Supporting the Sockets Interface over User-Level Communication Architecture: Design Issues and Performance Comparisons

Jae-Wan Jang
jwjang@camars.kaist.ac.kr

Jin-Soo Kim
jinsoo@cs.kaist.ac.kr

Division of Computer Science
Korea Advanced Institute of Science and Technology (KAIST)

Abstract

Since user-level communication architecture (ULC) provides only primitive operations for application programmers, many high-level communication layers have been developed on top of ULC. One of such high-level communication layers is the sockets interfaces and it can be supported over ULC architectures in several ways. The primary objective of this paper is to identify design issues and tradeoffs among these different approaches, and to quantitatively analyze their performance to understand the various costs associated with the communication.

In this paper, we design and implement KSOVIA, a kernel-level sockets layer over VIA, and compare it with the existing approaches such as a user-level sockets layer over VIA and an IP emulation layer over VIA. Our measurement results show that using an IP emulation layer exhibits the worst performance in terms of latency and bandwidth and a user-level sockets layer is useful for latency-sensitive applications. KSOVIA is found to be effective for applications which require high bandwidth or the full compatibility with the sockets interface.

1. Introduction

Cluster systems are becoming attractive as the needs of high performance computing grow rapidly. Since cluster systems are relatively easy to build using commodity off-the-shelf (COTS) components, they show higher performance/price ratio than traditional supercomputers. This makes cluster systems the most common computer architecture seen in the world's fastest TOP500 supercomputer list [1].

One of the main obstacles in constructing a scalable cluster system is the communication performance; as the number of cluster nodes increases, the communication traffic among them also increases inevitably, limiting the overall performance of cluster systems. Recently, many high-speed System Area Networks (SANs), such as Gigabit Ethernet, Myrinet, Quadrics, and InfiniBand Architecture, have appeared supporting raw bandwidth more than one gigabits per second. In spite of the availability of high-speed interconnection hardware, however, it is not easy to deliver the raw transmission speed to end users due to various software overheads involved in communication. Therefore, it is essential to devise an efficient communication architecture which minimizes the communication cost in order to achieve the scalability beyond several tens or hundreds of nodes.

User-level communication (ULC) architectures attempt to accelerate the communication performance by removing the operating system from the critical communication path. It rests on the observation that the traditional communication architecture based on TCP/IP protocol suite suffers from the TCP/IP protocol overhead in the cluster environment. Moreover, ULC architectures perform most of protocol processing in the user space, thus eliminating overheads associated with context switching and data copying between the user and the kernel space.

The Virtual Interface Architecture (VIA) [2] is an early industrial effort to standardize ULC architectures. Recently, InfiniBand Architecture (IBA) [15] has been standardized as one of the next generation interconnection networks borrowing many concepts from VIA. The VIA and IBA define a set of standard application programming interface (API), called VIPL (VI Provider Library) and VAPI (Verbs API), respectively. These APIs provide fully-protected, user-level access to a network hardware, allowing for efficient communication for scalable clusters.

Although VIPL and VAPI enable developers to exploit high performance network at user-level, they are considered to be at too low a level for general network application programming [3]. Hence, many researchers have endeavored to build another portable high-level communication layers over VIA and IBA [11][18][19] that can hide low-level details to end users. Among various popular high-level com-

Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)
0190-3918/05 $20.00 © 2005 **IEEE**

munication layers, one of possible candidates that can be used over ULC architecture is the Berkeley sockets API [4] considering its widespread use and acceptance in distributed environments. The sockets API is a de facto standard for network programming and provides a means for developing applications independent of network hardware or protocols.

There can be several different approaches to supporting the sockets API over ULC architecture. Specifically, we can make the sockets API available for use, either (1) by inserting an IP emulation layer inside the kernel which bridges the gap between IP layer and user-level communication device (e.g. LANEVI for VIA or IPoIB for IBA), (2) by providing a user-level sockets layer, or (3) by providing a kernel-level sockets layer. Both the user-level and the kernel-level sockets layers emulate the sockets API directly over VIA or IBA. However, the user-level sockets layer exists as a library in the user space, while the kernel-level sockets layer resides in the kernel space, which bypasses the TCP/IP protocol stack during data transfer.

