TCP SERVERS: A TCP/IP OFFLOADING ARCHITECTURE FOR INTERNET SERVERS, USING MEMORY-MAPPED COMMUNICATION

BY KALPANA S BANERJEE

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Liviu Iftode

and approved by

____

____

____

New Brunswick, New Jersey

Oct, 2002

ABSTRACT OF THE THESIS

TCP Servers: A TCP/IP Offloading Architecture for Internet Servers, using Memory-Mapped Communication

by Kalpana S Banerjee

Thesis Director: Liviu Iftode

TCP Server is a system architecture aiming to offload network processing from the host(s) running an Internet server. The TCP Server can be executed on a dedicated processor, node or intelligent network interface using low-overhead, non-intrusive communication between it and the host(s) running the server application. In this thesis, we present and evaluate an implementation of the TCP Server architecture for Internet servers on clusters built around a memory-mapped communication interconnect. We have quantified the impact of offloading on the performance of Internet servers for our TCP Server implementation, using a server application with realistic workloads. We were able to achieve performance gains of up to 30% due to offloading for the scenarios studied. Based on our experience and results, we conclude that offloading the network processing from the host processor using a TCP Server architecture is beneficial to server performance when the server is overloaded. A complete offloading of TCP/IP processing demands substantial computing resources on the TCP Server. Depending on the application workload, either the host processor, or the TCP Server, can become the bottleneck indicating the need for an adaptive scheme to balance the load between the host and the TCP Server.

Acknowledgements

I would like to thank my advisor Professor Liviu Iftode, for his help, support and insight at every stage of my research. His motivation has taught me the value of looking deeper and further beyond what is immediately evident.

My heartfelt and sincere thanks to Murali Rangarajan for always being ready to help. I would like to thank him for the many discussions we had regarding the design, implementation and evaluation of my work.

I would like to thank Professor Ricardo Bianchini and Professor Richard Martin for taking the time to be in my Master's thesis committee and for providing me with valuable inputs.

I would like to thank all the members of DiscoLab for their support and Aniruddha in particular for his help with the experimental setup for the performance evaluation. I am thankful to my friends Deepa, Xiaoyan and Srinath for the many good times we shared.

The support and motivation of my family - my parents, my parents-in-law, Viji, Ananth, Latha, Ravi, Suprio, Madhumita, Srikanth, Shalini and little Akshaya - has helped me at every single stage.

I am grateful to my husband Saikat for always being by my side.

Dedication

Dedicated to my parents, for their endless love, strength and guidance.

Table of Contents

Abstract ii

Acknowledgements iii

Dedication iv

List of Tables ix

List of Figures x

1. Introduction 1

1.1. Internet Servers 1

1.1.1. Server Application 2

1.1.2. Role of the Operating System 4

1.1.3. Server to OS Interaction 4

1.2. Internet Server Performance Issues 5

1.2.1. Network Processing Overheads 6

1.2.2. Experimental Measurement of the Network Processing Overheads . . . 6

1.2.3. Hardware Offloading Solutions 7

1.3. Our Approach - TCP Server Architecture 8

1.3.1. TCP Servers for Clusters 8

1.3.2. Contributions 10

1.4. Outline of the Rest of Document 10

2. Related Work 11

2.1. OS-Based Solutions 11

2.2. Cluster-Based Network Servers 12

2.3. New I/O Technology 13

2.4. TCP Servers for SMP Systems 15

3. TCP Server Architecture 16

3.1. Traditional Network Processing 16

3.1.1. Send and Receive Processing 16

3.1.2. Components of TCP/IP Processing 18

3.1.3. Breakdown of Network Processing Overheads 20

3.2. TCP Server Architecture 20

3.2.1. Scope for Optimization 24

4. Design of TCP Servers 27

4.1. Design Alternatives 27

4.1.1. Socket Call Processing at the Host 27

4.1.2. TCP Server 27

4.2. Network Processing Mechanism 29

4.3. Host to TCP Server Communication using VIA 30

4.3.1. VI Architecture Overview 31

4.4. Mapping Sockets to VIs 34

4.4.1. Alternatives Based on Request Processing at the TCP Server 34

4.4.2. Communication Library at Host Node 36

4.5. TCP Server Components 37

4.6. Optimizations 38

4.6.1. Asynchronous Send 39

4.6.2. Eager Receive 40

4.6.3. Eager Accept 42

4.6.4. Setup with Accept 42

4.6.5. Avoiding Data Copies at the Host 44

5. Implementation 45

5.1. Application Programming Interface 45

5.2. Host End Point 46
5.3. Request/Response Protocol 49
5.4. TCP Server Implementation 51
5.5. Optimizations 53
5.5.1. Asynchronous Sends 54
5.5.2. Eager Receive 58
5.5.3. Eager Accept 60
5.5.4. Avoiding Data Copies at the Host 60

6. Performance Evaluation 62
6.1. Experimental Setup 62
6.1.1. Hardware Platform 62
6.1.2. Web Server 62
6.1.3. Client Benchmarking Tool 63
6.2. Microbenchmarks 63
6.2.1. SAN Performance Characteristics 63
6.2.2. Cost of Send Call 63
6.2.3. Detailed Analysis 67
6.3. TTCP Benchmark Results 69
6.4. Web Server Performance 71
6.4.1. Initial Experiments with Fast Ethernet 72
6.4.2. HTTP/1.0 Performance 73
6.4.3. HTTP/1.1 Performance 80
6.4.4. Performance with Real Traces 81

7. Conclusions and Future Directions 84
7.1. Conclusions 84
7.2. Future Work 84
References 86

Appendix A. VI Primitives 90

List of Tables

6.1. VIA Microbenchmarks 63

6.2. Cost of `send` 64

6.3. Breakdown of the Cost of `send` 65

6.4. VIA One-Way Latency 66

6.5. Legends Used in the Graphs 69

6.6. Main Characteristics of WWW Server Traces 82

List of Figures

1.1. Client-Server Model . . . 2

1.2. Internet Server . . . 3

1.3. Apache Execution Time Breakdown . . . 7

1.4. Traditional Internet Server Architecture . . . 8

1.5. Internet Server Architecture based on TCP Servers . . . 9

1.6. TCP Servers for Clusters . . . 9

3.1. Network Processing in Linux . . . 17

3.2. Components of TCP/IP Processing . . . 18

3.3. Apache Execution Time Breakdown . . . 21

3.4. TCP Server Architecture . . . 21

3.5. Separation of Components of TCP/IP Processing . . . 22

3.6. Network Processing with TCP Servers . . . 23

4.1. Alternatives for TCP Servers over Clusters . . . 28

4.2. Network Processing with TCP Servers . . . 29

4.3. The VI Architecture Model . . . 31

4.4. Host and TCP Server Communication using VIA . . . 33

4.5. Design Alternative 1 . . . 35

4.6. Design Alternative 2 . . . 35

4.7. Design Alternative 3 . . . 35

4.8. Components of the TCP Server . . . 37

4.9. Synchronous Call Processing . . . 39

4.10. Asynchronous Send Processing . . . 40

4.11. Eager Receive (pull-based) Processing . . . 41

4.12. Eager Receive (push-based) Processing . . . 41

4.13. Eager Accept Processing 42
4.14. Setup With Accept 43
5.1. Socket to VI Mapping at the Host 47
5.2. Request/Response Format 49
5.3. Threaded Model of the TCP Server 51
5.4. Asynchronous Send Call Processing 54
5.5. Location of Results of Requests 56
5.6. Example Flow Control Processing 57
5.7. Eager Receive Buffer 59
6.1. Cost of `send` 64
6.2. Breakdown of the Cost of `send` 65
6.3. Flow of `send` 67
6.4. Flow of Synchronous send in the TCP Server Architecture 68
6.5. Flow of Asynchronous send in the TCP Server Architecture 68
6.6. Throughput Measured at the Host Node - the TTCP Transmitter 70
6.7. Throughput Measured at the Client Node - the TTCP Receiver 71
6.8. Web Server Throughput on Fast Ethernet 72
6.9. Web Server Throughput for HTTP/1.0 Static Loads 73
6.10. CPU Utilization for HTTP/1.0 Static Loads 74
6.11. Web Server Throughput with Repeated Requests to the Same File 76
6.12. Web Server Throughput for HTTP/1.0 Combined Loads 77
6.13. CPU Utilization for HTTP/1.0 Combined Loads 78
6.14. Web Server Throughput for HTTP/1.0 Static Loads, with 16K Transfer Size . . 79
6.15. CPU Utilization for HTTP/1.0 Static Loads, with 16K Transfer Size 80
6.16. Web Server Throughput for HTTP/1.1 Loads, with 16K Transfer Size 81
6.17. CPU Utilization for HTTP/1.1 Loads, with 16K Transfer Size 82
6.18. Web Server Throughput with a Real Trace (Forth) 83

Chapter 1

Introduction

The Internet of today with close to hundred million hosts has come a long way from its modest beginnings in the form of ARPANET in the 1960's. The continuously growing popularity of the Internet and World Wide Web applications places increasing demands on the performance of Internet servers. As the processing power of the Internet servers increases, the network subsystem plays a crucial role in determining server performance. This thesis is an effort to develop a TCP/IP offloading solution, and to study the impact of TCP/IP offloading, on the performance of Internet servers.

1.1 Internet Servers

Internet servers provide several services like `Hyper Text Transport Protocol` (HTTP), `File Transfer Protocol` (FTP), `telnet` etc, to clients across networks ranging from the `Local Area Network` (LAN) to the `Wide Area Network` (WAN). Most of these services are built over the popular transport layer protocol, the `Transmission Control Protocol` (TCP, RFC-793). TCP is a connection-oriented reliable transport layer protocol built over the unreliable, best-effort connection-less network layer protocol `Internet Protocol` IP. The TCP/IP protocol suite enables inter-networking between computers connected across heterogeneous physical networks, and is one of the most popular protocol suites in use today.

Client-Server Model: Internet servers are based on the traditional client-server model. Figure 1.1 shows a typical client-server communication scenario where an Internet server services several clients across the wide area network. The clients are the *service requestors* and send requests for services and/or data to the server. The server is the *service provider*. On receiving a client request, the server processes the request and replies back to the client from which the request originated. Figure 1.2 shows a schematic of the server. All communication from the

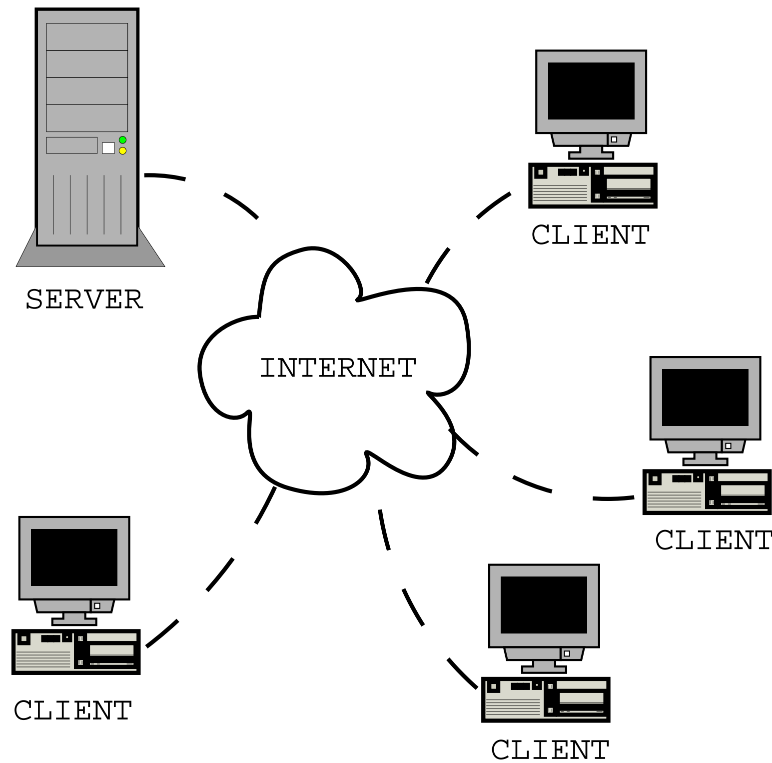

Figure 1.1: Client-Server Model

server application passes through the network subsystem of the operating system and is then directed to the outside world through the network adapter. Three major components that constitute an Internet server are the server application, the operating system and the interface from the server application to the operating system. In the following subsections, we discuss each of the three components.

1.1.1 Server Application

In the client-server model, the design and architecture of the server plays an important role in determining the performance and scalability of the entire system. Several design choices are available. Server applications are typically multi-threaded or multi-process to enable concurrent processing of multiple client requests. The multi-process model involves frequent context switching between the various processes and the use of Inter Process Communication primitives. In the case of the multi-threaded model, the threads can be user-level or kernel-level. In both cases, the working pool of threads or processes can be created *on-demand* or *pre-created*.

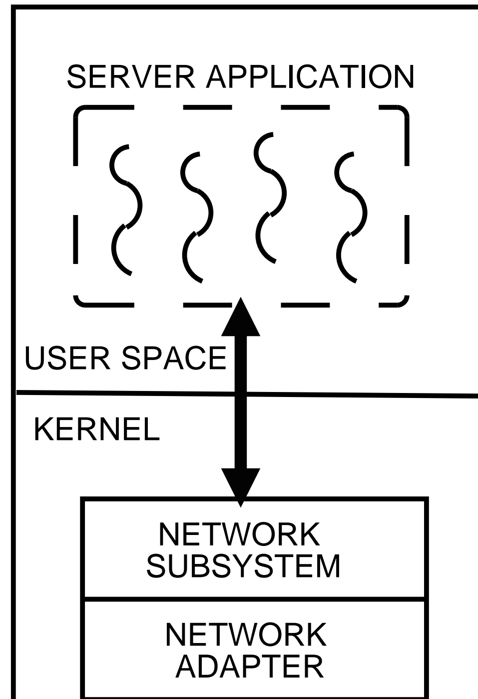

Figure 1.2: Internet Server

In the case of the *pre-created* working pool, the size of the working pool can be static, or can be dynamically adjusted depending on the server load.

The Web server is a typical example of an Internet server. It receives HTTP requests from clients and sends HTTP responses back to the clients. To be specific in our discussion, we use the Web server as a classical example of Internet servers. Most of the issues arising in the context of Web servers, however, are equally applicable to Internet servers in general. Clients send connection requests to the Web server on a well-known port. The Web server associates a socket, known as *listen socket*, with this well-known port. Incoming connection requests are received, accepted and queued on the *listen socket*. Each time the server issues an `accept`, a new socket is associated with this HTTP connection, and the server can then issue `recv` and `send` on this connected socket to communicate with the client. The client application at its end, issues a `connect` system call requesting the server for a connection. Once the connection has been established, it sends requests to the server on the established connection.

1.1.2 Role of the Operating System

The operating system is entirely responsible for the allocation and management of shared system resources. Access to the Network Adapter, disk, and other I/O devices is protected and under the control of the OS. The OS is also responsible for scheduling CPU time between various competing processes. Policies for dynamic scheduling, main memory allocation, and swapping are enforced to provide fairness in a time-shared system. Internet servers rely heavily on the network subsystem of the OS. In UNIX-based operating systems and many non-UNIX operating systems, the network subsystem is interrupt-driven. An incoming packet raises an asynchronous interrupt and passes upwards through the network protocol processing layers and then reaches the destination socket queue. Outgoing packets traverse through the multiple protocol layers and are then transmitted out on to the network. Thus, a substantial portion of the work involved in processing a request is under the control of the OS. Internet servers also rely on the storage subsystem to retrieve files that are requested by the clients.

1.1.3 Server to OS Interaction

System calls represent the interface between the server applications and the operating system and provide a mechanism for applications to request OS services. A system call is typically executed as a trap or interrupt instruction which causes the execution to jump to a fixed entry point in the OS. Once within the context of the OS, privileged operations like device access are performed by the OS on behalf of the requesting application. Before the execution switches from the user-level into the kernel, the application state is saved and then restored just before the call returns. Thus system calls are expensive. Socket calls which enable applications to perform network operations are implemented as system calls. While executing a system call, in addition to the execution state moving from the user application on to the kernel, data must also be copied from application buffers on to kernel buffers and vice versa. In the case of the `send` system call, data from the application communication buffers are copied on to kernel buffers and then transmitted on to the network. In the case of the `recv` system call, data is received from the network on to kernel buffers and then after the receive network processing for that data has been completed the data is copied on to the application receive buffers. The copy from

user to kernel buffers adds to the costs of the system calls.

1.2 Internet Server Performance Issues

The Internet server architecture outlined in the previous section is designed to service several simultaneous clients. However, with the explosive growth of the World Wide Web, the performance of the Internet servers is becoming increasingly critical. Internet servers must be able to scale to thousands and sometimes millions of clients. Often, the speed perceived by the users is dictated by the server performance. In addition to the increasing number of clients, with the availability of high-speed, gigabit-per-second networks, higher bandwidth requirements are being imposed on servers. The processing power of the Internet servers has also increased considerably. As the processing power of the Internet servers increases, the OS in general, and specifically the storage and the network subsystems of the OS prove to be performance bottlenecks.

The term *OS intrusion* was introduced in Piglet [38] to represent the mechanisms and policies of the operating system's resource management that have a considerable negative impact on the performance of applications. OS-based solutions like IO-Lite and Resource Containers [43, 7] have been proposed, to improve the mechanisms and policies used by the OS and therefore reduce the impact of *OS intrusions* on server applications. Caching, server clustering, and intelligent request distribution techniques have helped to alleviate some of the storage system bottlenecks by removing the disk access from the critical path of the request processing [42]. However, the same is not true of the network subsystem. In the case of network processing, every single data byte has to go through the complete processing path of the network protocol stack to and from the network device. In addition to "stealing" processor cycles from the applications, asynchronous interrupt processing and frequent context switching also cause indirect effects like cache and TLB pollution, which further increases the overheads due to the network processing.

1.2.1 Network Processing Overheads

The TCP/IP protocol processing requires a significant amount of host system resources and competes with the application processing time. This problem becomes more critical at high loads because

- A packet arrival results in an asynchronous interrupt which preempts the current execution irrespective of whether the server has sufficient resources to process the newly arrived packet in its entirety.
- When a packet finally reaches the socket queue, it may be dropped because sufficient resources are not available to complete its processing. The dropping of the packet however occurs only after a considerable amount of system resources have already been spent on the protocol processing of the packet.

Thus at high loads, the above conditions can cause the system to enter a state known as *receive livelock* [36]. In this state, the system spends considerable resources processing new incoming packets, only to drop them later because of lack of resources for the applications actually processing the data. This situation is all the more critical as high-speed, gigabit-per-sec networks are becoming increasingly available. The host processors can easily be saturated by the network protocol processing overheads limiting the potential gain in network bandwidth [4]. In a traditional system architecture, performance improvements in network processing can come only from optimizations in the protocol processing path [21]. Replacing the expensive TCP/IP protocol processing by a light-weight, more efficient transport layer protocol using user-level communication and memory-mapped communication based on standards such as the Virtual Interface Architecture (VIA) [22] and Infiniband (IB) [31] have been proposed [48, 44]. So far these have been used only in the context of intra-server communication.

1.2.2 Experimental Measurement of the Network Processing Overheads

In TCP Servers [45], the time allotted to network processing from the execution time of an Apache [5] Web server was quantified when the server was overloaded with client requests. In this experiment, a synthetic workload of repeated requests for a 16KB file cached in memory

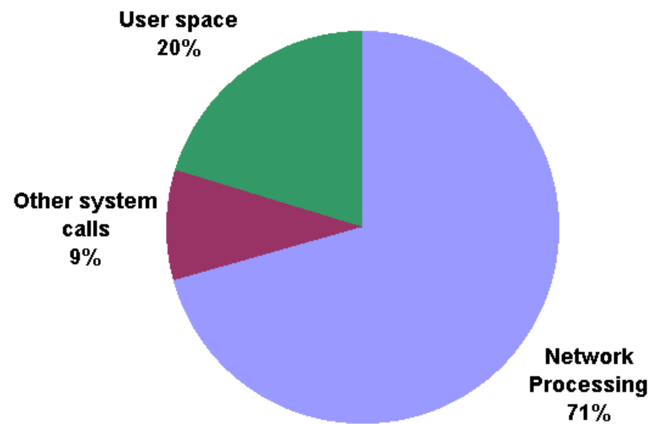

Figure 1.3: Apache Execution Time Breakdown

was used. Figure 1.3 shows the execution time breakdown on a dual Pentium 300 MHz system with 512 MB RAM and 256 KB L2 cache, running Linux 2.4.16. The Linux kernel was instrumented to measure the time spent in every function during the execution path of the `send` and `recv` system calls, as well as the time spent in interrupt processing.

The results show that the Web server spends only 20% of its execution time in user space. The entire TCP/IP processing takes 71% of the CPU time. The time spent in processing other system calls is about 9%. This shows that the time spent in TCP/IP processing is significantly higher than the time spent on actual application processing by the Web server.

1.2.3 Hardware Offloading Solutions

It has been recognized that the TCP/IP processing consumes considerable host system resources and there have been efforts at offloading the TCP/IP processing. Hardware solutions have been proposed to offload some or all of the TCP/IP protocol processing on to an Intelligent Network Interface card (I-NIC). Currently there are several cards available in the market [3, 19, 24, 32, 35, 53] which provide varying levels of TCP/IP offloading to speed up the common path of the protocol processing.

1.3 Our Approach - TCP Server Architecture

The aim of our work is to understand the design, implementation, and performance of server architectures that rely on TCP/IP offloading for client-server communication. We propose the TCP Server architecture, a system architecture which aims to de-couple application processing from the network processing. The TCP/IP processing is offloaded from the host(s) running an Internet Server and is executed on a dedicated processor, node, or intelligent network interface known as the TCP Server. The host(s) running the server application communicate with the TCP Server(s) using a low-overhead, non-intrusive communication medium.

Figure 1.4 shows the traditional Internet server architecture. In the conventional architecture,

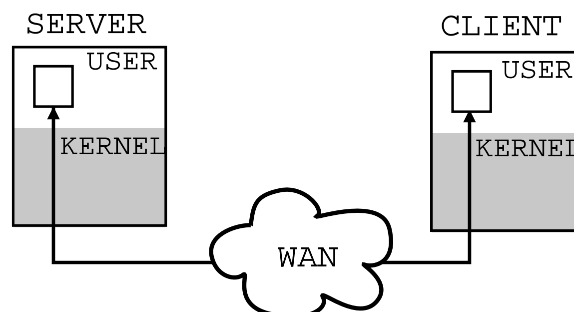

Figure 1.4: Traditional Internet Server Architecture

TCP/IP processing is done in the OS kernel of the node which executes the server application. Figure 1.5 is the Internet server architecture based on TCP Servers. Here, the application host avoids TCP/IP processing by tunneling the network processing to the TCP Server using fast communication channels. The TCP Server architecture separates the application processing from network processing and shields the application from OS overheads associated with network processing. The communication medium between the host and the TCP Server is critical to the success of the TCP Server architecture. It has to be efficient and must provide lightweight communication with minimal overheads, if any.

