J Supercomput (2007) 39: 205–226
DOI 10.1007/s11227-007-0109-5

Design issues and performance comparisons in supporting the sockets interface over user-level communication architecture

Jae-Wan Jang · Jin-Soo Kim

Published online: 17 February 2007

Abstract Since user-level communication (ULC) architecture provides only primitive operations for application programmers, there have been several researches to build a portable and standard communication interface, such as sockets, on top of ULC architecture. Basically there are three different approaches to supporting the sockets interface over ULC architecture: LAN emulation, a user-level sockets, and a kernel-level sockets. The primary objective of this paper is to compare these approaches in terms of their design, implementation, and performance.

We have developed and implemented a kernel-level sockets layer over ULC architecture, since there is currently no available implementation. We also present different design and implementation decisions on data receiving, data sending, connection management, etc. in the three approaches. Through the performance comparison, we show that LAN emulation approach exhibits the worst performance both in latency and bandwidth. Our experiments also show that a user-level sockets is useful for latency-sensitive applications and a kernel-level sockets is effective for applications which require high bandwidth and full compatibility with the legacy sockets interface.

Keywords Interconnection network · Sockets interface · User-level communication · VIA · cLAN

J.-W. Jang (✉) · J.-S. Kim
Division of Computer Science, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 305-701, South Korea
e-mail: jwjang@camars.kaist.ac.kr

J.-S. Kim
e-mail: jinsoo@cs.kaist.ac.kr

1 Introduction

As cluster systems are constructed with low-cost and high-performance commodity-off-the-shelf (COTS) components, they exhibit relatively high performance/cost ratio compared to the other proprietary computer architectures. Additionally, the loosely-coupled organization of nodes in cluster systems improves scalability, maintainability, and upgradeability. These advantages make cluster systems one of the most common computer architectures for high-performance computing. In the latest survey results of the world's fastest TOP500 supercomputer sites, 304 systems out of 500 are cluster systems and the number of cluster systems in the whole list is increasing continuously [21].

Since every node in cluster systems shares the same network infrastructure, network performance readily becomes a bottleneck of the entire cluster system. In order to improve network performance, high-performance system area networks (SANs), such as Gigabit Ethernet, Myrinet, Quadrics, and InfiniBand Architecture, are usually employed in cluster systems. Although SANs deliver high-speed network hardware, it is known that current cluster systems can not fully utilize their raw speed due to software overhead. Interrupt handling, network protocol processing, context switching, data copying between the user and the kernel space, and error checking in the operating system layer contribute to the software overhead. Therefore, it is essential to devise an efficient communication architecture which minimizes the software overhead in order to improve the performance of the entire cluster system.

User-level communication (ULC) architectures attempt to enhance communication performance by removing the operating system from the critical communication path. ULC architectures aim to relieve the aforementioned overhead by performing most of the tasks involved in communication only in the user space. When the ULC concept was first introduced, many proprietary architectures realizing the concept were developed. After those early days, industries eager to standardize ULC architectures and bring about the emergence of the Virtual Interface Architecture (VIA) [6, 8] and InfiniBand Architecture (IBA).

It is well known that primitives of ULC architectures are considered to be at too low a level for general network application programming [3]. Those primitives do not provide high-level functionalities such as buffer management or transport-level functionality. In order to exploit advantages of ULC architectures efficiently, application programmers should make those functions by themselves. Thus, an abstraction layer over ULC architecture supporting high-level operations is highly necessary. Many researchers have endeavored to build such layers over various ULC architectures [2, 11, 15, 16, 18, 20].

One of possible candidates that can be used over ULC architecture is the Berkeley sockets API [17] considering its widespread use and acceptance in distributed environments. The sockets API is the de facto standard for network programming and provides a means for developing applications independent of the network hardware.

Basically, there are three different approaches to supporting the sockets interface on top of ULC architecture. The simple and transparent approach is to insert an adaptation layer between IP layer and network hardware and to make the adaptation layer emulate LAN. In this approach, socket applications are executed through TCP/IP

layer on top of the adaptation layer. Another approach is to make a sockets layer using primitives of ULC architecture either in user-level or in kernel-level bypassing the TCP/IP stack.

The difference in the design of these approaches exhibits the different characteristics in the performance. This urges us to investigate the pros and cons of each approach both quantitatively and qualitatively. Since we have LAN emulation driver for VIA-aware NIC from Emulex corp. [10] and a user-level sockets layer over VIA from our previous work [15], we need a kernel-level sockets layer over VIA for the comparison on the same VIA platform. Thus, the first goal of this paper is to design and implement a kernel-level sockets layer over VIA (KSOVIA). The second goal is to evaluate the impact of different design and implementation decisions of these approaches on the sockets performance. Comparison among the approaches in view of design decisions enables us to quantitatively analyze the various reasons for the difference in performance. For example, if we compare a kernel-level sockets layer with a user-level sockets layer, we can measure the overhead of the kernel-level implementation. Also the comparison between LAN emulation layer and a kernel-level sockets layer will reveal the overhead of TCP/IP protocol processing.

The major contributions of this paper can be summarized as follows.

- We classify three different approaches to supporting the sockets interface over ULC architecture.
- We design and implement KSOVIA, a kernel-level sockets layer over VIA, in order to make the comparison with the other approaches.
- We examine design and implementation issues in supporting the sockets interface over ULC architecture, with paying attention to design differences and trade-offs among the three approaches.
- We measure the latency and the bandwidth of the three approaches and native VIA on the same platform to understand their relative performance.
- We quantitatively analyze the individual costs associated with communication (e.g., socket implementation overhead, TCP/IP overhead, overheads from the use of kernel services, etc.) by comparing the performance of the three approaches.

The rest of the paper is organized as follows. The next section presents the background of our work briefly, and describes the three different approaches to supporting the sockets interface over ULC architecture. Section 3 compares design and implementation issues of the different approaches. Section 4 analyzes the experimental results comparing the performance of each approach. Section 5 presents related work. Finally, we conclude and present future work in Sect. 6.

2 Background

2.1 Virtual Interface Architecture

The organization of VIA is briefly illustrated in Fig. 1. VIA consists of four basic components: Virtual Interfaces (VIs), VI Provider, VI Consumer, and Completion Queues (CQs).

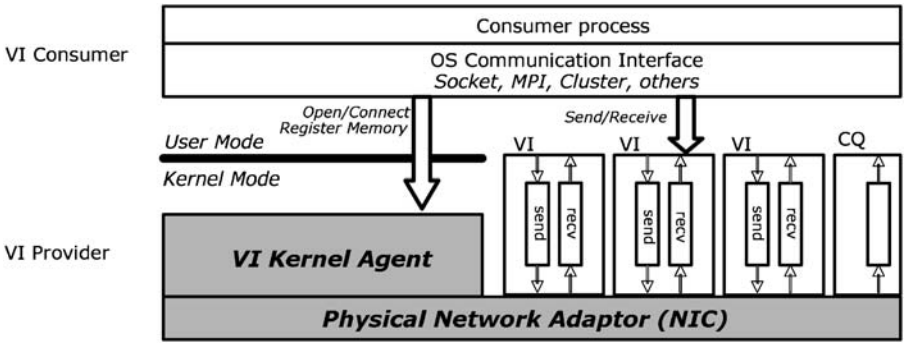


Fig. 1 The organization of the Virtual Interface Architecture

Virtual Interfaces. VIA provides a consumer process with a protected, directly-accessible interface to a network hardware called Virtual Interface (VI). A VI has a pair of work queues: a send queue and a receive queue. Each VI represents a communication endpoint and every send or receive operation is performed through VI.

