency across topologies fore each of the tree flow control modes.

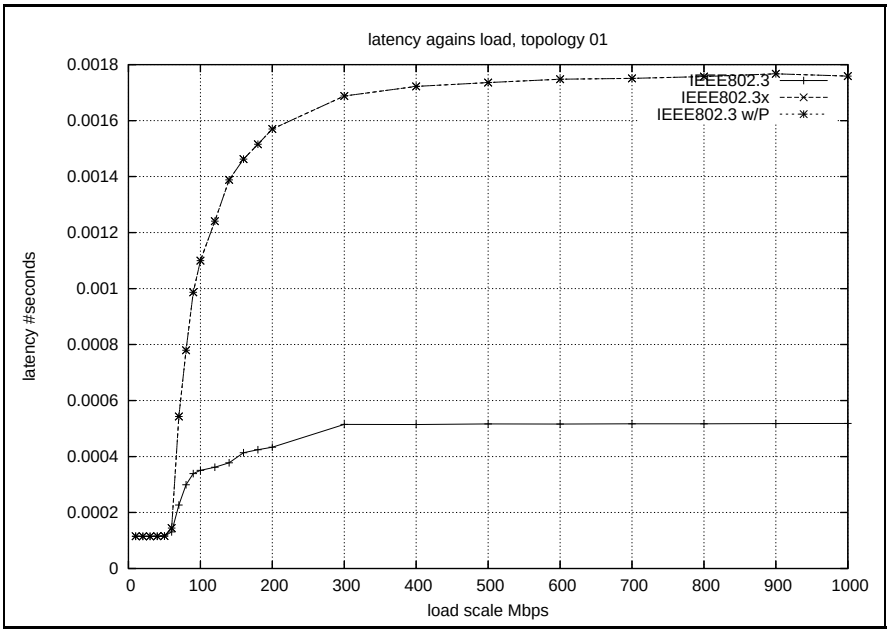

Figure 7.2: Latency for a single topology for each flow control mode

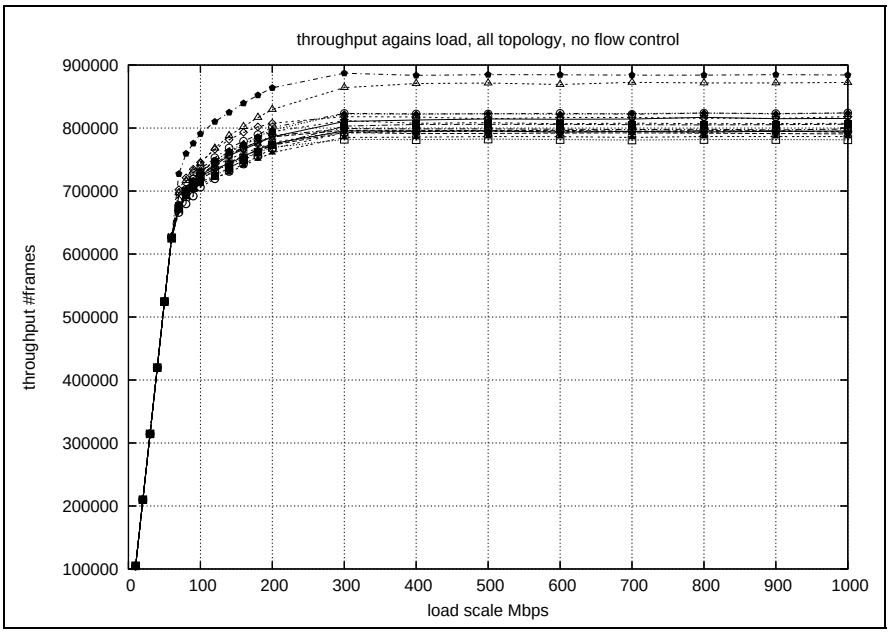

Figure 7.3: Variation in throughput without flow control

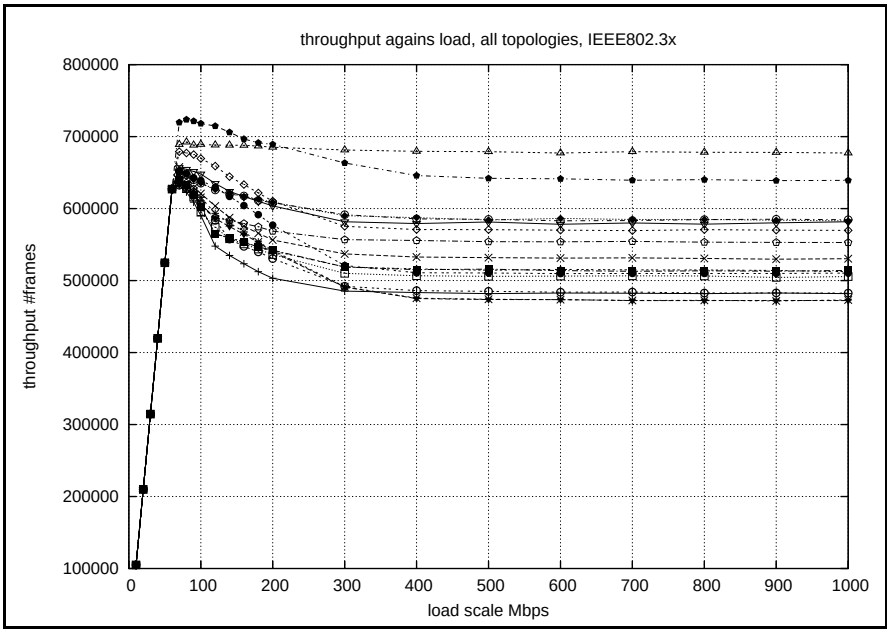

Figure 7.4: Variation in throughput using IEEE 802.3x flow control

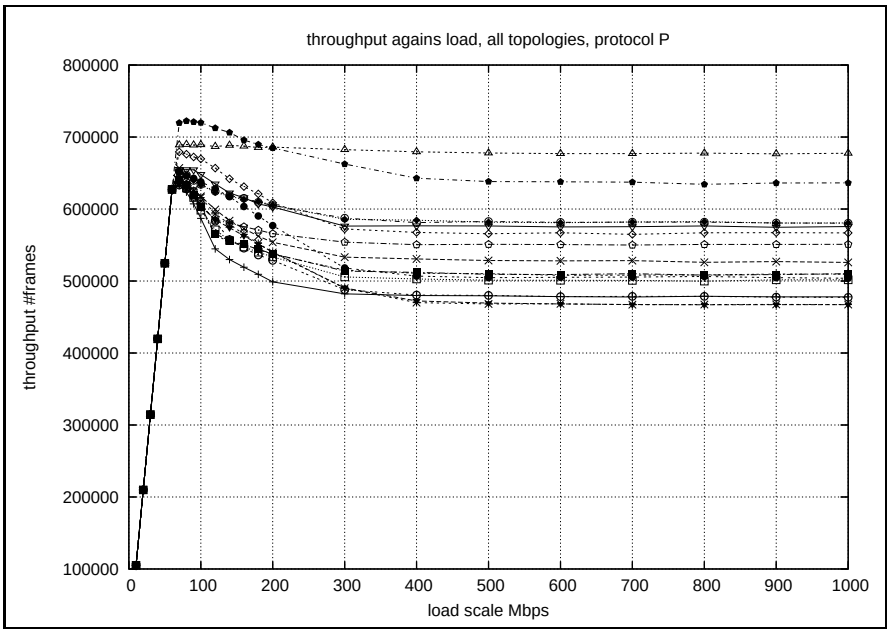

Figure 7.5: Variation in throughput using protocol **P** flow control

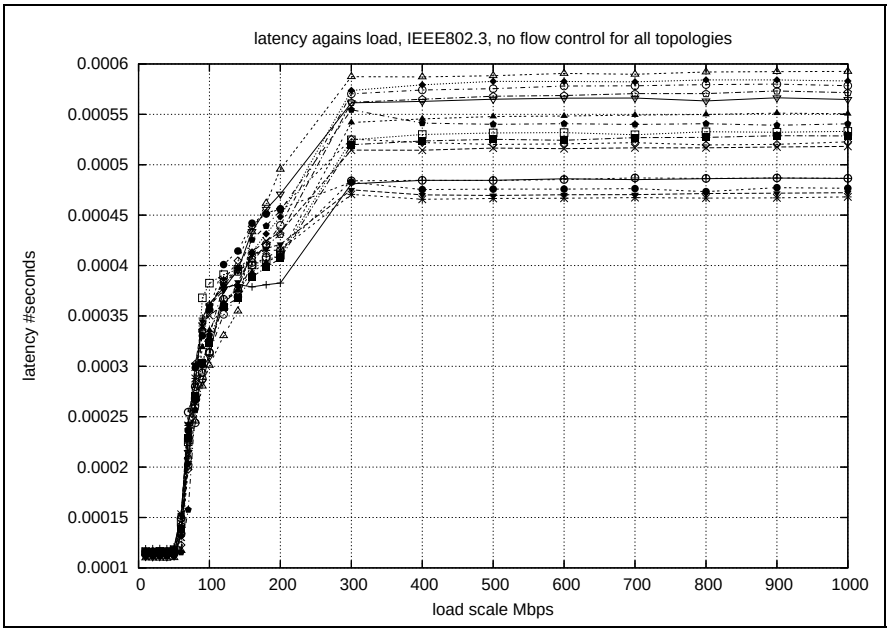

Figure 7.6: Variation in latency without flow control

F igure 7.3 shows throughput variation across all 16 topologies in the absence of flow control, while figures 7.4 and 7.5 show this for scenarios using either IEEE 802.3x or protocol **P** flow control. We observe that the curve shapes are strikingly similar for the two latter, while the no flow control scenario differs.

B ased on the material presented so far, we note that the difference between the three schemes starts to emerge around 60Mbps. Moreover, for all schemes, throughput seems to stabilize and remain at a specific amount around 400Mbps.

F igures 7.6 through 7.8 show latency variation for the same three cases as the previous illustration set. We observe again the curve shape similarities for the two flow control cases. Unlike seen for the throughput, the latency does not stabilize entirely for the flow control schemes, but increase extreme slowly with the increasing data rate.

