Impact of High Performance Sockets on Data Intensive Applications

PAVAN BALAJI, JIESHENG WU, TAHSIN KURC, UMIT CATALYUREK, DHABALESWAR K. PANDA, AND JOEL SALTZ

Technical Report
OSU-CISRC-1/03-TR05

Impact of High Performance Sockets on Data Intensive Applications *

Pavan Balaji†, Jiesheng Wu†, Tahsin Kurc‡,
Umit Catalyurek‡, Dhabaleswar K. Panda†, Joel Saltz‡

† Dept. of Computer and Information Science
The Ohio State University, Columbus, OH, 43210
{balaji,wuj,panda}@cis.ohio-state.edu

‡ Dept. of Biomedical Informatics
The Ohio State University, Columbus, OH, 43210
{kurc.1,catalyurek.1,saltz.3}@osu.edu

Abstract

PC clusters have become cost-effective alternatives to high-end computing systems for both compute-intensive and data-intensive applications. With the advent of user-level protocols like the Virtual Interface Architecture (VIA), the latency and bandwidth experienced by applications, has approached to that of the physical network on clusters. However, taking advantage of these protocols in the existing applications is still a challenging problem, as they provide only the low-level interfaces that support the minimum functionality required for inter-processor communication. In this paper, we design, develop and evaluate the performance of a socket layer on top of VIA, referred to here as SocketVIA*, to support applications implemented using sockets on TCP/IP. We experimentally evaluate the efficiency of this substrate using both micro-benchmarks and a component framework designed to provide runtime support for data intensive applications. Our results show that the sockets layer achieves a latency of 9.1μs for 4 byte messages, compared to the 45μs of TCP, and a peak bandwidth of 763Mbps compared to the 510Mbps given by TCP. The experimental results also show that the different performance characteristics of SocketVIA allow a more efficient partitioning of data at the source nodes, thus further improving the performance. This improvement is of an order of magnitude in some cases.*

Keywords: VIA, High-Performance Networking, Sockets, Data Intensive Computing, PC Clusters

*This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ACI-9982087, #EIA-9986052 and #CCR-0204429, Lawrence Livermore National Laboratory under Grant #B500288 and #B517095 (UC Subcontract #10184497), and Department of Energy's Grant #DE-FC02-01ER25506.

1 Introduction

In the past, most research in high end computing focused on development of methods for solving challenging compute intensive applications in science, engineering and medicine. These applications are generally run in batch mode and can generate very large datasets. Advanced sensor technologies also enable acquisition of high resolution multi-dimensional datasets. As a result, there is an increasing interest in developing applications, referred to here as data intensive or data analysis applications, that interactively explore, synthesize and analyze such large datasets.

PC clusters are becoming viable alternatives to mainstream supercomputers for a broad range of applications, including data intensive applications. Being built from commodity hardware, they are cost effective. A challenging issue in supporting data intensive applications on these platforms is that large volumes of data should be efficiently moved between processor memories. Data movement and processing operations should also be efficiently coordinated by a runtime support to achieve high performance.

With the advent of modern high speed interconnects such as GigaNet [17], Myrinet [8], Gigabit Ethernet [23], InfiniBand Architecture [4] and the Quadrics Network [37], the bottleneck in the communication has shifted to the messaging software. This bottleneck has been attacked by researchers, leading to the development of low-latency and high-bandwidth user-level protocols [18, 21, 35]. Along with these research efforts, several industries have taken up the initiative to standardize high-performance user-level protocols such as the Virtual Interface Architecture (VIA) [10, 18].

On the application development and runtime support side, component-based frameworks [7, 12, 34, 38] have been able to provide a flexible and efficient environment for data intensive applications on distributed platforms. In these frameworks, an application is developed from a set of interacting software components. Placement of components

onto computational resources represents an important degree of flexibility in optimizing application performance. Data-parallelism can be achieved by executing multiple copies of a component across a cluster of storage and processing nodes [7]. Pipelining is another possible mechanism for performance improvement. In many data intensive applications, the dataset can be partitioned into *user-defined data chunks*. Processing of the chunks can be pipelined. While computation and communication can be overlapped in this manner, the performance gain also depends on the granularity of computation and the size of the data messages (data chunks). While small chunks sizes would likely result in better load balance, higher overlap and pipelining, the overall performance may suffer from network latency.

While different alternatives have been developed to enhance the communication performance using user-level protocols, applications that have been developed on kernel-based protocols such as TCP/UDP using the sockets interface have largely been ignored. Rewriting such applications using user-level protocols is a very costly and impractical approach. In this paper, we develop a user-level sockets interface over VIA (SocketVIA) and examine its impact on the performance and behavior of a component-based framework, namely DataCutter [7].

The remaining part of the paper is organized as follows. In Section 2, we talk about the background of the TCP/IP protocol suite, the sockets interface and give a brief overview of the Virtual Interface Architecture. In Section 3, we discuss the design and implementation of the sockets layer over VIA, termed as SocketVIA. In Section 4, we give an overview of the component-based framework. We discuss some experimental results in Section 5, present some related work in Section 6 and conclude the paper in Section 7.

2 Background

In this section, we give a brief overview of the Transmission Control Protocol (TCP) protocol suite, the Sockets layer and the Virtual Interface Architecture (VIA).

2.1 TCP/IP Protocol Suite

Like most networking protocol suites, the TCP/IP protocol suite is a combination of different protocols at various layers (Figure 1), with each layer responsible for a different facet of the communications (Figure 2). TCP/IP is normally considered to be a 4-layered system.

The link layer, sometimes called the data-link layer or network interface layer, normally includes the device driver in

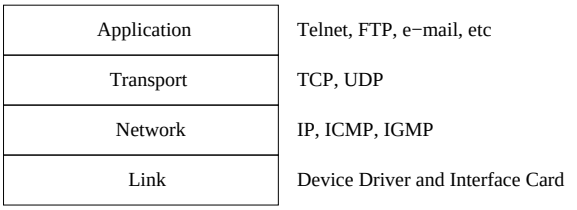

Figure 1. The four layers of the TCP/IP protocol suite

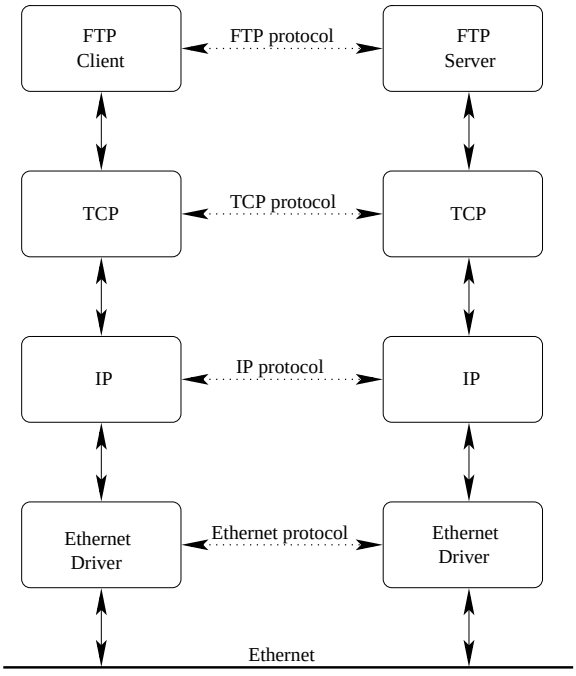

Figure 2. Two hosts on a LAN running FTP

the Operating System and the corresponding Network Interface Card in the computer. Together, they handle all the hardware details of physically interfacing with the cable (or whatever type of media being used).

The network layer (sometimes called the internet layer) handles the movement of packets around the network. Routing of packets, for example, takes place here. IP (Internet Protocol), ICMP (Internet Control Message Protocol) and IGMP (Internet Group Message Protocol) provide the network layer in the TCP/IP Protocol Suite.

The transport layer provides a flow of data between two hosts, for the application layer above. In the TCP/IP protocol suite there are two vastly different transport protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP provides a reliable flow of data between two hosts. It is concerned with things such as dividing the data passed to it from the application into appropriate sized chunks for the network layer below, acknowledging received packets, setting timeouts to make certain the other end acknowledges packets that are sent, and so on. Because this reliable flow of data is provided by the transport layer, the application layer can ignore all these details. UDP, on the other hand, provides a much simpler service to the application layer. It just sends packets of data called datagrams from one host to the other, but there is no guarantee that the datagrams reach the other end. Any desired reliability must be added by the application layer.

The application layer handles the details of the particular application. The common examples of TCP/IP applications are Telnet, FTP, SMTP and SNMP.

2.1.1 TCP/IP Layering

Figure 3 shows the various protocols at each layer of the TCP/IP protocol suite.

IP is the main protocol at the network layer. It is used by both TCP and UDP. Every piece of TCP and UDP data that gets transferred around an internet goes through the IP layer at both end systems and at every intermediate router.

ICMP is an adjunct to IP. It is used by the IP layer to exchange error messages and other vital information with the IP layer in another host or router. Although ICMP is primarily used by IP, it possible for applications to directly access it. Ping and Traceroute for example use ICMP messages directly. IGMP is mainly used for group communication operations such as multicasting: send a UDP datagram to multiple hosts.

ARP (Address Resolution Protocol) and RARP (Reverse Address Resolution Protocol) are specialized protocols used only with certain types of network interfaces (such as Ethernet and token ring) to convert between the addresses used by the IP layer (IP addresses) and the addresses used by the network interface (MAC addresses).

2.2 Sockets Interface

The socket abstraction was introduced with the 4.2BSD release in 1983 to provide a uniform interface to network and interprocess communication protocols. The socket layer maps protocol-independent requests from a process to the protocol-specific implementation selected when the socket was created (figure 4). For example, a "STREAM" socket corresponds to TCP, while a "Datagram" or "DGRAM" socket corresponds to UDP, and so on.

To allow standard Unix I/O system calls such as `read()` and `write()` to operate with network connections, the filesystem and networking facilities in BSD releases are integrated at the system call level. Network connections represented by sockets are accessed through a descriptor (a small integer) in the same way an open file is accessed through a descriptor. This allows the standard filesystem calls such as `read()` and `write()`, as well as network-specific system calls such as `sendmsg()` and `recvmsg()`, to work with a descriptor associated with a socket.

A socket represents one end of a communication link and holds or points to all the information associated with the link. This information includes the protocol to use, state information of the protocol (which includes source and destination addresses), queues of arriving connections, data buffers and option flags (figure 5).

