High Performance Sockets and RPC over Virtual Interface (VI) Architecture

Hemal V. Shah[1], Calton Pu[2], and Rajesh S. Madukkarumukumana[1,2]

[1] M/S CO3-202, Server Architecture Lab,
Intel Corporation,
5200 N.E. Elam Young Pkwy, Hillsboro, OR 97124.
{hemal.shah,rajesh.sankaran}@intel.com
[2] Department of Computer Science & Engineering,
Oregon Graduate Institute of Science and Technology,
P.O. Box 91000, Portland, OR 97291.
{calton,rajeshs}@cse.ogi.edu

Abstract. Standard user-level networking architecture such as Virtual Interface (VI) Architecture enables distributed applications to perform low overhead communication over System Area Networks (SANs). This paper describes how high-level communication paradigms like stream sockets and remote procedure call (RPC) can be efficiently built over user-level networking architectures. To evaluate performance benefits for standard client-server and multi-threaded environments, our focus is on off-the-shelf sockets and RPC interfaces and commercially available VI Architecture based SANs. The key design techniques developed in this research include credit-based flow control, decentralized user-level protocol processing, caching of pinned communication buffers, and deferred processing of completed send operations. The one-way bandwidth achieved by stream sockets over VI Architecture was 3 to 4 times better than the same achieved by running legacy protocols over the same interconnect. On the same SAN, high-performance stream sockets and RPC over VI Architecture achieve significantly better (between 2-3x) latency than conventional stream sockets and RPC over standard network protocols in Windows NT^{TM} 4.0 environment. Furthermore, our high-performance RPC transparently improved the network performance of Distributed Component Object Model (DCOM) by a factor of 2 to 3.
Keywords: Stream sockets, Remote Procedure Call (RPC), Virtual Interface (VI) Architecture, System Area Network (SAN), Distributed Component Object Model (DCOM).

1 Introduction

With the advent of System Area Networks (SANs), low latency and high bandwidth communication has become a reality. These networks have opened new horizons for cluster computing. But, the centralized in-kernel protocol processing in legacy transport protocols such as TCP/IP prohibits applications from realizing the raw hardware performance offered by underlying SANs. In order

to address this problem, Virtual Interface (VI) Architecture standard was developed. However, building high-level applications using primitives provided by VI Architecture is complex due to lack of transport functionality such as flow control, communication buffer management, fragmentation/re-assembly.

On the other hand, stream sockets and remote procedure calls (RPCs) provide simple and easy to use communication abstractions for distributed applications and distributed object computing frameworks such as DCOM [5], Java RMI [9], CORBA [12]. Stream sockets provide a connection-oriented, bi-directional byte-stream model for inter-process communication. RPC mechanism enables a program to call procedures that execute in other address space and it hides networking details from applications.

This paper provides prototype designs and implementations of 1) stream sockets over VI Architecture and 2) RPC over VI Architecture. The design goals considered were:

- Performance: deliver close to raw end-to-end performance to multi-threaded client/server applications.
- Legacy support: support RPC/sockets application programming interfaces as much as possible.
- CPU overhead: minimize CPU cycles spent per byte transferred.

Optimizations for special cases such as single-threaded applications and modifying kernel components such as virtual memory management system can be used to further improve performance. Most of these techniques trade off application transparency and ease of use, and thus, were not considered as design goals.

This paper contributes VI Architecture specific design techniques, developed for optimizing stream sockets and RPC performance, such as credit based flow control, decentralized user-level protocol processing, caching of pinned communication buffers, and deferred processing of completed send operations. On Windows NT^{TM} 4.0, user-level stream sockets and RPC over VI Architecture achieve significantly better performance (2-3x improvement in latency and 3-4x improvement in one-way and bi-directional bandwidths) than stream sockets and RPC with legacy network protocols (TCP/IP) on the same SAN. For small messages, RPC over VI Architecture achieves an end-to-end latency comparable to the latency achieved by the local RPC implementation on the same system. Further, this translated into 2-3x improvement in DCOM network performance transparently.

The rest of the paper provides the details of the design and evaluation of stream sockets and RPC over VI Architecture. The outline of the remaining paper is as follows. Section 2 provides an overview of VI Architecture. The design and performance evaluation of stream sockets over VI Architecture is described in Section 3. Section 4 describes the design of RPC over VI Architecture and evaluates its performance. Section 5 provides a summary of the related work. Finally, conclusions and future work are presented in Section 6.

2 Virtual Interface (VI) Architecture

VI Architecture is a user-level networking architecture designed to achieve low latency, high bandwidth communication within a computing cluster. To a user process, VI Architecture provides direct access to the network interface in a fully protected fashion. The VI Architecture avoids intermediate data copies and bypasses operating system to achieve low latency, high bandwidth data transfer. The VI Architecture Specification 1.0 [15] was jointly authored by Intel Corporation, Microsoft Corporation, and Compaq Computer Corporation.

