# SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture

Jin-Soo Kim, Kangho Kim, and Sung-In Jung
Electronics and Telecommunications Research Institute (ETRI)
Daejeon 305-350, Korea
E-mail: jinsoo@computer.org

## Abstract

*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 the application developers.*

SOVIA *realizes comparable performance to native VIA, showing the minimum latency of 10.5$\mu$sec and the peak bandwidth of 814Mbps on Giganet's cLAN. We have verified the functional compatibility with the existing Sockets API by porting FTP (File Transfer Protocol) and RPC (Remote Procedure Call) applications over the* SOVIA *layer. Compared to the Giganet's 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.*

## 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 [15, 5]. To enhance the communication 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 TCP/IP protocol suite is reported to fail in delivering raw-hardware performance to the end user, due to protocol overhead, context switching overhead, and data copying overhead between the user and 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 [19, 14, 16, 6].

The Virtual Interface Architecture (VIA) [7], promoted by Compaq, Intel, and Microsoft, is an industry effort to standardize user-level communication architectures. The VIA specification defines a software interface for fully-protected, user-level access to a network hardware. It 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, Fujitsu's Synfinity, and Compaq's ServerNet-II adapters.

The VIA specification provides a set of standard application programming interface (API) in the form of a user-level library called VIPL (VI Provider Library). 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 [1]. 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.

One of the possible candidates that can be used for general communication interface is the Berkeley Sockets API [12]. 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. In this paper, we design and implement a user-level Sockets layer over VIA, named SOVIA (Sockets Over VIA). Our goal is to support 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 VIA and related work. In section 3 and 4, we investigate several design issues for improving the performance and compatibility of the SOVIA layer. Section 5 presents experimental results of microbenchmarks and two real applications (FTP and 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 Virtual Interface Architecture with four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The VIA provides each consumer process with a protected, directly-accessible interface to a network hardware called Virtual Interface (VI). 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.

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 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.

The completion of data transfer can be discovered 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.

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 not needed any more for communication, it should be explicitly deregistered, whereupon the pages are released and made available for swapping out.

Several VIA implementations are available for Linux platforms. M-VIA (Modular VIA) [3] emulates the VIA specification by software for legacy Fast Ethernet and Gigabit Ethernet adapters. Berkeley VIA [4] implementation supports the VIA specification on Myrinet [2] by modifying its firmware. Finally, Giganet Inc. (now Emulex Corp.) has developed a proprietary, VIA-aware NIC called cLAN [9].

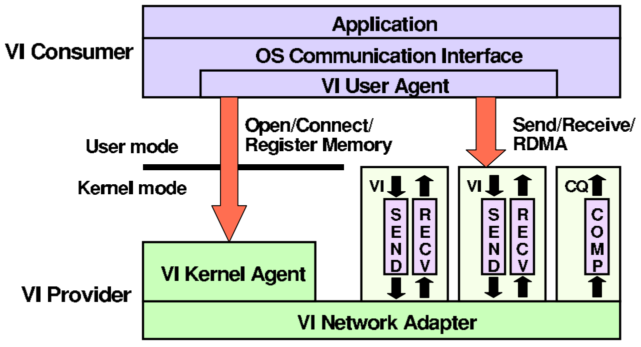


**Figure 1. The organization of the Virtual Interface Architecture**

### 2.2. Related Work

There can be several different approaches to support Sockets API, as illustrated in figure 2. Figure 2(a) shows the traditional communication architecture, in which the Sockets layer is located on top of TCP and UDP protocol stacks.

A simple way to support Sockets API on VIA is to insert an adaptation layer between IP and VI Kernel Agent, as shown in figure 2(b). This is the approach taken by the LANE (LAN Emulation) driver [8] supplied by Giganet for its cLAN adapters. As 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 on the connection-oriented VIA, and applications 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). Itoh et al. have recently proposed VIsocket [10], which provides Sockets functionality below the STREAMS module in Solaris. However, the design and performance details of VIsocket have not been published yet.