1.3.1 TCP Servers for Clusters

In this thesis, we propose a TCP Server architecture for cluster-based network servers using memory-mapped communication. System Area Networks provide low-latency and high-bandwidth communication, and are an attractive option for building high-speed clusters. The

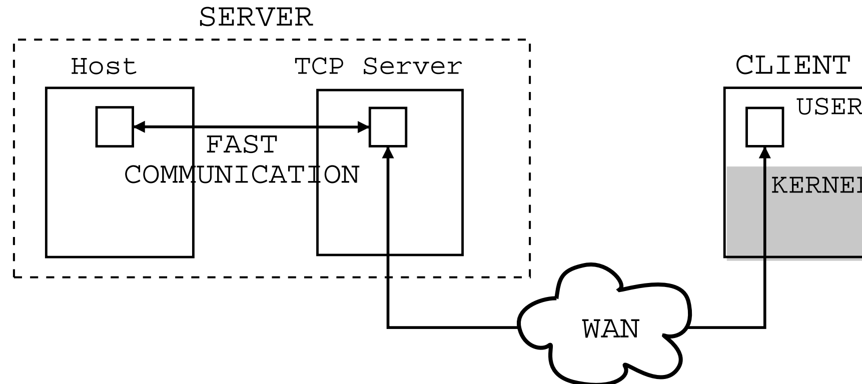

Figure 1.5: Internet Server Architecture based on TCP Servers

Virtual Interface Architecture (VIA) [22] is a standard for user-level memory-mapped communication that enables high-speed communication across a System Area Network (SAN). In the TCP Server architecture we dedicate one or more nodes of the cluster, referred to as TCP Server nodes, to handle network processing. The remaining nodes of the cluster, referred to as host nodes, perform application processing and OS functions not related to network processing. The host node(s), communicate with the TCP Server node(s) across the cluster through the low-overhead, non-intrusive memory-mapped communication provided by the SAN.

Figure 1.6 shows the TCP Server architecture for clusters. The host avoids TCP/IP processing

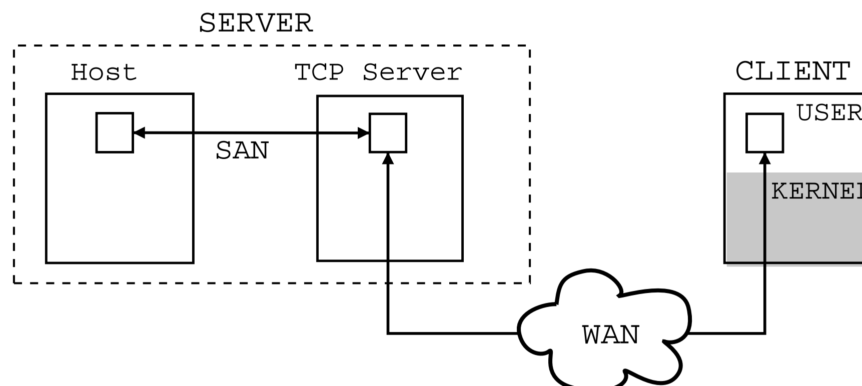

Figure 1.6: TCP Servers for Clusters

by tunneling network requests across the SAN to the TCP Server. The network processing is executed at the TCP Server. The TCP Server is the communication end-point for clients connecting across the Internet. The host nodes are provided with a programming interface to communicate with the TCP Server for network-related processing.

1.3.2 Contributions

The main contributions of our research work are:

- Design of the TCP Server architecture for Internet servers based on memory-mapped communication.
- Implementation of TCP Servers for Internet servers based on memory-mapped communication.
- Programming interface for applications to exploit the TCP Server architecture.
- Performance evaluation of a system using TCP Servers in a cluster.

1.4 Outline of the Rest of Document

The rest of the document is organized as follows:

- Chapter 2 provides a description of the related work.
- Chapter 3 gives the high level design of TCP Servers.
- Chapter 4 concentrates on the design details of TCP Servers, based on memory-mapped communication and the Virtual Interface Architecture.
- Chapter 5 gives details of our implementation.
- Chapter 6 provides performance evaluation results.
- Chapter 7 concludes with Future Work.

Chapter 2

Related Work

Work related to our TCP Server architecture for clusters can be broadly classified into the following categories:

- OS-Based Solutions
- Cluster-Based Network servers
- New I/O Technology

A description of the related work in each of these categories is given in the following sections. The TCP Server architecture on a Symmetric Multiprocessor (SMP) system is also discussed.

2.1 OS-Based Solutions

Overheads as a result of policies and mechanisms used by the OS can have a significant impact on server performance. OS mechanisms and policies specifically tailored for servers have been proposed in [21, 43, 7].

In Lazy Receiver Processing (LRP) [21], the network subsystem was optimized to provide better performance for network servers. The network interface demultiplexed incoming packets directly on to per socket queues and whenever the protocol semantics allowed it, protocol processing was performed lazily.

IO-Lite [43] proposed an unified buffering and caching system among various input-output (I/O) subsystems and applications for general purpose operating systems. The primary goal of IO-Lite was to improve the performance of server applications like Web servers and other I/O intensive applications. Redundant data copying and multiple buffering were avoided, and cross subsystem optimizations were proposed.

Resource Containers [7] targeted Web servers and provided improved resource management to the applications in the form of *resource containers*. A *resource container* is an abstract operating system entity that logically contained the system resources used by a given application. *Resource container*s could explicitly be controlled by the application thereby giving it control of the resources used by it.

While LRP, IO-Lite and Resource Containers recognize the existence of *OS intrusion* and suggest ways of reducing it, they do not study the effect of separating the application processing from network processing or shielding an application from *OS intrusion*.

End-Systems Optimizations [16] showed that on high-speed networks, the delivered performance was often limited by the sending and receiving hosts. They proposed optimizations above and below the TCP protocol stack to reduce host overheads.

An important factor in the performance of a server is its ability to handle extremely high volume of receive requests. Under such conditions, the system enters a *receive livelock*. This phenomenon was reported by Mogul and Ramakrishna [36]. Several researchers suggest the use of polling on the system to prevent receive livelock and for high performance [49, 34]. Aron and Druschel in [6] use the soft timer mechanism to poll on the network interface. The idea is extended in Piglet [38], where the application is isolated from the asynchronous event handling using a dedicated polling processor in a multiprocessor. In fact, one of the earliest studies on dedicating processors for I/O was done at IBM for the IBM TSS 360 [29] in a multiprocessor system. The main focus of the dedication was storage, and networking at that time was not an important design criterion.

2.2 Cluster-Based Network Servers

There has been considerable amount of research work done in the area of cluster based network servers. Clusters of commodity computers connected over a high speed interconnect are capable of providing the scalability required by internet servers. Cluster-based network servers typically consist of a front-end node which distributes requests amongst several back-end nodes. Locality-Aware Request Distribution (LARD) [42] proposed distribution of requests to the back-end servers based on the content of the request. This approach is different from a

content-oblivious request distribution based on parameters like server load [30, 17]. LARD resulted in performance gains due to *cache locality* and secondary storage scalability by facilitating server database partitions among the different back end nodes. PRESS [12] is a locality-aware cluster-based WWW server that uses the Virtual Interface Architecture (VIA) for intra-server communication. An evaluation of the impact of the features of the intra-server communication architecture on PRESS was studied in [14]. In this study, the effect of processor overhead, network bandwidth, remote memory writes, and zero-copy data transfers on the performance of the Web server were studied and evaluated.

2.3 New I/O Technology

Intelligent devices have been shown to be a promising innovation for servers, especially in the case of storage systems [27, 1, 10]. Intelligent Network Interfaces [39] have also been studied, but mostly for cluster interconnects in distributed shared memory [26] or distributed file systems [4]. Recently released Network Interface Cards have been equipped with hardware support to offload the TCP/IP protocol processing from the host [3, 35, 2, 19, 24, 53, 32]. Some of these cards also provide support to offload network protocol processing for network attached storage devices, including iSCSI, from software on the host processor to dedicated hardware on the adapter. While these approaches support offloading on specialized hardware, the goal of the TCP Server architecture is to provide a generic TCP/IP offloading architecture.

The introduction of Infiniband [31], a switch based serial I/O interconnect capable of operating at base speeds ranging from 2.5 to 30 Gb/s, has triggered considerable interest in industry and academia. Scalable server systems can be built by connecting host processors and I/O devices across an Infiniband fabric. In this architecture, host processors and devices can communicate amongst themselves at gigabit speeds breaking conventional I/O interconnect bottlenecks. In this context, our TCP Server architecture across clusters, can also be viewed as an emulation of host processors connected to Intelligent Networking devices across an Infiniband fabric. Infiniband is closely related to the Virtual Interface Architecture [22] and provides support for a number of features, including memory-mapped communication, that is currently provided by the Virtual Interface Architecture.

Recent work [13] was an effort to study the impact of next generation I/O architectures on the design and performance of network servers. In this work, modeling and simulations were used to analyze a range of scenarios, from providing conventional servers with high I/O bandwidth, to modifying servers to exploit user-level I/O and direct device-to-device communication, and re-designing the operating system to offload file system and networking functions from the host to intelligent devices.

In the Communication Services Platform (CSP) [48] project, the authors suggest a system architecture for scalable cluster-based servers, using dedicated network nodes and a VIA-based network to tunnel the TCP packets inside the cluster. This project has similar goals to our design, i.e., offloading the network processing to dedicated nodes. However, their results are very preliminary and their goal was to limit the network processing to a specific layer in a multi-tier data center architecture. Unlike CSP, we study the effect of such separation of functionality for a server system under real server application workloads. CSP also does not explore the issue of providing a programming interface which allows server applications to exploit performance gains from using an efficient low-latency memory-mapped communication layer.

QPIP [11] is an attempt to provide a lightweight protocol for applications which offloads network processing to the Network Interface Card (NIC). However, they implement only a subset of TCP/IP on the NIC. QPIP suggests an alternate interface to the traditional sockets API but does not define a programming interface that can be exploited by applications to achieve better performance. Moreover, performance evaluation presented in [11] was limited to communication between QP-aware applications over a SAN.

Sockets Direct Protocol (SDP) [44] originally developed to support server-clustering applications over VI architecture, has been adopted as part of the InfiniBand specification. The SDP interface makes use of InfiniBand capabilities and acceleration while emulating a standard socket interface for applications.

Fast Sockets [46] proposed a low-overhead communication protocol but this was only in the context of high-performance Local Area Networks.

Voltaire has proposed a TCP Termination Architecture [51] with the goals of solving the bandwidth and CPU bottlenecks which occur when other solutions such as IP Tunneling or bridging are used to connect InfiniBand Fabrics to TCP/IP networks.

Direct Access Transport (DAT) [20] is an initiative to provide a transport exploiting remote memory capabilities of interconnect technologies like VIA and Infiniband. However, the objective of DAT is to expose the benefits of remote memory semantics only to intra-server communication.

2.4 TCP Servers for SMP Systems

In TCP Servers [45], an implementation of the TCP Server architecture on a symmetric multiprocessor (SMP) system was presented. In a cluster of nodes a subset of nodes, the TCP Server nodes, are dedicated to network processing. The remaining nodes of the cluster, the host nodes, perform application processing and processing of operating system functionality not related to network processing. The host nodes and the TCP Server nodes communicate using memory-mapped communication across the cluster. In a SMP system, a subset of the processors are dedicated to network processing. The remaining processors (host processors) perform application processing and processing of operating system functionality not related to network processing. The host processors and the dedicated processors communicate using shared memory. Comparing the two architectures, it is seen that using shared memory in a SMP system results in lower latency and higher bandwidth of communication compared to using memory-mapped communication across clusters. The processor overhead for communication is also lower for a SMP system compared to clusters. However, clusters provide better insulation of application processing from network processing. For instance, if the TCP Server is subjected to a Denial of Service attack and a particular TCP Server node is brought down, a well configured system can restrict the fault to that single node and application processing on host nodes can be insulated from the fault. Clusters also provide ease and flexibility for building heterogeneous configurations. The processing power and memory of a TCP Server node can be varied independent of the processing power and memory of a host node. The TCP Server architecture over clusters could also potentially scale to very large systems.

Chapter 3

TCP Server Architecture

We begin this chapter with a discussion of the traditional network processing and identify components of the TCP/IP protocol processing that can be offloaded. We then discuss network processing using the TCP Server architecture.

3.1 Traditional Network Processing

The following discussion of the traditional network processing is based on an interrupt-driven, Unix-based networking stack. To be concrete in our explanation we have used the Linux TCP/IP protocol stack as an example. As shown in Figure 3.1 [1], the Linux network processing architecture is made up of a series of connected layers of software, similar to the layered architecture of the network protocols themselves. BSD sockets is the general socket interface for both networking and inter-process communication sockets. The INET socket layer supports the internet address family. The BSD socket layer invokes the INET layer socket support routines to perform work for it. Beneath the INET socket layer, are the transport layer protocols UDP (User Datagram Protocol) and TCP processing routines. Following this is the IP layer which finally interfaces with the Network devices.

3.1.1 Send and Receive Processing

After connection has been established between the sender and the receiver, to trace the data flow from one machine to the other, we describe the processing involved in the `recv` and `send` communication paths.

[1] Source: Linux Kernel Guide (Linux Documentation Project)

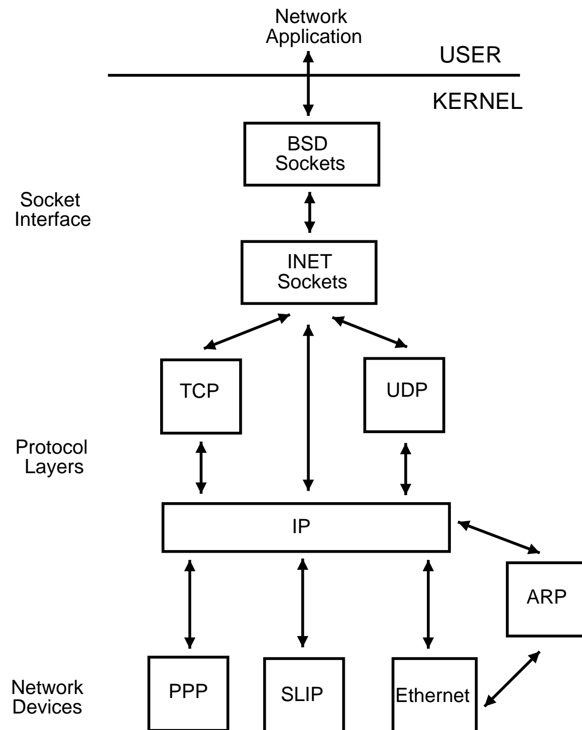

Figure 3.1: Network Processing in Linux

- **Receive Processing:**

  When a packet is received by the network card from the network, it triggers an interrupt. The interrupt handler converts the received packet into an `sk_buff` socket buffer data structure. The `sk_buff` is a data structure used by all the layers of the network subsystem and consists of a control structure and associated memory. The received `sk_buff`'s are then queued on to the `backlog` queue and the network bottom half[2] is flagged as ready to run. The `backlog` queue is a system wide queue where all packets received by the system are queued. When the bottom half handler is scheduled to run, it does the IP and TCP (or UDP) protocol processing and then writes the `sk_buff` on to the received queue for that socket. When the application posts a `recv` system call this data is then copied to the application buffers.

- **Send Processing:**

  When data is transmitted by the application, an `sk_buff` is created based on it. This copy of the application data on to the kernel buffer is made to prevent the data from being

[2]In Linux, "bottom half" refers to the "soft interrupt" part of the interrupt processing.

overwritten by the application before it is sent out. The `send` system call returns at this stage. As the `sk_buff` passes through the various protocol layers, the protocol headers are added on to it. It is then finally queued on to the network interface's transmit queue and finally sent out to the network. In the case of TCP an additional copy is also made to enable retransmission if required.

3.1.2 Components of TCP/IP Processing

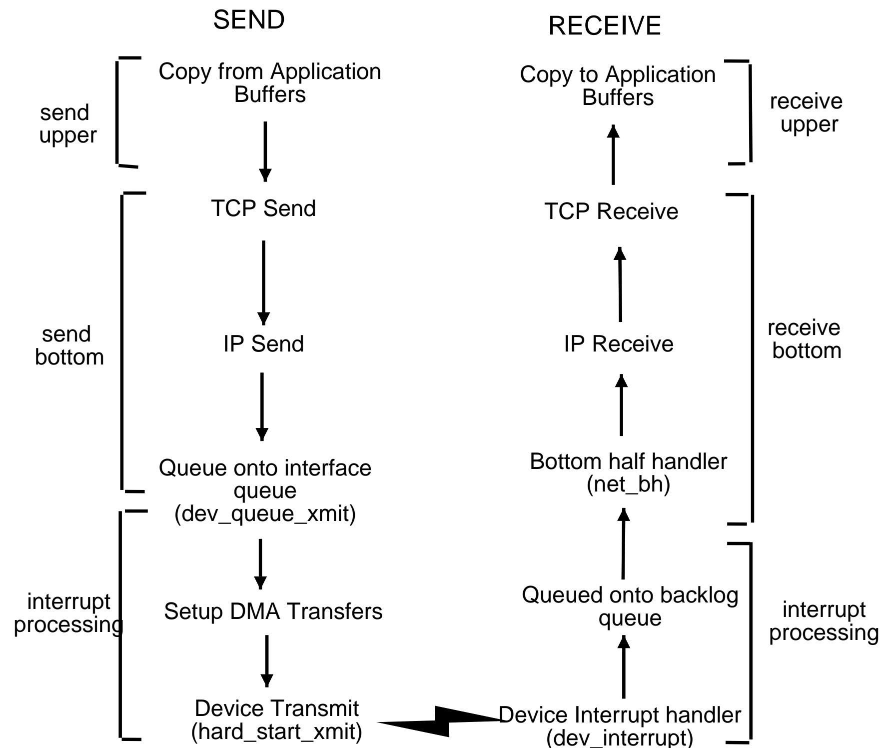

Figure 3.2: Components of TCP/IP Processing

Based on the traditional network processing outlined above we have identified five distinct components of TCP/IP processing. Figure 3.2 shows the breakup of the `send` and `recv` processing into the different components. We describe each of the five components below:

- **interrupt processing:** Each network device deals entirely in the transmission of network buffers from the protocols to the physical media, and in receiving and decoding the hardware generated responses. The **interrupt processing** component includes the time taken to service the *Network Interface Card* (NIC) interrupts and setup DMA transfers. In the path of the `recv` processing, an interrupt is signaled when a packet arrives

at the network interface. The device driver's interrupt handler routine (dev_interrupt) collects the received frames and sets up a sk_buff buffer and writes the packet on to it. It then calls netif_rx which places it on the backlog queue. On the sending side, the packets queued on the interface queue are transmitted out on to the network. The hard_start_xmit function is called passing to it a sk_buff for transmitting. When the buffer has been loaded into the hardware or, in the case of some DMA driven devices, when the hardware has indicated that transmission is complete, the driver releases the sk_buff.

- **receive bottom:** The next component is the receive processing, starting from retrieving packets from the backlog queue all the way up to the socket queue. It does not include the time taken to copy the data from the socket buffers on to the application buffers. Once **interrupt processing** completes, the bottom half handler net_bh dequeues packets from the backlog queue and passes them to the appropriate protocol handler. In the case of IP, it is ip_rcv. IP Receive processing is completed and then TCP Receive processing (tcp_rcv) is called. After the TCP Receive is completed, the sk_buff is queued on to the appropriate socket queue.

- **receive upper:** This refers to the top portion of the receive processing which copies data on to the application buffers. After data has reached the respective socket queue, when the application posts a recv, the data is copied on to the application buffers from the kernel buffers. If the application posts a recv before data is available in the socket queue, the application is blocked and when data arrives the blocked process is woken up and the data then copied on to the application buffers.

- **send upper:** This component refers to the top portion of the send processing which copies the data from the application buffers on to the socket buffers inside the kernel. When the application posts a send, it translates into a call to tcp_sendmsg which tests for certain error conditions, and if the tests succeed, it allocates an sk_buff, builds TCP and IP headers and then copies the data from the application buffers on to sk_buff. If the data is larger than the MSS (Maximum Segment Size) of a TCP Segment, the data is copied on to multiple TCP segments. Alternatively many small data buffers can also be

packaged into a single TCP segment.

- **send bottom:** This component refers to the send processing done *after* copying data on to the kernel buffers. TCP Send processing from the time data is available as `sk_buff`, involves calculating the checksum[3] and adding TCP protocol specific information on to the headers if required. TCP retransmission processing also takes place. Depending on the state of the TCP connection and arguments to the send call, TCP makes a copy of all, some or none of the data and processes them for retransmission. After the TCP Send is complete, IP Send is called. IP's transmit function `ip_queue_xmit` adds IP protocol specific header values (including checksum for IP). After this the packet is ready for transmission and is added on to the interface's queue.

3.1.3 Breakdown of Network Processing Overheads

In Chapter 1, Figure 1.3 we saw that TCP/IP processing takes 71% of the CPU time. In Figure 3.3 we revisit the earlier figure, and show the breakdown of the time spent in network processing based on the components identified above.

It is seen that **interrupt processing** takes 8% of the CPU time. The bottom half processing shown as 12% is a portion of the **send bottom** and **receive bottom**. The remainder of **receive bottom** takes 7%. **receive upper** is a hidden cost accounted with other system calls. **send upper** and **send bottom** take up a substantial amount of CPU time, around 45%. This is because the amount of data sent by a Web server is substantially larger than the amount of data that it receives. This breakdown gives a better understanding of the amount of overhead that each of the TCP/IP processing components result in.

3.2 TCP Server Architecture

TCP Server is a system architecture for offloading network processing from the application hosts to dedicated processors, nodes, or intelligent devices. This separation improves server performance by isolating the application from OS networking and by removing the harmful

[3]Though calculating the checksum is supposed to happen after copying the data, for the sake of optimization, a function `csum_partial_copy_fromuser` is available which carries out both actions in one step.

