



SÄHKÖTEKNIIKAN OSASTO
TIETOTEKNIIKAN KOULUTUSOHJELMA

HARDWARE-ASSISTED NETWORKING USING SCHEDULED TRANSFER PROTOCOL ON LINUX

Työn tekijä _____
Pekka Pietikäinen

Työn valvoja _____
Tino Pyssysalo

Hyväksytty _____ / _____ 2001

Arvosana _____

Pietikäinen P. (2001) Hardware-assisted Networking Using Scheduled Transfer Protocol on Linux. Department of Electrical Engineering, University of Oulu, Oulu, Finland. Diploma thesis, 78 p.

ABSTRACT

As network speeds have increased, the demands placed on protocol implementations have grown significantly. Gigabit Ethernet has made network speeds of 100 MB/s possible on commodity hardware, although achieving the full potential is difficult. The two greatest problems seen on Gigabit Ethernet networks are the large number of hardware interrupts and unnecessary memory copies.

This thesis studies the ways in which implementations of network protocols can be optimized. An emphasis is placed on methods which offload some of the protocol processing onto programmable network adapters.

Scheduled Transfer Protocol (STP) is a new ANSI specified data transfer protocol, which uses small control messages to pre-allocate buffers at the data destination before the data movement begins. This characteristic of the protocol reduces the state information required for protocol processing significantly, and thus makes hardware accelerated implementations of the protocol that copy data directly between the user buffer and network interface considerably simpler than traditional protocols such as TCP.

Experimental results of STP performance are obtained by studying the STP implementation for Linux 2.4 on Gigabit Ethernet and comparing the results to TCP.

The results obtained in this thesis show that partially offloading protocol processing to an intelligent network interface reduces the CPU utilization considerably and makes it possible to utilize the full capabilities of future network technologies.

Keywords: zero-copy networking, interrupt mitigation, network protocol implementation, Gigabit Ethernet, OS bypass.

Pietikäinen P. (2001) Laitteistoavustettu Scheduled Transfer -protokollan toteutus Linuxissa. Oulun yliopisto, sähkötekniikan osasto. Diplomityö, 78 s.

TIIVISTELMÄ

Verkkonopeuksien kasvaessa protokollatoteutuksiin kohdistetut vaatimukset ovat kasvaneet huomattavasti. Gigabit Ethernet on tehnyt 100 MB/s siirtonopeuksista mahdollisia myös yleisesti käytössä olevissa verkoissa, joskin täyden suorituskyvyn saavuttaminen on haastavaa. Suurimmat ongelmat nykyisissä Gigabit Ethernet-verkoissa ovat laitteistokeskeytyksien suuri määrä ja tarpeettomat muistikopiointit.

Tässä työssä tutkitaan menetelmiä, joilla verkkoprotokollien toteutuksia voidaan parantaa. Erytystä huomiota kiinnitetään tapoihin, joissa osa protokollakäsittelystä on siirretty ohjelmoitavalle verkkokortille.

Scheduled Transfer Protocol (STP) on uusi ANSI-standardoitu tietoliikenneprotokolla, joka käyttää pieniä ohjausviestejä puskureiden varaamiseen tiedon vastaanottajassa ennen tiedonsiirron aloittamista. Tämä ominaisuus vähentää protokollakäsittelyn vaatimaa tilainformaatiota huomattavasti ja tekee laitteistoavustetuista protokollaimplementaatioista, jotka kopioivat tietoa suoraan sovelluksen muistialueen ja verkkokortin välillä, erittäin yksinkertaisia verrattuna tavanomaisiin protokolleihin kuten TCP:hen.

Scheduled Transfer Protocolin suorituskykyä tutkitaan Gigabit Ethernet -verkoissa käyttäen Linux 2.4:lle tehtyä toteutusta ja vertaamalla sen suorituskykyä TCP:hen.

Työssä saadut tulokset osoittavat, että siirtämällä protokollakäsittely osittain verkkokortille on mahdollista vähentää huomattavasti verkkoliikenteen aiheuttamaa kuormaa päätelaitteille. Tämä mahdollistaa tulevaisuuden verkkotekniikoiden täyden suorituskyvyn käytön.

Avainsanat: kopioton protokollatoteutus, keskeytysten vähentäminen, Gigabit Ethernet, käyttöjärjestelmän ohitus.

TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

ABBREVIATIONS

1. INTRODUCTION	9
1.1. Background	9
1.2. Approach and goals of this thesis	11
2. EVOLUTION OF HIGH-SPEED TCP/IP NETWORKING	13
2.1. Introduction to UNIX networking stacks	13
2.1.1. Performance related aspects of the network interface layer . .	15
2.2. Key areas for performance	15
2.2.1. Interrupt load	15
2.2.2. Protocol processing overhead	16
2.2.3. Increased frame sizes	17
2.2.4. Hardware assisted IP fragmentation and TCP segmentation . .	18
2.2.5. Checksum calculation	18
2.2.6. Zero-copy networking	19
2.3. Summary	23
3. TECHNOLOGIES FOR HIGH-SPEED NETWORKING	24
3.1. OS bypass	24
3.2. Network interconnects	25
3.2.1. Ethernet	25
3.2.2. Gigabyte System Network	26
3.2.3. Infiniband	26
3.2.4. Proprietary System Area Networks	27
3.2.5. Summary of network technologies	27
3.3. Protocols for high-speed networking	27
3.3.1. Scheduled Transfer Protocol	28
3.3.2. Arsenic	29
3.3.3. Direct Winsock and WSDLite	30
3.3.4. TCP RDMA	30
3.3.5. U-Net	31
3.3.6. Virtual Interface Architecture	31
3.4. Analysis	33
4. SCHEDULED TRANSFER PROTOCOL	35
4.1. Concepts of STP	35

4.2.	Connection setup	37
4.3.	Data transfer operations in STP	37
4.3.1.	Non-persistent memory data transfers	37
4.3.2.	Persistent memory operations	39
4.4.	Upper Layer Protocols	40
4.4.1.	Sockets	40
4.4.2.	libst	41
4.4.3.	SCSI over STP	42
4.5.	Summary	42
5.	STP IMPLEMENTATION IN LINUX	44
5.1.	STP implementation in Linux	44
5.2.	Hardware acceleration	44
5.2.1.	Alteon Tigon II	45
5.2.2.	Modifications to the driver and firmware	45
5.3.	Experimental results	47
5.3.1.	Benchmark methods	47
5.3.2.	Effect of STU size	48
5.3.3.	Effect of Transfer size	49
5.3.4.	Comparison between STP and TCP	50
5.3.5.	Interrupt rate	53
5.3.6.	Effect of latency	53
5.3.7.	Effect of lost packets	54
5.4.	Summary	55
6.	DISCUSSION	57
6.1.	Benefits	57
6.1.1.	Hardware acceleration	57
6.1.2.	Zero-copy transmit and receive	58
6.1.3.	Reducing the number of interrupts	58
6.1.4.	OS bypass	58
6.2.	Drawbacks and weaknesses	59
6.2.1.	Requirement of firmware modifications	59
6.2.2.	Relying on hardware	60
6.2.3.	Performance on non-reliable networks	60
6.3.	Results obtained in this thesis	61
6.4.	Future directions	61
6.4.1.	STP	61
6.4.2.	Linux STP implementation	62
6.4.3.	Improvements to TCP	62
6.4.4.	Hardware accelerated TCP/IP	63
6.4.5.	Future network technologies	63
7.	CONCLUSION	64
8.	REFERENCES	65
9.	APPENDICES	70

FOREWORD

This work was done between June 2000 and August 2001 at the European Particle Physics Research Center (CERN) in Geneva, Switzerland. During this period, I worked as a Technical Student in the Physics Data Processing group of the Information Technology division.

I would like to thank Dr. Ben Segal, the supervisor of this thesis, and CERN for providing me the possibility of working in an extremely interesting and challenging environment. Thanks are also due to Professors Tino Pyssysalo and Juha Rönning for reviewing this work. I would also like to thank Brian Tierney for proofreading this thesis and Mika Korhonen for providing me the L^AT_EX style used for typesetting this thesis.

Finally, I would like to thank my family for supporting me throughout my studies.

Oulu, Finland 20.11.2001

Pekka Pietikäinen

ABBREVIATIONS

ANSI	American National Standards Institute
API	Application Programming Interface
BSD	Berkeley Software Distribution
Bufx	Buffer Index
CIFS	Common Internet File System
COW	Copy On Write
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checksum
CTS	Clear To Send
DMA	Direct Memory Access
GigE	Gigabit Ethernet
GSN	Gigabyte System Network
HIPPI	High-Performance Parallel Interface
HTTP	Hypertext Transfer Protocol
kB	kilobyte (1024 bytes)
IETF	Internet Engineering Task Force
I/O	Input/Output
IP	Internet Protocol
iSCSI	Internet SCSI
LAN	Local Area Network
LLP	Lower-Layer Protocol
MAN	Medium Area Network
Mbps	Megabits per second
MB/s	Megabytes per second ¹
MMU	Memory Management Unit
MPI	Message Passing Interface
MSS	Maximum Segment Size
MTU	Maximum Transfer Unit
Mx	Memory Index
NCITS	National Committee for Information Technology Standardization
NFS	Network File System
OSI	Open Systems Interconnection
NIC	Network Interface Card
RFC	Request For Comment
OS	Operating System
PCI	Peripheral Component Interconnect
pps	packets per second
QoS	Quality of Service
RDMA	Remote Direct Memory Access
RID	RDMA Identifier
RPC	Remote Procedure Call
RTS	Request To Send
RX	Receive

¹The experimental results of this thesis use 1 MB = 2^{20} = 1024*1024 bytes.

SAN	System/Storage Area Network
SCI	Scalable Coherent Interface
SCSI	Small Computer System Interface
SST	SCSI on Scheduled Transfer
STP ²	Scheduled Transfer Protocol
STU	Scheduled Transfer Unit
stvd	Scheduled transfer virtual device
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
TOS	Type Of Service
TX	Transmitter
UDP	User Datagram Protocol
ULP	Upper-Layer Protocol
VI	Virtual Interface
VIA	Virtual Interface Architecture
WAN	Wide Area Network
XID	Transaction Id

²The abbreviation ST is also commonly used e.g. in the ANSI standard, but the usage is discouraged since the same abbreviation is also used by IETF to refer to the Internet Stream Protocol defined in RFC 1819.

1. INTRODUCTION

1.1. Background

Within the last few decades, networking has evolved from dumb terminals connected to mainframes via 110 bps serial ports to packet-switched networks covering the entire world at speeds of hundreds of megabytes per second. Consequently the requirements for both the network protocols used and the way they are implemented in operating systems have grown significantly.

During the same time period, Moore's law has predicted the exponential growth of semiconductor speeds, and thus the speed at which data can be ultimately generated and processed. However, network speeds have increased at a rate which in recent years has even surpassed Moore's law. Moreover, the performance of memory and I/O systems have not been able to keep up with CPU or network speeds, which have made them the dominating factors in overall network performance [1, 2 pp. 519-521]

Another factor in network performance is operating system engineering. Although it is tempting to assume that advances in hardware will render any operating system limitations irrelevant, this is not the case. In fact, optimizations such as minimizing the memory accesses done by the network code are the only way to bridge the increasing gap between between network and memory speeds [3, 4].

Improvements to the operating system try to address limitations posed by the network protocols used as well as the capabilities of the Network Interface Cards (NIC). Protocols are usually designed to be able to utilize the network as efficiently as possible, but little attention is paid to how efficiently the host system resources can be used. When the hosts are significantly faster than the network, this is indeed the right approach. However, when the network is nearly as fast as the endpoints, as is the case on gigabit networks and currently available PCs, the endpoints are the limiting factor, not the network.

The traditional way of describing network protocols is to divide the protocols into multiple layers. The OSI Reference Model shown in Figure 1 is the most commonly used model for describing network architectures. In the OSI model, network protocols are split into seven layers with each layer communicating only with the layers below and above it [5].

While the OSI model is a good way of designing and describing network protocols, it is not a very good way of implementing them. An implementor would most likely prefer the model depicted in Figure 2, where data is transferred reliably between applications through an ideal network with infinite bandwidth and zero latency. In practice, neither the OSI model nor the ideal model are feasible.

In a layered approach, the problem of transferring data is divided into multiple smaller problems. This causes problems in the implementations, since the individual layers have no knowledge about the overall picture and thus may make bad decisions about how to buffer data. This may cause severely degraded and unpredictable performance [6, 7, 8].

The alternative approach is practical only when the applications are running on the same machine or when using specialized cluster interconnect hardware, as then the requirement of an "infinitely" fast reliable network can be fulfilled. For almost any other kind of networking some abstraction layers are required to to hide the details and de-

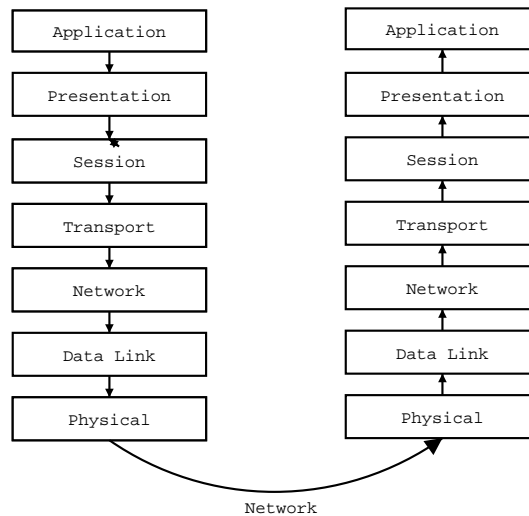


Figure 1. The OSI Reference Model.

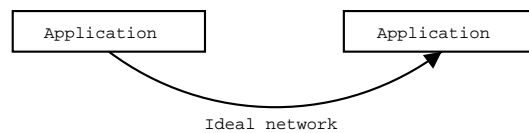


Figure 2. The ideal network model.

iciencies of the underlying networks and provide a network and protocol-independent interface for applications.

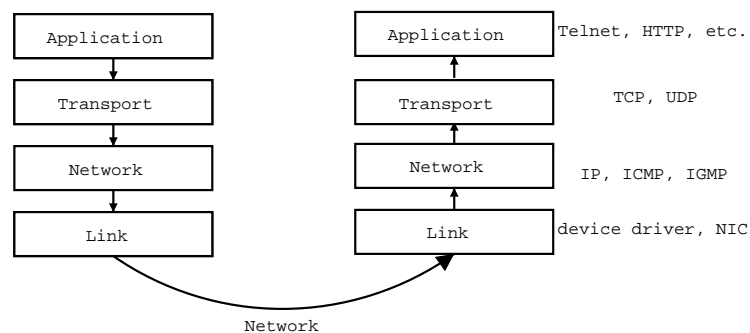


Figure 3. The Internet Reference Model.

A decade before the OSI model, the ARPANET project [9] started developing a protocol that would work in all types of networks. The end result was the TCP/IP protocol suite [10, 11], which are the most common protocols used. As can be seen from Figure 3, the TCP/IP protocol suite is much simpler than the OSI model.

Although TCP/IP is relatively light-weight, this alone is not enough to obtain good performance, as was evidenced by the disappointing performance of TCP/IP implementations in the late 1980's. Commonly used operating systems like SunOS achieved speeds around 400kB/s on 10Mbps Ethernet. This problem was usually attributed to either limitations in the hardware used or deficiencies in TCP/IP itself. Replacements for

TCP/IP such as XTP [12] were proposed, which were designed to be implementable in hardware for maximum performance.

In 1988, Van Jacobson proved that the prior speculation was groundless by making several modifications to the BSD Net-2 network code, and reaching wire-speed on the then-current hardware (Sun 3/60 workstations with 10Mbps Ethernet). His work proved that the performance problems were not caused by problems in the design of TCP/IP, but were instead problems in the implementation of the protocols [13].

Among other problems, previous implementations used a strict layering model which caused bad buffering decisions and unnecessary memory traffic. Another important optimization he discovered was that protocol processing could be accelerated significantly by predicting the next arriving packet based on packets that have been previously received. Even though TCP needs to be able to handle many kinds of different circumstances, e.g. connection setup and packet loss, in most cases packets arrive in order without errors in the data they carry, and it is this code path that should be optimized, even at the cost of making recovery from error conditions more expensive [15].

During the last decade networking has changed considerably. Gigabit network speeds can now be obtained with commodity hardware, as the peripheral buses and memory subsystems of standard PCs have reached an adequate level. Several extensions, such as RFC 1323 (TCP Extensions for High Performance) [16] and improvements in the implementation of network stacks have made it possible for TCP/IP to keep up with the increasing speeds reasonably well. In theory, the only things limiting TCP performance are the speed of light, which according to current knowledge cannot be increased, and the maximum window size of 1 GB, which is the largest value possible when the RFC 1323 TCP Window Scaling option is used [17 pp. 339-357].

Despite the success of TCP/IP, it is not the best possible solution for every application. One such application is clustering, where a large amount of smaller computers are combined to solve a single problem. Depending on the problem type, the machines involved may have to share a very large amount of data, and if the problem cannot be divided into small independent pieces the nodes must communicate with each other very frequently with extremely low latency.

The reason TCP is not necessarily suited well for these applications is due to its basic design, which is to work with all kinds of networks sharing the limited resources available in an efficient and fair manner. At the same time, this makes it very difficult to implement TCP in a manner that utilizes the underlying hardware as efficiently as possible.

Modern switched network technologies such as Gigabit Ethernet, ATM and GSN guarantee (with some limitations) a fast error-free path between the two endpoints. For these environments a new protocol called Scheduled Transfer Protocol (STP) [18] has been developed which is designed for applications where high bandwidth or low latency is required while using only small amounts of CPU time.

1.2. Approach and goals of this thesis

This thesis focuses on studying the ways in which network performance can be optimized. This consists of protocol design issues and how they affect implementations.

Emphasis has been placed on studying ways in which protocol processing can be offloaded to an intelligent NIC.

The approach of this thesis is to study Scheduled Transfer Protocol (STP), which is a recently standardized protocol for high-speed local area networks. The basic design principle of STP is to make hardware accelerated implementation of the protocol relatively simple and thus overcome the two greatest problems seen in high-speed networking today, memory copies and interrupt load.

The performance of STP was studied on Gigabit Ethernet networks using the Linux 2.4 implementation of the protocol from SGI Inc. During the writing of this thesis, the author was the de facto maintainer of the code and contributed several improvements to it.

Although STP has been used as the basis of this thesis, nearly all of the methods applying to STP are also applicable to other protocols, including TCP, although usually with some limitations. The goal of this thesis is evaluate the effectiveness of these methods, and the limitations involved with them.

To obtain a good quantitative comparison of the different solutions available, a detailed performance analysis comparing new technologies other than STP and zero-copy TCP would have been required. For practical reasons this was not possible, as many of the solutions presented are available for a specific operating system or network interface technology. For the same reason, direct comparisons between the benchmark results presented in this thesis and other publications have been avoided, as hardware differences would make any comparisons meaningless.

Due to the rapid developments in this area, some of the references used in this thesis are unpublished “work in progress” draft documents. In the case of IETF Internet-Drafts, the documents contain the following disclaimer:

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The use of such references has been noted in both the text and the references section.

2. EVOLUTION OF HIGH-SPEED TCP/IP NETWORKING

In this chapter, we will describe TCP/IP networking at the implementation level. This chapter is structured as follows. We will first provide an introduction to how network stacks are typically implemented. The description is based on the reference implementation of TCP/IP from the Computer Systems Research Group at the University of California at Berkeley, also known as 4.4BSD-Lite and Net/3. The code has been the starting point for many other implementations, for both UNIX and non-UNIX operating systems and is very well documented. After that, we will look at the key areas affecting TCP/IP performance and how they have been addressed in recent implementations.

2.1. Introduction to UNIX networking stacks

Many requirements are placed on modern network stacks. An important basic requirement is protecting applications from each other, so that unprivileged applications can neither send arbitrary packets to the network nor intercept data intended for another application. Applications should also be able to use the network through a uniform interface, preferably in a way similar to the way ordinary files are used. For these reasons, implementing the protocol inside the operating system is the most natural choice.

In this section, we will provide a brief introduction on the architecture of the networking code in the 4.4BSD-Lite UNIX kernel, which fulfills these conditions and has been used as the basis for several other implementations. Many details have been omitted, as the intention is only to describe the areas which are critical for performance.

