MemNet: Memory-Mapped Networking for Servers

Murali Rangarajan, Kalpana Banerjee, and Liviu Iftode*
Department of Computer Science
Rutgers University, Piscataway, NJ 08854-8019
{*muralir, kalpanab, iftode*}*@cs.rutgers.edu*

Abstract

TCP/IP protocol processing has become the dominant overhead in network servers. With the advent of memory-mapped SAN technologies such as VIA and InfiniBand, the intra-server transport protocols have been made extremely efficient. Our solution to reduce the client-server network processing overhead is to offload the TCP/IP protocol from the application hosts to dedicated processors, nodes or intelligent network interface. We called this architecture a TCP Server.

In this paper, we propose a Memory-Mapped Networking API and protocol (MemNet) to enable server applications to exploit the TCP Server architecture efficiently. The idea is to use low-overhead memory-mapped communication between application hosts and the TCP server to tunnel the socket I/O calls to be processed by the TCP Server. A user-level MemNet prototype has been implemented and evaluated using a web server and real http workloads.

1 Introduction

With increasing processing power, the two main performance bottlenecks in network servers are the storage and network subsystems. Network storage needs are being stretched as file volumes grow and enterprises distribute storage requirements across a wider array of files, web applications and database servers.

Protocols like DAFS [19] help reduce the cost of file access by enabling direct access to the storage subsystem for applications using user-level memory-mapped communication [6, 7]. A significant reduction of the impact of disk I/O on performance is possible by caching, combined with server clustering and request distribution techniques which result in removing disk accesses from the critical path of request processing [26].

However, the same is not true for the network subsystem, where every outgoing data byte has to go through the same processing path in the protocol stack down to the network device. As a result, increasing service demands on today's network servers can no longer be satisfied by conventional TCP/IP protocol processing without significant performance or scalability degradation. When alleviating the cost of disk I/O through caching and other techniques, TCP/IP protocol processing can become the dominant overhead in network servers [34, 22]. Furthermore, with gigabit-per-second networking technologies, protocol and interrupt processing overheads can quickly saturate the host processor with increased network processing loads thus limiting the potential gain in network bandwidth [3].

With the advent of SANs, servers have access to high speed networks which enable efficient low latency communication by means of a programming interface which includes memory-mapped semantics. Memory-mapped communication [7, 6] helps to avoid copies and the overheads associated with interrupt processing. Recent work has focussed on two approaches for network servers, using SAN technology and intelligent devices *(i)* offloading some of the TCP/IP processing to intelligent network interface cards (I-NIC) capable of speeding up the common path of the protocol [2, 8, 11, 15, 18, 33] and *(ii)* replacing the expensive TCP/IP processing with a lightweight, more efficient transport protocol [8, 12], using user-level and memory-to-memory communication based on standards such as VIA [14] and Infiniband [17]. The first approach attempts to alleviate the overheads associated with conventional host-based network processing and the second approach attempts to achieve better server performance within the data center by exploiting the characteristics of the underlying SAN.

In [29], we introduced TCP Servers, a system architecture for offloading network processing from network-based servers to dedicated processors, nodes, or intelligent network interfaces. However, in order for server applications to benefit from the TCP/IP offloading scheme, the programming interface for networking must exploit the benefits of the low latency communication between the host and TCP Server.

*Currently with the Department of Computer Science, University of Maryland, College Park, MD 20742

In this paper, we introduce the Memory-Mapped Networking API which enables applications to offload the TCP/IP processing to TCP Servers efficiently. We describe the design and implementation of the Memory-Mapped Networking System (MemNet) which provides an implementation of the MemNet API to applications.

We have also developed a prototype which provides a reference user-level implementation of the MemNet API. The TCP Server is connected to the application host by a VIA-based SAN. We present an evaluation of the MemNet system using a web server application under real workloads.