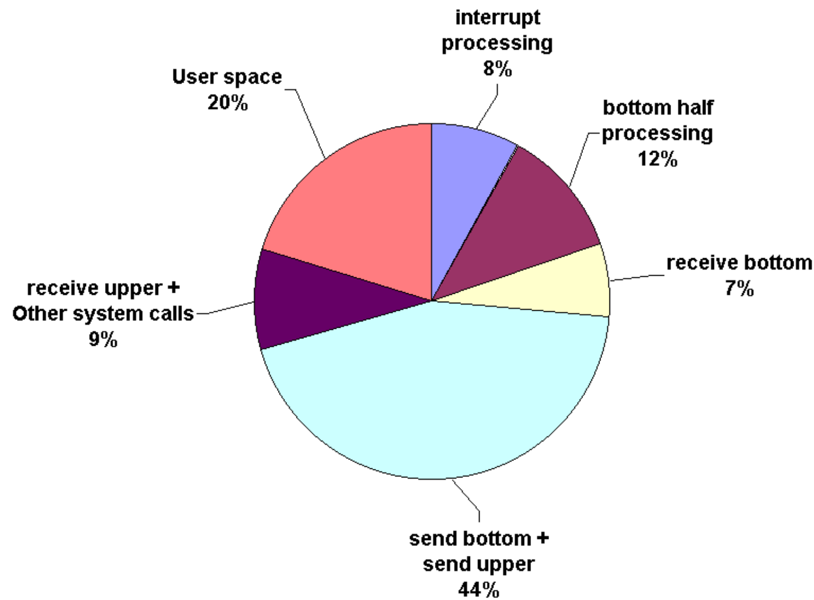

Figure 3.3: Apache Execution Time Breakdown

effect of co-habitation of various OS services. In a cluster of nodes, interconnected by a System Area Network a subset of nodes can be dedicated to network processing. These dedicated nodes are called TCP Server nodes. The rest of the nodes in the cluster, the host nodes can be dedicated to application processing. OS functions not related to network processing are performed on the host nodes. The SAN provides the high speed interconnect for the host to communicate with the TCP Server.

Figure 3.4 shows the TCP Server architecture for clusters. The host and the TCP Server

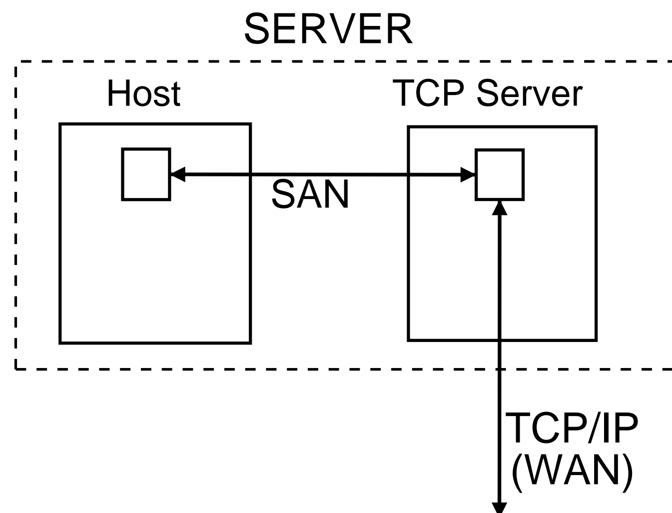

Figure 3.4: TCP Server Architecture

communicate across the SAN using memory-mapped communication. The host can offload the entire TCP/IP processing to the TCP Server, or it can split the TCP/IP processing between itself and the TCP Server.

Figure 3.5 shows the components of TCP/IP processing identified earlier. These components

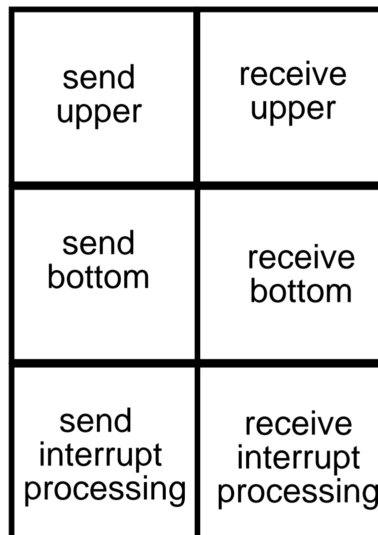

Figure 3.5: Separation of Components of TCP/IP Processing

can be divided between the host and the TCP Server in several different ways. Each of the individual components also interact with the other components. For example, in the case of TCP processing, for every packet sent, a corresponding acknowledgement is received. This requires both **send bottom** and **receive bottom** to share TCP state. The components may also refer to common data structures. Choosing a split of the components between the host and the TCP Server involves a trade-off of the following two factors:

1. The overhead or cost involved in processing a given component at the host.

2. The amount of host to TCP Server communication that offloading the component will result in.

Internet hosts use the socket programming interface to perform network related calls. In the case of the host and the TCP Server communicating across the SAN, components **interrupt processing**, **send bottom** and **receive bottom** can be offloaded to the TCP Server. This can be achieved using the semantics of the traditional socket interface, by intercepting the socket calls

and tunneling the network request across the SAN to the TCP Server. To offload **send upper** and **receive upper** components, the application buffers have to be exported to the TCP Server so that the TCP Server can directly write to and from application data buffers. This requires a modification to the traditional socket programming interface. Internet servers usually have an asymmetric flow of send and receive data volume. The receive data volume is much smaller than the send volume and we therefore expect the benefits due to offloading **receive upper** to be insignificant. In Figure 3.6, we show the sequence of steps involved when the host issues a

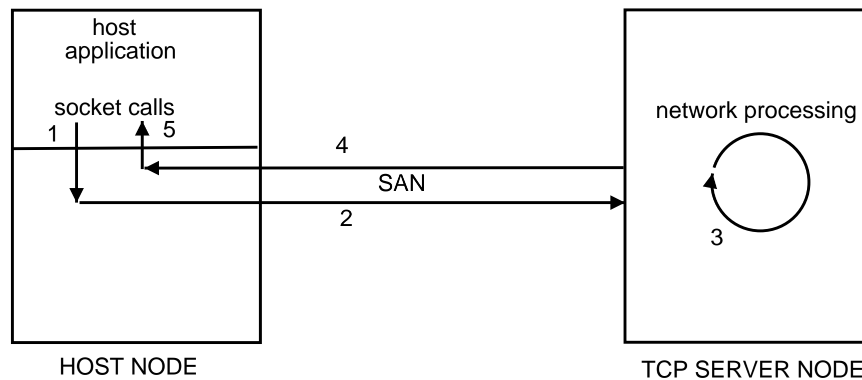

Figure 3.6: Network Processing with TCP Servers

socket call, in the TCP Server architecture.

1. The host application issues a socket call.

2. The socket call is intercepted and the request along with the parameters passed to it are tunneled across SAN to the TCP Server.

3. The TCP Server performs the network related processing for the call.

4. The TCP Server communicates the results of the call to the host.

5. The socket call at the host returns to the application.

The TCP Server node, therefore, is the communication end point for clients connecting across the network.

3.2.1 Scope for Optimization

Deciding a suitable split of the TCP/IP processing between the host and the TCP Server and the actual mechanism for offloading is critical to the TCP Server architecture. In addition, the performance of the TCP Server solution depends on two factors:

1. *The efficiency of the TCP Server:* The TCP Server is responsible for the network processing and therefore, the more efficient the TCP Server implementation, the better the overall server performance.

2. *The efficiency of the communication between the host and the TCP Server:* Since the goal of TCP/IP offloading is to reduce the network processing overhead at the host, using a faster and lighter communication channel for tunneling is essential.

In this section we focus on possible optimizations to these two areas to provide an efficient overall architecture.

Efficiency of the TCP Server: The first set of optimizations target the efficiency of the TCP Server implementation.

- **Avoiding interrupts:** Since the TCP Server performs only TCP/IP processing, interrupts can be beneficially replaced with polling. With this optimization we could potentially free up to 8% of the processing time at the TCP Server. This is the CPU time spent on **interrupt processing** as shown in Figure 3.3. However, the frequency of polling must be carefully controlled, as a very high rate would lead to bus congestion and a very low rate would result in inability to handle all events. The problem is aggravated by the higher layers in the TCP stack having unpredictable turnaround times and by multiple network interfaces.

- **Processing ahead:** Since the TCP Server is dedicated for network processing, idle cycles at the TCP Server can be used to perform certain operations ahead of time (before they are actually requested by the application). The operations that can be "eagerly" performed are the *accept* and *receive* system calls. This will move the cost of performing the operation out of the critical path and will result in lower latencies perceived by the host.

This optimization is beneficial only if the TCP Server has idle time available. It will not prove to be beneficial if the TCP Server is already overloaded with network processing.

- **Eliminating buffering at the TCP Server:** The TCP Server buffers data received from the application before sending it out to the network interface. It is possible to eliminate this extra buffering by having the TCP Server send data out directly from the buffers used for communication with the application host. However, if the **Processing ahead** optimization is used, the TCP Server has to maintain separate buffers for the "eagerly" received data and then copy it on to the buffers used for communication with the host.

Efficiency of Host to TCP Server Communication To improve the efficiency of the interaction between the host and the TCP Server we identify the following optimizations.

- **Bypassing the host kernel:** To achieve good performance, the application should communicate with the TCP Server from user-space directly, without involving the host OS kernel in the common case. Implementing the socket calls as a user-level library using a user-level communication paradigm like VIA will help bypass the kernel in the common case. This can be done without sacrificing protection by establishing a direct *socket channel* between the application and the TCP Server for each open socket. This is a one-time operation performed when the socket is created, hence the socket call remains a system call in order to guarantee protected communication.

- **Asynchronous socket API:** Tunneling the socket calls across the SAN may increase the latencies perceived by the application compared to a socket call that traps into the kernel and then returns. Asynchronous socket calls provide a mechanism for the application to hide the latency of a socket operation by overlapping it with useful computation. By using asynchronous socket calls, the application can exploit the TCP Server architecture to avoid the cost of blocking and rescheduling.

- **Avoiding data copies at the host:** To achieve this, the application must tolerate the wait for end-to-end completion of the send, i.e., when the data has been successfully received at the destination. If this is acceptable, the TCP Server can completely avoid data copying on a send operation. For retransmission, the TCP Server may have to read the data again

from the application send buffer using non-intrusive communication. Pinning application buffers to physical memory may be necessary in order to implement this optimization. In Figure 3.3, **send upper** and **send bottom** take about 45% of processing time. A substantial part of this processing time is due to the data copies involved. With this optimization we could potentially free up this processing time and use it for application processing. Combining this optimization with **Asynchronous socket API** may prove beneficial to the host.

- **Dynamic load balancing:** Depending on the application workload, it is possible that either the TCP Server or the host can get saturated. Network intensive work loads may saturate the TCP Server, whereas computation intensive work loads may saturate the host. An adaptive scheme to balance the load or resource allocation between the host and TCP Server will help improve overall server performance. While the host and the TCP Server can communicate with each other and share load information, incoming client requests will need to be redistributed between them based on this load information. This will require a front end request distributor, to which the current load information will have to be communicated. The host application will then have to direct socket calls to its own kernel or to the TCP Server to achieve uniform load balancing.

Chapter 4

Design of TCP Servers

4.1 Design Alternatives

In the TCP Server architecture, the host offloads the network related socket calls to the TCP Server. This offloading of socket calls leads to user-space vs. kernel-space design choices at the host and the TCP Server.

4.1.1 Socket Call Processing at the Host

At the host, a socket call is intercepted and the call is then tunneled across the SAN to the TCP Server. The socket call at the host can be intercepted after it traps into the kernel or directly from user space. As discussed earlier, under the optimization **Bypassing the host kernel**, it is advantageous for the application to communicate with the TCP Server in user-space, without involving the host kernel. Hence a user-level communication library is provided to host applications, which enables them to communicate with the TCP Server bypassing the host kernel in the common case.

4.1.2 TCP Server

Using a user-level communication library at the host, there are several alternatives for designing the TCP Server. Figure 4.1 shows the different design choices available. The two primary tasks of the TCP Server are to communicate with the host and to perform network processing. The small boxes in the figures indicate the data buffers.

- Figure 4.1(a) shows Alternative 1. The communication between the host and the TCP Server takes place in user-space. Once data from the host has reached the TCP Server node, network processing proceeds as in a traditional system, involving a trap into the

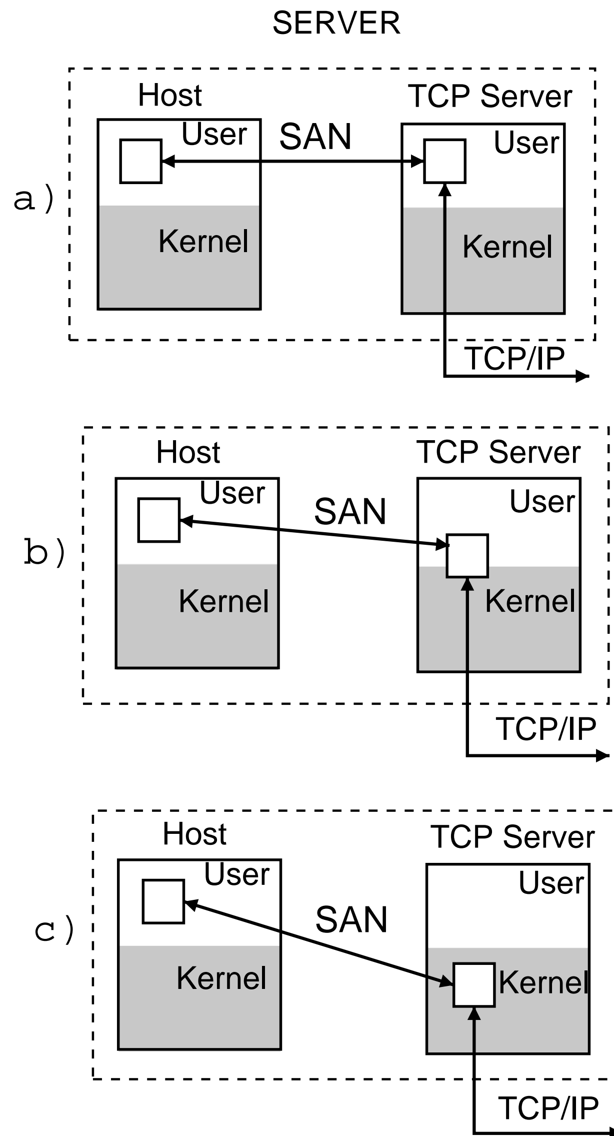

Figure 4.1: Alternatives for TCP Servers over Clusters

kernel. This alternative does not require any modifications to the native network subsystem on the TCP Server node.

- Figure 4.1(b) shows Alternative 2. As in Alternative 1, the communication between the host and the TCP Server takes place in user-space. Here too, the network processing proceeds as in a traditional system, involving a trap into the kernel. However, the buffers used for host to TCP Server communication are shared with the kernel and are also used for network processing. This requires the network subsystem on the TCP Server node to be modified. These buffers have to be pinned in memory and the mapping exported

to the kernel. Referring to the optimizations discussed earlier, **Eliminating buffering at the TCP Server** is achieved by this alternative.

- Figure 4.1(c) shows Alternative 3. On the TCP Server node, the communication between the host and the TCP Server takes place in the kernel. The kernel on the TCP Server node is modified such that the host to TCP Server communication and the network subsystem share common buffers. **Eliminating buffering at the TCP Server** is also achieved by this alternative.

The rest of this thesis elaborates on a TCP Server design and implementation using Alternative 1. The host to TCP Server communication takes place in user-space and the network subsystem on the TCP Server node is unmodified. Since the network subsystem on the TCP Server node is unmodified, traditional socket calls are used by the TCP Server to perform network processing.

4.2 Network Processing Mechanism

Based on the design alternatives discussed above, we present a more detailed version of the earlier Figure 3.6. Figure 4.2 shows the Network processing mechanism using TCP Servers.

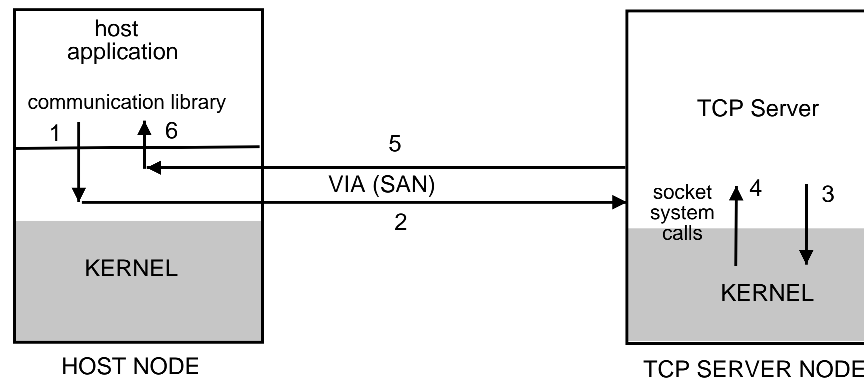

Figure 4.2: Network Processing with TCP Servers

1. The host application issues a socket call using our communication library.
2. The socket call is intercepted in user-space and the request along with the parameters passed to it, is tunneled across SAN to the TCP Server.

3. The TCP Server interprets the call and executes the appropriate socket call using the traditional BSD socket library.

4. The socket call is completed and control returns to the TCP Server.

5. The results of the socket call are sent back to the host.

6. The library call at the host returns to the application.

4.3 Host to TCP Server Communication using VIA

The communication between the host and the TCP Server across the SAN plays an important role in the design of TCP Servers for clusters. The host to TCP Server communication in our design is based on the Virtual Interface Architecture (VIA) [22]. In this section we provide an overview of the Virtual Interface Architecture (VI Architecture), and the data transfer models provided by it.

The VI Architecture is a memory-mapped user-level communication model that bypasses the kernel from the common communication path. The VI Architecture specification [18] was developed as a joint effort by Compaq, Intel and Microsoft. It is based on previous academic research on user-level communication including U-Net [8], Active Messages [23] and VMMC [15]. In the VI Architecture each consumer process is provided with a protected, directly accessible interface to the network hardware - a Virtual Interface (VI) which represents a communication endpoint. This is different from the traditional network architecture, in which the host operating system is responsible for managing the network resources and multiplexes accesses to the network hardware across process-specific logical communication end points. In the traditional architecture, every network access, involves a trap into the kernel increasing message latencies, which is avoided by the VI Architecture.

Several hardware and software implementations of VIA are available. Giganet [28] has a hardware implementation of the VI Architecture on its proprietary cLAN network interface card. Software emulations are available on ServerNet [47] and Myrinet [9, 39]. M-VIA [40] provides Linux software VIA drivers for various fast Ethernet cards.

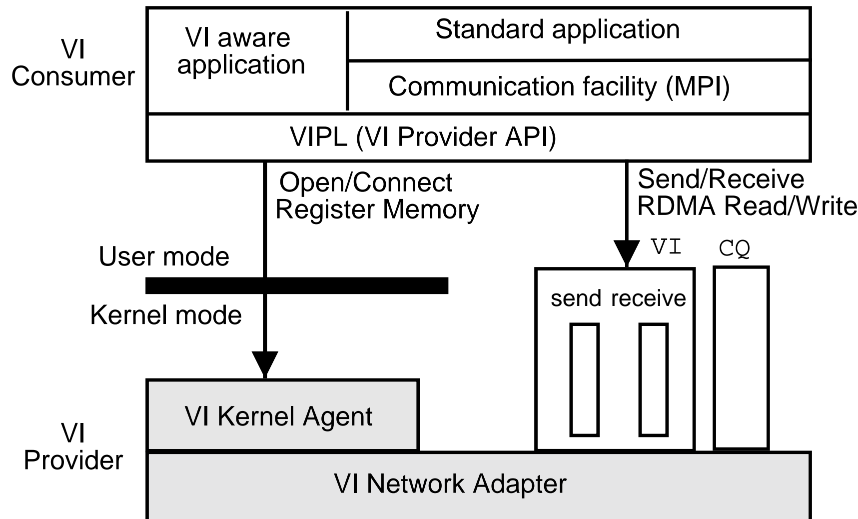

Figure 4.3: The VI Architecture Model

4.3.1 VI Architecture Overview

Figure 4.3 shows the details of the VI Architecture. The VI Architecture[1] is made up of four major components: Virtual Interface, Completion Queues, VI Providers and VI Consumers. The VI Provider consists of the VI network adapter and a kernel agent device driver. The VI consumer is the application that is the end user of the VI. The VI Consumer can be a "VI-aware" application which directly uses the VIPL (Virtual Interface provider Library) or can be a standard application which uses a communication facility like MPI [41] which internally uses the VIPL. The VI is the mechanism that allows a VI Consumer to directly access the VI Provider to perform data transfers. It acts as a communication end point similar to sockets in the TCP context. The VI is bi-directional and supports point-to-point data transfer. A VI consists of a pair of work queues, the send and the receive queue. VI Consumers format requests into descriptors and post them on to the work queues. A descriptor contains all the information required by the VI Provider to process this request including pointers to data buffers. VI Providers asynchronously process the posted descriptors and mark them as complete. The VI Consumer can either poll or wait for a descriptor to be completed. Once completed, the descriptor can be reused for consequent requests. A completion queue provides a single point for a VI Consumer to combine descriptor completion notifications of multiple VIs.

[1]The following discussion is based on the Virtual Interface Architecture Specification Version 1.0

Connection Mechanism:

The VI Architecture provides a connection-oriented data transfer model. A VI Consumer must connect a local VI to a remote VI before data can be transferred. The connection process follows a client-server model. The server side waits for incoming connection requests from clients and can either accept or reject them based on the remote VI's attributes. A process can maintain several connected VIs to transfer data between itself and other processes on different systems. After the data transfer is complete, the associated VIs must be disconnected.

Memory Registration:

The VI Architecture requires the VI consumer to register all memory used for communication prior to submitting a request. Memory registration enables the VI Provider to transfer data directly to and from the buffers of a VI Consumer and the network without any intermediate copy. The memory registration process locks the pages of a virtually contiguous memory region into physical memory and the virtual to physical memory translations are exposed to the VI NIC. The VI Consumer gets an opaque handle for each registered memory region. Using this handle and the virtual address, the VI Consumer can reference all registered memory. The process also allows the VI Consumer to reuse the registered memory.

Data Transfer Models:

The VI Architecture provides for two kinds of data transfer:

1. The traditional Send/Receive model to transfer data to and from communicating end points. The send and receive operations, complete asynchronously, and the descriptor completion has to be tracked by the VI Consumer. The receiving side has to pre-post a receive descriptor with buffers large enough to hold the incoming data.

2. The Remote Direct Memory Access (RDMA) model provides for remote data reads and writes. This allows a process to read from and write to the memory of a remote node without any involvement on the part of the remote node. To facilitate this the destination memory address and registered memory handle need to be exported to the source prior to the RDMA.

In the VI Architecture, kernel mode traps occur at the time of VI creation and destruction, connection establishment and connection tear-down, memory registration and deregistration.