We note that the both approaches, shown in figure 2(b) and 2(c), still require context switching and data copying between the user and kernel space to send or receive data, as the support for Sockets API is implemented inside the kernel. Unlike these approaches, we build the SOVIA layer entirely at 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 [17] was the first attempt to support Sockets API over a lightweight user-level protocol, Active

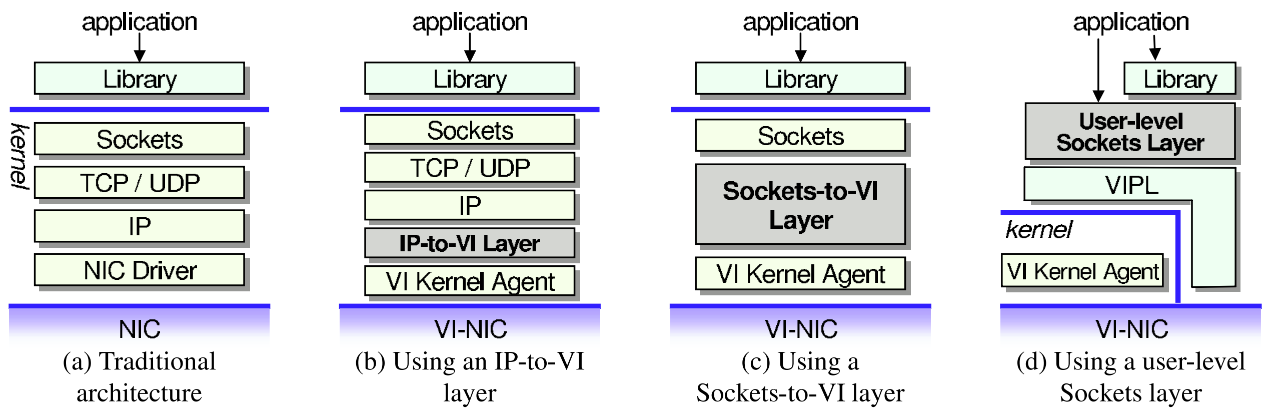


**Figure 2. Design alternatives for supporting Sockets API on Virtual Interface Architecture**

Messages (AM). 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.

The WSDP (Windows Sockets Direct Path) [13] technology developed by Microsoft also falls into this category. 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 Windows Sockets Provider below the Winsock library. For Unix-flavored systems, it is very difficult to emulate Sockets API completely at user-level, because Sockets-related data structures are kept inside the kernel and may be shared with child processes. In spite of that, 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 the application developers. To our best knowledge, SOVIA is the first implementation of a user-level Sockets layer over VIA on Unix/Linux platforms.

## 3. Performance Issues

The SOVIA layer should be lightweight so that the low-latency and high-bandwidth characteristics of VIA can be delivered to user applications. In this section, we first investigate several design issues directly related to the performance of the SOVIA layer.

### 3.1. Minimizing Latency

**Satisfying the pre-posting constraint.** The VIA requires that the receiver should pre-post a descriptor to the receive queue (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 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 receiver, with which the sender guarantees 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 the 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, the sender is allowed to transmit a DATA packet which carries real data. Normally, the receiver does not answer with the ACK until the user application at the receiving node calls `recv()`. Therefore, the receiver always knows the target buffer 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 DATA packets are sent to the receiver immediately after the sender receives the ACK for the previous data transfer. In this case, the DATA packet may arrive before the application calls `recv()` on the destination node, hence the receiver is required to buffer the incoming data temporarily. 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.

**Single-threading vs. Multi-threading.** In the traditional communication architecture, incoming messages 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 a queue and post a new one for each incoming message 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, and then wake up the application thread if it has been suspended on `recv()`.

Unfortunately, the Giganet's cLAN does not support the notify functions as yet. We can still emulate the role of notify functions by creating a dedicated handler thread manually which monitors a completion queue (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. We find, however, the synchronization cost between these two threads is expensive in Linux, sometimes up to tens of microseconds. Considering that the latency of native VIA is less than 10*μsec* on cLAN (cf. section 5.2), the high thread synchronization cost is the main source that increases latency.

Therefore, 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 messages are handled by the application thread when it calls communication-involved 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), it is possible to overlap the communication with the computation to some extent.

**Memory registration vs. copying.** As the buffers used by the receiver can be pre-registered, 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 is simply copied into the 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 simply use a hybrid approach, where data is 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 its size becomes larger than 2KB.