VI Provider. VI Provider consists of a physical network adapter and the VI Kernel Agent. The VI Kernel Agent is a kernel module which is responsible for registering communication memory and setting up and shutting down VIs.

VI Consumer. VI Consumer represents the user of a VI in Fig. 1 which can be optionally attached with some communication interfaces.

Completion queues. Completion queues store information used to notify VI consumers of the completion of send or receive operations. Using completion queues, a consumer process can check the completion of several send or receive operations at once.

Several VIA implementations are available for Linux platform. M-VIA [4] emulates the VIA specification by software for legacy Fast Ethernet and Gigabit Ethernet NICs. Berkeley VIA [5], SVIA [22], and MyVIA [7] support the VIA specification on Myrinet by modifying its firmware. Finally, Emulex Corp.(former Giganet Inc.) has developed a proprietary, VIA-aware NIC called cLAN [9]. In this paper, we investigate various issues related to the sockets support on cLAN because it is one of the most stable VIA implementations.

2.2 Approaches to supporting the sockets interface over VIA

Basically there are three different approaches to supporting the sockets interface over VIA as illustrated in Fig. 2b–d. Figure 2a shows the traditional communication architecture, in which the sockets layer resides on top of the TCP and IP protocol stack.

2.2.1 LAN emulation layer

The simple and transparent approach to supporting sockets over VIA is to insert an adaptation layer between IP layer and network hardware as depicted in Fig. 2b. The IP layer considers the adaptation layer as a kind of Ethernet device, and any IP-based

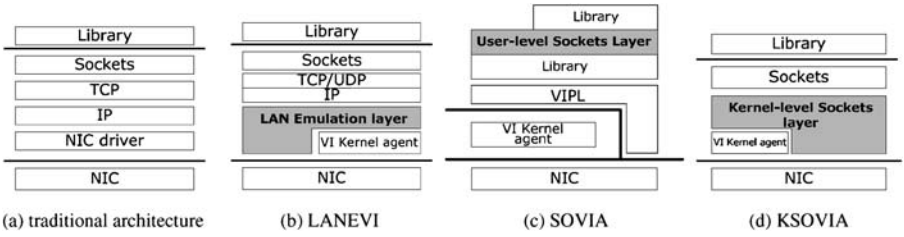


Fig. 2 Supporting the sockets interface over VIA

applications including socket applications can be executed on cLAN network without any modifications using this layer.

cLAN provides LANEVI (LAN emulation on VI) driver which realizes this approach. LANEVI consists of kernel-level interface to NIC and LAN emulation routine. After LANEVI driver is loaded into the kernel, a network interface named *clan0* is created and applications can access cLAN network through *clan0* whose role is similar to *eth0* for typical Ethernet device.

2.2.2 A user-level sockets layer over VIA

The second approach is to make a sockets layer using primitives of VIA in the user space, as shown in Fig. 2c. Since this layer is located in the user space, it provides the socket interface without sacrificing the performance of the underlying VIA layer. However, it is not easy to preserve sockets semantics perfectly because socket operations have to be implemented out of the kernel. In the typical Linux system, `fork()` and `exec()` is commonly used to create a new process. In the user-level sockets layer, however, calling `exec()` destroys the current process image where the sockets-related data structures are kept. Hence, a newly created process can not use the existing socket connection. This problem makes network servers such as *inetd* hard to operate on the user-level sockets layer. In order to solve this problem, complex mechanisms are necessary.

Our previous work, SOVIA [15] is an example of this approach. SOVIA implements every socket operation in a user-level library, and any socket applications linked to this library are able to send or receive data without the help of the kernel.

2.2.3 A kernel-level sockets layer over VIA

The final approach is to move the sockets layer from the user space into the kernel again, as shown in Fig. 2d. The kernel-level sockets layer exploits the sockets interface in the kernel but bypasses the network protocol stack. Also, this layer is free from the problems in the user-level sockets layer, since it maintains sockets-related data structures in the kernel. On the contrary, every socket operation should go though the kernel accompanying context switching and data copying overhead.

Although there exist implementations for the previous two approaches, to the best of our knowledge, a kernel-level sockets layer over VIA has currently no available implementation. Thus, we have designed and implemented a kernel-level sockets layer

over VIA (KSOVIA). Whereas SOVIA is developed as a user-level library, KSOVIA is developed as a kernel module which bypasses the TCP/IP protocol suite. For unbiased comparison, KSOVIA is designed similar to SOVIA as much as possible. Details of the design and implementation of KSOVIA will be provided in the next section.

3 A comparison of design and implementation issues

In this section, we compare design and implementation issues among LANEVI (a LAN emulation layer), SOVIA (a user-level sockets layer), and KSOVIA (a kernel-level sockets layer). The detailed information on LANEVI has not been published in the literature, and thus it is guessed from the source code. Although we briefly introduce the design and implementation of SOVIA, readers who want to know more on its internals are encouraged to refer to [15]. In this paper, we relatively elaborate upon the internal workings of KSOVIA which we have developed for the complete comparison among different approaches.

3.1 VIPL

cLAN provides primitive operations through VI Provider Library (VIPL). Since all the implementations we have compared are built on top of VIPL, we present basic operations of VIPL before we begin to compare the three approaches for clear and complete analysis.

Posting and reaping operations. Sending or receiving data using VIPL is performed through two distinct phases, namely the posting phase and the reaping phase. In the posting phase, a process posts a descriptor, which contains a request specification including data length and pointers to data buffers. The descriptor is posted either on a send queue (for data sending) or on a receive queue (for data receiving). When sending or receiving data finishes, NIC marks a DONE bit in the status field of the corresponding descriptor to indicate the completion of descriptor processing. Those completed descriptors are identified and then removed from the work queue in the reaping phase. The completed descriptors can be detected either by polling the status field of the head descriptor in the work queue, or by using a blocking call in which a calling process is signaled upon the completion of the request. The reaping phase is unavoidable because if the work queue is filled with the completed descriptors, the process can not post further descriptors to send or receive data. Note that a process need not perform the reaping phase right after the posting phase every time because multiple completed descriptors can be removed from the work queue in a single reaping phase.

Completion queue. A completion queue (CQ) allows a process to coalesce notifications of the completed descriptors from multiple work queues in a single location. When a VI is created, each work queue of the VI can be associated with a CQ. Once this association is established, notifications of the completed requests are automatically directed to the CQ.

Memory registration. All the memory regions used for communication should be registered to NIC before NIC uses them. NIC reads and writes data directly from and to the registered memory regions without the help of the virtual memory. When a memory region is no longer needed for communication, it should be explicitly deregistered, whereupon the corresponding physical pages are released and made available for swapping out.

3.2 Data receiving

A typical scenario to receive data with VIPL is as follows. A user process posts a descriptor, which contains a pointer to the user buffer, into the receive queue. When a packet arrives, NIC stores payload data to the user buffer specified in the descriptor at the head of the receive queue, and marks a DONE bit in the descriptor. And then the user process reaps the completed descriptor and accesses the received data in the user buffer. Basically, all the approaches follow the above steps to receive data, but details are slightly different.