2.3 Issues with the existing TCP Implementation

The present day physical networks are capable of giving a very high point-to-point bandwidth. With the advent of networks such as GigaNet [19, 20], Gigabit Ethernet [24] and Myrinet [9], researchers have been able to achieve throughputs in the order of Gigabits per second. Communication and Networking giants, like Cisco and Nortel have come up with faster and higher bandwidth providing optical networking systems. With the onset of technologies such as OC-48 and OC-192, backbone networks have been able to achieve a data transfer rate of up to a terra byte a second. With such high speed networks being developed, the communication bottleneck has shifted from the physical network to the messaging protocol overhead on the sending and the receiving side.

On the fastest networks, the application-to-application throughput is often limited by the capability of the end systems to generate, transmit, receive and process the data at network speeds. A number of end system factors limit

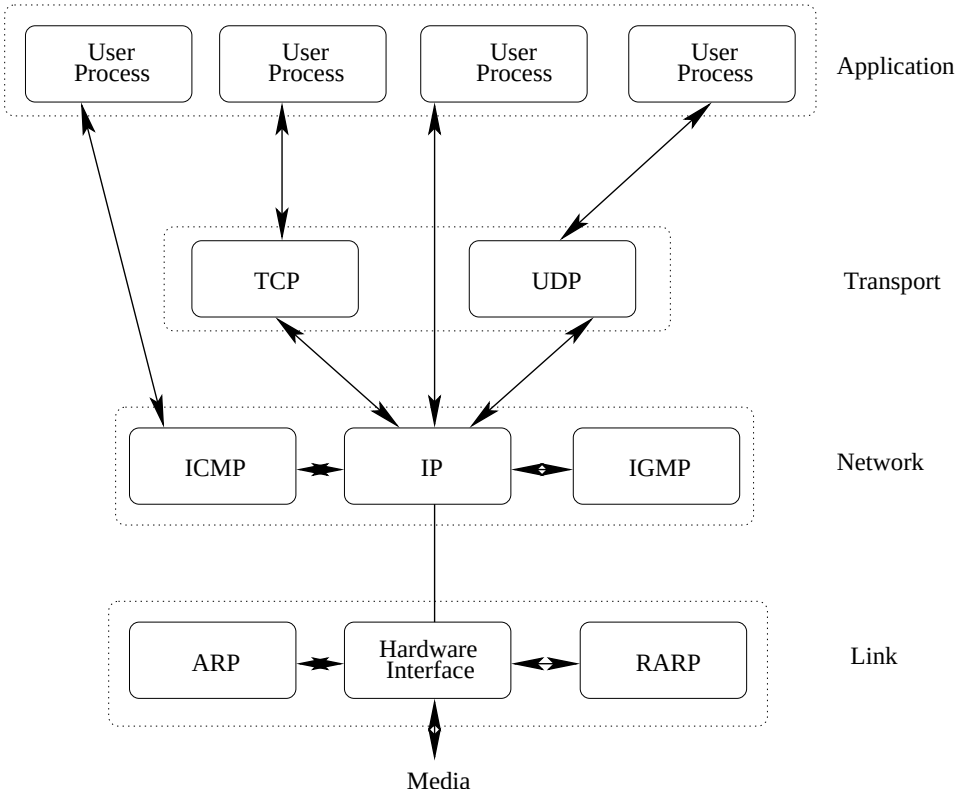

Figure 3. Various protocols at the different layers in the TCP/IP protocol suite

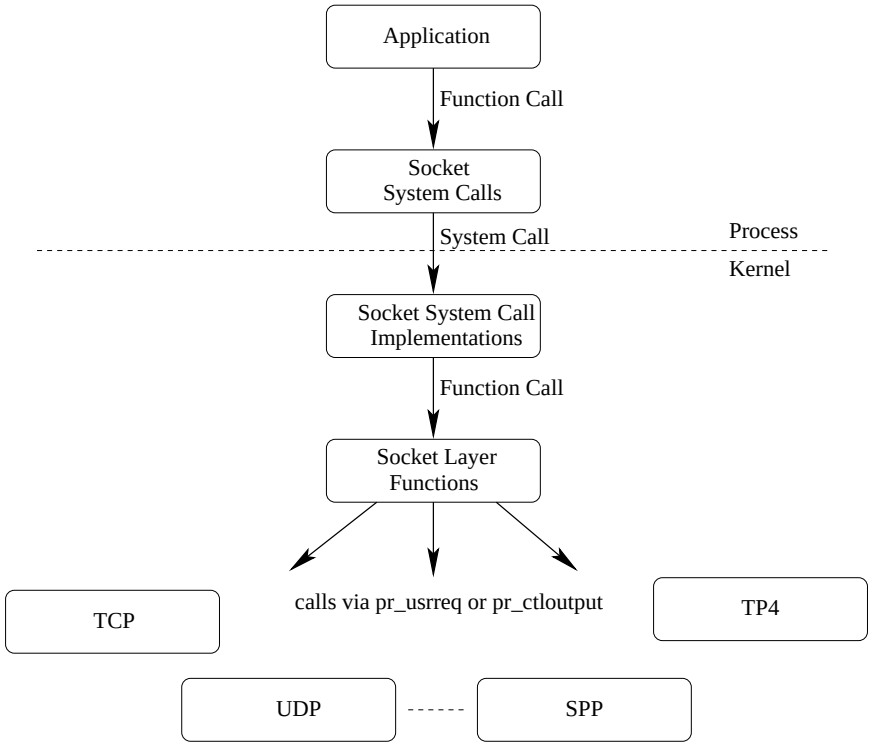

Figure 4. The sockets layer converts generic requests to specific protocol operations

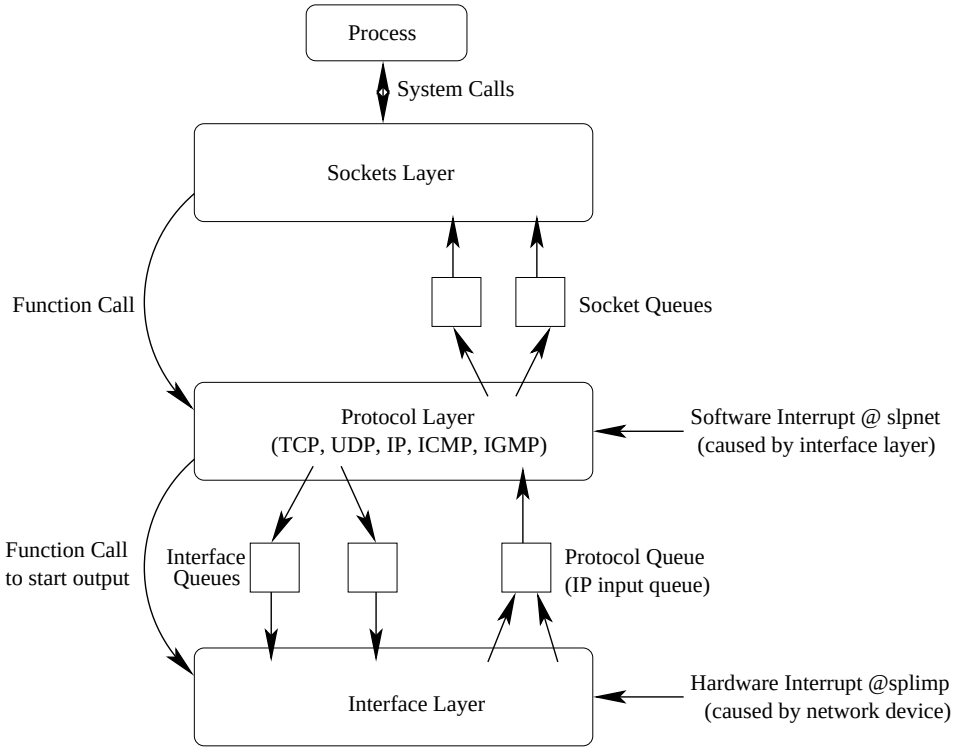

Figure 5. Communication between the layers for network input and output

the communication bandwidth on high-speed networks. With the exponential increase in the CPU power, given by Moore's law, the communication bandwidth limiting factor is not the CPU power but the ability to move data through the host I/O sub-system and memory. While it is possible to build systems with higher I/O and memory throughput, more bandwidth is invariably expensive for a given level of technology.

Traditionally, TCP has not been able to take advantage of the high performance provided by the physical networks due to the multiple copies and kernel context switches present in its critical message-passing path. A number of approaches [30, 33, 13, 40, 29, 14, 3, 27, 32, 25, 41, 26, 28, 15, 36] have been devised to improve the performance given by TCP. However, these approaches have had only limited success.

End systems incur CPU overhead for processing each network packet or frame. These per packet costs include the overhead to execute the TCP/IP protocol code, allocate and release memory buffers, and field device interrupts for packet arrival and transmit completion. TCP/IP implementations incur additional costs for each byte of data sent or received (Figure 6). These include overheads to move data within the end system, to compute checksums and to detect data corruption in the network.

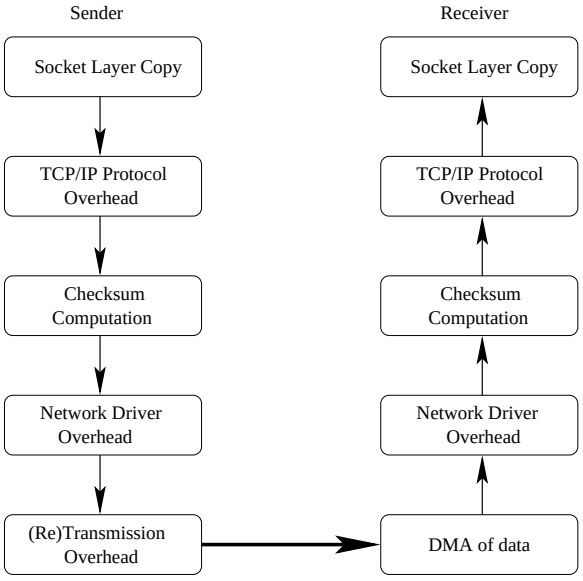

Figure 6. Overheads in the traditional TCP/IP implementation

A number of approaches have been proposed to minimize the per-packet and the per-byte overhead in the TCP/IP protocol implementation.

Jumbo Frames: Many Gigabit Ethernet vendors have followed proposals such as "Jumbo" frames, where the frame size is increased up to 9000 bytes in contrast to the 1500 bytes frames used by traditional Ethernet. This approach reduces the number of interrupts on the sending and the receiving sides for large messages. However this approach is not very beneficial for smaller messages. Also, this raises compatibility issues with the existing Ethernet base.

Interrupt Coalescing: Another approach to reduce the per-packet overhead is by using interrupt coalescing, where a number of interrupts are consolidated into a single interrupt, thus reducing the interrupt handling cost significantly. Though this approach is good for bandwidth sensitive applications where large streams of data keep coming in, it degrades the performance for latency sensitive applications due to the added delay for interrupt coalescing.