## 3.2. Maximizing Bandwidth

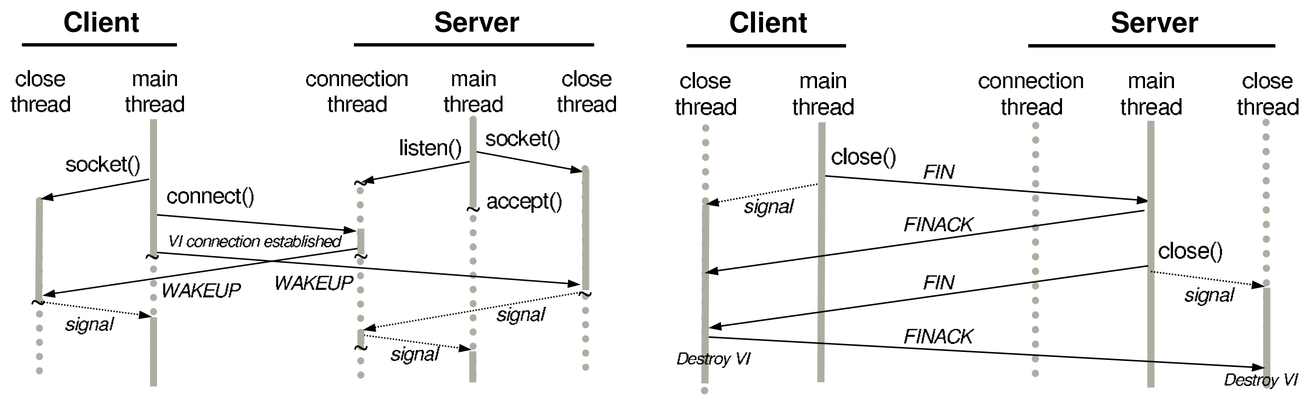
**Flow control.** Under the synchronization scheme described in section 3.1, the sender should wait for an ACK before requesting the next data transfer. The ACK packet informs that the receiver has pre-posted a descriptor to the corresponding RQ and is ready to receive another DATA. As a result, there 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* [18], which allows the sender to transmit multiple packets before it stops and waits for an acknowledgment. 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 acknowledgment. Initially, the receiver pre-posts $w$ descriptors to RQ. Whenever the sender transmits a DATA, it decreases $w$ by one meaning that one of the pre-posted descriptors on the receiving end has been 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 is delivered to the sender acknowledging one of the previous DATA packets.

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

SOVIA also takes advantage of delayed acknowledgments 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 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 acknowledgments are piggybacked with the DATA. We utilize the 32-bit *Immediate Data* field of the descriptor to record the packet type and the number of delayed acknowledgments.

**Combining small messages.** Another useful feature of TCP is the *Nagle algorithm* [18] enabled by default. The algorithm requires that when a TCP connection has outstanding data that has not yet been acknowledged, small messages cannot be sent until the outstanding data is acknowl-



(a) Establishing a connection (b) Closing a connection

**Figure 3. Connection management in SOVIA**

edged or until TCP can send a full-sized message. The Nagle algorithm is 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 one.

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). So far, a small message less than 2KB is copied into the buffer and then sent if the window size permits. However from now on, such a small message is appended into the buffer and the sender starts a timer which expires after, say, 100*msec*. Any other small messages requested within the expiration of the timer are also combined into the buffer. The data stored in the buffer is transmitted to the network either (1) when the timer expires, (2) when there is no enough room in the buffer for the new data transfer request, (3) when the requested message size is larger than 2KB, or (4) when the application calls `recv()` or `close()`. The maximum size that can be combined is 32KB, which is the message chunk size of SOVIA. For the messages larger than 2KB, it is too expensive to copy data, hence the current buffer is flushed and then the new message is transferred in a normal way.

## 4. Compatibility Issues

Now we discuss how the SOVIA layer supports the connection model used in Sockets API and enhances the portability for existing Sockets-based applications. Also, we mention the problems associated with `fork()` system call, which are the inherent limitation of any user-level Sockets implementation.

### 4.1. Connection Management

**Closing a connection.** SOVIA uses five types of packets; DATA, ACK, WAKEUP, FIN, and FINACK. To close a connection completely, both ends should agree on it by exchanging a FIN and a FINACK packet. A FIN packet is sent to the peer when the application issues `close()`, and an FINACK acknowledges the receipt of the FIN. However, the single-threaded implementation of SOVIA poses a problem when a connection is closed. The Sockets semantics requires the application return from `close()` after sending a FIN packet to the peer. Once the application executes the last `close()`, it does not call any Sockets API and there is no chance to handle the incoming FINACK and/or FIN packets from the peer, which are necessary to receive before destroying associated data structures.