LANEVI. LANEVI performs the following steps in order to receive data. First, LANEVI prepares a set of receive descriptors and temporary buffers when it is loaded into the kernel. The memory regions where the descriptors and buffers reside are registered to NIC in advance so that NIC can access them directly. Second, when a packet arrives, an interrupt is raised which enables LANEVI to reap the descriptor. And then LANEVI copies the received data specified in the reaped descriptor to a socket buffer (sk_buff). Third, the socket buffer is passed on to the upper layer and processed by the TCP/IP protocol stack. Fourth, when the user requests data through `recv()`, data in the socket buffer is copied to the user buffer. Finally, for the next incoming packets, LANEVI posts other descriptors into the receive queue. Note that there are two copies during these steps; one is between temporary buffers and socket buffers, and the other between socket buffers and user buffers.

SOVIA. While LANEVI supports the sockets interface with the help of the TCP/IP protocol stack, SOVIA implements it directly at the user space using VIPL. SOVIA prepares a set of descriptors and temporary buffers when it is initialized. Whenever an application calls `socket()`, SOVIA creates a new VI and connects its receive queue to a single completion queue. When a packet arrives, NIC transfers payload data to the temporary buffers specified in the descriptor, and SOVIA needs to check the completion queue in order to see if a packet arrives. There are two methods to check the completion queue. One is to use separate user-level handler thread which checks the completion queue all the time. If the handler thread detects any completed descriptors (i.e., new packet arrivals), the handler thread reaps them from the receive queue. The other is to emulate the handler thread without having a separate thread. In this case, every sockets-related function such as `send()` or `recv()` checks the completion queue to see if a new packet arrives. When it is detected, the application thread reaps the completed descriptors. While the first method is easy to implement, it has relatively high synchronization overhead between the application thread and the handler thread. Thus, SOVIA chooses the second method by default. After these steps, if the user process calls `recv()`, SOVIA copies the data from temporary buffers to user buffers.

KSOVIA. Initial steps to receive data in KSOVIA are similar to those in LANEVI. First, KSOVIA prepares a pool of descriptors and temporary buffers when it is loaded into the kernel. KSOVIA also creates a completion queue used in the handler routine which is responsible for checking new packet arrivals. Note that KSOVIA maintains only a single completion queue in the entire system, while SOVIA creates a completion queue for each process who wants to use sockets interface. Second, if a user process creates a socket, KSOVIA creates a VI and associates its receive queue with the completion queue. Third, when NIC receives a packet, it transfers payload data to the temporary buffer specified in the descriptor and raises an interrupt which schedules the handler routine in the bottom half of the Linux kernel. Fourth, when the kernel allows the handler routine to run, the handler routine checks completed descriptors and moves them to the queue of the corresponding socket. Finally, when the user process calls `recv()`, KSOVIA copies data in the temporary buffer to the user buffer. Used descriptors are recycled and KSOVIA posts them to the receive queue again for the next incoming packets.

We have applied an optimization to the receiving operation of KSOVIA in order to increase the bandwidth. A whole receiving operation of KSOVIA can be divided into two distinct phases. One is a system-initiated phase and the other is a user-initiated phase. The system-initiated phase begins with an interrupt notifying new data arrivals and ends with moving the completed descriptors to the corresponding sockets. The user-initiated phase begins with a request of the user through `recv()` and ends with copying data from temporary buffers in the kernel to user buffers. While the system-initiated phase is responsible for checking completed descriptors, we insert a routine which also checks the completed descriptors at the end of the user-initiated phase. This optimization increases the number of chances where the arrived data is extracted from the completion queue and shows higher bandwidth especially when there is burst communication traffic of small data.

One of the key issues in the data receiving is how to deal with VIA's pre-posting constraint. VIA requires that the receiver pre-posts a descriptor to the receive queue before a packet arrives. Unless a descriptor is posted in advance, the transferred packet is dropped at the receiving side without any notice. As we have seen previously, all the approaches rely on temporary buffers to satisfy the pre-posting constraint. As long as we use temporary buffers, at least one data copy from temporary buffers to user buffers is unavoidable. Notice that LANEVI experiences one more copy due to the intermediate socket buffers as described previously.

3.3 Data sending

LANEVI. Similar to the preparation step of the data receiving, LANEVI prepares a set of send descriptors and temporary buffers, and registers the corresponding memory regions in advance. When a user process calls `send()`, data in the user buffer is copied into the kernel space. And then the data is encapsulated in the socket buffers with TCP and IP headers. Those socket buffers are passed to LANEVI. Since the memory regions of socket buffers are not registered to NIC, LANEVI copies the contents of socket buffers to temporary buffers. After copying, LANEVI posts the descriptors associated with those temporary buffers into the send queue of the VI. After NIC actually sends the data, LANEVI reaps send descriptors from the send queue.

SOVIA. SOVIA provides two sending modes because NIC can send data only in the registered memory region. One is to copy user data to temporary buffers registered in advance and the other is to register memory region of the user data on the fly. Since the second mode does not copy data, it is faster than the first mode for relatively large data size. In the second mode, however, user process should not modify the data before the entire send operation finishes. In any case, SOVIA posts a descriptor specifying the pointer to the prepared memory region and reaps the posted descriptors after NIC sends the data completely.

KSOVIA. The send operation of KSOVIA is simple and straightforward. If a user process calls `send()`, data in the user buffer is copied to temporary buffers pre-registered in the kernel. And then the descriptor associated with the temporary buffers is posted into the send queue. Similar to SOVIA, KSOVIA reaps it after NIC sends the data completely. All these steps are occurred together at every `send()` request.

We have also applied an optimization to the sending operation of KSOVIA to increase the bandwidth. The sending operation of KSOVIA can be divided into two phases; the posting phase and the reaping phase. During the experiments, we observe that the reaping phase takes relatively long time waiting for the completion of the descriptor just posted. Since KSOVIA exploits several send descriptors, KSOVIA need not reap the send descriptor right after posting it. Thus, KSOVIA postpones the reaping task until there are no available send descriptors to post. This reduces the time for processing the send operation. This optimization can be used in SOVIA if SOVIA copies data using temporary buffers and uses multiple descriptors for send operations. During our experiments, however, we used a version of SOVIA where this optimization is not applied.

3.4 Connection management

LANEVI. The LAN emulation layer need not take care of connections. Instead, the connection is managed by the upper layer such as TCP. Since VIA adopts a connection-oriented model, however, communication between two nodes can not be done without making a VI connection between them. In the cLAN network, when a new node joins, it is notified to every other node. If LANEVI discovers the node insertion event, it creates a VI and automatically establishes a VI connection with a new node even before any data transfer request is passed from the IP layer. When a user process creates a socket and tries to open a connection to the other process in the remote site, special TCP packets are exchanged through the VI connection which is already established in advance. Thus, a socket communication between two user processes through LANEVI requires two connections. One is between two VIs at LANEVI and the other is between two TCPs. This mechanism is inefficient, but it is inevitable to emulate connectionless LAN over the connection-oriented VIA.