Checksum Offload: This approach deals with offloading the checksum calculation and verification to hardware. This approach is used in a number of implementations and is able to significantly reduce the cost for checksum calculation and verification. However, this does not deal with the other overheads in the TCP/IP protocol.

2.4 Overview of Virtual Interface Architecture

In the past couple of years, a large number of user-level protocols have been developed to reduce the gap between the performance capabilities of the physical network and that experienced by the application. VIA specifications [16], as an industry standard, had been developed and standardized mainly by Compaq, Intel and Microsoft with a similar idea. Since VIA has a very low level API which provides only the minimal communication primitives to the programmer, developing applications on VIA is considered a challenge in itself.

The Virtual Interface Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers and VI Consumers. The organization of these components is illustrated in Figure 7.

The VI provider is composed of a physical network adapter and a software kernel agent. The VI consumer is generally composed of an application program and an operating system communication facility. A Virtual Interface consists of a pair of Work Queues: a send queue and a receive queue. VI Consumers post requests, in the form of descriptors, on the work queues to send or receive data. A descriptor is a memory structure that contains all the information that the VI provider needs to process the request, such as pointers to data buffers. VI providers asynchronously process the posted descriptors and mark them with a status value when completed. VI consumers remove completed descriptors from the work queues and use them for subsequent requests. Each work queue has an associated doorbell that is used to notify the VI network adapter that a new descriptor has been posted to a work queue. The doorbell is directly implemented by the adapter and requires no Operating System intervention to operate. A Completion Queue allows a VI Consumer to coalesce notification of descriptor completions from the work queues of multiple VIs in a single location.

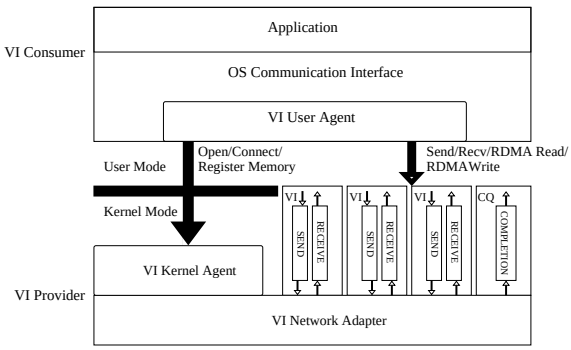

Figure 7. The Virtual Interface (VI) Architectural Model

VIA has been designed to provide high-bandwidth and low-latency support over a System Area Network (SAN). Since it's introduction, implementations of VIA have been made available on a variety of platforms. M-VIA (Modular VIA) [1] emulates the VIA specification by software for legacy Fast Ethernet and Gigabit Ethernet adapters. B-VIA (Berkeley VIA) [10] implementation supports the VIA specification on Myrinet, by modifying its firmware. Finally, GigaNet Incorporation (now called Emulex Corporation) has developed a proprietary VI-aware NIC called cLAN [18]. Our substrate has been implemented and evaluated on the GigaNet cLAN cards.

3 Sockets over VIA

In this section we give a brief overview of the existing socket implementations over VIA. We present the design issues of our Sockets over VIA layer, termed as *SocketVIA*, and give some insight on the performance enhancement techniques used in SocketVIA.

3.1 Previous Implementations of Socket over VIA

Inspite of the development of low-latency and high-bandwidth user-level protocols, a large number of applications have been developed previously on protocols such as TCP and UDP. Some of these applications took years to develop. Trying to rewrite these applications on user-level protocols is highly time-consuming and impractical. On the other hand, the sockets interface is widely used by a variety of applications written on protocols such as TCP and UDP. Implementing a socket layer over VIA can ease application development, enable existing applications to seamlessly run on VIA networks, and deliver high performance. At this point, the following open question arises:

Is there some way by which we can continue to use these applications and at the same time take advantage of the low-latency and high-bandwidth provided by the user-level protocols such as VIA?

The traditional communication architecture (Figure 8) involves just the application and the corresponding libraries in the user space, while the TCP/UDP, IP, etc., layers are present in the kernel space. This approach results in not only multiple copies in the TCP/IP protocol stack, but also context switches to the kernel for every communication step, thus adding a significant overhead.

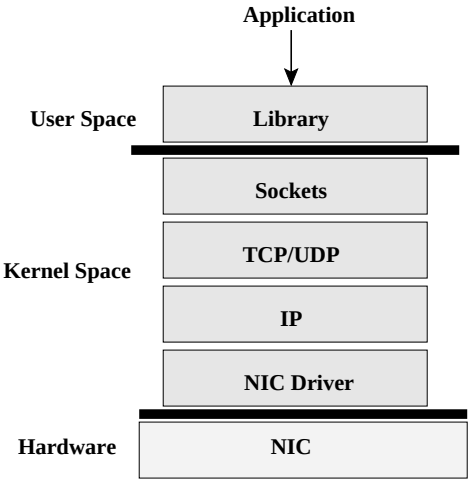

Figure 8. The Traditional Communication Architecture with the Sockets layer in the kernel space together with the TCP/UCP, IP and other layers

The LANE (LAN Emulator) driver supplied by GigaNet for its cLAN adapters [18] uses a simple approach. They provide an IP-to-VI layer which basically maps the IP layer to be compatible with their VI-aware cLAN NICs (Figure 9). However, TCP is still there, so are the multiple copies. Further, the whole setup is in the kernel, so the kernel context switch also is still there. Thus, it can be easily seen that, though this approach gives us the required compatibility, it does not give us any performance improvement.

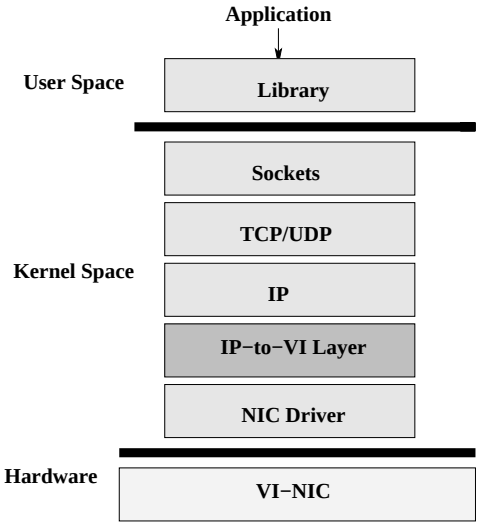

Figure 9. The LAN Emulator (LANE) Communication Architecture developed by GigaNet, with an IP-to-VI layer to map IP packets to be understood by their VI-aware NIC

To take advantage of the high-performance of VIA, two essential changes are required. Firstly, the TCP layer has to be removed, thus avoiding the multiple copies. This requires porting the sockets library directly on to VIA. Secondly, the substrate has to be moved from the kernel space to the user-space, in order to avoid the additional context switch to the kernel for every communication step, in essence removing the kernel from the critical message passing path.

There are implementations of socket layer over various User-Level Protocols [31, 39, 5] including VIA. However, these are not publicly available due to stability issues. Further, these do not consider certain issues such as deadlocks in sockets, supporting data streaming for TCP sockets and performance enhancement techniques such as Lazy Deregistration (explained in the later sections). Also, some implementations of Sockets over VIA use the kernel support for distinguishing between communication sockets and local file reads and writes. This adds a kernel context switch in the critical path, thus hampering the performance. Keeping in mind these requirements, we have implemented our own Sockets library over VIA, termed as socketVIA (Figure 10).

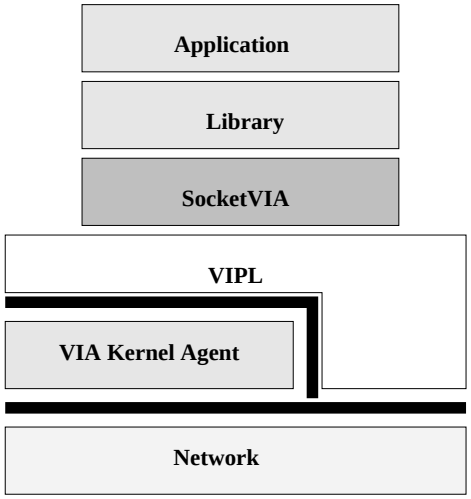

Figure 10. The proposed solution with Sockets implemented on top of the Virtual Interface Architecture (VIA) in user-space, thus getting rid of the multiple copies and kernel context switches of TCP/IP

3.2 Substrate Design

TCP has been developed to obtain a reliable, secure and fault tolerant base protocol to develop networking applications. On the other hand, the motivation for developing VIA is to obtain a high performance protocol to support communication intensive applications. We have identified several significant mismatches in these two protocols and implemented solutions so as to maintain the functionality and semantics similar to that of TCP, while not degrading the performance obtained from using VIA significantly.

3.2.1 Connection Management

For connection management, applications built on TCP use the `listen()` and `accept()` calls. The `listen()` call semantics dictate that, the Operating System be informed when a connection request arrives, the request be kept in a queue, and the client be informed of the status. The function returns as soon as the Operating System is informed to do so. The `accept()` call checks if a request has arrived and if it has not, waits (blocks) for one.

On the other hand, VIPL (VI Provider Library) supports two primitive calls – `VipConnectWait()` and `VipConnectAccept()`. `VipConnectWait()` call blocks till the connection request arrives and the control is returned to the user only after the request arrives. Since the number of connections cannot be known before hand, asynchronous connection requests cannot be dealt with this blocking function.

As we can see, there is no clear correspondence between the API supported by TCP and VIA. A number of approaches are possible to bridge this mismatch between the two APIs.

Peer-to-Peer Connections: One approach would be to use peer-to-peer connection primitives specific to the GigaNet cLAN implementation of VIA such as `VipPeerConnectRequest()`. The advantage with these calls is that, on the arrival of connection requests, they are queued by the GigaNet cLAN NIC, similar to that done by the Operating System in the `listen()` call. However, though these calls are non-blocking, they have been developed for the peer-to-peer communication model, where the accepting node knows exactly from whom the connection request is going to come. This does not fit in exactly with the client-server model of TCP, since in the client-server model, the server might not know the client from which the connection request is "expected". This difference is apparent in the wild-card `accept()` call, where the server can accept a connection from any client.

Connection Thread: To deal with this mismatch, we have used a multi-threaded approach (Figure 11). In the solution proposed, as soon as a `listen()` call is encountered, a separate thread is spawned. This thread will be referred to as the *Connection Thread* in all future references. When this thread is created, it calls the `VipConnectWait()` function and waits for the connection request. On the other hand, the main thread can carry on with its computation.