To solve this problem, SOVIA initially creates a *close thread* and asks it to take care of the incoming messages, as shown in figure 3(a). Whenever a new connection is established, a WAKEUP packet is exchanged between peers. The WAKEUP packet is used to inform the peer about the sender's socket descriptor, IP address, and port number. If the close thread receives the WAKEUP, it stops handling messages and is suspended until the last connection is closed (cf. figure 3(b)). Therefore, the close thread is only activated when the number of open connections becomes zero. In this way, the presence of the close thread does not affect the application's performance.

**Establishing a new connection.** Similarly, we use a special *connection thread* to accept a new connection. The Sockets API has a connection model that a server process specifies a willingness to accept incoming connections with `listen()`, and then a connection is accepted with `accept()`. If a remote process (i.e. a client) wants to initiate a connection, it calls `connect()`. The VIA has a

```
// find symbols in libc during initialization
dlhandle = dlopen("libc.so.6", RTLD_LAZY);
sockops->socket = dlsym(dlhandle, "socket");
sockops->bind = dlsym(dlhandle, "bind");
....
dlclose(dlhandle);
....

int socket (int domain, int type, int proto)
{
  if (type == SOCK_VIA)
    return sov_socket (domain, type, proto);
  else
    return sockops->socket (domain, type, proto);
}
```

```
int sov_socket (int domain, int type, int proto)
{
  int s = open("/dev/null", O_RDWR);
  sockdes[s] = sov_newsock(domain, type, proto);
  return s;
}

int write (int s, const void *buf, size_t size)
{
  if (sockdes[s])
    return sov_write (s, buf, size);
  else
    return __libc_write (s, buf, size);
}
```

**Figure 4. Example codes to override the existing interface**

slightly different connection model especially on the server side; `VipConnectWait()` is used for the server to look for incoming connection requests, and then `VipConnectAccept()` is called to accept the connection request and 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, it is the Sockets semantics that the client should be able to return from `connect()` successfully even though the server, after calling `listen()`, does not reach `accept()` yet. Therefore, we can not directly implement `accept()` with `VipConnectWait()` and `VipConnectAccept()`. This implies that another thread should be running to accept a VI connection behind the application thread.

SOVIA creates a connection thread each time the application issues `listen()` on a port, as illustrated in figure 3(a). 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.

## 4.2. Enhancing the portability

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;

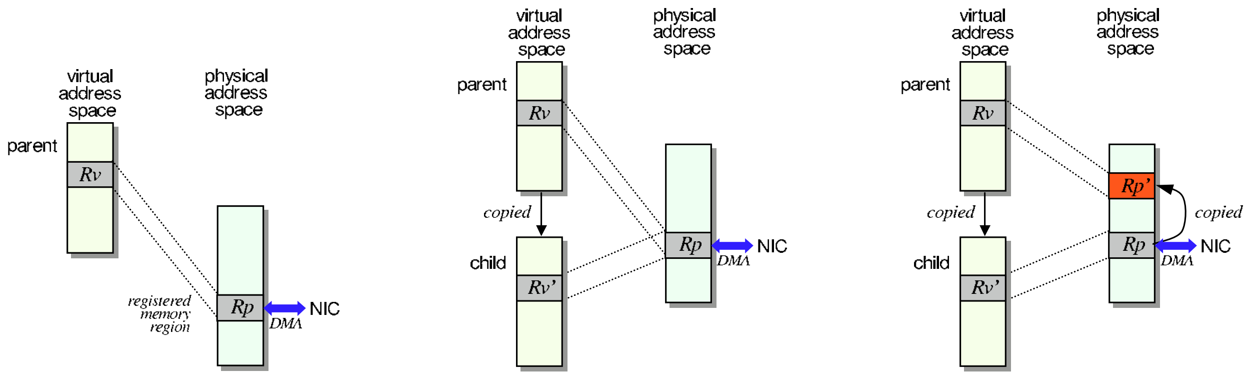
```
#define socket        sov_socket
#define close         sov_close
```

Such a source-level modification using a pre-processor has a couple of drawbacks. First, normal TCP/UDP sockets can not coexist with SOVIA, as all the `socket()` calls 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. System calls, such as `read()`, `write()` and `close()`, may operate on socket descriptors, but, for example, `close()` can not be replaced with `sov_close()`.