Each of the aforementioned approaches reveals different characteristics and design issues, not to mention the communication performance. This urges us to investigate the pros and cons of each approach both qualitatively and quantitatively. Since we have an access to the implementations of the first and second approaches from LANEVI driver of Emulex and our previous work [11], respectively, it is required to have a kernel-level sockets layer over VIA for making the comparison.

Thus, we first design and implement a kernel-level sockets layer, called KSOVIA (Kernel-level Sockets Over VIA), and then compare KSOVIA with other existing approaches. The primary objective of this paper is to identify design issues and trade-offs among different approaches supporting the sockets layer over ULC architecture, and to quantitatively analyze the various factors which affect the resulting communication performance.

The rest of the paper is organized as follows. The next section briefly overviews different approaches to supporting the sockets API over VIA, and presents motivations and contributions of our work. Section 3 compares design issues among different approaches, with emphasis on implementation details of KSOVIA. Section 4 shows experimental results and compares the performance of KSOVIA with other approaches. Section 5 presents related work. Finally, we conclude in section 6.

2. Background

2.1. Virtual interface architecture (VIA)

The organization of VIA is briefly depicted in Figure 1. VIA consists of four basic components: Virtual Interfaces, Completion Queues, VI Provider, and VI Consumer.

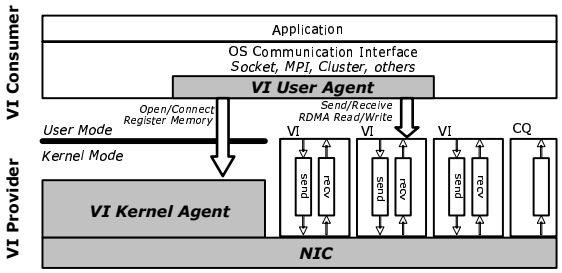


Figure 1. The organization of the Virtual Interface Architecture

VIA provides a consumer process with a protected, directly-accessible interface to a network hardware called Virtual Interface (VI), which is used as a communication endpoint. A VI consists of a pair of work queues: a send queue (SQ) and a receive queue (RQ). VI Provider consists of a physical network adapter and Kernel Agent, while VI Consumer represents the user of a VI.

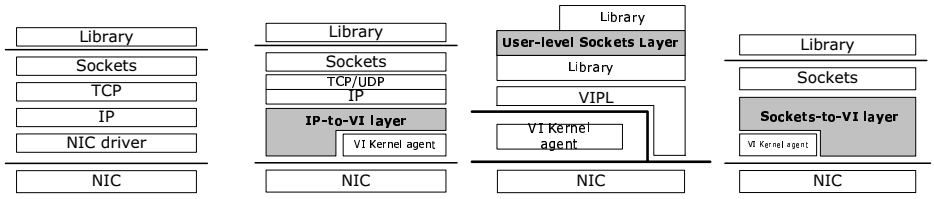
Sending or receiving data in VIA is comprised of two separate phases, namely the posting phase and the reaping phase. In the posting phase, VI Consumer posts a request on a work queue, in the form of a descriptor which contains all the information to transmit data. When the processing of the descriptor completes, the NIC marks a DONE bit in the status field of the descriptor. Those completed descriptors are identified and then removed from the work queue by VI Consumer in the reaping phase. A Completion Queue (CQ) allows a VI Consumer to coalesce notification of descriptor completions from multiple work queues in a single location. Once this association is established, notification of the completed requests for the work queue is automatically directed to the CQ.

Several VIA implementations are available for Linux platforms. M-VIA [5] emulates the VIA specification by software for legacy Fast Ethernet and Gigabit Ethernet NICs. Berkeley VIA [6], SVIA [7], and MyVIA [8] 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, as it is one of the most stable VIA implementations.