SOVIA. Contrary to LANEVI, SOVIA makes a VI connection only after an application tries to connect to the remote site. SOVIA maintains two user-level POSIX threads for connection management. One is the close thread which processes incoming packets after partial close of the connection. The other is the connection thread spawned as a result of `listen()`. Due to the slight semantic differences in connection models between sockets and VIA, the connection thread is necessary to accept an incoming VI connection request behind the application thread [15].

Table 1 A comparison of the design and implementation issues among LANEVI, SOVIA, and KSOVIA

Comparison items	LANEVI	SOVIA	KSOVIA
The number of data copies during data receiving	Two (one from temporary buffers to socket buffers, the other from socket buffers to user buffers)	One (from temporary buffers to user buffers)	One (from temporary buffers to user buffers)
The number of data copies during data sending	Two (one from user buffers to socket buffers, the other from socket buffers to temporary buffers)	Zero or one (no data copy or possibly one from user buffers to temporary buffers)	One (from user buffers to temporary buffers)
System call overhead	Exist	Exist when blocking VIPL APIs are used	Exist
TCP/IP protocol processing overhead	Exist	Not exist	Not exist
Connection management	Handled by the kernel (automatically establish a connection with a new node even if it is not used)	Use two POSIX threads (the close thread and the connection thread)	Use one kernel thread for connection establishment
Flow control	Resort to TCP	Use a customized credit-based flow control	Same as in SOVIA

KSOVIA. Similar to SOVIA, KSOVIA also makes a VI connection only after a user process tries to connect to the remote site. However, KSOVIA does not need a close thread since the kernel is able to receive any incoming packets without resort to the user process. Instead, KSOVIA employs a kernel thread to deal with VI connection requests and replies. As in SOVIA, this kernel thread is created after `listen()` is invoked.

3.5 Flow control

LANEVI has no concern for flow control because flow is mostly managed by the upper TCP layer. Although TCP has many complex flow control mechanisms, some of these mechanisms, such as congestion control, are not necessary in the reliable networks such as cluster interconnects. Thus, it is required to devise a lightweight flow control mechanism for SOVIA and KSOVIA. Note that since flow control affects the socket performance substantially, we adopt the same flow control mechanism of SOVIA to KSOVIA for the unbiased comparison of their performance.

The flow control mechanism used in SOVIA mainly focuses on increasing the bandwidth. SOVIA supports a credit-based flow control which is similar to the TCP's sliding window protocol. It has a notion of window size ω, which denotes the maximum number of data packets the sender is allowed to transmit without waiting for an acknowledgment. When a socket is created, ω receive descriptors are pre-posted in the receive queue. Whenever the sender transmits a data packet, it decreases ω by one to denote that one of the receive descriptors has been consumed. If ω reaches zero, the sender stops to transmit data packets until ω becomes a positive number. Window size ω is increased by one if an acknowledgment is delivered to the sender. In order to enhance the bandwidth further, acknowledgments can be delayed and piggybacked to data packets.

We summarize the design and implementation differences among LANEVI, SOVIA, and KSOVIA in Table 1.

4 Performance comparison

We perform various tests to quantitatively analyze the impact of different design and implementation decisions on the performance.

4.1 Experimental environments

We have measured the performance of LANEVI, SOVIA, and KSOVIA with the Linux kernel 2.4.18 and cLAN driver version 2.0.1. The hardware platform used is two Linux workstations, each consisting of 1.6 GHz Intel Pentium 4 processor, 512 KB L2 cache, and 768 MB of main memory. Two cLAN1000 network adapters are attached to a 32-bit 33 MHz PCI slot of each workstation, and we connect them without any intermediate switches.

We carry out microbenchmarks which measure the one-way latency and the unidirectional bandwidth. The one-way latency is measured by a half of the round-trip time from several tens of thousands of ping-pong tests. The unidirectional bandwidth is obtained by measuring the average time spent for sending 100,000 packets repeatedly. In addition, we use FTP server and client programs to verify the functionality and to evaluate the performance of real socket applications.

Communication mechanisms evaluated in this paper are presented in Table 2. All the measurement results are taken from the results of the socket-based microbenchmarks except VIPL and VIPL_POLL. The results of VIPL and VIPL_POLL are derived from the microbenchmarks written in VIPL APIs and they are considered as the baseline for the performance of the other results. SOVIA is tested using both non-blocking APIs (SOVIA_POLL) and blocking APIs (SOVIA). Note that a process using blocking APIs is blocked until the end of each I/O operation in the kernel space, while a process using non-blocking APIs polls the completion of each I/O operation in the user space. We use a SOVIA version which uses temporary buffers in both sending and receiving data. KSOVIA does not use non-blocking APIs because polling is not allowed inside the kernel.

Figure 3 illustrates various stacks involved in each communication mechanisms. All the communication with LANEVI, KSOVIA, VIPL, and SOVIA pass through the

Table 2 Evaluated communication mechanisms

Communication mechanisms	Description
VIPL_POLL	Use pure VIPL with non-blocking APIs in the user space
VIPL	Use pure VIPL with blocking APIs in the user space
LANEVI	Use the traditional TCP/IP using the LANEVI driver
SOVIA_POLL	Use a user-level sockets layer over VIA with non-blocking APIs
SOVIA	Use a user-level sockets layer over VIA with blocking APIs
KSOVIA	Use a kernel-level sockets layer over VIA

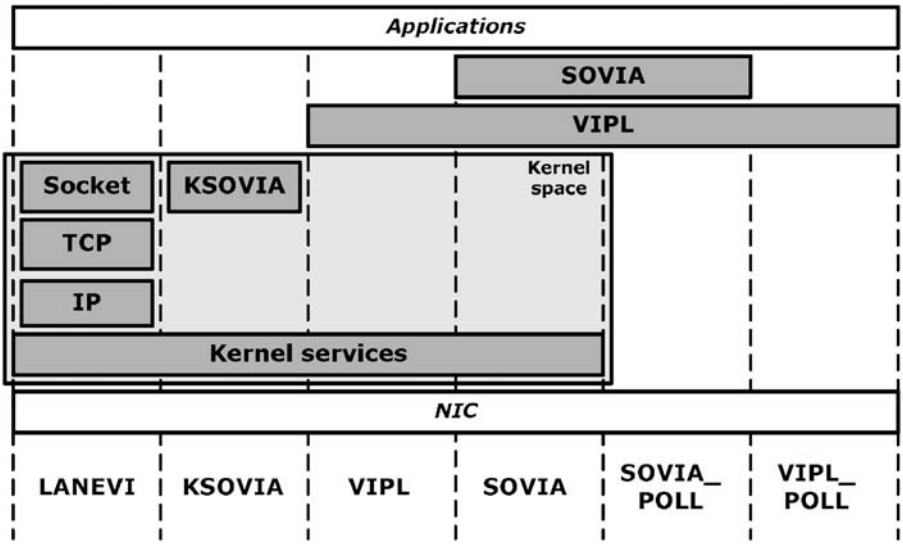


Fig. 3 Various stacks involved in each communication mechanism

kernel space, which accompanies various overheads resulted from context switching, interrupt handling, and process scheduling. We denote those kernel overheads as *kernel services* collectively. Even if SOVIA and VIPL are user-level communication mechanisms, they rely on kernel services since a process goes into the kernel when it is blocked in `send()` or `recv()`.