Previous work has shown how memory-mapped communication can be used to implement lightweight transport protocols [20, 28] for intra-server SAN communication. This paper, to our best knowledge, is the first study to propose and evaluate the use of memory-mapped communication in offloading TCP/IP processing to dedicated nodes, for TCP connections over WAN.

The remainder of this paper is organized as follows. Section 2 describes our motivation for this work in detail. In Section 3, we describe related work. In Section 4, we present the MemNet system architecture and the MemNet API. Section 6 describes the details of our MemNet prototype and Section 7 presents a performance evaluation of our prototype. Section 8 presents our conclusions and future work.

2 Motivation

The general approach to achieve scalability and availability in network servers has been to construct systems comprised of a large number of functional components each performing a specific functionality. Even though the networking functionality is mapped to a set of dedicated components in such systems, there has been no clean way of separating the TCP/IP processing from application processing.

Previous studies [10, 29] have shown that the cost of TCP/IP processing often dominates the cost incurred from application processing for server applications like web servers. In fact, under heavy load conditions, servers suffer from host CPU saturation as a result of the protocol processing and frequent interruptions from asynchronous network events. Asynchronous interrupt processing and frequent context switching contribute to the overheads due to effects like cache and TLB pollution. This led us to believe that separating the TCP/IP processing to execute it on dedicated components in the system would help in improving server performance.

TCP Server [29] is a system architecture for offloading TCP/IP processing to dedicated components in the system. The performance of server applications using the TCP Server architecture relies heavily on the efficiency of the communication between the TCP Server and the application host. Current standards like the Virtual Interface Architecture (VIA) have contributed to the success of SANs by standardizing user-level memory-mapped communication between components in a SAN. Memory-mapped communication using VIA helps achieve efficient low latency communication by enabling zero-copy communication in user space. Other standards like Infiniband define a new protocol for switch-based I/O using memory-mapped communication between components in a server system.

Applications can reap the performance benefits from using memory-mapped communication for TCP/IP offloading by pre-registering buffers used in communication. To this end, we provide an API to server applications, which can be exploited to achieve an efficient offloading of TCP/IP processing from the application host to the TCP Server. In Section 4.3, we present the programming interface provided by the MemNet system to applications.

3 Related Work

OS mechanisms and policies specifically tailored for servers have been proposed in [5, 13, 27]. There have been other efforts aimed at improving server performance by avoiding interrupts [4, 21, 31] and separation of the OS from the application execution [24].

In the Communication Services Platform (CSP) project [30], the authors suggest a system architecture for scalable cluster-based servers, using dedicated network nodes and a VIA-based SAN to tunnel TCP/IP packets inside the cluster. CSP was an architecture to offload the network processing to dedicated nodes. However, their results were preliminary and their work 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.

Sockets Direct Protocol (SDP) [28] originally developed by Microsoft 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.

Direct Access Transport (DAT) [12] is an initiative to exploit features like RDMA, available in interconnect technologies like VIA [14] and Infiniband to provide a transport which includes remote memory semantics. However, the objective of DAT is to

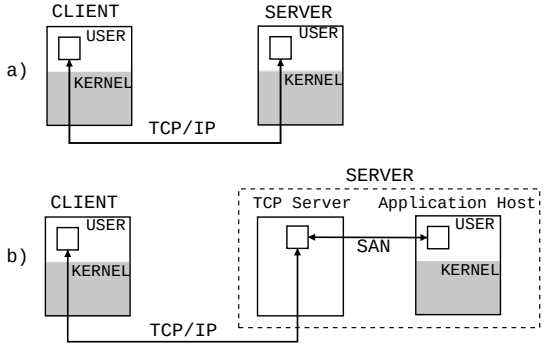

Figure 1: Network-based server architectures: a) Traditional vs b) With TCP Server

export the benefits of remote memory semantics only to intra-server communication.

Voltaire has proposed a TCP Termination Architecture [32] 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.

