MemNet: Efficient Offloading of TCP/IP Processing Using Memory-Mapped Communication *

Murali Rangarajan†, Kalpana Banerjee†, Jihwang Yeo‡ and Liviu Iftode‡
† *Department of Computer Science*
Rutgers University, Piscataway, NJ 08854-8019
{*muralir, kalpanab*}@*cs.rutgers.edu*
‡ *Department of Computer Science*
University of Maryland, College Park, MD 20742
{*jyeo, iftode*}@*cs.umd.edu*

Abstract

TCP/IP protocol processing has become the dominant overhead in network servers. Solutions to alleviate the overheads related to TCP/IP processing have focused on two approaches *(i)* minimizing the overheads using techniques like zero copy and asynchronous processing. *(ii)* offloading some (or all) of the TCP/IP processing to dedicated processors or nodes.

In this paper, we propose a Memory-Mapped Networking API and protocol (MemNet) to enable an efficient offloading of TCP/IP processing over a memory-mapped interconnect using lightweight intra-server protocols. To reduce the network processing overhead at the server, we had previously proposed the TCP Server architecture to offload TCP/IP processing from application hosts to dedicated processors, nodes or intelligent network interface. In MemNet, the idea is to export to the application, the benefits of the low-overhead memory-mapped communication used to offload the TCP/IP processing. We describe the implementation of a user-level MemNet prototype and present a performance evaluation to demonstrate an improvement in throughput for a web server using the MemNet API.

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 [20] help reduce the cost of file access by enabling direct access to remote storage for applications, using user-level memory-mapped communication [6, 7]. A significant reduction of the impact of disk I/O on performance is also possible by caching, combined with server clustering and request distribution techniques which result in removing disk accesses from the critical path of request processing [28].

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 [36, 24]. 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].

Two approaches have been proposed to alleviate the overheads involved in TCP/IP networking: *(i)* minimizing the overheads involved in TCP/IP processing using techniques like zero copy I/O [17, 23], unified buffering [29], and asynchronous socket processing. *(ii)* offloading some (or all) of the TCP/IP processing to intelligent network interface cards capable of speeding up the common path of the protocol [2, 8, 11, 15, 19, 35]. This approach alleviates the overheads associated with conventional host-based network processing. Other work has been done on confining execution of the TCP/IP protocol, system calls, and network interrupts to a dedicated processor of a multiprocessor server, but limited results have been reported [26].

In [31], we introduced TCP Servers, a system architecture for offloading network processing in net-

*This work is supported in part by the National Science Foundation under the NSF CAREER CCR 0133366.

work servers to dedicated processors, nodes, or intelligent network interfaces. Compared to hardware-based offloading approaches, the TCP Server architecture provides a generic and flexible mechanism for offloading TCP/IP processing over a low overhead communication channel. The efficiency of the communication between the host and the TCP Server node(s) plays an important role in the performance of the server applications: *(i)* Using a low-overhead fast communication medium between the host and the TCP Server node(s) is crucial to the performance of applications using the TCP Server architecture. *(ii)* For server applications to benefit from the TCP/IP offloading scheme, applications must be able to exploit the benefits of the low-overhead communication medium used for offloading.

In this paper, we introduce the Memory-Mapped Networking API which provides a programming interface to applications, enabling an efficient offloading of TCP/IP processing to TCP Servers. The MemNet API provides primitives allowing applications to efficiently offload the TCP/IP processing, using techniques like zero copy and asynchronous processing. We describe the design and implementation of the Memory-Mapped Networking System (MemNet) in user-space, which exports the MemNet API to applications. MemNet provides a framework for offloading TCP/IP processing from the host processor to an intelligent network interface using memory-mapped communication. In our implementation, the TCP/IP processing is offloaded from the application host to the TCP Server using a VIA-based SAN. Using a web server application and real workloads, we present an evaluation of the MemNet system to show that server applications can achieve an efficient offloading of TCP/IP processing using the MemNet API.