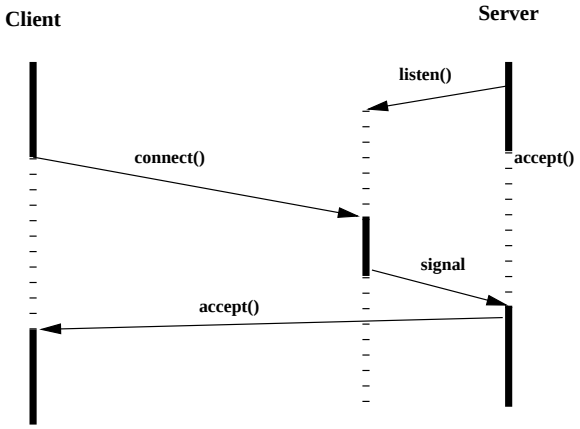

Figure 11. SocketVIA Connection Management: A Multi-Threaded Approach

The issues to be noted in this approach are the CPU cycles used by the connection thread and the connection time. One approach is to make the connection thread poll for con-

nections, which would result in a high CPU usage (about 50%) for the connection thread together with the synchronization cost between the two threads (upto 20μsecs). Alternatively, making the connection thread sleep would decrease the CPU cycles used, but would increase the connection time to the order of the Operating System scheduling granularity (order of milliseconds). We have looked at other options like polling for sometime and sleeping, but they have not been found to give any significant benefit, so we have restrained ourself to the second option on the assumption that the connection time delay is not a bottleneck for data intensive applications.

3.2.2 Asynchronous Message Arrivals

VIA has a descriptor pre-posting constraint. When a message arrives at a node, this node has to make sure that an appropriate descriptor has been posted to inform the NIC about where the incoming message is supposed to be placed. If this descriptor is not posted, the message is dropped. Further, if it is a reliable connection, this might even break the connection.

In TCP, since the incoming message is buffered at the kernel, there need not be a perfect synchronization between the sender and the receiver for sending data messages. This mandates that the socket layer has to buffer messages arriving asynchronously. To handle this, we have considered several mechanisms to post new descriptors:

- Interrupts
- Separate Communication Thread
- Rendezvous Approach
- Eager with Flow Control

Interrupts: In the first option, a number of descriptors are posted on the receiver side to handle asynchronous message arrivals. As soon as a message arrives, am interrupt is generated by the NIC, which calls a function to handle this message and post another descriptor. This method can be used on programmable NICs such as Myrinet to generate a hardware interrupt as soon as the message arrives. However, this approach cannot be used here, as GigaNet cLAN cards are not programmable and do not support the notification facility.

Separate Communication Thread: The second option was using a separate connection thread to keep polling for completed descriptors. This was evaluated and found to be very costly. With both the communication thread and the main application thread polling, the synchronization cost of the threads itself comes to about 20μsecs. In case of blocking threads, the Operating System scheduling granularity makes the response time too coarse for any performance benefit.

Rendezvous Approach: The third approach we looked at was the traditional rendezvous approach [22] where the sender and the receiver are explicitly synchronized (Figure 12). Once the sender sends the request, it blocks till it receives an acknowledgment from the receiver. The receiver on the other hand checks for the request when it calls the `read()` call, posts two descriptors – one for the incoming data message, one for the next request, and sends back the acknowledgment to the sender. The sender on receiving the acknowledgment, sends the data message. Effectively, the sender is blocked until the receiver has synchronized and once this is done, it is allowed to send the actual data message. This gets an additional synchronization cost in the latency.

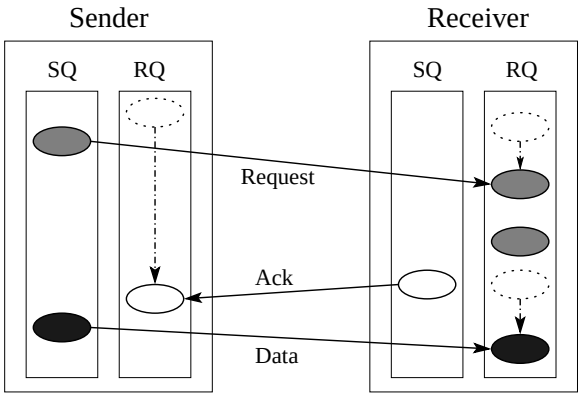

Figure 12. The Rendezvous Approach: Explicit synchronization between the sender and the receiver before any data transfer takes place

Though this approach looks straight forward, it has a number of issues associated with it. TCP supports data streaming. If the sender sends 10 bytes of data, TCP allows the receiver to read it as two sets of 5 bytes each, potentially into two different buffers. But, this option will disable this approach.

Another issue with this approach is the possibility of deadlocks. When the sender wants to send a data message, it first sends a request and waits for an acknowledgment, which is sent only when the receiver calls a receive. Consider the case when both the nodes want to send data to each other. Node #1 calls write1 and read2 in that order, and Node #2 calls write2 and read1. Note that TCP allows this as the incoming data message is stored in temporary buffers. Now,

using rendezvous in a case like this, Node #1 blocks on write1 waiting for an acknowledgment from Node #2, so read2 is not called. Similarly, Node #2 blocks on write2 waiting for an acknowledgment from Node #1, so read1 is not called. Thus, a deadlock (Figure 13).

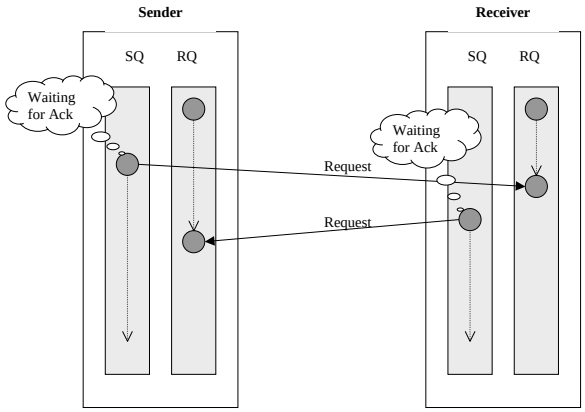

Figure 13. Deadlock in the Rendezvous Approach: Multiple Outstanding writes

Eager with Flow Control: In this project, we have implemented a mechanism similar to the eager approach of MPI [22], but with a flow control mechanism (Figure 14). In this mechanism, the receiver initially posts a descriptor. When the sender wants to send a data message, it goes ahead and sends the message, but for the second data message, it waits for an acknowledgment from the receiver, saying that another descriptor has been posted. Once this acknowledgment has been received, the sender can send the next message. The receiver side has to make sure that there is a descriptor posted for one asynchronous message. When a data message comes in, it uses up the pre-posted descriptor and is stored in a temporary buffer. Once the receiver calls the receive function, the data is copied into the user buffer, another descriptor is posted and an acknowledgment sent back to the sender. This involves an extra copy on the receiver side.

Note that even this solution is not completely free from the possibility of deadlocks. We have extended this idea by using more than one pre-posted descriptors. However, increasing the number of descriptors only reduces the possibility of a deadlock, but does not avoid it completely. Note that, so is the case with the normal sockets implementation of over TCP/IP, where the sockets layer handles asynchronous writes only till it has space in the socket buffer. Once it is filled up, it results in a deadlock.

We have extended the idea of "Eager with Flow Control", by pre-posting N descriptors instead of one, and allowing the sender to send up to N asynchronous messages before

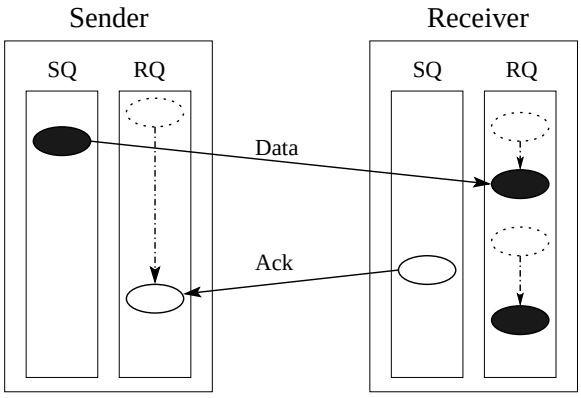

Figure 14. Eager with Flow Control approach. The sender sends the first data message asynchronously, but waits for the acknowledgment to send the next data message

waiting for an acknowledgment (Figure 15). This approach pipelines the data copy on the receiver side, thus reducing its effect on bandwidth. One main problem with applying this algorithm directly is that the acknowledgment message also uses up a descriptor and there is no way that the receiver would know when the descriptor is reposted unless the sender sends back another acknowledgment thus forming a cycle. To avoid this problem, we have investigated the following potential solutions:

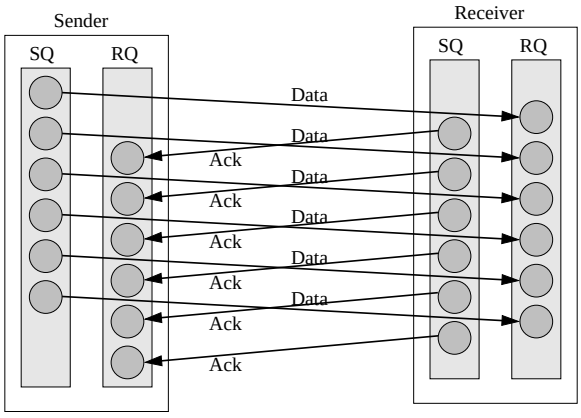

Figure 15. Eager with Flow Control approach extended to handle multiple outstanding write requests

Block the send call: In this approach, the send call is blocked till an acknowledgment is received from the receiver. In essence, the time taken for a `send()` call would increase to that of a round trip latency. Also, it requires synchronization from the receiver. Further, this approach

would throw away all the benefit obtained in avoiding deadlocks.

Post $2N$ descriptors: In this approach, $2N$ descriptors are posted where N is the number of *credits* given, which is analogous to window size in TCP. It can be proved that at any point of time, the number of unattended data and acknowledgment messages will not exceed $2N$. The only problem with this solution is the number of buffers registered. Since the arrival of a data message or the acknowledgment message cannot be predicted, a buffer has to be allocated corresponding to each of the descriptors.

Two VIs per connection: In this approach, the problem of additional buffers is avoided by using two different Virtual Interfaces, one for the data messages and the other for the acknowledgments. However, this would result in a decrease in performance [6].

Piggy Backing of Acknowledgments: In this approach, the acknowledgment is piggy backed on the returning data message. Though this approach is used in the substrate, we cannot rely only on this since this would not be possible for applications having one-way communication.

In our implementation, we use a hybrid of piggy-backing and $2N$ descriptors per connection.