Instead, we use a dynamic approach where our version of Sockets API is selected at run-time based on a socket descriptor. GNU's C library (`libc`) defines system call interfaces as weak symbols so that they can be overridden in user code. We provide wrapper functions for file system interfaces and statically link them with our Sockets-based applications to intercept system calls. Adding these wrapper functions significantly improves the source-level portability of the SOVIA layer.

First, 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 to obtain a file descriptor from the kernel and return the same number as a new socket descriptor. Afterwards, any system call on the new socket descriptor is intercepted and switched to the SOVIA layer by the wrapper functions, as shown in figure 4 briefly. For the system calls operating on normal file descriptors, it is necessary for the wrapper functions to locate the addresses of original interfaces in `libc`. If the original (strong) symbols are also exported by `libc`, which is the case for `write()` (`__libc_write()`) or `close()` (`__close()`), we can call those symbols directly in the wrapper functions. In other case, we can use the `dlsym()` function to obtain the address where a particular symbol in a dynamic library is loaded.

Note that messages can be also sent or received by means of the standard I/O library, as the following code fragments show;



(a) before `fork()` (b) after `fork()` (c) when the parent writes to $R_v$

**Figure 5. Copy-on-write problem**

```
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");
```

As the standard I/O library internally calls `__write()` and `__read()` in `libc`, we need to override these functions too. Once it is done, any access through the standard I/O library can be switched to the SOVIA layer transparently.

### 4.3. Problems with `fork()`

**Copy-on-write problem.** An FTP (File Transfer Protocol) 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 `fork()` system call[1]. We elaborate upon the situation in figure 5.

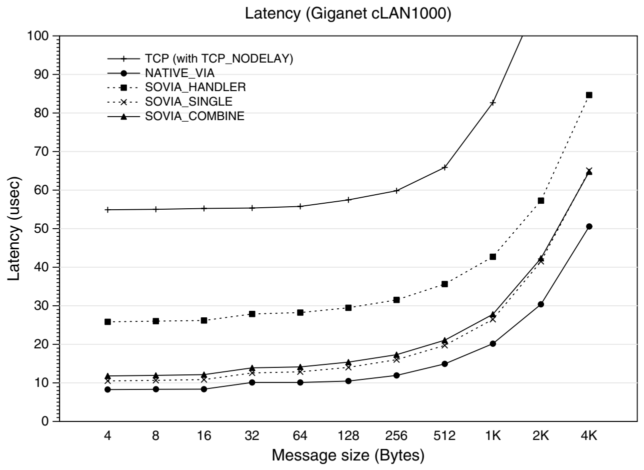
As described in section 2.1, a memory region $R_v$ needs to be registered before it is used for any communication. During the registration, the VI Kernel Agent converts the virtual page addresses for $R_v$ into physical addresses and pins the corresponding region $R_p$ in the physical memory. $R_p$ is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

[1]For example, Giganet's cLAN driver version 1.1.1 used in the 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.

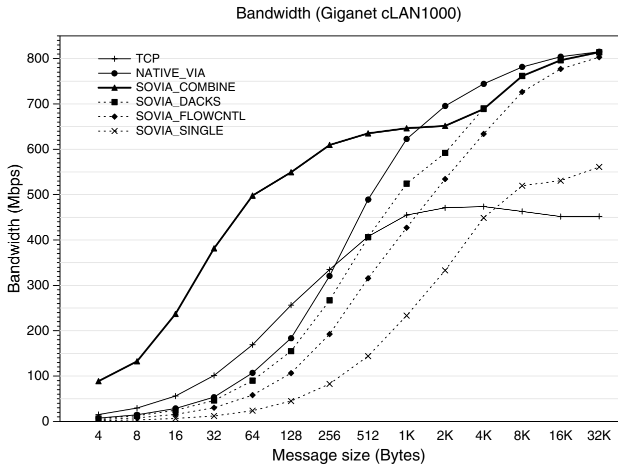
If the process forks a child, the Linux kernel optimizes the virtual memory system by copying the data structures describing virtual memory and by sharing the actual physical pages, as shown in figure 5(b). This optimization is called a *copy-on-write*. Copy-on-write is introduced to increase speed while decreasing memory usage, but 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 and the parent gets new physical pages $R'_p$ which are copies of $R_p$. As a result, 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 child process without causing the copy-on-write problem. We find that the cLAN driver registers 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.

