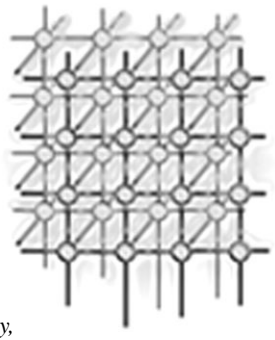
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:727–749 (DOI: 10.1002/cpe.721)

Design and implementation of a user-level Sockets layer over Virtual Interface Architecture



Jin-Soo Kim[1,*,†], Kangho Kim[2], Sung-In Jung[2] and Soonhoi Ha[3]

[1]*Division of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon 305-701, South Korea*
[2]*Computer Systems Research Department, Electronics and Telecommunications Research Institute, Daejeon 305-350, South Korea*
[3]*School of Computer Science and Engineering, Seoul National University, Seoul 151-742, South Korea*

SUMMARY

The Virtual Interface Architecture (VIA) is an industry standard user-level communication architecture for system area networks. The VIA provides a protected, directly-accessible interface to a network hardware, removing the operating system from the critical communication path. In this paper, we design and implement a user-level Sockets layer over VIA, named SOVIA (Sockets Over VIA). Our objective is to use the SOVIA layer to accelerate the existing Sockets-based applications with a reasonable effort and to provide a portable and high-performance communication library based on VIA to application developers.

SOVIA realizes comparable performance to native VIA, showing a minimum one-way latency of 10.5 μs and a peak bandwidth of 814 Mbps on Giganet's cLAN. We have shown the functional compatibility with the existing Sockets API by porting File Transfer Protocol (FTP) and Remote Procedure Call (RPC) applications over the SOVIA layer. Compared to the Giganet's LAN Emulation (LANE) driver which emulates TCP/IP inside the kernel, SOVIA easily doubles the file transfer bandwidth in FTP and reduces the latency of calling an empty remote procedure by 77% in RPC applications. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: Virtual Interface Architecture; Sockets; user-level communication architecture; cluster computing

1. INTRODUCTION

Cluster systems consisting of commodity-off-the-shelf (COTS) hardware components have become increasingly attractive as platforms for high-performance computation and scalable Internet servers [1,2]. Cluster systems are easy to build and have advantages such as scalability and cost-effectiveness, which lead to a high performance/price ratio. To enhance the communication

*Correspondence to: Jin-Soo Kim, Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 305-701, South Korea.
†E-mail: jinsoo@cs.kaist.ac.kr

Received 15 December 2001
Revised 1 August 2002

performance, many cluster systems employ system area networks (SANs) operating at gigabit speed, such as Myrinet, Gigabit Ethernet, SCI, etc. However, as network hardware becomes faster, the software overhead becomes a significant portion of the total time to send a message. More specifically, the traditional communication architecture based on the TCP/IP protocol suite is reported to fail in delivering raw-hardware performance to end users, due to protocol overhead, context switching overhead, and data copying overhead between the user and the kernel space. As a result, a number of user-level communication architectures have been proposed that remove the operating system from the critical communication path [3–6].

The Virtual Interface Architecture (VIA) [7], promoted by Compaq, Intel, and Microsoft, is an industry effort to standardize user-level communication architectures for SANs. The VIA specification defines a software interface for fully-protected, user-level access to a network hardware. The VIA can be emulated by software on the existing network interface cards (NICs). However, in order to achieve a true *zero-copy* protocol, NICs need to be designed to support the VIA mechanisms in hardware, in which case a portion of the processing required to send or receive messages is off-loaded to special hardware on the NIC. Examples of NICs with such VIA-aware hardware include Giganet's cLAN and Fujitsu's Synfinity adapters.

The VIA specification provides a standard application programming interface (API) in the form of a user-level library called the VI Provider Library (VIPL). Although the VIPL can be directly used for application developments, VIA is considered by many systems designers to be at too low a level for application programming [8]. This is because the VIA specification only provides a minimal set of primitives mainly for user-level data transfer of a single message, lacking many high-level features. Thus, we believe it is desirable to implement another lightweight and portable communication layer on top of the VIPL.

In this paper, we design and implement a user-level Sockets layer over VIA, named SOVIA (Sockets Over VIA). The Sockets API is a *de facto* standard for network programming and provides a means for developing applications which are independent of network protocols or hardwares. The SOVIA layer aims at supporting the communication model of Sockets at user-level, without sacrificing the performance of the underlying VIA layer.

The rest of the paper is organized as follows. Section 2 overviews the VIA and the related work. In Section 3, we investigate several design issues to improve the performance and the portability of the SOVIA layer. Section 4 briefly outlines implementation details and Section 5 presents the experimental results of microbenchmarks and two real applications, File Transfer Protocol (FTP) and Remote Procedure Call (RPC), executed over the SOVIA layer. Finally, we conclude in Section 6.

2. BACKGROUND

2.1. Virtual Interface Architecture (VIA)

Figure 1 depicts the organization of the VIA with four basic components: Virtual Interfaces (VIs), Completion Queues, VI Providers, and VI Consumers. The VIA provides each consumer process with a protected, directly-accessible interface to a network hardware called the Virtual Interface. Each VI represents a communication end-point and a pair of connected VIs support a bi-directional, point-to-point data transfer. The VI Provider consists of a physical network adapter and a software Kernel Agent, while the VI consumer represents the user of a VI.

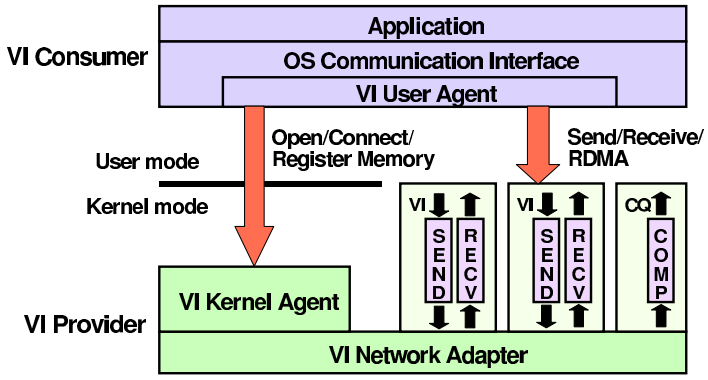


Figure 1. The organization of the VIA.

A VI consists of a pair of Work Queues: a Send Queue (SQ) and a Receive Queue (RQ). A VI Consumer posts requests, in the form of *descriptors*, on the Work Queues to send or receive data. A descriptor is a memory structure that contains all of the information that the VI Provider needs to process the request, such as pointers to data buffers. Each Work Queue has an associated *doorbell* that is used to notify the network adapter that a new descriptor has been posted to the Work Queue. Typically, the doorbell is directly implemented by the adapter and requires no kernel intervention to operate.

When descriptor processing completes, the NIC marks a done bit in the status field of the descriptor and the VI Consumer finally removes it from the Work Queue. The completion of data transfer can be detected either by polling the head of the Work Queue, or by using a blocking call in which a calling process is signaled upon the completion of a descriptor. Alternatively, a user-defined callback function (a *notify function*) may be executed when a descriptor completes. A Completion Queue (CQ) allows a VI Consumer to coalesce notification of descriptor completions from multiple Work Queues in a single location. When a VI is created, each Work Queue can be associated with a CQ. Once this association is established, notification of the completed requests for the Work Queue is automatically directed to the CQ.

All the memory regions used for communication should be registered prior to submitting the request. This is because the VIA allows the NIC to read and write data directly from and to parts of the user address space, thus enabling the zero-copy protocol. When a memory region is no longer needed for communication, it should be explicitly deregistered, whereupon the pages are released and made available for swapping out. The registered memory regions are protected from access by other processes using unique IDs called *Protection Tags* that are associated with both VIs and memory regions.

The VIA supports three levels of communication reliability at the NIC level: Unreliable Delivery, Reliable Delivery, and Reliable Reception. All VI-NICs are required to support the Unreliable Delivery level, where a send request will cause at most one receive descriptor to be consumed. Both Reliable

Delivery and Reliable Reception guarantee that data sent are received uncorrupted, only once, and in the order that they were sent. Reliable Reception is the highest level of reliability, and differs from Reliable Delivery in that a descriptor can not be marked completed until data have been transferred into the memory at the remote end-point.