In the area of TCP/IP networking within a SAN, recent work has focused on replacing the expensive TCP/IP processing with a more efficient lightweight transport protocol [12], based on user-level memory-mapped communication such as VIA [14] and Infiniband [18]. [21, 30] have shown how features of memory-mapped communication can be used to implement lightweight transport protocols for TCP/IP connections within a SAN. 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 a 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 5 describes the details of our MemNet prototype and Section 6 presents a performance evaluation of our prototype. Section 7 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 in such systems, the networking functionality is mapped to a set of dedicated components, there has been no clean way of separating the TCP/IP processing from application processing.

Previous studies [10, 31] 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 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 [31] 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. This is determined by two factors (1) the efficiency of the communication layer and (2) the ability of applications to take advantage of the communication layer.

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.

MemNet enables applications to use features of the underlying memory-mapped communication layer to offload the TCP/IP processing efficiently in the following two ways.

1. The MemNet API provides primitives which allow applications to transfer data directly to and from application buffers without intermediate copies.

2. The MemNet API provides support for performing socket operations asynchronously. Offloading the TCP/IP processing over a SAN adds the latency of a round trip across the SAN to the critical path of processing for every socket primitive. The asynchronous primitives allow applications

to hide the latency of the round trip to the TCP Server, by overlapping it with useful computation at the host.

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, 29]. Other efforts have tried to improve server performance by avoiding interrupts [4, 22, 33] and separating the OS from the application execution [26].

In the Communication Services Platform (CSP) project [32], 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 is an architecture to offload the network processing to dedicated nodes. However, their results were preliminary and their work does not explore the issue of providing a programming interface which allows server applications to exploit features of the underlying low-latency memory-mapped communication layer.

Sockets Direct Protocol (SDP) [30], 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 export the benefits of remote memory semantics only to communication within a SAN.

Voltaire [34] has proposed a TCP Termination Architecture 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 [27] 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, 19, 35]. 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.

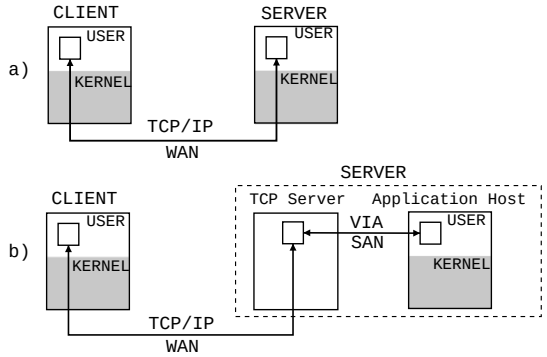

Figure 1: Network-based server architectures: a) Traditional b) With TCP Server

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 alternate 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 uses the TCP Server architecture [31] to offload the TCP/IP processing from the application host to a dedicated node. Figure 1 presents instances of two architectures for network-based servers, the first one based on the traditional architecture and the second one based on the TCP Server architecture. In Figure 1(b), 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

The TCP Server architecture provides a framework 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

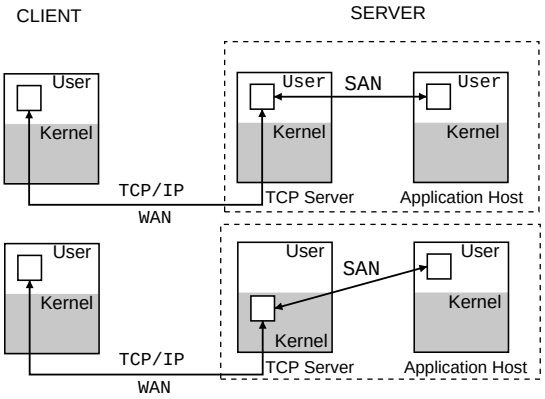

Figure 2: TCP Server Design Alternatives

from OS intrusion, and by removing the harmful effects 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. A detailed explanation of issues involved in the design of the TCP Server architecture and their impact on the application programming interface and performance can be found in [31].

Figure 2 shows two different design alternatives for the implementation on the TCP Server. In the first alternative, the TCP Server is implemented in user space and uses an unmodified native BSD socket implementation. With this approach, we still pay the complete cost of socket operations but on the TCP Server instead of the host. The second alternative uses an in-kernel implementation on the TCP Server. This avoids the overheads of redundant user-space processing, traps into the OS kernel and copying between the user-space and the kernel. We are currently working on an optimized in-kernel implementation on the TCP Server based on the second alternative. In this paper, we describe a prototype based on the first alternative.