I n the absence of flow control, shown in figure 7.6, we observe almost parallel constant lines. Although the left part of this figure looks like figures 7.7 and 7.8, there is one important difference, namely the values in the vertical axis. Without flow control the highest latency seen is less than

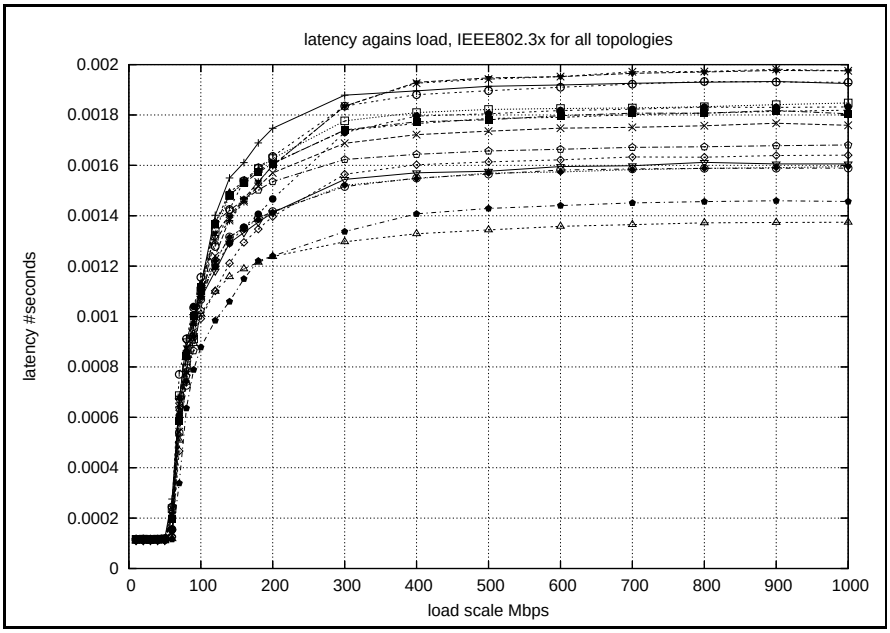

Figure 7.7: Variation in latency using IEEE 802.3x flow control

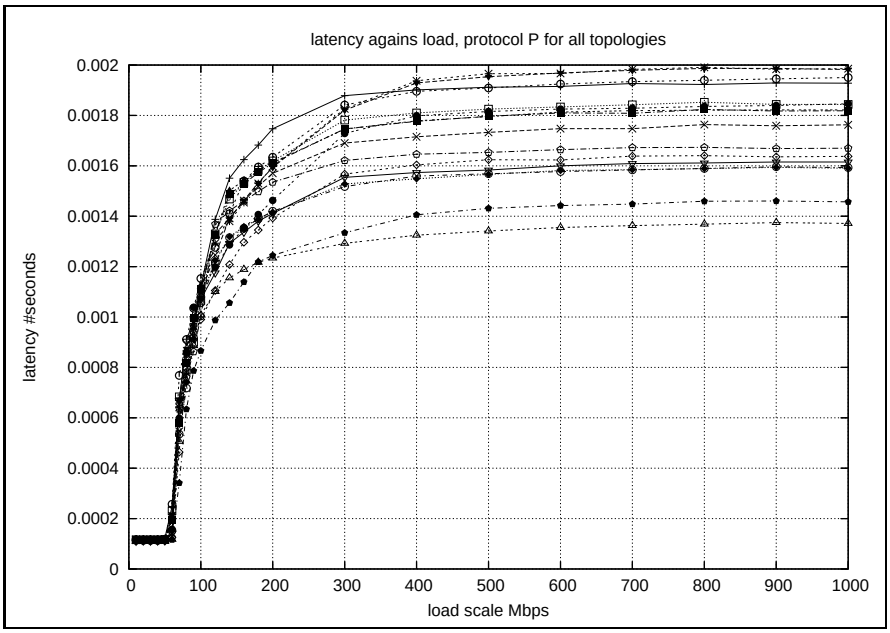

Figure 7.8: Variation in latency using protocol **P** flow control

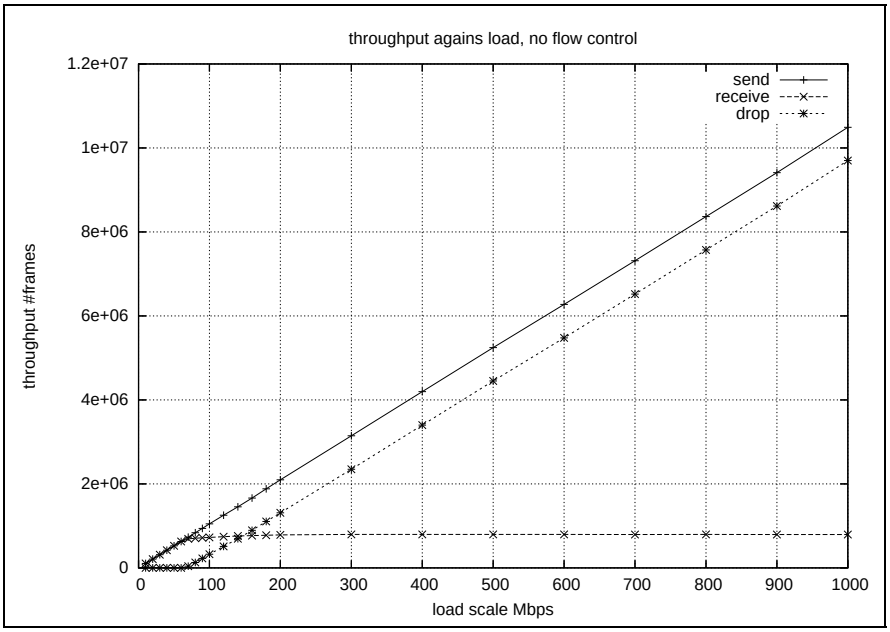

Figure 7.9: Sent, received and dropped frames in the absence of flow control

0.0006 seconds, while in the flow control scenarios this measure is as high as 0.002 seconds.

L et us look at throughput in the absence of flow control for a selected topology to illustrate packet drops. Figure 7.9 shows throughput in terms of received frames, in addition to amount of sent and dropped frames. The number of sent frames grows linear from 0 to slightly above 10000000, corresponding to the increasing data rate. From the first packet drop occurrence, around 60Mbps as pointed out earlier, the actual throughput stabilizes and remains at a fixed value. The amount of dropped packets however continue to increase linear, and parallel to, the amount of total sent frames. The gap between these two climbing lines equals the actual throughput observed.

B efore trying to explain the differences observed in variation, we look to the raw data log files to find minimum, maximum and average values within the data. Motivated by the preceding graphs, we choose 1000Mbps as the value for calculating these values in order to reduce variance.

A verages for throughput are 811621, 541863 and 506578 frames respectively for the three flow control modes. The actual range of each case is cor-

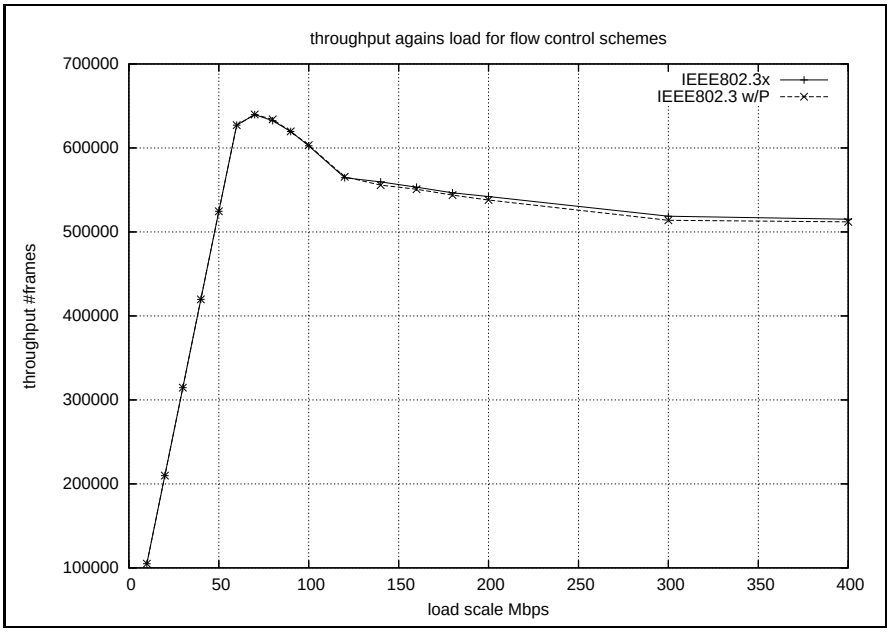

Figure 7.10: Throughput for flow control schemes

respondingly 781278 - 884285, 472092 - 677208 and 467193 - 677550 frames. While the case with flow control disabled has the highest throughput, and smallest variance interval, it is also the only one with packet loss. Protocol **P** has a lower average and min value than IEEE 802.3x but shows a slightly higher max value.

F or latency only min and max values are available due to floating point precision of the recorded values being inexact within our calculating tools. For figure 7.6, the range is approximately 0.00047 to 0.00059. In contrast, the corresponding values for figures 7.7 and 7.8 are 0.00137 to 0.00197, and 0.00137 to 0.00198. In other words, latency behavior is very much the same in the presence of flow control.

L et us now look closer at the lower data rates, 0 to 400 Mbps, for differences between the two flow control protocols. This is illustrated in figures 7.10 and 7.11 with a topology that yielded results very close to the averages. These curves are very typical, and are found for each of the topology sets. Also, even with the finer granularity at the horizontal axis, it is hard to spot the small deviation between the two schemes.

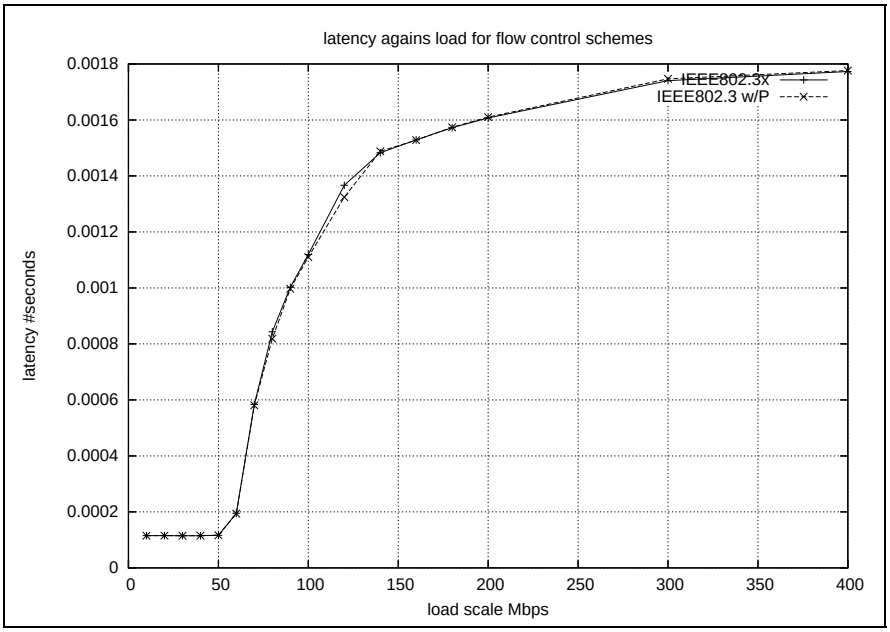

Figure 7.11: Latency for flow control schemes

W ith the above findings as input we now dedicate the reminder of this chapter to in-depth discussion and analysis of our research questions from chapter 1, except the literature survey which was covered in chapter 4.

7.4 Discussion

T he outline follows in general the order of the research questions as they were listed in the introduction. In addition, we include thoughts and observations around sources of errors within our research, as well as lessons learned about memory challenges related to simulation based studies. This content is highly subjective to our interpretation of results and earlier referred to literature. We only include references to other work to identify points where our view deviate from this.