Several VIA implementations are available for Linux platforms. Modular VIA (M-VIA) [9], developed by NERSC (National Energy Research Scientific Computing) emulates the VIA specification by software for legacy Fast Ethernet and Gigabit Ethernet adapters. Berkeley VIA [10] implementation supports the VIA specification on Myrinet [11] by modifying its firmware. Finally, Giganet Inc. (now Emulex Corp.) has developed a proprietary, VIA-aware NIC called cLAN [12].

2.2. Related work

Two possible candidates that can be used as an upper layer of VIPL are MPI (Message Passing Interface) [13] and Berkeley Sockets API [14], considering their widespread use and acceptance in cluster environments.

There exist several MPI implementations over VIA, such as MVICH [15], ParMa^2 [16], and MPI/Pro [17]. MVICH and ParMa^2 are the modifications of open MPI implementations, MPICH [18] and LAM/MPI [19] respectively, while MPI/Pro is a commercial product developed by MPI Software Technology Inc. Ong and Farrell [20] have compared the performance of MPICH, LAM/MPI, and MVICH on Gigabit Ethernet networks. They showed that the overhead incurred in the TCP/IP protocol stack is still high and the performance of MVICH (over M-VIA) is much superior to that of TCP/IP-based MPICH or LAM/MPI.

Although MPI is important for high-performance parallel computing, we consider Sockets API in this paper because it provides a simpler yet more versatile communication interface compared to MPI. Due to the simplicity of Sockets API, it is easy to optimize the communication performance, imposing less overhead on user applications. There can be several different approaches to the support of Sockets API on VIA, as illustrated in Figure 2. Figure 2(a) shows the traditional communication architecture in which the Sockets layer is located on top of the TCP and UDP protocol stacks.

A simple way to support Sockets API on VIA is to insert an adaptation layer between the IP and the VI Kernel Agent, as shown in Figure 2(b). This is the approach taken by the LAN Emulation (LANE) driver [21] supplied by Giganet for its cLAN adapters. As the IP is emulated on VIA, an IP address is assigned to the VI-NIC and the system becomes fully compatible with any of the existing IP-based network applications. However, it is not straightforward to emulate connectionless IP services (for example, broadcasting) on the connection-oriented VIA and applications will still suffer from the overhead of TCP/IP protocols.

The TCP/IP protocol stack is not required to transfer data between two end-points on the same cluster if the physical interconnect is reliable and provides transport-level functionality. The overhead of TCP/IP protocols can be eliminated on VIA by collapsing internal software layers and emulating Sockets API directly over the VI Kernel Agent, as can be seen in Figure 2(c). A similar approach called VIsocket has been recently proposed by Itoh *et al.* [22], which provides Sockets functionality below the STREAMS module in Solaris. However, the design and performance details of VIsocket have not yet been published.

We note that both approaches, shown in Figures 2(b) and 2(c), still require context switching and data copying between the user and the kernel space to send or receive data, as the support for Sockets

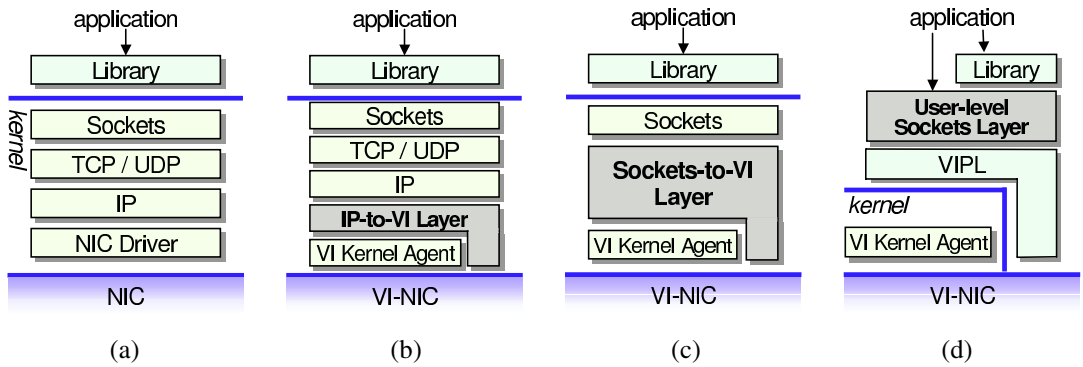


Figure 2. Design alternatives for supporting Sockets API on VIA: (a) traditional architecture; (b) using an IP-to-VI layer; (c) using a Sockets-to-VI layer; (d) using a user-level Sockets layer.

API is implemented inside the kernel. Unlike these approaches, we build the SOVIA layer entirely at the user-level (on top of VIPL) as shown in Figure 2(d), so as to fully utilize the VIA's user-level data transfer capability.

Fast Sockets [23] was the first attempt to support Sockets API over a lightweight user-level protocol, Active Messages (AM). Fast Sockets collapses protocol layers and uses simple buffer management strategies, while avoiding copy operations whenever possible. However, because AM is connectionless and has a unique message passing model in which a packet contains the name of a handler function, Fast Sockets can not be directly used for VIA.

A similar user-level Sockets library, called the SCI Sockets Library (SSLib) [24], has been proposed for the Scalable Coherent Interface (SCI). SSLib supports both stream and datagram sockets, where datagram sockets are emulated by sending UDP packets over a temporary, dedicated connection. For stream sockets, SSLib supports out-of-band (OOB) data used in TCP. One interesting design choice in SSLib is the use of a special daemon process called SCI Sockets Daemon (SSD), which is responsible for managing connections, providing support for special system calls, and controlling sharing of connections, etc. We note that SSLib is based on the remote memory access (RMA) facility of SCI which is very different from the descriptor processing model used in VIA. As the remote buffer is mapped into the address space of the sending process in SSLib, the actual data transfer mechanism is much simpler. Also, it is relatively easy to share a socket connection in SSLib; the remote buffer is mapped to multiple processes and the SSD employs semaphores to synchronize accesses to the shared buffer region, potentially blocking concurrent accesses by other processes.

Our approach is conceptually similar to the Windows Sockets Direct Path (WSDP) [25] technology developed by Microsoft for Windows NT platforms. The WSDP enables Windows Sockets applications that use TCP/IP to obtain the performance benefits of SANs without application modifications, by switching to the SAN Service Provider below the Winsock library. In WSDP, a normal TCP/IP socket is used to establish a TCP/IP connection. However, once the connection is established, a *companion*

socket is created between SAN Service Providers, either from or to which the actual data is being transferred.

Shah and Maddukkarumukumana [26] have also implemented Stream Sockets over VIA on the Windows NT platform with Giganet's cLAN. Stream Sockets has several features to improve the performance. First, it implements a credit-based flow control mechanism, where the sender sends a credit request (*CreditRequest*) and waits for the credit response (*CreditResponse*) if the number of send credits is not sufficient. Second, Stream Sockets maintains an LRU cache of registered application buffers to reduce memory registration cost. Finally, checking for completion of a send descriptor is deferred until either there are not enough send credits available or the entire message is posted, so that the CPU overhead can be reduced. We note that the credit-based flow control mechanism used in Stream Sockets tends to increase the average latency, because *CreditResponse* is not delivered to the sender until it is explicitly requested using *CreditRequest*. Also, caching memory registration information requires a modification to the default `malloc()` and `free()` routines, as the user application may free the (cached) registered memory region without notice. Other issues, such as connection management and dealing with special system calls, have not been addressed in Stream Sockets.

Recently, Myricom Inc. has released a software called Sockets-GM [27] for Myrinet interconnect, which works on Linux, Solaris, and Windows. Sockets-GM is built on top of Myrinet GM library and makes use of companion sockets in the same way as WSDP. It is intended to provide a binary compatibility with the existing applications written for the Sockets API. However, the current implementation of Sockets-GM simply stops using a companion socket and then reverts to a normal TCP/IP socket, once a connection is shared by other processes.