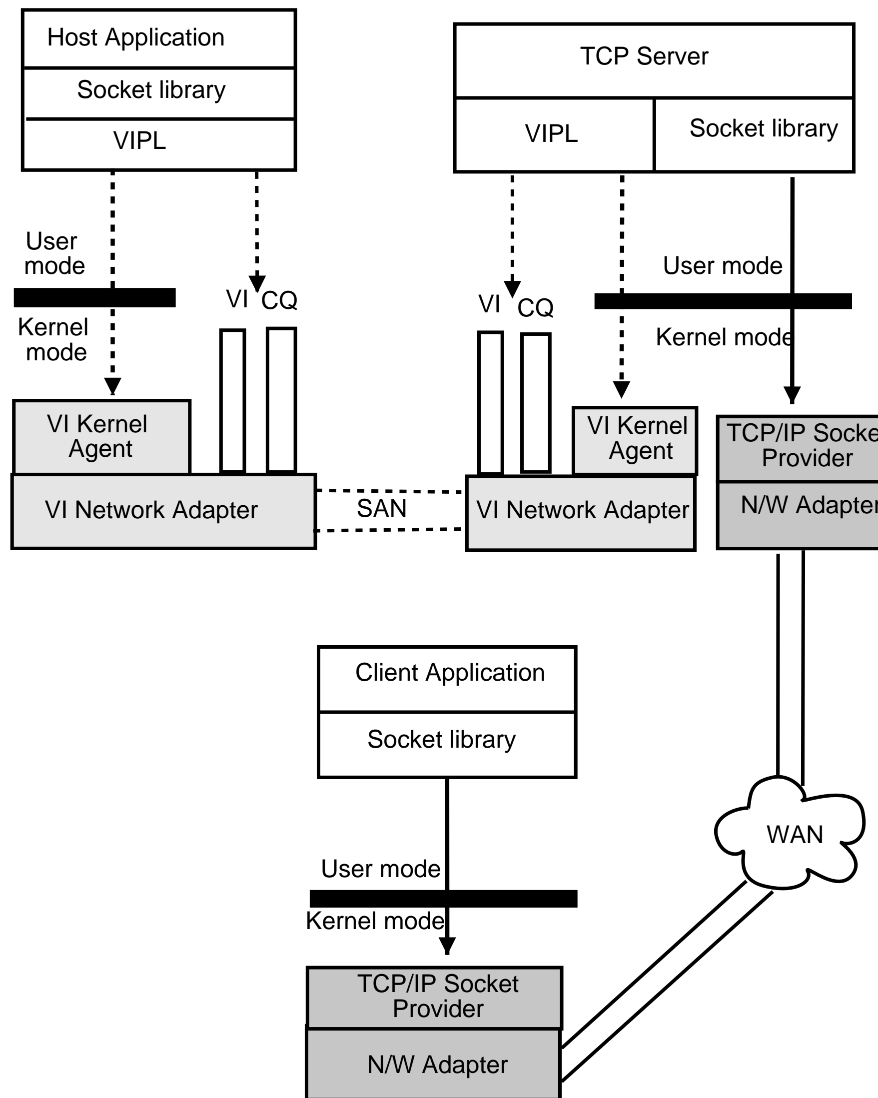

Figure 4.4: Host and TCP Server Communication using VIA

All other communication proceeds without any kernel involvement.

TCP Server Architecture Based on VIA

Figure 4.4 shows the role of VIA in the TCP Server architecture. The dotted lines represent communication across the SAN and the solid lines represent the communication across the WAN to the client(s). The key features of the architecture are:

1. The TCP/IP processing has been offloaded from the host node and the host kernel has been freed from TCP/IP network processing. The host node therefore has more resources to devote to application processing related tasks.

2. The socket library at the host is executed as a light-weight *Remote Procedure Call* at the TCP Server node.

3. The host and TCP Server node use memory-mapped communication across a high-speed interconnect - the System Area Network.

4.4 Mapping Sockets to VIs

Sockets are the natural representation of the communication end points for hosts across the network. Participating hosts use the socket as the identifier to establish a communication channel between them and then to send and receive data on the established channel. For intra-SAN communication, the VIs are the natural communication end points. Between the host node and the TCP Server node the host node requests VI connections from the TCP Server node and once communication is established, data can be transferred between them using it. Using a single VI to channel all socket calls from the host node to the TCP Server node will severely limit parallelism and will adversely affect the latency of each socket call. Therefore multiple VIs need to be established between the host and the TCP Server node. If multiple VIs are available, the socket calls on the host node need to be mapped on to the available VIs. Mapping of socket call requests to VIs is linked to the design of the TCP Server and the order in which it processes incoming requests. Since socket calls can be blocking, the TCP Server needs to be multi-threaded to enable parallel processing of requests. This leads us to several design choices which are explained below

4.4.1 Alternatives Based on Request Processing at the TCP Server

- Figure 4.5 shows the design alternative 1. At the TCP Server there is one thread per VI, which processes the incoming requests on a given VI in order. Thread1 processes calls received on VI1, Thread2 on VI2 etc. To channel a given socket call the host node selects any available VI at random. Such an arrangement will result in unfair processing delays. Request B for socket2 will be processed only after Request A for socket1 is completed. In fact, A could be a blocking call, e.g., `recv` in which case Thread1 could be blocked for quite a while before it is able to process the request B.

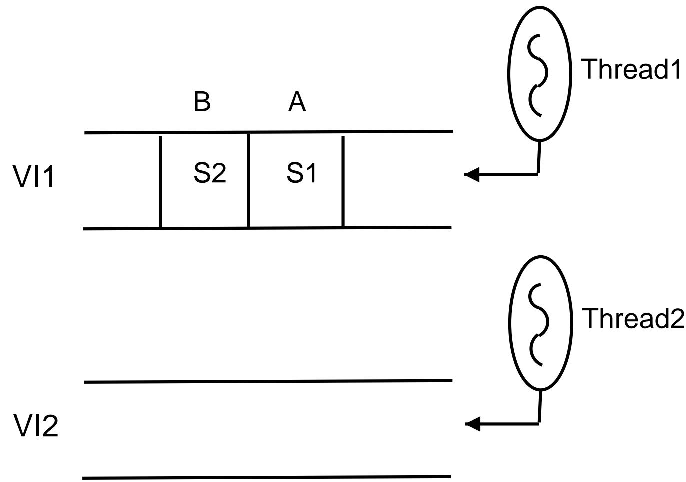

Figure 4.5: Design Alternative 1

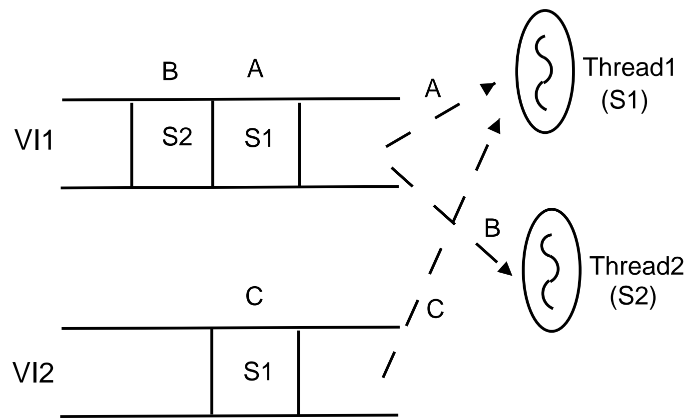

Figure 4.6: Design Alternative 2

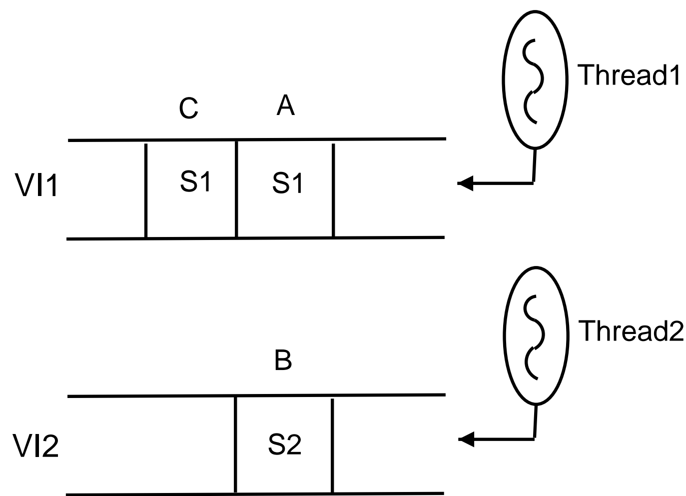

Figure 4.7: Design Alternative 3

- Figure 4.6 shows the design alternative 2. At the TCP Server there is one thread per socket instead of one thread per VI. Incoming requests on all VIs are parsed and then channeled to the appropriate thread for processing. Thus Thread1 will handle requests A and C and Thread2 will handle B. This avoids the problem faced by Design 1. Under synchronous call operations, control is returned from the library call only after the corresponding socket call is completed at the TCP Server node. Hence if the host application issues A, control will return to it only after A has been processed by the TCP Server. After this if the application issues C, then the processing order at the TCP Server will be the correct processing order intended by the host application. However, in the case of asynchronous calls control returns to the host application as soon as the request is queued for dispatch on the VI. In this case it is possible that C, transferred using a different VI will arrive earlier than A and may be processed before it. Asynchronous calls are described in more detail in Section 4.6.1.

- Figure 4.7 describes design alternative 3 in which case the host application maintains the socket to VI mapping and ensures that all socket calls for a given socket are sent on the same VI. This will avoid the problems discussed with the above two designs. In this case, at the TCP Server side there is one thread per VI. Since the host will send all calls relating to a socket on a given VI, this is equivalent to one thread per socket.

4.4.2 Communication Library at Host Node

The communication library at the host node is responsible for maintaining and managing the VIs and for directing the socket calls on to the correct VI. This happens transparent to the host application. In the discussion in Section 4.3, it was seen that the expensive VI operations that involve the kernel, are the creation of VIs, establishing connection on the VIs and registering the memory required for SAN communication. It is essential to avoid these costs in the common data path. Drawing a direct parallel between the VI and the socket life cycle, at the time of `socket` call, a VI can be created and a connection established with the TCP Server. At the time of the `close` socket call, the VI can be disconnected and destroyed. This simple parallel involves unnecessary creation and destruction of VIs. To avoid the overheads of creating and

destroying VIs, a pool of connected VIs is maintained. At the time of `socket` call, the free pool is first checked for an available VI, and if none exists, then a new VI is created and a connection is established on it. At the time of VI creation the memory regions to be used in the transfer are also allocated and registered. At the time of `close`, the VI is added to the free pool.

In a traditional Web server architecture, the *listen socket* is opened with the `socket` call. Incoming client connection requests are accepted using the `accept` call, which returns a new socket descriptor associated with this new connection. In the TCP Server architecture, the association of the VI to socket is done at the time of the `socket` call for the listening socket and for the newly accepted connections the association can be done either at the time of the `accept` call itself or can be postponed until the time of actual use of the new socket, i.e., until a socket call (typically `recv` or `send`) is actually issued on the new socket descriptor. The trade-offs with either association are discussed in Section 4.6.4.

4.5 TCP Server Components

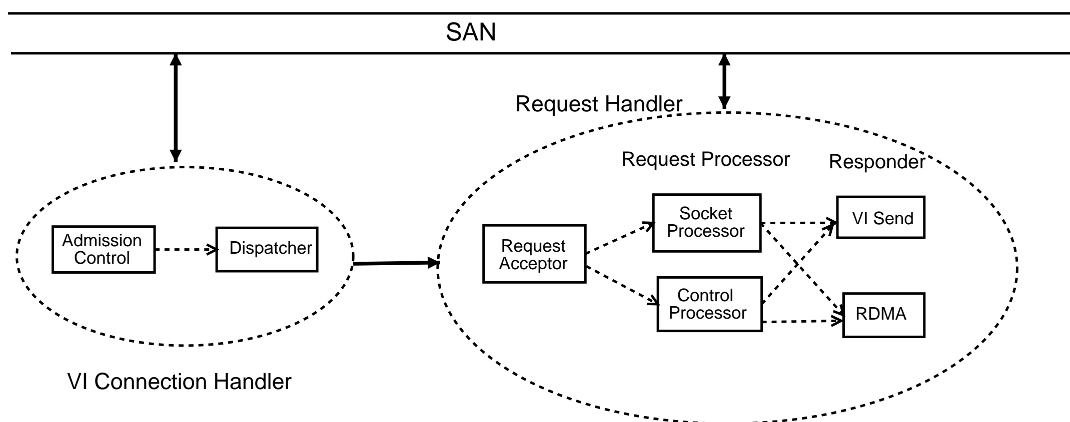

Figure 4.8: Components of the TCP Server

Figure 4.8 shows the modules that constitute the TCP Server. The *VI Connection Handler* is made up of an *Admission Control* module and a *Dispatcher*. Incoming VI Connection requests are handled by the *VI Connection Handler*. Depending on the current load, i.e., the number of VI connections that have already been accepted, the *Admission Control* module either accepts or rejects the incoming connection request. If the request is accepted it is then forwarded to the *Dispatcher*. The *Dispatcher* ensures that there is a *Request Handler* available to handle further

requests on this VI. All requests on the VI are henceforth directly handled by the designated *Request Handler*. The *Request Handler* consists of the *Request Acceptor* module which is responsible for accepting incoming requests on this VI. The *Request Processor* module identifies the kind of request. If the request is for a socket related operation the *Socket Processor* executes the corresponding socket call using the native BSD socket system calls. Requests could also be internal setup/control related in which case the *Control Processor* handles the request. The results of the *Request Processor* are directed to the *Responder* module. Depending on the nature of the request the response is, either through the regular `VI Send` format handled by the *VI Send* module, or through non-intrusive communication handled by the *RDMA* module. In addition to these basic modules, eager processing performed on a per socket basis will be executed in the context of the *Request Handler*. Some kinds of eager processing are system-wide.

4.6 Optimizations

In Chapter 6 we present the performance of the base TCP Server architecture and identify the additional gains that result due to the optimizations. To provide a basis for that discussion, we present the design of the optimizations in this chapter. In this section, we first describe the steps involved in processing a synchronous call. For optimizations that directly influence the socket call processing, we provide a time line representation of their execution. Each of the optimizations is explained in isolation, but can easily be combined together.

Synchronous Calls:

The synchronous calls follow the standard outline described in Section 4.2. Figure 4.9 shows the time line representation of the synchronous calls. The `accept` is issued at the host, after it reaches the TCP Server and is interpreted by the TCP Server, the BSD `accept` is issued at the TCP Server. After it returns, the results of the call are sent back to the host node and then the host node's `accept` call returns to the host application. The same sequence happens for the `recv` and `send` socket calls. This is used as the baseline for comparison with the other calls.

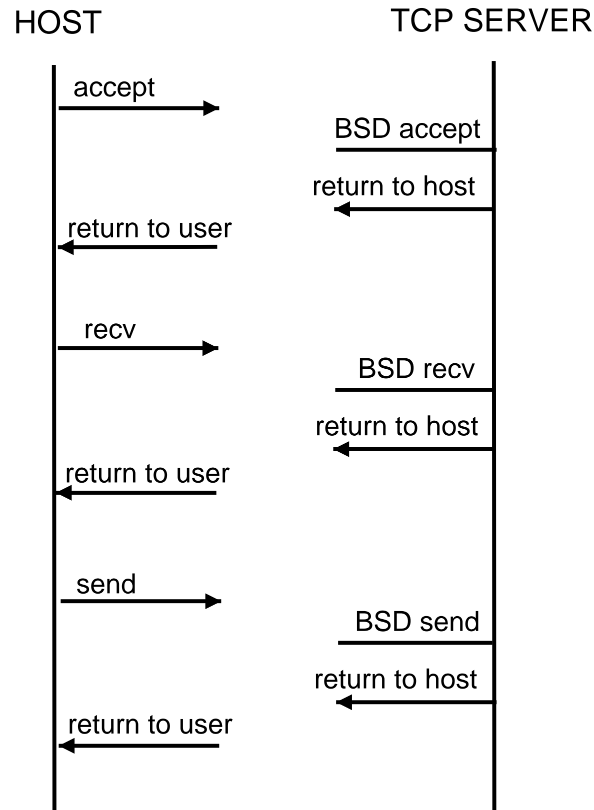

Figure 4.9: Synchronous Call Processing

4.6.1 Asynchronous Send

The volume of data sent out by Web servers is typically larger than the data received. We therefore focus our attention on optimizing the send, by providing an asynchronous send primitive to the applications. In the case of the *asynchronous send*, the `send` call returns to the user as soon as the request is queued at the host. To ensure that these sends are not lost, the host should only enqueue as many sends as the number of outstanding receive descriptors posted by the TCP Server. In addition the host should also be informed of the results of the previous sends by the TCP Server. Since the library call has already returned at this point there are no receive descriptors posted by the host to receive the results of the call. The TCP Server does a non intrusive silent write (RDMA) of the results of the previous sends to the host and of the outstanding number of receive descriptors available at its end. To enable RDMA, the host has to export the buffers in which it expects the RDMA results and also initiate the flow control for a new socket. Figure 4.10 shows the time line representation in the case of the *asynchronous send*. Since the same VI can be reused for several sockets, at the time of the first `send` on this

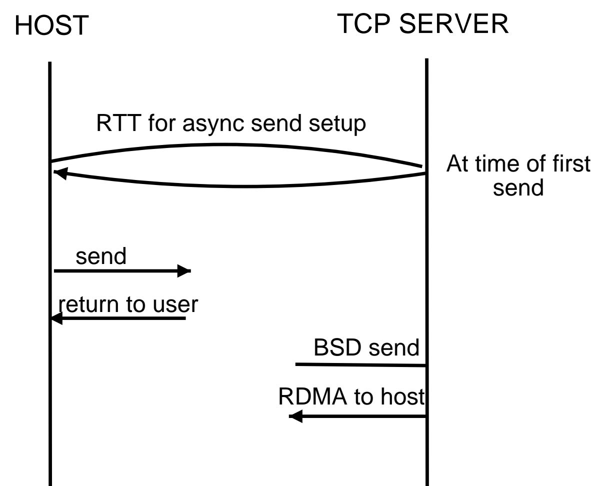

Figure 4.10: Asynchronous Send Processing

socket, the host has to export its RDMA enabled buffers to the TCP Server and also indicate to the TCP Server that it is a new socket association that has started now. This is shown by the additional *Round Trip Time* RTT at the time of the first *asynchronous send* on this socket. Consecutive sends can then proceed asynchronously, by the host verifying the status from its RDMA buffers.

4.6.2 Eager Receive

In Section 3.2.1, *eager receive* was described as part of the **Processing ahead** that the TCP Server could do. A *push-based* or a *pull-based* approach can be taken in *eager receive* processing. Figure 4.11 shows the *pull-based* approach. At the time of the first `recv` for a given socket, the host library indicates to the TCP Server to start *eager receive* on behalf of this socket. This indication has to be sent to the TCP Server once per socket because different sockets share the same VI and the TCP Server has to know the socket descriptor for which *eager receives* have to be posted. The *eager receive* optimization does not require any modification to the socket API. The communication library at the host internally formats the request along with the parameters required for the setup. After this setup, for consecutive `recv`s the `recv` happens at the TCP Server before the host posts a `recv`. When the host posts a `recv` if data has already been received by the TCP Server it is returned to the host. The *push-based* approach, as shown in Figure 4.12 goes one step further and the TCP Server sends any eagerly received data all the

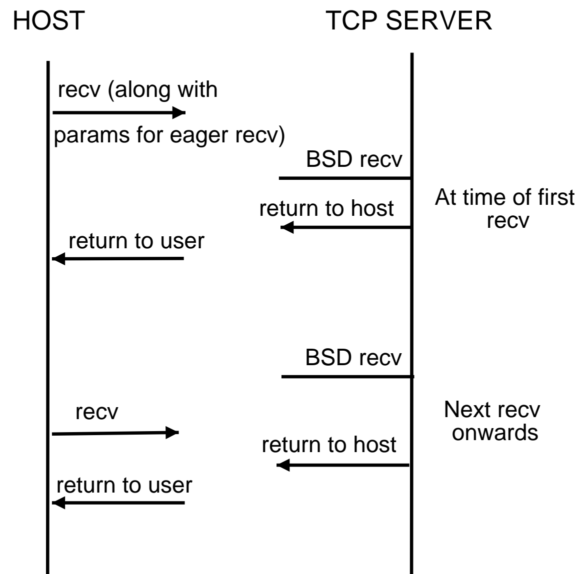

Figure 4.11: Eager Receive (pull-based) Processing

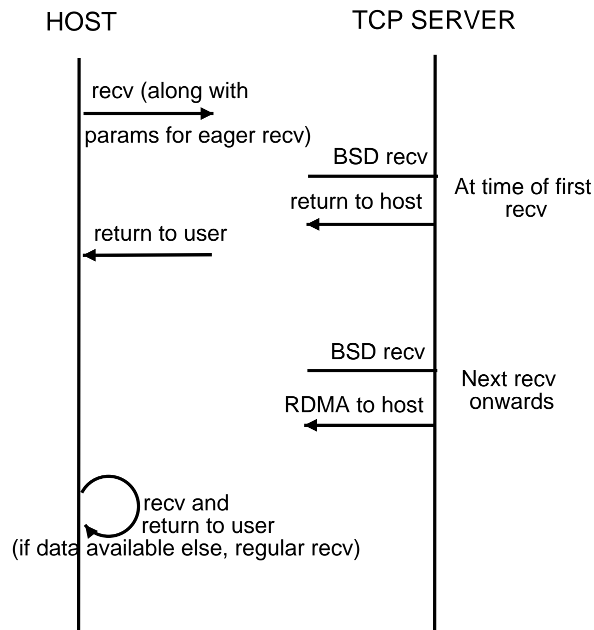

Figure 4.12: Eager Receive (push-based) Processing

way to the host. Thus when the host application posts a `recv`, data can be returned from the local buffers if it already has been eagerly received.

4.6.3 Eager Accept

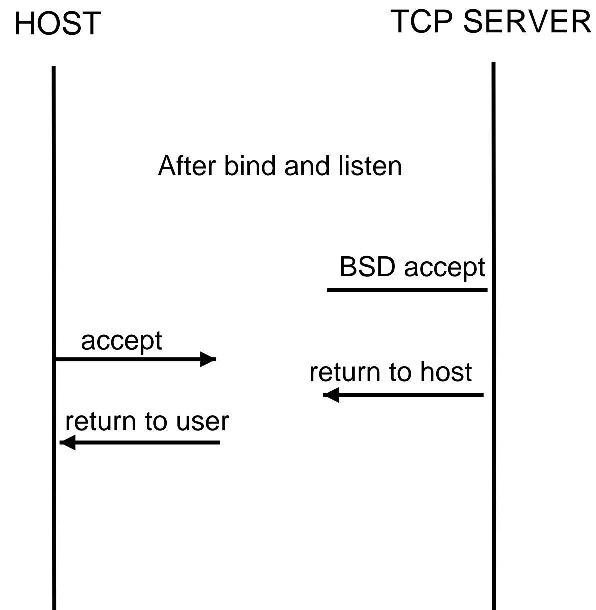

Figure 4.13: Eager Accept Processing

In the case of *eager accept*, the TCP Server eagerly posts `accept` and accepts new TCP connections on behalf of the host. These connections will be returned to the host as and when the host application posts the `accept` call. This is shown in Figure 4.13. The TCP Server can also perform admission control and restrict incoming client connections without affecting application processing at the host.