7.4.1 Comparing the flow control scenarios

Q uestion 2: *How does Protocol **P** behave compared to IEEE 802.3x and to the no flow control scenario with respect to performance (throughput and latency), backpressure, packetloss, deadlocks and livelocks in the Gigabit Ethernet context?*

T o our surprise, the two flow control schemes were found to be remarkably similar. Based on the selection of graphs reported, and figures 7.10 and 7.11 in particular, we claim that the measured difference in throughput and latency between IEEE 802.3x and protocol **P** is significantly small enough to be ignored. This is also supported by the minimum, maximum and average values found within the data sets.

A s illustrated in figure 7.9, the absence of flow control results in a highly predictable behavior under congestion. Lower latency and higher throughput is seen in the lossy scheme compared to the flow controlled schemes. However retransmissions of lost data frames are not taken into account. In other words, this scenario only provide a best effort service where extreme packet loss rations are accepted.

N one of the flow control schemes lead to packetloss, consequently neither to deadlocks or livelocks. We have compared the number of sent to the number of received frames in all scenarios, and in all cases we arrived at a true match. Since no frame is dropped, or lost within the network, a deadlock or livelock situation cannot be present.

L ast for this question we address the backpressure property. Our scenarios do not have rate mismatch, since both inter-switch and drop-links operate at 1Gbps, and network interfaces are wirespeed capable. What remains to account for the change in behavior when congestion builds up are link aggregation and buffer contention. In other words a question about resource allocation. As seen in figures 7.1 and 7.2, both throughput and latency stabilizes with little variance at some data rate. Attempting to inject traffic at a higher rate does not produce a significant change in performance.

T he above leads us to the conclusion that the effect of hop-by-hop flow control propagates in a backpressure manner all the way upstream to the source host nodes and throttles them, limiting the traffic accepted into the network to the maximum capacity of the current network configuration.

P ut in other words, in the absence of a higher layer protocol, the hop-by-hop flow control of both IEEE 802.3x and protocol **P** adopts the role of an end-to-end congestion control scheme, manging a long term congestion scenario. We believe that this end-to-end behavior might play a role in the similarities found between the two flow control schemes. Consequently,

in the presence of long term congestion, and identical policies for traffic admission to the network, the observed performance similarities have a plausible explanation.

B ased on the evidence given, we conclude that protocol **P** does not yield any additional performance beyond what standard IEEE 802.3x does. In other words, here **P** only contribute to increased buffer management cost.

7.4.2 Complexity of buffer management

Q uestion 3: *Are there any differences in control message overhead, buffer occupancy and bottleneck link utilization between Protocol **P** and IEEE 802.3x, and if so, what characterize these differences.*

B oth flow control schemes are implemented and run using the same simulation software and components. In other words, they use the same scheduling algorithm, same control frame format, same set of timers, same processing procedures, and same buffer occupancy thresholds for activating flow control. In the light of this information it is reasonable to expect few observed variations, and this is in fact what we found.

W hile protocol **P** might send updated transmit feedbacks in response to changing buffer occupancy, but still above the high threshold, the standard PAUSE scheme would send refresh feedback to extend an ongoing pause period. Alerted by the same trigger and examining the queue using identical thresholds, the outcome is also the same. In other words, differences in control message overhead is tied to implementation choices and not exclusively to the protocol properties. This also hold for the buffer occupancy.

A n interesting observation has been made on bottleneck link utilization. Due to the above mentioned end-to-end congestion scenario that emerges, we cannot target specific links at bottlenecks, since all parts of the network are assumed to be part of that congestion scenario (given the uniform address distribution). However, if we compare the steady state throughput and latency plotted for flow control enabled versus disabled, it is striking that not only do the uncontrolled frames spend less times waiting in buffer queues, more frames are actually being transmitted over the links in the absence of flow control.

T his leads to the conclusion that the flow control schemes impose a less optimal link utilization! This degrading is shown in for example figure fig:tp-compare-fc where the peak performance is at 70Mbps, and is reduced when the offered load increases beyond that point.

O ur implementation has not taken into account the extent of intraswitch delay, e.g. the processing cost of the buffer management scheme in general, and eligibility and level table maintenance in particular. We believe this depends on possible vendor specific implementations, but expect it to contribute significantly to increased latency compared to the standard PAUSE scheme.

T he hop-limit D puts restrictions on switch design leading to need for configurable switches and more functionality within software. This claim originates from the observation that D imperatively must be equal to or greater than the worst-case longest path in the network. Physical buffer is wasted if this value is set too high, since no packets will in that case be assigned with such high packet levels, and consequently, those buffers set aside to resolve buffer deadlocks will never be used. On the other hand, if a too low D is statically set, that switch would have a fixed upper limit to the size of the network it could be deployed in.

I n other words there is a strong reason for vendors to be cautious when implementing this part of the protocol.

W e also note that the buffer reservation scheme with 1 MTU worth of space available at each level above the shared pool will result in wasted space if not max MTU is received. This allocation however also apply to selected threshold margins for the simpler PAUSE scheme

7.4.3 Topology and routing

Q uestion 4: *In what extent do Protocol **P** apply to solving deadlocks in general, or is it limited to handling store-and-forward deadlocks?*

I t is not explicitly clear from the articles describing protocol **P** what kind of deadlock **P** is designed to solve, it can only be inferred from the context of the claims. For example, the phrase *packet networks* brings immediate associations to store-and-forward scenarios. Also, the following quote points

in that direction:"if peak amounts of bandwidth and buffers are available and are dedicated for all network traffic flows, then buffers do not overflow and deadlocks are not created"[43, pg. 924].

O ur explorations of deadlocks in chapter 3, and the properties stated by the creators of protocol **P** , including: "[This] technique is quite general and can be used for networks with various types of routing"[43, pg. 925], leads us to arriving at the conclusion that protocol **P** *is not a routing scheme and cannot solve routing deadlocks through its buffer management scheme*. It is important that all packets between a source-destination pair follow the same path through the network, thus schemes using alternative routing, or misrouting, for example, is not compatible with the protocol.

P rotocol **P** is commonly referred to, and evaluated, in the Ethernet context, using the MAC Control scheme, but it should be noted that this is not the only possible option. However, for this thesis, we cannot break compatibility with the Ethernet standards. Consequently we are stuck with the spanning tree protocol and its routing restrictions. Protocol **P** alone cannot solve the tasks of a routing algorithm, so replacing the STP with up*/down* or TBTP can only be done in a **P** scenario if it can also be done for the IEEE 802.3x scheme.

Q uestion 5: *In what extent is Protocol **P** suited for SAN ?* In the Ethernet context, we claim that protocol **P** is not better suited than the original PAUSE scheme. Hence all drawbacks and problems associated with this scheme reported in the literature, also apply to protocol **P** . This contrasts with what we believed at the onset of this project, but through close studying of protocol **P** it seems to us that its properties have been overestimated and its deadlock prevention misunderstood.

T his said, protocol **P** has during our implementation and study received a United States Patent, number 6859435, a fact which only strengthens our view that this invention is solid and well designed. Our skepticism relates to the areas **P** is implemented and used.

7.4.4 Sources of errors

D uring our implementation and subsequent analysis we have identified some problems related to the behavior of both our link scheduler and usage of the `pause_quanta` / pause field of the MAC Control. First we admit

to a implementation difference regarding the inter frame gap used by the link scheduler of the outgoing interfaces. It has been previously stated that we use an inter frame gap of 96ns or 96 bit times. However, the actual Java code uses a value set to the packet size left-shifted by 3. This is done consistently throughout the simulations, and is expected to affect all scenarios the same. It can however accommodate for some of the observed restrictions to throughput and latency in the steady state.

S econd, we shed light on the fact that we have used a maximum pause value of 255, which is a single byte value, while the IEEE standard defines this to be a 2 octet value with a maximum of 255 0xFFFF. This has severe implications for pause timing, but not for the overall on/off behavior. We are in fact in line with the standard by using a smaller range than allowed. But unfortunately this has lead to a much higher buffer check interval, timer granularity and frequency of sent control frames. With a higher pause duration per control frame, it can as previously described give more space for ordinary data on the link.

7.4.5 Memory challenges for simulations

D uring our work we have been constantly challenged by the memory requirements introduced by J-Sim in the computing environment. Repeatedly we have returned to our simulation output files to find that it has been abnormally terminated due to running out of memory on the computer, even when using the Condor system at campus. Our network is not very large, but there are many hidden components within each node that adds to the total.

A n example calculus of memory consumption has been presented by Hung-ying Tyan in his thesis on J-Sim[87], and he directs focus to all the hidden component costs like wires and ports. In particular, we have the additional cost of Ethernet frames wrapped within InetPackets, the level queue elements and also all the excess traffic generated but dropped before it was permitted to enter the network. We believe that the garbage collection routine of Java is not entirely capable of keeping up with the high memory consumption of our Ethernet and sophisticated buffer management approach. In the light of this we acknowledge that the implementation could have been carried out more optimal if we at the outset possessed the current knowledge.

Chapter 8

Conclusion

8.1 Conclusion

T he implementation and simulation based study of hop-by-hop link layer flow control in general, and protocol **P** in particular, is well summarized through the words of the people who invented the protocol:

> No packets will be dropped inside a packet network, even when congestion builds up, if congested nodes send back/-pressure feedback to neighboring nodes, informing them of unavailability of buffering capacity - stopping them from forwarding more packets until enough buffer becomes available.[43]

W e have in this thesis seen that usage of the terms 'congestion control' and 'flow control' vary both with respect to control point within the network, location within the protocol stack and over time.

O ur attention has been focused at three research areas and their intersection; the IEEE 802 Ethernet technology and standard, with emphasis on MAC issues; network congestion, with emphasis on flow control; and deadlock schemes. Buffer management has been the main focus through the thesis.

W e have developed J-Sim components, including a set of queue related tools to incorporate protocol **P** as well as IEEE802.3x flow control at the link layer of the simulator.

B ased on the evidence given in our findings, we conclude that protocol **P** does not yield any additional performance beyond what standard IEEE 802.3x does. In other words, here **P** only contribute to increased buffer management cost. Further, we have not any differences in control message overhead, buffer occupancy and bottleneck link utilization between Protocol **P** and IEEE 802.3x, and attribute this to the implementation similarities. Third, protocol **P** is not a routing scheme and cannot solve routing deadlocks through its buffer management scheme. It is thus only capable of handling store-and-forward deadlocks. Especially within the Ethernet context, protocol **P** is limited to the spanning tree protocol and associated routing algorithm, and it not better suited than the original PAUSE scheme to solve congestion problems.

8.2 Future Work

T hroughout our work we have arrived at interesting ideas for expanding the current scenario, if time permits and with appropriate resources allocated. First and foremost this includes running simulations with other routing schemes like up*/down* and TBTP, as well as testing adaptive routing. Second, we would like to test our claim of **P** not being able to handle routing deadlocks by running it in deadlock prone topologies, including setups that uses minimal path routing and uses all channel bandwidth. The latter is directed toward further exploring usage in the SAN domain.

I ntroducing priority mechanisms in protocol P enhanced nodes could be interesting, but we are somewhat skeptical due to the amount of existing literature stating that combining flow control and priorities is not a desirable solution.