4.2 One-way latency

Figure 4 shows the one-way latency when the data size varies from 4 bytes to 32 bytes. The results of VIPL_POLL and SOVIA_POLL, which use non-blocking APIs, show the shortest latency for all data sizes. Unlike the other communication mechanisms, VIPL_POLL and SOVIA_POLL are not blocked in the kernel to wait for the completion of sending or receiving operation, but continuously poll it in the user space. When a process is blocked in the kernel, the use of kernel services is unavoidable in order to wake up and reschedule the blocked process, and this overhead

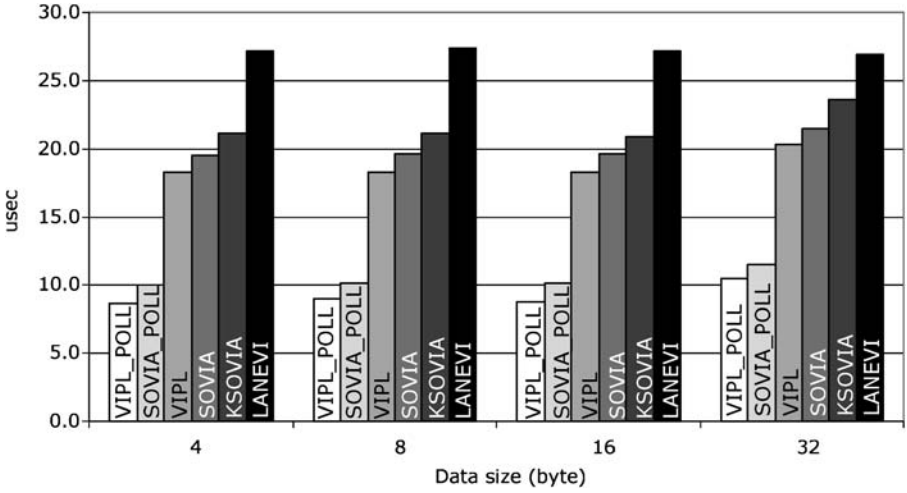


Fig. 4 One-way latency

increases the latency. In our experiments, the results which use non-blocking APIs show about the half of the latency of the ones using blocking APIs (9.2 µsec versus 18.8 µsec for VIPL and 10.5 µsec versus 20.0 µsec for SOVIA). In general, however, network applications can not use polling with non-blocking APIs because it wastes most of CPU times on busy waiting. Thus, the use of polling requires careful design consideration depending on the workload.

From Fig. 4, we can see that SOVIA shows the constantly higher latency than VIPL by about 1.3 µsec. It is mainly due to the increased complexity in the SOVIA layer to support the sockets interface as shown in Fig. 3.

Figure 4 also illustrates that the latency of KSOVIA is higher than that of SOVIA by about 1.7 µsec. As we have designed KSOVIA to behave similar to SOVIA as much as possible, we believe that the kernel-level implementation adds the latency. Compared to SOVIA, KSOVIA uses the bottom half of the Linux kernel in receiving data and multiplexes received data to the corresponding socket. These additional tasks from the kernel-level implementation require extra time.

LANEVI has the additional TCP/IP overhead compared to KSOVIA, exhibiting the worst latency in Fig. 4. From the latency difference between LANEVI and KSOVIA, we can conclude that TCP/IP protocol processing increases the latency of LANEVI by about 5.5 µsec.

We summarize various overheads from the latency results in Table 3.

4.3 Unidirectional bandwidth

4.3.1 Bandwidth comparison among several approaches

Figure 5 shows the unidirectional bandwidth of each communication mechanism studied in this paper. First, we can see that LANEVI shows the higher bandwidth than any other mechanisms when the data size is less than 2 Kbytes. This is because TCP combines small data together before a packet is transmitted, which is known as

Table 3 Various overheads in communication path

Overheads	Calculated from	Time (μsec)
Socket implementation	(SOVIA–VIPL)	1.3
Use of kernel services	(VIPL–VIPL_POLL)	9.6
Kernel-level implementation	(KSOVIA–SOVIA)	1.7
TCP/IP protocol processing	(LANEVI–KSOVIA)	5.5

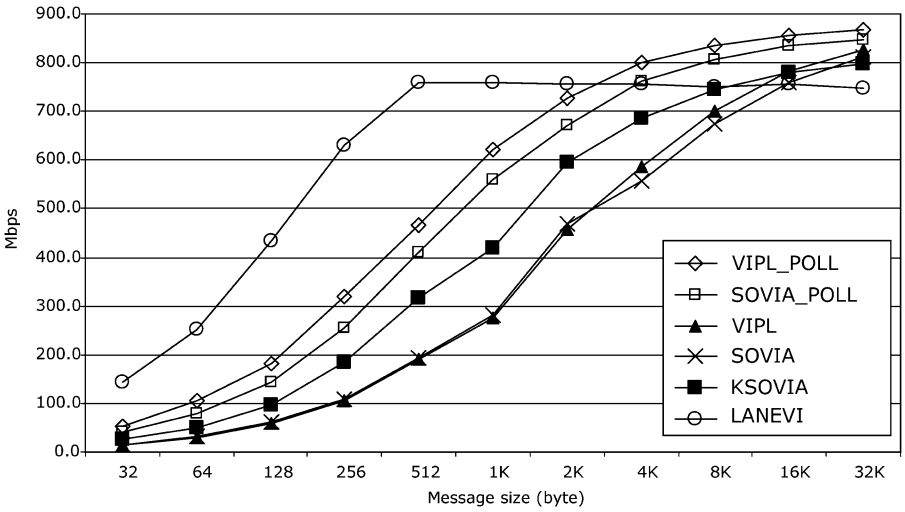


Fig. 5 Unidirectional bandwidth

Nagle algorithm. The bandwidth of LANEVI is saturated at 760 Mbps for 512-byte data size and becomes the smallest when the data size is larger than 16 Kbytes. When we enable similar algorithm in SOVIA and KSOVIA, we achieve better bandwidth compared to LANEVI as shown in Fig. 6. In this result, SOVIA and KSOVIA try to combine small data within 32-Kbytes buffer during up to 100 ms.

As SOVIA is made on top of VIPL, the performance of SOVIA and SOVIA_POLL are bound to that of VIPL and VIPL_POLL, respectively. The minor difference between SOVIA and VIPL is due to the protocol maintenance overhead in SOVIA, which has also increased the latency of SOVIA.