Intelligent network interfaces [25] have been studied, but mostly for cluster interconnects in distributed shared memory [16] or distributed file systems [3]. Recently released network interface cards have been equipped with hardware support to offload the TCP/IP protocol processing from the host [1, 2, 11, 15, 18, 33]. Some of these cards also provide support to offload networking protocol processing for network attached storage devices including iSCSI, from software on the host processor to dedicated hardware on the adapter. Modeling and simulation were used in [9] 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 offloading file system and networking functions from the host to intelligent devices.

QPIP [8] 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 alternative interface to the traditional sockets API but does not formalize or define a programming interface that can be exploited by applications to achieve better performance. Moreover, performance evaluation presented in [8] was limited to communication between QP-aware applications at both endpoints over a SAN.

4 MemNet System

The MemNet system relies on the TCP Server architecture [29] to offload the TCP/IP processing from the application host to a dedicated node. Figure 1 presents two architectures for network-based servers: a traditional architecture and an architecture based on TCP Servers. The TCP Server acts as the network endpoint for the outside world. Network data is tunneled between the application host and the TCP Server across the SAN using VI channels. We present a brief overview of the TCP Server architecture in Section 4.1. In Section 4.2, we present the MemNet system architecture and in Section 4.3, we describe the application programming interface provided by the MemNet system.

4.1 TCP Server Overview

TCP Server is a system architecture for offloading network processing from the application hosts to dedicated processors, nodes, or intelligent network interfaces. This separation aims to improve server performance by isolating the application from OS intrusion, and by removing the harmful effect of co-habitation of various OS services like cache pollution.

The application host avoids TCP/IP processing by tunneling socket I/O calls to the TCP Server using fast communication channels. In effect, TCP tunneling transforms socket calls into lightweight remote procedure calls. As the goal of TCP/IP offloading is to save network processing overhead at the host, using a faster and lighter communication channel for tunneling is essential.

The design of the TCP Server architecture addresses several key issues which can significantly impact the overall performance of the server. An explanation of these issues and their impact on the application programming interface and performance can be found in [29].

4.2 System Architecture

The MemNet system architecture is presented in Figure 2. The server application uses the MemNet Socket API to access the networking subsystem. The MemNet Socket Stub tunnels all the MemNet API calls over the SAN to the MemNet Socket Provider on the TCP Server. The MemNet Socket Provider performs the corresponding socket operation using the TCP/IP Socket Provider on the TCP Server and returns the result of the call to the application over the SAN. The MemNet API Provider uses VI channels to communicate with the MemNet Socket Provider on the TCP Server.

In the figure, the solid lines represent the flow of data to and from the server application. The dotted lines represent the path traversed in establishing a connection between the MemNet API Provider on the application host and the MemNet Socket Provider on the TCP Server. The MemNet system keeps the

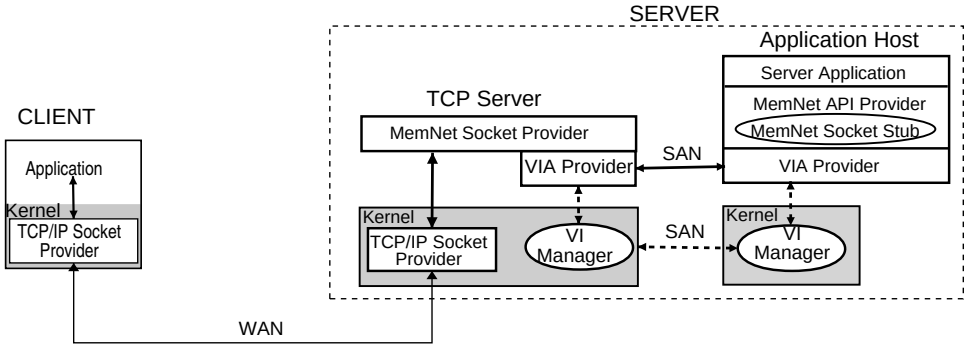

Figure 2: MemNet System Architecture

register_mem(IN buffer, IN length, OUT mem_handle)
deregister_mem(IN mem_handle)

sync_send(IN socket, IN mem_desc)
sync_recv(IN socket, IN mem_desc)
async_send(IN socket, IN mem_desc, OUT job_desc)
async_recv(IN socket, IN mem_desc, OUT job_desc)

job_wait(IN job_desc, IN timeout)
job_done(IN job_desc)