L ast, we would like to run TCP over the studied link layer protocols, to see how TCP retransmission and RED behaves with the underlying flow control. These scenarios could perhaps resemble those used by Noureddine in his thesis.

Appendix A

Source code: Queues

A.1 BufferBudgetDropTailQueue

```
package drcl.inet.core.queue;

import java.util.Vector;

import drcl.comp.*;
import drcl.comp.lib.*;
import drcl.data.*;
import drcl.inet.InetPacket;
import drcl.inet.core.ni.*;
import drcl.inet.core.queue.*;
import drcl.net.*;
import drcl.util.queue.*;

/**
 * Queue implementation special to switch nodes implementing Protocol
 * P. It is asumed that a switch allways has at least two interfaces.
 *
 * @author Bergfrid Marie Skaara
 * @version 1.0, 06/03/2004
 * @see drcl.inet.core.queue.DropTail
 */

public class BufferBudgetDropTailQueue extends drcl.inet.core.Queue implements BufferBudgetConstants
{
    // repository of all bbc at this node
  protected BufferBudgetCounter [] bbc_repository;

  protected VSFIFOLevelQueue q = null;
  public static final String EVENT_QLEN = "Instant Q Length";
```

```
protected int                    capacity;
protected int                    available;
protected int                    currentFeedbackLevel;

protected Component              parentComponent;
protected LevelTable             levelTable;

protected boolean                linkEmulation;
protected boolean                firstEncounter;

/**
 * Enqueues the object at the end of the queue. Choses the correct
 * BufferBudgetCounter based on information in the packet to be
 * enqueued.  Looks up packet level in the LevelTable with
 * destination as key.  Capacity for this (shared memory) switch
 * depends on buffer occupancy of this bbc.
 *
 * @param obj_ the object to be enqueued
 *
 * @return the object being dropped due to the enqueue; null
 * otherwise.
 */
public synchronized Object enqueue(Object obj_)
{
  if(obj_ == null || !(obj_ instanceof Packet)) return obj_;
  Packet      p                  = (Packet) obj_;
  int         psize              = isByteMode()? p.size: 1;

  long        p_ldestination   = ((InetPacket)p).getDestination();

  Long        p_Ldestination   = new Long(p_ldestination);


  double      pkey               = p_Ldestination.doubleValue();
  if(Double.isNaN(pkey))       return obj_;

  int         plevel             = levelTable.getLevel(p_Ldestination);
  int         qlevel             = plevel;
  int         feedback           ;

  P2PNI_PIn in_ = (P2PNI_PIn) (this.getParent()).getComponent(p.getInInterface());
  if(in_ == null)              return obj_;

  BufferBudgetCounter bbc    = getBufferBudgetCounter(p);
  if(bbc == null)              return obj_;

  /* CASE first time initializations */
  if (firstEncounter)
    populateBbc_repository();
```

```
if (q == null)
   q =   new VSFIFOLevelQueue();

/* CASE determine value of feedback */
if(linkEmulation)
   {
      /* find the highest value of j where mi_j is less than mtu */
      if(bbc.getSharedFree() >= MTU)
         feedback = 0; // default
      else
         {
            feedback= 1;
            while(feedback <= bbc.getMaxHops() && bbc.get_mi(feedback) < MTU) feedback++;
         }
   }
else
   {
      /* use received value stored in bbc */
      feedback       = bbc.getReceivedTF();
   }

/* CASE check packet level */
if(in_.isProtocolPaware())
   {
      if( ((InetPacket)p).getHops() == 2)
         {
            /* Packet arriving from network enter link, no level updates permitted.
             * Differentiate qlevel from plevel if needed
             */
            qlevel = (plevel == 0)? 1:plevel;
         }
      else
         {
            /* Packet arriving from network internal node */
            plevel = (plevel == 0)? 1:plevel;

            if (feedback + 1 > plevel) // Level Assignment Rule
               {
                  // now we kick over possibility of unknown source
                  updateLevel(plevel, feedback + 1, p_ldestination);
                  plevel = feedback + 1;
               }
            else if(qlevel == 0)          // no entry for this destination
               {
                  levelTable.setLevel(p_Ldestination, 1);
               }
            qlevel = plevel;              // plevel has been raised, keep them identical
```

```
                // LevelTable held the highest value, use this both for enqueue and reservation
            }
        }
    else
        { // std values for buffering without protocol P
            plevel = 0;
            qlevel = 1;
        }

    /* CASE no availaboe space, count and drop p */
    if ( psize > bbc.get_mi(qlevel) )
        {
            if (isGarbageEnabled())
                drop(p, "Out of memory: " + psize + ">" + bbc.get_mi(qlevel) );
            drainPort.doSending((InetPacket) p); // count drops
            return obj_;
        }

    /* CASE enqueue */
    if (bbc.reserveBufferBudget(psize, qlevel))
        {
            getAvailable();
            q.enqueue(pkey, p, psize);
            if (isDebugEnabled())
                {
                    EthFrame     frame;
                    long nr;
                    if (((InetPacket) p).getBody() instanceof EthFrame) {
                        frame = (EthFrame) ((InetPacket)p).getBody();
                        nr= frame.getFrameNumber();
                    }
                    else nr = -1;
                }
            return null;
        }

    /* CASE cannot enqueue packet of this level */
    if (isGarbageEnabled())
        drop(p, "Capacity exceeded at level: " + qlevel);
    drainPort.doSending((InetPacket)p);        // count drops

    return obj_;
}

/**
 * Dequeues and returns the first eligible object in the
 * queue. Traverse q and find first eligible packet to dequeue. This
 * is the basis of scheduling. Choses the correct
 * BufferBudgetCounter based on information in the packet
```

```
   * dequeued. Looks up packet level in the LevelTable with
   * destination as key.
   *
   * @return the object dequeued; null if queue is empty.
   */
  public synchronized Object dequeue()
  {
    int       psize   = 0;            // packet size
    int       plevel  = 0;            // packet level
    int       qlevel  = 0;            // level index for queue
    double    pkey;                     // packet key
    Packet p          = null;         // tmp packet
    BufferBudgetCounter pbbc; //bbc this packet is recorded on

    if (q == null || q.isEmpty())
      {
        if(isDebugEnabled()) debug ("NO Q or EMPTY");
        return null;
      }


    /* determine the key */
    pkey =   getEligibleKey(currentFeedbackLevel);

    if(Double.isNaN(pkey)) // no eligible packets
      {
        if(isDebugEnabled()) debug ("NO ELEGIBLE");
        return null;
      }

    /* performes dequeue and frees up Bufferbudget */
    p                    = (Packet) q.dequeue(pkey);
    if (p == null)
      {
        if(isDebugEnabled()) debug ("P NULL");
        return null;
      }

    plevel               = levelTable.getLevel(new Long (((InetPacket)p).getDestination()));
    qlevel               = (plevel == 0) ? 1 : plevel;

    pbbc                 = getBufferBudgetCounter((InetPacket)p);
    if (pbbc == null)
      return null;
    psize                = isByteMode()? p.size: pbbc.getMTU();

    pbbc.freeBufferBudget(psize, qlevel); // update BufferBudgetCounter

    if (qLenPort._isEventExportEnabled())
```

```
        qLenPort.exportEvent(EVENT_QLEN, new DoubleObj(q.getSize()), null);

    getAvailable();

    return p;
}


/**
 * Updates the BudgetBufferCounter for the given destination and
 * level. If ByteMode is used, packet size is not fixed.
 *
 * @param oldlevel      the previous level
 *          newlevel      the level to use for this destination
 *          destination the destination to raise level for
 */
protected synchronized void updateLevel(int oldlevel, int newlevel, long destination)
{
    int i = -1;
    BufferBudgetCounter bbc;

    Object o_ []    = q.retrieveAllBy(oldlevel); // objects of desired level
    if(o_ == null) return;
     while(++i < o_.length)
        {
          InetPacket p = (InetPacket) o_[i];
          if (p == null) return;
          if(p.getDestination() == destination)
             {
                bbc          = getBufferBudgetCounter(p);
                int size = isByteMode()? p.size: 1;
                bbc.liftLevel(oldlevel, newlevel, size);
             }
        }
     levelTable.setLevel(new Long (destination), newlevel);
}

/**
 * Returns the first key matching an eligible packet level
 *
 * @param level_
 *
 * @return eligible key
 */
protected double getEligibleKey(int level_)
{
    double keys [] = q.keys();
    int i = -1;
    while (++i < keys.length)
```

```
        {
          // Transmit Eligibility Rule
          if((levelTable.getLevel(new Long ((long)keys[i])))  >=  level_)
            return keys[i];
        }
      return Double.NaN;
    }

    /**
     * tests if a packet is eligible
     *
     * @param packet
     *
     * @return true if verified
     */
    public boolean verifyEligibility(InetPacket p)
    {
      if (levelTable.getLevel(new Long (p.getDestination())) >= currentFeedbackLevel)
          return true;
      return false;
    }

    /**
     * Calculates the amount of available space in the queue, defined as
     * sum available buffer budget in all BufferBudgetCounter at this
     * node, asuming switch is shared memory with max defined by the
     * combination of buffer in all interfaces. Calculates by traversing
     * the bbc_repository.
     *
     * @return available_
     */
    public int getAvailable()
    {
      int available_ = 0 ;
      int i = -1;

      while((++i < bbc_repository.length) && bbc_repository[i] != null)
        {
          available_ += bbc_repository[i].getAvailable();
        }
      available = available_;
      return available_;
    }
}
```