Virtual Interface Architecture uses a VI construct to present an illusion to each process that it owns the interface to the network. A VI is owned and maintained by a single process. Each VI consists of two work queues: one send queue and one receive queue. On each work queue, descriptors are used to describe work to be done by the network interface. A linked-list of variable length descriptors forms each queue. Ordering and data consistency rules are only maintained within one VI but not between different VIs. VI Architecture also provides a completion queue construct used to link completion notifications from multiple work queues to a single queue.

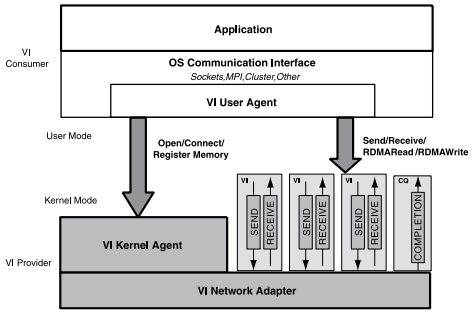

Fig. 1. VI Architecture

Memory protection for all VI operations is provided by protection tag (a unique identifier) mechanism. Protection tags are associated with VIs and memory regions. The memory regions used by descriptors and data buffers are registered prior to data transfer operations. Memory registration gives VI NIC a method to translate a virtual address to a physical address. The user receives an opaque memory handle as a result of memory registration. This allows a user to refer to a memory region using a memory handle/virtual address pair without worrying about crossing page boundaries and keeping track of the virtual address to tag mapping.

The VI Architecture defines two types of data transfer operations: 1) traditional send/receive operations, and 2) Remote-DMA (RDMA) read/write operations. A user process posts descriptors on work queues and uses either polling or blocking mechanism to synchronize with the completed operations. The two

descriptor processing models supported by VI Architecture are *the work queue model* and *the completion queue model*. In the work queue model, the VI consumer polls or waits for completions on a particular work queue. The VI consumer polls or waits for completions on a set of work queues in the completion queue model. The processing of descriptors posted on a VI is performed in FIFO order but there is no implicit relationship between the processing of descriptors posted on different VIs. For more details on VI Architecture, the interested reader is referred to [6, 15]. The next two sections describe design and implementation of stream sockets and RPC over VI Architecture.

3 Stream Sockets over VI Architecture

Stream sockets provide connection-oriented, bi-directional byte-stream oriented communication model. Windows Sockets 2 Architecture utilizes sockets paradigm and provides protocol-independent transport interface. Figure 2 shows an overview of Windows Sockets 2 architecture [16]. It consists of an application programming interface (API) used by applications and service provider interfaces (SPIs) implemented by service providers.

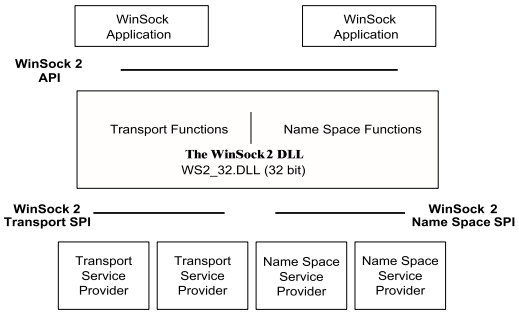

Fig. 2. Windows Sockets 2 Architecture

This extensible architecture allows multiple service providers to coexist. The transport service providers implement the actual transport protocol and the name space providers map WinSock's name space SPI to some existing name space. In this research, Windows Sockets 2 Architecture is used to provide high performance VI-based stream sockets. The new transport service provider was completely implemented at user-level.

3.1 Design and Implementation

User-level decentralized protocol processing, credit-based flow control, caching of pinned communication buffers, and minimization of CPU overhead are the main techniques used in our stream sockets over VI Architecture implementation. These techniques along with the design are described next.

Endpoint Mappings and Connection Management The connection-oriented design provided by VI Architecture maps well to stream sockets. Each stream socket (endpoint) is mapped to a VI. Each endpoint consists of send/receive descriptors, registered send/receive buffers, and information for credit based flow control. Each endpoint has a queue of received buffers containing data yet to be read by the application. In order to reduce number of memory registrations, global pools of send/receive descriptors are created and registered within a process during service provider initialization. During creation of an endpoint, descriptors are assigned from these global pools. Upon destruction of an endpoint, descriptors are returned back to the global pools. A queue of pending connection requests is maintained at each endpoint. A dedicated thread manages connection requests on the endpoint. IP port numbers are used as discriminators in underlying connection establishment between VIs.

Data Transfer and Flow Control The reliability mode used in data transfers is *Reliable Delivery*. Reliable delivery VI guarantees that the data submitted for transfer is delivered exactly once, intact, and in the order submitted, in the absence of errors. Transport errors are extremely rare and considered catastrophic. In network interfaces that emulate VI functionality, reliable delivery is commonly implemented in NIC firmware or software. In native VI NICs (such as GNN1000 [8]), the hardware provides reliable delivery. Due to the use of reliable delivery VIs, fragmentation of the messages can be handled without using sequence numbers. Furthermore, the transport service provider need not worry about managing acknowledgements and detecting duplicates. The timeout and retransmission mechanisms are not incorporated in the transport service provider as transport errors are rare and connection is broken when transport errors occur.