For Unix-flavored systems, it is very difficult to emulate Sockets API completely at the user-level, because Sockets-related data structures are kept inside the kernel and may be shared with child processes. In spite of the problem, our objective is to use the SOVIA layer to accelerate the existing Sockets-based applications with a reasonable effort and to provide a portable and high-performance communication library based on VIA to application developers. To the best of our knowledge, SOVIA is the first implementation of a user-level Sockets layer over VIA on Unix/Linux platforms.

3. DESIGN ISSUES

In this section, we investigate several design issues to improve the performance and the portability of the SOVIA layer. Our main design principle is to make the SOVIA layer as efficient as possible so that the performance of native VIA can be delivered to user applications, while retaining the portable Sockets semantics as much as possible.

3.1. Minimizing latency

3.1.1. Satisfying the pre-posting constraint

The VIA requires that the receiver should pre-post a descriptor to the RQ before the sender requests a data transfer. Otherwise, the transfer can be lost and the error is not even detected by the sending or the receiving side on an unreliable VI. We call this a *pre-posting constraint*. To satisfy the pre-posting constraint, there should be a high-level synchronization protocol between the sender and the

receiver, with which the sender can guarantee that at least one descriptor is available on the RQ of the destination VI.

One way to achieve this synchronization is to get an explicit permission from the receiver for each data transfer. Whenever the sender has data to transmit, it first sends a special REQ packet to the receiver asking for permission. If the receiver becomes ready, it pre-posts two descriptors on its RQ, one for data and the other for the next REQ, and replies to the sender with an ACK packet. Upon the receipt of the ACK packet, the sender is allowed to transmit a DATA packet which carries real data. Normally, the receiver does not answer with the ACK packet until the user application at the receiving node calls `recv()`. Therefore, the receiver always knows the target memory address and it is possible for the NIC to move the incoming data directly to the user space. This method has, however, the overhead of exchanging REQ and ACK packets before each data transfer, and this overhead has a substantial impact on the latency, especially for small messages.

Instead, SOVIA uses a simpler two-way handshaking, where a DATA packet is sent to the receiver immediately after the sender receives an ACK packet for the previous data transfer. In two-way handshaking, the DATA packet may arrive before the receiver calls `recv()`. Therefore, the application at the destination node should temporarily buffer the incoming DATA packet by pre-posting a descriptor into the RQ. Such an intermediate buffering at the receiving side also increases latency, but the overhead is much smaller than the cost of using the REQ packet.

3.1.2. Handling incoming packets

In a traditional communication architecture, incoming packets are handled by an interrupt handler in a transparent way to user applications. With VIA, however, the user application itself should extract the completed descriptors from the queue and post a new one for each incoming packet that is delivered asynchronously. Although the arrival of a packet is not automatically notified to user applications in VIA, it is possible to run a specific code upon the completion of a descriptor by registering a notify function in advance. The main task of the notify function will be to pre-post a descriptor, send an ACK packet, and then wake up the application thread if it has been suspended on `recv()`.

Unfortunately, Giganet's cLAN does not support notify functions as yet. We can still emulate the role of notify functions by creating a dedicated handler thread manually which monitors a CQ. If one of the pre-posted descriptors is completed, the handler thread locates the corresponding VI and performs the same task executed by the notify function.

Using the separate handler thread makes SOVIA a multi-threaded application, which means the main application thread may need to wait for a signal from the handler thread and any data shared by these two threads should be protected with mutexes. However, we find that the synchronization cost between two threads is expensive in Linux. In many cases, the handler thread is required to wake up the application thread by sending a signal on the critical communication path, which takes, according to our measurement, up to tens of microseconds. Considering that the latency of native VIA is less than 10 μs on cLAN, this high thread synchronization cost is the main source that increases latency (cf. Section 5.2). In addition, waiting on a condition variable or sending a signal requires kernel intervention as Linux supports kernel-level multi-threading, hence true user-level communication can not be achieved.

To overcome these problems, we have developed a single-threaded implementation of SOVIA, where the application thread itself is in charge of the handler thread's functionality. In SOVIA, the

incoming packets are handled by the application thread when it calls communication-related functions, such as `send()` or `recv()`. Communication may be delayed while the application thread at the receiving node is busy for computation, but by pre-posting multiple descriptors in advance (described in Section 3.2.1), it is possible to overlap communication and computation to some extent.

3.1.3. Avoiding memory registration cost

As the location of temporary buffers used by the receiver is fixed, they can be pre-registered during initialization. In contrast, the sender's data can not be pre-registered because we do not know which will be transmitted beforehand. Therefore, each data transfer experiences one memory registration at the sender side. The memory registration is the key element in VIA which enables the zero-copy protocol, but it is a relatively expensive operation for small messages. Instead, we can consider the use of sender-side buffering, where the outgoing data are simply copied into a pre-registered buffer before the corresponding descriptor is posted in a send queue. However, we note that the sender-side buffering is harmful for large messages, because the cost of memory copying increases more rapidly than the cost of memory registration.

To reduce the latency further, we use a simple hybrid approach, where data are copied into the buffer only if the requested size is small. Otherwise it is registered. According to our measurement result, we find it is reasonable to begin registering data as their size becomes larger than 2 kB.

3.2. Maximizing bandwidth

3.2.1. Flow control

A long message, requested to be sent by the application, is divided into a series of DATA packets. We define the *maximum transfer unit (MTU)* as the largest amount of data that may be transferred using a single DATA packet. The MTU is directly related to the bandwidth; the larger the MTU, the higher the bandwidth. However, usually the NIC places a limitation on the MTU[‡]. In SOVIA, we assume the default MTU of 32 kB unless otherwise stated.

Under the synchronization scheme described in Section 3.1.1, the sender should wait for an ACK packet in order to send the next DATA packet. The ACK packet informs that the receiver has pre-posted a descriptor to the corresponding RQ and is ready to receive another DATA packet. As a result, there is at most a single outstanding DATA packet on the VI at any given time, under-utilizing the physical resource. TCP has a flow control algorithm called a *sliding window protocol* [28], which allows the sender to transmit multiple packets before it stops and waits for an acknowledgement. This leads to higher bandwidth since the flow of packets can be pipelined.

SOVIA supports a flow control mechanism similar to the TCP's sliding window protocol. Our implementation of SOVIA also has the notion of *window size* w, which denotes the maximum number of DATA packets the sender is allowed to transmit without waiting for an acknowledgement. Initially, the receiver pre-posts w descriptors to RQ. Whenever the sender transmits a DATA packet, it decreases w by one meaning that one of the pre-posted descriptors on the receiving end has been

[‡]For example, the MTU can not be set beyond 65 486 bytes on Giganet's cLAN.

consumed. If w reaches zero, there are no available descriptors on the receiver and further transmission is on hold until w becomes a positive number. The window size w is increased by one if an ACK packet is delivered to the sender acknowledging one of the previous DATA packets. As the window is maintained on a per-connection basis, our flow control mechanism ensures correct operation even when a server talks to many clients simultaneously.

3.2.2. Delayed acknowledgements and piggybacking

Normally, a single ACK packet is generated on the receiving end for each DATA packet. The number of ACK packets can be reduced by combining several acknowledgements together directed to the same sender. In TCP, the receiver delays sending an ACK packet, typically up to 200 ms, hoping to have data going in the same direction as the ACK packet. If there are data to send, all the delayed ACK packets are *piggybacked*, i.e. sent along with the data.

SOVIA also takes advantage of delayed acknowledgements and piggybacking, by using an adaptation of the TCP's algorithm. In SOVIA, the receiver simply counts the number of ACK packets (d) that are being delayed. If d becomes greater than the predefined threshold t, where $t < w$, an ACK packet is delivered to the sender piggybacking d. This will increase the sender's window size w by d. On the other hand, when the receiver has a DATA packet for the same direction before d reaches t, delayed acknowledgements are piggybacked with the DATA packet.

3.2.3. Combining small messages

Another useful feature of TCP is the *Nagle algorithm* [28] enabled by default. The algorithm requires that when a TCP connection has outstanding data that have not yet been acknowledged, small messages can not be sent until the outstanding data are acknowledged or until TCP can send a full-sized packet. The Nagle algorithm was originally developed as a way to avoid congestion on wide-area networks, but has a side-effect to batch small messages together. For SOVIA, it is also desirable to have a similar feature, where the consecutive data transfer requests of small-sized messages are combined into a larger packet.