A.2 BufferBudgetCounter

```
package drcl.comp.lib;
import drcl.comp.*;
import drcl.comp.lib.*;
import java.io.*;

/**
 * A counter for Buffer Budget per NI
 *
 * @author Bergfrid Marie Skaara
 * @version 1.0, 20040527
*/
public class BufferBudgetCounter extends Component implements BufferBudgetConstants
{
   int lastReceivedTF;        // transmitt feedback
   int lastSentTF;            // transmitt feedback

   int budgetTotal;           // total buffer for this NI
   int mtu;                      // maximum packet size for the link determined by the NI
   int maxHops;                     // maximum number of valid hops in the network
   int dDedicated;            // dedicated buffer per destination–level > 1
   int shared_b1;             // shared memory, buffer budget level 1
   int inUse;                  // total memory in use for virtual receiving queue

   /* Tables for buffer budget values, all 1–>dHops size, slot 0 unused */
   int [] size_bi;            // size memory at level i
   int [] threshold_Bi;       // upper threshold   for level i
   int [] used_ni;            // combined packet sizes of level i
   int [] combinedFree_mi; // combined buffer including lvl i that is not taken by packets of lvl <

   protected int fcType = FC_TYPE_IEEE; // used to determine threshold marks

   /**
    * Constructor for setting local variables
    *
    * @param maxHops_        // hoplimit for network
    *           budgetTotal_ // capacity
    *           dDedicated_  // bytes dedicated buffer for each level > 1
    */
   public void initBufferBudgetCounter(int maxHops_, int budgetTotal_, int dDedicated_)
   {
      maxHops             = maxHops_;
      lastReceivedTF   = 0;
      lastSentTF         = 0;

      size_bi             = new int [maxHops+1];
      threshold_Bi       = new int [maxHops+1];
      used_ni             = new int [maxHops+1];
```

```
    combinedFree_mi = new int [maxHops+1];

    budgetTotal      = budgetTotal_;
    mtu              = MTU;
    dDedicated       = dDedicated_;
    shared_b1        = budgetTotal - (dDedicated_*(maxHops_-1));
    inUse            = 0;

    setDebugEnabled(false);
    setGarbageEnabled(false);
  }

  /**
   * Initiates the buffer budget values in the tables
   */
  protected void initTables()
  {
    int i = 1;
    size_bi          [i] = shared_b1;
    threshold_Bi     [i] = shared_b1;
    used_ni          [i] = 0 ;
    combinedFree_mi[i] = shared_b1;

    while(++i <= maxHops)
      {
        size_bi          [i] = dDedicated ;
        threshold_Bi     [i] = threshold_Bi [i-1] + dDedicated;
        used_ni          [i] = 0;
        combinedFree_mi[i] = combinedFree_mi[i-1] + dDedicated;
      }
  }

  /**
   * Reports the total amount of free memory in the virtual input
   * queue
   *
   * @param data_     that arrived
   *          inPort_ data arrived at
   */
  public synchronized void process(Object data_, drcl.comp.Port inPort_)
  {
    if (0 == combinedFree_mi[maxHops])
      inPort_.doLastSending("Out of Memory");
    else {
      inPort_.doLastSending("Available " + combinedFree_mi[maxHops]);
    }
  }

  /**
```

```
  * Reserves one unit of buffer budget at the specified
  * packetlevel. Updates budget variables oposit to freeBufferBudget.
  *
  * @param psize_  amount to reserve
  *           plevel_ to reserve at
  *
  * @return true if reservation is ok, false if out of memory on that
  * level
  */
public synchronized boolean reserveBufferBudget(int psize_, int plevel_)
{
   /* cannot reserve if BufferBudget is exhausted */
   if(isFull())
      {
         if(isGarbageEnabled())
            debug("Out of memory");
         return false;
      }

   int psize = (psize_ == 1)? mtu: psize_; //if packetmode use std size mtu
   int plevel = plevel_;

   /* Check if BufferBudget is available for a packet of psize_ of
    * the given level, if available reserve by updating used_ni,
    * inUse and combinedFree_mi.
    */
   if(combinedFree_mi[plevel_] >= psize)
      {
         used_ni[plevel_] += psize;
         inUse += psize;

         while(plevel <= maxHops)
            {
               combinedFree_mi[plevel++] -= psize;
            }
         return true;
      }
   return false;
}

/**
  * Frees one unit of buffer budget at the specified packet
  * level. Updates budget variables oposit to reserveBufferBudget.
  *
  * @param psize_  amount to reserve
  *           plevel_ to reserve at
  *
  * @return true if free is ok, false if free cannot be performed.
  */
```

```
public synchronized boolean freeBufferBudget(int psize_, int plevel_)
{
   int psize  = (psize_ == 1)? mtu: psize_; //if packetmode use std size mtu
   int plevel = plevel_;

   /* Make sure there really is minimum psize_ reserved before
    * proceding
    */
   if(inUse >= psize_ && used_ni[plevel_] >= psize_)
     {
        used_ni[plevel_] -= psize;
        inUse -= psize;

        while(plevel <= maxHops)
          {
             combinedFree_mi[plevel++] += psize_;
          }
        return true;
     }
   return false;
}

/**
 * Lifts the level for one packet from oldLevel to newLevel. Calls
 * liftLevel forwarding parameters and adding count=1.
 *
 * @param    oldLevel_
 *           newLevel_
 *           psize_
 *
 * @return
 */
public synchronized boolean liftLevel(int oldLevel_, int newLevel_, int psize_)
{
   int psize = (psize_ == 1)? mtu: psize_; //if packetmode use std size mtu
   return liftLevel(oldLevel_, newLevel_, psize, 1);
}

/**
 * Lifts the level for count number of packets with common
 * destination packet from oldLevel to newLevel. Free BufferBudget
 * at oldLevel, and reserve at newLevel. Note that the inUse remains
 * constant! combinedFree_mi is lifted at the levels from old and to
 * below new.
 *
 *<p> only valid for count > 1 if network has constant psize!
 *
 * @param    oldLevel_
 *           newLevel_
```

```
     *              psize_
     *              count_
     *
     * @return
     */
    public synchronized boolean liftLevel(int oldLevel_, int newLevel_, int psize_, int count_)
    {
      int oldLevel = oldLevel_;
      int newLevel = (newLevel_ > maxHops) ? maxHops : newLevel_;
      int psize      = (psize_ == 1)? mtu: psize_; //if packetmode use std size mtu

      if(newLevel <= oldLevel)
        {
          if(isDebugEnabled())
            debug("Cannot reduse level!");  //REMOVE
          return false;
        }

      used_ni[oldLevel] -= psize*count_; //free BufferBudget at oldLevel
      used_ni[newLevel] += psize*count_; //reserve BufferBudget at newLevel

      /* Adjust combinedFree_mi for the levels between old and new, as
       * the packet(s) don't take up space here any more
       */
      while(oldLevel < newLevel)
        {
          combinedFree_mi[oldLevel++] += psize*count_;
        }
      return true;
    }

    /**
     * Returns true if there is no space left in shared level 1, meaning
     * flow control has to be activated
     */
    public boolean isAboveHighMark(boolean linkEmulation_)
    {
      return isAbove_Threshold(linkEmulation_);
    }

    /**
     * Returns true if there is space left in shared level 1,
     * meaning flow control can be deactivated
     */
    public boolean isBelowLowMark(boolean linkEmulation_)
    {
      int margin = linkEmulation_? MARGIN:0;
      if(fcType == 1)
        margin +=3;
```

```
    if(shared_b1 >= inUse + margin*mtu)
      return true;
    return false;
  }

  /**
   * Returns true if there is space left in shared level 1,
   * meaning flow control can be deactivated
   */
  public boolean isBelowLowMark_Arriving(boolean linkEmulation_, boolean arriving)
  {
    int margin = linkEmulation_? MARGIN:0;
    if(arriving)
      margin +=1; // add 1 for margin to the arriving frame
    if(fcType == 1)
      margin +=3;
    if(fcType == 2)
      margin +=3; // test for overflow ttl=2 in protocol p
    if(shared_b1 >= inUse + margin*mtu)
      return true;
    return false;
  }

  /**
   * Returns true if there is no space left in shared level 1, meaning
   * flow control has to be activated
   */
  public boolean isAbove_Threshold(boolean linkEmulation_)
  {
    int margin = linkEmulation_? MARGIN:0;
    if(shared_b1 <= (inUse + margin*mtu))
      return true;
    return false;
  }

   /**
   * Returns true if there is no space left in shared level 1, meaning
   * flow control has to be activated
   */
  public boolean isAbove_Threshold_Arriving(boolean linkEmulation_, boolean arriving_)
  {
    int margin = linkEmulation_? MARGIN:0;
    if(arriving_) margin +=1; // add 1 for margin to the arriving frame

    if(shared_b1 <= (inUse + margin*mtu))
      return true;
    return false;
  }
```

```
  /** Returns true if there is no room for another mtu size packet */
  public boolean isFull()
  {
    if(budgetTotal  < inUse + mtu)
      return true;
    return false;
  }

  protected int sum_ni(int plevel_)
  {
    int plevel = plevel_;
    int sum = 0;
    while(plevel > 0)
      {
        sum += used_ni[plevel--];
      }
    return sum;
  }

  protected int calc_mi(int plevel_)
  {
    return (threshold_Bi[plevel_] - sum_ni(plevel_));
  }

  public int get_mi(int level_)
  {
    if(level_ == 0) return 0;
    return calc_mi(level_);
  }
}
```

A.3 LevelTable

```
package drcl.comp.lib;

import drcl.comp.*;
import java.util.Hashtable;

/**
 * Component that records the level assigned to a network destination
 * foreach destination currently having packets inside the
 * switch. Protocol P assumes destination based routing, hence all
 * packets intended for the same host are bound to leave the switch
 * via the same port. The (destination,level) pairs are hence global
 * to the switch, and can be updated and checked from all NI
 * implementing Protocol P.
 *
```

```
 * @author  Bergfrid Marie Skaara
 * @version 1.0, 20040527
*/

public class LevelTable extends Component
{
  protected Hashtable destinationLevelPair; // the level table for (destination,level) pairs
  public int dHops; // the value D according to Protcol P

  /**
   * Resets level value for a given destination in the
   * destinationLevelPair HashTable to 0.
   *
   * @param destination_ to reset
   */
  public synchronized void clearLevel(Long destination_)
  {
    if(destination_ == null) return;
    if (destinationLevelPair.containsKey(destination_))
      {
        destinationLevelPair.remove(destination_);
      }
  }

  /**
   * Overwrites the previous level value for a given destination in
   * the destinationLevelPair HashTable. Only one entry per
   * destination, so remove old entry if it excists and insert new
   * with the updated value
   *
   * @param destination_
   *            level_
   */
  public synchronized void setLevel(Long destination_, int level_)
  {
    if(destination_ == null) return;
    if(level_ == 0)
      {
        clearLevel(destination_);
        return;
      }

    if (destinationLevelPair.containsKey(destination_))
      {
        destinationLevelPair.remove(destination_);
      }

    destinationLevelPair.put(destination_, new Integer(level_));
  }
```

```
   /**
    * Gets the level assosiated with the given destination, if no entry
    * is found, default value is 0.
    *
    * @param destination_
    *
    * @return level, 0 as default
    */
   public synchronized int getLevel(Long destination_)
   {
      if(destination_ == null) return 0;
      if (destinationLevelPair.containsKey(destination_))
         {
            return ((Integer)(destinationLevelPair.get(destination_))).intValue();
         }
      else
         return 0;
   }

}
```

A.4 LevelQElement

```
/**
 *
 * @author Bergfrid Marie Skaara
 * @version 1.0, 06/14/2004
 * @see drcl.util.queue._Element
 */

package drcl.inet.core.queue;

public class LevelQElement extends drcl.DrclObj implements drcl.util.queue.Element
{
   double          key; // corresponding to level
   int             size;
   Object          obj;
   LevelQElement   next;

   /**
    *Preferred constructor
    */
   LevelQElement (double key_, Object o_, int size_)
   {
      key  = key_;
      size = size_;
```

```
      obj  = o_;
      next = null;
   }

   void recycle()
   {
      obj  = null;
      next = null;
   }

   public Object getObject()
   {
      return obj;
   }

   public int getSize()
   {
      return size;
   }

   public double getKey()
   {
      return key;
   }
}
```

