A Portable Client/Server Communication Middleware over SANs: Design and Performance Evaluation with Virtual Interface and InfiniBand[1]

J. Liu M. Banikazemi[†] B. Abali[†] D. K. Panda

Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
email: {liuj, panda}@cis.ohio-state.edu

[†]Server Technology
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
email: {mb, abali}@us.ibm.com

***Abstract*— In this paper, we address the problem of using high performance SAN technologies to provide efficient communication support in client/server type of environments. Our objective is to build a portable substrate which not only bridges the functionality gap between applications and SAN networks and preserves the performance of the underlying communication layer, but also provides the required flexibility for analyzing the impact of various modern communication techniques. We propose an architecture which exposes a simple queue based interface to applications. We take advantage of the features of modern SANs (such as OS bypass, zero copy, and RDMA) and use several techniques to improve the performance of our communication protocol. We have implemented our design for both InfiniBand and Virtual Interface Architecture (VIA). Our performance evaluation shows that the protocol processing overhead is low and the underlying communication performance can be preserved. We also discuss the performance impact of various communication techniques.**

I. Introduction

During the last ten years, the research and industry communities have been proposing and implementing user-level communication systems to address some of the problems associated with traditional networking protocols [1]–[3]. These communication subsystems form System Area Networks (SANs) that provide very high communication performance and very low host CPU overhead. The Virtual Interface Architecture (VIA) [4] was proposed to standardize these efforts. Recently, InfiniBand Architecture (IBA) [5] has been introduced which combines storage I/O with Inter-Process Communication (IPC).

Although SAN networks such as those connected by InfiniBand offer superior raw performance, for using them many communication issues such as descriptor management, buffer registration and flow control should be dealt with. Furthermore, different SANs have different programming interfaces, making it difficult to write portable software. There are ongoing efforts [6], [7] trying to address the portability problem by providing unified programming APIs, but the APIs such as those defined in DAT [6] still present a low level interface and the users have to take care of the communication details. Several groups in academia and industry have addressed this problem by implementing well known standards and APIs [8], [9] and developing new middleware [10]. Furtheremore, several standards [11], [12] are being developed in order to take advantage of features provided in modern SANs. There are also a few applications that have been developed directly over the system area communication protocols. This approach has been used to construct web servers and database storage systems [13], [14]. In this approach, more development effort is needed to handle all the communication details and the software may not be portable.

As we can see, it would be beneficial if we

[1]The research was started during Jiuxing Liu's visit to IBM T. J. Watson Research Center as a summer intern. It is supported in part by DOE grant #DEFC02-01ER25506, Sandia National Laboratory contract #30505 and NSF grants #EIA-9986052 and #CCR-0204429.

can provide a portable communication middleware which hides the communication details. Many SANs offer features such as OS bypass, zero copy, and Remote Direct Memory Access (RDMA). In this paper, we explore how to use these features to provide efficient communication support for cluster-based database and web servers systems. Our objective is to build a portable substrate which not only bridges the functionality gap between various applications and SAN networks and preserves the performance of the underlying communication layer, but also provides the required flexibility for analyzing the impact of various modern communication techniques. By design and implementation of such a communication system we can evaluate the performance of SANs in a more realistic way.

We propose an architecture which exposes a simple but flexible communication interface to applications. In designing this architecture, we use client/server type of environments as our primary target environments. In such systems, communication operations are primarily used for transferring request and response messages between clients and servers. Besides hiding communication details and providing flexibility, the proposed architecture also achieves high performance. It does so by using the following:

- Carefully designed interfaces. Our design consists of two interfaces, which provide both required functionality and portability.
- Communication protocol optimization. We use many techniques, such as Eager Write and Pin-down Cache, to improve communication performance.
- Flexible mechanisms to ensure communication progress. Instead of forcing one particular progress method, we have provided a mechanism upon which multiple ways can be implemented to make communication progress.

We have implemented prototypes of our design on both InfiniBand and VIA. Our performance evaluation shows that the protocol processing overhead is low and thus the underlying communication performance can be preserved. We have also quantified the performance impact of several protocol optimizations and communication progress methods.

The rest of this paper is organized as follows: In Section II we describe the basic architecture of our proposed middleware. Design issues are discussed in Section III and performance results are presented in Section IV. Conclusions are presented in In Section V.