The result of KSOVIA resides between the results of SOVIA_POLL and SOVIA, while the latency of KSOVIA is slightly higher than that of SOVIA. This is because of the difference in data transfer mechanisms between SOVIA and KSOVIA. SOVIA handles the communication without having a separate thread. Thus, every socket-related routine has to do additional tasks as well as its own task. For instance, during the unidirectional bandwidth test, each `send()` in SOVIA has to check new data arrivals even though there is nothing in the completion queue. These redundant tasks increase the processing time slightly. However, this redundancy is inevitable to make SOVIA a single-threaded application in order to avoid thread synchronization over-

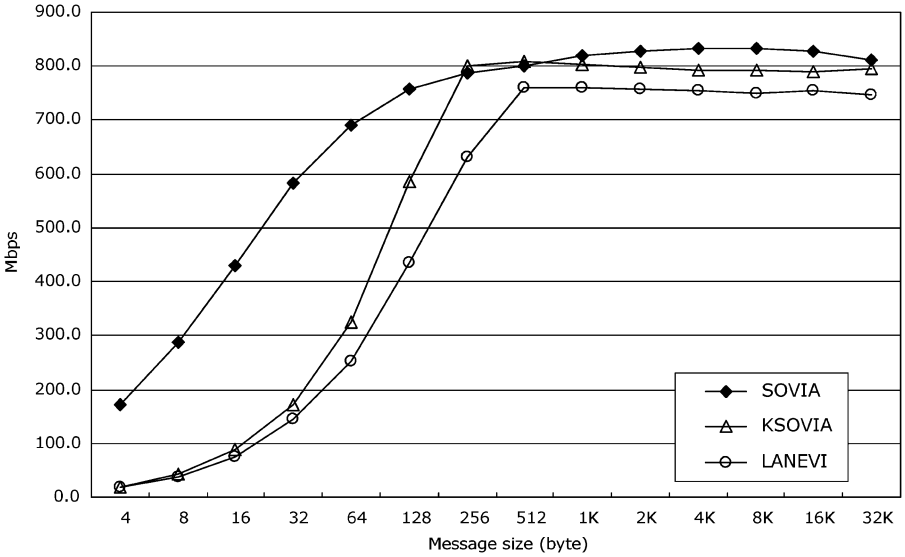


Fig. 6 Unidirectional bandwidth when SOVIA and KSOVIA combine small data

head [15]. On the contrary, KSOVIA does not require such a complex design allowing it to achieve higher bandwidth. In KSOVIA, the handler routine is located in the kernel and runs only if there is something to process in the completion queue, which is notified by an interrupt.

4.3.2 Bandwidth optimizations in KSOVIA

In order to overcome disadvantages that KSOVIA resides in the kernel, we apply two optimizations in receiving and sending operation, respectively. Details of those optimization techniques are already presented in Sects. 3.2 and 3.3. We illustrate the impact of these optimizations in Fig. 7.

The improvement of the receiving optimization decreases as the data size grows. When the size of data moving through the network link increases, pending interrupts are decreased and the chance to process them at the end of `recv()` also diminishes. Thus, the relative gain over the non-optimized version is dropped and this optimization imposes more overhead for large data.

On the contrary, the improvement of the sending optimization outperforms the non-optimized version continuously. The peak bandwidth reaches 848.2 Mbps for 32 Kbytes data which is even greater than that of SOVIA_POLL. Although KSOVIA uses blocking APIs, KSOVIA exhibits the best performance, which shows that the use of the multiple send descriptors is more useful than the use of non-blocking APIs.

4.4 Throughput analysis

In this subsection, we present a throughput model of KSOVIA and SOVIA for theoretical analysis of the previous bandwidth results. Basically, when the sender trans-

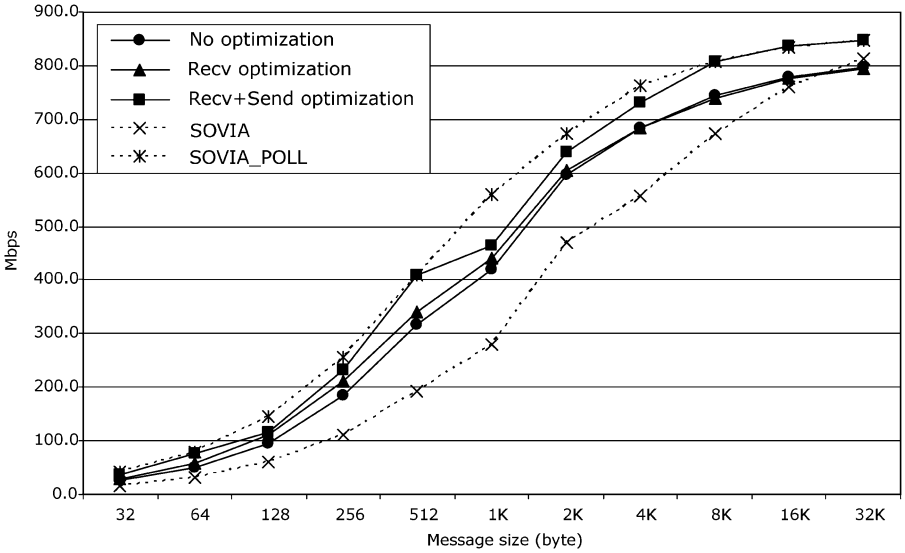


Fig. 7 Effect of the bandwidth optimizations

mits N packets of length L, the total throughput is given by

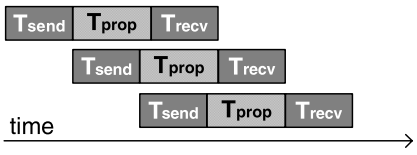
$$BW(L) = \frac{N \cdot L}{T_{\text{elapsed}}}, \tag{1}$$

where T_{elapsed} represents the elapsed time to transmit all the packets.

Our model is not based on the loss rate contrary to most throughput models of TCP because LAN is reliable and packet drop hardly occurs. Rather, it is characterized mostly by the sending rate as shown in Fig. 8. An individual horizontal bar in Fig. 8 illustrates the elapsed time for transmitting a single packet from the sender to the receiver, where T_{send}, T_{prop}, and T_{recv} denote the time spent for send operation in the sender, the propagation time over the physical link, and the time for receive operation in the receiver, respectively. As the latency of $T_{\text{prop}} + T_{\text{recv}}$ can be hidden by the transmission of the subsequent packets, T_{send} effectively determines the overall throughput[1]. Hence, T_{elapsed} can be calculated as

$$T_{\text{elapsed}} = T_{\text{send}} \cdot (N - 1) + (T_{\text{send}} + T_{\text{prop}} + T_{\text{recv}}). \tag{2}$$

Fig. 8 Timing model



[1] Although both KSOVIA and SOVIA employ a credit-based flow control mechanism, our measurement results show that the packet transmission is not delayed due to the lack of available window size, since the acknowledgments are delivered to the sender sufficiently fast during unidirectional bandwidth tests.

If N is sufficiently large, Eq. 2 becomes

$$T_{\text{elapsed}} \approx T_{\text{send}} \cdot N. \tag{3}$$

Equations 1 and 3 make

$$BW(L) = \frac{L}{T_{\text{send}}}. \tag{4}$$

In KSOVIA, T_{send} can be represented as the summation of several components as follows.

$$T_{\text{send,KSOVIA}} = T_{\text{syscall}} + T_{\text{memcpy}} + T_{\text{post}} + T_{\text{DMA}} + T_{\text{overhead}}. \tag{5}$$

In Eq. 5, T_{syscall}, T_{memcpy}, and T_{post} denote the time to issue a send system call, the time to copy user-space data into the kernel space, and the time to post a send descriptor, respectively. T_{DMA} is the time to DMA data in the kernel memory into the SDRAM of cLAN NIC. Since we use 32-bit 33 MHz PCI bus, the maximum DMA bandwidth is limited to 132 MB/s. Finally, T_{overhead} describes the additional overhead that constantly exists in the communication path. It includes, for example, the time to execute user applications and internal protocol processing codes. Note that in Eq. 5, T_{syscall}, T_{post}, and T_{overhead} are independent of the packet size, while T_{memcpy} and T_{DMA} are directly proportional to the packet size.