A.5 VSFIFOLevelQueue

```
package drcl.inet.core.queue;

import java.util.*;
import drcl.util.queue.*;
import drcl.comp.lib.*;

/**
 * Variable-size version of {@link FIFOQueue} with special LevelQ addition.
 *
 * @author Bergfrid Marie Skaara
 * @version 1.0, 06/14/2004
 * @see drcl.util.queue.VSFIFOQueue
 */

public class VSFIFOLevelQueue implements BufferBudgetConstants
{
   LevelQElement head;
   LevelQElement tail;
```

```
int             size;
int             length;

/**
 * Enqueues the element in FIFO manner with level = 1 (default
 * endnode) and std MTU. Forwards request to enqueue(double, Object,
 * int)
 *
 * @param element_ to be enqueued
 */
public void enqueue(Object element_)
{
   enqueue((double)1, element_, MTU);
}

/**
 * Enqueues the element in FIFO manner with level = 1 (default
 * endnode) and specified size. Forwards request to enqueue(double,
 * Object, int)
 *
 * @param element_ to be enqueued
 *          size_      of the element
 */
public void enqueue(Object element_, int size_)
{
   enqueue((double)1, element_, size_);
}

/**
 * Enqueues the element in FIFO manner with specified key and std
 * MTU. Forwards request to enqueue(double, Object, int)
 *
 * @param element_ to be enqueued
 *          size_      of the element
 */
public void enqueue(double key_, Object element_)
{
     enqueue(key_, element_, MTU);
}

/**
 * Enqueues the element in FIFO manner with the associated key and
 * specified size. Key is connected to destination level, not the
 * std q interpretation.
 *
 * @param key_       of the element
 *          element_ to be enqueued
 *          size_      of the element
 */
```

```
  public void enqueue(double key_, Object element_, int size_)
  {
    LevelQElement e_   = head;
    LevelQElement new_ = new LevelQElement(key_, element_, size_);

    if (tail == null)
      head.next = new_;
    else
      tail.next = new_;

    tail      = new_;
    size      += size_;
    length    ++;
  }


  /**
   * Dequeues the first element
   *
   * @return dequeued object
   */
  public Object dequeue()
  {
    if (head.next == null)
      return null;

    LevelQElement e_ = head.next;
    Object        o_ = e_.obj;
    head.next        = e_.next;
    size             -= e_.size;

    if (--length == 0)
      tail = null;
    e_.recycle();

    return o_;
  }

  /**
   * Dequeues the first element matching the specified key
   *
   * @param key_
   *
   * @return dequeued object
   */
  public Object dequeue(double key_)
  {
    for (LevelQElement e_ = head; e_.next != null; e_ = e_.next)
      {
```

```
            if (e_.next.key == key_)
               {
                  LevelQElement out_ = e_.next;
                  Object o_          = out_.obj;
                  e_.next            = out_.next;
                  size               -= out_.size;

                  if (--length == 0)
                     tail = null;
                  else if (e_.next == null)
                     tail = e_;

                  out_.recycle();
                  return o_;
               }
         }
      return null;
   }

   /**
    * Removes the first object matching element and key from the queue
    *
    * @param   key_       of the element to remove
    *             element_ tp remove
    *
    * @return object removed
    */
   public Object remove(double key_, Object element_)
   {
      if (head == null)
         return null;

      for (LevelQElement e_ = head; e_.next != null; e_ = e_.next)
         {
            LevelQElement tmp_ = e_.next;
            Object o_          = tmp_.obj;
            if (key_ == tmp_.key && (o_ == element_ || o_ != null && o_.equals(element_)))
               {
                  e_.next          = tmp_.next;
                  size             -= ((LevelQElement)o_).getSize();

                  if (--length == 0)
                     tail = null;
                  else if (e_.next == null)
                     tail = e_;

                  tmp_.recycle();
                  return o_;
               }
```

```
        }
    return null;
}

/**
 * Removes the first object matching element from the queue
 *
 * @param   element_
 *
 * @return object removed
 */
public Object remove(Object element_)
{
    for (LevelQElement e_ = head; e_ != null && e_.next != null; e_ = e_.next)
        {
            LevelQElement tmp_ = e_.next;
            Object o_          = tmp_.obj;

            if (o_ == element_ || o_ != null && o_.equals(element_))
                {
                    e_.next     = tmp_.next;
                    size       -= tmp_.size;

                    if (--length == 0)
                        tail = null;
                    else if (e_.next == null)
                        tail = e_;

                    tmp_.recycle();
                    return o_;
                }
        }
    return null;
}

/**
 * Removes all objects matching element from the queue
 *
 * @param element_
 */
public void removeAll(Object element_)
{
    for (LevelQElement e_ = head; e_ != null && e_.next != null; e_ = e_.next)
        {
        LevelQElement tmp_ = e_.next;
        Object o_          = tmp_.obj;
        if (o_ == element_ || o_ != null && o_.equals(element_))
            {
            e_.next        = tmp_.next;
```

```
            size                -= tmp_.size;

            if (--length == 0)
               tail = null;
            else if (e_.next == null)
               tail = e_;
            tmp_.recycle();
         }
      }
   }

   /**
    * Removes all objects matching element and key from the queue
    *
    * @param key_
    *            element_
    */
   public void removeAll(double key_, Object element_)
   {
      for (LevelQElement e_ = head; e_ != null && e_.next != null; e_ = e_.next)
         {
         LevelQElement tmp_ = e_.next;
         Object o_          = tmp_.obj;
         if (tmp_.key == key_ &&      (o_ == element_ || o_ != null && o_.equals(element_))
            {
            e_.next         = tmp_.next;
            size            -= tmp_.size;

            if (--length == 0)
               tail = null;
            else if (e_.next == null)
               tail = e_;
            tmp_.recycle();
         }
      }
   }

   /**
    * Removes the nth element of the queue
    *
    * @param n_
    *
    * @return object removed
    */
   public Object remove(int n_)
   {
      LevelQElement e_= head;
      for (int i=0; i<n_ && e_.next != null; i++, e_ = e_.next); // traverse list
```

```
    if (e_.next == null)
      return null;

    LevelQElement tmp_ = e_.next;
    Object o_          = tmp_.obj;
    e_.next            = tmp_.next;
    size               -= tmp_.size;

    if (--length == 0)
      tail = null;
    else if (e_.next == null)
      tail = e_;
    tmp_.recycle();

    return o_;
  }

  /**
   * Returns the first element of the queue
   *
   * @return object
   */
  public Object firstElement()
  { return head.next == null? null: head.next.obj; }

  /**
   * Returns the last element of the queue
   *
   * @return object
   */
  public Object lastElement()
  { return tail == null? null: tail.obj; }

  /**
   * Returns the first key of the queue
   *
   * @return key
   */
  public double firstKey()
  { return head.next == null? Double.NaN: head.next.key; }

  /**
   * Returns the first key found at or above level
   *
   * @param level_
   *
   * @return eligible key
   */
  public double getEligibleKey(int level_)
```

```
{
   for (LevelQElement e_ = head; e_.next != null; e_ = e_.next)
      {
         if (e_.next.key >= level_)
            return e_.next.key;
      }
   return Double.NaN;
}
        /**
 * Retieves the object at the given pos without performing dequeue
 *
 * @param n_
 *
 * @return object
 */
public Object retrieveAt(int n_)
{
   LevelQElement e_= head.next;
   for (int i=0; i<n_ && e_ != null; i++, e_ = e_.next); // traverse list

   return e_ == null? null: e_.obj;
}

/**
 * Retieves the key at the given pos without performing dequeue
 *
 * @param n_
 *
 * @return key
 */
public double retrieveKeyAt(int n_)
{
   LevelQElement e_= head.next;
   for (int i=0; i<n_ && e_ != null; i++, e_ = e_.next); // traverse litst

   return e_ == null? Double.NaN: e_.key;
}

/**
 * Retrieves the first object matching the key
 *
 * @param key_
 *
 * @return object
 */
public Object retrieveBy(double key_)
{
   for (LevelQElement e_ = head; e_.next != null; e_ = e_.next)
      if (e_.next.key == key_)
```

```
            return e_.next.obj;
        return null;
    }

    /**
     * Retrieves the all objects matching the key
     *
     * @param key_
     *
     * @return object []
     */
    public Object [] retrieveAllBy(double key_)
    {
        Object[] o_ = new Object[length];
        int      i  = 0;

        for (LevelQElement e_ = head; e_.next != null; e_ = e_.next)
            {
                if (e_.next.key == key_)
                    o_[i++] = e_.obj;
            }
        return (Object [])o_;
    }

    /**
     * Returns true if the list contains that element
     *
     * @param element_
     *
     * @return true if the list contains that element
     */
    public boolean contains(Object element_)
    {
        for (LevelQElement e_ = head.next; e_ != null; e_ = e_.next)
            if (e_.obj.equals(element_)) return true;
        return false;
    }

    /**
     * Returns true if the list contains that key
     *
     * @param key_
     *
     * @return true if the list contains that element
     */
    public boolean containsKey(double key_)
    {
        for (LevelQElement e_ = head; e_.next != null; e_ = e_.next)
            if (e_.next.key == key_)
```

```
            return true;
        return false;
    }

    public double[] keys()
    {
        double[] keys_ = new double[length];
        LevelQElement e_ = head.next;
        for (int i=0; i<keys_.length; i++) {
            keys_[i] = e_.key;
            e_ = e_.next;
        }
        return keys_;
    }

    /**
     * Returns the length of the list.
     *
     * @return length
     */
    public int getLength()
    {          return length;          }

    /**
     * Returns the length of the list.
     *
     * @return size
     */
    public int getSize()
    {          return size;            }

    public boolean isEmpty()
    {         return (length == 0)? true: false; }


}
```