Three types of messages used in data transfer are *CreditRequest*, *CreditResponse*, and *Data*. The transport service provider is responsible for managing end-to-end flow control between two endpoints. For providing end-to-end flow control, a credit-based scheme is used. If the number of send credits is sufficient, then the sender prepares and sends the packet. Otherwise, the sender sends a credit request (*CreditRequest*) and waits for the credit response (*CreditResponse*). Upon receiving credit response, it continues sending packets. In response to sender's request for credit update, the receiver sends the credit response only when it has enough receive credits (above the low water mark). In the case of not having enough credits when the credit request arrives, the receiver defers the sending of credit response until sufficient receive credits are available. As application consumes the received data, receive credits are regained. Credit-based flow control scheme and use of reliable delivery VIs provide low overhead user-level protocol processing.

Descriptor Processing In VI Architecture, a data transfer operation is split into two phases: initiation of the operation (posting a descriptor) and completion of the operation (polling or waitiing for a descriptor to complete on a work queue). Due to push model of processing and high-speed reliable SANs, each

send descriptor completes quickly once it reaches the head of the send queue. So in order to reduce interrupts, polling is used for checking completion of send descriptors. Checking completion of a send descriptor is deferred until either there are not enough send credits available or the entire message is posted. This type of deferred de-queuing of send descriptors reduces CPU overhead compared to when polling immediately after posting each send descriptor.

The transport service provider maintains a small-sized LRU cache of registered application buffers. This allows zero-copy sends for frequently used send buffers. The application data is copied into pre-registered send buffers only when the application buffer is not found in the cache and is not added to the cache. To allow application specific tuning, the maximum number of LRU cache entries and the minimum size of registered application buffer are kept configurable.

Receive descriptors need to be pre-posted prior to posting of the matching send descriptors on the sender side. The data is always copied from the registered received buffers to the buffers supplied by the application for receiving data. The copying of data on the receiver side can be overlapped with VI NIC processing and physical communication. The receiver waits when there is no data available on the socket. When the receiver wakes up due to completion of a receive descriptor, the receiver de-queues as many completed receive descriptors as possible. This scheme for processing receive descriptors reduces the number of interrupts on the host system.

The transport service provider for stream sockets over VI Architecture was implemented at user-level. This allows decentralized protocol processing on per process basis. The user-level buffer management and flow control scheme do not experience kernel like restrictive environment. The communication subsystem becomes an integrated part of the application and this allows for an application specific tuning. The next subsection provides experimental evaluation of stream sockets over VI Architecture.

3.2 Experimental Evaluation

In the experiments involving micro-benchmarks, a pair of server systems, with four 400 MHZ Pentium^R II Xeon^{TM} processors (512K L2 cache), Intel AD450NX 64-bit PCI chipset, and 256 MB main memory, was used as a pair of host nodes. GigaNet's cLAN^{TM} GNN1000 interconnect (full duplex, 1.25 Gbps one-way) [8] with VI functionality implemented on NIC hardware is used as VI NIC. The software environment used for all the experiments included Windows NT^{TM} 4.0 with service pack 3 and Microsoft Visual C++ 6.0. As a default, the Maximum Transfer Unit (MTU) per packet used by the stream sockets over VI Architecture was 8 K bytes and credit-based flow control scheme reserved an initial receive credits of 32 for each connection. Unless stated, all the experimental results were obtained using these default values.

Round-Trip Latency In distributed applications, round-trip latencies of small messages play an important role in the performance and scalability of the sys-

tem. In order to measure round-trip latency, a ping-pong test was used in the experiments. Figure 3 compares the application-to-application round-trip latency achieved (averaged over 10000 runs) by raw VI Architecture primitives, stream sockets over VI Architecture (GNN1000), TCP/IP over Gigabit Ethernet, and TCP/IP over GNN1000. The round-trip latency achieved by stream sockets over VI Architecture is 2-3 times better than the round-trip latency achieved by both TCP/IP over Gigabit Ethernet and TCP/IP over GNN1000. Moreover, the average round-trip latency achieved for a given message size is within 50% of the round-trip latency achieved using raw VI architecture primitives.