2.2. Supporting the sockets API over VIA

There can be several different approaches for supporting the sockets interface over VIA, as illustrated in Figure 2. Figure 2(a) shows the traditional communication architecture, in which the sockets layer is located on top of the TCP/IP protocol stack.

A simple way to support the sockets interface on top of VIA is to insert an adaptation layer between IP and VI Kernel Agent, as depicted in Figure 2(b). As the IP layer is



(a) traditional architecture (b) LANEVI (c) SOVIA (d) KSOVIA

Figure 2. Supporting the sockets API over VIA

emulated on VIA, an IP address is assigned to the NIC and the existing IP-based network applications can be run without any modification. The LANEVI (LAN Emulation on VI) [10] driver supplied by Emulex for its cLAN NICs is an example of such layers. Internally, the LANEVI driver relies on a thin layer called kVIPL, which is a kernel-level counterpart to VIPL allowing the kernel to use VIPL-like interfaces to access the NIC. Due to restrictions in the kernel space, however, not all of VIPL interfaces are implemented in kVIPL.

In order to eliminate overheads incurred by the TCP/IP protocol stack and to fully utilize the VIA's user-level data transfer capability, a user-level sockets layer over VIA, such as SOVIA [11], has been proposed. SOVIA is a lightweight and portable communication layer, which aims at providing the sockets interface entirely at user-level without sacrificing the performance of the underlying VIA layer. As Figure 2(c) shows, user-level sockets layers are generally implemented as a user-level library on top of VIPL. It is reported that the SOVIA layer successfully realizes comparable performance to native VIA, while offering the portable sockets semantics to application developers.

User-level sockets layers implemented on top of ULC architectures have been also introduced for Myrinet [12]. Gigabit Ethernet [13], and SCI [14]. However, all of those user-level sockets layers show a compatibility problem in that they hardly support the exec() system call, since sockets-related data structures maintained at the user level are eliminated during exec(). In addition, it is very complicated to share sockets connections between parent and child processes after the fork() system call, which makes it difficult to support concurrent server daemons or "super-server" daemons such as *inetd*. These problems are inherent limitations in any user-level sockets implementations.

Another approach that can solve the compatibility problem of user-level sockets layers is to use a kernel-level sockets layer, which moves the sockets support back into the kernel space, as shown in Figure 2(d). The kernel-level sockets layer supports the sockets API inside the kernel, but still bypasses the TCP/IP protocol stack interacting directly with the VI-aware NIC. It can freely access sockets-related data structures kept inside the kernel and is able to preserve most sockets semantics easily.

2.3. Motivations and Contributions

Our work is motivated by a desire to compare different approaches to supporting the sockets API over ULC architecture, both qualitatively and quantitatively. In order to do that, we first had to develop KSOVIA, a kernel-level sockets layer over VIA, because there was no corresponding implementation that could be used for the comparison.

The KSOVIA layer is intended to behave the same as the existing SOVIA layer as much as possible in terms of protocol processing such as internal state transitions and flow control algorithms, with the only exception being located in the kernel space. There are, however, several cases where some design changes are required in KSOVIA due to intrinsic differences between the user and the kernel space.

The development of KSOVIA enables us to compare three different approaches, LANEVI, SOVIA, and KSOVIA, on the same cLAN-based platform. Because SOVIA and KSOVIA work basically the same way, we can accurately identify the amount of overheads added in the kernel-level implementation (such as context switching overhead) by comparing their performance. Similarly, we can roughly figure out the TCP/IP protocol overhead by comparing the performance of KSOVIA and LANEVI. In the next section, we also describe design differences among LANEVI, SOVIA, and KSOVIA, with respect to data sending, data receiving, connection management, and flow control.