Figure 3: MemNet API Primitives

average latency of the MemNet primitives low by implementing frequently used primitives like send/recv entirely in user space. The kernel(VI Manager) is involved in the critical path only when VI channels are set up for communication between the MemNet Socket Stub and the MemNet Socket Provider, or when the application registers/deregisters memory for communication.

4.3 MemNet Programming Interface

Figure 3 lists the key primitives provided by the MemNet API. In the figure, input parameters are indicated by the keyword IN and output parameters are indicated by OUT.

The MemNet API allows applications to perform sends and receives both synchronously and asynchronously. The send/receive primitives provided by the MemNet API allow data to be transferred directly to and from application buffers. In order to achieve this, the application needs to register its communication buffers with the MemNet system. `register_mem` and `deregister_mem` enable the application to register and deregister memory with the MemNet system.

The `async_send/async_recv` primitives immediately return job descriptors to the application. The job descriptors can be used by the application to check the completion status of asynchronous operations. The application has the option of using `job_wait` or `job_done` to wait or poll respectively, for completion of the asynchronous operation specified in the job descriptor. To guarantee correctness,

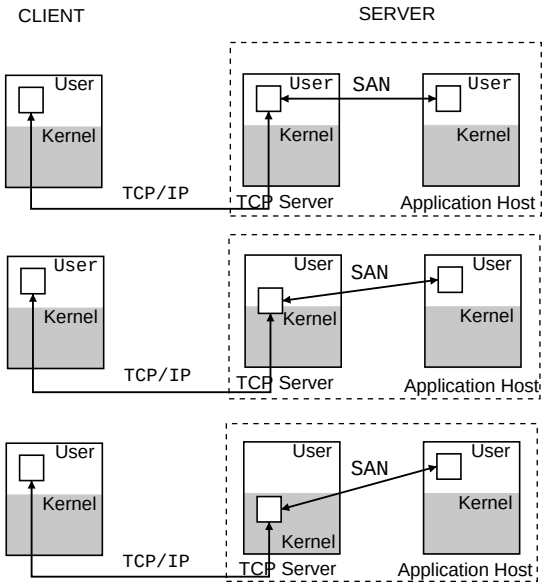

Figure 4: TCP Server Design Alternatives

the MemNet system assumes that applications do not overwrite buffers specified in `mem_desc` as part of an asynchronous operation before the operation completes.

5 Design Alternatives

Figure 4 shows the various design alternatives to implement the TCP Server. In the first two alternatives, the MemNet Socket Provider is implemented in user space. In the first alternative, the standard Linux socket implementation is used for the TCP/IP Socket Provider. With this approach, we still pay the complete cost of socket operations but on the TCP Server instead of the host. The second alternative avoids the copy between user-space and kernel-space by sharing the data buffers between the user-space MemNet Socket Provider and the kernel. The third alternative avoids the copy between user-space and the kernel by using an in-kernel MemNet Socket provider. We used the first alternative for the prototype described

in this paper. We are currently working on an optimized implementation based on the second and third alternatives.

6 MemNet Prototype

We developed a prototype of the MemNet system using PCs connected by a VIA-based SAN. We used two 300 MHz Pentium II PCs, one each for the application host and the TCP server, that communicate over 32-bit 33 MHz Emulex cLAN interfaces and an 8-port Emulex switch. The TCP Server was installed with a 3Com 3c996B-T Gigabit Ethernet adapter. Both the host and the TCP server run Linux-2.4.16.