The networking code in the 4.4BSD-Lite kernel is organized into three layers, as shown in Figure 4: the socket layer, the protocol layer and the interface layer.

The socket layer is used to provide a uniform interface for applications. Several types of semantics are offered depending on the type of protocol being used. `SOCK_STREAM` provides sequenced, reliable, two-way, connection-oriented byte streams for protocols like TCP, while `SOCK_DGRAM` is used for unreliable, connectionless protocols, such as UDP [17].

When an application wishes to use network services, it creates a socket, which is the kernel object associated with the connection end point. After the socket has been created and (in the case of connection-oriented protocols) connected to the remote host, the application can simply write data from its buffers to the file descriptor of the socket using `write()` or `sendmsg()` system calls.

It should be noted, that the buffer to be used by the network code is allocated and specified by the application. The application sees the buffer simply as a contiguous area of memory, any part of which can be freely used for data transmission. From a hardware point of view, the buffer consists of multiple physical memory pages which are either scattered around the physical memory of the machine or swapped out onto disk. As we will show later, this distinction will prove to be important for hardware-assisted network protocols.

When a `write()` system call is performed on a socket, the socket layer calls a protocol-specific transmit function, which copies the data into special data structures used by the network code to store individual packets, which in BSD are called *mbufs*.

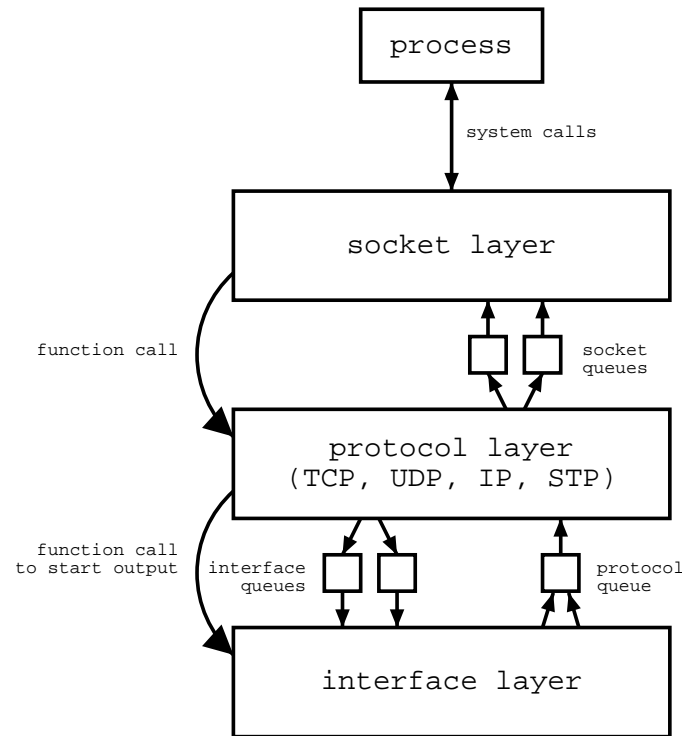


Figure 4. Structure of BSD network stack.

In addition to the user data, mbufs contain a variety of other miscellaneous data: source and destination addresses, socket options, and so on.

The protocol layer is responsible for placing an appropriate protocol-specific header before the user data and placing the packet onto the transmit queue of the correct interface. The device is also told that it should start transmitting the packets in its transmit queue. When the packet has been transmitted, the NIC generates an interrupt, which tells the driver resources required for transmitting the packet can be freed.

When the network interface card (NIC) receives a packet, a hardware interrupt is generated and the NIC interrupt handler is called. The device driver checks the type of interrupt by examining the NIC registers. If the interrupt was caused by a normal receive-complete condition, the data is copied from the device into data structures in the kernel. Then a generic network-medium dependent receive routine is called which enters the packet into a protocol-specific queue. Finally a software interrupt is raised, which causes the queue to be processed. The device's interrupt handling is then complete.

In the case of IP packets, the input routine goes through all the packets in the queue, checks the validity of the checksum, handles possible IP options, and then checks if the packet was destined for this machine or, if the machine was configured as a router, should it be forwarded instead. Finally, the IP stack determines the socket for which the packet was destined for and places the payload on the receive buffer of the socket. The application can then read the data from the buffer [19 pp. 5-23].

2.1.1. Performance related aspects of the network interface layer

The network interface layer is responsible for providing a hardware-independent interface for transferring mbufs to the physical network. There are several aspects of network hardware and driver design that affect performance. For the purposes of this thesis, the critical factor is how data can be transferred from host memory onto the NIC and what resources are consumed by the transfer.

Communication between the OS and the NIC can be accomplished in several ways:

- programmed I/O where special I/O instructions are used to transfer data one byte at a time to the NIC,
- memory-mapped I/O where communication is done through a memory window on the NIC that is mapped to host memory, and
- Direct Memory Access (DMA), where the NIC independently copies data directly from host memory [2].

The model used by nearly all new NICs is an extension of DMA called *scatter-gather DMA*. In scatter-gather DMA the device driver passes a list describing fragments of which the packet is composed of. For example, the packets may consist of two fragments, protocol headers generated by the kernel followed by data from the application buffers. The advantage of this method is that it is unnecessary to copy the packet fragments into a single physically contiguous region of memory before the packet can be transferred.

2.2. Key areas for performance

Network performance is affected by all layers of the system, from the user application to the network hardware. The processing overhead associated with networking can roughly be divided into two categories, *per-packet* and *per-byte* costs.

Per-packet overheads are fixed costs involved in processing each packet of data independent of its size. The most important sources of per-packet costs are *interrupt load* and *protocol processing overheads*. A generic way of reducing their effect is by using larger packets, although physical limitations of most network technologies often make this impossible.

Per-byte costs depend on the amount of data being processed. They are associated with copying data, or performing some operation on the entire data block. Several techniques are available to minimize their effect by reducing the need for handling the actual data block when it is transferred. The most important techniques for accomplishing this are *checksum calculation* and *zero-copy networking* [14, 15].

2.2.1. Interrupt load

Traditionally, an interrupt is generated for each packet received or transmitted. For low-speed networks such as 10Mbps Ethernet this is not a significant problem, since

the number of interrupts is still only a few thousand per second even with small packets. Gigabit Ethernet running at full speed using the standard 1500 byte packets generating an interrupt for each packet would cause nearly 85000 interrupts per second, which leaves only $12\mu\text{s}$ to handle each interrupt. With smaller packets the problem is even worse.

The reason interrupts cause a problem is that hardware interrupts usually have priority over everything else running on the same system. When an interrupt is raised, the machine has to perform a context switch, i.e. save its current state, run the interrupt handler, and then return to the previous task. If the interrupt rate is high enough, this can generate a situation commonly referred to as interrupt livelock, where the OS is so busy processing the interrupts coming from the NIC that it has very little time to handle anything else.

There are several approaches to minimize the number of interrupts, usually referred to as interrupt mitigation or interrupt coalescing. One approach is having the driver poll the card for several new packets after each interrupt if a burst of packets is detected. The system may even switch completely into a polled mode of operation if the load is high enough, but revert to interrupt driven I/O once the load becomes low enough. Polling is particularly useful for routers, where the proportion of small packets is often large. Even for end systems polling may be useful, but only when the load is high enough to cause livelock [20, 21].

Another possibility is making the adapter wait for a specified amount of time for more packets to arrive, or requiring a certain number of packets to have arrived before interrupting the host CPU. The interrupt service routine must then handle all the packets in the queue every time it is called [22]. The downside of these approaches is that they increase latency, which is not desirable in some applications. In addition, interrupt coalescing does nothing to reduce the overhead that each packet incurs in the network stack [24].

2.2.2. Protocol processing overhead

Protocol processing overheads have been widely researched during the last decade. Although the exact ways of optimizing performance depend on the protocol being used, there is one basic rule that applies to all protocols: *Optimize for the most common case: packets arriving in order without any errors.*

For TCP/IP an this can be done by using an algorithm introduced by Van Jacobson called *header prediction*, that splits protocol processing into two separate paths, the fast path and the slow path. The algorithm looks for incoming segments the receiver is expecting to receive next; namely, segments that:

1. are for connections that have been established,
2. do not have bits set that would signify urgent data or connection shutdown,
3. are the expected next segment in the sequence (i.e., continue where the previous segment left off),
4. have not changed the window size,

5. are for connections that are not retransmitting data, and
6. are either acknowledgements for data that has been sent or new data arriving, but not both at the same time.

If the segment satisfies all of these conditions, it can be processed in the fast path using only a few instructions, which consists largely of simply stripping the header off and passing the data up to the application [23 pp. 236-239].

If the incoming segment fails any of these conditions, protocol processing falls back to the full protocol processing procedure presented in RFC 793 [11], which is designed to handle all possible situations, such as connection establishment or starting the TCP error recovery process [25].

A second common operation that is required by network protocols is to look up the protocol control blocks for connections. This is especially important when the number of simultaneous connections is high [23]. Current implementations use data structures, such as hash tables, that have been optimized for the access patterns of each protocols, e.g. a simple cache that assumes that the next UDP packet is to the same connection as the previous one has been shown to have a hit rate of 80% [26].

2.2.3. Increased frame sizes

Another way of reducing the number of interrupts is to reduce the number of packets by increasing their sizes. The Ethernet standard limits the maximum size of a frame to 1500 bytes, which was adequate for the hardware and networks used 20 years ago, but is insufficient for modern networks. Ethernet technology itself does not impose any absolute limit for a maximum frame size, although the CRC-32 algorithm it uses for detecting errors can only maintain its accuracy for packets up to 11 kilobytes.

Alteon Websystems has developed an extension for Ethernet called “jumbo frames” that allows the use of frames of up to 9018 bytes. In addition to reducing the number of interrupts, jumbo frames are large enough to accommodate for entire memory pages, which can be used to optimize implementations.

Jumbo frames are supported by most GigE NIC vendors, but by only a few switch and router vendors. In practice, they can only be deployed on a separate network, as they must be supported by all of the network equipment used on a LAN [27].

Unfortunately it seems 1500-byte frames will be the standard even in the future, as the upcoming 10 Gigabit Ethernet standard will be using them. Alternate network standards such as GSN [43] and Myrinet[45] offer large packet sizes (64 kB-4 GB), but have not gained wide acceptance, mainly due to their high costs.

Research done with current high-speed networks shows that an MTU of 8 kB is adequate, as beyond that the data-touching overheads and hardware limitations become the dominating factors [3]. Larger packets may also reduce possibilities for parallelism in the protocol implementation as the host may become idle waiting for the NIC to deliver the next packet [17 pp. 342-344].

2.2.4. Hardware assisted IP fragmentation and TCP segmentation

As was shown earlier, interrupt processing overhead is a major problem on Ethernet networks due to the small 1500 byte maximum packet size. Jumbo frames and interrupt coalescing can offer some help, but both have problems associated to them; Jumbo frames are a non-standard extension of Ethernet with compatibility problems, and interrupt coalescing can make latency unacceptable for some applications. Moreover, even when they are used together they are just barely enough for GigE, but will not be adequate for future multi-gigabit networks.

An alternative approach that has been proposed is to transfer data in larger units to and from the NIC and implement protocol-level fragmentation in hardware. By placing the fragmentation and defragmentation process in the NIC, it is possible to present an MTU of 64k to the operating system, yet use 1500 byte packets on the wire. TCP/IP provides two possibilities for accomplishing this, IP fragments and TCP segmentation.

When the NIC receives an IP packet that is larger than the physical MTU from the host it splits it into multiple IP fragments. Similarly, when the NIC receives IP fragments it combines them into a complete packet before passing it on to the host. If one of the fragments is missing the entire frame is discarded.

Unlike jumbo frames, this approach is fully compatible with existing implementations, as an unmodified receiver sees standard 1500 byte fragmented IP packets which can be defragmented in the OS.

A prototype implementation of hardware-assisted IP fragmentation for Linux using a modified firmware for the Alteon Tigon-II Ethernet chip achieves similar speeds, in both cases essentially limited by the PCI bus and the NIC, as the standard firmware, but with a substantial decrease in CPU utilization [24].

A similar approach can also be taken at the TCP level by splitting TCP segments that are larger than the physical MTU into smaller sub-segments. On the receiver the segments can be combined into larger ones. This approach has been implemented on the Broadcom 5700 Gigabit Ethernet chips, which is used e.g. in the 3com 996 GigE adapter.

The greatest problem in these approaches is implementing them in a way that can handle every possible situation in a robust manner. Situations that may cause problems are e.g. packets already fragmented by the OS that are larger than the physical MTU and packets destined for remote hosts where the path MTU is less than the standard Ethernet MTU. Handling all possible special cases in a complex protocol like TCP needs to perform correctly is not trivial, especially in an embedded system with limited resources.

2.2.5. Checksum calculation

Nearly all currently used network protocols include a checksum of the data to ensure reliable transport, as most network media cannot be considered reliable. For TCP/IP-based protocols this is a simple 16-bit 1's complement sum of the data as specified by RFC 1071 [28].

The speed of calculating a checksum is ultimately limited by the speed of memory, which as we mentioned in the introduction, has not been able to keep up with network speeds, and thus checksum computation can also cause a bottleneck.

The overhead introduced by checksum calculation can be reduced using several approaches. The two most widely used approaches are the copy-and-checksum technique and hardware offload of the checksum calculation.

Copy-and-checksum was first implemented¹ in TCP/IP by Van Jacobson. He found, that on current pipelined processors used today, the copy and checksum operations can be combined into one optimized routine, which requires little to no additional resources compared to performing only the copy operation [13].

Another way of removing the overhead of checksums is to offload the calculation to the NIC. This is supported by nearly all NICs made today, although operating systems have only started using the feature recently. The usefulness of hardware checksumming is limited by the fact that all NICs claiming to support the feature do not support it reliably, or place alignment restrictions on the data to be checksummed.

Research has shown that checksum offloading alone gives only a negligible benefit on modern machines due to memory caches [30]. However, as we will later describe, in certain cases checksum offload makes it possible to avoid the data copy in the kernel (zero-copy), which provides a significant performance benefit.

More controversial approaches for checksumming have also been proposed. By placing the checksum after the data as a trailer instead of in the packet headers, packet transmission can be started before checksum computation has been finished. On the receiving end, checksum trailers offer no benefit as all of the data needs to be received and the checksum verified before it can be passed on to the application. For existing protocols, such a change in the protocol is impractical, as the change would be incompatible with existing implementations, although some new protocols such as STP support it as an optional feature in addition to the traditional way of placing the checksum in the header.

Another proposal that has been presented is to omit the checksum completely on local area networks, where the link-level CRC checksum can be assumed to find any data corruption. Current experience has shown it to be a bad idea, as hardware errors are still quite common [31].

To some extent, offloading TCP/IP checksumming to the NIC also has similar problems as it makes it impossible to detect if the data was corrupted while it was copied to the NIC, e.g. due to a bus error or bad memory on the NIC. The TCP checksum is computed using the corrupted data and the error becomes undetectable unless application-level checksums are used in addition to the ones provided by the protocol [31].

2.2.6. Zero-copy networking

In the simplified model of the BSD network stack we presented above, data is copied twice² on both transmit and receive, from the application to kernel buffers and from there onto the NIC. If the hardware being used supports both scatter-gather DMA

¹This technique was proposed as early as 1982 by David Clark in RFC817 [29]

²In older stacks there were even more copies due to a heavily layered model

and hardware checksumming, the intermediate copy to kernel data structures can be avoided during transmit and replaced with a list containing only the start locations of individual fragments.

The semantics of the BSD socket interface require that after the `write()` system call has completed, the buffer can be reused immediately. If the data is copied fully to kernel buffers, this is the case, but with zero-copy the only copy of the data is the one in the application buffers. If the data is still being sent or retransmitted, any new data in the buffer could be sent instead.

To preserve the existing semantics, memory pages containing the data are marked as Copy-On-Write (COW) by the Memory Management Unit (MMU), which disallows the application from writing to them. If the application attempts to overwrite the data, the OS creates a copy of the original data and transparently maps the copy into the user virtual address space. After the data has been acknowledged the COW mapping can be released.

It is not yet clear if this method results in any real-life performance improvement. For applications which do not reuse the buffer immediately, performance increases are possible, as the cost of the MMU operations is smaller than the cost of the copy. If the buffer is reused before the data has been acknowledged, performance is worse than it would be if the data had originally been copied [3, 32].

To get around this limitation, some operating systems have introduced a new `sendfile()` system call for transmitting data, which transfers data between two file descriptors from a physical file to a socket. As the data transfer is performed entirely by the kernel, it can be guaranteed that the data will not be changed before the transmission is complete. Therefore, there is no need to protect the data against modification using MMU operations [33].

For applications such as web servers, the use of `sendfile()` can offer a significant increase in performance. If zero-copy is not supported by the hardware, performance of `sendfile()` is similar to that of `write()`, assuming the file is already in the buffer cache of the OS and does not need to be read from disk.

Because `sendfile()` has not been standardized in any way, the exact semantics and even the name of the function may vary³. Some OSs, such as HP-UX, have an option of including a header with the data. The intention is to reduce the number of system calls, as most application level protocols such as HTTP include some response information before the actual data. In Linux, headers are sent using a separate `write()` call before the data, as system call latency is low enough not to cause problems.

Avoiding the copy on the receiver is much more difficult, as the NIC would need to know enough about the protocol state to be able to determine where the data should be placed. Also the memory page to which the data is to be transferred must be locked, so that it cannot be paged to disk.

If the packet payload is an exact multiple of the page size, avoiding the copy is possible using a technique called page flipping. When page flipping is used, the data is mapped into the virtual address space of the application using the MMU. This requires both the packet payload and the user buffer to be aligned to page boundaries.

³`splice()` is used by the paper [33] introducing the concept and `TransmitFile()` is used by Windows NT

The concept is demonstrated by Figure 5, where an application is receiving 8592 bytes of data into a buffer which starts from offset 3696 inside a 4 kB page. Three packets arrive, containing 4096, 400 and 4096 bytes respectively. Although the first packet contains a full memory page of data, page remapping cannot be used as the user buffer is not aligned, therefore the packet has to be copied. For the third packet, page remapping can be used, since the data is to be received to a complete page.

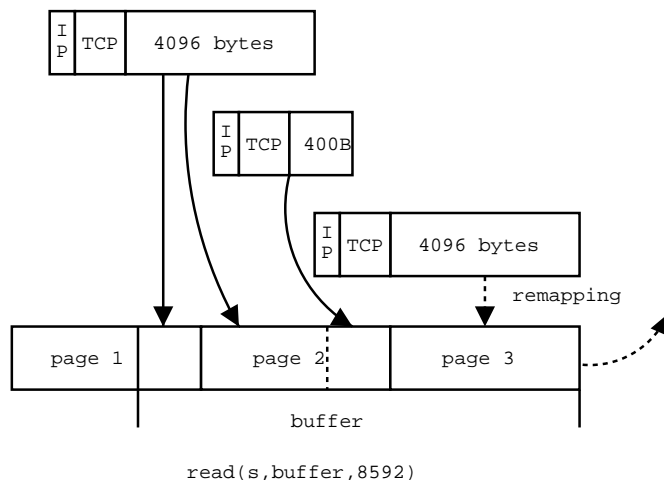


Figure 5. Page flipping.

Naturally, if this method is used, the network driver needs to place the incoming data in such a way that the data will be on a separate memory page from the headers. To ensure this, the NIC must be able to parse the protocol headers and copy the headers to a separate location from the data.

Page flipping remains a controversial topic. Implementations using it have shown a remarkable reduction in CPU use in benchmarks [34]. However, the benchmarks used test only the optimal case, where page remapping can be used for all incoming packets, and the data is never actually used afterwards. For real-life applications, this is seldom the case. Remapping pages also requires all CPUs in the machine to flush the remapped page from their translation look-aside buffer (TLB). This operation can be very costly on machines with many CPUs. Performing the copy on the receiver also has a beneficial side-effect of storing the data in the CPU cache. This makes the performance loss for performing the copy smaller, especially if the data is accessed immediately by the application.

Zero-copy TCP has been implemented for several operating systems. Here we will describe implementation for Linux, FreeBSD and Solaris for which information is available through source code or published papers.

In addition to the implementations covered here, at least IRIX⁴ and Windows 2000 support zero-copy TCP, but no published information could be found about the implementation details.