On our evaluation platform, T_{syscall}, T_{post}, and T_{overhead} are found to be 1.2 μsec, 0.4 μsec, and 5.0 μsec, respectively. We model T_{memcpy} as

$$T_{\text{memcpy}} = \rho \cdot L + c, \tag{6}$$

where $\rho = 0.00182$ μsec/byte and $c = 0.08$ μsec, by interpolating actual measurement results using `memcpy()` on the same platform. If we let δ be the DMA bandwidth, T_{DMA} can be expressed as

$$T_{\text{DMA}} = L/\delta, \tag{7}$$

where δ = 132 MB/s in our platform.

Using Eqs. 4–7, the throughput model of KSOVIA is given by

$$BW_{\text{KSOVIA}}(L) = \frac{L}{\alpha \cdot L + \beta}, \tag{8}$$

where $\alpha = (\rho + 1/\delta) = 0.0094$ μsec/byte and $\beta = T_{\text{syscall}} + T_{\text{post}} + T_{\text{overhead}} + c = 6.68$ μsec.

Figure 9 compares the throughput predicted by the performance model in Eq. 8 with the actual values obtained from our experiments shown in Fig. 5. It can be seen that our model successfully approximates the actual KSOVIA throughput.

For SOVIA, T_{send} does not include any system call overhead, hence the throughput is given by

$$BW_{\text{SOVIA}}(L) = \frac{L}{T_{\text{send,SOVIA}}} \tag{9}$$

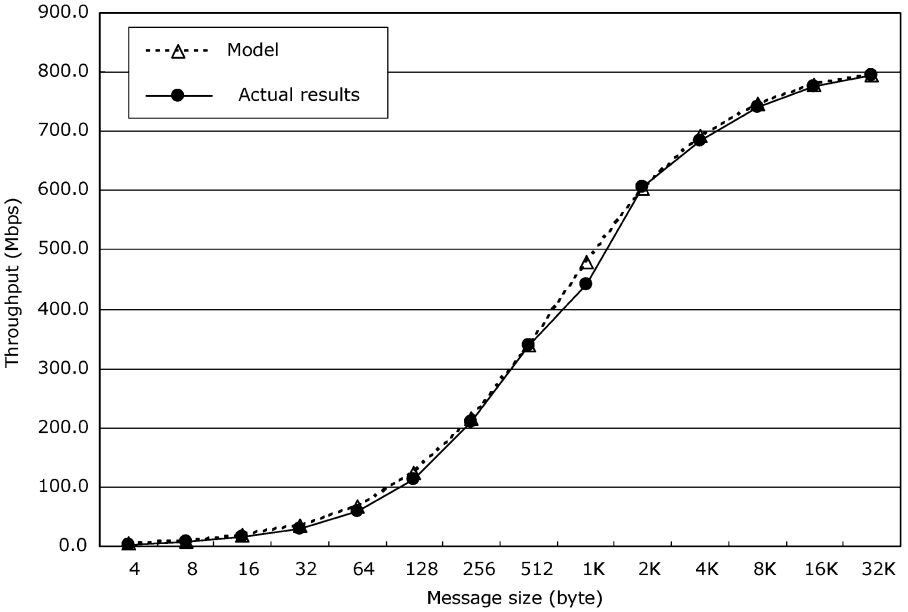


Fig. 9 KSOVIA throughput comparison of actual values vs. modeled values

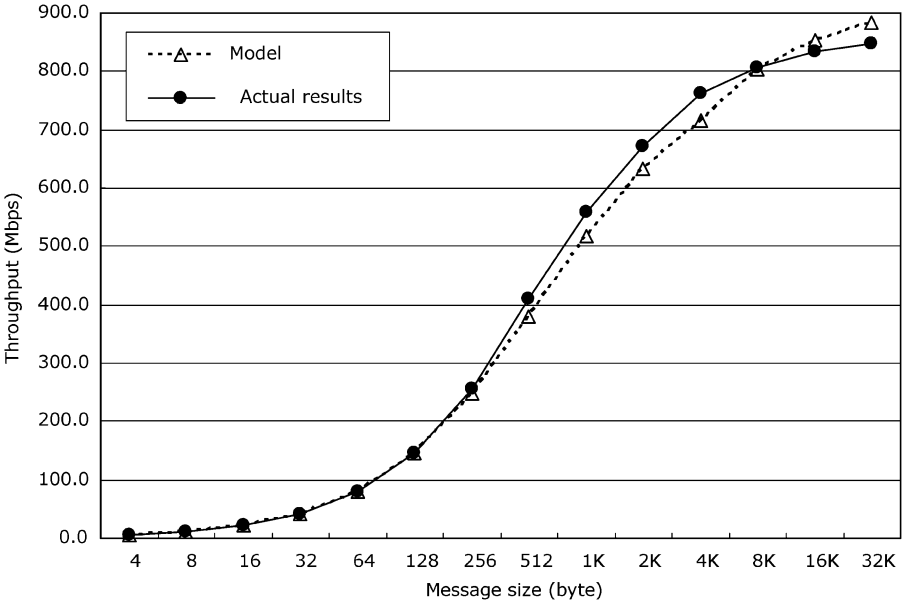


Fig. 10 SOVIA throughput comparison of actual values vs. modeled values

$$= \frac{L}{T_{\text{memcpy}} + T_{\text{post}} + T_{\text{DMA}} + T_{\text{overhead}}} \tag{10}$$

$$= \frac{L}{\alpha \cdot L + \beta'} \tag{11}$$

with the value of α is the same as that of KSOVIA and $\beta' = 5.48$ μsec. Figure 10 compares the predicted and the actual throughput for SOVIA. We can observe that our simple throughput model approximates the actual throughput reasonably well both for KSOVIA and for SOVIA.

4.5 FTP performance

We run an FTP server (linux-ftpd-0.17) and client (netkit-ftp-0.17) over LANEVI, SOVIA, and KSOVIA, in order to verify their functional validity and to measure the performance of real socket applications. Two files are transferred from the server to the client; one is small (12.75 MB) and the other is rather big (161.1 MB). We have arranged that all the files reside in RAM disks to avoid the influence of the disk subsystems.

Table 4 summarizes the performance of file transfers. Since the bandwidth of local copy using *cp* command is large enough exceeding 1 Gbps, FTP performance is not bounded by the RAM disks performance. Table 4 shows that LANEVI achieves 91.2% (for small file) and 89.6% (for big file) of the bandwidth of KSOVIA. These results are analogous to those obtained from unidirectional bandwidth microbenchmarks. As described in the previous section, KSOVIA shows a little higher bandwidth than SOVIA owing to the bandwidth optimizations described in Sects. 3.2 and 3.3.

Table 4 FTP bandwidth

	File A 13,374,187 bytes		File B 168,919,040 bytes	
Fast Ethernet	1.14 sec	89.51 Mbps	14.4 sec	89.50 Mbps
LANEVI	0.137 sec	744.80 Mbps	1.74 sec	740.60 Mbps
SOVIA	0.134 sec	778.91 Mbps	1.8 sec	762.57 Mbps
KSOVIA	0.125 sec	816.30 Mbps	1.56 sec	826.12 Mbps
Local copy	0.063 sec	1619.63 Mbps	1.255 sec	1026.89 Mbps

5 Related work

The most popular communication interfaces used in cluster environments are MPI (Message Passing Interface) and Berkeley sockets interface [17]. Since sockets interface provides more general communication interface, many researches have been performed to support the sockets API on VIA platform. VIsocket [14] is similar to KSOVIA in that it provides socket functionality below the STREAMS module in Solaris by collapsing internal TCP/IP layer. However, the design and performance details of VIsocket have not been published yet.