Appendix B

Tcl scripts and functions

B.1 Template script for main scenario

```
#a=rate, b=fctype, c=topology, d=logTp, e=logLat

source "simulationSettings.tcl"
source "simulationTools.tcl"
source "simulationRoutingTools.tcl"

# Create root components
cd [mkdir drcl.comp.Component $simRoot]
set link [java::new drcl.inet.Link]
$link setPropDelay $linkPropagation

#Settings

set minSendRate $a
set maxSendRate $minSendRate

#0-1-2
set fcType $b
if {$fcType == 0} {
    set flowControlStatus disableFlowControl
    set protocolPStatus disableProtocolP
}
if {$fcType == 1} {
    set flowControlStatus enableFlowControl
    set protocolPStatus disableProtocolP
}
if {$fcType == 2} {
    set flowControlStatus enableFlowControl
    set protocolPStatus enableProtocolP
}
```

```
#fName topology
set currentTopologyLASH $c
puts $currentTopologyLASH

set debugStatus   false
set garbageStatus false

proc setInitNI { path bandwidth propagation} {
    global protocolPStatus
    global flowControlStatus
    global debugStatus
    global garbageStatus

    [! $path] setBandwidth      $bandwidth
    [! $path] setPropDelay      $propagation
    [! $path] setDebugEnabled   $debugStatus
    [! $path] setGarbageEnabled $garbageStatus
    [! $path] $flowControlStatus
    [! $path] $protocolPStatus
}

# build topology
puts "Building topology"
buildSpanningTreeTopology

set hostBuilder [mkdir drcl.net.PNodeBuilder .hostBuilder]
set switchBuilder [mkdir drcl.net.PNodeBuilder .switchBuilder]
set cslBuilder [mkdir drcl.net.ProtocolPBuilder .cslBuilder]

# build hosts
puts -nonewline "Building hosts"
for {set i $nSwitches} {$i < [expr $nHosts + $nSwitches]} {incr i} {
    $hostBuilder build [! h$i] $cslBuilder

    cd $simRoot/h$i/

    # add source
    set src_model [java::new drcl.net.traffic.traffic_PeakRate
                    $minPacketSize $maxPacketSize $minSendRate $maxSendRate]
    mkdir [java::call drcl.net.traffic.TrafficAssistant
           getTrafficComponent $src_model] source

    # address modifyer
    setAm "$simRoot/h$i/" "$i" $nSwitches $nHosts

    # stats collector: packets sent and received
    set stats [mkdir drcl.net.PStatCollector .stats]
    $stats setStats $nHosts $nSwitches
```

```
    connect -c csl/0@down           -to $stats/send@
    connect -c csl/pd/100@up        -to $stats/received@

    setInitNI "$simRoot/h$i/csl/nip_out"
              $hostIFBandwidth $linkPropagation
    setInitNI "$simRoot/h$i/csl/nip_in"
              $hostIFBandwidth $linkPropagation

    cd ..
    puts -nonewline "."
}
puts ""

# build switches
puts -nonewline "Building switches"
for {set i 0} {$i < $nSwitches} {incr i} {
    $switchBuilder build [! n$i] $cslBuilder

    for {set j 0} {$j < $switchNumPorts} {incr j} {
        [! $simRoot/n$i/csl/bbc$j] setFcType $fcType
        setInitNI "$simRoot/n$i/csl/nip_out$j"
                  $switchIFBandwidth $linkPropagation
        setInitNI "$simRoot/n$i/csl/nip_in$j"
                  $switchIFBandwidth $linkPropagation
    }

    cd $simRoot/n$i

    # stats collector: packets dropped, packets received at PD
    set stats [mkdir drcl.net.PStatCollector .stats]
    $stats setStats $nHosts $nSwitches
    connect -c csl/drain@       -to $stats/dropped@
    connect -c csl/pd/100@up    -to $stats/received@

    cd ..

    puts -nonewline "."
}

puts ""
puts "Done building"

# setup routes
puts "Setting up routes"
setSpanningTreeRoutes $nHosts $nSwitches

puts "Starting simulation"

set fNameThroughput $d
```

```
set fNameLatency    $e

cd $simRoot
set sim [attach_simulator .]
run .

script -at $SIM_RUNTIME "! /eth/h*/source stop"                    -on $sim
script -at $SIM_STOPTIME "dumpThroughput
                $nSwitches $nHosts $fNameThroughput $minSendRate" -on $sim
script -at $SIM_STOPTIME "dumpLatency
                $nSwitches $nHosts $fNameLatency    $minSendRate" -on $sim

$sim stopAt $SIM_STOPTIME
```

B.2 Utility Tcl scripts

```
proc setSpanningTreeRoutes { nHosts nSwitches } {
    global simRoot

    puts -nonewline "% Setting up spanning tree routes "
    for {set i $nSwitches} {$i < [expr $nHosts + $nSwitches]} {incr i} {
        puts -nonewline "$i."
        for {set j $nSwitches} {$j < [expr $nHosts + $nSwitches]} {incr j} {
            if {$i == $j} {
                continue
            }
            java::call drcl.inet.InetUtil setupRoutes
                        [! $simRoot/h$i] [! $simRoot/h$j] "bidirection"
        }
    }
}

proc buildSpanningTreeTopology {} {
    global simRoot
    global currentTopologyLASH
    global link
    global nSwitches
    global nHosts
    global nHostsPerSwitch

    cd $simRoot
    set iu [java::new drcl.inet.InetUtil]
    set adjMatrix_ [$iu getAdjMatrixFromFile $currentTopologyLASH]
    set k [java::new drcl.comp.lib.MSTKruskal]
    $k readAdjMatrix $adjMatrix_
    $k performKruskal
    set adjMatrixMST [$k createAdjMatrixFromMST
```

```
                    $nSwitches $nHosts $nHostsPerSwitch]
    java::call drcl.inet.InetUtil createTopology [! .]
                                  $adjMatrixMST $link
}


proc setAm {hostpath hostID nSwitches nHosts} {
    cd $hostpath
    mkdir drcl.net.AddressModifier am
    ! am setNumSwitches $nSwitches
    ! am setNumHosts $nHosts
    ! am setAddressDistribution 0
    # 1 for static destination, 0 for uniform
    ! am setAddress [! . getDefaultAddress]
    connect -c source/down@ -to am/up@
    connect -c am/down@ -to csl/100@up
}

# host stats
proc dumpThroughput {nSwitches nHosts fName rate} {
    set pktsent         0
    set pktreceiveddata 0
    set pktdropped      0

    if {$fName != "stdout"} {
        if [catch {open $fName a} fID] {
            puts stderr "Cannot open $fName: $fID"
        } else {
            set fName $fID
        }
    }

    puts $fName "($rate) "
    for {set i $nSwitches} {$i < [expr $nHosts + $nSwitches]} {incr i} {
        # send and received from hosts
        set tmpsent         [! /eth/h$i/.stats getTotalSent]
        set tmpreceiveddata [! /eth/h$i/.stats getTotalDataReceived]

        set pktsent         [expr $pktsent + $tmpsent]
        set pktreceiveddata [expr $pktreceiveddata + $tmpreceiveddata]

        # dropped from switches
        if {[expr $i - $nSwitches] < $nSwitches} {
            set tmpdropped  [! /eth/n[expr $i - $nSwitches]/.stats getTotalDropped]
            set pktdropped  [expr $pktdropped + $tmpdropped]
        } else {
            set tmpdropped 0
        }
        puts $fName   "$tmpsent $tmpreceiveddata $tmpdropped"
    }
```

```
    puts $fName   "TOTAL: "
    puts $fName   "$pktsent $pktreceiveddata $pktdropped"
}

# dump latency for all packets
proc dumpLatency {nSwitches nHosts fName rate} {

    set sumLat         0
    set sumPkt         0
    set avg            0

    if {$fName != "stdout"} {
        if [catch {open $fName a} fID] {
            puts stderr "Cannot open $fName: $fID"
        } else {
            set fName $fID
        }
    }
    puts $fName "($rate)"
    for {set i $nSwitches} {$i < [expr $nHosts + $nSwitches]} {incr i} {
        set tmpLatency [! /eth/h$i/.stats getAccNetworkLatencyArray]

        set lat [java::call java.lang.reflect.Array getDouble $tmpLatency 0]
        set nPackets [java::call java.lang.reflect.Array getDouble $tmpLatency 1

        set sumLat [expr $sumLat + $lat]
        set sumPkt [expr $sumPkt + $nPackets]

        puts $fName "$lat $nPackets"
    }
    set avg [expr $sumLat / $sumPkt]
    puts $fName "$sumLat $sumPkt $avg"
}
```

Bibliography

[1] gnuplot homepage. http://www.gnuplot.info/, March 2007.

[2] Eitan Altman and Tania Jimènez. Ns simulator for beginners. http://www-sop.inria.fr/maestro/personnel/Eitan.Altman/COURS-NS/n3.pdf, Dec. 2003.

[3] I. T. Assosiation. Infiniband architechture specification. Technical report, I. T. Assosiation, 2000.

[4] James Aweya, Michel Ouellette, and Delfin Y. Montuno. Interworking of switched ethernet and atm flow control mechanisms. *Int. J. Netw. Manag.*, 12(6):357–366, 2002. issn = 1099-1190.

[5] A. Bermúdez, R. Casado, F. J. Alfaro, F. J. Quiles, J. L. Sánchez, and J. Duato. On the performance of up*/down* routing. In *The Fourth Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing* [33], pages 61–72.

[6] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, second edition, 1992. fist edition 1987.

[7] 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.

[8] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *SIGMETRICS*, pages 80–90, 1996.

[9] W. Bux, W.E. Denzel, T. Engbersen, A. Herkersdorf, and R.P. Luijten. Technologies and building blocks for fast packet forwarding. *IEEE Communication Magazine*, january 2001.

[10] J. S. Carson. Modeling and simulation worldviews. In *Winter Simulation Conference (WSC '93)*, pages 18–23, New York, dec 1993. ACM Association for Computing Machinery. ISBN = 0-7803-1381-X.

[11] H. Chen and P. Wyckoff. High performance commodity interconnects for clustered scientific and engineering computing. Ldrd 98-0260 final report, Sandia National Laboratories, aug 2000.

[12] David R. Cheriton. Sirpent: A high-performance internetworking approach. In *SIGCOMM*, pages 158–169, 1989.

[13] I. Cidon, J. M. Jaffe, and M. Sidi. Distributed store-and-forward deadlock detection and resolution algorithms. *IEEE trans. on commun.*, COM-35, 11:1139–1145, 1987.

[14] Rene L. Cruz. A calculus for network delay. i. network elements in isolation. *Information Theory, IEEE Transactions on*, 37(1):114 – 13, jan 1991.

[15] Rene L. Cruz. A calculus for network delay. ii. network analysis. *Information Theory, IEEE Transactions on*, 37(1):132 – 141, jan 1991.

[16] A.V.S. Cui-Qing Yang Reddy. A taxonomy for congestion control algorithms in packet switching networks. In *Network, IEEE*, volume 9, pages 34 – 45. Dept. of Comput. Sci., North Texas Univ., Denton, TX, USA, jul 1995.

[17] William James Dally and Brian Towels. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers - Elsevier, 500 Sansome Street, Suite 400, San Francisco, CA 94111, 2004. ISBN: 0-12-200751-4.

[18] Duato. A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. *IEEETPDS: IEEE Transactions on Parallel and Distributed Systems*, 7, 1996.

[19] L. Breslau et al. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[20] O. Feuser and A. Wenzel. On the effects of the IEEE 802.3x flow control in full-duplex Ethernet LANs. In IEEE, editor, *LCN'99: proceedings: 24th Conference on Local Computer Networks*, pages 160–163, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, oct 1999. IEEE Computer Society Press. ISBN 0-7695-0309-8.

[21] George S. Fishman. *Discrete-event simulation : modeling, programming, and analysis*. Springer series in operations research. Springer, New York, 2001. ISBN: 0-387-95160-1, ib.

[22] Sally Floyd and Van Jacobson. Random early detec tion gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[23] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. Tcp performance re-visited. In *ISPASS-2003 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, 2003.

[24] Jean-Philippe Georges, Eric Rondeau, and Thierry Divoux. Evaluation of switched ethernet in an industrial context by using the network calculus. In *4th IEEE International Workshop on Factory Communication Systems* [32], pages 19–26.

[25] Mario Gerla and Leonard Kleinrock. Flow control: A comparative survey. *IEEE Transactions on Communications*, COM-28(4):553–574, april 1980.

[26] A. Giessler, J. Hanle, A. Konig, and E. Dade. Free buffer allocation - an investigation by simulation. *Computer Networks*, 2:191–208, 1978.

[27] S. Jamaloddin Golestani. Congestion-free communication in high-speed packet networks. *IEEE Transactions on Communications*, 39(12):1802–1812, december 1991.