⁴According to [33], IRIX uses COW mapping and page-flipping

Case study: Linux

Zero-copy TCP for the transmit side was introduced into the standard Linux kernel in version 2.4.4. In the 2.4.4 kernel, zero-copy transmit is only supported through the `sendfile()` interface. In an earlier version of the zero-copy code[35] `sendmsg()` was also supported using a COW mapping for the data when a special `MSG_NOCOPY` flag was used to notify the kernel that the application did not intend to reuse the buffer immediately and thus a COW mapping could be made without the potential performance loss. Support for it was not included in the final version because it required some controversial changes to the virtual memory management code. The adapters supported are loopback, 3com 90x, Alteon AceNIC, Sun GEM and Sun HME.

Case study: FreeBSD

Support for zero-copy TCP is available as a separate patch for FreeBSD [34]. The patch includes a modified firmware for the Alteon Tigon-II chipset, which adds support for header splitting, which places the payload on a separate memory page if it is an exact multiple of the page size. In addition to TCP headers, the modified firmware is able to parse NFS headers, so that zero-copy NFS read operations are possible. Transmits are handled using COW and `write()`, although the `sendfile()` interface is also supported. Zero-copy is also available for Myrinet networks using a customized messaging interface on which TCP is run called Trapeze [3, 36].

Case study: Solaris

Support for zero-copy TCP was first included in Solaris 2.6 when using the “SunATM adaptor 2.0” ATM interface card. Both transmit and receive paths have been modified for zero-copy.

To avoid the problem of the data buffer being reused before the data has been completely transferred, zero-copy is only used when the total size of write buffers exceeds a specified threshold. This is done to minimize the possibility for COW faults, as zero-copy will only be used when the `write()` is guaranteed to block i.e. there is more data to be sent left than there is space in the socket buffer.

For receives, zero-copy is also achieved using page-flipping techniques, but in a way that does not require hardware support. When zero-copy receives are enabled, the TCP code will negotiate a MSS (maximum segment size) option of 8kB instead of the 9108 bytes used by IP over ATM, which is large enough to accommodate two pages of data. The network driver aligns incoming frames in a way that places the data in a typical TCP frame on a separate page. This approach works in fact for most TCP packets, as after connection establishment nearly all TCP implementations use 52 byte TCP headers, 40 bytes for IP and TCP, 10 bytes for the TCP timestamp option and two bytes for padding. If the incoming packet fails this condition, the packet can still be processed by the standard single-copy approach [32].

Starting with Solaris 8, `sendfile()` is also supported by Solaris, and it appears it will become the standard way of performing zero-copy transmits.

2.3. Summary

The network stack in the kernel is accessed through the socket layer, which presents a protocol-independent interface for networking. When data is transmitted and received, at least one copy of the data is generated in kernel memory space.

The two greatest problems affecting TCP/IP performance today are interrupt load and memory copies, both of which can be reduced using a combination of different techniques.

The copy on transmit can be avoided when the NIC supports scatter-gather DMA and hardware checksum offload, and precautions are taken against the data being modified while it is still being transmitted. For the receiver, no simple solution exists when the standard socket interface is used.

Interrupt load can be reduced by increasing the size of packets and reducing the number of interrupts by using interrupt coalescing. Both solutions have drawbacks; larger packets require support from the underlying network, and interrupt coalescing introduces some additional latency.

By combining zero-copy TCP and reducing the number of interrupts it is possible to obtain TCP/IP speeds over 100 MB/s with only modest CPU consumption on the transmitter. The receiver, however, remains heavily loaded as there is no general way of avoiding the memory copy.

In the next three chapters, we will describe solutions, which go beyond classical sockets-based TCP/IP to fully utilize the fastest networks available.

3. TECHNOLOGIES FOR HIGH-SPEED NETWORKING

In this chapter, we will review research and standards relevant to the work presented in this thesis. The approaches presented here are ones which go beyond the classic socket and TCP/IP paradigms to fully utilize current high-speed networks. The two key techniques used by the approaches presented in this chapter are *zero copy*, which was discussed in the previous chapter in conjunction with TCP, and *OS bypass*, where user applications access the network hardware directly.

This chapter is structured as follows. First, we will describe the OS bypass technique. After that, we will briefly describe currently available network technologies and how their characteristics and capabilities affect protocol implementations. Finally, we will outline several new approaches for high-speed networking.

3.1. OS bypass

After all other sources of overhead in networking have been dealt with, the biggest overhead becomes the operating system that stands between the application and the network. A modern operating system is expected to divide resources on the machine evenly and prevent misbehaving applications from affecting other processes. At the same time, the operating system incurs a significant overhead in the form of e.g. system call latency and context switches.

If very low latency is desired, a technique called *OS bypass*, where the applications communicate with the NIC directly, may be advantageous. The simplest form is to simply allow applications to write directly to the NIC registers. This approach suffers from a number of drawbacks, as it is completely hardware specific and precludes multiple applications from accessing the network simultaneously.

A better way is to allow applications controlled access to the resources of the NIC and OS kernel for connection and resource management. For optimal performance, the NIC should have at least some level of support for OS bypass, such as multiple transmit and receive queues that can be split between applications. The NIC on the receiver must also be able to distinguish between different applications and place the incoming packets on the correct receive queue [37, 38].

To enable applications use of these facilities several different interfaces, such as U-Net [38], libst [39, 40] and VIA [41] have been developed. These libraries provide a hardware-independent method of using OS bypass, yet remaining relatively low-level to provide the best possible performance.

The greatest drawback of OS bypass is that it makes applications significantly more complex compared to traditional networked applications, as the applications are required to manage all aspects of networking, such as placing data into packets, managing the transmit and receive descriptors and waiting for new incoming data. This paradigm is quite different from the one application-writers are used to. To simplify this task, generic higher-level libraries, such as MPI [42] and socket emulation, running over technology-specific OS bypass solutions are often used. Although the use of these libraries can introduce a performance penalty, they are still useful for bringing some of the benefits of OS bypass into existing applications.

3.2. Network interconnects

The properties of network technologies influence the types of protocols that can be used with them. The aspects of network technologies that are most relevant to the for this thesis are: *bandwidth, latency, support for hardware acceleration, frame size, reliability* and *cost*. High-speed local area networking can be roughly divided into two categories, traditional Local Area Networks (LAN) and System Area Networks (SAN¹).

SANs differ from LANs in that SANs are more coupled with the end systems. They offer reliable data transmission on the hardware level, making it unnecessary to use reliable data transfer protocols with them. They typically use Remote Direct Memory Accesses (RDMA), where data can be transferred directly between buffers in the two end systems without involving the host CPU at either end. A SAN is usually used to interconnect nodes within a distributed computer system, such as a cluster. The network is assumed to be physically secure.

The network interconnects described here are:

- Ethernet,
- Gigabyte System Network,
- Infiniband, and
- Proprietary System Area Networks.

Of these network types, Ethernet represents a LAN technology, whereas the rest are closer to SAN technologies.

3.2.1. Ethernet

Ethernet is by far the most common type of network technology used today. During the last three decades it has evolved from the original half-duplex shared media 3Mbps version of 1973 into the current switched Gigabit Ethernet. Still, the basic technology has remained, making GigE networks completely compatible with existing 10 Mbps and 100 Mbps Ethernet networks. Although reliable packet delivery is not guaranteed on the hardware level, packet switching has also made Ethernet reasonably reliable due to the lack of collisions,

Backwards compatibility also brings problems, as the original 1500 byte maximum packet length has not been increased, which makes the interrupt rate high on GigE networks unless interrupt coalescing is used. Compared to the other technologies presented in this section, the latency of Ethernet is high (ranging from 20 μ s to several hundred μ s depending on the NIC hardware, network speed and OS).

The most appealing feature of Ethernet is its price. The cheapest GigE NICs can be obtained for only \$150-200 USD (November 2001), far less than competing technologies. The popularity of Ethernet has also made many kinds of NICs available, ranging

¹The same acronym is used for both System and Storage Area Networks

from extremely cheap and simple designs to programmable NICs with an on-board general-purpose CPU.

In 2002, 10 Gigabit Ethernet will become available. In addition to the higher speed, it will also be possible to run 10 GigE directly over OC-192 networks, making it possible to run Ethernet over long distances. For hosts, 10 GigE will be problematic, as potentially millions of packets per second will have to be processed. This will make interrupt mitigation techniques even more important than they have been in the past.

3.2.2. Gigabyte System Network

Gigabyte System Network (GSN) [43] is the successor for HIPPI, a 800 Mbps point-to-point network used during the 1990's for supercomputing applications. GSN runs at 6400 Mbps and has a latency of 5 μ s, which puts it far ahead of other currently available systems. It is mainly used for interconnecting supercomputers like SGI Origin 2000/3000's. It guarantees reliable packet delivery on the hardware level, and supports messages of up to 4 GB.

The primary protocol used with GSN is STP, which is the only protocol available that fully utilizes GSN. The SuMAC chipset from SGI, which is used by nearly all GSN implementations, supports STP acceleration on the hardware level. It is also possible to run other protocols such as TCP, but performance is much lower, only around 400 MB/s. The high performance comes with a cost, however, as a single GSN interface for PCI costs as much as \$15000 USD (March 2001).

3.2.3. Infiniband

Infiniband [44] is a new SAN technology sponsored by nearly all major computing vendors, including Intel, IBM, Microsoft, Sun and HP. It supports a wide range of applications from a backplane interconnect of a single host (replacing PCI) to connecting a large number of independent hosts and I/O components (replacing Ethernet and Fibre Channel). Addressing between different nodes on a Infiniband network is done using IPv6 addresses. The first implementations of Infiniband are scheduled to arrive in 2002.

Infiniband is not just a hardware solution. The standard includes a full featured protocol layer that is very similar to software-based solutions such as STP, presented below.

Infiniband comes in three different versions, 1x, 4x and 12x, which run at 2.5 Gbps, 10 Gbps and 30 Gbps, respectively. On the physical level it uses 4096 byte frames, but supports reliable transfers in units of up to 2 GB. Details of how the software interface works were a tightly held secret in November 2001, but it is foreseeable that some parts of the Infiniband stack will be placed in the OS, with the Infiniband hardware accelerating performance critical functions.

3.2.4. Proprietary System Area Networks

In addition to the standardized network types described earlier, several proprietary high-speed network interconnects such as GigaNet, Myrinet [45] and Servernet are available. They typically offer speeds of 1 Gbps-4 Gbps. They are intended for situations, where standard LANs like Ethernet cannot offer the required performance. An example of such an application are clusters, where latency and RDMA functionality are important.

3.2.5. Summary of network technologies

Ethernet will almost certainly remain the dominant LAN technology for a long time, despite its problems like the small maximum frame size. Using a variety of techniques including hardware acceleration, it can be a very attractive low-cost solution for nearly all applications, especially those involving wide-area networking.

GSN is the fastest currently available network interconnect technology, but due to the high cost it has not become popular outside a few specialized applications. GSN also has very few supporters, with the majority of vendors behind Infiniband. However, GSN is faster than initial Infiniband products running at 2.5Gbps (1x speed) will be, meaning it will continue to be the fastest network technology for the near term.

Infiniband will most likely become the dominant SAN technology due to the large support behind it. Its main obstacle is probably defining its role, as it tries to be everything for everybody; a storage bus, a peripheral bus and a network interconnect. Whether it succeeds in all of these goals remains to be seen. It is also possible that Infiniband will remain a high-end solution, unlike Gigabit Ethernet.

The capabilities of the network technologies presented are summarized in Table 1.

Table 1. Capabilities of different network interconnects.

	Ethernet	GSN	Infiniband	Proprietary SAN
Speed (Gbps)	1 ^a	800	2.5/10/30	1-4
MTU	1500 ^b	4GB	2GB	varies
Cost	Low	High	Medium(?)	Medium
Hardware acceleration	Some ^c	Yes	Yes	Yes
Latency	High	Low	Low	Low
Reliable transport	No	Yes	Yes	Yes

^aWith 10 Gbps available in 2002

^bWith 9000 byte jumbo frames as a non-standard extension

^cOn some NICs

3.3. Protocols for high-speed networking

In this section, we will present several protocols, which utilize zero copy, OS bypass and hardware acceleration.

The solutions presented here are:

- Scheduled Transfer Protocol,
- Arsenic,
- Direct Winsock and WSDLite,
- TCP RDMA,
- U-Net, and
- Virtual Interface Architecture.

3.3.1. Scheduled Transfer Protocol

The Scheduled Transfer Protocol (STP) [18] is a new connection-oriented data transfer protocol designed for high-speed local area networks. STP was originally designed as part of the ANSI/NCITS standardization work for Gigabyte System Network (GSN), a high-speed network media supporting speeds of up to 6400 Mbps, since it was obvious traditional network protocols could not meet the needs of GSN.

As the standardization effort continued, it was obvious that STP would not be finished at the same time as GSN. It was also clear that STP would be useful on other high-speed networks such as Gigabit Ethernet. Therefore work on STP was broken off into a separate standard.

The main platform for STP is IRIX, where STP has been demonstrated to run at speeds of 798 MB/s with a latency of $5 \mu s$ over a single GSN link between two Origin 3000 servers. SGI has also released an implementation based on the IRIX code for Linux. Third-party versions are also available for Solaris and Tru64 UNIX, primarily for use as a networked storage solution.

STP is heavily based on the Hamlyn system from HP [46, 47], which introduced the concept of *sender-based memory management*. The basic idea behind sender-based memory management is giving the sender the responsibility for laying out messages in the receiver's memory. This avoids the possibility of buffer overrun on the receiver, as a buffer must be reserved on the receiver before any data transfer can happen.

In STP small control messages are used to allocate buffers on the remote host before any data transfer. This reduces the workload of the receiver considerably and makes hardware acceleration relatively simple to implement. Unlike traditional protocols, STP achieves zero-copy without requiring the data in the user buffers to be aligned to page boundaries so that tricks like copy-on-write or page-flipping can be used. To accomplish these goals, STP must make several compromises. The network is assumed to be secure, reliable and have very low latency.

STP has two basic different modes of operation. The method used for high-bandwidth bulk data transfers uses the flow-controlled Read and Write sequences. In this mode of operation, every time data is transferred a new buffer has to be reserved on the receiver. For low latency messaging, STP also supports non-flow-controlled persistent-memory Get, Put and FetchOp sequences, which can be available directly

from userspace, bypassing the kernel entirely. In this mode, a buffer on the remote host is reserved permanently and subsequent operations operate in this memory area.

STP can be used through several different APIs, including BSD sockets and libst [39, 40], which is a OS bypass library designed for use with STP.

STP is also being adopted as a solution for networked storage. The SCSI over STP (SST) standard [48] defines a method for encapsulating SCSI commands, data transfers and responses using STP. Even though the SST standard has yet to be ratified, commercial solutions utilizing it are available from some vendors.

3.3.2. *Arsenic*

Arsenic [30] is an interface to network adapters that supports user-space networking. It extends on the ideas of U-net and other similar solutions by introducing a sophisticated way for the card to distinguish between different flows.

As we described earlier in this chapter, user-level networking requires the NIC to be capable of distinguishing different flows by analyzing the headers of the packet. This makes the support of multiple different user-mode protocols complicated, as support has to be added for each one of them separately into the NIC. Additionally, the data-structures required for the protocol-specific lookup tables require memory on the NIC which, as in all embedded systems, is usually in short supply.

Arsenic solves this problem by dynamically generating a transmit and receive packet filter for each flow, which get uploaded to the NIC when the flow is established. The packet filters are generated as native machine code for the NIC CPU; this makes matching packets a relatively simple task, requiring only about 50-70 CPU cycles per packet.

When transmitting data, the application places a descriptor for the data to be transmitted as a scatter-gather list onto the transmit queue. To prevent rogue applications from falsifying packet headers, the NIC verifies the packet against the transmit packet filter and passes only packets that match.

For the receive descriptor, the application may tag different fragments to hold a packet's header, payload or trailer. When the receive packet filter is executed, it returns the virtual interface to which the packet should be delivered and the offsets of various parts of the packet.

Arsenic also provides some Quality of Service (QoS) features. Applications can tune the interrupt generation policy for individual flows. An application performing bulk data transfer can request a very low rate of notifications, whereas a latency-sensitive application may request an interrupt as soon as the receive ring becomes non-empty.

The current release of Arsenic, 1.0, includes a modified firmware for the Alteon Tigon-II Gigabit Ethernet NIC and modifications to Linux 2.3.29 to utilize the modified firmware.

To let existing applications use the facilities provided as easily as possible, Arsenic includes a full port of the Linux TCP/IP protocol stack implemented as a standard user-space shared library. When an application is linked against the library, the C library functions dealing with sockets are replaced from ones performing system calls into ones utilizing the user-mode TCP implementation.

3.3.3. *Direct Winsock and WSDLite*

When a network medium that ensures data integrity and reliability, such as most proprietary SAN technologies, is used, the overhead involved in providing the same functionality in the network protocol is not necessary. This means, however, that applications will need to be modified to use these protocols when using a reliable medium. For existing applications, this is not always possible.

Direct Winsock [49] is a library included with Windows 2000 Datacenter, that allows unmodified existing applications to utilize specialized SAN-boards. It does this by trapping WinSock2 API calls to the standard TCP/IP protocol stack. If a connection is established to a remote host that can be reached through a fast SAN connection, the connection is diverted to the native API of the SAN, which can be e.g. VIA. Since the network is reliable, there is no need to run TCP/IP, but instead simply provide a two-way reliable connection with the same, SOCK_STREAM, semantics as TCP. Direct Winsock implements the entire WinSock2 API making it possible to run nearly all existing applications through it.

WSDLite [50] is a simplified version of the full Direct Winsock done by Speight et al. It only implements the basic subset of WinSock 2 functionality that is required to run most applications, and is only slightly slower than the native API. The performance of TCP on the same network is approximately half of the performance of WSDLite.

3.3.4. *TCP RDMA*

To make the implementation of zero-copy TCP stacks easier, there have been proposals for TCP extensions which would make it easier for network cards to copy the payload directly into the user buffers. Instead of having to add support into the network card for different application protocols such as CIFS, NFS and HTTP separately, it is enough to have one common way of storing the location of the payload in the packet.

At the end of year 2000 there were two IETF Internet-Draft documents concerning TCP Remote DMA (RDMA) in development. Both of the proposals were at a very early stage and the descriptions here are merely to provide an idea on the approaches that have been considered.

TCP RDMA is a proposal from Cisco Systems to make it possible to use simple hardware acceleration with standard TCP-based services such as NFS, CIFS, HTTP and SCSI over TCP. It does this by adding a TCP option header (number 25), which is shown in Figure 6.

Each RDMA transfer has a unique 46-bit identifier (RID), which is used to map the transfer to an application buffer. The format of the RID depends on the application, e.g. in NFS the 32-bit RPC transaction ID (XID) is used [51].

The other draft is from James Williams. Instead of making any changes to the TCP protocol itself, it only defines a common data format for encapsulating RDMA information within a TCP data stream. The same format can then be shared by multiple client protocols such as VI/TCP and iSCSI, making it simpler to implement hardware acceleration in the NICs [52].

Both TCP RDMA drafts had expired in 2001 and have been removed from the IETF draft repository. Some interest in this area still exists, including a TCP encapsulation

Byte	0								1								2								3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	25								Length								A	U	RDMA ID													
4	RDMA ID																															
8	Buffer Offset																															
12	Data Offset								Data Length																							
16	Total RDMA Length																															

Figure 6. TCP RDMA header.

of VIA and similar RDMA solutions for use with iSCSI (Internet SCSI, which is SCSI run over IP). However, none of these proposals had progressed beyond early drafts and discussion.

3.3.5. U-Net

U-Net is a protected, user-level network architecture for low-latency and high-bandwidth communication developed at Cornell University.

The last release of U-Net, 2.1, supports Myrinet, Fore ATM and DEC tulip Fast Ethernet under Linux, BSD/OS and Solaris.

U-Net is the first communication architecture using commodity hardware that provides applications with a virtual view of a network interface to enable user-level access to high-speed communication devices.

Depending on how sophisticated the hardware is, the application may access either the hardware directly, a special memory location that is interpreted by the OS, or a combination of the two.

When an application wishes to access the network, it must first create one or more U-Net endpoints, and then associate a communication segment as well as a set of send, receive and free message queues with each endpoint.