**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 two processes at the VIPL level. Although the VIPL for cLAN is designed to be multithread-safe, it is not sufficient to protect resources in a shared-memory segment from simultaneous access by multiple processes, because Linux currently does



(a) Latency



(b) Bandwidth

**Figure 6. The latency and bandwidth on Giganet's cLAN**

not support process-shared mutexes and conditions [11].

To minimize modifications in the VIPL, 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 opens a new VI connection with the client. The server and 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 through `inetd`.

## 5. Experimental Results

### 5.1. Evaluation Methodology

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-500MHz microprocessor with 512KB of L2 cache and 256MB of main memory. Two cLAN1000 network adapters are attached to a 32-bit 33MHz 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 bandwidth. The latency is measured by a half of round-trip time in a ping-pong test. To measure the (unidirectional) bandwidth, the sender issues a long rapid sequence of `send()`'s for a given time and waits until an acknowledgment message arrives from the receiver. 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 (File Transfer Protocol) and RPC (Remote Procedure Call) applications over the SOVIA layer in order to verify the functional compatibility with the existing Sockets API. The performance details of these applications will be discussed in section 5.3 and 5.4, respectively.

### 5.2. Microbenchmarks

Figure 6(a) compares the latency of SOVIA with that of TCP and native VIA on Giganet's cLAN. First of all, we note that native VIA outperforms TCP as expected; native VIA shows the latency of 8.5$\mu sec$ for 4-byte messages, while TCP shows 55$\mu sec$ 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, the thread synchronization cost is responsible for the gap between SOVIA_HANDLER and SOVIA_SINGLE, which increases latency more than 15$\mu sec$. Figure 6(a) also shows the changes in latency when we try to combine small messages together (labeled SOVIA_COMBINE). Combining small messages increases the latency of SOVIA by 1 – 2 $\mu sec$ to manage a software timer. However, note that 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` option. Overall, we can reduce the latency of SOVIA layer as low as 10.5$\mu sec$ for cLAN on our platforms, adding only 2$\mu sec$ of overhead to the native VIA's latency.

Figure 6(b) illustrates the measured bandwidths of TCP, native VIA and SOVIA. Again, native VIA shows higher

**Table 1. The performance of file transfers using FTP**

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

bandwidth than TCP when the message size is larger than 256 bytes[2]; the bandwidth of TCP for 32KB messages is limited to about 450Mbps, attaining only 55% of native VIA's performance (815Mbps).

In figure 6(b), SOVIA_SINGLE represents the bandwidth obtained by the single-threaded implementation with conditional sender-side buffering, which has minimized latency in figure 6(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 delayed acknowledgments to SOVIA_SINGLE. We can see that the bandwidth of SOVIA_DACKS is notably improved compared to SOVIA_SINGLE. Especially, if the message size is greater than or equal to 16KB, SOVIA_DACKS 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 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 2KB.

## 5.3. FTP Performance

We have measured the performance of file transfers between two nodes by modifying the FTP server (`linux-ftpd-0.16`) and client (`netkit-ftp-0.16`) contained in Linux NetKit 0.16. Table 1 compares the measured bandwidth and 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 ramdisks.

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 6(b). The measured bandwidth is, however, slightly lower than the expected one, showing 573Mbps and 532Mbps for 19MB and 145MB files, respectively. This is because the actual data transfer is bounded by the file system performance, which has the bandwidth of 538Mbps – 611Mbps for local ramdisk-to-ramdisk copy of the same files.

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

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 815Mbps, FTP applications result in bandwidths less than 300Mbps 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. Like a local function call, the calling arguments are passed to the 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 generated automatically by specifying what their interfaces should be. RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports. This transport independence of RPC, together with the fact that it is implemented as a user-level library, make it possible for RPC applications to benefit from the 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 a base transport by specifying "`via`" when it calls `clnt_create()` function and there is no other changes visible to the application developers.