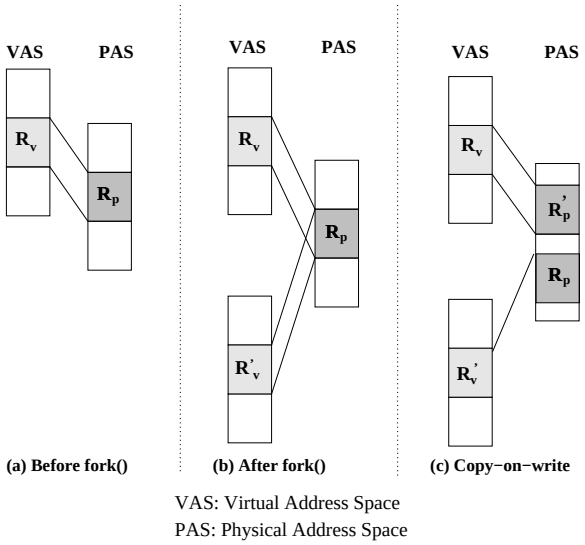

Figure 16. The Copy-on-write problem associated with the *fork()* system call. The parent and the child process share the same physical address space till one of the attempts to write to it, when a copy of the physical page is made

3.2.3 Problems with *fork()*

The *fork()* system call has an inherent property called copy-on-write. When a process forks to form another process, both of them share the same physical address till one of them attempts to write to it. As soon as one of them attempts to write to it, a copy of the physical address page is created. Now, the new physical address space is not registered, so this would return an error (Figure 16).

This problem requires us to make sure that a buffer is registered every time we try to send some data from it. However, the problem is not just restricted to this case. It is possible that the sender can modify the data during the actual sending.

It is too restrictive if the user applications are not allowed to create child processes while they are using VIA. We have come up with a solution which avoids this problem in two approaches – either by copying the data into a temporary registered buffer, or by blocking the `send()` call during the data transfer.

Copying Data: In this approach, the data is copied into a temporary buffer before transmitting it. Since this buffer belongs to the sockets interface and is not written to by the process, the copy-on-write problem does not arise in this approach.

Blocking the `send()` call: In this approach, the `send()` call is blocked till the entire data is transmitted by the sockets layer. In this approach, since the thread calling the `send()` call is blocked, it cannot write to the buffer through which the data is being sent. On the other hand, if the forked thread tries to write to the buffer from which the data is being sent, a new copy of the physical page is created and the unregistered copy is given to the forked thread. Therefore this does not effect the data transfer in progress. However, we have to make sure that each time we try to send data, the buffer we are sending from is registered.

3.2.4 UNIX sockets compatibility

In SocketVIA, socket functions that have just one interpretation, such as `listen()` and `accept()` are mapped onto the corresponding VIA functions. These can be done in a number of ways.

- Function Overriding
- Minor Changes in the Application
- Overloading Function calls

Function Overriding: In this approach, the TCP function calls are directly mapped to the corresponding VIA function calls by overriding them. This approach works

for calls such as `listen()`, `accept()`, etc., which have only one interpretation. But, for calls such as `read()` and `write()`, which can be used for communication as well as file access, this approach does not work.

Minor Changes to the Application: In this approach, a parameter is added to the functions which allows the substrate to distinguish between a call to the VIPL (Virtual Interface Provider Library) and one to the libc library. This approach gives the flexibility of using both sockets over VIA as well as sockets over TCP. However, since we didn't want to make any changes in the application, we did not go ahead with this approach.

Function Space Overloading: In this approach, no changes are made to the application. Functions such as `read()` and `write()` can be used for communication as well as for file access. To distinguish these uses, SocketVIA keeps track of file descriptors created using communication specific calls. If a request uses one of the file descriptors that have been created using a communication specific call, the control is passed to the socket layer. For the rest, the control is passed to the standard I/O library.

3.3 Performance Enhancement Techniques

Due to the mismatches present between the API of TCP/UDP and VIA, as we have mentioned, several design issues had to be faced and changes made to the implementation, in order to bridge these mismatches. Due to these, the performance given by the sockets layer loses some of its performance. In order to improve the performance given by socketVIA, we have come up with several techniques.

3.3.1 Registration/Copy Hybrid

VIA requires that data be sent from pinned buffers so that the pages are not swapped out during the course of the data transfer. Similar is the case on the receiver side. We have examined two potential solutions to address this problem.

Temporary Registered Buffer: In this solution, a temporary buffer is registered initially. When `send()` is called, the data is copied into this temporary buffer and the data transfer is carried out from this buffer. Once the acknowledgment is received, the buffer is free for reuse and can be used for the next transfer. In this case, the user buffer is not pinned at all, so the copy-on-write problem does not arise.

User Buffer Copy: In this solution, the user buffer itself is pinned and the data transfer done directly from the user buffer. Once the acknowledgment for this transfer is received, the buffer is free to be deregistered. In this case, the `send()` call is blocked till the data transfer completes, so that the buffer is not modified before the transfer completes.

Both these solutions have their own advantages and disadvantages. Memory registration is very costly for small messages, whereas memory copy becomes costlier for larger messages. On the receiver side, a copy is inevitable in order to support data streaming, but on the sender side a hybrid of these approaches is possible. We use a memory copy for small messages (less than 2Kbytes) and registration of the user buffer for large messages (greater than 2Kbytes).

3.3.2 Lazy Deregistration

Since the user buffer is being registered (for large message sizes) before sending the data, there is a possibility that the next time the user sends a message, it might be in the same buffer (or a subset of the buffer). This is quite common in applications. Based on this observation, certain enhancements for the implementation are possible. For large message sizes, when the `send()` is called, we register the user buffer and send the data, but do not deregister the buffer immediately. When the next `send()` is called, we check if this buffer is the same (or a subset) as the previously registered buffer. If it is, we continue with sending the data without registering the buffer. This enhancement does not affect the worst case performance, but improves the best case performance.

3.3.3 Delayed Acknowledgments

The substrate also uses techniques such as delaying the acknowledgments in order to improve the bandwidth achieved. Instead of sending an acknowledgment for every data message received, it sends an acknowledgment message when half the window size is used up (Figure 17). This ensures that there is lesser network traffic and lesser work done by the sender NIC, thus improving the throughput.

4 Runtime Support for Data Intensive Applications

As processing power and capacity of disks continue to increase, the potential for applications to create and store multi-gigabyte and multi-terabyte datasets is becoming more feasible. Interactive exploration of such large datasets is a critical step in many application domains, including medical applications and engineering applications.

An example from medical domain is digitized microscopy. The software support required to store, retrieve, and process digitized slides to provide interactive response times for the standard behavior of a physical microscope is a challenging issue [2, 11]. At a basic level, the software system should

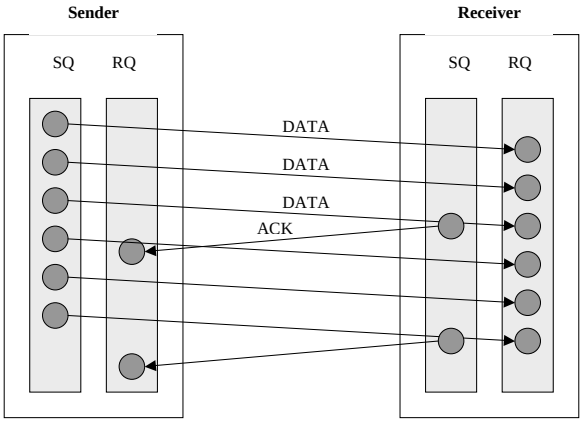

Figure 17. Delaying Acknowledgments to be sent only after half the credit size has been used up.

emulate the use of a physical microscope, including continuously moving the stage and changing magnification. The processing of client queries requires projecting high resolution data onto a grid of suitable resolution and appropriately compositing pixels mapping onto a single grid point. The main difficulty in providing the require functionality is handling of large volumes of image data. With a digitizing microscope a single 200X spot of a slide at a single depth of focus can be acquired at a resolution of 1000 by 1000 pixels. With a three-byte RGB color value per pixel, an image at that resolution produces a data size of 3 MB. Digitizing the complete slide can result in an uncompressed file size of 10.5 GB, corresponding to a single focal plane. Slides are usually acquired at multiple focal planes, further increasing the size of the dataset. Storage and processing needs are exacerbated by the fact that hospitals can generate many thousands of slides per year.

Another example is the iso-surface based rendering of datasets from reservoir models that involve simulation of the transport and reaction of various chemicals over many time steps on a three-dimensional grid that represents the reservoir. In large scale oil reservoir studies, a scientist carries out an ensemble of simulations of a given reservoir model using different geostatistic model parameters, well placements, and material porosity values. For example, a black-oil (three phase) flow problem on a grid with 9,000 cells involves simulation of seventeen separate variables, including oil saturation, water pressure, gas pressure, and water velocity. The values of these variables are output for each node in the grid. For a simulaton of a total of 10,000 time steps, te total output stored is about 6.9 GB. With several hundred realizations, the total amount of data easily exceeds several terabytes. In such simulation studies, iso-surface rendering is a well-suited method to visualize the density distributions of various elements (e.g., oil, water, gas) in a region. A client query retrieves 3-dimensional grids over a set of time steps. For each time step, a set of surfaces are extracted from scalar values of the variables selected by the query and rendered into a 2-dimensional image.

Processing of data in applications that query and manipulate scientific datasets can often be represented as an acyclic, coarse grain data flow, from one or more data sources (e.g., one or more datasets distributed across storage systems) to the processing nodes to the client. For a given query, first the data of interest should be retrieved from the corresponding datasets. The data is then processed via a sequence of operations on the processing nodes.

4.1 Performance Considerations

For data-intensive applications, performance can be improved in several ways. First, datasets can be declustered across the system to achieve parallelism in I/O when retrieving the data of interest for a query. With good declustering, a query will hit as many disks as possible. Second, the computational power of the system can be efficiently used if the application can be designed to exploit data parallelism for processing the data. Another factor that can improve the performance, especially in interactive exploration of datasets, is *pipelined* execution. By dividing the data into chunks and pipelining the processing of these chunks, the overall execution of the application can be decreased. In many applications, pipelining also provides a mechanism to gradually create the output data product. In other words, the user does not have to wait for the processing of the entire query to be completed; partial results of the query can be gradually generated. Although this may not actually reduce the overall response time, such a feature is very effective in an interactive setting, especially if the region of interest moves continuously.

Component-based frameworks can provide an effective environment to address performance issues in data intensive applications. Components can be placed onto different computational resources, and task- and data-parallelism can be achieved by pipelined execution of multiple copies of these components.