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

- We classify different approaches to supporting the sockets interface over ULC architecture such as VIA.
- We design and implement KSOVIA, a kernel-level sockets layer over VIA, in order to perform the comparison with the existing LANEVI and SOVIA layer.
- We examine design and implementation issues in supporting the sockets interface, with paying attention to design differences and trade-offs among LANEVI, SOVIA, and KSOVIA.
- We measure the latency and the bandwidth of LANEVI, SOVIA, KSOVIA, and native VIA, on the same platform to understand their relative performance.

- We quantitatively analyze the individual costs associated with the communication (for example, context switching overhead, data copying overhead, and the TCP/IP protocol overhead) by comparing the performance of LANEVI, SOVIA, and KSOVIA.

3. A comparison of design and implementation issues

In this section, we present the internal workings of KSOVIA, and compare its design and implementation issues with LANEVI and SOVIA. The detailed description on LANEVI has not been published in the literature, and thus it is guessed from the source code. Since we are unable to cover SOVIA completely in this paper due to the space limitation, readers are encouraged to refer to [11] for further details on SOVIA.

3.1. Data receiving

In the traditional TCP/IP-based communication architecture, data receiving is handled by an interrupt handler in a transparent way to user applications. In VIA, however, VI Consumer itself should extract completed descriptors from the receive queue (RQ) and post a new one for each incoming data that is delivered asynchronously.

As the LANEVI driver works at the network device driver level, data receiving is handled similarly to the traditional architecture. First, LANEVI prepares a set of receive descriptors and temporary buffers when it is loaded into the kernel. These descriptors and buffers are registered in advance so that the NIC can access them via DMA operations. When a packet arrives from the peer, an interrupt occurs which enables LANEVI to reap the descriptor after copying received data to a socket buffer. The socket buffer is passed on to the upper layer and processed by the TCP/IP protocol stack.

Whereas LANEVI merely transfers IP packets to and from VIA using the kernel-level interfaces provided by kVIPL, SOVIA emulates the sockets interface directly at user-level using VIPL. Whenever an application calls socket(), SOVIA creates a new VI. All the RQs of VIs are connected to a completion queue (CQ) to get the notification of data arrival in a single location. Usually, the application itself checks the CQ either by polling or by using a blocking VIPL interface, to see if there is any pending data. As in LANEVI, SOVIA also uses temporary buffers to store data sent from the peer.

While SOVIA creates a CQ for each process in the system, KSOVIA uses only one system-wide CQ as it is located in the kernel. KSOVIA also utilizes temporary buffers as SOVIA does, although they are allocated in the kernel

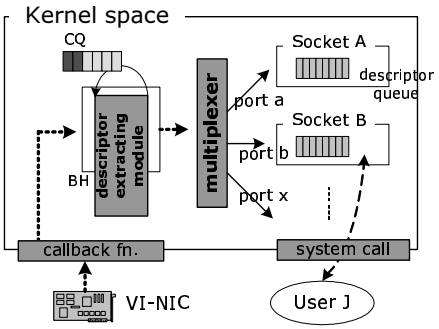


Figure 3. Data receiving in KSOVIA

space in KSOVIA. We elaborate upon the KSOVIA's implementation of recv() in Figure 3. The entire data receiving phase is divided into two sub-phases in KSOVIA. The first phase is to receive data from the peer and the second phase is to multiplex them to a corresponding socket.

When the NIC receives data from the peer, it generates an interrupt which schedules the callback function in KSOVIA. The callback function recognizes the data reception event and delegates the processing to descriptor extracting module (DEM) in the bottom half of the Linux kernel. The DEM obtains a VI handle from the CQ and extracts the completed descriptor from the VI. Since we specify source and destination port numbers in the descriptor, the multiplexer module can enqueue the extracted descriptor to the descriptor queue of the corresponding socket.

As we have seen previously, all of LANEVI, SOVIA, and KSOVIA rely on temporary buffers to satisfy the preposting constraint, because there is excessive synchronization overhead otherwise [11]. As long as we use temporary buffers, one data copy from the temporary buffer to the user buffer is unavoidable. Note that LANEVI experiences one more data copy due to the existence of the kernel socket buffer; one from the temporary buffer to the socket buffer, and the other from the socket buffer to the user buffer.