4.2 System Architecture

The MemNet system architecture is presented in Figure 3. The server application uses the MemNet Socket API to access the networking subsystem. The MemNet API Provider tunnels all the MemNet API calls over the SAN to the MemNet Socket Provider on the TCP Server. The MemNet Socket Provider uses the native BSD Sockets implementation on the TCP Server to perform the required socket operation 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 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 API Provider and the MemNet Socket Provider, or when the application registers/deregisters memory.

4.3 MemNet Programming Interface

Figure 4 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 provides applications with two key features.

- *Direct data transfer to and from application buffers:* The `send_direct/recv_direct` 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 pre-register its communication buffers with the MemNet system. The `register_mem` primitive returns a memory handle which can be used by the application in `send_direct` and `recv_direct`.

- *Asynchronous operations:* The `async_send` and `async_recv` primitives remove the latency of waiting for completion of the operation from the critical path. These primitives post the request on the communication channel to the TCP Server and 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. `async_send` and `async_recv` require the application to pre-register the communication buffers and pass in memory handles to specify the communication buffers. To guarantee correctness, applications must not overwrite buffers specified in the `mem_handle` passed to the asynchronous primitives, before the operation completes.

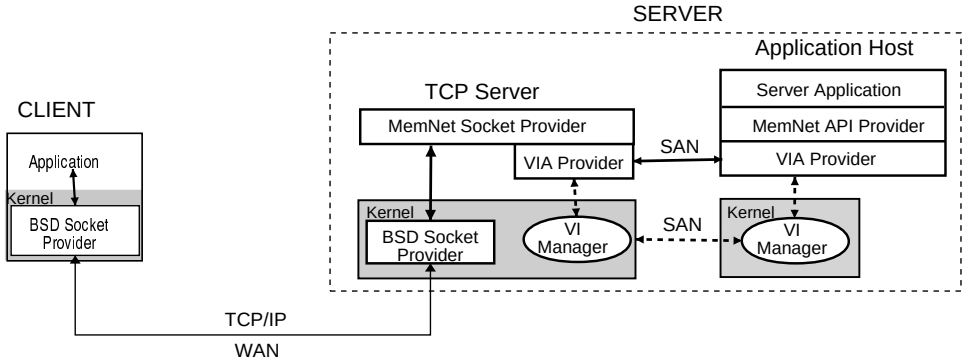

Figure 3: MemNet System Architecture

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

send_direct(IN socket, IN mem_handle, IN len)
recv_direct(IN socket, IN mem_handle, IN len)
async_send(IN socket, IN mem_handle, IN len, OUT job_desc)
async_recv(IN socket, IN mem_handle, IN len, OUT job_desc)

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

Figure 4: MemNet API Primitives

5 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

5.1 Basic Framework

Each socket used by the application is mapped to a VI channel and has a corresponding real 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. Also associated with each VI channel is a set of descriptors which are used for sending and receiving data across the VI channel.

An RDMA-based signalling scheme is used for flow control between the application and the TCP Server, for using the descriptors as well as the send and receive buffers associated with each socket. The RDMA-based flow control is also used in the context of `async_send/async_recv`, to regulate the length of the RPC pipeline.

The memory regions associated with each VI channel are used mainly to perform RDMA transfers of control information between the application and the TCP Server. When the application does not use pre-registered buffers, it cannot use the MemNet primitives (`send_direct/recv_direct`) which enable direct transfer of data to and from the application buffers. In this scenario, the memory regions are also used for the send and receive buffers associated with each socket.

As creating VIs and connecting them are expensive operations, at the time of initialization, 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 current load. On accepting a VI connection request, the main thread hands over the VI connection to a worker thread responsible for handling all data transfers on that VI.