II. Basic Architecture

The basic structure of our proposed middleware is shown in Figure 1. It consists of two software layers: Message Processing Layer and SAN Adaption Layer. These two layers together hide the communication details from the upper layer software. The communication supported by our middleware can be one-sided (no reply needed) or client/server type of messages.

There are two interfaces in the architecture: Message Interface and Low Level Communication Interface. The Low Level Communication Interface is a set of well defined functions which provide functionalities similar to VIA functions and InfiniBand Verbs. The SAN Adaption Layer makes our substrate portable among different SAN interfaces such as VIA, DAT and InfiniBand by hiding the difference between them and adapting them to the common Low Level Communication Interface. The other interface is the Message Interface. Upper layer software submits communication requests (both one-sided and client/server type of messages) through the *Primary Queues*. Unexpected incoming messages (request coming from other nodes) can be fetched from *Unexpected Queues*. In a client/server environment, Primary Queues are used by clients to submit requests and Unexpected Queues are used by servers to process incoming requests. However, our middleware provides a unified interface for both cases and therefore can also be used in a peer-to-peer environment.

Primary and unexpected requests have similar data structure which is shown in Figure 2. They both consist of three parts: Control Segment, Request Data Segments and Reply Data Segments. However, their Control Segments are different. Both Request Data Segments and Reply Data Segments may contain multiple Address Segments. An Address Segment describes a chunk of contiguous virtual memory. The memory in an Address Segment can be at the local host or a remote node. In the later case, the Address Segment will contain necessary information for accessing the memory remotely.

For a primary request, the Request Data Segments describe the locations of request data and the Reply Data Segments describe where we expect the remote node (server) to send the respond data to. For an unexpected request, the Request Data Segments describe where the request data is (local or remote) and the Reply Data Segments describe where the response data should be put.

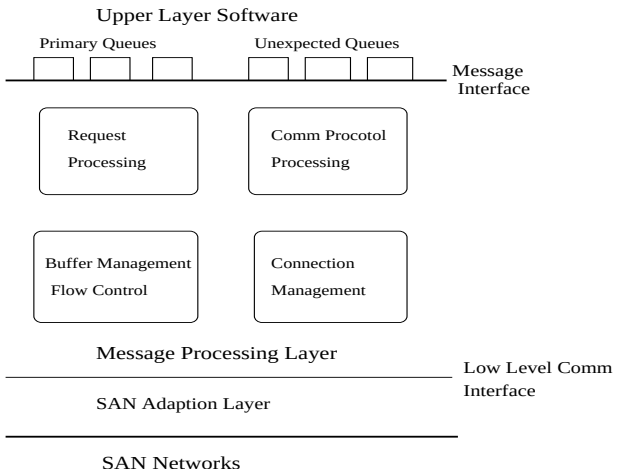

Fig. 1. Basic Architecture

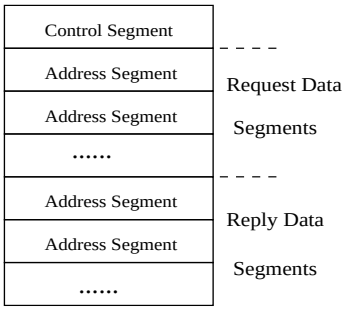

Fig. 2. Request Data Structure

To use the Message Interface, a client can follow these steps:

```
/* build a request */
...
/* submit the request */
submit_request(&request, &handle, service_id);
...
/* later, check the status */
check_request(handle, &status);
```

On the server side, it is slightly more complicated because the server needs to check if the data is in local memory or not. Thus, the server will follow the following steps:

```
/* get a request */
get_request(&request_in);

/* check the data */
if (request_in.seg[0].type == REMOTE) {
/* use an RDMA read to fetch request */
...
}

/* process request */
...
/* build reply and submit */
...
submit_request(&reply, &rhandle, client_id);
...
```

Most of the functionality provided by our substrate lies in the Message Processing Layer. This layer consists of several components such as Reqeust Processing, Communication Protocol Processing, Buffer and Flow Control Management and Connection Management. These components take care of many communication details and thus the upper layer software developers do not need to worry about them. In spite of the functionality provided by this layer, we can still keep the overhead low by using a number of techniques such as optimization for small data segments, pin-down cache, sender managed RDMA write buffer, etc. These techniques will be discussed in detail in the next section.