Our algorithm to combine small messages works as follows. The implementation of SOVIA already has an internal buffer which is used for sender-side buffering (cf. Section 3.1.3). Previously, a small message of less than 2 kB was copied into the buffer and then sent immediately if the window size permits. However, now such a small message is simply appended into the buffer and the sender starts a timer which expires after, say, 100 ms. Any other small messages requested within the expiration of the timer are also combined as long as there is enough room in the buffer. For messages larger than 2 kB, it is too expensive to copy data, hence the current buffer is flushed and then the new message is transferred in a normal way.

Let S_{buf} be the message size currently held in the buffer and S_{req} be the new message size requested by the application, where S_{buf} can not be larger than the MTU. We can summarize that data stored in the buffer are transmitted to the network either (1) when the timer expires, (2) when $S_{\text{buf}} + S_{\text{req}} >$ MTU, (3) when $S_{\text{req}} > 2$ kB, or (4) when the application calls `recv()` or `close()`.

3.3. Connection management

Our choice of using the single-threaded implementation in Section 3.1.2 poses some problems related to the connection management. The Sockets semantics allows us to close a socket connection partially

in one direction. To support the *partial close*, data structures related to the current connection should not be removed at the time of calling `close()`. Instead, the application is required to return immediately from `close()`, after sending only an intention of the close to the peer. TCP uses two special packets for this purpose, namely FIN and FINACK. The FIN packet is used to notify the peer of the intention to close the current connection, and the FINACK acknowledges the receipt of the FIN. To close a connection completely, both end-points should agree on it by exchanging a FIN and the associated FINACK packet. However, in our single-threaded implementation, once the application executes the last `close()`, it does not call any Sockets API function and there is no chance to handle incoming FINACK and/or FIN packets from the peer, which are necessary to receive before removing the associated data structures.

To solve this problem, SOVIA creates a *close thread* and asks it to handle incoming packets whenever the number of open socket connections becomes zero. If a new connection is established while the close thread is running, the close thread is disabled so that its presence does not affect the application's communication performance.

The second problem occurs due to the difference in the connection model. The Sockets API has a connection model that a server process specifies a willingness to accept an incoming connection with `listen()` and accepts it with `accept()`. If a remote process (i.e. a client) wants to initiate a connection, it calls `connect()`. The VIA has a slightly different connection model especially on the server side. `VipConnectWait()` is used for a server to wait for incoming connection requests, and, if there is any, `VipConnectAccept()` is called to associate the connection with a local VI end-point.

`VipConnectWait()` is conceptually similar to `accept()` in that both of them are blocked until a connection is requested. However, the Sockets semantics requires that the client should be able to return from `connect()` successfully even though the server, after calling `listen()`, has not yet reached `accept()`. We can not meet this Sockets behavior if `accept()` is directly implemented with `VipConnectWait()`. This implies another thread should be running to accept a VI connection request behind the application thread and SOVIA uses a special *connection thread* for this task.

3.4. Enhancing portability

It is one of our goals to enable application programs written for the Sockets API to take advantage of the high performance of VIA seamlessly through the SOVIA layer. Therefore, it is important for the SOVIA layer to provide enough portability so as to minimize any code modification when the existing Sockets-based applications are migrated to the SOVIA layer.

SOVIA provides its own version of Sockets API, such as `sov_socket()`, `sov_connect()`, `sov_send()`, `sov_close()`, etc. In order to exploit the SOVIA layer transparently at the source level, we may consider a static replacement of the existing interfaces as follows:

```
#define socket          sov_socket
#define close           sov_close
```

However, such a source-level modification relying on macro definitions has a couple of drawbacks. First, normal TCP/UDP sockets can not coexist with SOVIA, as every `socket()` call will be replaced with `sov_socket()`. Second, Sockets are also accessed by file system interfaces due to the fact that socket descriptors are treated as file descriptors in Unix. We can not replace all the `close()`


```
int  s;
FILE *fp;
...
s = socket (AF_INET, SOCK_STREAM, 0);
connect (s, (struct sockaddr *) &server, sizeof(server));
fp = fdopen (s, "w");
fprintf (fp, "Hello, world...\n");
```

Figure 3. Accessing Sockets through the standard I/O library.

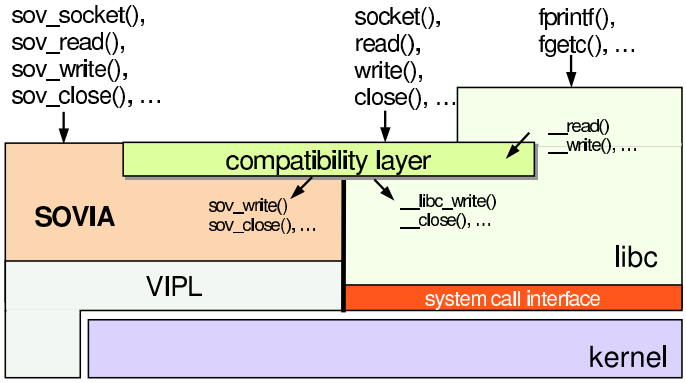


Figure 4. Enhancing portability by adding a compatibility layer.

system calls with `sov_close()`, because some of them will be used for closing files instead of socket connections. This means application programmers need to examine their source codes carefully and make sure that only the system calls on socket descriptors are converted to the matching SOVIA interfaces properly. A worse problem of the static replacement approach is that it can not deal with the situation where Sockets are accessed through a number of standard I/O library routines. For example, once the pointer to the `FILE` structure is obtained from a socket descriptor via `fdopen()`, messages can be also sent using any of library routines (such as `fprintf()`), as a code fragment in Figure 3 shows.

To cope with the aforementioned problems, we use a dynamic approach where our version of Sockets API is selected at run-time based on the socket descriptor. Fortunately, GNU's C library (`libc`) defines system call interfaces as weak symbols so that they can be overridden in user codes. We implement a thin *compatibility layer* on top of SOVIA, which has wrapper functions for file system interfaces, as illustrated in Figure 4. The compatibility layer is statically linked with the existing Sockets-based application and is used to intercept system calls issued by the application. Note that it is also possible to preload the object code of SOVIA dynamically by setting the `LD_LIBRARY_PATH` environment variable.

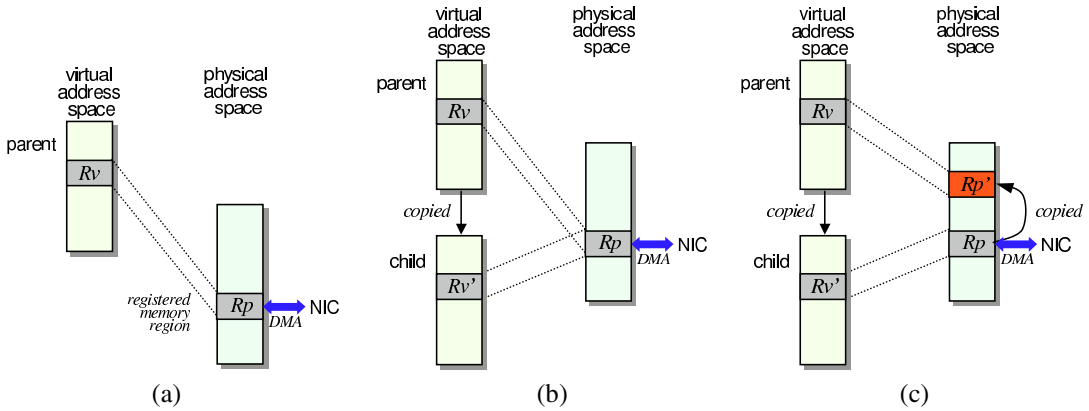


Figure 5. Copy-on-write problem: (a) before `fork()`; (b) after `fork()`; (c) when the parent writes to R_v.