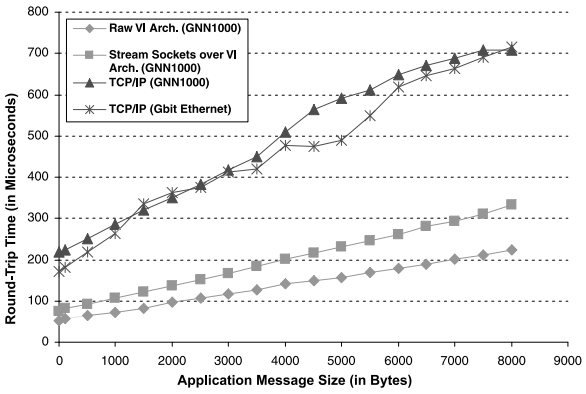

Fig. 3. Round-Trip Latency

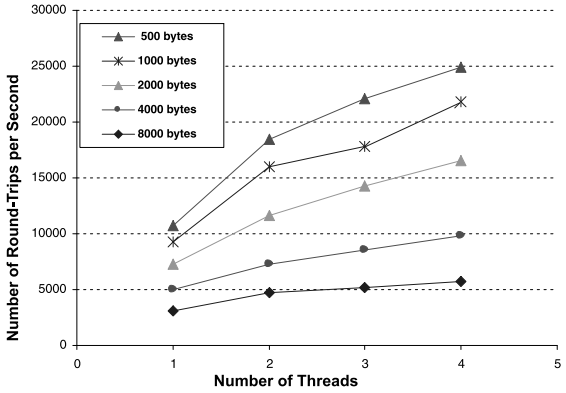

Fig. 4. SMP Scalability

In order to measure SMP scalability of stream sockets over VI Architecture, a multi-threaded ping-pong test (where each thread independently ping-pongs messages on a dedicated socket) was used. Figure 4 provides the SMP scalability for various message sizes. The metric used to measure scalability was the number of aggregate round-trips per second. Due to availability of multiple CPUs, most

of the protocol processing was performed in parallel. The primary reasons for limited scalability for large messages were the use of single VI NIC for processing the messages sent and received by multiple threads and availability of single I/O channel on host system (PCI bus). VI NIC performs the tasks of multiplexing, de-multiplexing, putting user-level data on the wire, copying received data into user-level buffer, and data transfer scheduling. Hence, for large messages, the VI NIC processing overhead can become significant. This suggests that having multiple VI NICs and multiple I/O channels can further improve SMP scalability.

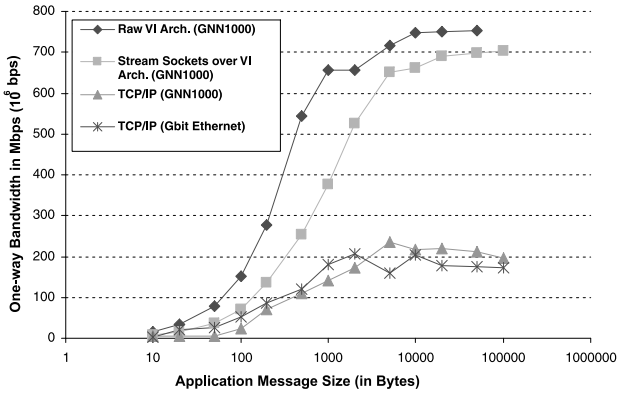

Fig. 5. One-way Receiver Bandwidth

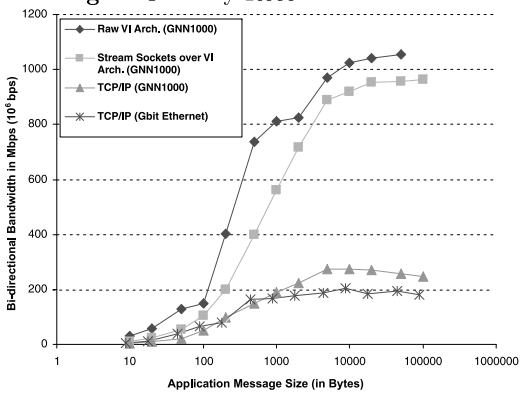

Fig. 6. Bi-Directional Bandwidth

Bandwidth Figure 5 provides comparison of application-to-application one-way bandwidths (measured using a stream of 10000 messages) achieved by raw VI Architecture, stream sockets over VI Architecture (GNN1000), TCP/IP over Gigabit Ethernet, and TCP/IP over GNN1000. For large messages, the one-way bandwidth achieved by stream sockets over VI Architecture is 3 to 4 times better

than the same achieved by running legacy protocols (TCP/IP) over both the same SAN and Gigabit Ethernet. The one-way bandwidth achieved by stream sockets over VI Architecture is within 10% of the one-way bandwidth achieved by the raw VI Architecture primitives using same descriptor processing models and synchronization mechanisms.

Similarly, Figure 6 compares the bi-directional bandwidths achieved by various messaging layers. The bi-directional bandwidth test has two threads (one sender and one receiver) using two separate communication endpoints. The sender thread sends a stream of messages on one endpoint and the receiver thread receives a stream of messages on another endpoint. The receiver bandwidths achieved on both nodes were added to obtain bi-directional bandwidth. Similar to one-way bandwidth, for large messages, the bi-directional bandwidth achieved by stream sockets over VI Architecture is 3-4 (4-5) times better than the same achieved by running legacy TCP/IP protocol over GNN1000 (Gigabit Ethernet). For large messages, the bi-directional bandwidth achieved by stream sockets over VI Architecture stays within 10% of the bi-directional bandwidth achieved by using raw VI Architecture primitives. Figure 7 shows the number of CPU cycles spent per byte transferred at different messaging layers. These experiments demonstrate that stream sockets over VI architecture not only achieve substantially better performance than legacy transport protocols, but also spend significantly less host CPU cycles per byte transferred than TCP/IP. Table 1 summarizes 4-byte round-trip latency and 50000-byte one-way bandwidth achieved by various messaging layers.