Microbenchmarks reported in Table 1 reveal performance characteristics of the VIA-based SAN used in our prototype. The Send overhead denotes the average time taken to post a send request using VIA.

One-way latency(μs)	Bandwidth (MB/s)	Send overhead(μs)
8.6 (4B pkt)	96 (16KB pkt)	2.1

Table 1: Emulex VIA Microbenchmarks

6.1 Basic Framework

Each socket used by the application is mapped to a VI channel and has a corresponding socket endpoint on the TCP Server. The MemNet system associates with each VI channel a registered memory region which is used internally by the system. Since the mapping of a socket to a VI and its associated memory regions is maintained for the lifetime of the socket, these memory regions can be used by the MemNet system to perform RDMA transfers of control information and data between the application and the TCP Server. These memory regions include the send and receive buffers associated with each socket. An RDMA-based signalling scheme is used for flow control between the application and the TCP Server, for using the socket send and receive buffers.

As creating VIs and connecting them are expensive operations, the MemNet API Provider creates a pool of VIs and requests connections on them from the MemNet Socket Provider, at the time of initialization. The MemNet Socket Provider is a multi-threaded user-level process running on the TCP Server. The main thread of the MemNet Socket Provider accepts or rejects VI connection requests from the host depending on its existing load. On accepting a VI connection request, the main thread then hands over this VI connection to a worker thread which is then responsible for handling all data transfers on that VI.

We have used the standard Linux socket implementation for the TCP/IP Socket Provider in our prototype. This guarantees reliable transmission of data once a socket send is performed on the TCP Server. To guarantee correct operation, buffers used in send should not be overwritten until the entire buffer is sent to the TCP Server. In `sync_send`, control returns to the application only after the entire buffer is sent using the TCP/IP Socket Provider. In `async_send`, control returns to the application as soon as the send is posted on the VI channel corresponding to the socket. The application has to avoid overwriting buffers used in asynchronous sends until the operation completes.

6.2 Implementation of MemNet API

When the application invokes a primitive which is part of the traditional socket API, the MemNet Socket Stub tunnels the socket call parameters to the MemNet Socket Provider on the TCP Server. The MemNet Socket Provider processes the socket call and responds to the application host, and the socket call then returns to the application. In the case of the send primitive, this includes a transfer of data from the application buffers to the MemNet Socket Provider on the TCP Server. In the case of the receive, this includes a transfer of received data from the MemNet Socket Provider to the application buffers.

The primitives for memory registration and deregistration are directly mapped to the corresponding primitives available from the VIA Provider. The registration process pins the buffer being registered to physical memory and registers the buffer with the underlying VIA Provider.

The implementation of the `sync_send/sync_recv` primitives is similar to that of the traditional socket API primitives. The application uses registered memory for communication buffers and this enables the MemNet system to transfer the data to the TCP/IP Socket Provider on the TCP Server without any intermediate copies.

The asynchronous send/receive primitives return to the application as soon as the arguments to the primitives are tunneled to the TCP Server. The MemNet system returns a job descriptor to the application for the asynchronous primitive just invoked. The MemNet Socket Provider performs the operation in the background and returns the result to the application host asynchronously without interrupting the application. The application uses the job descriptor to check the completion status of the asynchronous primitive. An RDMA-based flow control mechanism is used to control the number of requests that can be pipelined to the TCP Server.

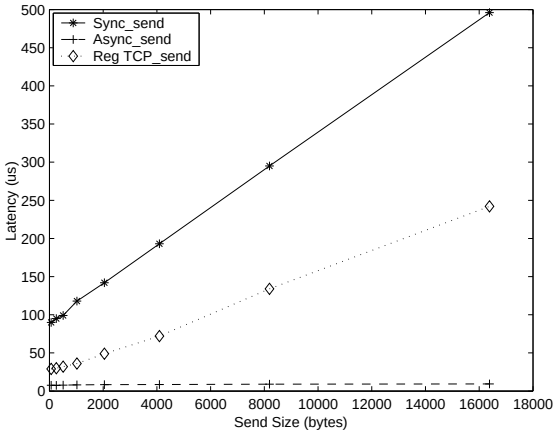

Figure 5: Cost of `send`

Optimizations: Our prototype also includes two optimizations to improve the performance of server applications.