The granularity of the work and the size of data chunks affect the performance of pipelined execution. The chunk size should be carefully selected by taking into account the network bandwidth and latency. As the chunk size is increased, the number of messages required to transfer the data of interest decreases. In that case, bandwidth becomes more important than latency. However, with a bigger chunk

size, processing time per chunk also increases. As a result, the system becomes less responsive, i.e., the frequency of gradual updates decreases. On the other hand, if the chunk size is small, the number of messages increases. As a result, latency may become a dominant factor in the overall efficiency of the application. Similarly, smaller chunks can result in better load balance among the copies of application components, but communication overheads may offset the performance gain.

4.2 DataCutter

For the evaluation of the SocketVIA implementation, we used a component-based infrastructure, called DataCutter [7], which is designed to support data intensive applications in distributed environments. In this section we briefly describe the DataCutter framework. DataCutter implements a filter-stream programming model for developing data-intensive applications. In this model, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. The interface for a *filter*, consists of three functions: (1) an initialization function (*init*), in which any required resources such as memory for data structures are allocated and initialized, (2) a processing function (*process*), in which user-defined operations are applied on data elements, and (3) a finalization function (*finalize*), in which the resources allocated in *init* are released.

Filters are connected via *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only. We define a *data buffer* as an array of data elements transferred from one filter to another. The current implementation of the logical stream delivers data in fixed size buffers, and uses TCP for point-to-point stream communication.

The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. When a filter group is instantiated to process an application query, the runtime system establishes socket connections between filters placed on different hosts before starting the execution of the application query. Filters placed on the same host execute as separate threads. An application query is handled as a *unit of work* (UOW) by the filter group. An example is a visualization of a dataset from a viewing angle. The processing of a UOW can be done in a pipelined fashion; different filters can work on different data elements simultaneously. Processing of a UOW starts when the filtering service calls the filter **init** function, which is where any required resources such as memory can be pre-allocated. Next the **process** function is called to read from any input streams, work on data buffers received, and write to any output streams. A special marker is sent by the runtime system after the last buffer to mark the end for the current UOW (see Figure 18(a)). The **finalize** function is called after all processing is finished for the current UOW, to allow release of allocated resources such as scratch space. The interface functions *may* be called again to process another UOW.

The programming model provides several abstractions to facilitate performance optimizations. A *transparent filter copy* is a copy of a filter in a filter group (see Figure 18(b)). The filter copy is transparent in the sense that it shares the same *logical* input and output streams of the original filter. A transparent copy of a filter can be made if the semantics of the filter group are not affected. That is, the output of a unit of work should be the same, regardless of the number of transparent copies. The transparent copies enable data-parallelism for execution of a single query, while multiple filter groups allow concurrency among multiple queries.

The filter runtime system maintains the illusion of a single logical point-to-point stream for communication between a logical producer filter and a logical consumer filter. It is responsible for scheduling elements (or buffers) in a data stream among the transparent copies of a filter. For example, in Figure 18(b), if copy P_1 issues a buffer write operation to the logical stream that connects filter P to filter F, the buffer can be sent to the copies on $host_3$ or $host_4$. For distribution between transparent copies, the runtime system supports a Round-Robin (RR) mechanism and a Demand Driven (DD) mechanism based on buffer consumption rate. DD aims to send buffers to the filter that will process them fastest. When a consumer filter starts processing of a buffer received from a producer filter, it sends an acknowledgment message to the producer filter to indicate that the buffer is being processed. A producer filter chooses the consumer filter with the minimum number of unacknowledged buffers to send a data buffer to, thus achieving a better balancing of the load.

5 Experimental Results

In this paper, we present two groups of results. First, we look at the peak performance delivered by SocketVIA in the form of latency and bandwidth micro-benchmarks. Second, we examine the performance delivered by the substrate on an application. This evaluation was carried out using a synthetic application implemented using DataCutter in order to evaluate both latency and bandwidth aspects in a controlled way. The experiments were carried out on a PC cluster which consists of 16 Dell Power Edge 6400 nodes connected by GigaNet cLAN and Fast Ethernet. We use cLAN 1000 Host Adapters and cLAN5300 Cluster switches. Each

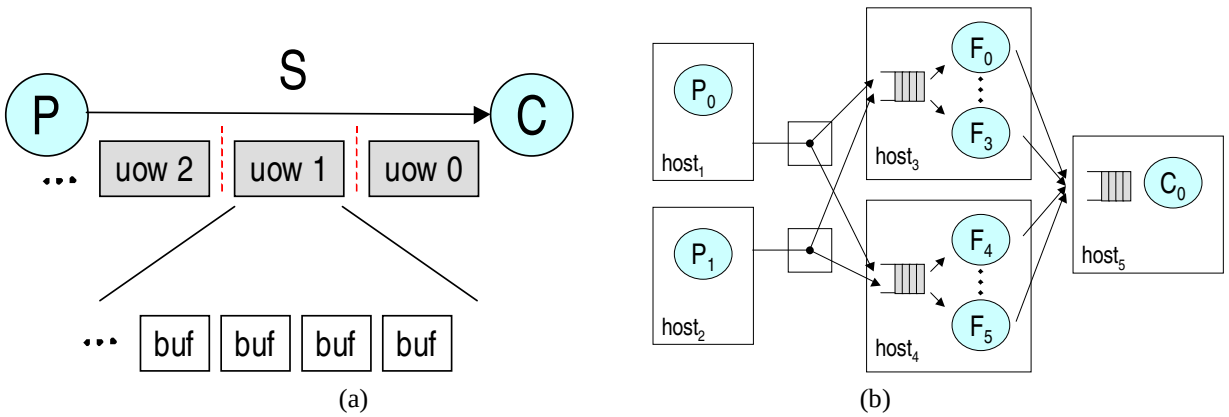

Figure 18. DataCutter stream abstraction and support for copies. (a) Data buffers and end-of-work markers on a stream. (b) P,F,C filter group instantiated using transparent copies.

node has two 1GHz Pentium III Xeon processors, built around the ServerWorks ServerSet III HE chipset, which has a 32-bit 33-MHz PCI bus. These nodes are equipped with 512MB of SDRAM and 256K L2-level cache. The Linux kernel version is 2.2.17. For all the experiments, a credit size of 32 was used in SocketVIA, with each temporary buffer of size 8Kbytes.

5.1 Micro-Benchmarks

Figure 19 shows the latency achieved by our substrate compared to that achieved by the traditional implementation of sockets on top of TCP and a direct VIA implementation (base VIA). Our sockets layer gives a latency of as low as 9.1μs, which is very close to that given by VIA. Also, it is nearly a factor of five improvement over the latency given by the traditional sockets layer over TCP/IP.

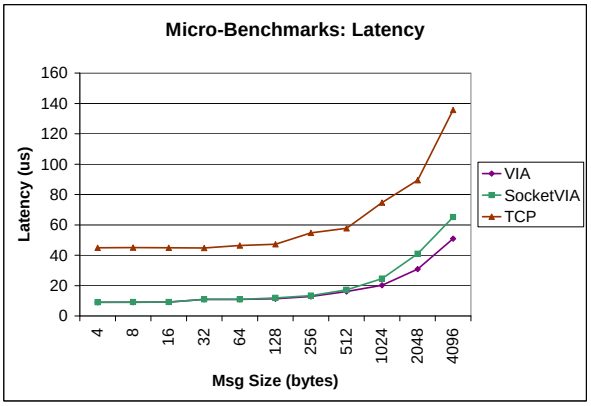

Figure 19. Micro-Benchmarks: Latency

Figure 20 shows the bandwidth achieved by our substrate compared to that of the traditional sockets implementation and base VIA implementation. SocketVIA achieves a peak bandwidth of 763Mbps compared to 795Mbps given by VIA and 510Mbps given by the traditional TCP implementation.

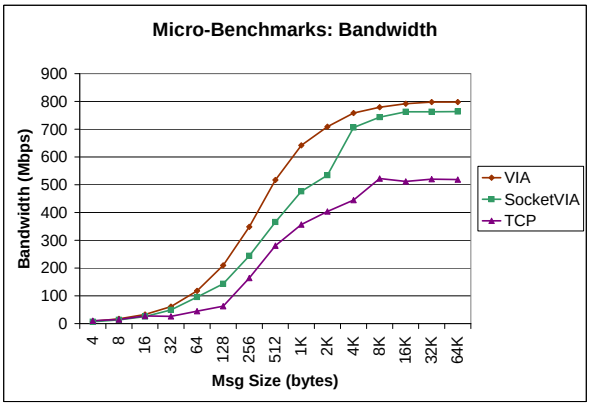

Figure 20. Micro-Benchmarks: Bandwidth

It is to be noted that the latency achieved by the substrate for small messages is very close to that achieved by base VIA. However, as the message size increases, although our implementation remains significantly better compared to TCP, its performance relative to base VIA degrades. The main reason for this is the extra copy required on the receiver side. However, for a certain class of applications this copy operation can be avoided. This version of socketVIA allows the user application to use up the temporary buffer in which it receives the data, assuming that the application does not modify the buffer. Also, this requires certain minor changes in the application in the API. We call this enhanced

version of SocketVIA as Zero-Copy SocketVIA.

Figure 21 and Figure 22 show the latency and bandwidth achieved by Zero-Copy SocketVIA respectively. Our results show that there is not much change in the bandwidth compared to normal SocketVIA. This shows that the pipelining achieved by SocketVIA is able to overlap the time to copy the data with the actual communication time effectively. For the remaining experiments, the normal version of SocketVIA is used.

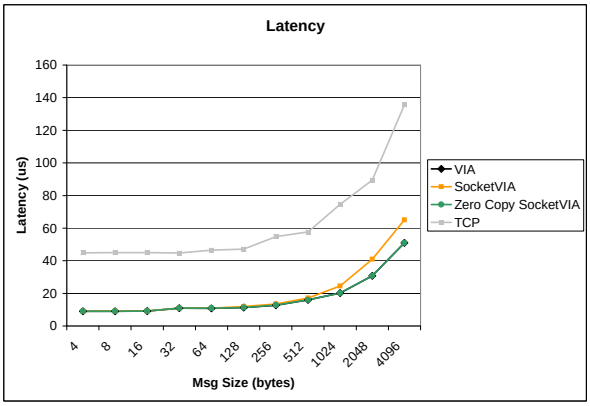

Figure 21. Micro-Benchmarks: Latency achieved by Zero-Copy socketVIA

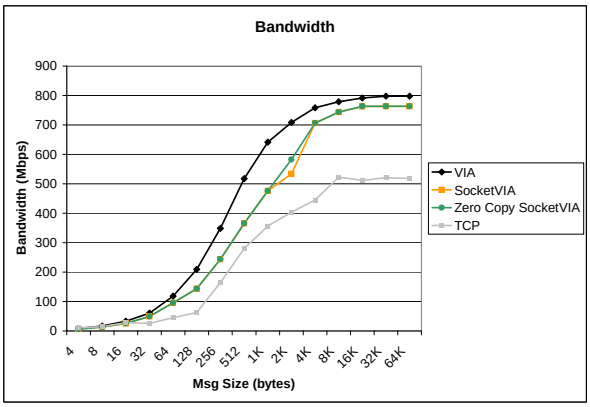

Figure 22. Micro-Benchmarks: Bandwidth achieved by Zero-Copy socketVIA

5.2 Application Performance and Behavior

In this section we look at the impacts caused by High Performance Sockets on the performance and optimality issues of applications.