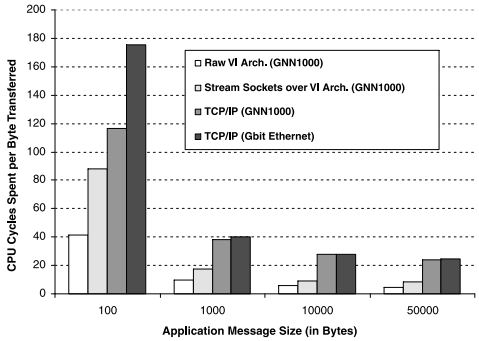

Fig. 7. CPU Overhead

4 RPC over VI Architecture

In order to examine the advantage in efficiently supporting an off-the-shelf RPC system over user-level networking architectures, we instrumented the Microsoft Remote Procedure Call (MSRPC) and DCOM facilities available in Windows

Table 1. Performance of Various Messaging Layers

Messaging Layer	4-byte Round-trip Latency (in μs)	50000-byte One-way Bandwidth (in 10^6 bps)
Raw VI Architecture Primitives	52.4	753.2
Stream Sockets over VI Architecture	74.9	697.3
TCP/IP over GNN1000	219.1	211
TCP/IP over Gigabit Ethernet	170.5	175

NT^{TM} 4.0. Experimental results in our earlier research [10] have shown that the latency incurred in MSRPC over conventional high-speed networks such as Gigabit Ethernet is dominated by the overhead in the legacy transport protocol (TCP/IP) stack. The primary focus of this prototype effort was to transparently improve the performance of RPC and DCOM applications over SANs by reducing the transport protocol overheads.

4.1 Operational Overview

MSRPC system is primarily composed of the IDL compiler generated proxy and stubs, the RPC runtime and the various dynamically loadable transport modules. The remoting architecture in DCOM [5] is abstracted as an Object RPC (ORPC) layer built over MSRPC. As a natural consequence, DCOM wire protocol performance directly follows the performance of MSRPC. MSRPC provides a high performance local RPC transport implementation for procedure calls across address spaces on the same machine. For procedure calls across the network, the transport interface in MSRPC supports both connectionless and connection-oriented transport interfaces. Figure 8 shows the operational view of the MSRPC system supporting multiple transport providers. Building high performance RPC over our stream sockets prototype was definitely a viable option. Rather, we added a connection-oriented MSRPC loadable transport directly over VI architecture to avoid additional layering overheads. Our prototype RPC transport[1] makes use of the low-latency and high reliability properties of SANs. The following sub-sections describe in detail the design and implementation trade-off made in the RPC transport for SANs.

4.2 RPC Transport Design for SANs

The main objective for the transport design was to improve the network performance of multi-threaded RPC and DCOM client/server applications. Any performance improvement possible by modifying the MSRPC runtime and/or the

[1] Rajesh S. Madukkarumukumana implemented the RPC transport at Oregon Graduate Institute of Science & Technology (OGI). Access to MSRPC source was provided under Microsoft Windows NT^{TM} source code agreement between Microsoft and Oregon Graduate Institute for educational research purposes.

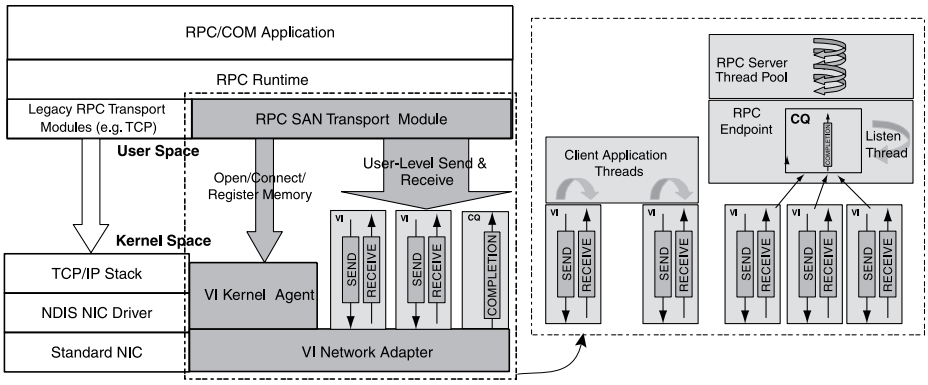

Fig. 8. MSRPC Operational Overview

marshaling engines was traded-off for application transparency and simplicity. These design choices were further motivated by the fact that the legacy network protocol stack contributed to the most part of the standard RPC latency. To co-exist with other transports, the SAN transport supports a new RPC protocol sequence (ncacn_san_vi) and is registered with the system using the standard WinNT^{TM} registry. Our transport uses a connection-oriented design for performance and utilizes the higher reliability modes offered by SANs. The transport module performs various operations like static/dynamic endpoint mapping, listening for connections, data transfers, connection tear-downs, etc. The transport module also provides the RPC runtime system information like the MTU of the transport and the buffer size needed to manage each endpoint and connection.