4.6.4 Setup with Accept

The association of sockets to VI is done at the time of the `socket` call or in the case of accepted connections, it can be done at the time of the `accept` call or can be put off until the time of actual use. In certain host applications, it is possible that the parent process accepts calls and then a child process handles the actual communication on the socket. If the mapping of socket to VI is done at the time of the `accept` call then the VI will be in the parent's process address space and has to be shared with the child process. To accommodate such host applications it is

simpler if the mapping between the socket to VI is done at the time of the first socket operation on the new socket descriptor, in which case the VI will be in the child process' address space. Another important design consideration is that the maximum number of available VIs is limited. In cLAN the hardware provides for a maximum of 1024 VIs. These have to be multiplexed with the number of socket connections concurrently opened. Delaying the mapping of socket to VI to the time of first use on the socket will help increase the concurrency. However, it is seen that both *asynchronous send* and *eager receive* require some form of setup to inform the TCP Server to process in the context of the new socket that is now associated with the current VI. This is done at the expense of an additional RTT in the case of *asynchronous send*, or at the time of the first `recv` call in the case of *eager receive*, in which case the eager processing can continue only from the next receive onwards.

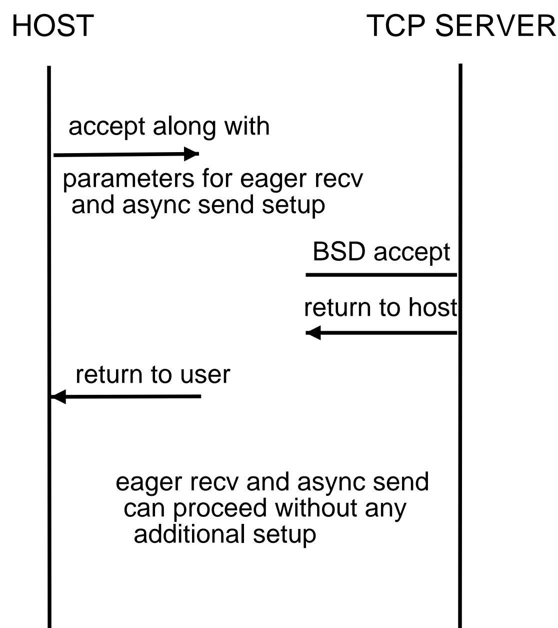

Figure 4.14: Setup With Accept

This setup information can be sent along with the `accept` call, provided the VI to socket association is done at the time of the `accept` call itself. The feature of *setup with accept* associates the VI to the new socket that will be returned by the `accept` and therefore sends the necessary information to setup *asynchronous send* and *eager receive* for this new socket. Figure 4.14 shows this. It is to be noted that after the `accept` call *asynchronous send* and *eager receive* can proceed with no additional cost.

4.6.5 Avoiding Data Copies at the Host

The buffers used for VI communication between the host and the TCP Server have to be registered. The host application uses its own data buffers for the various socket calls. To transfer application data from the host, across the SAN to the TCP Server, the host library maintains a set of registered buffers which it uses for VI communication. The host application's data is copied on to these buffers and then transferred to the TCP Server. In the case of `recv`, data is received on to these buffers and then copied on to the application data buffers. This additional copy may add significant overheads especially in the case of the `send` socket call, where large amounts of data are usually transferred. An alternative would be to register the host application data buffers in the context of each call. This will include the cost of memory registration of the data buffers in the critical path. This will involve a kernel transition in each call which will be expensive.

To ensure that no additional copy/memory overheads are introduced by the TCP Server architecture, additional APIs are provided to the host application. These APIs allow the host application to pre-register a set of data buffers that it will use for communication, and then use these buffers for the `send` and `recv` calls. However, this optimization requires a co-operative application which is aware of the underlying TCP Server architecture. The application is responsible for buffer management of these pre-registered data buffers.

Chapter 5

Implementation

In this chapter we describe our implementation of TCP Servers. We describe the details of processing at the host and at the TCP Server. We also explain the optimizations that we have implemented. Our implementation was developed using the hardware VI architecture provided by Giganet. Appendix A lists the primary VI primitives used in our implementation.

5.1 Application Programming Interface

The host application is provided with a programming interface to communicate with the TCP Server for network related processing. This interface has to be robust and easy to use. In this section we list the interfaces provided by our implementation. The interfaces can be divided into the following categories:

- **Traditional socket calls:** These calls are analogous to the BSD socket call interface.

```
int tcps_socket(int domain, int type, int protocol);
int tcps_bind(int socketfd, struct sockaddr *paddr, int addrLen);
int tcps_listen(int socketfd, int backLog);
int tcps_accept(int socketfd, void *paddr, int *paddrLen);
int tcps_connect(int socketfd, const struct sockaddr *paddr, int addrLen);
int tcps_send(int socketfd, const void *msg, int len, int flags);
int tcps_recv(int socketfd, void *buf, int len, unsigned int flags);
int tcps_close(int socketfd);
```

- **Pre-registered Buffers:** These calls are provided to the host application to enable them to pre-register the buffers used for SAN communication. These calls help avoid the

overheads of additional copy from the host nodes to the TCP Server.

Memory Registration:

```
int tcps_register_memory(void *pmemory, int length,
       NS_MEM_HANDLE *pmemoryHandle);
int tcps_deregister_memory(void *pmemory, NS_MEM_HANDLE memoryHandle);
```

Synchronous calls with registered buffers:

```
int tcps_send_registered(int socketfd, const void *msg, int len,
       int flags, NS_MEM_HANDLE memHandle);
int tcps_recv_registered(int socketFd, void *buf, int len,
       unsigned int flags, NS_MEM_HANDLE memHandle);
```

Asynchronous calls with registered buffers:

```
int tcps_send_async_registered(int socketfd, const void *msg, int len,
       int flags, NS_MEM_HANDLE memHandle, NS_IO_RESULT *pres);
int tcps_io_done(NS_IO_RESULT *pres);
int tcps_io_wait(NS_IO_RESULT *pres, unsigned long timeOut);
```

5.2 Host End Point

The communication library at the host is responsible for creating and maintaining VIs. It sends VI Connection requests to the TCP Server to establish a connection between its local VI and a remote VI at the TCP Server. To improve performance, a configurable number of VIs are created and connected at the time of initialization. If additional VIs are required they are created on-demand. A corresponding exit routine destroys the VI connections. The library also manages the socket to VI mapping for the lifetime of a socket. When the application issues a socket call, using the provided APIs, the communication library retrieves the associated VI (or associates one if none is associated as yet). It then packages the request into the format required for VI communication and then sends the request to the TCP Server. On receiving a response from the TCP Server, it unpacks the response and returns the result of the call to the

host application. In this section, we identify the key data structures required by the host and their role in this communication. Figure 5.1 shows the organization of information at the host. The key data structures are described below:

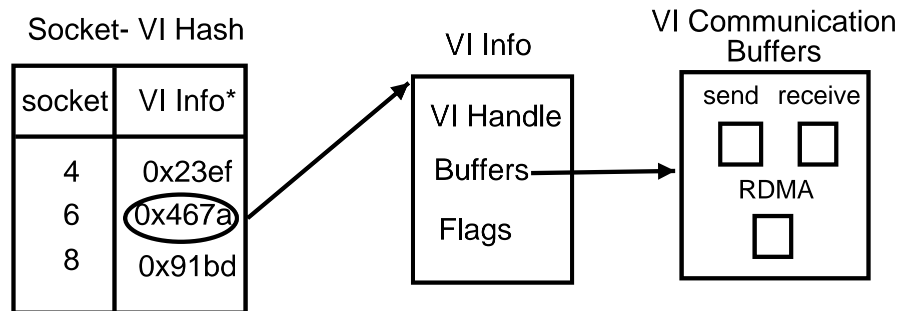

Figure 5.1: Socket to VI Mapping at the Host

- `ViInfo` : The `ViInfo` is the primary data structure used at the host to maintain all the information required for communication on a given VI. It contains the send and receive descriptors associated with this VI and the `HostRDMASend`, a data structure which maintains information required for RDMA communication. The key fields of the `ViInfo` data structure are shown below

```
typedef struct
{
   VIP_VI_HANDLE viHandle; /* identifier of the VI */
   /* descriptor used for asynchronous sends*/
   VIP_DESCRIPTOR *psendExtraDescr;
   /* send descriptor used for regular calls */
   VIP_DESCRIPTOR *psendDescr;
   VIP_DESCRIPTOR *precvDescr; /* receive descriptor */
   VIP_MEM_HANDLE memoryHandle; /* registered memory handle */
   HostRDMASend *pRDMASend; /* RDMA related information */
   .......
}ViInfo, *pViInfo;
```

- Registered Buffers: For a given VI, all the required memory is allocated in contiguous blocks and are registered under a single `memoryHandle`. This includes the memory used for the descriptors and for the buffers used for transferring data. The descriptors contain pointers to the start of the memory buffers used for sending and receiving data. The registered memory is actually unrelated to a given VI. The memory used can be maintained as a separate pool. Each time a free memory buffer can be allotted from this pool for communication on a given VI, and the descriptors altered to point to the relevant memory region. In our design, for convenience we associate the memory pointers with the VI in the `struct ViInfo`.

- VI Free List and Hash table : `FreeViList` maintains the connected VIs that are free and currently unused. When a given `ViInfo` is associated with a socket, it is removed from the `FreeViList` and added to a hash table wherein each entry is in the form of a `SocketVi`. Every consequent socket call uses the hash table to retrieve the VI corresponding to the given socket. At the time of `close` the `ViInfo` is removed from the hash and re-inserted into `FreeViList`.

```
typedef struct _FreeViList FreeViList, *pFreeViList;
struct _FreeViList
{
   pViInfo viInfo;
   pFreeViList next;
};
typedef struct _SocketVi SocketVi, *pSocketVi;
struct _SocketVi
{
   int socketFd;
   pViInfo vi;
   /* each hash entry is a linked list to accommodate collisions */
   pSocketVi next;
};
```

5.3 Request/Response Protocol

The host and the TCP Server follow a pre-determined request/response protocol, as shown in Figure 5.2. The requests sent by the host, identify the network operation requested by the

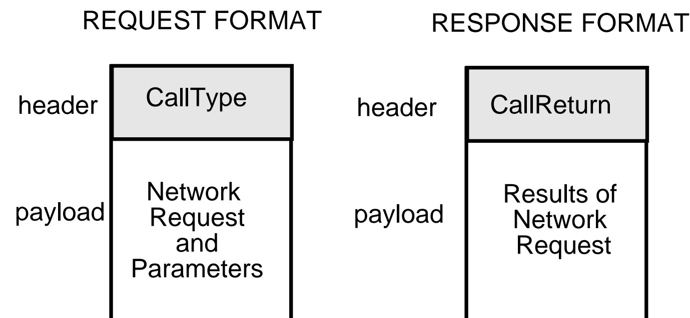

Figure 5.2: Request/Response Format

application, along with the parameters associated with the network request. The TCP Server on receiving a request identifies the network operation and performs the required processing. The response is then packaged, identifying the network operation and the results of the call. This request/response format is also used for internal control messages between the host and the TCP Server. These control messages are typically used for flow control, initiating eager processing at the TCP Server and for identifying RDMA data buffers. The request packet contains a `CallType` as its header. This identifies the type of call being sent from the host to the TCP Server.

```
typedef enum
{
   SOCKET_CALL,
   BIND_CALL,
   LISTEN_CALL,
   ACCEPT_CALL,
   ....
}CallType;
```

The request payload contains a structure associated with this `CallType`. This structure contains the data relevant to the call. For instance, `struct AcceptData` contains the parameters passed

to the `accept` call, `struct SendData` contains the parameters passed to the `send` call etc. The response packet contains a `CallReturn` as its header. It identifies the call (`CallType` field) and gives an indication of success or failure of the call.

```
typedef struct
{
   CallType call;
   int error; /* indicates if the call was a success or failure */
   int errorNo; /* errorNo at the TCP server node in case of error */
}CallReturn;
```

In the case of failures, the response packet does not contain a payload. Successful calls contain a payload which is a return structure containing the results of the call. For instance, `struct AcceptReturn` contains the results of the `accept` call, `struct RecvReturn` contains the results of the `recv` call etc.

It is to be noted that the socket descriptors in our architecture are valid in the context of the TCP Server node. Hence in the case of errors if the host application issues `perror` or uses `errorno` to query the status of the error, it will result in incorrect behavior since the host node is oblivious to the error that has taken place at the TCP Server node. To provide for correct behavior, the `errorno` that resulted in the TCP Server is passed back to the host and the library call sets the host node's `errorno` to the value on the TCP Server node.

The request/response format described above is used for the Send/Recv data transfer model. In the case of RDMA transfer [1], the basic structure required is the `RDMAInfo`. The destination memory address and memory handle must be exposed to the sender before it can initiate an RDMA transfer.

```
typedef struct
{
   char *pdata; /* pointer to remote buffer */
   VIP_MEM_HANDLE memoryHandle; /* memory handle of buffer */
}RDMAInfo, *pRDMAInfo;
```

[1] Giganet's cLAN does not provide RDMA Read, only RDMA Write is provided

5.4 TCP Server Implementation

Thread Model

The following is a discussion of the threads that constitute the TCP Server. As shown in Figure

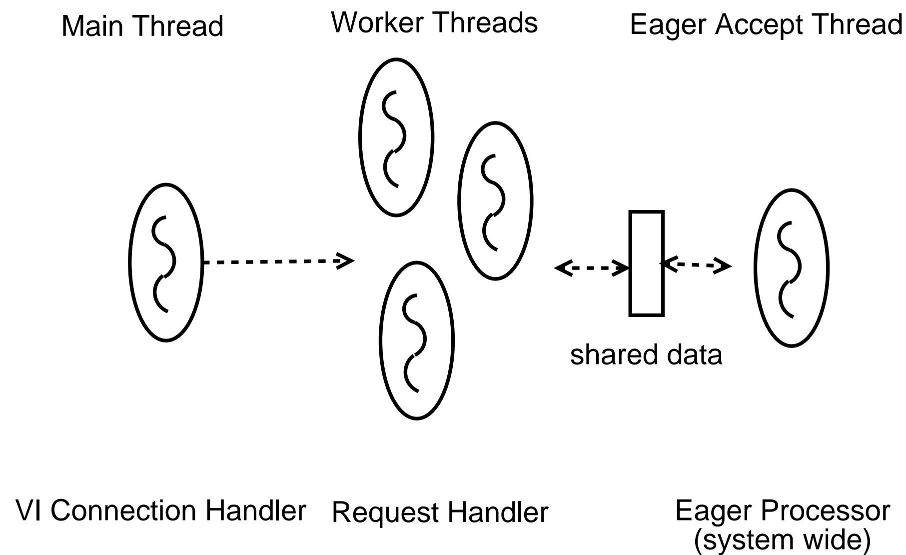

Figure 5.3: Threaded Model of the TCP Server

5.3, the *main* thread plays the role of the *VI Connection Handler*. It waits for VI Connection Requests. Once a VI Connection Request comes in, and the *Admission Controller* logic decides to accept the Connection, a new VI is created and the VI is handed over to a *worker* thread. The *worker* thread serves as the *Request Handler*. It processes all requests received on this VI and is responsible for responding back to the host. A system wide *eager accept* thread is responsible for *eager accept* processing and communicates to the worker threads using shared data structures. In this model, each *worker* thread individually calls `VipRecvWait` and waits for a request to arrive on the VI. An alternative model would be to associate all the VIs with a Completion Queue and have a single *Completion* thread wait for a request on any of the VIs using `VipCQWait`. On receiving a request the appropriate *worker* thread could be awakened. This alternative model was originally used but it introduced additional latency because the handling of requests was now dependent on the scheduling of the *Completion* thread. It was therefore replaced by the current design.

Data Structures used by the TCP Server

At the TCP Server, the main data structures used are the `ViDetail` which holds a mapping of a

given VI to the `ViConnThread` data structure. When the *main* thread decides to accept a given VI Connection request, it creates the above structures and passes a pointer to the `ViDetail` to a *worker* thread. Each *worker* thread works off a single `ViDetail` and is responsible for processing all requests that come on that VI connection. The `ViConnThread` contains all the requisite fields related to the VI, worker thread and VI connection. The key fields are shown below.

```
typedef struct
{
   VIP_VI_HANDLE viHandle; /* identifier of the VI */
   pViConnThread pviConn;
}ViDetail, *pViDetail;

typedef struct
{
   /* send descriptor used for regular calls */
   VIP_DESCRIPTOR *psendDescr;
   /* an array of receive descriptors are used to enable pipelining of
     asynchronous sends */
   VIP_DESCRIPTOR *precvDescr[RECV_DESCRS];
   VIP_MEM_HANDLE memoryHandle; /* registered memory handle */
   /* descriptor used for asynchronous send*/
   VIP_DESCRIPTOR *pRDMASendDescr;
   /* memory handle for asynchronous sends */
   VIP_MEM_HANDLE RDMADescrHandle;
   VIP_CONN_HANDLE connection; /* VI connection identifier */
   /* thread id of worker thread responsible for this VI */
   pthread_t thread;
   NicRDMASend *pRDMASend; /* structure used for asynchronous send */
   NicRDMARecv *pRDMARecv; /* structure used for eager receive */
   .....
```

```
}ViConnThread, *pViConnThread;
```

5.5 Optimizations

We first describe the Synchronous call processing mechanism, which is the base-line implementation and then discuss each of the optimizations implemented.

Synchronous call processing

In the case of processing synchronous calls, the library routine at the host follows the basic format outlined below.

```
SynchronousCall(....)
{
        SocketToViLookup();
        MarshallParams();
        PostRequest();
        WaitForResponse();
        Unmarshallparams();
        ReturnToApplication();
}
```

The `SocketToViLookup` routine, searches for a socket to VI mapping in the hash table, and if none is found, picks one from the free pool. If there is no free connected VI then a new one is created and a connection is established on it. If the selected VI was not found in the hash table, its entry is added. The `WaitForResponse` routine waits until the request sent by the host reaches the TCP Server, is processed by the TCP Server and the response is sent back and received by the host.

At the TCP Server side, a *worker* thread representing the *Request Handler* processes incoming requests as follows:

```
while( WaitForRequest() )
{
   UnmarshallParams();
   ProcessRequest();
```

```
    MarshallReturnParams();
    PostResponse();
}
```

5.5.1 Asynchronous Sends

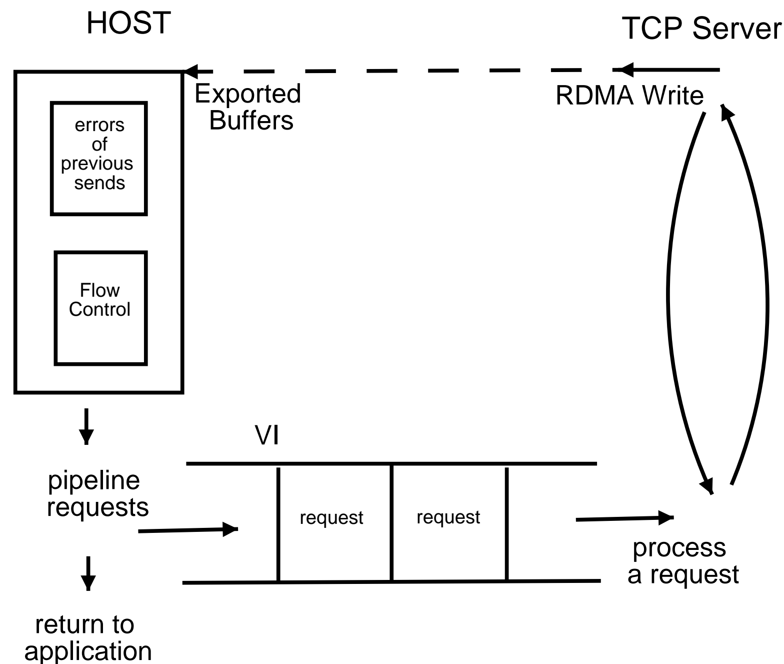

Figure 5.4: Asynchronous Send Call Processing

Figure 5.4 shows the sequence of steps that take place at the host and at the TCP Server to process an *asynchronous send* call. The host first checks for errors of previous sends, if any. It then ensures that there are sufficient receive descriptors available at the TCP Server. It then pipelines the request on the VI and immediately returns to the application. The TCP Server processes the incoming request and then does a RDMA Write of the results of the call on to the exported host buffers. It also updates the information required for flow control. Flow Control is a major component of asynchronous call processing. To enable *asynchronous send*, the flow control mechanism between the host and the TCP Server has to be set up initially. RECV_DESCRS represents the number of receive descriptors that the TCP Server has posted per VI. Leaving one receive descriptor free to process the synchronous calls, `maxPipeline` the maximum number of asynchronous requests that can be pipelined by the host

is RECV_DESCRS-1. To ensure that requests are not dropped, the host has to make sure that the number of pending requests does not increase `maxPipeline`. `SendCtl` is the basic structure which contains the fields that the TCP Server updates after it processes an *asynchronous send*. After updating its local copy of the `SendCtl`, the TCP Server performs a Remote Memory Write (RDMA Write), on to the remote copy at the host.

```
typedef struct
{
  int processedRecvs;
  int error[maxPipeline];
}SendCtl, *pSendCtl;
```

`SendCtl` contains `processedRecvs` which is incremented every time an asynchronous send is processed by the TCP Server. An array of size `maxPipeline` is used to report the results/errors of the previous asynchronous sends.

```
typedef struct
{
  /* RDMA write destination */
  pSendCtl pcontrol;
  /* memory handle of RDMA Write destination */
  VIP_MEM_HANDLE ctlHandle;
  int postedRecvs;
  int checkedRecvs;
  /* indicates is flow control setup has been completed */
  int fremoteSetup;
}HostRDMASend, *pHostRDMASend;
```

The `HostRDMASend` is maintained by the host. In addition to the `SendCtl`, it also contains `postedRecvs` a count of the number of *asynchronous sends* posted so far. The difference between `postedRecvs` and `processedRecvs` represents the number of receive descriptors at the TCP Server that the host has used up. The `postedRecvs` and `processedRecvs` are used by the host and TCP Server as an index into the `error` array (with modulo function for wrap

around). As shown in Figure 5.5, error[0] holds the results of request1, error[1] of request2

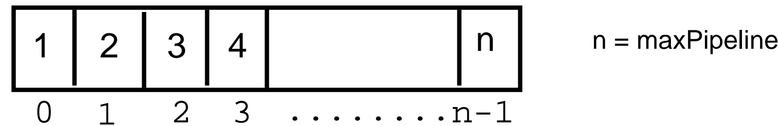

Figure 5.5: Location of Results of Requests

etc. The `checkedRecvs` is used by the host to keep track of the index up to which the results of previous sends have already been checked and errors if any reported back to the application. At the time of flow control setup, an `RDMAInfo` containing the address and memory handle of `pcontrol` - the RDMA Write destination, is sent by the host to the TCP Server. This is sent as a separate request with `CallType` SEND_FLOW_SETUP. Once this setup has been completed, when an *asynchronous send* is issued by the host, the following sequence of steps are involved:

```
AsynchronousSend(...)
{
   SocketToViLookup();
   if(!setup)
   {
      SendFlowCtlSetup();
   }
   ReportErrorsOfPreviousSends();
   /* busy wait until a receive descriptor is available
      at the TCP Server */
   while(postedRecvs - processedRecvs) >= maxPipeline);
   MarshallParams();
   PostRequest();
   postedRecvs++;
   ReturnToApplication();
}
```

For every *asynchronous send*, the host first checks if the previous sends have completed successfully. If there has been an error in any of them, it reports it back to the application.