InfiniBand Architecture (IBA) [13] has been standardized by the industry to develop the next-generation high-performance interconnection network. As IBA adopts many features from VIA, its software layer is very similar to VIPL. Currently, there are many ongoing research projects to build new convenient layers on top of IBA, such as IPoIB [12] and SDP [12]. IPoIB provides the standardized IP encapsulation over IBA fabrics, whose role is identical to that of LANEVI in a conceptual viewpoint. Sockets Direct Protocol (SDP) defines a standard wire protocol to support stream sockets over IBA bypassing the traditional TCP/IP protocol stack. SDP is essentially a kernel-level sockets layer over IBA and its purpose is the same as that of KSOVIA. Even though some implementations of SDP are available [1, 12, 19], there is few publicly available documentation of internal implementation and comparison among several design choices we focus on. Due to many similarities between VIA and IBA, we believe that the results obtained in this paper are also useful in developing such layers as IPoIB and SDP.

6 Concluding remarks

In this paper, we compare the three different approaches to supporting the sockets interface over VIA. LANEVI emulates LAN over VIA, while SOVIA and KSOVIA stand for a user-level sockets layer over VIA and a kernel-level sockets layer over VIA, respectively. Since there is currently no reference implementation for kernel-level sockets layers, we have designed and implemented KSOVIA which is our own kernel-level sockets layer over VIA.

Through the performance measurements in our platform, we find out the followings. First, as we expected, TCP/IP protocol processing is the main source of significant overheads; about 20.1% of the whole latency of LANEVI is consumed in processing TCP/IP protocol. Second, the use of kernel services incurs more serious overhead than TCP/IP protocol processing. About 47.8% of SOVIA latency and about 35.1% of LANEVI latency are spent on the use of kernel services. Third, a single-threaded design of SOVIA results in relatively lower bandwidth compared to KSOVIA. Extra tasks due to the single-threaded design prevent SOVIA from achieving a higher bandwidth. Lastly, the use of multiple send descriptors in KSOVIA is noticeably effective, even outperforming than SOVIA with non-blocking APIs.

We conclude that there is no reason to use LANEVI for socket-based applications. It shows the worst performance in terms of latency and bandwidth as well as the additional overhead in connection management. On the other hand, SOVIA and KSOVIA have their own pros and cons. Even though SOVIA presents a little lower bandwidth than KSOVIA, it is useful when applications are latency sensitive and do not use `exec()` or `fork()`. KSOVIA can be used effectively for applications which require high bandwidth or the full compatibility with the legacy socket-based applications.

We are going to observe the behavior of various cluster and Grid applications to see the impact of the different approaches on the real-world applications. Additionally, we plan to extend our work onto InfiniBand Architecture by investigating such layers as IPoIB and SDP.

Acknowledgements This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031).

References

1. Balaji P, Narravula S, Vaidyanathan K, Krishnamoorthy S, Wu J, Panda DK (2004) Sockets direct protocol over InfiniBand in clusters: is it beneficial? In: Proceedings of the IEEE international symposium on performance analysis of systems and software, IEEE Computer Society Press, Los Alamitos, pp 28–35
2. Balaji P, Shivan P, Wyckoff P, Panda D (2002) High performance user level sockets over gigabit ethernet. In: Proceedings of the IEEE international conference on cluster computing, IEEE Computer Society Press, Los Alamitos, pp 179–186
3. Baker M (ed) (December 2000) Cluster computing white paper (Version 2.0). IEEE Task Force on Cluster Computing
4. Bozeman P, Saphir B (1999) A modular high performance implementation of the Virtual Interface Architecture. In: Proceedings of extreme linux conference
5. Buonadonna P, Geweke A, Culler D (1998) An implementation and analysis of the Virtual Interface Architecture. In: Proceedings of ACM/IEEE international conference on supercomputing (CDROM), IEEE Computer Society Press, Los Alamitos, pp 1–15
6. Cameron D, Regnier G (2002) The virtual interface architecture. Intel Press
7. Chen Y, Wang X, Jiao Z, Xie J, Du Z, Li S (2002) MyVIA: a design and implementation of the high performance Virtual Interface Architecture. In: Proceedings of the IEEE international conference on cluster computing, IEEE Computer Society Press, Los Alamitos, pp 160–167
8. Compaq Corporation, Intel Corporation, and Microsoft Corporation (1997) Virtual interface architecture specification, Version 1.0
9. Emulex Corporation-Products. http://www.emulex.com/products/legacy/vi/clan1000.html
10. Emulex Inc (2001) cLAN for Linux. Software User's Guide
11. Fischer M (2001) GMSOCKS—a direct sockets implementations on myrinet. In: Proceedings of the IEEE international conference on cluster computing, IEEE Computer Society Press, Los Alamitos, pp 204–211
12. InfiniBand Linux SourceForge Project: http://infiniband.sourceforge.net
13. InfiniBand Trade Association. http://www.infinibandta.org
14. Itoh M, Ishizaki T, Kishimoto M (2000) Accelerated socket communication in system area network. In: Proceedings of the IEEE international conference on cluster computing, IEEE Computer Society Press, Los Alamitos, pp 357–358
15. Kim J-S, Kim K, Jung S-I, Ha S. (2003) Design and implementation of a user-level sockets layer over Virtual Interface Architecture. Concurr Comput Pract Exp 15(7–8):727–749
16. Liu J, Wu J, Panda DK (2003) High performance RDMA-based MPI implementation over InfiniBand. In: Proceedings of the 17th annual international conference on supercomputing, ACM Press, New York, pp 295–304
17. McKusick MK, Bostic K, Karels MJ, Quarterman JS (1996) The design and implementation of the 4.4 BSD operating system. Addison-Wesley, Reading
18. MVICH—MPI for the Virtual Interface Architecture. http://old-www.nersc.gov/research/FTG/mvich
19. OpenIB.org. http://www.openib.org
20. Seifert F, Kohmann H SCI SOCKET—a fast socket implementation over SCI. White paper, Dophin Interconnect Solutions Inc
21. TOP500 Supercomputer sites. http://www.top500.org
22. Yu J-L, Lee M-S, Maeng S-R (2001) An efficient implementation of Virtual Interface Architecture using adaptive transfer mechanism on myrinet. In: Proceedings of international conference on parallel and distributed systems, IEEE Computer Society Press, Los Alamitos, pp 741–747

Jae-Wan Jang received his B.S. degree in computer engineering from Kyungpook National University in 2002, and M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2004, where he is a Ph.D. candidate. His research interests include distributed system, networked system, and virtual machines.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He was with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. Since 2002, he has been a faculty member in the department of electrical engineering and computer science at Korea Advanced Institute of Science and Technology (KAIST). Before joining KAIST, he was a senior member of research staff at Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002. His research interests include operating systems, embedded systems, cluster computing, and storage systems. He is a member of the IEEE and the ACM.