For the system calls operating on normal file descriptors, it is necessary for the compatibility layer to locate the addresses of the original functions in `libc`. For many system calls, such as `write()` and `close()`, the original (strong) symbols are also exported by `libc` (`__libc_write()` and `__close()` respectively) and we can call those symbols directly in the compatibility layer. In other cases, we can use the `dlsym()` function to obtain the address where a particular symbol is loaded in a dynamic library. We also need to override some of other functions, such as `__write()` and `__read()`, which are internally called in `libc` by standard I/O library routines. Once they are captured in the compatibility layer, any access through the standard I/O library can be switched to the SOVIA layer transparently.

3.5. Problems with `fork()`

3.5.1. Copy-on-write problem

An FTP server process forks a child process when it receives a '`dir`' command from a client. The child process executes '`/bin/ls -lgA`' on the current directory and the output is redirected to the FTP server via pipes, which will eventually be transferred to the FTP client as a response of the '`dir`' command. However, a naive port of the FTP server may not work if the VI Kernel Agent is not implemented carefully; it may cause a problem when a child process is created using the `fork()` system call. We elaborate upon the situation in Figure 5.

As described in Section 2.1, a memory region R_v needs to be registered before it is used for any communication. During the registration, the VI Kernel Agent converts the virtual page addresses for R_v into physical addresses and pins the corresponding region R_p in the physical memory. R_p is then used by the NIC hardware for Direct Memory Access (DMA) (cf. Figure 5(a)).

If the process forks a child, the Linux kernel optimizes the virtual memory system by copying the data structures describing virtual memory and by sharing the actual physical pages, as shown in Figure 5(b). A problem occurs when the parent process accesses the registered region R_v after `fork()`; if a write is done, the child gets the physical pages R_p and the parent gets new physical pages R'_p which are copies of R_p, as illustrated in Figure 5(c). This optimization is called a *copy-on-write*. The copy-on-write policy is introduced to reduce the latency of forking a process while decreasing the memory usage. However, now the NIC hardware will use physical pages R_p that are no longer mapped to virtual addresses of the parent, R_v. Note that this problem happens even if the child process is not involved in any communication.

It is too restrictive if user applications are not allowed to create child processes while they are using VIA. SOVIA solves this problem by allocating pre-registered descriptors and data buffers on a shared memory segment. The pages located in shared memory segments are shared between the parent and the child processes without causing the copy-on-write problem. We find that both cLAN and M-VIA drivers register a region of memory inside the VIPL to implement completion queues, and we had to modify the VIPL so that the region is also allocated on a shared memory segment[§].

3.5.2. Supporting concurrent server daemons or `inetd`

Sharing a socket connection between two processes is even more difficult. This is a required feature to support concurrent server daemons or 'super-server' daemons such as `inetd`. Data structures used by the SOVIA layer may be shared among processes through shared memory segments. However, to completely share a socket connection, it should be possible to share a VI connection between multiple processes, which will require extensive modification in the implementation of VI Kernel Agent and VIPL.

Instead, we choose a partial solution, where the client connects to `inetd` in a normal way using a TCP socket. If a server daemon is forked from `inetd`, the server closes the TCP socket and then opens a new VI connection with the client. The server and the client codes need to be modified slightly to make one more connection at the beginning, but this change is transparent to `inetd`. In this way, we could invoke the FTP server daemon via `inetd`.

4. IMPLEMENTATION

In this section, we briefly describe the implementation details of several representative Sockets API functions.

4.1. `socket()`

The organization of main data structures used in SOVIA is outlined in Figure 6. SOVIA initializes its global data structures when the first `socket()` function is called. The global data structures

[§]The Giganet's cLAN driver version 1.1.1 used in this paper has this problem. The latest cLAN driver version 1.3.0 has fixed the problem by setting a `PG_reserved` flag to the registered pages. However, M-VIA implementation still has this problem.

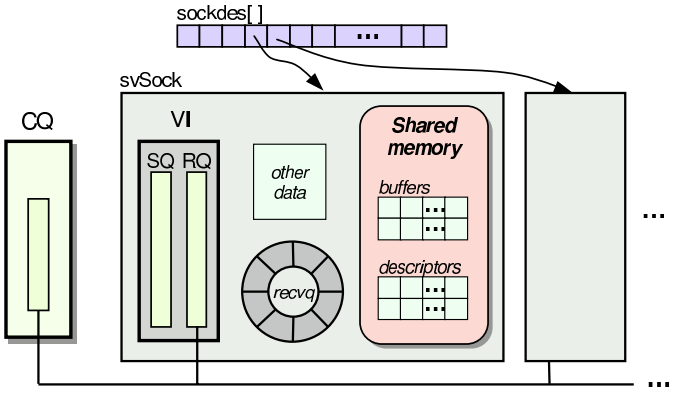


Figure 6. Data structures used in the SOVIA layer.

include a CQ and a `sockdes[]` table. The CQ is associated with a RQ whenever a new VI is created. The `sockdes[]` table roughly corresponds to the per-process file descriptor table in the kernel and holds a pointer to the socket-specific data structure called `svSock`. In addition, a close thread is created and activated during initialization. Note that creating a connection thread is postponed until the application issues `listen()`.

We introduce a new socket type called `SOCK_VIA`, which is similar to the `SOCK_STREAM` type used for TCP sockets. When a user calls '`socket (AF_INET, SOCK_VIA, 0)`' to create a SOVIA-type socket, we open a dummy file ('`/dev/null`') to obtain a file descriptor s from the kernel. SOVIA then allocates and initializes a `svSock` structure for the new socket, storing its address at `sockdes[`s`]`. Finally, `socket()` returns s as a new socket descriptor. Later, if a system call on s' is intercepted by the compatibility layer, SOVIA first checks if the entry in `sockdes[`s'`]` is valid or not. A valid entry in `sockdes[`s'`]` means s' is a socket descriptor of type `SOCK_VIA` and the requested system call is switched to the corresponding routine in the SOVIA layer. Otherwise, the original system call interface in `libc` is invoked.

The `svSock` structure keeps all the information required for a particular socket connection. It contains a VI, a circular queue (`recvq`), pre-registered descriptors and data buffers, mutexes and conditional variables to synchronize accesses between multiple threads, and other data to maintain the state of the connection. As discussed in Section 3.5.1, descriptors and data buffers are allocated in a shared memory segment to avoid the copy-on-write problem. The `recvq` is used to link the incoming data in a circular fashion which are not yet read by the application. A VI is not actually created until the application requests a connection establishment via `listen()` or `connect()`.

4.2. `listen()`, `accept()`, `connect()`, and `close()`

SOVIA uses five types of packets: DATA, ACK, WAKEUP, FIN, and FINACK. The type of a packet is encoded in the 32-bit *Immediate Data* field of the descriptor without consuming any extra memory.

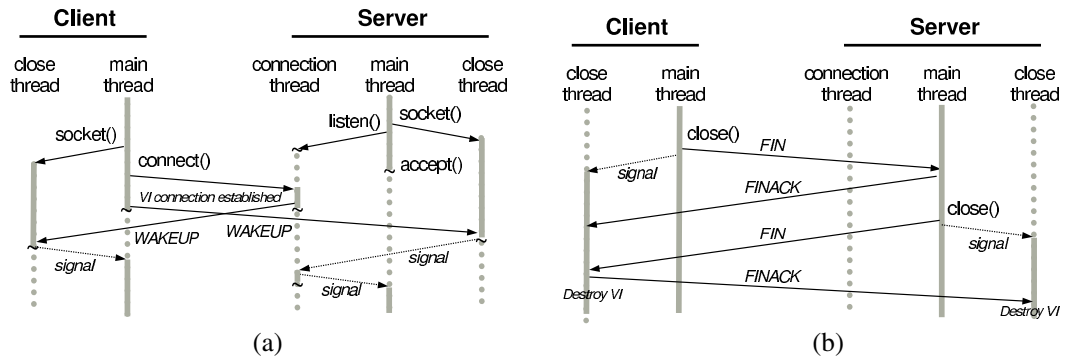


Figure 7. Establishing and closing a connection in SOVIA: (a) establishing a connection; (b) closing a connection.