In our prototype, we have used the native Linux-based BSD socket implementation on the TCP Server. This guarantees reliable transmission of data

once a socket send is performed on the TCP Server. In `send_direct`, control returns to the application only after the entire buffer is sent using the BSD socket implementation. 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 must avoid overwriting buffers used in asynchronous sends until the operation completes.

5.2 Implementation of MemNet API

When the application invokes a primitive of the traditional socket API, the MemNet API Provider 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`, this includes a transfer of data from the application buffers to the MemNet Socket Provider on the TCP Server. In the case of the `recv`, 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 `send_direct` and `recv_direct` primitives is similar to that of the traditional socket API primitives. The application uses registered memory for communication buffers, enabling the the data transfer 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 their arguments 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.

6 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`). We chose to examine the send primitive as it plays an important role in server applications like web servers.

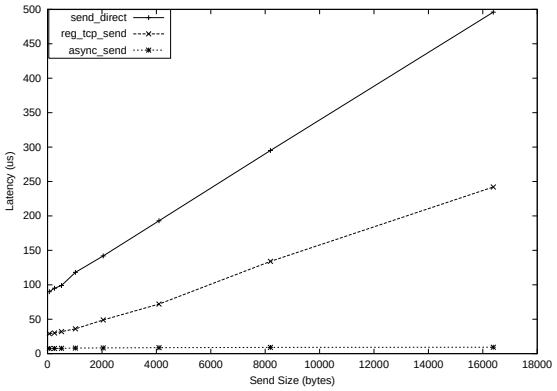

Figure 5: Cost of `send`

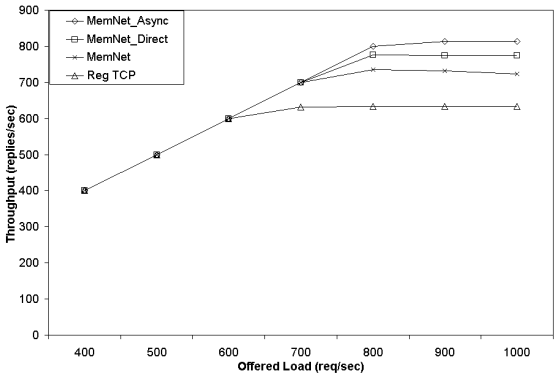

Figure 6: Web server throughput with HTTP/1.0

We can see that `async_send` is least expensive and the cost ($< 8\mu$s) is close to the cost of posting a send on the VI channel (Table 1). However, `send_direct` 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.

6.1 Web Server Performance

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

We used httperf [25] as the client benchmarking tool to generate the required workloads. We used a standalone PC with an unmodified Linux socket im-