[28] I. S. Gopal. Prevention of store-and-forward deadlock in computer networks. *IEEE trans. on commun.*, C-33, 12:1258–1264, 1985.

[29] Inder S. Gopal. Prevention of store-and forward deadlock in computer networks. In Isaac D. Scherson and Abdou S. Youssef, editors, *Interconnection Networks for High-Performance Parallel Computers*, pages 338–344, Los Alamitos-Washington-Brussels-Tokyo, 1994. IEEE Computer Society Press. Originally in: IEEE Trans. Communications, Vol. COM-33, No. 12, Dec. 1985 pp. 1258-1264.

[30] K. D. Günther. Prevention of deadlocks in packet-switched data transport systems. *IEEE trans. on commun.*, COM-29:512–524, 1981.

[31] Justin Hurwitz and Wu chen Feng. Initial end-to-end performance evaluation of 10-gigabit ethernet. In *Proceedings of IEEE Hot Interconnects: 11th Symposium on High-Performance Interconnects*. IEEE, IEEE, aug 2003.

[32] IEEE. *4th IEEE International Workshop on Factory Communication Systems*, aug 2002.

[33] IEEE Computer Society. *The Fourth Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, jan 2000.

[34] Driftsseksjonen Inf. Condor på institutt for informatikk, uio. http://www.ifi.uio.no/condor/.

[35] S. Yalamanchili J. Duato and L. Ni. *Interconnection Networks an Engineering Approach.* IEEE Computer Society, 1997. Revised Printing.

[36] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, aug 1988.

[37] R. Jain. Congestion control in computer networks: issues and trends. In *IEEE Network Magazine*, volume 4-3, pages 24 – 30. Digital Equipment Corp., Littleton, MA, USA, may 1990. ISSN: 0890-8044.

[38] R. Jain. Myths about congestion management in high-speed networks. *Internetworking: Research and Experience*, 3:101–113, 1992. (Technical Report-726, Digital Equipment Corporation, October 1990).

[39] Raj Jain and K. K. Ramakrishnan. Congestion avoidance in computer networks with a connectionless network layer: Concepts, goals and methodology. In *Proc. IEEE Computer Networking Symposium, Washington D.C.*, pages 134–143, april 1988.

[40] Jurgen Jaspernite, Peter Neumann, Michael Theis, and Kym Watson. Deterministic real-time communication with switched ethernet. In *4th IEEE International Workshop on Factory Communication Systems* [32], pages 11–18.

[41] R. Jing-Fei Ren, Landry. Flow control and congestion avoidance in switched ethernet lans. In IEEE, editor, *ICC 97*, volume 1, pages 508–512, 1997.

[42] Mark Karol, S. Jamaloddin Golestani, and David Lee. Prevention of deadlocks and livelocks in lossless, backpressured packet networks. In *Proceedings of the 2000 IEEE Computer and Communications Societies Conference on Computer Communications (INFOCOM-00)*, pages 1333–1342, Los Alamitos, mar 2000. IEEE.

[43] Mark Karol, S. Jamaloddin Golestani, and David Lee. Prevention of deadlocks and livelocks in lossless backpressured packet networks. In *IEEE/ACM Trans. Netw.*, volume 11-6, pages 923–934. ACM Press, 2003. issn=1063-6692.

[44] Mark Karol, David Lee, and S. Jamaloddin Golestani. A simple technique that prevents packet loss and deadlocks in gigabit ethernet. In *Proc. 1999 International Symposium on Communications (ISCOM'99)*, pages 26–30, november 1999.

[45] Simon S. Lam and Martin Reiser. Congestion control of store-and-forward networks by input buffer limits - an analysis. *IEEE Transactions on Communications*, COM-29(1):127–134, january 1979.

[46] Duke Lee, Sinem Coleri, Xuanming Dong, and Mustafa Ergen. Florax - flow-rate based hop by hop back-pressure control for ieee 802.3x. citeseer.ist.psu.edu/535134.html.

[47] P. M. Merlin and P. J. Schweitzer. Deadlock avoidance in store-and-forward networks - I: store-and-forward deadlock. *IEEE trans. on commun.*, COM-28:345–354, 1980.

[48] R. M. Metcalfe and D. R. Boggs. ETHERNET: distributed packet switching for local area networks. *Computer networks / ACM*, 19(5):395–404, 1976.

[49] Robert M. Metcalfe. *Packet Communication*. Phd thesis, Harward University, december 1973.

[50] Robert M. Metcalfe, David R. Boggs, Charles P. Thacker, and Butler W. Lampson. Multipoint data communication system with collision detection, united states patent 4063220, december 1977. www.freepatentsonline.com/4063220.html.

[51] Partho P. Mishra and Hermant Kanakia. A hop-by-hop rate-based congestion control. In *SIGCOMM '92: Conference procedings on Communications architectures & protocols*, pages 112–123, Ner York, NY, USA, 1992. ACM Press. ISBN: 0-89791-525-9.

[52] W. Noureddine and F. Tobagi. Selective backpressure in switched ethernet lans. In *Proceedings of IEEE GLOBECOM*, pages 1256–1263, 1999.

[53] Wael Noureddine. *Improving the Performance of TCP Applications Using Network Assisted Mechanisms*. PhD thesis, Stanford University, june 2002.

[54] Wael Noureddine and Fouad Tobagi. The transmission control protocol, an introcuction to tcp and research survey. Technical report, Stanford University, july 2002.

[55] G. Omidyar, C.G. Pujolle. Guest editorial - introduction to flow and congestion control. In *Communications Magazine, IEEE*, volume 34, page 30, nov 1996. ISSN: 0163-6804.

[56] C. M. Ozveren, R. Simcoe, and G. Varghese. Reliable and efficient hop-by-hop flow control. In *Selected Areas in Communications, IEEE Journal on*, volume 13, pages 642 – 650. Digital Equipment Corp., Littleton, MA, USA;, may 1995. ISSN: 0733-8716.

[57] Carlos M. D. Pazos, Juan C. Sanchez-Agrelo, and Mario Gerla. Using back-pressure to improve TCP performance with many flows. In *INFOCOM (2)*, pages 431–438, 1999.

[58] Francesco De Pellegrini, David Starobinski, Mark G. Karpovsky, and Lev B. Levitin. Scalable cycle-breaking algorithms for gigabit ethernet backbones. In *INFOCOM*, 2004.

[59] Larry L. Peterson and Bruce S. Davie. *Computer Networks, A Systems Approach*. Morgan Kaufman Publishers, 2000.

[60] Louis Pouzin. Methods, tools, and observations on flow control in packet-switched data networks. *IEEE Transactions on Communications*, COM-29(4):413–426, april 1981.

[61] W. Prue and J. Postel. Rfc 1016 - something a host could do with source quench:. Technical report, Network Working Group, july 1987.

[62] E. Raubold and J. Haenle. A method of deadlock-free resource allocation and flow control in packet networks. In *Proceedings of the Third International Conference on Computer Communication*, 1976.

[63] Renato John Recio. Server i/o networks past, present, and future. In *ACM SIGCOMM 2003 Workshops*, pages 163–178, aug 2003.

[64] José Carlos Sancho, Antonio Robles, and José Duato. A new methodology to compute deadlock-free routing tables for irregular networks. In *The Fourth Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing* [33], pages 45–60.

[65] T. J. Schriber and D. T. Brunner. Inside simulation software: how it works and why it matters. In *Winter Simulation Conference (WSC '96)*, pages 23–30, New York, 1996. ACM Association for Computing Machinery.

[66] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterhwaite, and C. P Thacker. Autonet: A high-speed, self-configuring local area network using pointto-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, oct 1991.

[67] Rich Seifert. *The Switch Book : The Complete Guide to LAN Switching Technology*. John Wiley & Sons, Inc., 605 Third Avenue, NY, 2000.

[68] Tor Skeie, Svein Johannessen, and Øyvind Holmeide. The road to an end-to-end deterministic ethernet. In *4th IEEE International Workshop on Factory Communication Systems* [32], pages 3–9. ABB Corporate Research.

[69] IEEE Computer Society. Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks - specofoc requirements part 3: Carrier sence multiole access with collision detection(csma/cd) acces method an physical layer specifications. Technical report, IEEE Standards Institution, march 2002. IEEE Std 802.3-2002.

[70] JTC 1 ISO Standards. Iso/ice 7498-1:1994 information technology – open systems interconnection – basic reference model: The basic model, 1994. www.iso.org.

[71] W. Stevens. Rfc 2001 - tcp slow start, congestion avoidance, fast retransmit. Technical report, Network Working Group, january 1997.

[72] Andrew S. Tanenbaum. *Computer Networks (Fourth Edition)*. Prentice Hall PTR (Pearson Education Inc.), Upper Saddle River, New Jersey 07458, 2003.

[73] the Conbdor Project. Condor high throughput computing. http://www.cs.wisc.edu/condor/.

[74] I. Theiss. Evaluering av metodar for svitsja sci. Master's thesis, Department of Informatics, University of Oslo, Nowrway, feb 1999.

[75] VINT. The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[76] VINT. Virtual internetwork testbed. a collaboration among ucs/isi, xerox parc, lbnl and ubc. http://www.isi.edu/nsnam/vint/index.html.

[77] J. Wechta, A. Eberlein, and F. Halsall. The impact of topology and choice of tcp window size on the performance of switched lans. *Computer Communications*, 22:955–965, 1999.

[78] J. Wechta, A. Eberlein, F. Halsall, and M. Spratt. Simulation-based analysis of the interaction of end-to-end and hop-by-hop flow control schemes in packet switching lans. In *Proceedings of the Fifteenth UK Teletraffic Symposium on Performance Engineering in Information Systems*, 1998.

[79] J. Wechta, Armin Eberlein, and F. Halsall. The interaction of the TCP flow control procedure in end nodes on the proposed flow control mechanism for use in IEEE 802.3 switches. In *HPN*, pages 515–534, 1998.

[80] J. Wechta, Armin Eberlein, and F. Halsall. An investigation into the performance of switched lans. In *in Proceedings of the Conference on Network and Optical Communications*, Manchester, UK, 1998.

[81] J. Wechta, M. Fricker, and F. Halsall. Hop-by-hop flow control as a method to improve qos in 802.3 lans. *Proccedings of the Seventh International Workshop on Quality of Service*, pages 239–247, may 1999.

[82] X. Xiao and L. M. Ni. Internet qos: A big picture. *IEEE Network*, 13(2):8–18, mar 1999.

[83] A. Koubaa Loria Y. Song and F. Simont. Switched ethernet for real-time industrial communication: Modelling and message buffering delay evaluation. In *4th IEEE International Workshop on Factory Communication Systems* [32], pages 27–35.

[84] Hung ying Tyan.

[85] Hung ying Tyan.

[86] Hung ying Tyan. J-sim. http://www.j-sim.org.

[87] Hung ying Tyan. *Desig, realization and evaluation of a component-based compositional software architechture for network simulation*. Ph.d., The Ohio State University, 2002.

[88] K. Yoshigoe and K. Christensen. Rate control for bandwidth allocated services in ieee 802.3 ethernet. citeseer.ist.psu.edu/471682.html.