3.2. Data sending

In LANEVI, sending data via sockets is accomplished with the help of TCP/IP protocols. User data are encapsulated in the socket buffers with TCP and IP headers as they are passed through the TCP/IP protocol stack. Those socket buffers are finally handed over to the LANEVI driver for the transmission. As socket buffers can not be used directly for DMA operations, LANEVI copies the contents of socket buffers to the pre-registered temporary buffers in the kernel. LANEVI also forms send descriptors, and posts them to the SQ.

Unlike LANEVI, SOVIA does not make use of temporary buffers in general. Instead, SOVIA registers the memory region of user data before actual transmission, so that the NIC can access them via DMA operations. This ap-

proach avoids the unnecessary data copying and achieves the zero-copy protocol during data sending. However, as users should not modify their data before the NIC finishes the entire sending operation, SOVIA is unable to support the non-blocking mode of send(). For this reason, SOVIA also provides a sending mode that performs one copy from user data to temporary buffers.

In the case of KSOVIA, it is inevitable to use temporary buffers as the user data should be moved into the kernel space first. If users request the send() system call, KSOVIA copies data to temporary buffers located in the kernel, and posts send descriptors to the SQ (the posting phase). After the NIC finishes the sending operation, KSOVIA dequeues them from the SQ (the reaping phase).

3.3. Connection management

Connection management is one of the noticeably different parts among LANEVI, SOVIA, and KSOVIA. In cLAN, when a new node joins, its LANEVI driver automatically connects to every other node in the same cLAN network because VIA is based on the connection-oriented communication model. Establishing a logical connection between two sockets uses special TCP packets and these packets are exchanged through the pre-created VIA connection between two nodes. Thus, when a socket makes a connection with the peer, two separate connection establishments are performed in LANEVI. This mechanism is inefficient, but it is inevitable to emulate connectionless IP services over the connection-oriented VIA.

SOVIA maintains two 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 the listen() system call. 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.

KSOVIA does not need a close thread since the kernel is able to receive any incoming packets without resort to the user application. KSOVIA, however, employs a kernel thread to deal with VIA connection requests and replies. As in SOVIA, this kernel thread is created after the listen() system call.

3.4. Flow control

LANEVI has no concern for flow control because flow is mostly managed by the upper TCP layer. Although TCP has many complicated flow control mechanisms, some features of them, such as congestion window, are not necessary in the reliable cluster interconnection networks. Thus, it is required to devise a lightweight flow control mechanism for SOVIA and KSOVIA.

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 windows size w, which denotes the maximum number of data packets the sender is allowed to transmit without waiting for an acknowledgment. When a socket is created, w receive descriptors are pre-posted in the RQ. Whenever the sender transmits a data packet, it decreases w by one to denote that one of the receive descriptors has been consumed. If w reaches zero, the sender stops to transmit data packets until w becomes a positive number. Windows size w is increased by one if an acknowledgement is delivered to the sender. To enhance the bandwidth further, acknowledgements can be delayed and piggybacked to data packets. We have implemented the same flow control algorithm in KSOVIA.

4. Evaluations

4.1. Experimental setup

We have measured the performance of LANEVI, SOVIA and KSOVIA with the Linux kernel 2.4.18 and cLAN driver version 2.0.1. As the original kVIPL supplied by Emulex does not provide connection management APIs, we have slightly extended kVIPL to implement missing interfaces. The hardware platform used for performance evaluation is two Linux servers, each consisting of 1.6GHz Intel Pentium 4 processor, 512KB L2 cache, and 768MB of main memory. Two cLAN1000 network adapters are attached to a 32-bit 33MHz PCI slot of each server without any intermediate switch.

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 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 sockets applications.