The DATA and the ACK packets are used for transmitting data and delayed acknowledgements, respectively. The FIN and the FINACK packets are used for closing a connection as described in Section 3.3. The WAKEUP is a special packet that is exchanged between two end-points as soon as a new connection is established. It is used to inform the peer about the sender's socket descriptor, IP address, and port number.

An example sequence of operations during the connection establishment is shown in Figure 7(a). As noted in Section 4.1, a close thread is created when the first `socket()` function is called. On the server side, SOVIA creates a connection thread each time the application issues `listen()` on a port. Initially, the connection thread waits for incoming connection requests in `VipConnectWait()`. For each incoming request, the connection thread accepts it by calling `VipConnectAccept()`, and then stores the information in a queue shared with the application thread. If the application thread finds the queue empty on `accept()`, it is suspended waiting for a signal from the connection thread. Otherwise, the application thread extracts an entry from the queue and completes `accept()` returning a new socket descriptor.

One of the challenging issues is to deactivate the close thread when a new connection is established, in order to minimize the overhead during the actual data transfer. We solve this problem using the WAKEUP packet. Recall that the WAKEUP packet is the first packet received from the peer after a new connection is set up. Therefore, if the WAKEUP packet is received while the close thread is handling incoming packets, it means a new connection has been established and we suspend the execution of the close thread immediately. Obviously the WAKEUP packet may be received by the application thread if there are one or more open connections, but no special action is taken in this case.

Figure 7(b) illustrates the situation where a connection is closed. The application thread returns from `close()` after sending a FIN packet to the peer but the associated `svSock` structure is not destroyed until both end-points completely exchange the FIN and the FINACK packets. In addition, if the currently closed connection is the last one in the system, the suspended close thread is enabled again as a result of calling `close()`.

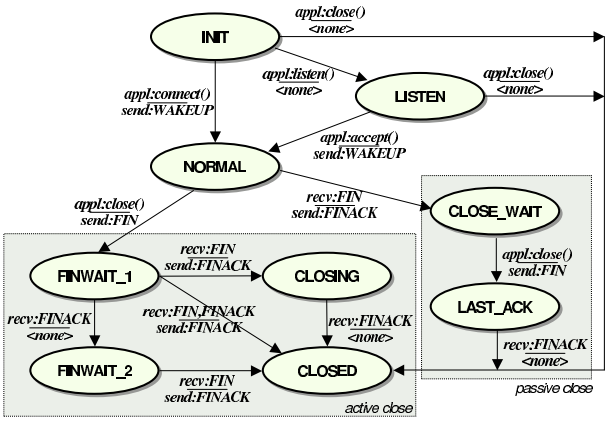


Figure 8. The state transition diagram of SOVIA.

4.3. `send()` and `recv()`

Figure 8 summarizes the states and the state transitions in SOVIA, which are very similar to the TCP's [28]. The notation $\frac{appl:close()}{send:FIN}$ denotes a state transition when an application issues `close()`, and specifies that a FIN packet is sent to the peer as a result of the transition. The states in the shaded areas in Figure 8 are used for supporting active and passive closes. The NORMAL state represents that a connection has been established between two end-points and all the data transfers are performed in the NORMAL state.

A sending operation may be blocked if there is not enough window size on the connection. In that case, the application thread executes an internal packet handling routine called `sovia_handler()` until the window size is increased. The window size is increased by ACK packets or by delayed acknowledgements piggybacked in DATA packets. Similarly, when the application requests `recv()`, SOVIA first checks the `recvq` to see if any data arrived before the `recv()`. If the `recvq` is empty, the application thread executes the `sovia_handler()` expecting a DATA packet from the peer. The `sovia_handler()` is responsible for receiving packets by monitoring the CQ, and updating state information of the connection according to the state transition diagram shown in Figure 8. The same `sovia_handler()` is also used by the close thread.

As described in Section 3.2.1, our flow control mechanism requires pre-posting a number of descriptors to the RQ. For a given window size w and a delayed acknowledgement threshold t, there can be maximum w/t outstanding ACK packets for a particular connection. As the peer may send a FIN packet immediately after w/t ACK packets, at least $P = w + w/t + 1$ descriptors should be pre-posted to each RQ in advance. Similarly, we can calculate that the `recvq` should have $P + 1$ slots to store pointers to P incoming packets in the worst case, where one additional slot is used to correctly implement a circular queue.

4.4. Finalization

Before the application terminates, we should ensure a clean-up routine is called for the following reasons.

- SOVIA pre-registers a region of memory that is used for descriptors and data buffers. They should be deregistered before the application exits.
- The completion queue (CQ) should be destroyed as it also internally registers a certain amount of memory in many VIPL implementations.
- SOVIA allocates descriptors and data buffers in shared memory segments, which should be detached upon the completion of program execution.
- Applications may exit without closing established connections. For a graceful termination, we should close active connections by calling `close()` inside the clean-up routine.
- If an application exits after calling `close()` on an active connection, it is possible that the FIN and the FINACK packets have not yet arrived from the peer at the time of termination. In that case, we should wait until the socket connection reaches the CLOSED state.
- The close thread (and the connection thread, if any) should be explicitly killed because it may hang in the system after program termination.

For a normal termination, SOVIA registers the clean-up routine by calling `atexit()` during initialization. This allows us to invoke the registered clean-up routine automatically when the program terminates with `exit()`.

Even though we use `atexit()`, it is possible for the application to abort its execution upon the receipt of signals (such as `SIGQUIT` or `SIGSEGV`) without invoking the clean-up routine. For such an abnormal termination, SOVIA initially registers a signal handler whose task is similar to that of the clean-up routine. However, as the signal handler is executed in an emergency situation, it simply disconnections all the current VI connections without making any effort for a graceful termination. In some implementations, the lost VI connection can be discovered by the peer through the use of `VipErrorCallback()`. Note that our signal handler may be overridden if the application registers its own signal handler for the same signal. In such a case, we may either disable the application's signal handler, or chain two signal handlers by intercepting the `signal()` system call at the compatibility layer.

5. EXPERIMENTAL RESULTS

5.1. Evaluation platform and benchmarks

The hardware platform used for performance evaluation is two Linux servers based on Intel L440GX+ motherboards running Linux kernel 2.2.16. Each server consists of a Pentium III-500 MHz microprocessor with 512 kB of L2 cache and 256 MB of main memory. Two cLAN1000 network adapters are attached to a 32-bit 33 MHz PCI slot of each server without an intermediate switch. We have used cLAN driver version 1.1.1 and the TCP performance on cLAN is measured using the LANE driver supplied by Giganet.

To quantify the impact of each optimization on the application's performance, we use microbenchmarks which measure the latency and the bandwidth. The (one-way) latency is measured by a half of the round-trip time in a ping-pong test. To measure the (unidirectional) bandwidth, the sender issues a long rapid sequence of `send()`'s for more than 60 s and waits for a reply message, which is sent by the receiver after reading the last transmitted data successfully. The same benchmarks are implemented using the VIPL as well, to compare the performance of SOVIA with that of native VIA.

In addition, we have ported FTP and RPC applications over the SOVIA layer in order to show the functional compatibility with the existing Sockets API. The performance details of these applications will be discussed in Sections 5.3 and 5.4, respectively.

5.2. Performance on microbenchmarks

Figure 9(a) compares the latency of SOVIA with those of TCP and native VIA on Giganet's cLAN. First of all, we note that native VIA outperforms TCP as expected; native VIA shows a latency of 8.5 μs for 4-byte packets, while TCP shows 55 μs for the same condition.

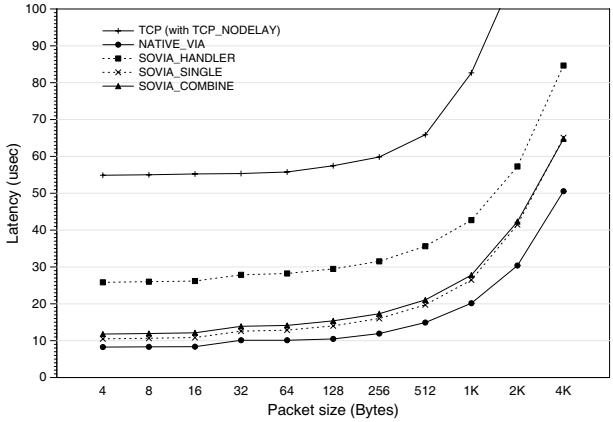
Looking at the performance results of SOVIA, we can see that the implementation using a separate handler thread (SOVIA_HANDLER) results in significantly higher latency compared to the single-threaded implementation (SOVIA_SINGLE). As described in Section 3.1.2, the thread synchronization cost is responsible for the gap between SOVIA_HANDLER and SOVIA_SINGLE, which increases latency by about 16 μs.