Messages are sent by placing a descriptor for the message onto the send queue. The network interface code polls the different send queues continuously and sends the messages placed in them into the network as soon as possible. When the transmission has been completed, the NIC sets a transmission complete flag in the descriptor to let the application know that the descriptor can be reused [38].

Development on U-Net ended in 1998, but the ideas behind it still live on in newer systems such as VIA.

3.3.6. Virtual Interface Architecture

The Virtual Interface Architecture (VIA) [41] is a high-performance local area network architecture specified by a consortium of Intel, Compaq and Microsoft. VIA is heavily based on research projects such as Active Messages [53], U-Net [38] and Fast Messages [54].

The unusual thing about VIA is that only the architecture and the API [55], not the protocol used on the wire, have been specified. Naturally, this makes any two implementations of VIA incompatible, which precludes the use of it in heterogeneous networks. The API defines several levels of functionality which implementations can follow, Early Adopter, Functional and Full conformance.

The structure of VIA is shown in Figure 7 [41]. It consists of four components: Virtual Interfaces (VI), Completion Queues, VI Providers and VI Consumers. The VI Provider consists of a physical network adapter and a software kernel agent. The VI Consumer consists of an application program and an operating system communication facility.

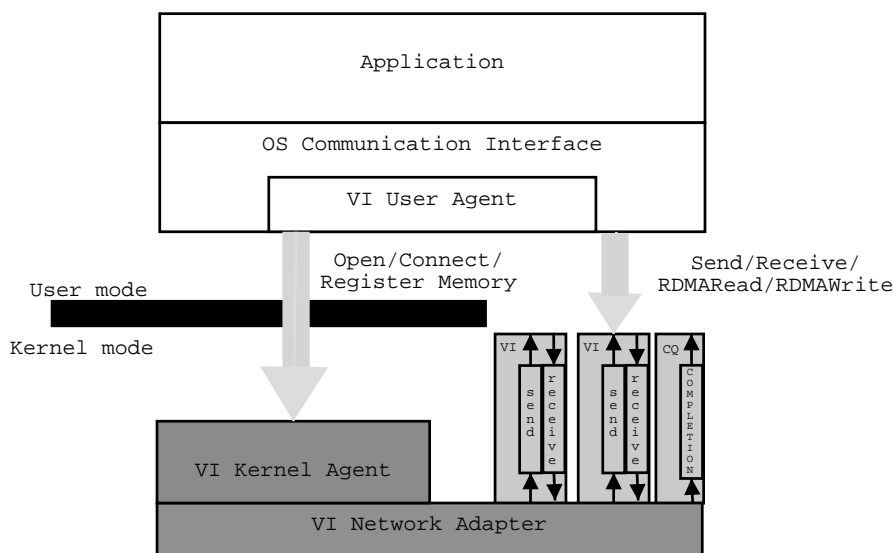


Figure 7. The VI Architectural Model.

VIA is designed to allow zero-copy data transfers, place the protocol stack entirely in user-space, avoid interrupts and make hardware implementations simple. It does this by creating a virtual interface for each application. Like ordinary network interfaces, these interfaces consist of transmit and receive queues, where software may place and receive packet descriptors. The descriptors are processed by the NIC either asynchronously or through a notification mechanism, and removed when they have been processed. Completion Queues are used to coalesce notifications about the status multiple VIs in a single location.

The VI Consumer represents the user of a Virtual Interface. Applications using VIA can do so through a number of interfaces. The VI API can be either used directly, or through higher level interfaces like MPI [42] or sockets. The OS interface makes system calls to communicate with the kernel agent, which is used for VI creation and destruction, connection setup and teardown, interrupt processing, memory management and error handling. Before any data can be transferred, the user must register the memory that will be used for networking. This removes the need for data copies when transferring data.

Due to the lack of compatible implementations, VIA has only been popular as a common interface for proprietary SAN adapters. Although different SAN adapters usually do not interoperate, applications can (in principle) be easily adapted for new

VIA-capable SAN adapters. Proposals have also been made to encapsulate VIA over TCP [56], akin to the approach taken by TCP RDMA.

M-VIA [57] is an implementation of VIA for Ethernet networks. Due to the limited capabilities of most Ethernet board, M-VIA emulates most of the functionality of a VIA capable NIC in software. Even without hardware acceleration it is able to achieve about 65% less latency on a Fast Ethernet network compared to a kernel-based TCP implementation.

3.4. Analysis

Table 2 presents a summary of the solutions presented in this thesis.

Table 2. Summary of the solutions presented in this thesis.

	STP	VIA	Zero-copy TCP	Direct Winsock	Arsenic	U-Net
Zero-copy transmit	Yes	Yes	Yes	Yes	Yes	No
Zero-copy receive	Yes	Yes	Limited	Yes	Yes	No
Socket API	Yes	No	Yes	Yes	Yes	No
OS Bypass	Yes	Yes	No	Yes	No	Yes
Compatible between implementations	Yes	No	Yes	No ^a	Yes	No
Modifications required to applications	Some	Yes	No	No	No	Yes

^aImplementations will fall back to TCP over a standard network

Scheduled Transfer Protocol is a standardized protocol for use in high-speed networks. It has been widely accepted in the high-performance networking community as the standard protocol used by GSN. On networks other than GSN there has not been much interest in STP, even though it can offer some benefits even on GigE.

The lack of a standardized wire protocol limits the use VIA on commodity networks. It has, however, gained acceptance as an API for SAN adapters. The future of VIA looks uncertain, as it is no longer actively supported by its developers, Intel, Microsoft and Compaq, who have already moved on to Infiniband. However, user-level networking as performed by VIA (and its predecessor U-Net) has proved its worth, and it is likely that one of the application interfaces supported by Infiniband will be heavily based on VIA,

Direct Winsock and WSDLite were examples of ways on how to better utilize SAN technologies with current applications that have been written for LANs. It is difficult to say whether such a solution will be practical, or if it would be better to run TCP over the SAN or to rewrite the application to use more suitable APIs. This will probably become a very important question when Infiniband starts being deployed

TCP RDMA is still in the early planning stage. It is unlikely it will ever be widely accepted outside perhaps a few specialized applications like iSCSI. This is due to the

very nature of TCP, which seeks keep the protocol general purpose and avoid any major changes to it.

Arsenic has some very interesting and unique ideas behind it. It promises a generalized solution to a relatively complicated problem, analyzing packets on a NIC and associating the packets with different connections. It is, however, also a very specialized solution that is dependent on the NIC being programmable.

Of the technologies presented, STP is the most versatile. It can be used on a variety of networks, including Ethernet and GSN and supports both traditional kernel-based networking as well as OS bypass. The availability of an early version of STP on Linux released by SGI as Open Source software also made it possible to study and improve the protocol implementation, which made it very attractive for the purposes of this thesis. In the next two chapters, we will study STP in more detail.

4. SCHEDULED TRANSFER PROTOCOL

In this chapter, we will study Scheduled Transfer Protocol (STP) in more detail. This chapter is structured as follows. First, we will describe the basic concepts of STP including a description of the protocol header. After that, we will describe the most important operations, connection establishment and data transfer. Finally, we will look at the different interfaces through which STP can be used.

4.1. Concepts of STP

The basic design principle of STP is that as much work as possible should be performed by the transmitter. Before any data transfer happens, small control messages are transmitted to pre-allocate buffers at the receiver before the data movement begins. The data can then be directly moved from the network into host memory.

In STP, data is transmitted in Transfers¹, which have a predetermined length. As shown in Figure 8, Transfers consist of one or more Blocks, which are the level flow-control is performed on. Blocks are further divided into STUs (Scheduled Transfer Unit), which correspond to physical packets on the wire.

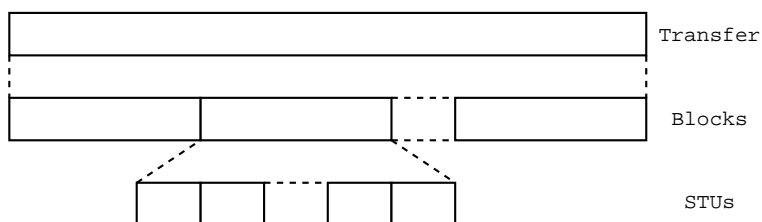


Figure 8. Data hierarchy.

The protocol header used in STP is 40 bytes, and is shown in Figure 9. Control operations may also contain exactly 32 bytes of Upper Layer Protocol (ULP) specific options, such as human-readable error messages. Data operations cannot have any options, which makes them much easier to handle in a hardware-accelerated implementation.

The 5-bit Op field contains the operation code of the packet, such as Data, Request_To_Send and Request_Connection. The other fields have different uses depending on the operation type, and may not even be used at all in some operations, in which case they are transmitted as zeros.

The 11-bit Flags field contains flags pertaining to the current operation. In connection establishment, they are used to denote the capabilities of the endpoints (support for persistent memory operations, checksums etc.).

The Interrupt and Silent bits are used to denote how the receiver should process the incoming packet. When the Silent bit is on, the packet header is not delivered to any ULP. This can be used for reducing overhead by suppressing all but the final Schedule Header delivery to the ULP for a Block containing many STUs. The Interrupt bit requests that a signal or interrupt be generated and delivered to the appropriate ULP.

¹The capitalized forms are used throughout this thesis when referring to STP Transfers and Blocks

Byte	0								1								2								3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	Op				Flags								Param																			
4	D_Port								S_Port																							
8	D_Key																															
12	Checksum								B_id																							
16	Bufx																															
20	Offset																															
24	Sync																															
28	B_num																															
32	D_id																															
36	S_id																															

Figure 9. STP header

These bits used together provide for three delivery modes for Data operations: silent, polled, or interrupt-driven. When the Send_State bit is set the receiver returns an acknowledgment that a Data STU has been received correctly.

The Flags field is also used for setting which Data Channel should be used for data transfer. This is useful on GSN, which supports multiple data channels with different characteristics.

The 16-bit Param field is used for operation specific parameters, e.g. in Data operations it contains the STU number inside the block, and in Clear_To_Send operations it contains the block size the receiver is expecting to receive.

The D_Port and S_Port contain the STP ports of the destination and source. They are assigned by each local end device upon connection setup and have no meaning to the remote end device. In addition to the port, a 32-bit Key is chosen for each connection. It should be noted that the source and destination addresses are not included in the header at all, only the D_Port and Key are verified. All responses are sent to the host from which the packet originated. This allows for striping connections over multiple interfaces, e.g. by sending simultaneous Clear_To_Send for different Blocks through separate interfaces.

The Checksum field contains a 16-bit checksum that is similar to the standard IP checksum defined in RFC 791 [10] to ensure data integrity on unreliable networks. The only change is allowing checksums to be omitted by setting the field to zero. Additionally, checksums can be calculated over segments containing multiple packets and be included with either the last packet of the segment or a separate Data STU containing only the checksum.

For Data transfer operations the B_id field contains the Memory Index (Mx), which may be assigned by the data Destination to provide an association between an expected Data operation and the Clear_To_Send, Memory_Region_Available, Get or FetchOp operations. It may be used at to aid quick parameter validation and address lookup. In the next chapter we will describe one possible scheme for the use of the Mx field in more detail.

The 32-bit Bux and Offset fields contain the Buffer Index (Bux) of the operation and the offset inside the buffer. A Buffer Index is a locally significant identifier for a physically contiguous memory region, typically individual memory pages.

The Sync field provides a mechanism for a data source to identify and track subsequent operations within a sequence. For example, if a Data operation has the Send_State bit set, the Sync field specified in the Data STU is used in the reply.

Finally, the B_num field is used for storing the Block number and the D_id and S_id fields are used for distinguishing different simultaneous Transfers [18].

4.2. Connection setup

Before any data can be transmitted a Virtual Connection has to be setup between the endpoints. To open a STP Virtual Connection, the Initiator sends a Request_Connection operation to the responder. The request consists of the port number at both the Initiator and Responder as well as the capabilities of the Initiator such as the amount of slots available at the Initiator, the maximum STU size and whether it is capable of sending and receiving Blocks in any order. Both ends also assign a Key for the connection.

When the Responder receives a Request_Connection, it responds with a Connection_Answer. It may either reject the request, or accept the connection including the capabilities of the Responder as above.

Other methods of setting up connections are also allowed by the standard. This may be beneficial if a large number of connections between several nodes are required. For example in a cluster with a static amount of nodes one possibility is setting up a connection to all other nodes when the machine is started.

4.3. Data transfer operations in STP

STP has two different modes of transferring data, non-persistent-memory “long messages” and persistent-memory “short messages” data transfer operations. When long messages are used, a new buffer is reserved for each incoming Block. Long messages are primarily used for bulk data transfers, where the cost of setting up a new buffer is offset by the savings obtained by the use of STP. Short messages on the other hand allocate a memory region once and reuse it for all subsequent operations. They are primarily used when low latency is desired, even at the cost of making applications significantly more complex.

4.3.1. Non-persistent memory data transfers

There are two types of non-persistent-memory data transfer operations, Read and Write operations.

The Write sequence is shown in Figure 10. When the Initiator wishes to send data, it issues a Request_To_Send operation. If the Responder is not ready to receive data at this time (e.g. the application is not performing a `read()`), it responds with a Request_Answer notifying the Initiator that it could not receive more data at this time.

If it is ready to receive data, it reserves buffers for each Block it is ready to receive and issues a `Clear_To_Send` for each of them. The Initiator then sends the data as one or more STUs. STP places an important restriction on the size of STUs: *STUs may never cross buffer boundaries on the receiver*. This simplifies hardware-based implementations, as this guarantees that the memory is physically contiguous in host memory.

Implementations may also set the `Send_State` bit in `Request_To_Send` and `Data` operations that ask the Responder to notify the Initiator when a Block has been completely received. This is necessary on unreliable networks, where packets might get lost.

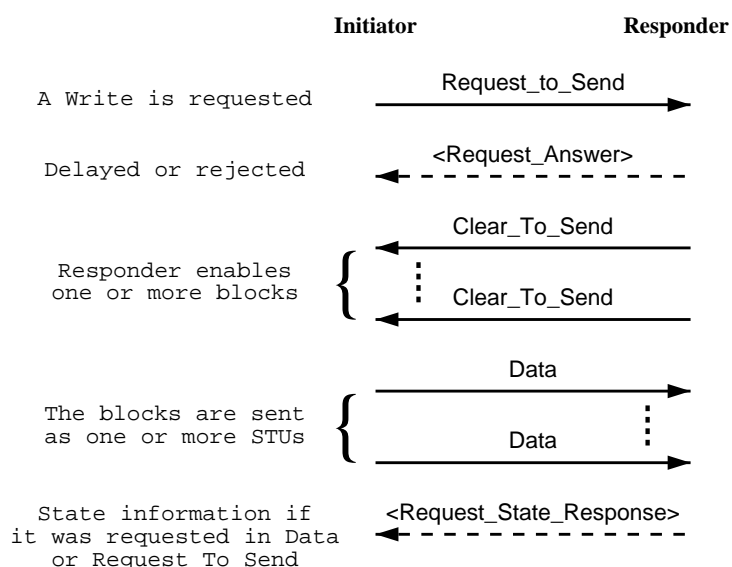


Figure 10. Non-persistent memory data transfer.

The Read sequence is otherwise identical to the Write sequence, except that the `Request_To_Send` is preceded by a `Request_To_Receive` and the roles of the Initiator and Responder are switched.

Figure 11 shows how a Transfer of 5632 bytes could be sent using 2k blocks into offset 512 of a 4 kB buffer using 1 kB STUs. The data is sent as Blocks of 2048, 2048 and 1536 bytes. Since STUs cannot cross buffer boundaries, the second Block is split into 3 STUs of 1024, 512 and 512 bytes.

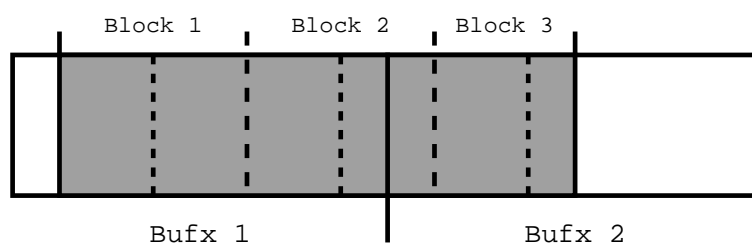


Figure 11. Example of non-persistent memory data transfer.

A better way of transferring the data would be to send Blocks of 1536, 2048 and 2048 bytes. This is allowed by the STP specification (the first and last Block of a

Transfer can be short) and has the advantage of making all Blocks after the first one aligned, and requiring one packet less to be sent, as the buffer boundary is not crossed by the second Block.

4.3.2. Persistent memory operations

For low-latency communication, the extra round-trip required for reserving a buffer on the receiver in the Read and Write operations is unacceptable. For these applications, STP supports a second type of data transfer, where a memory region on the remote host is reserved only once. The three operations, Get, Put and FetchOp are shown in Figure 12.

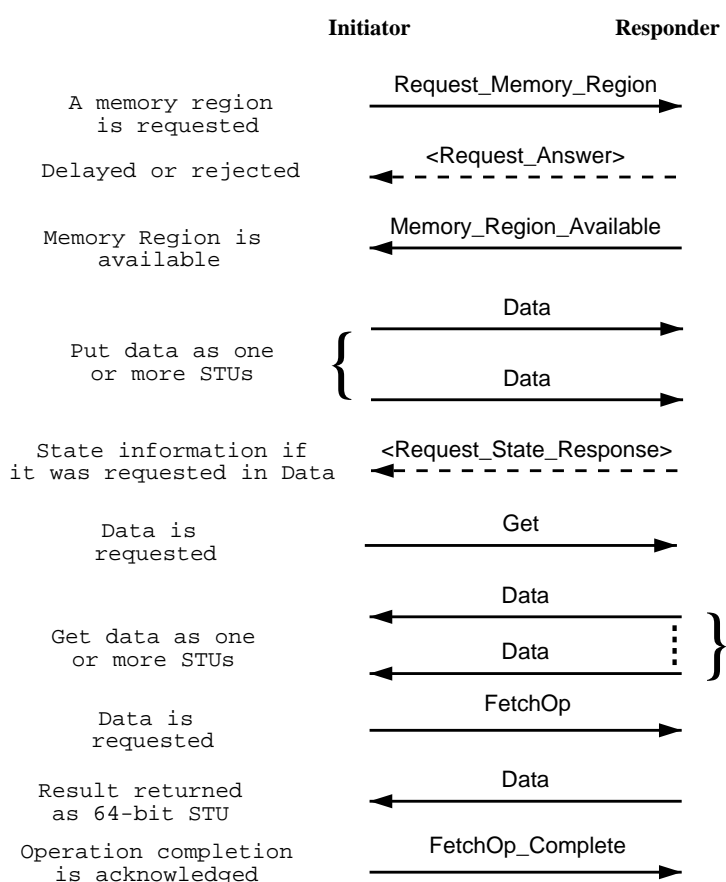


Figure 12. Persistent memory data transfer.

To request a persistent memory region, the Initiator sends a `Request_Memory_Region` operation to the remote host. If the request could not be satisfied, a `Request_Answer` is returned telling whether the request was rejected completely, or whether it can be fulfilled at a later time, for example after some resource becomes available. If the request could be fulfilled, a `Memory_Region_Available` is returned by the Responder.

Put operations are used to transmit data into the remote memory region. No explicit operation code is required, as the sender can simply send `Data` STUs corresponding to the area to be updated. The Initiator may request a confirmation of a successful operation by setting the `Send_State` bit in the `Flags` field of the `Data` STU.

The Get operation is used to request data from the remote memory region. The Responder returns the requested data as one or more STUs. No acknowledgment is required, as the Initiator can simply resend the Get if no response is received.

The FetchOp operation performs an atomic operation, such as fetch and increment, on a 64-bit memory region and returns the original value. This can be used for implementing locking primitives such as semaphores across the network in a distributed application. The Initiator must acknowledge the operation with a FetchOp_Complete to notify the Responder that the original value has been received correctly. The Responder can then process further FetchOp operations from other hosts on the same memory region.

4.4. Upper Layer Protocols

Currently there are three different ULPs that are commonly used in STP implementations, BSD sockets, SCSI over STP (SST) and libst. In this section, we will briefly describe these interfaces and how they are used.