Communication mechanisms evaluated in this paper are summarized in Table 1. The measurement results of VIPL and VIPL_POLL are obtained from microbenchmarks written in VIPL and they are considered as the baseline of other results. Note that SOVIA can be implemented either using non-blocking APIs (SOVIA_POLL) or using blocking APIs (SOVIA). On the contrary, KSOVIA does not use non-blocking APIs because polling is not allowed inside the kernel.

Table 1. Evaluated communication mechanisms

Communication mechanisms	Description
VIPL_POLL	Use VIPL with non-blocking APIs in the user space
VIPL	Use 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 (blocking APIs are used implicitly)

4.2. One-way latency

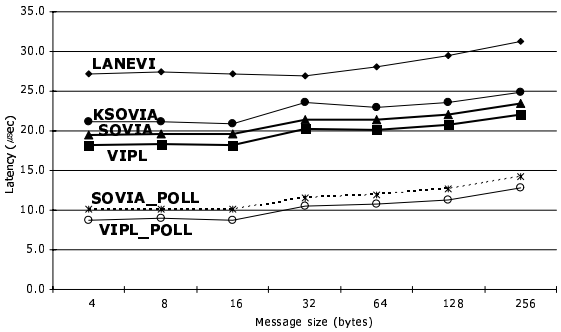


Figure 4. One-way latency for small messages

Figure 4 shows the measured one-way latency when the message size varies from 4 bytes to 256 bytes. As expected, VIPL_POLL shows the smallest latency for all the message sizes. Note that both VIPL_POLL and SOVIA_POLL check the arrival of new data by polling the completion queue. In general, network applications can not use this polling mechanism because it wastes most of CPU times on busy waiting. Thus, these results are only meaningful as the practical lower bound of the one-way latency. Since other communication mechanisms, such as VIPL, SOVIA, KSOVIA, and LANEVI, are based on blocking APIs, they exhibit the higher latency than VIPL_POLL and SOVIA_POLL.

From Figure 4, it can be seen that SOVIA shows the constantly higher latency than VIPL by about 1.3 μsec. It is mainly due to the added complexity in the SOVIA layer to support sockets interface. Figure 4 also illustrates that the latency of KSOVIA is higher than that of SOVIA by 1.7 μsec.

As we have designed KSOVIA to behave the same as SOVIA, the gap between KSOVIA and SOVIA mostly comes from the fact that KSOVIA is located in the kernel. For further analysis on the factors affecting the latency, we measure the time spent for calling a system call (labeled as *send()+recv()*) and for data copy (labeled as *memory copy*). We present the results in Figure 5 in conjunction with the latency of VIPL and SOVIA. Note that *baseline transmission*

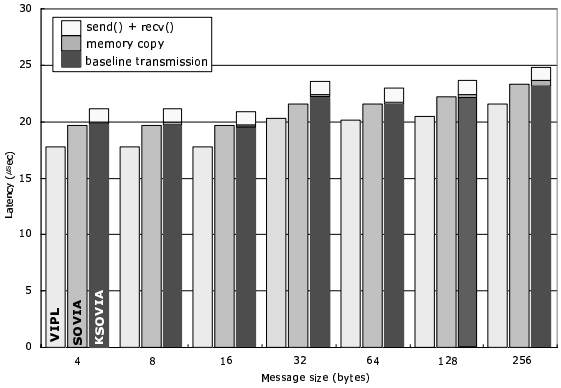


Figure 5. Break down of the one-way latency of KSOVIA for small messages

denotes the latency of KSOVIA except the time for system calls and data copy. We can observe that most of the difference in latency between KSOVIA and SOVIA stems from the context switching overhead between the kernel and the user space around the send() and recv() system calls. The data copying overhead is negligible in Figure 5, as the message size is small.

Compared to KSOVIA, LANEVI has the additional TCP/IP overhead, exhibiting the worst latency in Figure 4. From the difference in latency between LANEVI and KSOVIA, we can roughly conjecture that LANEVI spends 22% of its time (about 6 μsec out of 27.2 μsec) for the TCP/IP protocol processing.