Figure 5.6 gives an example of how the results of previous sends are verified. Consider

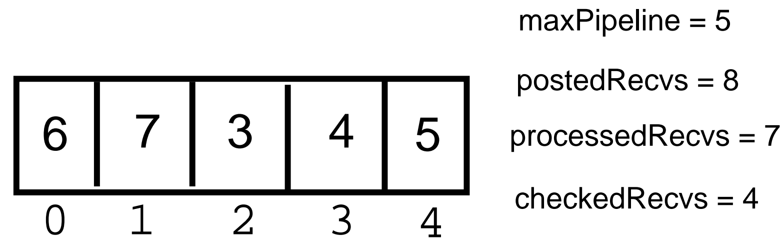

Figure 5.6: Example Flow Control Processing

the stage in processing as shown. The host has posted 8 sends so far, the TCP Server has processed 7 of them, whereas the host has verified the results for only 4 so far. At this stage if a *asynchronous send* call is made, `ReportErrorsOfPreviousSends` will have to look at the results of sends 5,6 and 7, starting from the earliest unreported error. It therefore looks at index 4, 0, and 1 of the `error` array and returns to the application any errors it may find. It also updates the `checkedRecvs` to indicate the number it has already checked. In case there are no errors, in the previous example `checkedRecvs` will be updated to 7. After `ReportErrorsOfPreviousSends` completes, the host can post a request only if the difference between `postedRecvs` and `processedRecvs` is less than `maxPipeline`. However if it is equal to `maxPipeline` then the host has to wait for the `processedRecvs` to be updated by the TCP Server. When the TCP Server indicates the availability of a free descriptor, the host posts the request. After the request has been posted, the library call returns to the application. The `send` call returns the number of bytes that have been successfully sent. In the case of *asynchronous send* the number of bytes that were successfully sent is the total number of bytes requested to be sent by the application. This is because, once the data reaches the TCP Server, the TCP Server takes care of transient errors and retransmissions and ensures that the data is eventually sent out, unless there is a serious error in which case it is reported back to the application. The following traces the processing steps involved at the TCP Server to handle an *asynchronous send*. This happens in the context of the *Request Handler*

```
ProcessRequest(..)
{
   ...
   case ASYNC_SEND:
        if( !setup)
        {
          ReturnError();
        }
        send(); /* actual send socket call */
        UpdateSrcSendCtlFields();
        RDMAWriteSendCtl();
        break;
   ...
}
```

5.5.2 Eager Receive

Eager receive occurs in the context of each *worker* thread that processes requests on a given VI. This processing starts only after the host has indicated to the TCP Server to start *eager receives* for the given socket. Each time the *worker* thread processes any request, before it loops back to wait for another request, it does the *eager receive* processing. The `NicRDMARecv` is maintained by the TCP Server, to facilitate *eager receive* processing.

```
typedef struct
{
  char *pdata;
  VIP_MEM_HANDLE dataHandle;
  /* position up to which data has been eagerly received */
  int writeOffset;
  /* position up to which data has been reported back to host */
  int readOffset;
  int socketFd;
```

```
  int flags;
  ...
}NicRDMARecv, *pNicRDMARecv;
```

pdata points to a memory region of size EAGER_RECV_SIZE which is typically around 32KB. This is the *eager receive* buffer, shown in Figure 5.7 and is allocated per VI. *Eager*

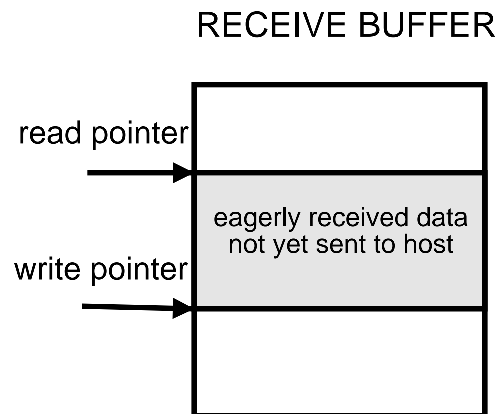

Figure 5.7: Eager Receive Buffer

receive proceeds depending on the rate at which the host consumes the data. EAGER_RECV_THRESH, typically (EAGER_RECV_SIZE/2) is the limiting threshold. If previous eager receives have already read EAGER_RECV_THRESH data and the host has still not issued a recv to consume that data, then *eager receive* is automatically stopped. It will resume once the host consumes some of this data. Data is eagerly received in chunks of EAGER_RECV_UNIT which is set around 2 KB and can then be increased or decreased depending on the number of bytes successfully read by the recv. The *worker* thread first issues a poll system call, with NO_WAIT option. This checks if the socket is readable, and if it is, then recv is issued and data read on to pdata at locations determined by the current positions of the readOffset and writeOffset. If there is an error then the *eager receive* is stopped and data that was previously read, up to that point, is first returned to the host and then the error reported. With *eager receive* every time the host issues a recv, the TCP Server first checks NicRDMARecv to determine if data has been eagerly received. If yes, then the send descriptor is setup as a gather list of buffers and the data sent to the host. The current implementation represents a pull-based approach. A push-based approach will carry the *eager receive* one step

further and the TCP Server will do a RDMA Write of eagerly received data back on to host buffers.

5.5.3 Eager Accept

Eager accept, unlike *eager receive*, is not per *worker* thread but is system-wide. The *worker* thread that receives the `listen` call, creates the *accept* thread. The *accept* thread eagerly posts `accept` socket call, adds the accepted connections to a mutex protected linked list, and posts a semaphore to indicate an available accepted connection. Each *worker* thread that receives an `accept` request from the host, waits on the semaphore, and then retrieves one of the eagerly accepted connections from the linked list and returns it to the host. When the *worker* thread that created the *accept* thread receives a `close` from the host, it issues a cleanup to stop the *eager accept* processing. The *accept* thread, similar to *eager receive* processing, stops the *eager accept* if the number of previously accepted connections exceeds EAGER_ACCEPT_SIZE which is a configurable parameter.

5.5.4 Avoiding Data Copies at the Host

In the Send/Receive model of VIA data transfer, the descriptor format allows the source and destination to be described as a scatter-gather list of buffers within each descriptor.

In `ns_send_registered` and `ns_send_async_registered` the application passes a pointer to a pre registered memory region along with the memory handle associated with that region. The library call, while formatting the send descriptor, formats it as a gather list with multiple data segments. The first data segment is used from the memory region associated with the `ViInfo`. This is used to transfer the relevant request headers (`CallType` and Data `struct`) associated with this call. The next segment points to the buffer passed in by the application and contains the actual data transferred in the `send` call. Therefore in the `MarshallParams` routine there is no need to copy the user data on to the memory regions associated with the `ViInfo`. Instead the descriptor is formatted differently avoiding a `memcpy`.

In the case of `ns_send_async_registered`, the call returns as soon as the library enqueues the request to be sent, therefore a job identifier NS_IO_RESULT is returned from the call. The application can later query the status of the send, by passing the job identifier to `ns_io_done` or

ns_io_wait. After the job identifier is marked as complete the application can then reuse the memory buffers associated with that send. However until that point, the application must make sure that it does not overwrite the data in the send buffers.

In the case of ns_recv_registered the receive descriptor uses a scatter list of buffers and is formatted with multiple data segments. The first segment points to the memory associated with the ViInfo. The relevant response headers (CallReturn and Return struct) associated with this call are received in this segment. The next segment points to the buffer that is passed in by the application. The actual data of the recv call is received directly into these buffers. The UnmarshallParams routine does not have to copy the received data on to the application buffers. Thus a memcpy is avoided on the receive path.

The flip side to this scheme is that the application is now responsible for pre-registering memory regions, and has to have its own buffer management scheme to ensure that the pre-registered memory is appropriately used in place of the regular send and recv buffers.

Chapter 6

Performance Evaluation

The goal of our performance evaluation was to study the behavior of Internet servers under high loads, using the TCP Server architecture. The contributions to server performance by each of the different TCP Server optimizations was evaluated. We studied the server performance using the HTTP/1.0 and HTTP/1.1 protocols.

6.1 Experimental Setup

6.1.1 Hardware Platform

The hardware environment that was used to evaluate the performance of the TCP Server architecture consisted of two 300 MHz Pentium II PCs, with 512 MB DRAM and 256 KB L2 cache. One PC served as the host node and the other served as the TCP Server node. The host and the TCP Server communicated over 32-bit 33 MHz Emulex cLAN interfaces and an 8-port Emulex [24] switch. The TCP Server node was installed with a 3Com 996B-T Gigabit Ethernet adapter which was used for client-server communication. For the client node, we selected a more powerful 550 MHz processor, so that it could generate sufficient requests to saturate the server. A 3Com 996B-T Gigabit Ethernet adapter was also installed on the client. The underlying operating system on all the machines was Linux-2.4.16.

6.1.2 Web Server

For our experiments, we built a multi-threaded Web server which served as the host application and ran on the host node. The Web server follows the HTTP protocol and services HTTP requests from clients. We used a custom-built Web server, instead of an existing Web server, to provide us with increased flexibility in manipulating application buffering. To fully realize the

potential of the TCP Server architecture, the Web server was also modified as required to use the additional APIs for asynchronous call processing and pre-registering data buffers.

6.1.3 Client Benchmarking Tool

Httperf [37] was used as the client benchmarking tool to generate the required workloads to study the performance of the Web server using traditional TCP/IP and using the TCP Server architecture. The key idea behind httperf is that an Internet based server could potentially service hundreds of millions of users. Simulating these client requests using a fixed and small number of clients is not sufficient. Httperf generates and sustains overload which in effect simulates an infinite user base.

6.2 Microbenchmarks

Since the SAN was the interconnect between the host and TCP Server, in this section we present microbenchmarks measured on the SAN.

6.2.1 SAN Performance Characteristics

Table 6.1 describes the performance characteristics of the VIA-based SAN that was used for our experimental evaluation. Latency denotes the time taken to transfer a 1 word packet between two nodes using VIA. PostSend denotes the average time taken to post a send request on VIA. The last row represents the cost of the `VipRegisterMem` operation used to register a memory region of size 4096 bytes using VIA.

One-way latency (1 word)	8.2 μs
Bandwidth (32 KB)	102 MB/s
PostSend (4 KB)	2.1 μs
RegisterMem (4 KB)	4.3 μs

Table 6.1: VIA Microbenchmarks

6.2.2 Cost of Send Call

Figure 6.1 shows the cost of the `send` socket call for different data sizes. It represents the latency perceived by the application from the time the call is posted to the time it returns. For

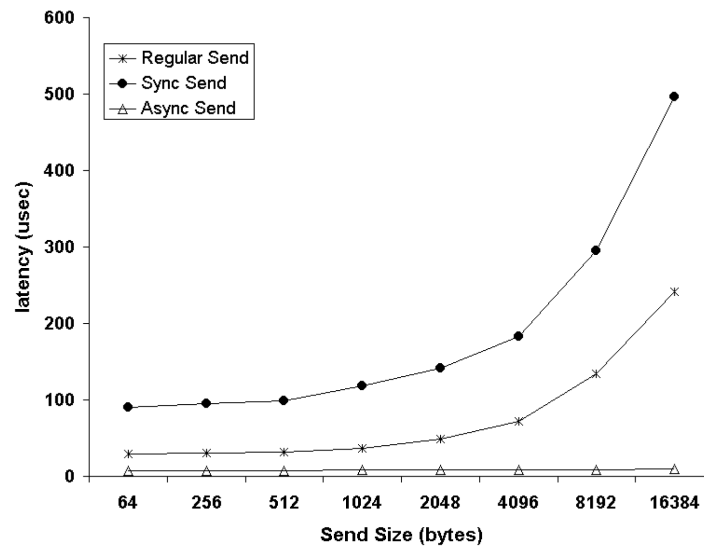

Figure 6.1: Cost of send

further clarity, Table 6.2 lists the actual values. The send socket call is studied in detail because

Data Size (Bytes)	Time Taken for send (μs)		
	Regular Send	Sync Send	Async Send
64	29	90	7.56
256	30	95	7.47
512	32	99	7.73
1024	36	118	8.13
2048	49	142	8.34
4096	72	183	8.62
8192	134	285	9.11
16384	242	496	9.29

Table 6.2: Cost of send

it plays an important role in server applications like Web servers. Regular Send represents the time taken by the traditional socket library routine. Sync Send represents the time taken by the synchronous implementation of send using the TCP Server architecture. Async Send represents the latency of the *asynchronous send*. The latency of Sync Send is inflated because the library call at the host blocks, waiting for the TCP Server to execute the traditional send and then return the results back to the host node. Thus it includes a round trip across the SAN in addition to the cost seen by Regular Send. Async Send eliminates the waiting time at the host by immediately returning to the application as soon as the send is scheduled over the VIA. The time taken for this is significantly lower than that of Regular Send and Sync Send. Async Send

allows the application to overlap the `send` with other useful operations.

To provide a better understanding of the sub components which contribute to the cost of Sync

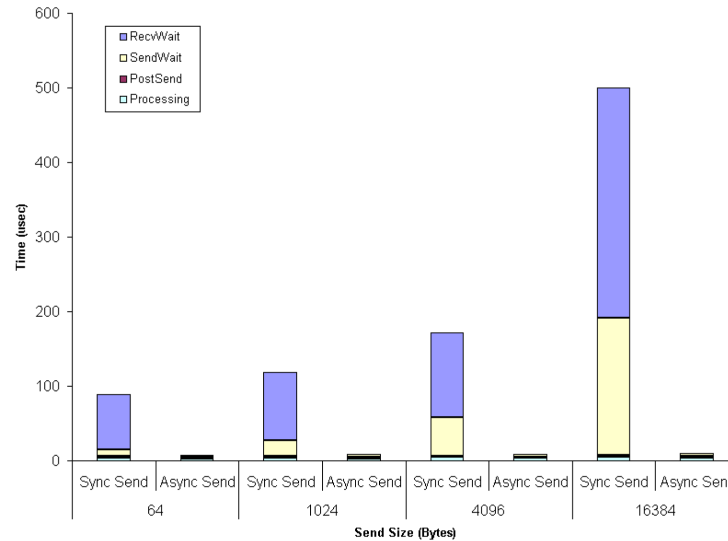

Figure 6.2: Breakdown of the Cost of `send`

Send and Async send, in Figure 6.2, we provide a breakdown of the cost. For further clarity, Table 6.3 lists the actual values. The breakdown shows four major components. Processing

Data Size (Bytes)	Send Type	Time Taken (μs)			
		Processing	PostSend	SendWait	RecvWait
64	Sync Send	4.13	2.04	8.15	74.76
	Async Send	2.8	2.06	2.97	0
1024	Sync Send	4.22	2.03	21.33	91.12
	Async Send	3.02	2.16	3.21	0
4096	Sync Send	4.4	1.97	51.9	112.83
	Async Send	3.24	2.27	3.39	0
16384	Sync Send	5.41	2.14	183.36	308.59
	Async Send	3.5	2.35	3.63	0

Table 6.3: Breakdown of the Cost of `send`

includes the time involved in pre-processing (including socket to VI lookup, marshalling parameters) and post-processing (including unmarshalling the results). PostSend is the time taken to queue the request to the send work queue of the VI and translates to a `VipPostSend` request. SendWait is the time taken to send the data to the network, i.e., it indicates the completion of the send request processing by VIA and translates to `VipSendWait`. RecvWait includes the time taken for the data to reach the TCP Server and for the TCP Server to process the request and

return the results of the call to the host node. The host node waits on a call to `VipRecvWait`. It is seen that in the case of Sync Send the cost of RecvWait is the most dominating factor which increases as the size of the data transferred increases. In the case of Async Send, since the call returns as soon as the request is submitted for processing, there is no cost associated with RecvWait. It is to be noted that the cost of SendWait in the case of Async Send is lower than that of Sync Send. In the case of Sync Send the SendWait immediately follows the PostSend and therefore waits for VI to complete the current send request. However, in the case of Async Send, the call returns after PostSend, and SendWait is done at the time of the next Async Send request. By this time the request is expected to have completed and hence SendWait takes less time.

The latency to transfer data across the SAN, is a major component of the cost of Sync Send.

Data Size (Bytes)	One-way latency (μs)	
	Wait	Poll
16	26.4	10.95
32	28.3	12.1
64	28.4	13
128	28.7	15.8
512	32.75	17.55
1024	39.45	22.6
4096	71.5	51.4
8192	110.75	90.3
16384	188.15	167.35

Table 6.4: VIA One-Way Latency

Table 6.4, lists the one-way latencies measured using `tvia` a VIA benchmarking tool provided by Emulex. The benchmarking tool includes the time taken for `VipPostSend`, `VipSendWait` and `VipRecvWait` in the calculation of one-way latency. Thus the time taken by the Sync Send components, excluding the Processing time, should be approximately equal to the sum of one-way latency from host to TCP Server, time for `send` processing at TCP Server, and one-way latency from TCP Server to host. From the above values, it is seen that this holds true. It is to be noted that the one-way latency from host to TCP Server and TCP Server to host will not be the same because it is dependent on the size of data transferred. This size is asymmetric; while the host may send 16KB of data to the TCP Server, the TCP Server on processing the call returns the results of the `send` which is around 64 Bytes. Of interest in Table 6.4, is the fact

that the one-way latency varies depending on whether the blocking or polling VI primitives are used. Using the polling primitives, will result in lower latency, but this will be at the cost of increased CPU usage and therefore we use the blocking primitives in our implementation.

6.2.3 Detailed Analysis

Based on the benchmark numbers presented in the previous section we present a detailed analysis of the `send` call. This analysis is presented for the data size of 64 Bytes. Figure 6.3 shows

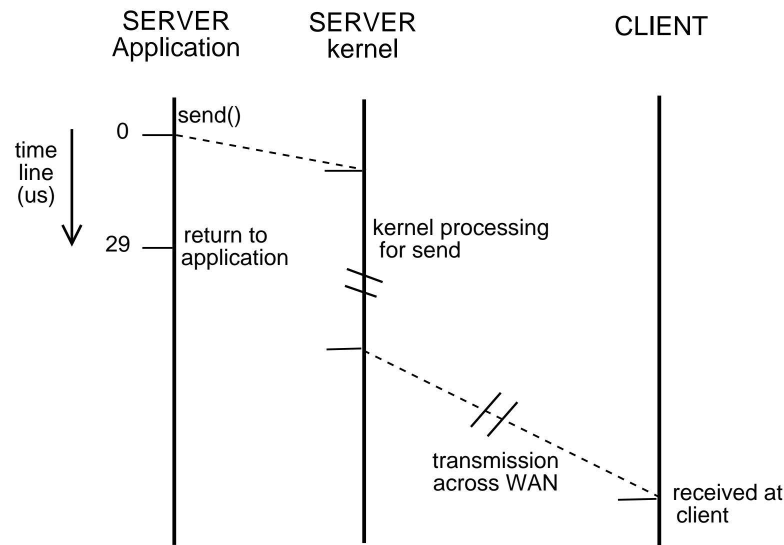

Figure 6.3: Flow of `send`

the flow of the traditional `send` call from the time it is issued by the Server application. The vertical lines represent the time line in μs. The time when the `send` is issued by the server application is considered as the start-time. The `send` reaches the Server kernel, and after the copy on to kernel buffers is complete, returns to the application in 29 μs. The kernel then completes its processing for the `send` and transmits the data on to the network. The data is then received by the client. Figure 6.4 shows the flow of the Synchronous `send` in the case of the TCP Server architecture. The cumulative time taken is presented at every stage of the processing. The host application issues the `tcps_send`. The figure shows the time taken for the processing at the host to complete and the request to reach the TCP Server. The TCP Server then issues the traditional `send` call and then returns the results of the call back to the host. The call then returns to the host application after 89.6 μs. This latency is higher than that of the traditional

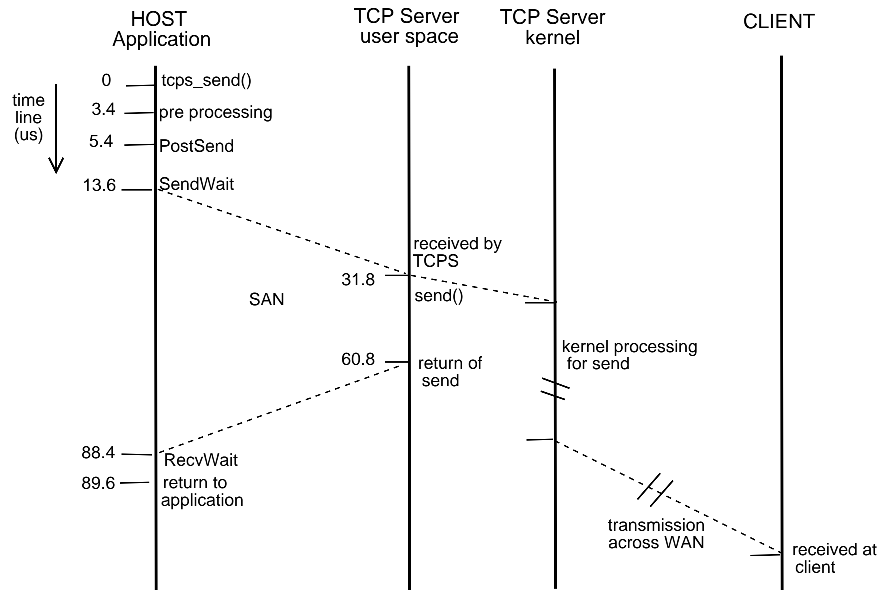

Figure 6.4: Flow of Synchronous send in the TCP Server Architecture

send. However, during the time between SendWait and RecvWait the application thread is only blocked. Therefore in terms of CPU overhead, the Synchronous send only takes about 15 μs compared to the 29 μs taken by the traditional send. Also, tcps_send could in fact return after the SendWait, i.e., once the request has been transmitted on to the SAN. In Figure 6.5 the time

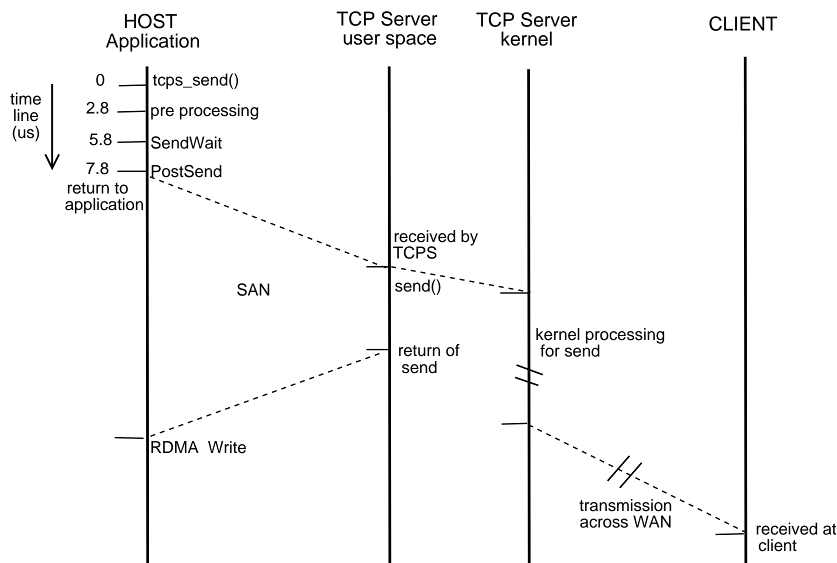

Figure 6.5: Flow of Asynchronous send in the TCP Server Architecture

taken for the *asynchronous send* is shown. The call returns to the application after the PostSend. SendWait is done at the time of the next send and therefore occurs before the PostSend.