4.4.1. Sockets

Since sockets are already used by nearly all applications that use the network, supporting them as an ULP for STP is natural. A typical TCP-based application can be converted to use STP simply by changing the socket type to `SOCK_SEQPACKET` and the protocol to `IPPROTO_STP`. The functions in the socket API match the operations in STP roughly as shown in Table 3.

Table 3. STP socket operations.

Socket API	STP Operation
<code>connect()</code>	Request_Connection
<code>accept()</code>	Connection_Answer
<code>write()</code>	Request_To_Send, Data
<code>read()</code>	Clear_To_Send
<code>shutdown()</code>	Request_Disconnect

As can be seen from the table, a `write()` maps into a single Transfer. The kernel splits the Transfer into suitably sized Blocks and STUs. Since setting up a Transfer is a relatively complex operation, the amount of data written should be as large as possible, preferably 128k or more. The semantics of `SOCK_SEQPACKET` differ slightly from the one used by TCP. Instead of the transmitter and receiver being able to write and read any amount of data, with the kernel buffering data on the receiver until it is read, `SOCK_SEQPACKET` requires the receiver to receive exactly the same amount of data that has been sent by the corresponding `write()`. If the reader tries to read less than the complete Transfer, the Transfer is refused and the `write()` returns a “Message too large” error.

The downside of sockets is that they only allow the use of a very small subset of the features in STP, connection management and Write sequences. However, they enable existing applications to be used with STP with only small modifications.

4.4.2. *libst*

Libst is a library that provides low latency access to high-speed network hardware using STP. It achieves low latency by providing a small network hardware independent layer to access network hardware that bypasses the operating system. Programming for libst is considerably more complicated than for sockets, as the application needs to manage memory and network resources.

There are two versions of libst, v1.0 [39] and v2.0[40]. The latter is a complete redesign based on the experiences gained on using the IRIX implementation of libst v1.0. Even version v2.0 is still under development, although a version of it is included in IRIX 6.5.12.

Libst provides an object-oriented interface for accessing network hardware directly. The objects it provides are: *Interface*, *Connector*, *Bufxrange*, and *Mx*.

An Interface represents a physical device. When an application wishes to utilize OS bypass, it must first open the device (e.g. `/dev/gsn0`). Depending on the capabilities of the hardware, several applications can use the same physical device at the same time.

Connectors are allocated from an Interface. They represent a dedicated port on a physical device that can be used to communicate with other Connectors. Control and Data STUs are transmitted through the `tx_ctl()` and `tx_data()` methods. The headers of the packet must first be filled in by the application.

Bufxranges represents a range of Bufx's, which are physically contiguous areas of memory (usually entire memory pages). Bufx's are assigned to a device by the kernel. User allocated memory can only be used for transmit and receive operations after it has been mapped to a Bufxrange by the kernel.

Memory Indices (Mx) are used to associate a Connector to a Bufxrange. They are used to correctly route incoming messages on an ST port to allocated memory (Bufxrange). They are also used for authentication.

The steps required for preparing a buffer are:

1. allocate the memory,
2. align the memory to page boundaries,
3. optionally pin the memory (preventing it from being swapped out),
4. allocate Bufx's to cover the memory,
5. associate the memory with the Bufxrange, and
6. for receive buffers, allocate a Mx to associate the Connector with the Bufxrange.

The user is responsible for steps 1 and 2. The Bufxrange object supports steps 3,4 and 5. The Mx object supports step 6.

After a connection has been established with libst, the application is responsible for processing all subsequent packets. This involves filling out the complete packet

header for outgoing packets and processing incoming Clear_To_Send operations and transmitting the correct data to the correct location on the receiver [40].

Since libst is relatively complicated to use, SGI has implemented MPI, a high-level message passing API on top of libst. This offers most the benefits of libst (including low latency), yet makes it significantly easier to write applications, as the user does not have to worry about processing individual packets or memory regions.

4.4.3. SCSI over STP

One attractive use of STP is storage area networking (SAN), where storage devices are attached directly to the network. The SCSI over STP standard specifies a method for encapsulating SCSI commands on top of STP.

A SST connection is established by making a STP virtual connection to the well-known port number for SST. After the connection has been established, the end-points negotiate SST parameters using a STP Nop operation, which includes information like the maximum supported number of SCSI targets, data alignment restrictions and whether the SST endpoint is capable of performing as a SCSI Target or Initiator.

Endpoints which are capable of being SCSI Targets request a persistent memory region on the Initiator, which is used to store status information for every SCSI command the Initiator sends.

SCSI Read and Write commands are mapped into STP Request_To_Receive and Request_To_Send operations. For commands, where no data is to be transferred, such as a reset, the command is sent as the option payload of a STP Nop operation. After the command has been executed on the Target, it sends the response as a STP Put operation into the persistent memory region established during connection setup [48].

4.5. Summary

STP differs from traditional protocols such as TCP in many ways. Whereas TCP uses a sophisticated scheme that attempts to estimate available network bandwidth to maximize the congestion window, STP uses a simple RTS/CTS followed by Data packets that constitute a Block. The extra round-trip is required for reserving buffers on the receiver. After the buffer has been setup, the process of actually receiving the data becomes extremely simple, since the receiver only needs to verify that the packet is valid and copy data to the correct location. The amount of state information is small enough, that the receive path can be implemented in hardware, providing zero-copy receives. Many of the header fields (Bufx, Mx etc.) are only significant on the host that sets them and are echoed back by the other end. This allows implementations to choose their values in a way that is optimal for their performance.

Due to the extra round-trip involved in reserving the buffer on the receiver, the latency of STP is quite poor. This is offset by the bandwidth benefit gained, and also the reduction in host CPU utilization.

For low latency applications, STP offers a second mode of data transfer, persistent-memory Get, Put and FetchOp operations, where the buffers on the receiver are reserved only once and subsequent operations reuse the area. To gain the best possible

benefit from persistent-memory operations, they are usually implemented in a manner which lets applications bypass the operating system completely. This avoids the context switches and system calls involved with traditional kernel-based networking, although OS bypass libraries like libst are significantly more complicated to use than sockets.

Another difference is in the semantics provided by the protocols. TCP provides a reliable two-way byte stream. STP is record-oriented, both the transmitter and the receiver must agree on the amount of data being sent in a single Transfer.

5. STP IMPLEMENTATION IN LINUX

In this chapter, we will describe STP at the implementation level. This was done by studying the Linux STP implementation from SGI [58]. This chapter is structured as follows. First, we will give an overview of the STP code for Linux. After that, we will describe how the STP is accelerated in hardware using the Alteon Tigon II Gigabit Ethernet chip. Finally, we will provide experimental results of the performance of STP and how it compares to the zero-copy TCP stack provided in Linux 2.4.

5.1. STP implementation in Linux

To bring STP to commodity hardware SGI has released a Linux port of the code [58] under the GNU General Public License. The current version of the code, 0.33, runs on Linux 2.4 kernels and supports hardware acceleration on Alteon Tigon II [59] based GigE NICs.

Figure 13 depicts the basic structure of the STP implementation of Linux. The implementation is divided into four layers, the Upper Layer Protocols (ULP), the STP core component, the STP virtual device (stvd) and network device drivers with varying levels of STP support. The ULPs are the interfaces through which STP is used, such as sockets and SCSI. The STP core component implements the buffer management and various finite-state machines required for the operation of the protocol.

To enable hardware acceleration on NICs, the STP core component advertises a `stp_netdevice` interface. The functionality required for the network driver includes tiling Blocks that are being transmitted into individual STUs suitable for the network being used, and encapsulating them inside Ethernet or IP packets. Similarly, it must handle incoming STP packets and notify the STP core when an entire Block or a non-Data operation has been received.

The `stvd` module provides a software-based implementation of this interface making it possible to use STP with unmodified network drivers. Network drivers supporting hardware offload of the protocol can use `stvd` for the methods that have not been implemented.

5.2. Hardware acceleration

One of the main design principles of STP was to make hardware acceleration of the protocol as easy as possible. The advantage STP offers compared to traditional protocols is that it is not necessary to implement the entire protocol in hardware, only the most common operation: receiving data packets without errors. The basic idea is to notify the NIC about each buffer when they are requested by `Request_To_Send`. The NIC can then analyze incoming data packets and place the payload into the correct location inside the user buffer. This completely removes the need to perform the intermediate copy from kernel to user buffers.

The Linux implementation of STP supports hardware acceleration on the Alteon Tigon II GigE chipset. In this section, we describe how the device driver and firmware of the Tigon II was modified to support hardware acceleration of STP.

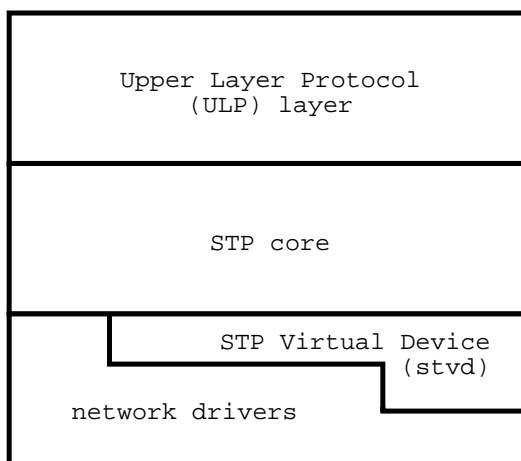


Figure 13. Structure of the Linux STP stack.

5.2.1. Alteon Tigon II

The Alteon Tigon II is one of the most popular GigE chipsets available on the market. It is used in Ethernet adapters from several different vendors such as Alteon AceNIC, 3com 985B and NetGear GA620.

The Tigon II includes two programmable 88MHz RISC processors, that have an instruction set based on the popular MIPS R4000 processor, with some modifications made to be better suited for an embedded network processor. Both CPUs have some private scratchpad memory available, 16k for the first CPU and 8k for the second. The chip supports up to 8 MB of external memory, which is used to store the firmware and the transmit and receive buffers. Commercially available NICs using the Tigon-II generally include 512 kB or 1 MB of external memory, with approximately 250 kB used by the firmware.

What made Tigon II especially attractive for this project was that the complete source code for the firmware could be freely downloaded from Alteon. This makes it possible for end-users to implement new functionality such as hardware acceleration of STP.

5.2.2. Modifications to the driver and firmware

As we described in the previous chapter, the basic design principle of STP hardware acceleration is to have information available to the NIC about the allowed range of STP Buffer Indexes (Bufx) for each incoming Block, and which physical memory pages the Bufxs correspond to. This makes hardware acceleration relatively simple to implement: the Linux implementation required less than 500 lines of changes to the Ethernet driver and about 2000 lines of code in the firmware. The acceleration code could also be cleanly separated from the standard driver and firmware.

Due to the limited amount of memory available on the NIC and the lack of dynamic memory allocation, the data structures required for transferring data to the correct buffer need to be kept as simple and small as possible. Still, the lookup process

must be fast, as potentially tens of thousands of packets have to be processed every second. In a network stack implemented in a kernel, a hash table is often an appropriate choice for storing the data, as lookups can be performed rapidly at the cost of using more memory. For the firmware implementation of STP this was not an option due to the limitations imposed by the hardware environment, and an alternative approach, modeled after one suggested in the STP standard, was used.

The basic principle of validating Data STUs is shown in Figure 14. The approach uses the 16-bit Mx field in the header as an index to an array containing a validation table for each Block. Before the STP stack sends a `Clear_To_Send` to the remote host it creates a validation table entry on the NIC, which contains the connection-specific destination Port and Key and the allowed Bufx range for that Block. The firmware also uses some additional fields internally to track out of order STUs inside the Block.

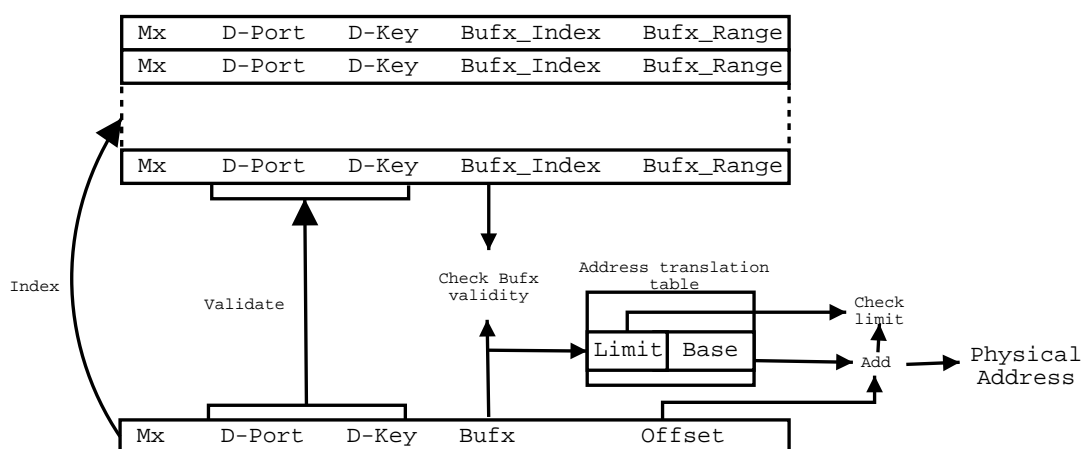


Figure 14. Validation of incoming Data STUs.

When the STP accelerated driver is initialized, it calls an `egast_init_st()` function, which allocates a physically contiguous block of memory to store Bufx information in host memory. The driver then notifies the firmware of the location of the buffer and asks it to initialize the STP acceleration code in the firmware.

When the interface is activated, the driver registers itself to the STP code through the `stp_netdevice` interface and provides hardware-assisted methods for Bufx and Mx management. All other methods are provided by the `stvd` package.

When a Data packet is received, the firmware fetches the validation entry corresponding to the Mx field and compares the other fields in the packet to the ones specified in the validation table. If they all match and the data to be received does not cross a buffer boundary, the firmware consults a small private Bufx cache that maps Bufx numbers into physical addresses. If the Bufx entry is not found from the cache, it is fetched from host memory and placed in the cache.

The firmware now knows where the data should be placed, so it can initiate a DMA transfer. If the STU was the last one for the Block (Interrupt bit set) and all STUs for the Block have been received, it is also passed on to the host to notify that the Block has been completely received.

5.3. Experimental results

In this section, we will provide experimental results of various aspects of STP performance. This section is structured as follows. First we will describe the problems associated with benchmarking high-speed networks. After that, we will describe the different tests that were performed and their results.

The tests that were performed were:

- Effect of STU size,
- Effect of Transfer size,
- Comparison between STP and TCP performance,
- Interrupt rates of STP and TCP,
- Effect of latency, and
- Effect of lost packets.

The tests were performed between a dual 500 MHz Pentium III transmitting to a dual 450 MHz Pentium II on a dedicated switched GigE network. A complete description of the test setup and how the tests were run as well as raw data obtained from the measurements can be found in Appendices 1 and 2.

5.3.1. Benchmark methods

Several network benchmark tools are available for UNIX systems, starting from simple test tools like *ttcp* to featureful packages such as *netperf* [60]. However, none of the benchmark programs available were suitable for the purposes of this thesis. The simple test tools, such as *ttcp*, were written in an TCP-centric way and adding new protocols or interfaces like `sendfile()` would have required almost a complete rewrite. *Netperf* on the other hand had the necessary infrastructure, but is too complicated for quickly implementing new kinds of tests.

For these reasons a new benchmark program, *yantt*[61], was written for the purposes of this thesis. It is written in a completely modular fashion, which makes adding features like the use of `libst` instead of sockets easy. The most important features of *yantt* are:

- Support for multiple protocols (currently TCP, UDP and STP),
- transmitting data from different kinds of sources (memory buffer or from a file using `sendfile()`, `read()+write()` or `mmap()+write()`),
- processing received data in different ways (no processing, touching the data, calculating checksum), and
- setting socket options like socket buffer size.

One of the problems with benchmarking networks is obtaining accurate results for CPU usage. UNIX offers a `getrusage()` system call which returns a structure containing the total time the process has spent in user and kernel contexts. However, this will only report the time spent in the application and the socket layer, as the rest of protocol processing is triggered by interrupts and thus the resources used are not accountable to any individual process.

To get around this problem, `yant` uses a different approach, which was inspired heavily by the method used by `netperf`. Instead of trying to measure the CPU use of the process directly, separate background threads running on every CPU of the system at a low priority measure the amount of CPU available to them. The overall resource use of networking can be deduced by comparing the amount of CPU the threads receive compared to an idle system.

The numbers reported by `yant` are percentages of the total amount of CPU, thus on a dual-processor machine 100% utilization means both CPUs are being fully utilized. To determine the load of individual CPUs, the benchmark threads would have to be bound to specific CPUs, which is not supported by the standard Linux kernel, although patches [63] are available to make this possible.

A second problem in measuring high-speed network performance is caused by the fact that nearly all of the hardware used is running at its limit. This makes analyzing the capabilities of protocols difficult, as it is often difficult to pinpoint the exact location of the bottleneck. The problem may reside in the protocol stack, the device driver, the I/O bus or the NIC. For example, setting a PCI register that disables the use of memory write and invalidate operations increases network bandwidth by nearly 10% on the Alteon Tigon II¹.

A related problem is different versions of the NIC driver and firmware being used by the standard and accelerated firmwares. Initial attempts in updating the STP-accelerated `acenic` driver to version 0.81 resulted in a performance loss of approximately 15%, which was caused by hardware checksum calculation being enabled in the firmware for zero-copy TCP and the NIC transmit ring being moved to host memory (previously it was stored on the NIC). Both of these changes in the driver were introduced to increase TCP performance, but had a negative effect on STP.

As a result, the performance of TCP and STP cannot be directly be compared based on the merits of the protocols themselves, as the device driver has a large effect on performance. For both protocols, the driver version, which gave the best performance was used. For TCP this was the v0.81 driver included with the standard Linux kernel. For STP the STP-enhanced derivative of `acenic.c` v0.49 was used.

5.3.2. *Effect of STU size*

In this test, we measure the performance of STP with different STU sizes. The values used were 512, 1024, 2048 and 4096 bytes, which are the values possible on x86 platforms. The Transfer size used for all tests was 512k, with 32 STUs per Block. The results of the test are summarized in Figure 15. For reference, also the number of packets per second (pps) transmitted is also shown.

¹It was disabled by accident in the Alteon driver of the first Linux zero-copy patches, which at first made it seem like the zero-copy patches actually reduced performance.

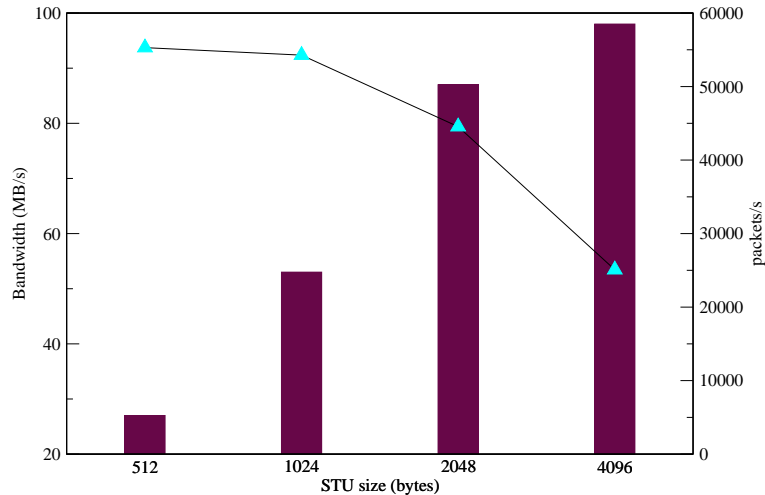


Figure 15. Effect of STU size on STP performance.

As can be seen from the results, the performance goes up almost linearly for STU sizes of 512, 1024 and 2048 bytes, with the packet rate being roughly 55000 pps with 512 and 1024 byte STUs. This is caused by limitations in the Tigon II DMA hardware and firmware, which introduce a $5\mu s$ overhead for initiating a DMA [62]. In practice this is seen as a limit on the number of packets that can be delivered per second.

With 4096 byte STUs the performance is 98 MB/s with neither the receiver nor the transmitter CPU being fully utilized. The most likely bottleneck is the 32-bit/33MHz PCI bus on the receiver, which has a theoretical peak bandwidth of just slightly over 125 MB/s. Another possibility is the NIC itself, although this seems unlikely as the board has been reported [34] to run TCP at 117 MB/s when used with a 64-bit/66 MHz PCI bus.