Endpoint Mapping In response to the RPC binding calls from the application, the MSRPC runtime calls the transport module to setup static and dynamic endpoints. The RPC transport takes machine name as the network address and accepts any string as an endpoint. The endpoint strings are used as discriminators in connection establishment between VIs. Dynamic endpoints use universally unique identifiers (UUIDs) as discriminators. On the server side, the receive queues of all VI connections created under an RPC endpoint are attached to a Completion Queue (CQ). The send queue for each VI connection is managed independently. The completion queue provides a mechanism to poll or wait for receive completions from multiple VIs associated with a given endpoint. This is equivalent to a traditional select functionality, but has the distinct advantage that the receive operation (after the select) does not need a kernel transition to de-queue the completed receive buffer. Since each send queue is managed separately, send operations can be performed independently on any connection. Connection requests from the RPC runtime on behalf of any client application thread is mapped directly to a unique VI. The client side does not use completion queues (CQs), but manages the receive and send queues for each

connection independently. The connection management and flow control design are discussed next.

Connection Management and Flow Control Today, most SANs provide a message-oriented reliable transport service, but do not provide flow control or support for out-of-band data. In addition to this, user-level networking architectures like VI Architecture require explicit registration of communication buffers. Compared to the centralized in-kernel implementation of flow control and buffering in legacy transport protocol stacks (TCP/IP), the RPC transport for SAN uses a de-centralized approach for user-level connection management and flow control. Upon endpoint creation request from the server side RPC runtime, the transport creates a CQ and registers the needed receive buffers and descriptors. The endpoint accepts connection requests by creating/binding VI handles with pre-posted receive buffers. After connection setup, a three-trip protocol between the client and the server-transports negotiates the MTU and the available credits on either side. The server-side RPC runtime is notified of newly accepted connections by using callbacks.

Similar to stream sockets over VI Architecture, the credit based flow control scheme uses three types of messages: *Data* (contains requests/replies from MSRPC runtime), *RTS* (sent to get more credits from receivers), and *CTS* (sent in response to an *RTS* message). The flow control information consists of packet sequence number, number of available credits and the sequence number of the last received packet. The transport encodes the flow control information into the *ImmediateData* field of VI descriptors. The sender is allowed to send a data packet only if there are enough credits available. If enough credits are not available, more credits can be requested by sending an RTS request message and waiting for a CTS reply message. An automatic RTS/CTS credit synchronization happens whenever the number of send credits goes below low watermark. Due to lightweight nature of this RTS/CTS scheme and the efficient re-use of receive buffers, automatic credit update schemes were not considered. The service-oriented abstraction and synchronous nature of RPC keep the buffer management much simpler than stream sockets where receive buffers are asynchronously consumed by the application.

Buffer Management and Data Transfer The RPC transport achieves zero-copy sends of marshaled buffers passed from the RPC runtime by pinning them first if needed, then fragmenting and queuing it on the send queue of the appropriate VI. Polling is used to complete send operations without kernel transitions. To avoid the costly registration and de-registration of buffers on each send operation, a cache of previously marshaled buffers is maintained in LRU fashion. Since RPC runtime itself uses cached buffers to marshal data on a per connection basis, a large percentage of cache hits was observed in the experiments. Any small-sized send buffer (typical in RPC) that misses this cache is copied to a pre-registered send buffer before posting the send descriptor.

The RPC transport interface supports two types of receive semantics for network I/O: *ReceiveAny* and *ReceiveDirect*. The *ReceiveAny* interface uses a single thread at a time to service a set of clients connected to an endpoint. In legacy transport protocols such as TCP/IP, the *ReceiveDirect* path dedicates a thread per connection and saves an extra kernel call compared to the *ReceiveAny* path. The use of CQs and user-level receive operations in our transport allows efficient implementation of *ReceiveAny*. This reduces the performance advantages of additionally implementing *ReceiveDirect*. Both the client and server perform a single copy of data from the receive buffers to RPC runtime buffers. The RPC runtime is called back to do any re-allocation of the marshaled buffer for re-assembly of multiple fragments. After copying the message, the de-queued receive descriptors are re-posted immediately on the receive queue to keep the flow-control credits fairly constant.

The use of reliable delivery VIs (as discussed in Section 3.1), efficient caching of marshaled send buffers, zero-copy user-level sends and single copy receive completions in the RPC transport contributes to the high performance achieved. In addition, to enable DCOM applications to transparently run over SANs, the DCOM Service Control Manager (SCM) in WinNT^{TM} was modified to listen on the new RPC protocol (in addition to other legacy protocols) for COM object invocation requests. The performance analysis of RPC and DCOM applications over the SAN transport is discussed next.

4.3 Experimental Results

In order to evaluate the RPC and DCOM performance improvements over the user-level RPC transport, a set of experiments was carried out using the same experimental setup described in Section 3.2. All DCOM and RPC latency measurements used bi-directional conformant arrays as method parameters. Figures 9 and 10 compare the round-trip RPC and DCOM method call latencies (averaged over 10000 runs) across various RPC transports respectively. On the same SAN, RPC and DCOM over VI Architecture achieve significantly (2-3x) better latency than RPC and DCOM over legacy network protocols (TCP/IP). Furthermore, application-to-application latency achieved by RPC and DCOM is comparable to the latency achieved by local RPC (LRPC) and COM.