5.2.1 Guarantee based Performance Evaluation

In these experiments, we used a synthetic application emulating a visualization server implemented using DataCutter. This application uses a 4-stage pipeline with a visualization filter at the last stage. Also, we executed three copies of each filter in the pipeline to improve the end bandwidth (Figure 23). The user visualizes an image at the visualization node, on which the visualization filter is placed. The required data is fetched from a data repository and passed onto other filters, each of which is placed on a different node in the system, in the pipeline.

Each image viewed by the user requires 16MB of data to be retrieved and processed. This data is stored in the form of chunks with pre-defined size, referred to here as the distribution block size. For a typical distribution block size, a complete image is made up of several blocks. When the user asks for an update to an image, the corresponding chunks have to be fetched. Each chunk is retrieved as a whole, potentially resulting in some additional unnecessary data to be transferred over the network.

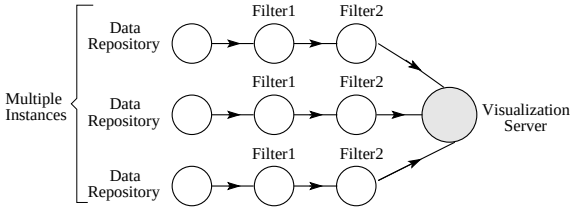

Figure 23. Guarantee Based Performance Evaluation: Experimental Setup

Two kinds of queries were emulated. The first query is a complete update or a request for a new image. This requires all the blocks to be fetched. The second query is a partial update, in which case the user moves the visualization window by a small amount. This query requires only the excess blocks to be fetched. Note that, if the block size is too large, the partial update will likely take long time, since the entire block is fetched even if a small portion of one block is required. However, if the block size is too small, the complete update will likely take long time, since many small blocks will need to be retrieved.

Effect on Average Latency with guarantees on Updates per Second: In the first set of experiments, the user wants to achieve a certain frame rate (i.e., the number of new images generated or full updates done per second). With this constraint, we look at the average latency observed when a partial update query is submitted. Figures 24 and 25 show the performance achieved by the application. For a given frame rate for new images, TCP requires a certain message

size to attain the required bandwidth. With data chunking done to suit this requirement, the latency for a partial update using TCP would be the latency for this message chunk (depicted as legend 'TCP'). With the same chunk size, SocketVIA inherently achieves a higher performance (legend 'SocketVIA'). However, SocketVIA requires a much smaller message size to attain the bandwidth for full updates. Thus, by repartitioning the data taking SocketVIA's latency and bandwidth into consideration, the latency can be further reduced (legend 'SocketVIA (with DR)'). Figure 24 shows the performance with no computation. This experiment emphasizes the actual benefit obtained by using SocketVIA, without being affected by the presence of computation costs at each node. We observe, here, that TCP cannot meet an update constraint greater than 3.25 full updates per second. However, SocketVIA (with DR) can still achieve this frame rate without much degradation in the performance. The results obtained in this experiment show an improvement of more than 3.5 times without any repartitioning and more than 10 times with repartitioning of data. Figure 25 depicts the performance with a computation cost that is linear with message size in the experiments. We had timed the computation required in the visualization part of the Virtual Microscope [11] application on DataCutter and found it to be 18ns per byte of the message. Applications such as these involving browsing of digitized microscopy slides have such low computation costs per pixel. These are the applications that will benefit most from low latency and high bandwidth substrates. So we have focussed on such applications in this paper.

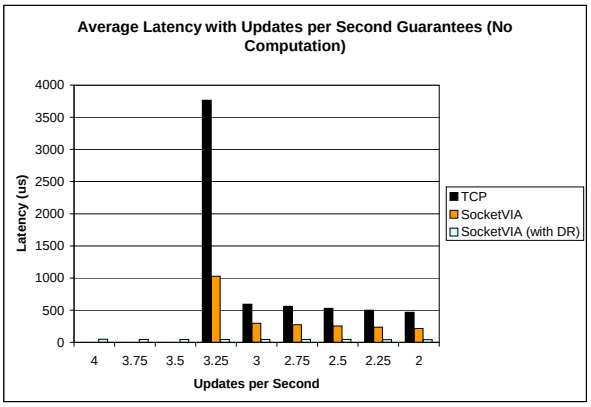

Figure 24. Effect of High Performance Sockets on Average Latency with guarantees on Updates per Second and No Computation Cost

In this experiment, even SocketVIA (with DR) is not able to achieve an update rate greater than 3.25, unlike the previous experiment. The reason for this is that the bandwidth given by SocketVIA is bounded by the computation costs at each node. For this experiment, we observe an improvement of more than 4 and 12 times without and with repartitioning of data, respectively.

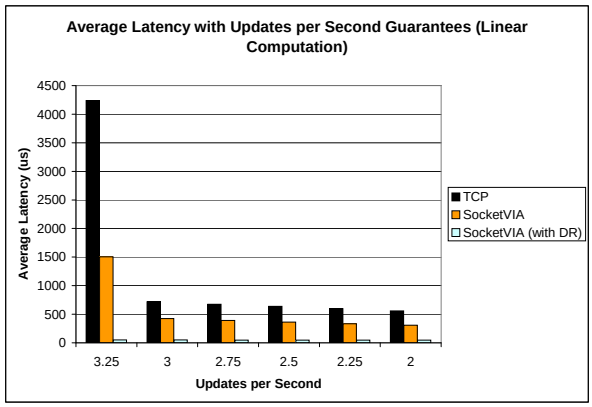

Figure 25. Effect of High Performance Sockets on Average Latency with guarantees on Updates per Second and Linear Computation Cost

Effect on Updates per Second with Latency Guarantees: In the second set of experiments, we try to maximize the number of full updates per second when a particular latency is targeted for a partial update query. Figures 26 and 27 depict the performance achieved by the application. For a given latency constraint, TCP cannot have a block size greater than a certain value. With data chunking done to suit this requirement, the bandwidth it can achieve is quite limited as seen in the figure under legend 'TCP'. With the same block size, SocketVIA achieves a much better performance, shown by legend 'SocketVIA'. However, a re-chunking of data that takes the latency and bandwidth of SocketVIA into consideration results in a much higher performance, as shown by the performancey numbers for 'SocketVIA (with DR)'. Figure 26 gives the performance with no computation, while computation cost, which varies linearly with the size of the chunk, is introduced in the experiments for Figure 27. With no computation cost, as the latency constraint becomes as low as 100μs, TCP drops out. However, SocketVIA continues to give a performance close to the peak value. The results of this experiment show an improvement of more than 6 times without any repartitioning of data, and more than 8 times with repartitioning of data. With a computation cost, we see that for a large latency guarantee, TCP and SocketVIA perform very closely. The reason for this is the computation cost in the message path. With a computation cost of 18ns per byte, process-

ing of data becomes a bottleneck with VIA. However, with TCP, the communication is still the bottleneck. Because of the same reason, unlike TCP, the frame rate achieved by SocketVIA does not change very much as the requested latency is decreased. The results for this experiment show a performance improvement of up to 4 times.

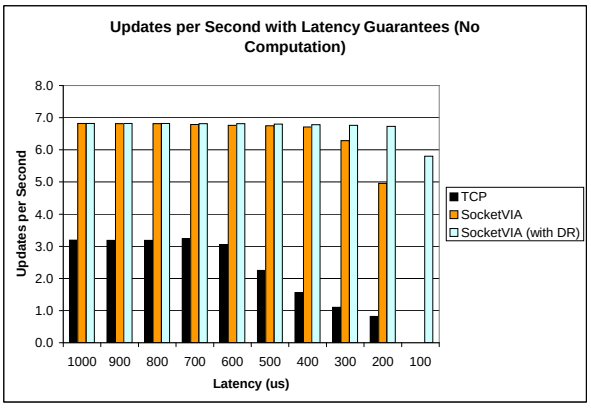

Figure 26. Effect of High Performance Sockets on Updates per Second with Latency Guarantees and No Computation Cost

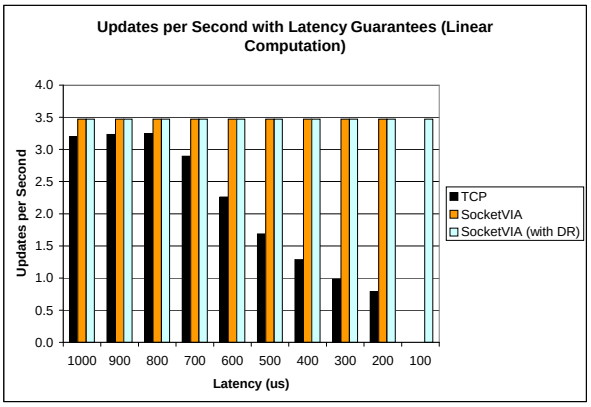

Figure 27. Effect of High Performance Sockets on Updates per Second with Latency Guarantees and Linear Computation Cost

Effect of Multiple queries on Average Response Time: In the third set of experiments, we consider a model where there are two kinds of queries. The first query type is a zoom or a magnification query, while the second one is a complete update query. The first query covers a small region of the image, requiring only 4 data chunks to be retrieved. However, the second query covers the entire image, hence all the data chunks should be retrieved and processed. Figures 28 and 29 display the average response time to queries. The x-axis shows the fraction of queries that correspond to the second type. The volume of data chunks accessed for each query depends on the partitioning of the dataset into data chunks. Since the fraction of queries of each kind may not be known a priori, we analyze the performance given by TCP and SocketVIA with different partition sizes. If the dataset is not partitioned into chunks, a query has to access the entire data, so the timings do not vary with varying fractions of the queries. The benefit we see for SocketVIA compared to TCP is just the inherent benefit of SocketVIA and has nothing to do with the partition sizes. However, with a partitioning of the dataset into smaller chunks, the rate of increase in the response time is very high for TCP compared to SocketVIA. Therefore, for any given average response time, SocketVIA can tolerate a higher variation in the fraction of different query types than TCP. For example, for an average response time of 150ms and 64 partitions per block, TCP can support a variation from 0% to 60%, but fails after that. However, for the same constraint, SocketVIA can support a variation from 0% to 90% before failing. This shows that in cases where the block size cannot be pre-defined, or just an estimate of the block size is available, SocketVIA can do much better.