- **Eager Receive** is an optimization to the network receive processing. The TCP Server eagerly performs receive operations on behalf of the host and when the application issues a receive call, data is transferred from the TCP Server to the application host. The TCP Server posts receive eagerly for a number of bytes, and continues with further eager receive processing depending on the rate of data consumed by the host. The MemNet Socket Provider uses the `poll` system call to verify if any data is ready to be read from that socket before issuing an eager recv. The MemNet Socket Provider keeps the received data on the TCP Server and transfers directly into the application buffers when the application invokes a receive.

- **Eager Accept** is an optimization to the connection processing. A dedicated thread of the MemNet Socket Provider eagerly accepts connections upto a pre-determined maximum. When the application issues an `accept`, one of the previously accepted connections is returned. We expect this optimization to reduce the processing time for the `accept` primitive.

7 MemNet Performance

In our prototype, each call to the MemNet API is tunneled through VI channels to the TCP Server. In Figure 5, we compare the latency perceived by applications for synchronous and asynchronous sends in the MemNet system with the latency of send in a traditional system (Reg TCP_send), for a stream socket. We chose the send primitive as the send plays an important role in server applications like web servers.

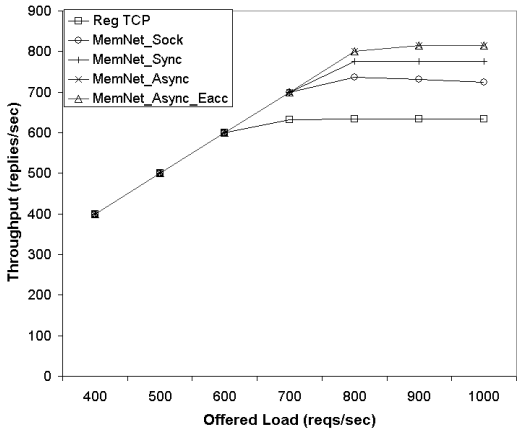

Figure 6: Web server throughput with HTTP/1.0

We can see that the cost of `async_send` ($< 8\mu$s) is close to the cost of posting a send on the VI channel (Table 1). However, the `sync_send` is expensive since it includes the cost of tunneling over the SAN and the cost of a traditional socket send at the TCP Server. `async_send` allows applications to hide the latency of the send by returning to the application immediately after the send is posted on the VI channel.

7.1 Web Server Performance

We evaluate the MemNet system by analyzing the performance of a simple multi-threaded http server. The MemNet API subsumes the traditional socket API. We compare the performance of the http server using the traditional socket API in the MemNet system (*MemNet_Sock*) and using the primitives provided by the MemNet API (*MemNet_Sync* and *MemNet_Async*) which require buffers used in communication to be pre-registered. We also present the performance of the http server using a standalone Linux host-based socket implementation (*Reg TCP*) for comparison.

We used httperf [23] as the client benchmarking tool to generate the required workloads. We used a standalone PC with an unmodified Linux socket implementation for the client. We present the performance analysis for a synthetic workload using HTTP 1.0 and HTTP 1.1.