III. Design

In this section we focus on the design issues related to the communication protocol in Message Processing Layer. This layer also handles many other issues including thread safety, locking, queue management, flow control and connection management.

A. Communication Protocols

The most important function of the Message Processing Layer is to move data between the request node and the server node by using the communication service provided by SAN networks. The basic idea of our design is to use send/receive operations

for control messages and RDMA read or RDMA write operations for data transfers. One issue in zero-copy communication is that communication buffers must be pinned (registered). To reduce the overhead for sending and receiving control messages, we maintain a global pool of buffers which are already pinned for control messages.

The most frequently used communication pattern in our target applications consists of a request and a reply. Thus we optimize our communication protocol for handling this case.

1) Basic Protocol: To simplify the explanation, we assume that the client has submitted a request which has one request data segment and one reply data segment. The basic communication protocol to handle this request is shown in Figure 3. It involves the following steps:

1) At the client side, the client submits the request to the Primary Queue.
2) The Message Processing Layer dequeues the request and begins processing. First, it checks the request and reply data segments and pins them down in memory.
3) The Message Processing Layer then builds a control packet which contains information about the request, including the request and reply data segments. It sends out the packet to the server by using the funtions provided by the Low Level Communication Interface.
4) The Message Processing Layer at the server receives the control packet. It builds an unexpected request data structure and puts it into the Unexpected Queue. The address segments in the packet are translated and copied to the request data structure. Note that the request data is still at the client node.
5) The server application, which is upper layer software with regard to the Message Processing Layer, dequeues the request from the Unexpected Queue and begins serving the request. First, it checks the address segments and notices that the data is still at a remote node.
6) In order to get the request data, the server application builds a special RDMA read request and submits it to the Primary Queue. Before that, it also allocates a local memory buffer and specifies it to be the destination of the RDMA read request.
7) The Message Processing Layer at the server node processes the RDMA read transaction. It pins down the local destination buffer and issues an RDMA read operation through the Low Level Communication Interface.
8) After the RDMA read operation is finished, the local destination buffer is unpinned and the RDMA read request is marked as completed.
9) The server application processes the original request. It then submits a reply to the Primary Queue.
10) The Message Processing Layer at the server pins down the reply data buffer and uses RDMA write operation to transfer data from the server to the client. Then it sends a control packet to the client indicating that the original request is finished.
11) After the RDMA write operation is finished, the reply data buffer is unpinned at the server.
12) At the client side, the control packet is received. The buffers specified in the request and reply data segments are unpinned, and the original request is marked as complete.

Our substrate using this basic protocol has several advantages. First, the upper layer software is not involved in many communication operations such as buffer pinning and unpinning. These operations are carried out in the Message Processing Layer. Second, there are no extra data copies during the communication.

However, there are also several disadvantages in using the above protocol. First, it involves lots of buffer pinning and unpinning. These operations go through the kernel and usually involves some interaction between the host and the NIC. Thus, they have a quite large overhead. Another problem is that the server needs to use another RDMA read request to get the request data explicitly. This increases the latency in two ways: First, the server has to wait for the RDMA read request to finish before it can serve the request; Second, the control must be transfered between the upper layer software and the Message Processing Layer. Finally, the basic protocol uses RDMA read to get the request data. However, this operation is not available in all SAN networks. VIA, for example, specifies that RDMA read is optional.

In order to address these issues, we have proposed several enhancements/optimizations to the basic protocol.

2) Small Data Transfer: As we have mentioned, it is not efficient to use RDMA operations to transfer small data because of the buffer pinning and unpinning overhead. To address this problem, we attach small request data to the end of control packets. Similarly, the server can attach its reply data with the reply packet, and the data will be copied to the reply data buffer when the packet arrives at the request node (Figure 3). This optimization is only turned on for small data buffer. Since copying overhead is negligible in this case, this technique increases communication performance.