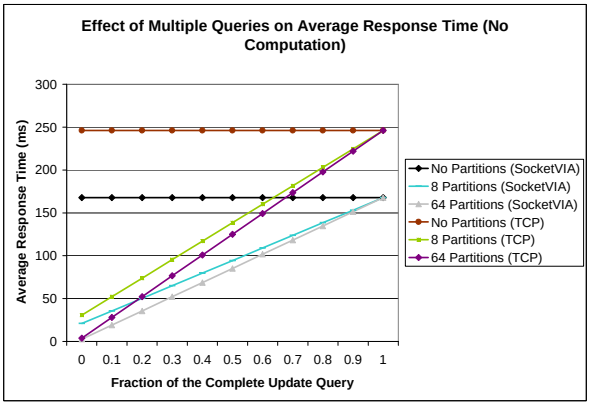

Figure 28. Effect of High Performance Sockets on Updates per Second with Latency Guarantees and No Computation Cost

5.2.2 Effect of SocketVIA on Heterogeneous Clusters

In this experiment, we analyze the effect of SocketVIA on a cluster with a collection of heterogeneous compute nodes. We emulate slower nodes in the network by making some of the nodes do the processing on the data more than once. For protocols like TCP, which is host-based, a decrease in the

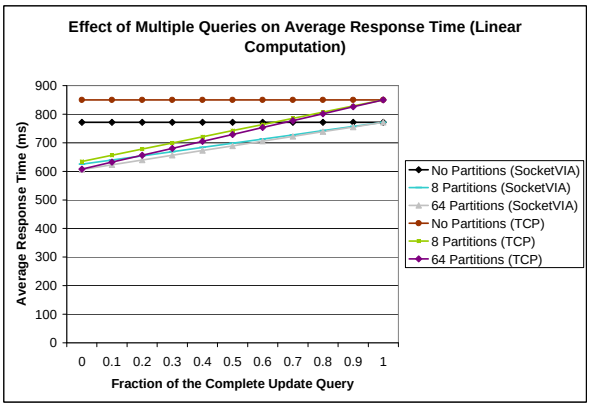

Figure 29. Effect of High Performance Sockets on Updates per Second with Latency Guarantees and Linear Computation Cost

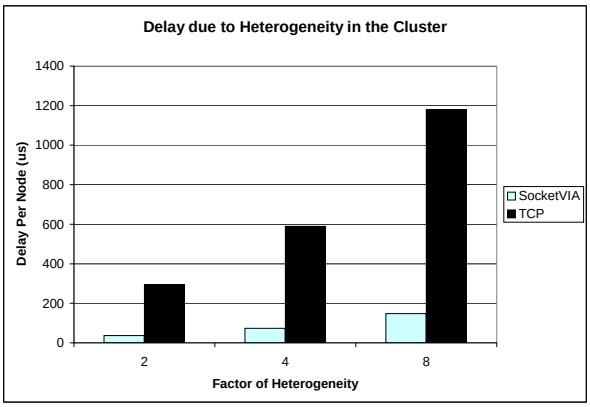

Figure 31. Effect of Heterogeneity in Processing Speed on Load Balancing

processing speed would result in a degradation in the communication time, together with a degradation in the computation time. However, in this experiment, we assume that the communication time remains constant and only the computation time varies.

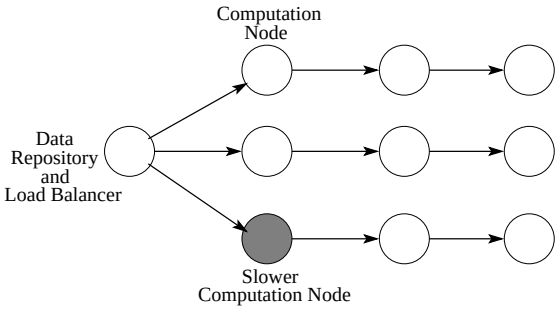

Figure 30. Effect of Heterogeneous Clusters: Experimental Setup

For this experiment, we examine the impact on performance of the round-robin (RR) buffer scheduling in DataCutter when TCP and SocketVIA are employed. In order to achieve perfect pipelining, the time taken to transfer the data to a node should be equal to the processing time of the data on each of the nodes. For this experiment, we have considered load balancing between the filters of the Visualization Application (the first nodes in the pipeline, Figure 30). The processing time of the data in each filter is linear with message size (18ns per byte of message). With TCP, a perfect pipeline was observed to be achieved by 16KB message. But, with SocketVIA, this was achieved by 2KB messages. Thus, load balancing can be done at a much finer granularity.

Figure 31 shows the amount of time wasted per fast node, with increasing factor of heterogeneity in the network. The factor of heterogeneity is the ratio of the processing speeds of the fastest and the slowest processors. The results for this experiment show that with SocketVIA, the amount of time wasted falls by a factor of 8 compared to TCP.

6 Related Work

In this section, we discuss some of the related work done in the area of High Performance Sockets.

Hemal V. Shah et al.[39] and Jin-Soo Kim et al.[31] described the implementation of High Performance Sockets and Remote Procedure Call (RPC) over Virtual Interface Architecture (VIA). However, these works did not explicitly consider certain aspects in the design of the sockets interface such as deadlocks associated with the standard rendezvous approach and certain performance enhancement techniques such as Lazy Deregistration (also known as Pin-Down Cache) in their design (mentioned in the Sockets over VIA section). Also, these implementations use kernel support to distinguish between the communication sockets and local file reads and writes. This adds an additional kernel context switch in the critical message passing path, thus hampering the performance. Finally, these implementations are not publicly available due to stability issues.

In this paper, we give a user-level High Performance Sockets implementation, whixh tries to add-on to their findings by developing additional techniques for a more efficient Stream Sockets implementation over the Virtual Interface Architecture (VIA).

Together with the user-level sockets approach, several industries have started working on hardware support to enhance the performance of TCP sockets based applications. The TCP Offload Engine (TOE) is one such effort by Intel Corporation. TOE is a complete offload of the entire TCP stack including the sockets layer onto hardware. Its a chip compatible with the Gigabit Ethernet cards and is capable of giving a bandwidth as high as 940Mbps and a message processing latency as low as 2-4μsecs for 4 byte messages, all this with as low as a 37% CPU utilization.

7 Conclusions

In this paper we present the design and implementation of a socket layer on top of VIA (SocketVIA) to support applications implemented using sockets on TCP/IP. Our experimental evaluation of the performance of the implementation show that SocketVIA achieves a latency of 9.1μs for 4 byte messages, compared to 45μs by TCP, and a peak bandwidth of 763Mbps compared to 510Mbps given by TCP. We also experimentally evaluate SocketVIA using a component-based framework, namely DataCutter and show that the different performance characterictics of SocketVIA allow applications to do more efficient partitioning of data at the source nodes, thus further improving their performance. Our experiments show that this can result in performance improvements of up to an order of magnitude for some cases.

References

[1] M-VIA: A High Performance Modular VIA for Linux.

[2] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[3] Mark Allman and Aaron Falk. On the Effective Evaluation of TCP.

[4] Infiniband Trade Association. http://www.infinibandta.org.

[5] P. Balaji, P. Shivam, P. Wyckoff, and D.K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Cluster Computing*, September 2002.

[6] M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishnan, P. Sadayappan, H. Shah, and D. K. Panda. VIBe: A Micro-benchmark for evaluating the Virtual Interface Architecture (VIA) implementations. In *Proceedings of IPDPS*, April 2001.

[7] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, October 2001.

[8] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network.

[10] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *Proceedings of Supercomputing*, 1998.

[11] U. Catalyurek, M. D. Beynon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The virtual microscope. *IEEE Transactions on Information Technology in Biomedicine*. To appear.

[12] Common Component Architecture Forum. *http://www.cca-forum.org*.

[13] Jeff Chase, Andrew Gallatin, and Ken Yocum. End-System Optimizations for High-Speed TCP.

[14] Guo Chuanxiong and Zheng Shaoren. Analysis and Evaluation of the TCP/IP Protocol Stack of Linux.

[15] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols.

[16] Compaq, Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture (VIA) Specifications.

[17] GigaNet Corporations. http://www.giganet.com.

[18] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[19] GigaNet Corporations. http://www.giganet.com.

[20] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[21] Myricom Corporations. The GM Message Passing System.

[22] MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing*, 1993.

[23] H. Frazier and H. Johnson. Gigabit Ethernet: From 100 to 1000Mbps.

[24] H. Frazier and H. Johnson. Gigabit Ethernet: From 100 to 1000Mbps.

[25] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds.

[26] Robert Horst. IP Storage and the CPU Consumption Myth.

[27] Jau-Hsiung Huang and Chi-Wen Chen. On Performance Measurements of TCP/IP and its Device Driver.

[28] Jonathan Kay and Joseph Pasquale. Measurement, Analysis and Improvement of UDP/IP Throughput for the DECstation 5000.

[29] Jonathan Kay and Joseph Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP.

[30] Hyok Kim, Hongki Sung, and Hoonbock Lee. Performance Analysis of the TCP/IP Protocol under UNIX Operating Systems for High Performance Computing and Communications. In *the Proceedings of International Conference on High Performance Computing (HPC)*, 1997.

[31] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *Proceedings of Cluster Computing*, 2001.

[32] Evangelos P. Markatos. Speeding up TCP/IP: Faster Processors are not Enough.

[33] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. Technical report, University of Arizona, 1996.

[34] R. Oldfield and D. Kotz. Armada: A parallel file system for computational. In *Proceedings of CCGrid2001*, May 2001.

[35] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of Supercomputing*, 1995.

[36] Vern Paxson. End-to-end internet packet dynamics. *IEEE/ACM Transactions on Networking*, pages 277–292, 1997.

[37] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects*, 2001.

[38] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *HPDC*, August 2000.

[39] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *Proceedings of CANPC workshop*, 1999.

[40] Huseyin Simitci, Chris Malakapalli, and Vamsi Gunturu. Evaluation of SCSI over TCP/IP and SCSI over Fibre Channel Connections.

[41] Evan Speight, Hazim Shafi, and John K. Bennett. WSDLite: A Lightweight Alternative to Windows Sockets Direct Path.