RPC and distributed object abstractions are becoming norm to build multi-threaded client/server applications. While several previous research efforts [1, 3, 11, 17] have shown the performance improvement possible through efficient thread and function dispatching in specialized environments, our research focuses more on transparently improving the performance of commercial multi-threaded implementation like MSRPC.

The RPC transport implementation creates a new VI connection to the server endpoint for each client application thread using the same RPC binding handle. The de-centralized user-level protocol processing done independently on a per connection basis eliminates major synchronization and multiplexing overheads otherwise required for buffer management in the transport. Figure 11 shows the scalability achieved on a 4-way SMP system with multiple application threads

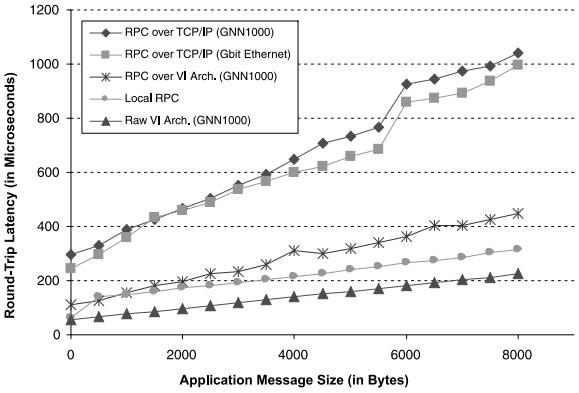

Fig. 9. Round-Trip Latency

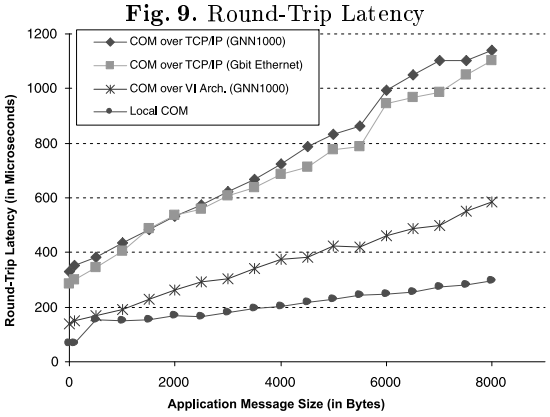

Fig. 10. DCOM Round-Trip Latency

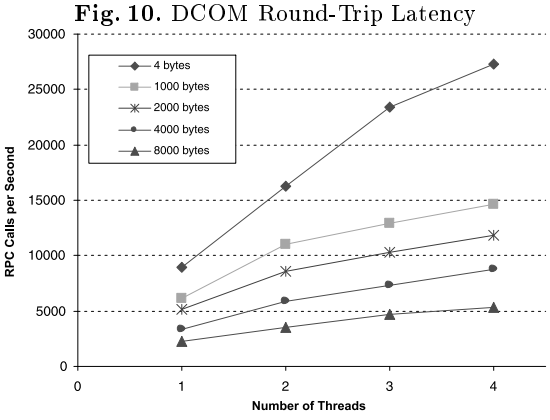

Fig. 11. SMP Scalability

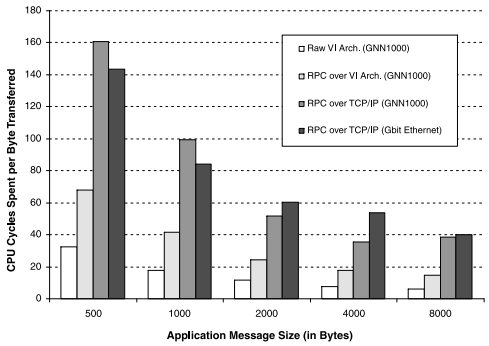

Fig. 12. CPU Overhead

using a single RPC binding handle over the RPC transport. This demonstrates the efficiency of credit based flow control and buffer management. Figure 12 shows that the host CPU cycles spent per byte transferred using the high performance RPC transport is substantially lower than using legacy transport protocol (TCP/IP) over the same SAN and Gigabit Ethernet. Table 2 summarizes 4-byte round-trip latencies and 8000-byte round-trip latencies achieved by various RPC transports.

Table 2. MSRPC Performance Across Various Transports

Messaging Layer	**4-byte Round-trip Latency (in μs)**	**8000-byte Round-trip Latency (in μs)**
Local RPC (LRPC)	62.7	314.8
RPC over VI Architecture (GNN1000)	111.21	448.1
RPC over TCP/IP (GNN1000)	297.1	1039.7
RPC over TCP/IP (Gigabit Ethernet)	243.5	995.8

5 Related Work

Damianikis et al. [4] described the implementation of high performance stream sockets compatible abstraction over virtual memory mapped communication (VMMC) in the SHRIMP multi-computer using a custom designed network interface. Thorsten von Eicken et al. [7] showed how traditional protocols (TCP/IP, UDP/IP) can be efficiently supported over U-Net (a user-level networking architecture). Fast sockets [14] were implemented to provide low overhead protocol layer on top of high performance transport mechanism (active messages). Pakin et al. [13] implemented user-level sockets library over low-level FM messaging

layer. In all of these previous efforts, the focus was on to build efficient socket abstractions using custom low overhead messaging layer. In this paper, we add-on to their findings by developing techniques for efficient stream sockets implementation over standard user-level networking architecture (VI Architecture).