The call returns to the application sooner than in the traditional send. The time taken by the kernel to process the call and the transmission across the WAN to the client, remain the same in both the traditional and the TCP Server architecture and are not presented.

6.3 TTCP Benchmark Results

Before we proceed to the performance results using our Web server, in this section we present performance results using the TTCP benchmark [50]. In the TTCP setup, there is a transmitter and a receiver. The transmitter requests a TCP connection from the receiver by issuing a connect request. Once connection is established, the transmitter transmits data over the TCP connection. The receiver issues an accept to receive and accept the incoming client connection. In the case of typical servers, the flow of data from the server to the clients is much more than the data received by the server. Hence, in the TCP Server architecture, the host serves as the TTCP transmitter. The TTCP receiver is an external client. Table 6.5 explains the legends used in the graphs in this and the following sections. For the sake of consistency, we use the

Legend (in graph)	Explanation
Regular	Single Node, using traditional BSD sockets
Sync	TCP Server, using synchronous calls
AsyncSend	TCP Server, using *asynchronous send*
ERecv	TCP Server, using *eager receive*
EAccept	TCP Server, using *eager accept*
Setup	TCP Server, using *setup with accept*

Table 6.5: Legends Used in the Graphs

legends to represent the corresponding optimization in the text. For example, AsyncSend refers to the *asynchronous send* optimization. A combination as in AsyncSend + EAccept refers to a corresponding combination of the optimizations *asynchronous send* and *eager accept*. All the above variants of the TCP Server use pre-registered memory buffers for communication. Figure 6.6 shows the throughput measured for different data transfer buffer sizes, as measured by the transmitter, the host node. Figure 6.7 shows the throughput as measured by the receiver. The bars represent the throughput as measured by the real time taken (indicated on Y1). The curves represent the ratio of throughput to CPU utilization (indicated on Y2). This ratio is referred to as the performance efficiency (PE) index [33]. A high PE index indicates high

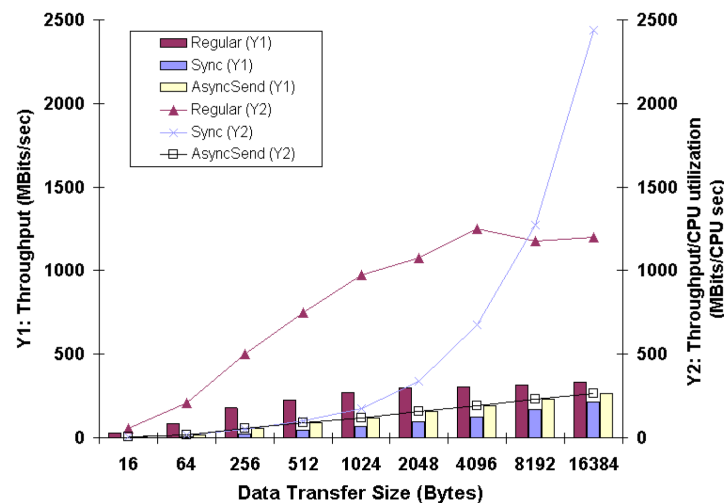

Figure 6.6: Throughput Measured at the Host Node - the TTCP Transmitter

throughput with low CPU utilization and suggests a favorable overall system performance. Regular shows an increased throughput measured using the real time compared to Sync and AsyncSend. The reason for this is that the TTCP benchmark involves transmission and reception of data over a single TCP connection; in the TCP Server architecture, every call includes an additional latency due to the transfer across the SAN from the host to the TCP Server. This does not happen in the case of real applications as discussed in Section 6.4, where the additional latency due to the TCP Server architecture is overcome by the parallelism of the server application and by multiple TCP connections. The throughput of AsyncSend is higher than that of Sync, since it allows the transmitter to pipeline more requests. In Figure 6.6, analyzing the ratio of throughput to CPU utilization for Sync and AsyncSend, we observe the following - Sync has a high CPU utilization for small message sizes, due to the higher processor utilization for small messages by clan VI [25]. For larger message sizes, the CPU utilization dramatically decreases, resulting in a higher PE index. AsyncSend however has a low PE index for all message sizes with this benchmark. AsyncSend reduces the latency of the `send` call providing a way for the application to perform application processing, and overlap network processing with application processing. In the case of the TTCP benchmark, the transmitter continuously transmits data and does no other application processing. Hence, when a `send` call returns faster, the transmitter issues another `send`. The host pipelines as many calls as possible to the TCP

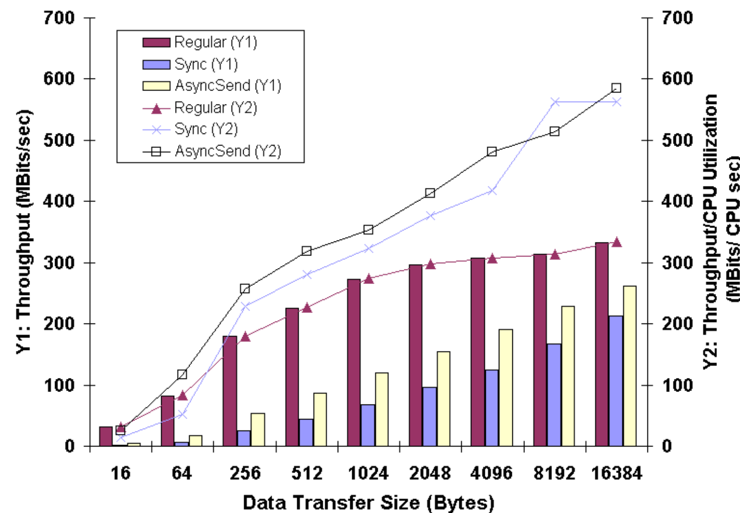

Figure 6.7: Throughput Measured at the Client Node - the TTCP Receiver

Server, but after a limit, it has to wait until the TCP Server can accept more requests. It does a "busy wait" checking the contents of the flow control buffers. Therefore, the CPU usage, has a high user time component which explains the low PE index for AsyncSend in the TTCP benchmark. Instead of doing a "busy wait", an alternative design would be to block the application thread. A separate flow control thread that receives flow control information from the TCP Server can then wake up the appropriate thread. This design may prove beneficial in the case of a single connection, but in the case of multiple connections, the latency involved in waiting for the flow control thread to be scheduled will outweigh the benefits and therefore this design was not chosen for implementation. In the case of real applications with multiple connections, our current implementation of AsyncSend performs well, as shown in Section 6.4. The PE index at the TTCP Receiver (Figure 6.7), is highest for AsyncSend. Both Sync and AsyncSend have a higher PE index compared to that of Regular.

6.4 Web Server Performance

In this section we present the performance of our Web server using different kinds of workloads. The TCP Server architecture is compared against a Web server running on a single node using the traditional BSD sockets. The aim of this evaluation is to quantify the performance gains

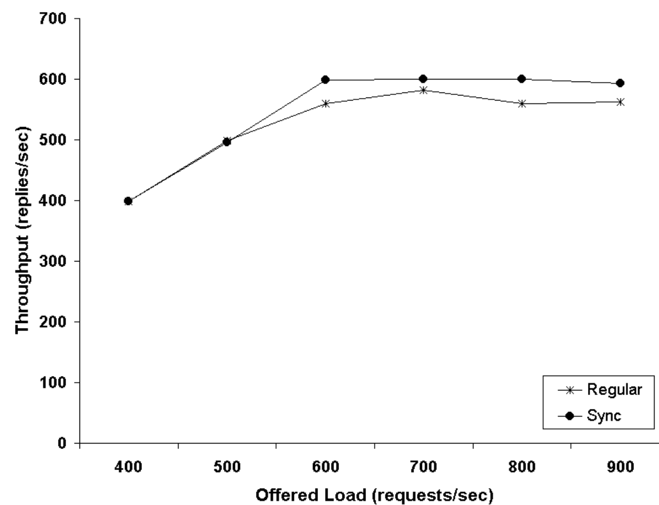

Figure 6.8: Web Server Throughput on Fast Ethernet

that can be achieved as a result of offloading the TCP/IP processing.

6.4.1 Initial Experiments with Fast Ethernet

The first set of experiments were run with the client connecting to the server over a Fast Ethernet (100 Mbps) network. A synthetic workload with repeated requests for a 16KB file was used. These experiments, in fact did not provide the anticipated results.

In Figure 6.8, it is seen that the performance of both Regular and Sync are the same and both saturate around the same offered load. The TCP Server apparently provided no additional gain over that of a stand alone single node server. Observing the throughput, it was seen that the throughput was around 78 Mbps, which was close to the network saturation point for Fast Ethernet. It was then evident that the network was the bottleneck preventing additional requests from being satisfied. To make sure that the network did not prove to be the bottleneck, further experiments were conducted after installing 3Com 996-BT gigabit Ethernet adapters on the client and server nodes and using a dedicated cable to connect the client and the server nodes. In the TCP Server setup the 3Com 996-BT gigabit ethernet adapter was installed on the TCP Server node, since the TCP Server node is the communication end point for the external world.

6.4.2 HTTP/1.0 Performance

Static Workloads:

The workload used in this case was a synthetic workload which consisted of requests for static files, 16KB in size. The working set was chosen to be larger than the L2 hardware cache, so that the returned file would not be cached. With this workload we were able to achieve performance gains of up to 17% using the TCP Server architecture.

Figure 6.9 presents the throughput of the server as a function of the offered load in re-

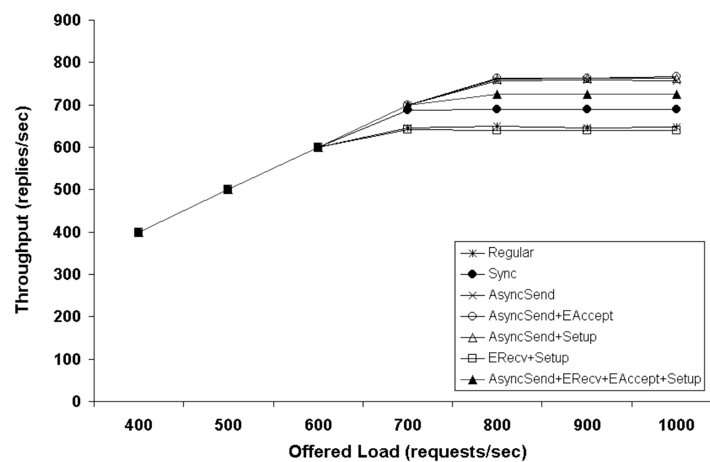

Figure 6.9: Web Server Throughput for HTTP/1.0 Static Loads

quests/second. At lower loads (less than 600 reqs/sec), all systems are able to satisfy the offered load. At high request rates, we see a difference in performance when Regular saturates at around 650 reqs/sec. An improvement of 7% in performance was obtained by Sync compared to Regular. This is the gain due to offloading of the TCP/IP processing. AsyncSend obtained a 17% increase in throughput compared to Regular. The additional improvement achieved by AsyncSend over Sync is due to the fact that the latency of individual `send` calls is reduced. This allows a better pipelining of network sends and helps the application overlap the latency of offloading the send primitive over the SAN with useful processing at the host.

AsyncSend + EAccept exhibits the same behavior as that of AsyncSend. The addition of EAccept was not able to provide additional gains over that of Async Send. This is because EAccept

helps improve the connection time. However it is not the connection time but the actual request processing time that dominates the network processing. AsyncSend + Setup removes one round trip across the VIA channels from the critical path of the request processing, but does not improve the `send` performance over that of AsyncSend.

ERecv alone does not help for this workload since it is the first `recv` on the socket that triggers ERecv processing and in HTTP/1.0 there is only one request per connection. Since ERecv + Setup sets up ERecv processing at the time of `accept`, it ensures that ERecv is triggered even before the first `recv`. It does not perform well because after processing any request from the host, the TCP Server executes the `poll` system call to verify if there is data on the socket to be received. This adds processing overhead to the network processing on the TCP Server node, and hence the performance is almost similar to that of Regular. Finally, contrary to our expectations, putting together all the optimizations AsyncSend+ERecv+EAccept+Setup does not provide the maximum gain. This is because of ERecv and the `poll` system call overhead that accompanies it.

Figure 6.10 provides better insight into the performance of the different systems by presenting the CPU utilization for the host and the TCP Server nodes. For each variant of the TCP Server architecture, the CPU utilization on the host node is represented by (Host) and the CPU utilization on the TCP Server node is represented by (TCPS). It is interesting to note that, even

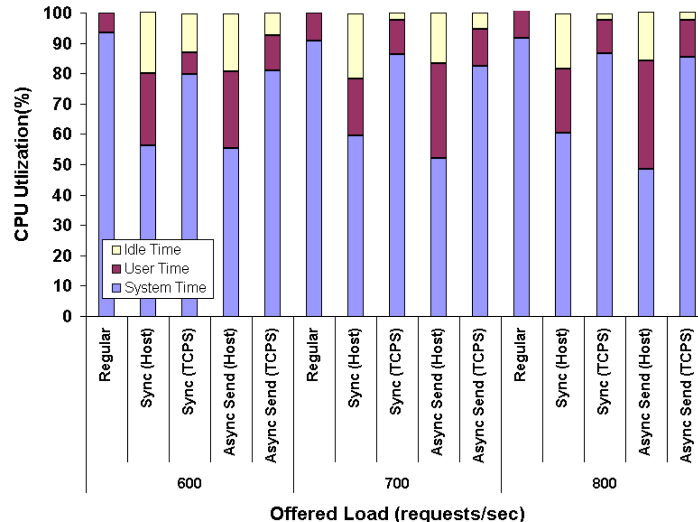

Figure 6.10: CPU Utilization for HTTP/1.0 Static Loads

before saturation point (load of 600 reqs/sec), with Regular, the CPU does not have any idle time. In the case of Sync and AsyncSend, the host has idle time available since the network processing has been offloaded to the TCP Server node. At higher loads (800 reqs/sec), the CPU usage on the TCP Server node for Sync and Async is similar to that of Regular with the network processing at the TCP Server node, finally saturating it. Analyzing the CPU utilization at the host nodes, it is seen that even after the network processing has been offloaded to the TCP Server node, the system time usage is substantial. This is due to the high system overheads associated with the Emulex VIA implementation that was used. In addition, the host also incurs processing overheads due to the file system processing which still occurs at the host node. Another observation is that the user time spent by AsyncSend host is higher than that of Sync host. Since AsyncSend processes more requests than Sync, it pumps more requests to the TCP Server node, than the TCP Server can process. There is a limit on the number of requests that can be asynchronously queued on a given VI. Once this limit is exceeded the host does a "busy wait", checking the contents of the flow control memory buffers. The TCP Server does an RDMA Write to update these buffers whenever it processes a request.

Further gains with this setup are not possible because the TCP Server node is excessively loaded. In fact, this explains why some of our combined optimizations, e.g., AsyncSend + EAccept, do not improve throughput beyond that of AsyncSend. These optimizations are intended to improve the performance of the host application at the cost of more processing at the TCP Server node. It is to be noted that speeding up the host does not really help overall performance because, at some point, the performance becomes limited by the TCP Server node. This problem can be alleviated in three different ways: by moving some of the load back to the application node (either statically or dynamically), by using a faster TCP Server, or by using multiple TCP Servers per application node. These approaches have to be further explored.

Figure 6.11 shows an experiment, wherein the *same* 16KB size file was repeatedly requested. This workload shows no additional gains using the TCP Server architecture's Sync variant. While the number of requests satisfied by Sync is similar to that shown in Figure 6.9, the number of requests satisfied by Regular has increased and is now similar to that of Sync. Since the same file is repeatedly requested, the file is available in the hardware L2 cache for Regular and is returned with no additional expense. This helps increase the number of requests that Regular

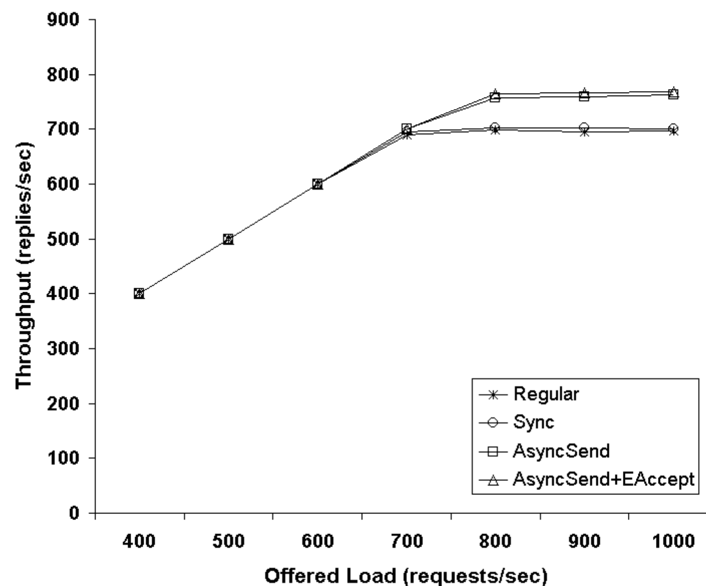

Figure 6.11: Web Server Throughput with Repeated Requests to the Same File

is able to satisfy. In the TCP Server architecture, the L2 cache effect is on the host and not on the TCP Server node. Requests received by the TCP Server, continue to be propagated across the SAN up to the host node, and therefore there is no increase in performance.

Combined Workloads:

To study a scenario which required application processing, in addition to network processing we used a combined workload. The workload contained requests for static and dynamic content (at a ratio of 20% dynamic content to 80% static content). The dynamic content behavior is that suggested by the WebStone 2.0.1 [52] benchmark, i.e., the data returned is computed by making a call to a random number function for each byte returned. No threads or processes are created on the critical path of the dynamic content requests. The same performance trade-offs for this workload was studied, using this more realistic combination of static and dynamic content requests. With this workload we were able to achieve performance gains of up to 18% using the TCP Server architecture.

Figure 6.12 presents the server throughput as a function of the offered load. The general performance trend is similar to that seen in Figure 6.9. Sync and AsyncSend saturate at higher loads compared to Regular. The gain in throughput for Sync compared to Regular is about 11%, which is higher than the corresponding gain with the static workload. The gain provided

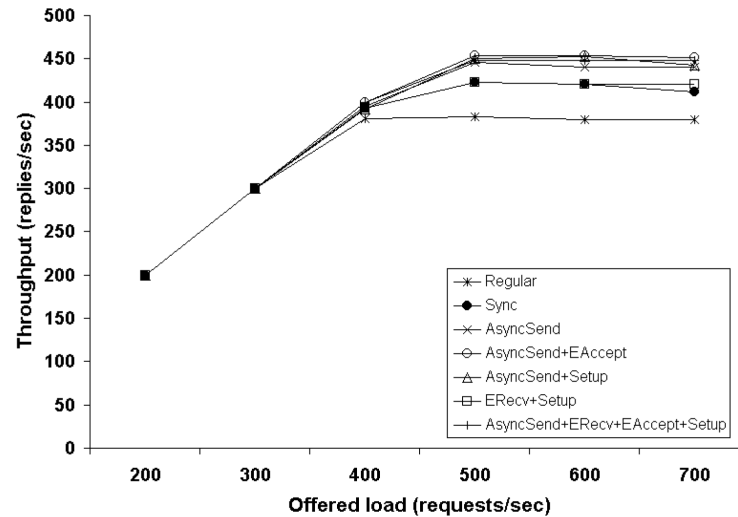

Figure 6.12: Web Server Throughput for HTTP/1.0 Combined Loads

by AsyncSend is about 18% and is only marginally higher than the gain found with the static workload. AsyncSend + Setup and AsyncSend + EAccept are similar to AsyncSend, as seen in the case of the static workload.

This is clear from Figure 6.13 which shows that at an offered load of about 500 reqs/sec, the host CPU is saturated both for Sync and AsyncSend. In Regular there is an increase in the user-time component of the CPU usage, since the application processing time now competes with the network processing time. The TCP Server node for Sync and Async has idle time available. Since the host node is saturated, this does not translate to any increased gain. In Figure 6.12, the performance of ERecv + Setup is similar to that of Sync, unlike the case of static loads. Though there is additional overhead in processing the `poll` system calls, the TCP Server node has sufficient idle time to absorb it. The combination of all optimizations Async-Send+ERecv+EAccept+Setup performs similar to that of Async Send. As mentioned earlier, the addition of ERecv does not decrease performance. These results show that, in this setup, greater gains are not possible because the host node is excessively loaded for this architecture/workload combination. We can alleviate this problem in ways similar to those discussed under static workloads. The focus, in this case, is on reducing the load on the host node.

Overall, we can infer from the results for the two workloads that balancing the load between the host and the TCP Server depends heavily on the particular characteristics of the workload.

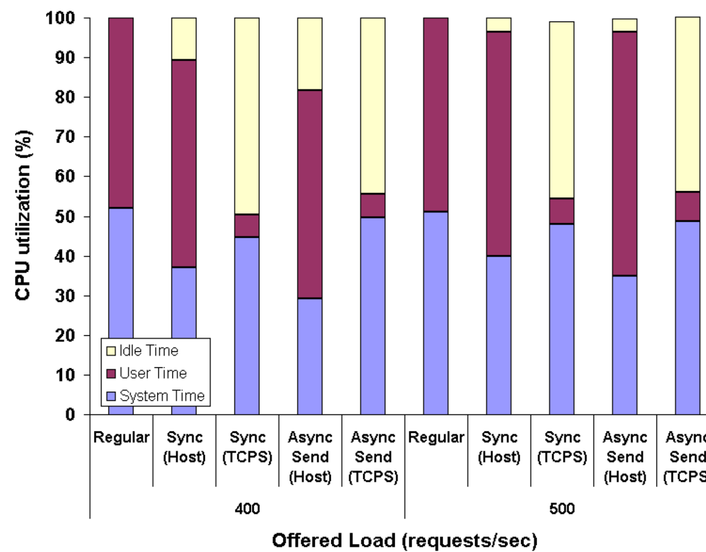

Figure 6.13: CPU Utilization for HTTP/1.0 Combined Loads

When these characteristics are fairly static and known in advance, a balanced system can be implemented. When either of these conditions does not hold, a dynamic scheme for balancing the load between host and TCP Server nodes is required for ideal performance.