5.3.3. Effect of Transfer size

In this experiment, we measured the performance of STP with different Transfer sizes from 4kB to 512kB. The test was done with both standard and jumbo frames, with a maximum of 32 STUs per Block. The results of this test are shown in Figure 16.

For small Transfer sizes, the performance is largely dominated by the link latency, as a RTS/CTS pair is required before any data can be transferred. Performance is therefore quite poor, around 10 MB/s with 4 kB Transfers. For larger Transfers, the cost for the extra round-trip associated with each Block becomes negligible.

CPU utilization is very low (6-12% out of both CPUs) on the receiver and slightly decreases with both frame sizes as the Transfer size increases. The reason for this is hardware acceleration, which means only the last STU of each Block is processed by the host. For the same reason, there is almost no difference in CPU utilization between 1024 and 4096 byte STUs; the number of packets per second has no effect, only the number of Blocks per second.

CPU utilization on the transmitter is slightly greater than on the receiver and rises as the Transfer size increases (with very small Transfers the cost of setting up the Transfer is also visible). This is caused by the fact, that transmits are not accelerated

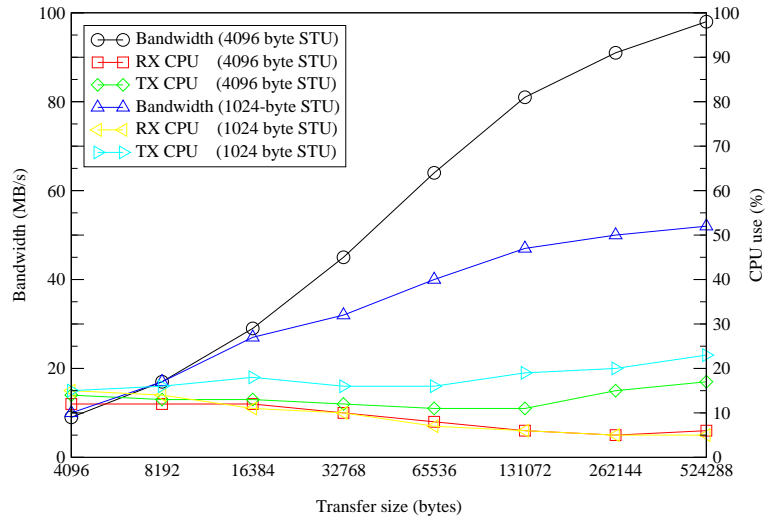


Figure 16. Effect of Transfer size on STP performance.

in any way. Transmit interrupts are generated and processed in the same way they are for traditional protocols. With 1024 byte STUs, CPU utilization is noticeably higher due to the higher number of packets.

5.3.4. Comparison between STP and TCP

In this experiment, we compare the performance of STP and TCP. The tests were done by using the options that gave the best performance for both protocols, including different Ethernet drivers (0.49-stp and 0.81), 512k blocks for STP and 512k socket buffers for TCP. The tests were performed with both standard and jumbo frames. This allowed TCP to use larger packets (9kB), because the use of memory pages as STP buffers limits STUs to 4 kB on the x86 architecture.

Zero-copy transmits with TCP were obtained by the use of `sendfile()`. The file that was sent was a 50 MB sparse file² to ensure that disk performance did not affect the results in any way. Zero-copy receives were also simulated by the use of the `MSG_TRUNC` flag in the `recv()` system call. The `MSG_TRUNC` flag makes the kernel skip the copy from kernel buffers to the user buffer when receiving data. Naturally the data is not available to the application, but this feature is still extremely useful for providing a basis for comparisons.

The results of the tests are shown in Figures 17 and 18 for standard and jumbo frames.

With standard frames, TCP performance is slightly better than STP performance, although the receiver is overloaded (one CPU is completely utilized). This is caused by the limitation imposed by the DMA engine of the NIC, which limits the maximum rate to about 55000 pps, as was shown in 5.3.2. Since TCP is able to use 40% larger payloads, this means the theoretical maximum bandwidth is also 40% higher. This is nearly achieved by using `sendfile()` and `MSG_TRUNC`.

²A file consisting of zeros that only uses one disk block

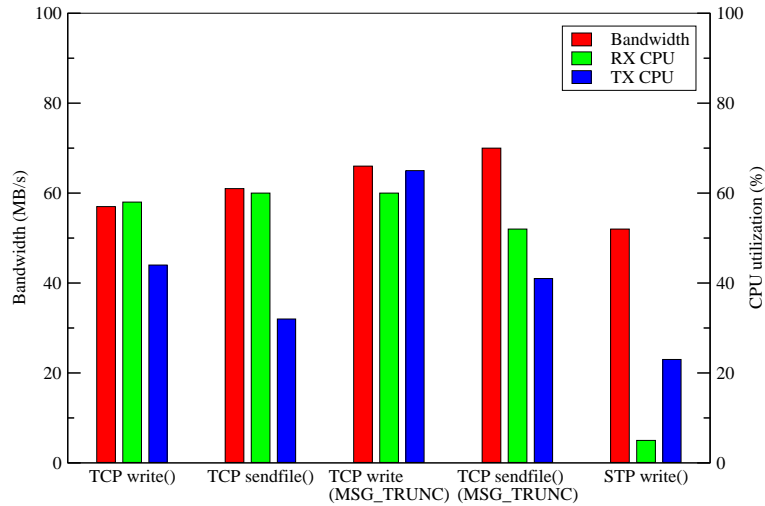


Figure 17. Comparison of STP and TCP (MTU 1500)

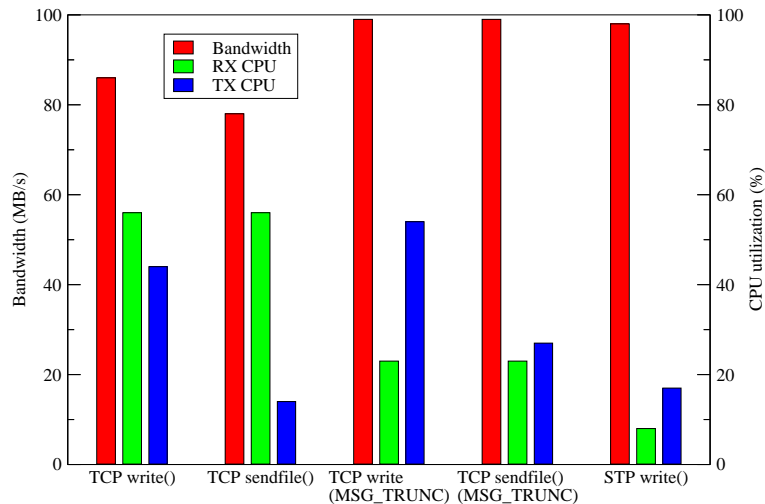


Figure 18. Comparison of STP and TCP (MTU 9000)

When `sendfile()` is used with TCP, the CPU utilization on the transmitter drops slightly. It is interesting to note that `MSG_TRUNC` does not have a significant effect on CPU utilization; the load is caused almost completely by interrupt and protocol processing.

On the transmitter, there is no significant difference between CPU utilization between `sendfile()` and STP when the difference in transfer speed is taken into account. The difference on the receiver, however, is enormous. The hardware acceleration of STP removes the overhead caused by interrupt processing and memory copies almost completely.

When jumbo frames are used, the DMA engine of the NIC no longer is a bottleneck. Subsequently, the bottleneck with TCP becomes the receiver memory bandwidth, which limits the the bandwidth to 86 MB/s. The sender, which is a faster machine, is also heavily loaded. Zero-copy transmit drops CPU utilization from 56% (one CPU completely utilized) down to 14%. There is also a small reduction in bandwidth. When the copy on the receiver is skipped, performance jumps up to 99 MB/s, which

appears to be the physical limit of the PCI bus on the receiver. STP has a similar level of performance, yet the data is copied to the user buffer and the CPU utilization is again extremely low (6%).

It should be noted, that in the test shown above the data was not accessed at all on the receiver. This makes the results for STP better than they would be in a real-life application, as the data is not in the CPU cache as a result of the kernel having to perform the data copy to user buffers. For this reason a second test was performed, where the test program accessed every byte that was read (–touch option in yantt). The results of this test are summarized in Figure 19.

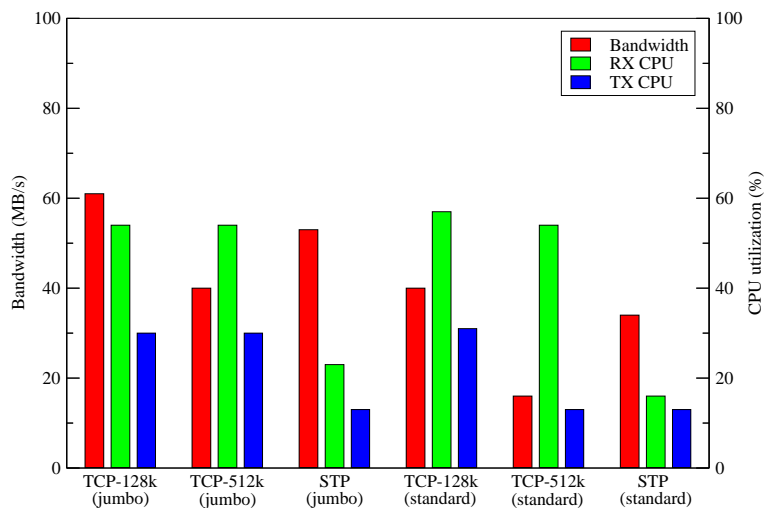


Figure 19. Comparison of STP and TCP when receiver touches the data.

For TCP, two results were measured: reading data 128 kB and 512 kB at a time. This was necessary due to the fact that the amount read (and touched) affects the performance significantly. This difference is caused by the CPU cache, which is 512 kB on the Pentium II used for the receiver. When the data is read and accessed 128 kB at a time, it fits completely inside the L2 cache of the CPU. For 512 kB reads, the data is only partially in the cache when it is accessed, making it necessary to reload it from normal memory.

When jumbo frames are used, the performance of STP drops nearly 50% to 53 MB/s when the receiver touches the data, whereas TCP performance drops only 30% to 61 MB/s for 128 kB reads, due to the data being in the CPU cache. For 512 kB reads, where the data does not fit entirely into the cache, TCP performance drops to 40 MB/s. In both cases, the TCP receiver is completely saturated.

With standard frames, the situation is somewhat different, as the receiver is so overloaded due to interrupt load when TCP is used, that performance drops 60% to only 16 MB/s when the data does not fit in the cache. When it does, performance is significantly better, at 40 MB/s, while STP performance is slightly lower, at 34 MB/s.

In both cases, the CPU use of STP remains extremely low, with plenty of CPU available for processing the data.

5.3.5. Interrupt rate

In this experiment, we measure what effect hardware acceleration and interrupt coalescing have on the interrupt rate and performance. The results of the test are shown in Figure 20.

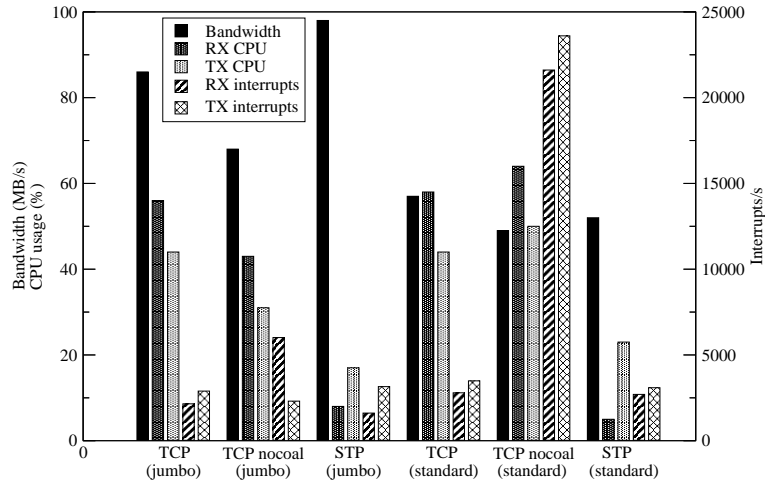


Figure 20. Interrupt rates of STP and TCP.

When jumbo frames are used with TCP, interrupt coalescing reduces the interrupt rate by about 65% from 6000 to 2150 interrupts/s. Performance also improves about 20%. Hardware acceleration of STP further reduces this to 1600 interrupts/sec.

The greatest benefit of interrupt coalescing is seen with standard frames, where the interrupt rate drops almost 90% when coalescing or STP acceleration is used. The performance gain is similar to that obtained with jumbo frames, but the high interrupt rate (22000/s) itself does not cause any major performance problems, although CPU utilization is somewhat higher. If the data on the receiver was processed in any way, performance would suffer, even more so than it did in 5.3.4.

5.3.6. Effect of latency

In this experiment, we measure how latency affects the performance of STP. Additional latency was simulated by using NIST Net version 2.0.10, a freely available network simulator for Linux. The network simulation software was run on a separate machine, and all traffic between the end systems was routed through that machine. Adding the third machine effectively halved the network capacity, as the network adapters on the network simulator machine were in the same 32-bit/33 MHz PCI bus.

The test was run using two different STP Transfer sizes, 128k and 512k. For comparison, numbers for TCP using window sizes of 128k and 512k have also been included. The results of this experiment are shown in Figure 21, where the latency is shown on a logarithmic scale. The amount of latency shown on the x -axis is the additional round-trip latency introduced by the network simulator software in addition to the natural round-trip latency of the link, which was measured to be approximately $300 \mu s$ using

the ping utility. The delay was split evenly in both directions, i.e. a delay of 10 ms means that packets from both directions were delayed by 5 ms.

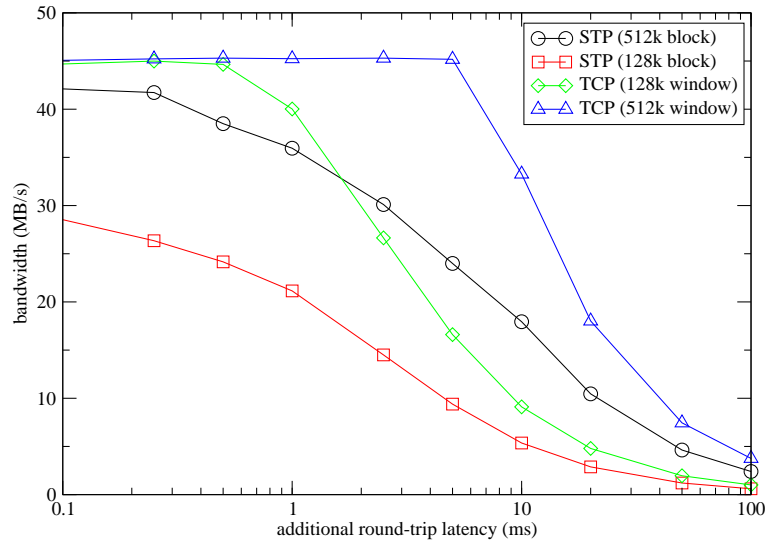


Figure 21. Effect of latency on STP and TCP performance.

As can be seen from the graph, the performance of both protocols decreases as latency increases. However, TCP is able to maintain the maximum speed until the bandwidth-delay product becomes too large for the transfer window, whereas the performance of STP quickly starts decreasing as latency increases. When larger Transfer sizes are used, the situation is improved somewhat, as the number of extra round-trips required for setting up Transfers is reduced.

Another interesting result of this experiment is the performances of the protocols when the end-systems are not the bottleneck. TCP is able to utilize the available bandwidth fully, whereas STP is nearly 5% slower when no extra latency is introduced. This differs from the situation, where the hosts were the bottleneck. Although STP is not able to use the network as efficiently as TCP, better performance was obtained simply by utilizing the hardware resources on the hosts more efficiently.

5.3.7. Effect of lost packets

In this experiment, we measure how STP is able to cope with the loss of STUs. Packets were only dropped in the direction the data was flowing. Although this is not a completely realistic situation (packet loss would probably occur in both directions on a real network) it models a situation where the network is overloaded in one direction and thus packets have to be dropped by a router. The results of this experiment are shown in Figure 22.

Like in the previous experiment, the performance of STP quickly deteriorates when the network is not “perfect”. Even a 0.001% rate of dropped packets is enough to drop the performance of STP by 50%. This is due to the basic nature of STP, if a single packet is lost, the entire Block must be discarded. Traditional protocols that use a transfer window can simply retransmit the packet that was lost. In the case of TCP, a 0.1% packet loss reduces performance by only 10%. STP fails to function at all

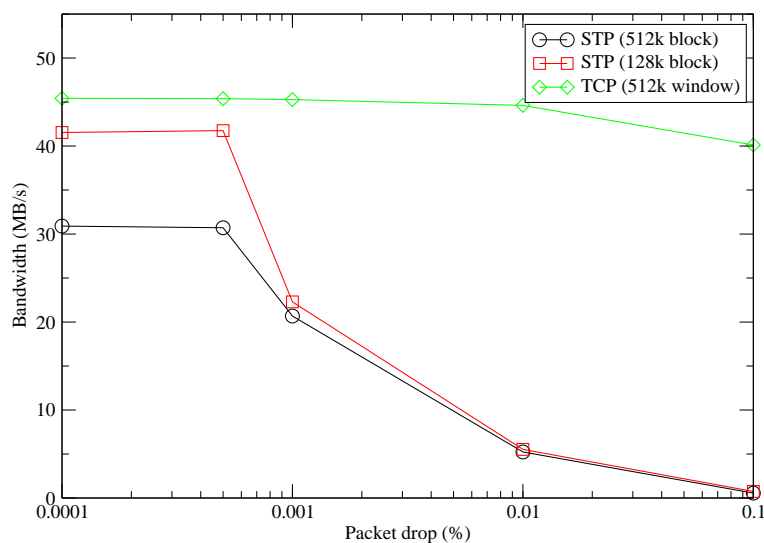


Figure 22. Effect of packet loss on STP and TCP performance.

with similar packet loss. With smaller Blocks, the situation might have been slightly better, but as was shown in previous experiments, large Blocks are required for good performance.

The poor performance of STP can partially be explained by the long timeout values used in the Linux implementation, which are 5 seconds for Data operations and 2 seconds for control operations. The STP standard leaves the definition of such timeouts to the implementation.

5.4. Summary

The results obtained from these experiments indicate that STP is at its best when the network speed is higher than the host hardware can handle. On Gigabit Ethernet using Tigon II adapters this required jumbo frames, due to a hardware limitation of the NIC, which limits packet rate to around 55000 pps. This limited STP performance with standard 1500 byte Ethernet frames, because STP is only able to use 1024 bytes as the payload.

Hardware acceleration gives a significant benefit on the receiver, where the CPU utilization is under 10% (of one CPU). In contrast, TCP running at a similar speed consumes all available CPU on the receiver, which limits performance on the receiver significantly.

The bottlenecks for TCP and STP are quite different. The bottlenecks for TCP are caused by memory copies and interrupt load. Interrupt coalescing, jumbo frames and `sendfile()` offer a satisfactory solution when used together, although the receiver memory bandwidth remains a problem. However, TCP is able to efficiently utilize gigabit networks as long as the hosts do not limit performance.

STP, on the other hand, is able to use the limited hardware resources available very efficiently by doing zero-copy on both the sender and the receiver. The interrupt rate also remains low on the receiver due to hardware acceleration. The biggest bottleneck for STP is utilizing the *network* fully, especially when it is not “perfect”, i.e. exhibits

packet loss or has a latency of more than 1 ms. This, combined with a relatively low amount of security, make STP only applicable on dedicated, high-speed local area networks like Gigabit Ethernet and GSN.

6. DISCUSSION

In this chapter, we will discuss the benefits provided by STP and how the results are applicable to other protocols such as TCP. Similarly, we will outline some of the drawbacks and weaknesses and describe briefly how they could be solved in the future. After pointing out the benefits and drawbacks, we will analyze the results of this thesis. Finally, we will also describe possibilities for future research in this area.

6.1. Benefits

STP provides a number of benefits compared to traditional protocols such as TCP. These benefits were already described and measured in the earlier chapters, but we will summarize them here. These benefits are:

- hardware acceleration,
- zero-copy transmit and receive,
- reducing the number of interrupts, and
- OS bypass.