Muller et al. [11] showed how specialization techniques like partial evaluation can be applied to improve RPC performance. Cheriton et al. [17] proposed a specialization methodology for applications to modify and tune the RPC system to meet specific requirements. While these techniques are possible in custom application environments, our focus in this research is to transparently improve performance of applications using commercial RPC systems. Bilas and Felten [1] described a SunRPC implementation over the SHRIMP multi-computer, but concentrated on achieving the best absolute performance in single threaded environments. Zimmer and Chien [18] described in detail the impact of inexpensive communication to MSRPC performance by implementing a RPC datagram transport over Fast Messages (FM). Their implementation exposed the pessimistic assumptions made by the MSRPC runtime about datagram transports. Chang et al. [3] proposed a software architecture for zero-copy RPC in Java across user-level network interfaces. Their work showed interesting RPC latency improvements, but required manual generation of proxy/stub code. Our user-level RPC transport's connection-oriented design and credit based flow control utilizes reliable delivery mode offered by SANs to achieve significant performance improvements transparently.

6 Conclusions and Future Work

User-level networking architecture like VI Architecture provides low level primitives for high performance communication over SANs. Building high-level scalable distributed applications require this type of communication performance without sacrificing ease of programming. This paper demonstrates how high level communication abstractions like stream sockets and RPC can be efficiently implemented over VI Architecture. Our prototype implementations achieve significantly better performance (3-4x bandwidth improvement for stream sockets, 2-3x latency improvement for stream sockets, MSRPC, and DCOM) over legacy network protocols (TCP/IP) on the same SAN. We are currently investigating variants of credit based flow control and optimized large data transfers with RDMA operations provided by VI Architecture. A possible direction for future work is to extend support for overlapped and asynchronous operations in stream sockets and RPC over VI Architecture. Another interesting direction is to experiment with high level RPC and distributed objects based applications such as transaction processing monitors, web servers, and N-tier applications over VI Architecture.

Acknowledgements

We would like to thank many people at Intel corporation especially Natalie Bates, Ellen Deleganes, Chris Dodd, David Fair, Ed Gronke, Roy Larsen, Dave

Minturn, George Moakley, Wire Moore, Justin Rattner, Greg Regnier, and Brad Rullman for their helpful suggestions and support.

References

1. Angelo Bilas and Edward W. Felten: Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. J. of Parallel and Distributed Computing. **40(1)** (1997) 138–146
2. M. Blumrich et al.: A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In Proceedings of the 21st Annual Symposium on Computer Architecture. (1994) 142–153
3. Chi-Chao Chang and Thorsten von Eicken: A Software Architecture for Zero-Copy RPC in Java. CS Technical Report 98-1708, Cornell University. (1998)
4. Stefanos N. Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. In Proceedings of 1st International Workshop on Communication and Architectural Support for Network-Based Parallel Computing. (1997)
5. DCOM Architecture. Microsoft Corporation. (1997)
6. Dave Dunning et al.: The Virtual Interface Architecture: A Protected, Zero Copy, User-level Interface to Networks. IEEE MICRO. **18(2)** (1998) 66–76
7. Thorsten von Eicken et al.: U-Net: A User-Level Network Interface for Parallel and Distributed Computing. Proceedings of the 15th ACM Symposium on Operating System Principles. (1995) 40–53
8. GigaNet Incorporated. GigaNet cLAN Product Family. http://www.giganet.com/products.
9. Java^{TM} Remote Method Invocation Specification. ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.
10. Rajesh S. Madukkarumukumana, Calton Pu, and Hemal V. Shah: Harnessing User-level Networking Architectures for Distributed Object Computing over High-Speed Networks. In Proc. of 2nd USENIX Windows NT Symposium. (1998) 127–135
11. Gilles Muller et al.: Fast, Optimized Sun RPC Using Automatic Program Specialization. In Proceedings of the International Conference on Distributed Computing Systems (ICDCS-18). (1998) 240–249
12. Object Management Group: The Common Object Request Broker: Architecture and Specification. 2.0 Edition. (1995)
13. Scott Pakin, Vijay Karamcheti, and Andrew A. Chien: Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. IEEE Concurrency. **5(2)** (1997) 60–73
14. Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler: High-Performance Local Area Communication with Fast Sockets. USENIX 1997 Conference. (1997)
15. Virtual Interface Architecture Specification. Version 1.0. http://www.viarch.org/. (1997)
16. Windows Sockets 2 Service Provider Interface. ftp://ftp.microsoft.com/bussys/winsock/winsock2/.
17. Matthew J. Zelesko and David R. Cheriton: Specializing Object-Oriented RPC for Functionality and Performance. In Proceedings of the International Conference on Distributed Computing Systems (ICDCS-16). (1996)
18. Oolan M. Zimmer and Andrew A. Chien: The Impact of Inexpensive Communication on a Commercial RPC System. Concurrent System Architecture Group Memo, University of Illinois at Urbana-Champaign and University of California at San Diego. (1998)