Larger Transfer Size, Static Workloads:

In the previous sections, for the given HTTP/1.0 workloads, the Web server application used a buffer size of 4KB for the `send` calls. This was the send data transfer size. This meant that four transfers of 4KB each, were needed for a 16 KB file. In this section, the buffer transfer size was increased from 4KB to 16KB. Synthetic traces consisting of a static workload of 16KB files was used. With this workload we were able to achieve performance gains of up to 30% using the TCP Server architecture.

Figure 6.14 shows the throughput of the server as a function of the offered load. The behavior of Regular was almost similar to that seen in Figure 6.9. However all the implementations of the TCP Server architecture were able to sustain much higher loads. In the traditional TCP/IP processing, performing four `sends` of 4KB size instead of a single 16KB `send` did not make much of a difference to the processing. In the case of the TCP Server architecture, this resulted in a difference of doing four sends across VI (though of smaller transfer size) instead of a single transfer. A single transfer resulted in significantly lower VI resource usage and this translated into an increased performance gain.

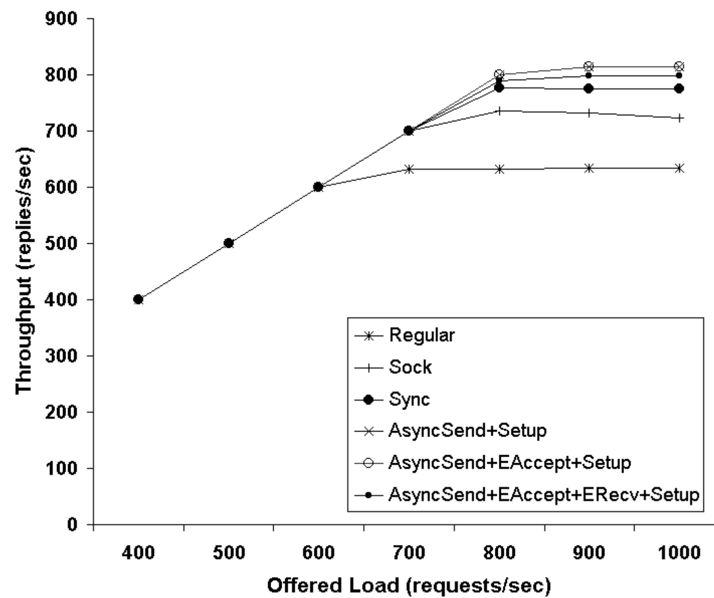

Figure 6.14: Web Server Throughput for HTTP/1.0 Static Loads, with 16K Transfer Size

In Sync a performance gain of 22% over that of Regular was observed. Since there was only one transfer per send, the cost of an additional Round Trip Time required by AsyncSend was not amortized, as it was in the case of 4KB transfer size. Therefore AsyncSend + Setup has been used in place of AsyncSend. AsyncSend + Setup achieved a performance gain of up to 30% over that of Regular. The other variants show the same trend displayed in Figure 6.9 with correspondingly higher gains. An additional variant showed is that of Sock. This represents the standard socket call primitives provided by the TCP Server architecture, i.e., the calls that do not require the application to use pre-registered data buffers. For a given `send` there is an additional copy from the application data buffers on to VI registered memory buffers used for host to TCP Server communication. The performance of Sock lies between that of Regular and Sync. It achieved a gain of 15% due to offloading in spite of the additional `memcpy` involved.

Figure 6.15 shows the CPU utilization for this set of experiments. In Figure 6.10 we observed that AsyncSend (Host) had a higher user time component compared to Sync (Host). In the current case however that is not so. This is because there is only one transfer from the host to the TCP Server node and therefore the host does not queue more requests than can be handled by the TCP Server. There is no “busy wait” on the part of the host waiting for a free receive descriptor available at the TCP Server. The descriptor flow control, which played a significant

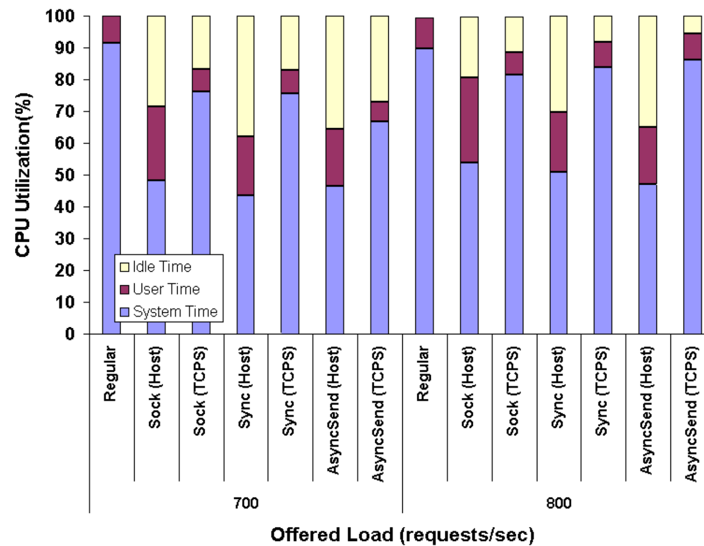

Figure 6.15: CPU Utilization for HTTP/1.0 Static Loads, with 16K Transfer Size

role in the previous case, is avoided here. The CPU usage for Sock (Host) shows a higher user time component compared to Sync (Host) and AsyncSend (Host) because of the additional time taken for `memcpy`.

6.4.3 HTTP/1.1 Performance

HTTP/1.1 includes features to alleviate some of the TCP/IP processing overheads. The use of persistent connections enables reuse of a TCP connection for multiple requests and amortizes the cost of connection setup and teardown over several requests. HTTP/1.1 also allows for pipelining of requests on a connection. The workload used for this study is the same as that used for HTTP/1.0. Synthetic traces with an application buffer size of 16KB was used. Multiple requests (six) were sent over each socket connection, in bursts of three. This meant that requests were pipelined three at a time and the host application could process up to three requests as a result of a single `recv` call.

Figure 6.16 shows the Web server throughput in this case. The performance gain achieved by Sync is about 17%, and by AsyncSend is 27%, over that of Regular. These performance gains show that the TCP Server architecture is able to provide substantial gains over that of a traditional networking system, even while using HTTP/1.1 features aimed at reducing networking overheads for application servers.

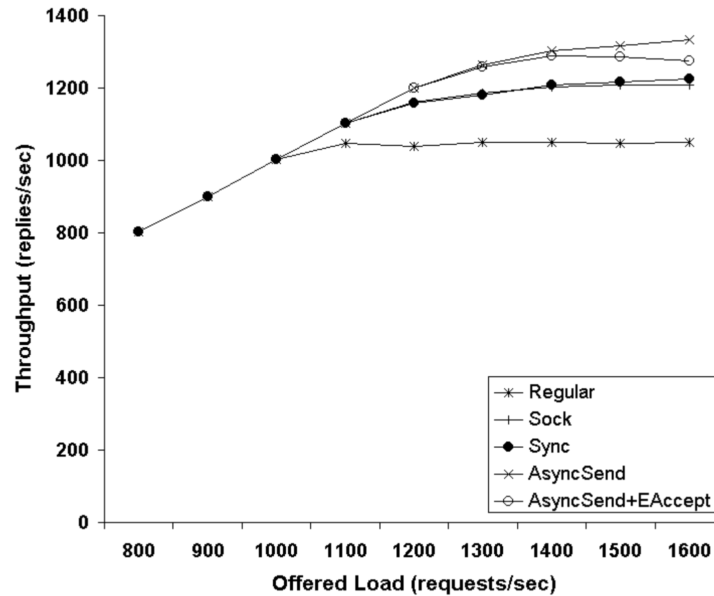

Figure 6.16: Web Server Throughput for HTTP/1.1 Loads, with 16K Transfer Size

Since multiple requests were sent on a single socket, the lifetime of a socket was longer compared to HTTP/1.0. Associating a socket to a VI at the time of `accept` (done by Setup) reduced concurrency and caused a significant drop in performance. Variations without using Setup have therefore been used. The performance gap between Sync and Sock was reduced compared to HTTP/1.0. Looking at their CPU utilization in Figure 6.17, it is seen that Sock has a significantly higher user time component compared to that of Sync. However, since requests are pipelined, the host is able to continue processing additional requests without affecting the network performance even in the case of Sock. The figure also shows that the TCP Server node is completely saturated in all cases like Regular. At the host, the user time component of AsyncSend is higher than that of Sync as seen in Figure 6.10. Several requests are sent across the same VI and the host ends up queuing more than the TCP Server can process, resulting in a "busy wait".

6.4.4 Performance with Real Traces

All the experiments so far were performed using a synthetic trace. To observe the behavior of the system, using real traces, we studied the performance of the Web server, using Forth. Forth is a real trace taken from the FORTH Institute in Greece. Table 6.6 describes the main

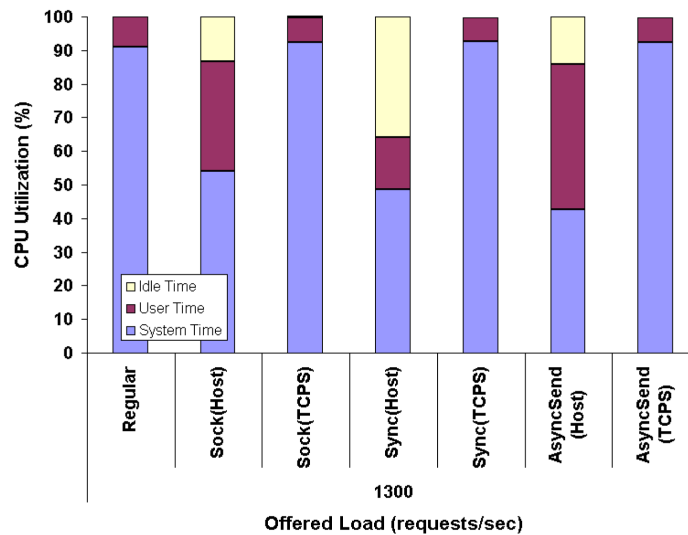

Figure 6.17: CPU Utilization for HTTP/1.1 Loads, with 16K Transfer Size

Logs	# files	Avg file size	Avg req size
Forth	11931	19.3 KB	8.8 KB
Synthetic	128	16.0 KB	16.0 KB

Table 6.6: Main Characteristics of WWW Server Traces

characteristics of Forth compared to Synthetic used in the earlier experiments.

Figure 6.18 shows the performance of the Web server as a function of offered load, using Forth. The trace involved HTTP/1.0 static requests and the Web server used a 16KB send buffer size. The trend is comparable to that observed for HTTP/1.0 workloads using Synthetic. Sync achieved a performance gain of 19% and AsyncSend a performance gain of 26%, compared to Regular.

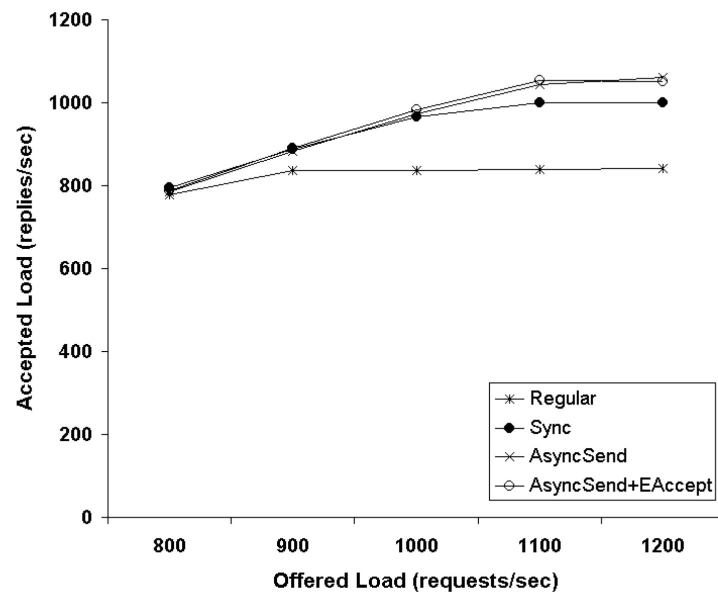

Figure 6.18: Web Server Throughput with a Real Trace (Forth)

Chapter 7

Conclusions and Future Directions

7.1 Conclusions

In this thesis, we have described the TCP Server architecture, a system architecture based on offloading network processing to dedicated TCP Servers. We have implemented and evaluated the TCP Server architecture for clusters, by separating the network functionality from the host node(s) (which run the host application) on to the TCP Server node(s) (dedicated to network processing). The architecture is built around a memory-mapped communication interconnect. Using our evaluations, we have quantified the impact of TCP/IP offloading on the performance of network servers.

Based on our experience and results, we draw several conclusions:

- Offloading TCP/IP processing is beneficial to overall system performance when the server is overloaded. Performance gains of up to 30% were achieved, due to offloading, in the scenarios we studied.
- TCP Servers demand substantial computing resources for complete offloading.
- The type of workload plays a significant role in the efficiency of TCP Servers. We observed that, depending on the application workload, either the host node, or the TCP Server node can become the bottleneck. Hence, a scheme to balance the load between the host and the TCP Server would be beneficial for server performance.

7.2 Future Work

Future directions for this work would be to:

- Explore the other design models presented for the TCP Server in Chapter 4.

- Optimize the TCP/IP processing on the TCP Server node: Since the resources of the TCP Server node are exclusively used for network processing, an optimized TCP/IP implementation can be used to further improve performance.

- Study the effect of heterogeneous configurations, by varying the processing power on the TCP Server nodes compared to that of the host nodes.

- Study the performance gains using *eager receive* and *eager accept* on receive-oriented network applications: The current TCP Server implementation was evaluated using a web server application. Optimizations such as *eager receive* and *eager accept* did not prove beneficial for a web server application. However, they may contribute to performance gains using certain receive-oriented network applications, e.g., a submission site. The benefits of using these optimizations to the TCP Server architecture for such applications can be further explored.

- Compare the TCP Server architecture against a cluster-based web server: In this thesis, the performance of the TCP Server architecture was compared against a web server running on a single node in the cluster. It would be interesting to compare the TCP Server architecture against a cluster-based web server running on both the nodes. This study will also depend on the mechanism for distributing requests between the two nodes.

- Compare the TCP Server architecture against hardware offloading solutions provided by intelligent Network Interface Cards.

- Explore the use of the TCP Server architecture to counter Denial Of Service attacks: Additional checks could be introduced at the TCP Server to help counter certain kinds of Denial Of Service attacks. Since the host application can communicate easily with the TCP Server, application specific metrics can be exported to the TCP Server to filter out unwanted connection requests. In this case the result of an `accept` at the TCP Server can be selectively propagated back to the host. Connections can be terminated at the TCP Server node without interrupting or affecting the host node.

References

[1] Acharya, A., Uysal, M., and Saltz, J. H. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the 8th ASPLOS* (1998), pp. 81–91.

[2] Adaptec ASA-7211 and ANA-7711. http://www.adaptec.com.

[3] Alacritech Storage and Network Acceleration. http://www.alacritech.com.

[4] Anderson, D. C., Chase, J. S., Gadde, S., Gallatin, A. J., Yocum, K. G., and Feeley, M. J. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the 1998 USENIX Technical Conference* (June 1998), pp. 143–154.

[5] Apache HTTP Server. http://httpd.apache.org.

[6] Aron, M., and Druschel, P. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems 18*, 3 (2000), 197–228.

[7] Banga, G., Druschel, P., and Mogul, J. C. Resource Containers: A New Facility for Resource Management in Server Systems. In *Operating Systems Design and Implementation* (1999), pp. 45–58.

[8] Basu, A., Buch, V., Vogels, W., and von Eicken, T. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995).

[9] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W.-K. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro 15*, 1 (Feb. 1995), 29–36.

[10] Brown, A., Oppenheimer, D., Keeton, K., Thomas, R., Kubiatowicz, J., and Patterson, D. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)* (March 1999).

[11] Buonadonna, P., and Culler, D. Queue pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th Annual Symposium on Computer Architecture* (May 2002).

[12] Carrera, E. V., and Bianchini, R. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practise of Parallel programming* (June 2001).

[13] Carrera, E. V., Rangarajan, M., Bianchini, R., and Iftode, L. Impact of Next-Generation I/O Architectures on the Design and Performance of Network Servers. In *Proc. of the Workshop on Novel Uses of System Area Networks, SAN-1* (February 2002).

[14] CARRERA, E. V., RAO, S., IFTODE, L., AND BIANCHINI, R. User-Level Communication in Cluster-Based Servers. In *Proceedings of the 8th IEEE International Symposium on High Performance Computer Architecture* (February 2001).

[15] CEZARY DUBNICKI, ANGELOS BILAS, K. L., AND PHILBIN, J. F. Design and Implementation of Virtual memory-mapped Communication on Myrinet. In *Proceedings of the 11th International Parallel Processing Symposium* (Apr. 1997).

[16] CHASE, J. S., GALLATIN, A. J., AND YOCUM, K. G. End-System Optimizations for High-Speed TCP. *IEEE Communications, special issue on TCP Performance in Future Networking Environments 39*, 4 (April 2001).

[17] Cisco systems inc. localdirector. http://www.cisco.com.

[18] COMPAQ CORPORATION, INTEL CORPORATION AND MICROSOFT CORPORATION. *Virtual Interface Architecture Specification, Version 1.0*. http://www.viarch.org, 1997.

[19] Cyclone Intelligent I/O. http://www.cyclone.com.

[20] The DAT Collaborative. http://www.datcalloborative.org.

[21] DRUSCHEL, P., AND BANGA, G. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Operating Systems Design and Implementation* (1996), pp. 261–275.

[22] DUNNING, D., REGNIER, G., MCALPINE, G., CAMERON, D., SHUBERT, B., BERRY, F., MERRITT, A. M., GRONKE, E., AND DODD, C. The Virtual Interface Architecture. *IEEE Micro 18*, 2 (1998).

[23] EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture* (May 1992), pp. 256–266.

[24] Emulex, Inc. http://www.emulex.com.

[25] EVAN SPEIGHT, H. A.-S., AND BENNETT, J. K. Realizing the performance Potential of the Virtual Interface Architecture. In *Proceedings of the International Conference on Supercomputing* (June 1999).

[26] FELTEN, E. W., ALPERT, R. D., BILAS, A., BLUMRICH, M. A., CLARK, D. W., DAMIANAKIS, S., DUBNICKI, C., IFTODE, L., AND LI, K. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture* (May 1996).

[27] GIBSON, G. A., NAGLE, D. F., AMIRI, K., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th ASPLOS* (October 1998).

[28] GigaNet. http://www.giganet.com.

[29] Architecture of the IBM System/360. http://www. research. ibm. com/ journal/ rd/ 441/ amdahl. pdf, 1964.

[30] IBM Corporation. IBM Interactive Network Dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[31] The Infiniband Trade Association. http://www.infinibandta.org, August 2000.

[32] Intel Server Adapters. http://www.intel.com.

[33] Key Performance Considerations for Selecting a Gigabit Server Adapter. http://www.intel.com/cn/gb/network/connectivity/resources/doc_library/data_sheets/Key_Performance_Gig.pdf.

[34] LANGENDOEN, K., ROMEIN, J., BHOEDJANG, R., AND BAL, H. Integrating Polling, Interrupts, and Thread Management. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation* (October 1996).

[35] Lucent Optistar GE1000. http://www.lucent.com.

[36] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA* (Berkeley, CA, USA, Jan. 1996), pp. 99–111.

[37] MOSBERGER, D., AND JIN, T. httperf – A Tool for Measuring Web Server Performance, 1998.

[38] MUIR, S., AND SMITH, J. Functional divisions in the Piglet multiprocessor operating system. In *Eighth ACM SIGOPS European Workshop* (September 1998).

[39] Myricom: Creators of Myrinet. http://www.myri.com.

[40] NATIONAL ENERGY RESEARCH SCIENTIFIC COMPUTING CENTER. *M-VIA: A High Performance Modular VIA for Linux*. http://www.nersc.gov/research/FTG/via, 1999.

[41] NATIONAL ENERGY RESEARCH SCIENTIFIC COMPUTING CENTER. *MVICH: MPI for Virtual Interface Architecture*. http://www.nersc.gov/research/FTG/mvich/index.html, 1999.

[42] PAI, V., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W., AND NAHUM, E. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems* (1998).

[43] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems 18*, 1 (2000), 37–66.

[44] PINKERTON, J. SDP: Sockets Direct Protocol. In *Infiniband Developers Conference* (Fall 2001).

[45] RANGARAJAN, M., BOHRA, A., BANERJEE, K., CARRERA, E. V., BIANCHINI, R., IFTODE, L., AND ZWAENEPOEL, W. TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation and Performance. Tech. Rep. DCS-TR-481, Rutgers University, 2002.

[46] RODRIGUES, S. H., ANDERSON, T. E., AND CULLER, D. E. High-Performance Local-Area Communication With Fast Sockets. In *Proceedings of the USENIX Technical Conference* (1997), pp. 257–274.

[47] ServerNet. http://www.servernet.com.

[48] SHAH, H. V., MINTURN, D. B., FOONG, A., MCALPINE, G. L., AND MADUKKARU-MUKUMANA, R. S. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *Proceedings of 3rd USENIX Symposium on Internet Technologies and Systems* (March 2001).

[49] SMITH, J. M., AND TRAW, C. B. S. Giving Applications Access to Gb/s Networking. *IEEE Network 7*, 4 (July 1993), 44–52.

[50] TTCP Network Benchmark. http://www.netcordia.com/network-services.html.

[51] Voltaire TCP Termination Architecture. http://www.voltaire.com/pdf/Breakingthrough-thebottleneck.pdf.

[52] Webstone 2.0.1. http://www. mindcraft. com/ webstone/ ws201-descr. html.

[53] Tornado for Intelligent Network Acceleration. http://www.windriver.com.

Appendix A

VI Primitives

The following is a list of the key VI primitives that were used in our implementation of the TCP Server.

Hardware Connection

`VipOpenNic - Open the VI NIC`

`VipCloseNic - Close the VI NIC`

VI Creation and Destruction

`VipCreateVi - Create a VI`

`VipDestroyVi - Destroy a VI`

Connection Management

`VipConnectWait - Wait for incoming VI connection requests`

`VipConnectAccept - Accept a received connection request`

`VipConnectReject - Reject a received connection request`

`VipConnectRequest - Request a VI connection`

`VipDisconnect - Disconnect an established connection`

Memory Registration

`VipRegisterMem - Register a region of memory with the VI NIC`

`VipDeregisterMem - Deregister a previously registered memory region`

Data transfer

```
VipPostSend - Post a send descriptor

VipSendDone - poll to check if send descriptor has completed

VipSendWait - blocking call to check if send descriptor has completed

VipPostRecv - Post a receive descriptor

VipRecvDone - poll to check if receive descriptor has completed

VipRecvWait - blocking call to check if receive descriptor has completed
```