4.3. Unidirectional bandwidth

Figure 6 illustrates 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 message size is less then 2Kbytes. This is because TCP's Nagle algorithm combines small messages together before a packet is transmitted. Although we do not enable this feature for SOVIA and KSOVIA in the measurement, it is already reported in [11] that SOVIA outperforms LANEVI even for the small mes-

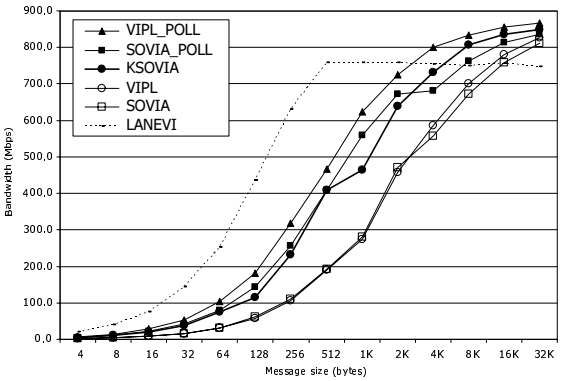


Figure 6. Unidirectional bandwidth

sage sizes if we add the similar ability to SOVIA. The bandwidth of LANEVI is saturated to 760 Mbps at the 512-byte message size and remains the smallest when the message size is larger than 16 Kbytes.

As SOVIA is implemented using 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.

It is impressive that KSOVIA delivers much higher bandwidth than SOVIA, in spite of the context switching overhead which has increased the latency slightly. SOVIA performs the posting phase and the reaping phase in the same send() system call, while KSOVIA defers the reaping phase to the bottom half in order to process multiple descriptors in a single run. This maximizes the concurrency during data sending in KSOVIA and leads to the bandwidth comparable to SOVIA_POLL. If SOVIA supports this kind of concurrency in sending, the bandwidth of SOVIA is likely to increase further.

4.4. FTP performance

Table 2. FTP bandwdith

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

We run an FTP server (linux-ftpd-0.17) and client (netkit-ftp-0.17) over different communication mechanisms such as LANEVI, SOVIA, and KSOVIA, in order to verify their functional validity and to measure the performance of real sockets applications. Two files are transferred from the server to the client; one is small (12.75 MB) and the other is rather large (161.1 MB). We have arranged that all the files reside in ram disks to avoid the influence of the disk subsystems. Table 2 summarizes the performance of file transfers. First of all, as the bandwidth of local copy using *cp* command is large enough exceeding 1 Gbps, we can see that the FTP performance is not bounded by the ram disks performance.

Table 2 shows that LANEVI achieves 91.2% (for small file) and 89.6% (for large file) of the bandwidth of KSOVIA. These results are analogous to those obtained from microbenchmarks. As described in the previous section, KSOVIA shows a little higher bandwidth than SOVIA due to several kernel-level optimizations.

5. Related work

The most popular communication layers used in the cluster environments are MPI (Message Passing Interface) and Berkeley sockets interface [4]. Since sockets interface provides more general communication interface, many researches have been performed to support the sockets API on VIA. VIsocket [16] proposed by Itoh et al. is similar to KSOVIA in that it provides sockets functionality below the STREAMS module in Solaris by collapsing internal TCP/IP layers. However, the design and performance details of VIsocket have not been published yet.

Recently, InfiniBand Architecture (IBA) [15] 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 [17] and SDP [17]. 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 availble [17][20][21], there is little publicly available documetation of internal implementation and comparison among several design choices we focus on. Due to many similarities between VIA and IBA, we believe the results obtained in this paper are also useful in developing such layers as IPoIB and SDP.

6. Concluding remarks

This paper compares three different approaches to supporting the sockets interface over ULC architecture. We

utilize VIA, one of ULC architectures, for the comparison. We compare LANEVI, SOVIA, and KSOVIA which is developed based on VIA. LANEVI emulates the IP layer on VIA, while SOVIA and KSOVIA represent a user-level and a kernel-level sockets layer over VIA, respectively. Since there is no reference implementation for kernel-level sockets layers, we have designed and implemented the KSOVIA layer in this paper.