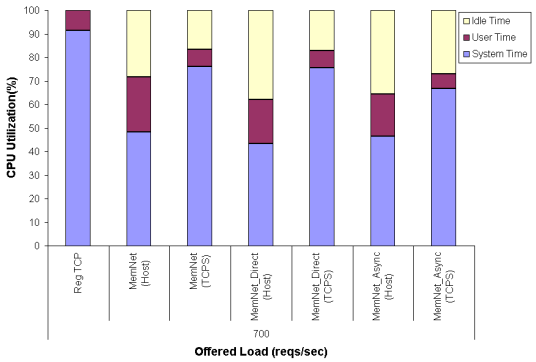

Figure 7: CPU usage for web server using HTTP/1.0

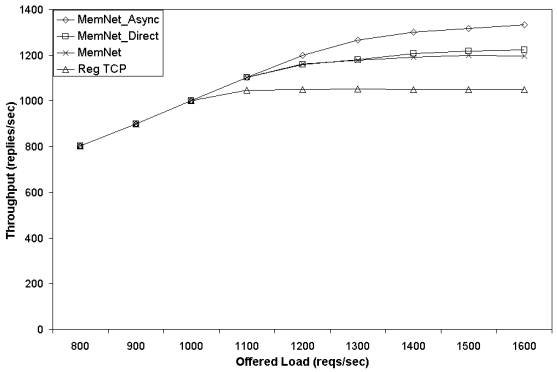

Figure 8: Web server throughput with HTTP/1.1

plementation 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: 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. *Reg TCP* saturates at high request rates and we see a difference in performance between *Reg TCP* and the other implementations. The web server shows an improvement of 15% in performance with *MemNet* over *Reg TCP*. Using the `send_direct/recv_direct` primitives in the MemNet API (*MemNet_Direct*), the web server is able to achieve a performance improvement of 22%. *MemNet_Async* shows a performance gain of about 30% with the web server using asynchronous primitives like `async_send`. *MemNet_Direct* shows a gain in performance as a result of avoiding copies on the host and due to the offloading of network sends to the TCP Server. *MemNet_Async* in addition allows a better pipelining of network sends and helps the application hide the latency of offloading the send primitive over the SAN by overlapping it with computation at the host.

In Figure 7, we present the CPU utilization on the application host (Host) and TCP Server (TCPS) for various systems, at the load at which *RegTCP* saturates. At this load, the host CPU saturates for *RegTCP* whereas the *MemNet_Direct (Host)* and *MemNet_Async (Host)* have about 40% idle time. With *MemNet*, since the web 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 (Host)* to 29%. For *MemNet*, *MemNet_Direct* and *MemNet_Async*, we also present the CPU utilization on the TCP Server (denoted by *(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 saturated at high request rates. The user-space implementation on the TCP Server also imposes overheads in terms of redundant processing by the MemNet Socket Provider and copies between the MemNet Socket Provider and the kernel. We expect our in-kernel implementation of the MemNet Socket Provider to alleviate this problem and further improve the performance of server applications. This problem can also be alleviated by a flexible offloading of TCP/IP processing from the host based on the load on the application host and the TCP Server.

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 except for sending multiple requests (six) over each socket connection, in bursts of three. Figure 8 shows the web server throughput in this case. With *MemNet*, the web server demonstrates only a gain of 13%, since with *Reg TCP*, the optimized HTTP/1.1 protocol is able to reduce the overheads associated with the network processing. Using the features of the MemNet API, the web server is able to achieve higher performance gains. The performance gain achieved by *MemNet_Direct* is about 17%, and by *MemNet_Async* is 27%, These are slightly 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 server applications.

7 Conclusions and Future Work

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

We have described a prototype of the MemNet system implemented in user-space. We have shown that the MemNet system enables an efficient offloading of TCP/IP processing over a memory-mapped interconnect, using features like zero copy and asynchronous processing. We have shown that performance gains of up to 30% can be achieved for a web server application using features provided in the MemNet API, compared to a standalone host-based socket implementation. About 15% of this gain can be attributed to the use of primitives provided by the MemNet API. We have shown that the application was able to achieve maximum performance gain using MemNet API primitives that require pre-registered buffers for communication.

For the workloads we studied, we realized that the TCP Server saturated and became the bottleneck at high request rates. In the prototype described in this paper, the MemNet Socket Provider is implemented in user-space and uses the standard BSD socket implementation available in the Linux kernel. This contributes to overheads which limits the performance gains achieved by server applications. We are currently working on an optimized in-kernel implementation of the MemNet Socket Provider which can avoid redundant user-space processing, traps into the OS, and data copying from user-space to kernel-space on the TCP Server. We believe that with the in-kernel implementation, we will be able to achieve higher performance gains.

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] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *USENIX Annual Technical Conference* (1996), pp. 253–264.

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

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

[20] KATCHER, J., AND KLEIMAN, S. An Introduction to the Direct Access File System, 6 2000.

[21] 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* (June 2001).

[22] 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).

[23] M. THADANI AND Y. KHALIDI. An efficient zero-copy I/O framework for UNIX, 1995.

[24] 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.

[25] MOSBERGER, D., AND JIN, T. httperf – a tool for measureing web server performance, 1998.

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

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

[28] 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).

[29] 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.

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

[31] RANGARAJAN, M., BOHRA, A., BANERJEE, K., CARRERA, E. V., BIANCHINI, R., IFTODE, L., AND ZWAENEPOEL, W. TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance. Submitted for publication. Rutgers University, Department of Computer Science Technical Report, DCS-TR-481, March 2002.

[32] 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).

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

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

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

[36] 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.