Throughput with HTTP/1.0: The workload used for HTTP/1.0 consists of requests for 16-KByte static files, making sure that the requested file is not available in the L2 hardware cache. Figure 6 shows the throughput of the web server as a function of the offered load in requests/second. All systems are able to satisfy the offered load at low request rates. At high request rates, we see a difference in performance when *Reg TCP* saturates at an offered load of 700 requests/second. The http server shows an improvement of 15% in performance with *MemNet_Sock*

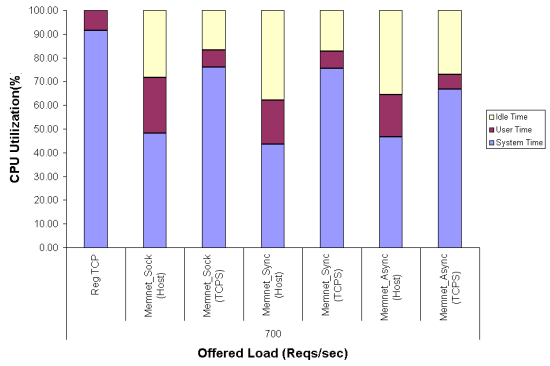

Figure 7: CPU usage for web server using HTTP/1.0

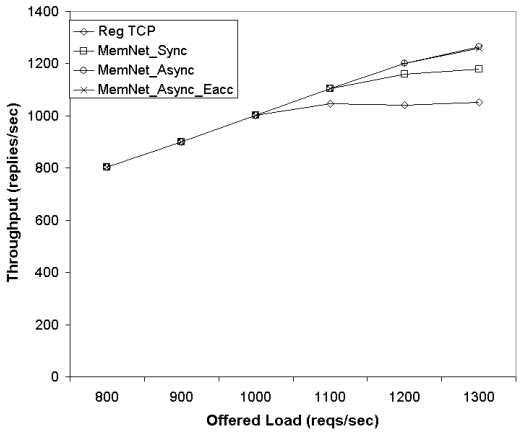

Figure 8: Web server throughput with HTTP/1.1

over *Reg TCP*. Using the synchronous primitives in the MemNet API (*MemNet_Sync*), the http server is able to achieve a performance improvement of 22%. *MemNet_Async* shows a performance gain of about 30% with the http server using asynchronous primitives like `async_send`. *MemNet_Sync* shows a gain in performance due to offloading of network sends to the TCP Server. *MemNet_Async* in addition allows a better pipelining of network sends and helps the application overlap the latency of offloading the send primitive over the SAN with computation at the host. *MemNet_Async_Eacc* includes the *Eager Accept* optimization in addition to *MemNet_Async*. This provided no additional gain since it is not the connection time, but the actual request processing time that dominates the network processing.

We also observed that the *Eager Receive* optimization (not presented) does not contribute to any performance gain. The TCP Server uses the `poll` system call to verify if data has arrived on a given socket and this leads to a slight performance degradation at high request rates by taking up some CPU time when the TCP Server is already saturated.

In Figure 7, we present the CPU utilization on the application host (and TCP Server) for various systems at the load when *RegTCP* saturates. At this load, the host CPU saturates for *RegTCP* whereas the *MemNet_Sync (Host)* and *MemNet_Async (Host)* have about 40% idle time. With *MemNet_Sock*, since the http server uses only the traditional socket based MemNet API, it does not pre-register buffers used in communication. As a result, copies take up CPU time and reduce the idle time in *MemNet_Sock (Host)* to 29%. For *MemNet_Sock*, *MemNet_Sync* and *MemNet_Async*, We also present the CPU utilization on the TCP Server (*MemNet_Sock (TCPS) MemNet_Sync (TCPS) and MemNet_Async (TCPS)*) to show that the entire TCP/IP processing overhead has been shifted to the TCP Server in these systems. We have also observed that at higher loads, the network processing at the TCP Server proves to be the bottleneck and eventually saturates the processor on the TCP Server.

Greater gains are not possible with this workload because the TCP Server node is excessively loaded. In fact, this explains why our optimizations of *Eager Receive* and *Eager Accept*, do not improve throughput beyond that of *MemNet_Async*. These optimizations are intended to improve the performance of the host application at the cost of more processing at the TCP server node. However, 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 load balancing between the application host and TCP Server (either statically or dynamically), by using a faster TCP server, or by using multiple TCP servers per application node. We are currently investigating these approaches.