For the experiment, we have used `sunrpc` implementation in `glibc-2.1.3`, which is available on our platforms by default. Figure 7 compares the average elapsed time for a single RPC for various argument sizes ranging from 0 to 4KB. An argument is passed to a remote procedure as a character string, and the body of the remote procedure is

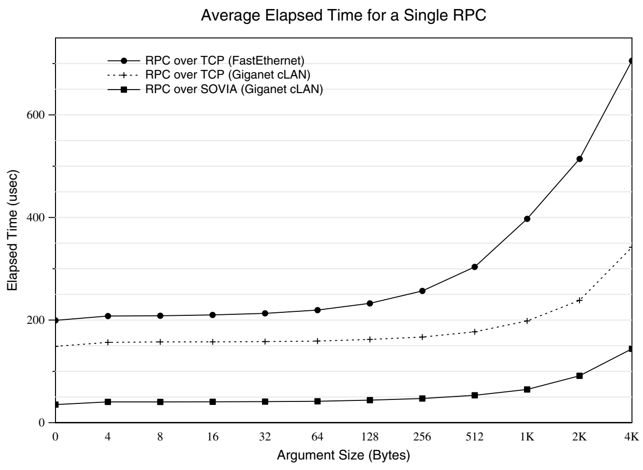


**Figure 7. Average elapsed time to call an empty remote procedure**

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, messages are exchanged between the server and client containing an RPC header, 44 bytes for request and 28 bytes for response.

Unlike the FTP, RPC is a latency-sensitive application. Our measurement result shows that calling an empty remote procedure with the null argument takes about 200$\mu sec$ on Fast Ethernet and 149$\mu sec$ on cLAN, when the traditional TCP is used as a transport. On the contrary, making an RPC over the SOVIA layer takes only 35$\mu sec$ 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

In this paper, we design and implement SOVIA, a user-level Sockets layer over VIA. Because adding a new software layer introduces an overhead inevitably, special attention has been paid to make the SOVIA layer lightweight, while retaining the portable Sockets semantics.

SOVIA shows the minimum latency of 10.5$\mu sec$ and the peak bandwidth of 814Mbps on Giganet's cLAN, which is comparable to the native VIA's performance. The functional compatibility with the existing Sockets API is verified by porting the FTP and RPC applications over the SOVIA layer. Compared to the Giganet's LANE driver, SOVIA easily doubles the file transfer bandwidth in FTP and reduces the latency of calling an empty remote procedure by 77% in RPC applications.

We expect application programs written in Sockets API can seamlessly take advantage of the performance of VIA through the SOVIA layer. We plan to port a user-level parallel file system and a software distributed shared-memory (DSM) system over the SOVIA layer in the near future to demonstrate this.

## References

[1] M. Baker, editor. *Cluster Computing White Paper (Version 2.0)*. IEEE Task Force on Cluster Computing, Dec. 2000.

[2] N. J. Boden et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, Feb. 1995.

[3] P. Bozeman and B. Saphir. A Modular High Performance Inplementation of the Virtual Interface Architecture. In *Proc. Extreme Linux Conference*, 1999.

[4] P. Buonadonna, A. Geweke, and D. E. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proc. SC '98*, 1998.

[5] R. Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, Inc., 1999.

[6] B. Chun, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. *IEEE Micro*, 31(1):53–63, Jan. 1998.

[7] Compaq Computer Corp., Intel Corp. and Microsoft Corp. Virtual Interface Architecture Specification Draft Revision 1.0. `http://www.viarch.org/`, Dec. 1997.

[8] Giganet Inc. cLAN for Linux: Software User's Guide, 2001.

[9] Giganet Inc. cLAN Hardware Installation Guide, 2001.

[10] M. Itoh, T. Ishizaki, and M. Kishimoto. Accelerated Socket Communications in System Area Networks. In *Proc. Cluster 2000*, pages 357–358, 2000.

[11] X. Leroy. LinuxThreads Frequently Asked Questions. `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[12] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996.

[13] Microsoft Corp. Winsock Direct Concepts. `http://www.microsoft.com/windows2000/en/datacenter/help/wsd_concepts.htm`, 2001.

[14] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. Supercomputing*, 1995.

[15] G. Pfister. *In Search of Clusters, Second Edition*. Prentice Hall, Inc., 1998.

[16] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking: a Myrinet Experience. Technical Report 97-22, LIP-ENS Lyons, France, Jul. 1997.

[17] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proc. USENIX*, 1997.

[18] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Publishing Co., 1994.

[19] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. Symposium on Operating System Principles*, pages 303–316, 1995.