6.1.1. Hardware acceleration

Offloading the “fast path” of protocol processing is very tempting, as was evidenced by the results shown in the previous chapter. For most protocols, however, implementing hardware acceleration is challenging, defining an appropriate layer for dividing the software and hardware components is very difficult. Moving the entire protocol into hardware is an option that would make this easier, but for economic (and often technical) reasons it is seldom feasible.

Hardware acceleration as done by STP follows the golden design principle of optimizing for the common case of data packets arriving in order without errors. The entire protocol is in fact *designed* around that very principle, even at the cost of making error recovery much more expensive. Extra effort (involving an entire extra round-trip before each Block) is taken by the protocol simply to make it possible for the NIC to be able to just take a quick look at the header and know instantly where the data belongs in host memory. With traditional protocols this is not possible.

For TCP, finding such clear boundaries is not as easy. However, there are some areas where it is feasible, such as the approaches described in Chapter 2: checksum calculation, IP fragmentation and TCP segmentation.

The benefits of hardware acceleration in STP are not due to the processing of incoming packets in hardware. Instead, the largest benefits are the result of hardware acceleration: namely zero-copy receive and the reduction of the interrupt rate.

6.1.2. Zero-copy transmit and receive

Avoiding unnecessary copies in the network stack reduces CPU utilization significantly, as shown by the results in the previous chapters. Although zero-copy TCP is possible using the methods described in Chapter 2, it can only be done without problems on the transmitter using a `sendfile()` type interface.

With STP, the overhead imposed by pinning the pages to be transmitted is not as large an issue as it is with TCP, because the `write()` system call will only return after the Transfer has completed, preventing the application from reusing the buffer immediately. To prevent the pages containing data that is to be transmitted from being paged to disk, the pages still need to be locked in memory, which introduces some overhead, but the worst-case scenario where the application immediately reuses the buffer is not possible with STP.

On the receiver, the hardware acceleration code dramatically reduces the CPU utilization, since the network board can directly copy data into the final destination in memory.

A negative side-effect of zero-copy on the receiver is that the data is not in the CPU cache when the application first accesses the data after the data has been received, which reduces performance somewhat. Nevertheless, even without the benefit of the CPU cache, the CPU utilization of STP is significantly lower with a similar performance level.

6.1.3. Reducing the number of interrupts

The second benefit of hardware acceleration in addition to zero-copy is the reduction in interrupt rate. When STP is used, the host is only interrupted when a complete Block has been received. For OS bypass, the applications poll a private memory region for new incoming packets, and thus it is unnecessary to generate interrupts

The benefit of STP hardware acceleration compared to interrupt coalescing is not very big (a 35% reduction in the interrupt rate when the difference in the bandwidth of STP and TCP is taken into account). Unlike coalescing, however, STP hardware acceleration does not have a latency tradeoff¹.

A similar effect to STP hardware acceleration can be obtained with TCP/IP by using the hardware assisted fragmentation and reassembly approach presented in Chapter 2, although this can only be done for up to 64 kB at a time due to the limitations of IP.

6.1.4. OS bypass

Although not covered by the experimental portion of this thesis, OS bypass is one of the most important aspects of STP. It provides applications direct, protected access to the NIC. This removes the overhead incurred by the OS, which includes system call latency and context switches. By reserving a persistent memory region, it is not

¹However, STP latency when long messages are used is still worse than TCP latency due to the extra round-trip for establishing Transfers. Short messages do not have this weakness.

necessary to reserve a new buffer on the receiver for each Block like it is, when the long message Read and Write operations are used.

OS bypass requires the ability to virtualize the NIC, so that each application is able to place outgoing packets into a private transmit queue, and have packets placed into their receive queue by the NIC. The principle behind this is similar to the one used for acceleration of long message operations. Validation entries based on the Mx field are entered for each persistent memory region and Data packets are copied directly by the NIC to the correct location. In addition, control operations and Data STUs with the Silent bit off need to be passed on to the application for further processing.

The downside of OS bypass is that it makes applications more complicated, as they are required to manage memory and network resources. To avoid this, higher level APIs such as MPI can be layered on top of libst.

The functionality provided by OS bypass is currently not possible with TCP/IP. TCP RDMA [51, 52] or Internet VI [56] may provide a solution to this problem, but it is too early to say whether they will be viable solutions.

6.2. Drawbacks and weaknesses

In this section, we will discuss the drawbacks and weaknesses of STP and outline briefly ways to minimize or eliminate them in the future.

The drawbacks include:

- Requirement of firmware modifications and programmable NICs,
- Relying on hardware, and
- Performance on non-reliable networks.

6.2.1. Requirement of firmware modifications

The greatest problem limiting the use of the technologies presented in this thesis is the availability of programmable NICs. To make GigE a commodity technology, the price of adapters has had to drop significantly. This naturally has reduced profit margins for NIC vendors, and “unnecessary” features, such as programmability, are the first things to be dropped to make NICs cheaper to manufacture.

For standard applications, this is completely acceptable, as perfectly good performance can still be obtained using traditional protocols and non-programmable NICs at the cost of highly loaded end systems. To fully utilize current high-speed networks to obtain high-end performance (high bandwidth and low latency) on *current* hardware, the techniques used in protocols like STP are still required.

For the purposes of this thesis, the aging Alteon Tigon-II could be used, but boards using the Tigon II chips are rapidly vanishing from the market, replaced by cheaper, faster boards, none of which were end-user programmable in July 2001. The successor to the Tigon II is marketed under the name of Broadcom BCM 5700 and used in the 3com 996 NIC. The chip remains programmable, although no low-level informa-

tion has been published. It is also unclear, whether there is enough memory for any meaningful hardware acceleration.

In some cases it may even be possible that NICs based on generic programmable CPUs introduce enough latency that any gains made possible by programmable NICs are negligible compared to the speed and cost benefits provided by their fully hardware-based counterparts.

6.2.2. Relying on hardware

In addition to the availability of programmable hardware being a problem, moving even the most trivial tasks such as protocol checksumming to the NIC introduces a risk. When checksums are offloaded, data integrity is not necessarily guaranteed between host memory and the NIC. While the expected behaviour is to transmit the data given by the application reliably to the remote host the actual behaviour might actually be corrupting a few bytes every million packets when copying data to NIC memory and calculating the checksum based on the new corrupted data [31].

For more complicated operations, such as protocol offloading the problem is even worse, since the complexity of the implementation increases significantly; instead of matching TCP and UDP packets, calculating the checksum and placing it in the correct location, a protocol implementation requires a large amount of state information. The NIC must also be able to communicate with the operating system about matters like connection and buffer management. Embedded systems such as NIC firmwares are also more complicated to program and diagnosing and fixing problems is often extremely difficult.

STP overcomes this problem by keeping the portion of the protocol that is implemented in hardware as simple as possible. The information that needs to be maintained about each incoming Block is very small, only the Block and connection identifiers and the buffers that can be written to.

6.2.3. Performance on non-reliable networks

The third problem with STP is coping with packet loss and a latency of more than a few hundred μ s. If a packet is lost, the entire Block must be discarded. Also for every Block that is sent, an extra round-trip must be performed to reserve a buffer on the remote host. These properties of STP make performance on high-latency, lossful networks abysmal. The extra round-trip required to reserve a buffer for the next Block in non-persistent-memory data transfer operations also introduces a significant amount of latency.

Both of these were inherent design decisions in the protocol, and come from the original goal of designing a protocol that could utilize GSN fully. GSN offers error correction on the hardware level, making it unnecessary to perform error correction at the protocol level. Similarly, the maximum distance of GSN is only a couple hundred meters at best, which means latency will always be very low. For this kind of environment, the compromises STP makes to be able to run fast are completely justified.

The best solution to this problem is to run protocols like STP on isolated, dedicated, networks using (mostly) reliable network technologies like switched Gigabit Ethernet, GSN and Infiniband. This is also required for security, since STP offers protection only from misbehaving implementations, not malicious hackers. Additionally, for latency-critical applications the non-persistent-memory operations should be avoided, and OS bypass should be used instead.

6.3. Results obtained in this thesis

There is no single solution for high-performance networking in the future. Truly low-latency networking requires application-specific protocols that bypass the OS completely. Wide area networking requires a standardized protocol such as TCP that is able to adapt itself to every kind of network. High-speed networking requires the ability to utilize all of the available hardware fully, not only the network. Combining all these requirements into a single solution is very challenging.

On networks, where reliability is guaranteed on the hardware level, protocols such as STP can offer significant increases to network performance. STP addresses the two major problems seen today with TCP, the high amount of interrupts and data copies, by making it simple to implement the most common operations in hardware.

On GigE networks, the benefit was measured to be an increase of over 10% in bandwidth, and more importantly, a 90% decrease in CPU utilization on the receiver when hardware acceleration is utilized. Even then, the performance was limited by the I/O-bus of the end systems, not the operating system nor the network.

Although hardware limitations kept transfer speeds slightly under 100 MB/s, there is no reason to believe that STP performance would not scale to 10 Gbps and beyond.

6.4. Future directions

In this section, we will briefly discuss some of the most important challenges of future research in high-speed networking. The areas we will describe are:

- STP,
- the Linux STP implementation,
- improvements to TCP,
- hardware-accelerated TCP, and
- future network technologies.

6.4.1. STP

Even though STP has been ratified as an ANSI standard, work is still needed to make the protocol truly general purpose. The libst and SST standards are still being drafted, although commercial implementations of both are already available.

Libst allows the full potential of STP to be realized, but programming for such a low-level API requires significant changes to the way applications are written. It is therefore necessary to layer higher-level APIs on top of libst that allow STP to be used efficiently yet maintain a high level of abstraction. One possibility for such an API is MPI, which has already been implemented on top of libst for IRIX.

6.4.2. Linux STP implementation

Although the basic functionality required for a high-performance STP implementation are already available, there are still several areas where the implementation could be improved.

For low-latency networking, the OS bypass functionality of STP should be implemented. Partial support for this already exists in the kernel and the NIC firmware, but further work is needed to make it fully functional.

In the current version of STP, the firmware implements a user-space transmit ring, which is mapped by the driver into userspace when a `mmap()` call is performed on the socket. The corresponding receive ring has not been implemented. This makes sending Get and Put operations possible, but not processing them. Data operations can be handled in the same way as when using the socket API; when a persistent memory region is allocated a validation entry is uploaded to the NIC.

A partial implementation of the libst 1.0 library, based on the IRIX version, has been completed, but a complete rewrite will be required to implement libst 2.0.

The CPU usage on the transmitter could be further reduced by moving the tiling (splitting Blocks into individual STUs) to the NIC. This method might reduce the interrupt rate on the transmitter. For GigE, this is probably not worthwhile as the CPU utilization is already quite low, and interrupt coalescing gives most of the benefits (the coalescing settings can be much more aggressive than for receiving, as there is no latency tradeoff).

6.4.3. Improvements to TCP

An aspect of high-speed networking, which has been nearly completely neglected in this thesis, is performance on wide-area networks such as the Internet. In this case, the problem changes from being able to send packets as fast as possible from user buffer to user buffer into being able to accurately estimate the maximum rate data can be transferred without packets being dropped and recovering from any errors as quickly as possible.

TCP is able to accomplish this with reasonable success, although some improvements are still needed. The problem is, that all modifications made to it must be fully backwards-compatible with existing TCP implementations, work even on slow networks and maintain the “politeness” of TCP (which prevents the Internet from collapsing). These restrictions make any radical changes to the protocol impossible.

Current research in this area includes improvements to the TCP selective-acknowledgement option [64] and congestion management [65], choosing the correct size for socket buffers and the use of parallel streams [66].

6.4.4. Hardware accelerated TCP/IP

Despite the difficulties involved, hardware accelerated TCP/IP is still very attractive. The simplest form of acceleration, hardware checksums, are now a standard feature in every new NIC, although it is often unusable due to hardware problems.

For the majority of client-server applications, such as HTTP, NFS and CIFS, where a server connected to a high-speed network serves files to multiple clients on slower networks, hardware checksums and the zero-copy transmit they allow are in fact adequate.

TCP/IP implemented entirely in hardware will undoubtedly be tried again in the future. Past experience has shown it to be a bad idea, due to both economical and technical reasons. In addition to making protocol bugs, many of which are only apparent on fast networks, impossible to fix for anyone except the vendor, they also make upgrading to new protocols such as IPv6 impossible without upgrading the hardware. The extra messaging required between the OS and the NIC also often removes any benefit gained [67].

One possible direction which has not yet been investigated to the authors knowledge is basing interrupt coalescing decisions on the packet headers. A simple way to do this would be to use the Type of Service (TOS) field of IP packets. When a packet marked as “Maximize throughput” arrives, aggressive settings for minimizing interrupts can be used. Similarly, a packet marked as “Minimize delay” would cause an interrupt to be issued immediately. For packets where the field is not set or which do not carry IP traffic, a default setting can be used.

6.4.5. Future network technologies

Future network technologies will determine what kinds of enhancements to protocols really are necessary. The two most likely candidates for the networks that will be used are 10 Gigabit Ethernet and Infiniband.

The biggest problem with 10 GigE will be the enormous number of packets that have to be processed. STP hardware acceleration will definitely be very useful in reducing the load. If TCP is to be used, other measures have to be taken. More intelligent interrupt coalescing algorithms could be used, possibly combined with switching completely to a polled mode of operation when the load is high enough. Finding the optimal approach for operating systems and NICs for this will require further research.

Infiniband, on the other hand, already includes technologies similar to the ones provided by STP, zero-copy, OS bypass and hardware acceleration. STP could still be of use on Infiniband networks as a method of bridging Ethernet and Infiniband networks. This could be used for a variety of applications, where hosts on the Ethernet network want to communicate with hardware (host memory, storage, etc.) on a Infiniband network.

7. CONCLUSION

As network speeds increase in the future, it will be a great challenge to utilize the full potential provided by the networks. The two main problems currently seen are unnecessary memory copies and the high number of interrupts. In this thesis, we have studied several ways of optimizing the implementations of network protocols, and how a recently developed protocol, Scheduled Transfer Protocol (STP), addresses the currently seen problems. A quantitative analysis of the properties of STP was made by measuring the performance of the STP implementation in Linux 2.4 and comparing it to the performance of TCP on Gigabit Ethernet.

The basic design principle of STP is to perform as much work as as possible on the transmitter. This makes it relatively simple to implement the protocol partially on a programmable network interface card, making it possible to copy the data directly to the application buffers on the receiver, thus reducing the number of interrupts significantly.

STP offers two types of data transfer operations, non-persistent memory data transfer operations, which is used for bulk data transfers, and persistent memory data transfer operations, which are used for low-latency OS bypass communication. This makes it useful for a variety of applications.

The results obtained in this thesis show that, although STP offers only a modest 10-25% improvement in bandwidth on Gigabit Ethernet due to hardware limitations, the effect on CPU utilization is substantial, with a nearly 90% decrease on the receiver compared to TCP. On future networks such as 10 Gigabit Ethernet the performance gains will be even more dramatic as memory technology (and to a lesser extent CPU technology) have not improved as quickly in performance as network speeds.

To accomplish this STP must make several compromises. The protocol optimizes for the common case of error free data transmission, and utilizes the hardware on the end systems as fully as possible. At the same time, it does not use the network as efficiently as traditional protocols like TCP. Recovery from any kind of error such as a lost datagram is an extremely expensive operation, resulting in a substantial loss of performance for even a small packet drop rate. Similarly, STP also abandons the concept of a transfer window that automatically resizes to accommodate for the properties of the underlying network. This is acceptable on local area networks, but also means the performance drops significantly as latency or packet loss increases.

8. REFERENCES

- [1] Ousterhout, J.K. (1990) Why Aren't Operating Systems Getting Faster as Fast as Hardware?". Proc. 1990 Summer USENIX Conf., Anaheim, California, June 11-15, 1990.
- [2] Patterson D.A. & Hennessy J.L. (1994) Computer Organization and Design: The Hardware/software Interface. Morgan Kaufmann, San Mateo, California. 648 p.
- [3] Chase J., Gallatin A & Yocum K. (2000) End-System optimizations for High-Speed TCP. IEEE Communications, special issue on TCP Performance in Future Networking Environments, vol. 39 no. 4, 8 p.
- [4] Sapuntzakis C. & Romanow A. (2000, unpublished) The Case for RDMA. IETF Internet-Draft (draft-csapuntz-caserdma-00.txt)
- [5] Day J.D. & Zimmerman H. (1983) The OSI Reference Model. Proceedings of the IEEE, vol. 71, pp. 1334-1340.
- [6] Crowcroft J., Wakeman I, Wang Z. & Sirovica D. (1992) Is Layering Harmful? IEEE Network Magazine, vol. 6, no. 1, pp. 20-24
- [7] Clark D. & Tennenhouse D. (1990) Architectural Considerations for a New Generation of Protocols. SIGCOMM '90
- [8] Jacobson V. (1993) Some Design Issues for High-speed Networks. Networkshop '93, November 30, Melbourne, Australia, 21 p.
- [9] Hauben M. (20.11.2001) History of ARPANET. URL: <http://www.dei.isep.ipp.pt/docs/arpa.html>.
- [10] Internet Protocol (1981). Information Sciences Institute, RFC 791, 45 p.
- [11] Postel J. (1981) Transmission Control Protocol. RFC 793, 85 p.
- [12] Xpress Transport Protocol Specification: XTP Revision 4.0 (1995). XTP Forum, Santa Barbara, CA, USA, 111 p.
- [13] Jacobson V. (1992) Design Changes to the Kernel Network Architecture for 4.4BSD. 4.4BSD Class, May 5&7 1992, Berkeley, CA, 29 p.
- [14] Kay J. & Pasqale J. (1996) Profiling and Reducing Overheads in TCP/IP. IEEE/ACM Transactions on Networking, vol. 4., no. 6., pp. 817-828.
- [15] Clark D. D., Jacobson V., Romkey J.& Salwen H. (1989) An Analysis of TCP Processing Overhead. IEEE Communications Magazine, vol. 27, no. 6., pp. 23-29
- [16] Jacobson V., Braden R. & Borman D. (1992) TCP Extensions for High Performance. RFC 1323, 37 p.
- [17] Stevens W.R. (1994) TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley Publishing Company, Reading, Mass., 576 p.

- [18] ANSI NCITS 337-2000, Information Technology - Scheduled Transfer (ST) (2000). American National Standards Institute, Inc., Washington DC, 112 p.
- [19] Wright G.R. & Stevens W.R. (1995) TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley Publishing Company, Reading, Mass., 1174 p.
- [20] Mogul J.C. & Ramakrishnan K.K. (1997) Eliminating Receive Live-lock in an Interrupt-driven Kernel. ACM Transactions on Computer Systems, vol. 15, no. 3, pp. 217-252.
- [21] Salim J.H., Olsson R. & Kuznetsov A. (2001) Beyond Softnet. URL: <http://www.cyberus.ca/~hadi/usenix-paper.tgz>, 8 p.
- [22] Smith J. & Traw C. (1993) Giving Applications Access to Gb/s Networking. IEEE Network, vol. 7, no. 4, pp. 44-52
- [23] Partridge C. (1993) Gigabit Networking. Addison-Wesley Publishing Company, Reading, Mass., 396 p.
- [24] Gilfeather P. & Underwood T. (2001) Fragmentation and High Performance IP. In: Workshop on Communication Architecture for Clusters CAC '01, April 27, San Francisco, USA, 11 p.
- [25] Jacobson V. (1990) 4BSD Header Prediction. ACM Computer Communication Review, vol. 20, no. 1, pp. 13-15.
- [26] Miller D.S. (9.11.2001) Overview of Socket Hashing Changes, Plus New Patch. URL: <http://groups.yahoo.com/group/freebsd-hackers/message/2239>.
- [27] Extended Frame Sizes for Next Generation Ethernets (1998). Alteon Networks, San Jose, USA, 6 p.
- [28] Braden R.T., Borman D.A., Partridge C. (1988) Computing the Internet Checksum. RFC 1071, 24 p.
- [29] Clark D. (1982) Modularity and Efficiency in Protocol Implementation. RFC 817, 26 p.
- [30] Pratt I. & Fraser K. (2001) Arsenic: A User-Accessible Gigabit Ethernet Interface, In: IEEE INFOCOMM 2001, April 22-26 2001, Anchorage, Alaska, USA, 11 p.
- [31] Stone J. & Partridge C. (2000) When the CRC and TCP Checksum Disagree. In: ACM SIGCOMM 2000, August 28-September 1 2000, Stockholm, Sweden, 10 p.
- [32] Chu H-K. J. (1996) Zero-Copy TCP in Solaris. Proceedings of the USENIX 1996 Annual Technical Conference, San Diego, California, January 1996. 12 p.
- [33] McVoy L. (1998) The splice I/O model. URL: <http://www.bitmover.com/lm/papers/splice.ps>, 5 p.