From the measurement results, we observe that LANEVI exhibits the worst performance among all the approaches evaluated in this paper. By comparing the latency of LANEVI and KSOVIA, we can roughly conjecture that LANEVI spends 22% of its time (about 6 μsec out of 27.2 μsec) for the TCP/IP protocol processing. It is also identified from the difference in latency between KSOVIA and SOVIA that the context switching overhead is less than 2 μsec. SOVIA or KSOVIA has the added complexity to support sockets interface compared to native VIA, which increases the latency by about 1 μsec.

Putting it all together, we can conclude that there is no reason to use the LANEVI driver for sockets-based applications. It shows the worst performance both in terms of latency and bandwidth, and has the additional overhead in connection management to emulate connectionless IP services on the connection-oriented VIA. On the other hand, SOVIA and KSOVIA have their own pros and cons. Even though SOVIA presents the slightly lower bandwidth than KSOVIA, it is useful when applications are latency-sensitive and do not use fork() or exec() system calls. KSOVIA can be used effectively for applications which require high bandwidth or the full compatibility with the existing sockets interface.

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

References

[1] http://www.top500.org, TOP500 Supercomputer sites.

[2] Compaq Corporation, Intel Corporation, and Microsoft Corporation, *Virtual Interface Architecture Specification*, Version 1.0, 1997.

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

[4] 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 Inc., 1996.

[5] P. Bozeman and B. Saphir, "A Modular High Performance Implementation of the Virtual Interface Architecture," *Proceedings of Extreme Linux Conference*, 1999.

[6] P. Buonadonna, A. Geweke, and D. Culler, "An Implementation and Analysis of the Virtual Interface Architecture," *Proceedings of ACM International Conference on Supercomputing*, 1998.

[7] J.-L. Yu, M.-S. Lee and S.-R. Maeng, "An Efficient Implementation of Virtual Interface Architecture using Adaptive Transfer Mechanism on Myrinet," *Proceedings of International Conference on Parallel and Distributed Systems*, 2001.

[8] Y. Chen, X. Wang, Z. Jiao, J. Xie, Z. Du, and S. Li, "MyVIA: A Design and Implementation of the High Performance Virtual Interface Architecture," *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[9] http://www.emulex.com/products/legacy/vi/clan1000.html, Emulex Corporation-Products, 2003.

[10] Emulex Inc., cLAN for Linux: Software User's Guide, 2001.

[11] J.-S. Kim, K. Kim, S.-I. Jung, and S. Ha, "Design and Implementation of a User-level Sockets Layer over Virtual Interface Architecture," *Concurrency and Computation: Practice and Experience*, Volume 15, Issue 7-8, 2003.

[12] M. Fischer, "GMSOCKS - A Direct Sockets Implementations on Myrinet," *Proceedings of the IEEE International Conference on Cluster Computing*, 2001.

[13] P. Balaji, P. Shivan, P. Wyckoff, and D. Panda, "High Performance User Level Sockets over Gigabit Ethernet," *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[14] F. Seifert and H. Kohmann, "SCI SOCKET - A Fast Socket Implementation over SCI," White paper, Dophin Interconnect Solutions Inc.

[15] http://www.infinibandta.org, InfiniBand Trade Association.

[16] M. Itoh, T. Ishizaki and M. Kishimoto, "Accelerated Socket Communication in System Area Network," *Proceedings of the IEEE International Conference on Cluster Computing*, 2000.

[17] http://infiniband.sourceforge.net, InfiniBand Linux SourceForge Project.

[18] J. Liu, J. Wu, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," *International Journal of Parallel Programming*, 2004.

[19] http://old-www.nersc.gov/research/FTG/mvich, MVICH - MPI for the Virtual Interface Architecture.

[20] http://www.openib.org, OpenIB.org.

[21] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Sockets Direct Procotol over InfiniBand in Clusters: Is it Beneficial?," *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.