This observation has been confirmed using a benchmark program which accurately measures T_{sync}, the cost to wake up a blocked thread by sending a signal in a multi-threaded application. We find that T_{sync} is about 17.0 μs on the same platform, which is very close to the difference in latency between SOVIA_HANDLER and SOVIA_SINGLE. Unlike general multi-threaded applications, running SOVIA_HANDLER on SMP systems does not improve the latency, as T_{sync} is rather increased to 31.0 μs on a dual-processor system. This is because rescheduling a blocked application thread requires kernel services and SMP kernels have an additional overhead of acquiring and releasing a spinlock to access kernel data structures. The latest Linux 2.4 kernels are known to show better performance than Linux 2.2 kernels, but T_{sync} has not been improved much; under Linux 2.4.8, T_{sync} was still 17.4 μs on a uniprocessor system and 29.3 μs on a dual-processor system. This suggests that modern kernels can hardly improve the performance of SOVIA_HANDLER.

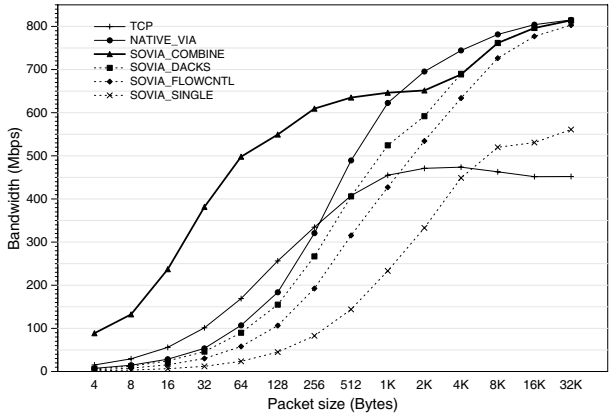
Figure 9(a) also shows the changes in latency when we try to combine small messages together (SOVIA_COMBINE). Combining small messages increases the latency of SOVIA by 1–2 μs to manage a software timer. However, this feature may be turned off at run-time for latency-sensitive applications in the same way as TCP, where the Nagle algorithm is disabled by specifying the `TCP_NODELAY` socket option. Overall, we can reduce the latency of the SOVIA layer to as low as 10.5 μs for cLAN on our platform, adding only 2 μs of overhead to the native VIA's latency.

Figure 9(b) illustrates the measured bandwidth of TCP, native VIA, and SOVIA. Again, native VIA shows higher bandwidth than TCP when the packet size is larger than 256 bytes[¶]; the bandwidth of

[¶]The socket buffer size of TCP is increased to the maximum (131 170 bytes) during the measurement.



(a)



(b)

Figure 9. The latency and bandwidth on Giganet's cLAN: (a) latency; (b) bandwidth.

TCP for 32 kB packets is limited to about 450 Mbps, attaining only 55% of native VIA's performance (815 Mbps).

In Figure 9(b), SOVIA_SINGLE represents the bandwidth obtained by the single-threaded implementation with conditional sender-side buffering, which has minimized latency in Figure 9(a). The graph labeled SOVIA_FLOWCTRL denotes the bandwidth when our flow control mechanism is added to SOVIA_SINGLE. SOVIA_DACKS adds the flow control and the delayed acknowledgements to SOVIA_SINGLE. We can see that the bandwidth of SOVIA_DACKS is notably improved compared to SOVIA_SINGLE. In particular, if the packet size is greater than or equal to 16 kB, SOVIA_DACKS

Table I. The performance of file transfers using FTP.

	File 1 size 19 090 223 bytes		File 2 size 145 864 380 bytes	
TCP/IP on Fast Ethernet	90 Mbps	(1.63 s)	90 Mbps	(12.7 s)
TCP/IP on cLAN	262 Mbps	(0.59 s)	254 Mbps	(4.61 s)
SOVIA on cLAN	573 Mbps	(0.27 s)	532 Mbps	(2.20 s)
Local copy (on RAM disks)	611 Mbps	(0.25 s)	538 Mbps	(2.17 s)

shows roughly the same bandwidth as native VIA. In this experiment, we have used the window size of 32 ($w = 32$) and the threshold is set to 16 ($t = 16$).

TCP shows higher bandwidth than native VIA for small messages of less than 256 bytes, due to the Nagle algorithm. Similarly, we can see that SOVIA_COMBINE, which adds the ability to combine small messages to SOVIA_DACKS, improves the bandwidth substantially for messages less than 2 kB.

Note that the results shown in Figure 9 are obtained from a version of SOVIA, which polls a send queue or a completion queue to find completed descriptors. Alternatively, the VIA specification provides several blocking APIs, with which we can perform the same task without polling. When we use blocking APIs, the latency of SOVIA is increased to 19.4 μs for 4-byte packets and the peak bandwidth is reduced to 780 Mbps. In spite of the slight performance degradation, using blocking APIs have an advantage that they do not spend any CPU cycles on waiting for completion of posted actions.

5.3. FTP performance

We have measured the performance of file transfers between two nodes by modifying an FTP server (`linux-ftpd-0.16`) and a client (`netkit-ftp-0.16`) contained in Linux NetKit 0.16. Table I compares the measured bandwidth and the elapsed time reported by the FTP client for two different sizes of files. To remove the effect of disk speed, the source and destination files are stored in RAM disks.

The core loop of the file transfer operation is actually the same as our microbenchmark which measures the bandwidth. Therefore, it is expected for FTP applications to achieve the peak bandwidth shown in Figure 9(b). The measured bandwidth is, however, slightly lower than the expected one, showing 573 Mbps and 532 Mbps for 19 MB and 145 MB files, respectively. This is because the actual data transfer is bounded by the file system performance, which has a bandwidth of 538 Mbps to 611 Mbps for local RAM disk-to-RAM disk copy of the same files. When the file size is small, the transfer completes by writing the entire file into buffer caches even without going to the file system layer. This is why the smaller file transfer results in higher bandwidth in Table I. For TCP/IP on Fast Ethernet, the performance does not depend on the file size, as the network becomes a bottleneck in this case.

Finally, we note that the raw performance of VIA is not delivered to user applications efficiently using the kernel-level TCP/IP driver on cLAN. Although the peak bandwidth of native VIA is

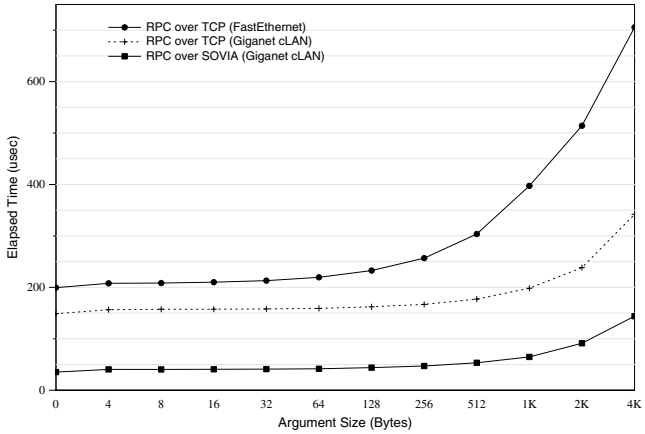


Figure 10. Average elapsed time to call an empty remote procedure.

815 Mbps, FTP applications result in bandwidths of less than 300 Mbps with the TCP/IP driver, exploiting only 32% of the available bandwidth. Overall, the SOVIA layer easily doubles the performance of FTP applications compared to the LANE driver.