Throughput with HTTP/1.1: 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 HTTP/1.1 is the same as that used for HTTP/1.0. However, multiple requests (six) were sent over each socket connection, in bursts of three. Figure 8 shows the http server throughput in this case. The performance gain achieved by *MemNet_Sync* is about 12%, and by *MemNet_Async* is 20%, over that of *Reg TCP*. These performance gains, are lower than those achieved with HTTP/1.0. However, they show that our MemNet system 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.

8 Conclusions and Future Work

In this paper, we described the Memory-Mapped Networking system, which proposes a new programming interface for network-based server applications. The MemNet system uses a lightweight protocol for offloading the network processing to a TCP Server connected to the application host by a high speed interconnect. The MemNet system enables a new paradigm for access to the networking subsystem that goes beyond the capabilities of traditional systems.

We have implemented a prototype of the MemNet system in user-space. We have shown that performance gains of upto 30% can be achieved for a web server application using the MemNet system compared to a standalone host-based socket implementation. We have also shown that the application was able to achieve maximum performance gain using MemNet API primitives which require pre-registered buffers for communication. A performance gain of 15% was obtained by the application using pre-registered buffers compared to using primitives which do not require pre-registered buffers.

We are currently working on an optimized implementation of the TCP Server which can avoid the data copying from user-space to kernel-space in the TCP/IP Socket Provider. We are also currently enhancing our MemNet system prototype to support the complete socket API.

References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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).

[7] Blumrich, M., Li, K., Alpert, R., Dubnicki, C., Felten, E., and Sandberg, J. A virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture* (Apr. 1994), pp. 142–153.

[8] 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).

[9] 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).

[10] Chase, J., Yocum, K., and Gallatin, A. End-System Optimizations for High-Speed TCP. IEEE Communications, special issue on TCP Performance in Future Networking Environments, vol. 39 no. 4, April 2001, 2001.

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

[12] The DAT Collaborative. http://www.datcollaborative.org.

[13] 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.

[14] 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).

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

[16] 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).

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

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

[19] Katcher, J., and Kleiman, S. An Introduction to the Direct Access File System, 6 2000.

[20] Kim, J.-S., Kim, K., and Jung, S.-I. Building a high-performance communication layer over virtual interface architecture on Linux clusters. In *Proceedings of the International Conference on Supercomputing.*

[21] 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).

[22] 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.

[23] Mosberger, D., and Jin, T. httperf – a tool for measureing web server performance, 1998.

[24] Muir, S., and Smith, J. Functional divisions in the Piglet multiprocessor operating system. In *Eighth ACM SIGOPS European Workshop* (September 1998).

[25] Myricom: Creators of myrinet. http://www.myri.com.

[26] 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 International Conference on Architectural Support for Programming Languages and Operating Systems* (1998).

[27] 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.

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

[29] Rangarajan, M., Bohra, A., Banerjee, K., Carrera, E. V., Bianchini, R., and Iftode, L. TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation, and Performance. Submitted for publication. Rutgers University, Department of Computer Science Technical Report, DCS-TR-481, March 2002.

[30] Shah, H. V., Minturn, D. B., Foong, A., McAlpine, G. L., and Madukkarumukumana, 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).

[31] Smith, J. M., and Traw, C. B. S. Giving Applications Access to Gb/s Networking. *IEEE Network 7*, 4 (July 1993), 44–52.

[32] Voltaire TCP Termination Architecture. http://www.voltaire.com/pdf/Breaking through the bottleneck.pdf.

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

[34] Yiming Hu, Ashwini Nanda, Q. Y. Measurement analysis and performance improvement of the apache web server. Tech. Rep. 1097-0001, University of Rhode Island, Department of Electrical and Computer Engineering, October 1997.