- [34] Zero copy sockets and NFS patches for FreeBSD (13.7.2001) URL: http://people.freebsd.org/~ken/zero_copy/
- [35] Patch to enable zero-copy sendmsg() on Linux (13.7.2001) URL: <ftp://ftp.inr.ac.ru/ip-routing/zerocopy-sendfile-001207vmhacks.dif.gz>
- [36] Gallatin A., Chase J. & Yocum K. (1999) Trapeze/IP: TCP/IP at Near-Gigabit Speeds. 1999 USENIX Technical Conference (Freenix Track), Monterey, CA, June 1999, 11 p.
- [37] Tekkath C., Nguyen T., Moy E. & Lazowska E. (1993) Implementing Network Protocols at User Level. In: SIGCOMM '93, September 13-17, San Francisco, USA, pp. 64-73.
- [38] von Eicken T., Basu A., Buch V. & Vogels W. (1995) U-Net: A User-level Network Interface for Parallel and Distributed Computing. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp. 40-53
- [39] Salo E., Pinkerton J. et al. (2000, unpublished) libST API (STP OS Bypass API Level 0) draft v07, 28 p.
- [40] Bruggencate M., Lamb G., Voellm A. et al. (2001, unpublished) Libst 2.0 design, 23 p.
- [41] Virtual Interface Architecture Specification (1997). Compaq Computer Corp; Intel Corporation, Microsoft Corporation, 83 p.
- [42] Message Passing Interface Forum (7.11.2001) URL: <http://www.mpi-forum.org/>
- [43] ANSI NCITS 323-1998, Information Technology - High-Performance Parallel Interface - 6400 Mbit/s Physical Layer (HIPPI-6400-PH) (1998). American National Standards Institute, Inc., Washington DC, 59 p.
- [44] InfiniBand Architecture Specification Volume 1 (2001). Infiniband Trade Association, 913 p.
- [45] Boden N., Cohen D., Felderman R., Kulawik A., Seitz C., Seizovic J. & Su W-K. (1995) Myrinet - a Gigabit-per-second Local Area Network. IEEE Micro, vol. 15, no. 1, pp. 29-36
- [46] Wilkes J. (1992) Hamlyn—an Interface for Sender-based Communications. Technical report HPL-OSR-92-13. Operating Systems Research Department, Hewlett-Packard Laboratories, Palo Alto, CA, 18 p.
- [47] Buzzard G., Jacobson D., Marovich S. & Wilkes J. (1995) Hamlyn: a High-performance Network Interface with Sender-based Memory Management. Technical report HPL-95-98. Computer Systems Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA, 13 p.

- [48] Project 1380-D, SCSI on Scheduled Transfer Protocol (SST) Working Draft Revision 05 (2000, unpublished). National Committee for Information Technology Standards T.10, 35 p.
- [49] Winsock Direct (7.11.2001). Microsoft Corp. URL: http://www.microsoft.com/windows2000/en/datacenter/help/wsd_concepts.htm.
- [50] Speight E., Shafi H. & Bennett J.K. (2000) WSDLite: A Lightweight Alternative to Windows Sockets Direct Path., In: 4th USENIX Windows Systems symposium, August 3-4, Seattle, WA, USA, 13 p.
- [51] Sapuntzakis C. & Cheriton D. (2000, unpublished) TCP RDMA Option. IETF Internet-Draft (draft-csapuntz-tcprdma-00.txt), 19 p.
- [52] Williams J. (2000, unpublished) RDMA / TCP. IETF Internet-Draft (draft-williams-rdmatcp-00.txt), 9 p.
- [53] von Eicken T., Culler D., Goldstein S. & Schauser K. (1992) Active Messages: a Mechanism for Integrated Communication and Computation. In: Proceedings of the 19th International Symposium on Computer Architecture, May 1992, pp. 256-266
- [54] Pakin S., Karamcheti V., Chien A. & Efficient M. (1997) Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. IEEE Concurrency, vol. 5, no. 2, pp. 60-73
- [55] Intel Virtual Interface (VI) Architecture Developer's Guide (1997). Intel Corporation, 95 p.
- [56] DiCecco S. & Williams J. (2000, unpublished) VI/TCP (Internet VI). IETF Internet-Draft (draft-dicecco-vitcp-00.txt), 19 p.
- [57] M-VIA: A High Performance Modular VIA For Linux (7.11.2001) URL: <http://www.nersc.gov/research/FTG/via/>.
- [58] Scheduled Transfer Protocol on Linux (11.7.2001). SGI Inc. URL: <http://oss.sgi.com/projects/stp/>.
- [59] Tigon/PCI Ethernet Controller (1997). Alteon Networks, San Jose, USA, 128 p.
- [60] Netperf: A Network Performance Benchmark (1995). Information Networks Division, Hewlett-Packard Company, 35 p.
- [61] Pietikäinen P. (2001) Yantt Network Benchmark Tool 0.1. URL: <http://ppieta.home.cern.ch/ppieta/yantt.tgz>.
- [62] Gigabit Ethernet/PCI Network Interface Card Host/NIC Software Interface Definition revision 12.4.13 (1999). Alteon WebSystems, Inc., San Jose, USA, 80 p.
- [63] CPU Affinity Patch for Linux 2.3.99-pre5 (13.7.2001) URL: http://oss.software.ibm.com/developer/opensource/linux/patches/numa_dlp/ar/affinity_patch.

- [64] Floyd S., Mahdavi J., Mathis M. & Podolsky M. (2000) An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883, 17 p.
- [65] Floyd S. (2001) Congestion Control Principles. RFC 2914, 17 p.
- [66] Tierney B. (2001) TCP Tuning Guide for Distributed Application on Wide Area Networks. Usenix ;login Journal, February 2001, pp. 33-39
- [67] Ang B.S. (2001) An Evaluation of an Attempt at Offloading TCP/IP Protocol Processing onto an i960RN-based NIC. Technical report HPL-2001-8. Computer Systems and Technology Laboratory, HP Laboratories, Palo Alto, CA, 33 p.

9. APPENDICES

Appendix 1 Configuration used for generating results

Appendix 2 Experimental results

Appendix 1: Hardware configuration used for generating results

The setup used in generating the results in this thesis is shown in Figure 1. To simulate real-life conditions better the endpoints were connected to separate Alteon AceSWITCH 180 switches. The third machine was used as a network simulator to simulate latency and packet-loss in the tests described in 5.3.6 and 5.3.7.

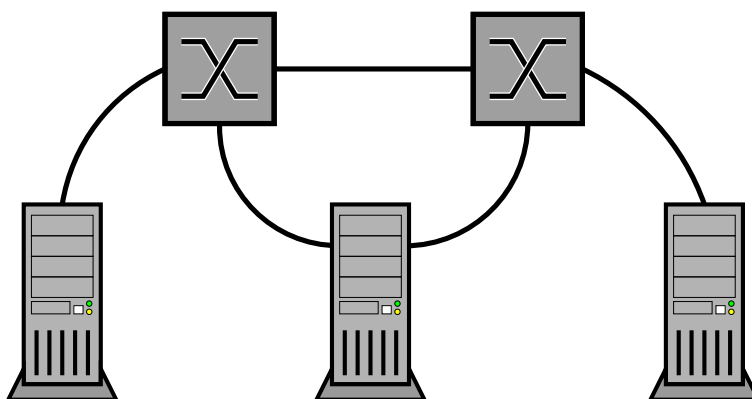


Figure 1. Test setup used in this thesis.

Each test was run three times with each run lasting one minute, with the median of the results being used.. The tests, which required the use of the network simulator used IP encapsulation of STP. In all other tests STP was run over raw Ethernet.

The configurations of the machines was:

Transmitter:

- Intel L440GX+ motherboard with two 500 MHz Pentium III CPU's
- 128 MB of memory
- 32-bit/33 MHz and 32-bit/66 MHz PCI buses
- DEC Gigabit Ethernet Adapter (in 66 MHz PCI slot)

Receiver:

- Tyan Thunder 100 motherboard with two 400 MHz Pentium II CPU's
- 128MB of memory
- 32-bit/33 MHz PCI bus
- DEC Gigabit Ethernet Adapter

Network simulator:

- Intel N440BX motherboard with two 450 MHz Pentium II CPU's

- 128MB of memory
- 32-bit/33 MHz PCI bus
- Netgear GA620 (receiver) and 3com 985 (transmitter)
- Linux kernel 2.2.19
- NIST Network simulator 2.0.10

The Linux kernel version used on both endpoints was 2.4.6 with the STP patch version 0.33 and the XFS-2001-07-05 filesystem patch from SGI. The kernels were compiled using gcc-2.96-81 from Red Hat Linux 7.1. The configuration options the kernel was compiled with were:

```
CONFIG_X86=y
CONFIG_ISA=y
CONFIG_UID16=y
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES=y
CONFIG_MODVERSIONS=y
CONFIG_KMOD=y
CONFIG_M686=y
CONFIG_X86_WP_WORKS_OK=y
CONFIG_X86_INVLPG=y
CONFIG_X86_CMPXCHG=y
CONFIG_X86_XADD=y
CONFIG_X86_BSWAP=y
CONFIG_X86_POPAD_OK=y
CONFIG_RWSEM_XCHGADD_ALGORITHM=y
CONFIG_X86_L1_CACHE_SHIFT=5
CONFIG_X86_TSC=y
CONFIG_X86_GOOD_APIC=y
CONFIG_X86_PGE=y
CONFIG_X86_USE_PPRO_CHECKSUM=y
CONFIG_MICROCODE=y
CONFIG_X86_MSR=y
CONFIG_X86_CPUID=y
CONFIG_NOHIGHMEM=y
CONFIG_MTRR=y
CONFIG_SMP=y
CONFIG_HAVE_DEC_LOCK=y
CONFIG_NET=y
CONFIG_X86_IO_APIC=y
CONFIG_X86_LOCAL_APIC=y
CONFIG_PCI=y
CONFIG_PCI_GOANY=y
CONFIG_PCI_BIOS=y
CONFIG_PCI_DIRECT=y
```

```
CONFIG_PCI_NAMES=y
CONFIG_HOTPLUG=y
CONFIG_SYSVIPC=y
CONFIG_SYSCTL=y
CONFIG_KCORE_ELF=y
CONFIG_BINFMT_AOUT=y
CONFIG_BINFMT_ELF=y
CONFIG_BINFMT_MISC=y
CONFIG_PNP=y
CONFIG_ISAPNP=y
CONFIG_BLK_DEV_FD=y
CONFIG_BLK_DEV_LOOP=m
CONFIG_BLK_DEV_NBD=m
CONFIG_BLK_DEV_RAM=m
CONFIG_BLK_DEV_RAM_SIZE=4096
CONFIG_PACKET=y
CONFIG_PACKET_MMAP=y
CONFIG_NETLINK=y
CONFIG_RTNETLINK=y
CONFIG_NETLINK_DEV=y
CONFIG_UNIX=y
CONFIG_INET=y
CONFIG_IP_MULTICAST=y
CONFIG_IPV6=m
CONFIG_STP=m
CONFIG_ACENIC_STP=m
CONFIG_ACENIC_STP=m
CONFIG_IDE=y
CONFIG_BLK_DEV_IDE=y
CONFIG_BLK_DEV_IDEDISK=y
CONFIG_BLK_DEV_IDECD=y
CONFIG_BLK_DEV_IDEFLOPPY=y
CONFIG_BLK_DEV_CMD640=y
CONFIG_BLK_DEV_RZ1000=y
CONFIG_BLK_DEV_IDEPCI=y
CONFIG_IDEPCI_SHARE_IRQ=y
CONFIG_BLK_DEV_IDEDMA_PCI=y
CONFIG_BLK_DEV_IDEDMA=y
CONFIG_BLK_DEV_IDE_MODES=y
CONFIG_SCSI=y
CONFIG_BLK_DEV_SD=y
CONFIG_SD_EXTRA_DEVS=40
CONFIG_SCSI_DEBUG_QUEUES=y
CONFIG_SCSI_MULTI_LUN=y
CONFIG_SCSI_CONSTANTS=y
CONFIG_SCSI_AIC7XXX=y
CONFIG_AIC7XXX_CMDS_PER_DEVICE=8
```

```
CONFIG_AIC7XXX_RESET_DELAY_MS=15000
CONFIG_SCSI_BUSLOGIC=y
CONFIG_SCSI_OMIT_FLASHPOINT=y
CONFIG_SCSI_SYM53C8XX=y
CONFIG_SCSI_NCR53C8XX_DEFAULT_TAGS=4
CONFIG_SCSI_NCR53C8XX_MAX_TAGS=32
CONFIG_SCSI_NCR53C8XX_SYNC=20
CONFIG_NETDEVICES=y
CONFIG_DUMMY=m
CONFIG_NET_ETHERNET=y
CONFIG_NET_PCI=y
CONFIG_TULIP=m
CONFIG_DE4X5=m
CONFIG_EEPRO100=m
CONFIG_ACENIC=m
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_SERIAL=y
CONFIG_SERIAL_CONSOLE=y
CONFIG_UNIX98_PTYS=y
CONFIG_UNIX98_PTY_COUNT=256
CONFIG_MOUSE=y
CONFIG_PSMOUSE=y
CONFIG_RTC=y
CONFIG_AGP=y
CONFIG_AGP_INTEL=y
CONFIG_REISERFS_FS=m
CONFIG_ISO9660_FS=y
CONFIG_JOLIET=y
CONFIG_PROC_FS=y
CONFIG_DEVPTS_FS=y
CONFIG_EXT2_FS=y
CONFIG_PAGE_BUF=y
CONFIG_XFS_FS=y
CONFIG_HAVE_ATTRCTL=y
CONFIG_XFS_DMAPI=y
CONFIG_NFS_FS=y
CONFIG_NFS_V3=y
CONFIG_NFSD=y
CONFIG_NFSD_V3=y
CONFIG_SUNRPC=y
CONFIG_LOCKD=y
CONFIG_LOCKD_V4=y
CONFIG_MSDOS_PARTITION=y
CONFIG_NLS=y
CONFIG_NLS_DEFAULT="iso8859-1"
CONFIG_NLS_CODEPAGE_437=m
```

```
CONFIG_NLS_CODEPAGE_850=m  
CONFIG_NLS_ISO8859_1=m  
CONFIG_VGA_CONSOLE=y  
CONFIG_MAGIC_SYSRQ=y  
CONFIG_KDB=y  
CONFIG_KALLSYMS=y  
CONFIG_FRAME_POINTER=y
```

The following tunable kernel options were changed from the default values

```
net.core.rmem_max = 524288  
net.core.wmem_max = 524288
```

The switches were configured to support jumbo frames and IEEE 802.3x flow-control.

Appendix 2: Raw data for experimental results

The following tables contain the raw data from which the experimental results in Chapter 5 were obtained.

Table 1. Comparison of STP and TCP

test	protocol	MTU	bw (MB/s)	RX cpu	TX cpu	bw/RX eff.	bw/TX eff.
writing from buffer	STP	9000	98	8	17	12.25	5.76
touch	STP	9000	53	23	9	2.30	5.89
writing from buffer	STP	1500	52	5	23	10.40	2.26
touch	STP	1500	34	16	13	2.12	2.62
buffer	TCP	9000	86	56	44	1.54	1.95
+MSG_TRUNC	TCP	9000	99	23	54	4.30	1.83
sendfile()	TCP	9000	78	56	14	1.39	5.57
+MSG_TRUNC	TCP	9000	99	23	27	4.30	3.67
touch-128k	TCP	9000	61	54	30	1.13	2.03
touch-512k	TCP	9000	40	54	27	0.74	1.48
buffer	TCP	1500	57	58	44	0.98	1.30
+MSG_TRUNC	TCP	1500	66	60	65	1.10	1.02
sendfile()	TCP	1500	61	60	32	1.02	1.91
+MSG_TRUNC	TCP	1500	70	52	41	1.35	1.71
touch-128k	TCP	1500	40	57	31	0.70	1.29
touch-512k	TCP	1500	16	54	13	0.30	1.23
Transmitter: <code>yantt -b <record_size> -P <stp/tcp> -c <server_ip> [sendfile]</code> Receiver: <code>yantt -b <record_size> -P <stp/tcp> [truncate touch]</code> sendfile: <code>-i filename:sendfile</code> truncate: <code>--trunc</code> touch: <code>--touch</code> <i>record_size</i> is 524288 in all tests except TCP touch-128k							

Table 2. Interrupt rates of STP and TCP.

Protocol	MTU	bw (MB/s)	RX ints/sec	TX ints/sec	RX CPU (%)	TX CPU (%)
STP	9000	98	1609	3154	8	17
TCP	9000	86	2149	2893	56	44
TCP, no coal.	9000	68	6014	6236	43	31
STP	1500	52	2700	3088	5	23
TCP	1500	57	2793	3488	58	44
TCP, no coal.	1500	49	21611	23605	64	50
Transmitter: <code>yantt -b 524288 -P <stp/tcp> -c <server_ip></code> Receiver: <code>yantt -b 524288 -P <stp/tcp></code> Interrupt rate: <code>sar -I <NIC interrupt> 60</code>						

Table 3. Effect of STU size on STP performance

STU size	bandwidth (MB/s)	RX CPU (%)	TX CPU (%)
512	27	4	18
1024	53	6	18
2048	87	6	18
4096	98	6	17
MTU: 576, 1088, 2112, 4160 Transmitter: <code>yantt -b 524288 -P stp -c <server_ip></code> Receiver: <code>yantt -b 524288 -P stp</code>			

Table 4. Effect of latency on STP and TCP.

round-trip latency (ms)	STP 512k	STP 128k	TCP 512k	TCP 128k
0	42.54	31	44.9	44.35
0.25	41.73	26.35	45.22	44.99
0.5	38.5	24.16	45.3	44.65
1	35.95	21.14	45.24	40.02
2.5	30.11	15	45.3	26.64
5	23.99	9.39	45.18	16.61
10	17.94	5.36	33.27	9.11
20	10.46	2.88	18.02	4.79
50	4.63	1.21	7.45	1.96
100	2.4	0.61	3.76	1.01
MTU: 9000 Transmitter: <code>yantt -b <record_size> -P <stp/tcp> -c <server_ip></code> Receiver: <code>yantt -b <record_size> -P <stp/tcp></code> round-trip latency is divided equally in both directions				

Table 5. Effect of packet loss on STP and TCP performance.

packet drop (%)	STP 128k	STP 512k	TCP 512k
0	30.92	41.65	45.4
0.0001	30.9	41.54	45.42
0.0005	30.71	41.75	45.38
0.001	20.68	22.29	45.28
0.01	5.23	5.52	44.62
0.1	0.56	0.75	40.12
MTU: 9000 Transmitter: <code>yanttt -b <record_size> -P <stp/tcp> -c <server_ip></code> Receiver: <code>yanttt -b <record_size> -P <stp/tcp></code> Reply packets are not dropped			

Table 6. Effect of Transfer size on STP performance (jumbo frames)

Transfer size	bandwidth (MB/s)	RX CPU (%)	TX CPU (%)
4096	9	12	14
8192	17	12	13
16384	29	12	13
32768	45	10	12
65536	64	8	11
131072	81	6	11
262144	91	5	15
524288	98	6	17
MTU: 9000 Transmitter: <code>yanttt -b <record_size> -P stp -c <server_ip></code> Receiver: <code>yanttt -b <record_size> -P stp</code>			

Table 7. Effect of Transfer size on STP performance (standard frames)

Transfer size	bandwidth (MB/s)	RX CPU (%)	TX CPU (%)
4096	10	15	15
8192	17	14	16
16384	27	11	18
32768	32	10	16
65536	40	7	16
131072	47	6	19
262144	50	5	20
524288	52	5	23
MTU: 1500 Transmitter: <code>yanttt -b <record_size> -P stp -c <server_ip></code> Receiver: <code>yanttt -b <record_size> -P stp</code>			