5.4. RPC performance

The goal of RPC is to make a network function call as simple as any local function call. In RPC, the calling arguments are passed to a remote procedure and the caller waits for a response to be returned from the remote procedure. RPC hides all the network code in client '*stubs*' and server '*skeletons*' that are automatically generated from the specification of remote procedures. RPC also isolates the application from the physical and logical elements of data communication mechanisms and allows the application to use a variety of transports. This transport independence of RPC and the fact that it is usually implemented as a user-level library, ensure RPC applications easily benefit from high-performance, user-level communication protocols such as VIA.

If the RPC protocol is implemented directly over VIPL, it will require extensive modification in the RPC layer. Instead, we slightly modify the `rpcgen` tool to generate VIA-specific interface modules and link them with the SOVIA layer. As the SOVIA layer emulates the Sockets API, the modification in the RPC layer is minimized. The client simply selects SOVIA as the base transport by specifying '`via`' when it calls `clnt_create()` and there are no other changes visible to application developers.

For the experiment, we have used `sunrpc` implementation in `glibc-2.1.3`, which is available on our platform by default. Figure 10 compares the average elapsed time for a single RPC for various argument sizes ranging from 0 to 4 kB. An argument is passed to a remote procedure as a character string, and the body of the remote procedure is empty returning an integer value. The argument size of zero ($x = 0$) represents the case where the argument of the remote procedure is defined as

`void`. Note that even for the null argument, packets are exchanged between the server and the client containing an RPC header: 44 bytes for request, and 28 bytes for response.

Unlike FTP, RPC is a latency-sensitive application. Our measurement result shows that calling an empty remote procedure with the null argument takes about 200 μs on Fast Ethernet and 149 μs on cLAN, when the traditional TCP is used as a transport. In contrast, making an RPC over the SOVIA layer takes only 35 μs for the same condition, which is 4.3 times faster than the case with the kernel-level TCP/IP driver on cLAN.

6. CONCLUDING REMARKS

This paper presents the design, implementation, and performance evaluation results of SOVIA, a user-level Sockets layer over VIA. Because adding a new software layer introduces an overhead inevitably, special attention has been paid to making the SOVIA layer lightweight, while retaining the portable Sockets semantics. SOVIA shows a minimum one-way latency of 10.5 μs and a peak bandwidth of 814 Mbps on Giganet's cLAN, which is comparable to the native VIA's performance. The functional compatibility with the existing Sockets API is shown by porting FTP and RPC applications over the SOVIA layer. In these applications, the SOVIA layer significantly outperforms the Giganet's LANE driver which emulates TCP/IP inside the kernel. Using the SOVIA layer easily doubles the file transfer bandwidth in FTP and reduces the latency of calling an empty remote procedure by 77% in RPC applications.

The goal of the SOVIA layer is three-fold. The first goal is to provide a portable and high-performance communication library based on VIA to application developers. The second goal is to migrate the existing Sockets-based applications with a reasonable effort so as to accelerate their performance on VIA-enabled cluster systems. We are currently porting a user-level parallel file system over the SOVIA layer to demonstrate this. Finally, we expect that the SOVIA layer can be a target for further upper layers as proved by our simple port of the RPC layer.

Currently, the SOVIA layer has the following limitations. First, the `exec()` system call is not supported since it will destroy all user-level data structures. Second, as discussed in Section 3.5.2, a socket connection currently can not be shared between the parent and the child processes. Third, OOB data is not yet supported. We believe, however, that SOVIA can be easily extended to support OOB data by providing a separate VI connection or by reserving some descriptors for OOB data. Finally, applications should not terminate with `_exit()` because it will bypass the clean-up routine registered by `atexit()`. We can see that most of the problems occur due to the nature of user-level implementation.

Full compatibility can be attained when we move down into the kernel-level. In the near future, we will also implement a kernel-level Sockets layer over VIA to examine trade-offs between user-level and kernel-level implementations. In addition, we plan to extend our work on to the recent InfiniBand Architecture [29].

REFERENCES

1. Pfister G. *In Search of Clusters* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1998.
2. Buyya R (ed.). *High Performance Cluster Computing: Architectures and Systems*. Prentice-Hall: Englewood Cliffs, NJ, 1999.

3. von Eicken T, Basu A, Buch V, Vogels W. U-Net: A user-level network interface for parallel and distributed computing. *Proceedings of the Symposium on Operating System Principles*. ACM Press: New York, NY, 1995; 303–316.
4. Pakin S, Lauria M, Chien A. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. *Proceedings of Supercomputing*. IEEE Computer Society Press: Los Alamitos, CA, 1995.
5. Prylli L, Tourancheau B. Protocol design for high performance networking: A Myrinet experience. *Technical Report 97-22*, LIP-ENS Lyons, France, July 1997.
6. Chun B, Mainwaring A, Culler D. Virtual network transport protocols for Myrinet. *IEEE Micro* 1998; **31**:53–63.
7. Compaq Computer Corp., Intel Corp. and Microsoft Corp. Virtual interface architecture specification draft revision 1.0. http://www.viarch.org/ [December 1997].
8. Baker M (ed.). *Cluster Computing White Paper (Version 2.0)*. IEEE Task Force on Cluster Computing, December 2000.
9. Bozeman P, Saphir B. A modular high performance implementation of the virtual interface architecture. *Proceedings of the Extreme Linux Conference*, 1999.
10. Buonadonna P, Geweke A, Culler DE. An implementation and analysis of the virtual interface architecture. *Proceedings SC '98*. IEEE Computer Society Press: Los Alamitos, CA, 1998.
11. Boden NJ *et al.* Myrinet: A gigabit-per-second local area network. *IEEE Micro* 1995; 29–36.
12. Giganet Inc. *cLAN Hardware Installation Guide*, 2001.
13. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. *MPI: The Complete Reference*. The MIT Press: Cambridge, MA, 1996.
14. McKusick MK, Bostic K, Karels MJ, Quarterman JS. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley: Reading, MA, 1996.
15. National Energy Research Scientific Computing Center. MVICH: MPI for virtual interface architecture. http://www.nersc.gov/research/FTG/mvich/.
16. Conte G *et al.* ParMa^2: Porting VIA in LAM/MPI, University of Parma, Italy. http://www.ce.unipr.it/research/parma2/ [2000].
17. Dimitrov R, Skjellum A. An efficient MPI Implementation for Virtual Interface (VI) architecture-enabled cluster computing. *Proceedings of the 3rd MPI Developer's Conference*, 1999.
18. Gropp W, Lusk E. User's Guide for MPICH, a Portable Implementation of MPI, Argonne National Lab/Mississippi State University. http://www-unix.mcs.anl.gov/mpi/mpich/ [2000].
19. Ohio Supercomputer Center. MPI Primer/Developing with LAM. http://www.lam-mpi.org/ [1996].
20. Ong H, Farrell PA. Performance comparison of LAM/MPI, MPICH, and MVICH on a Linux cluster connected by a gigabit ethernet network. *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.
21. Giganet Inc. *cLAN for Linux: Software User's Guide*, 2001.
22. Itoh M, Ishizaki T, Kishimoto M. Accelerated socket communications in system area networks. *Proceedings Cluster 2000*. IEEE Computer Society Press: Los Alamitos, CA, 2000; 357–358.
23. Rodrigues SH, Anderson TE, Culler DE. High-performance local area communication with fast sockets. *Proceedings of the USENIX*, 1997.
24. Hellwagner H, Weidendorfer J. SCI Sockets Library. *SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters* (*Lecture Notes in Computer Science*, vol. 1734), Hellwagner H, Reinefeld A (eds.). Springer, 1999.
25. Microsoft Corp. Winsock Direct Concepts. http://www.microsoft.com/windows2000/en/datacenter/help/wsd_concepts.htm [2001].
26. Shah HV, Madukkarumukumana RS. Design and implementation of efficient communication abstractions on the Virtual Interface Architecture: Stream sockets and RPC experience. *Software—Practice and Experience* 2001; **31**:1043–1065.
27. Fischer M. Sockets-GM. *Myrinet User's Group Conference*, 2002.
28. Stevens WR. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley: Reading, MA, 1994.
29. InfiniBand Trade Association. InfiniBand Architecture Specification Release 1.0. http://www.infinibandta.org/ [October 2000].