3) Pin-down Cache: Although pinning and unpinning overhead for small buffers can be avoided using the previous technique, for large buffers the overhead can still be significant. To deal with this, we use the pin-down cache technique, which was first proposed in [15]. The main idea is to maintain a cache of pinned buffers. When a buffer is first pinned, it enters the cache. When the user wants to unpin the buffer, the actual unpin operation is not carried out and the buffer stays in the cache. Thus the next time when the buffer needs to be pinned, we need not to do anything because it is already pinned. A buffer is unpinned only when it is evicted from the cache.

The effectiveness of pin-down cache depends on how often buffers are reused. In many applications, buffer reuse rate is high because of temporal locality. Therefore, we expect that pin-down cache will improve communication performance in most cases.

4) Eager Write for Request Data: Another disadvantage in the basic protocol is that the server needs to get the request data explicitly by using an RDMA read request and this increases latency. By using our small buffer optimization, the overhead can be avoided for small request data. But the problem still exists if the request data size is large.

To address this problem, we use a sender managed RDMA write buffer pool. This scheme works as follows: First the receiver allocates a number of buffers and pins them down. Then the receiver puts information about the buffer into a control packet and sends it to the sender. When the packet arrives, the sender processes it by putting these buffers into a free list. Later when the sender sends out a request, it can get a buffer from the free list and use RDMA write to send the data directly. After the request is finished (a reply is returned from the server), the buffer can be returned to the free list. The detailed steps of the communication protocol are shown in Figure 4.

By using this technique, the overhead incurred by the extra RDMA read request can be avoided in all cases except when we run out of RDMA write buffers in the free list. If the server can process the buffer in place, zero copy can still be achieved.

5) Protocol Selection: For sending out a request, we have three protocols which use small data optimization, eager write and RDMA read, respectively. If a request contains multiple request data segments, the protocol is determined separately for each data segment. The selection of protocols is based on data size. If data can fit in a control packet, small data optimization is used. If data size is less than the size of RDMA write buffer, eager write may be used. Otherwise, RDMA read is needed to get the request data segment.

6) RDMA Read Emulation: To address the problem that some SAN networks do not have RDMA read support, we introduce a technique called RDMA Read Emulation in the SAN Adaption Layer. By doing this in the SAN Adaption Layer, the Message Processing Layer can always assume that RDMA read operation is available. The emulation is done by using control messages and RDMA write operation. The steps are shown in Figure 4. The performance of RDMA read emulation is not as good as native RDMA read. However, we can void RDMA read by using eager write protocol. Another point we should note is that the performance of RDMA read emulation also depends on how fast the remote side can respond to the RDMA read request packet. This in turn depends on how fast the communication can make progress on the remote side. We will talk about communication progress in the next subsection.

B. Communication Progress

Communication progress is an important issue in a communication subsystem. Since the processor must handle both processing and communication, care must be taken so that the communication can

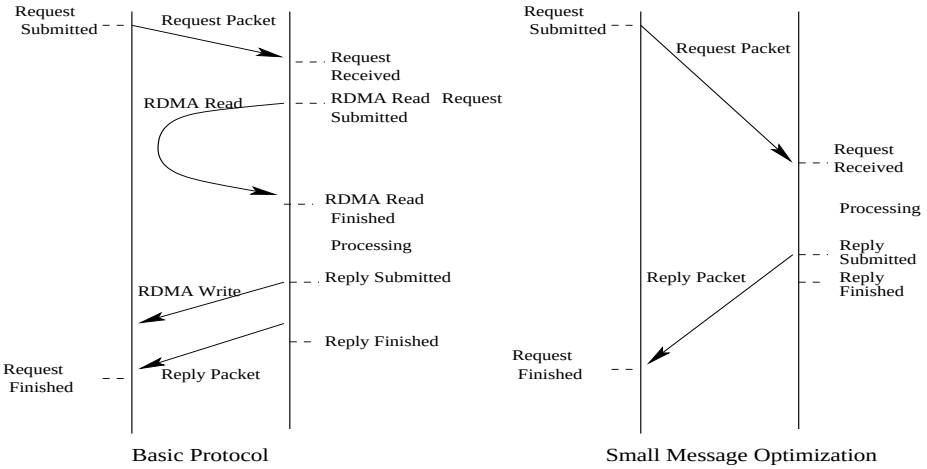

Fig. 3. Basic Protocol and Small Message Optimization

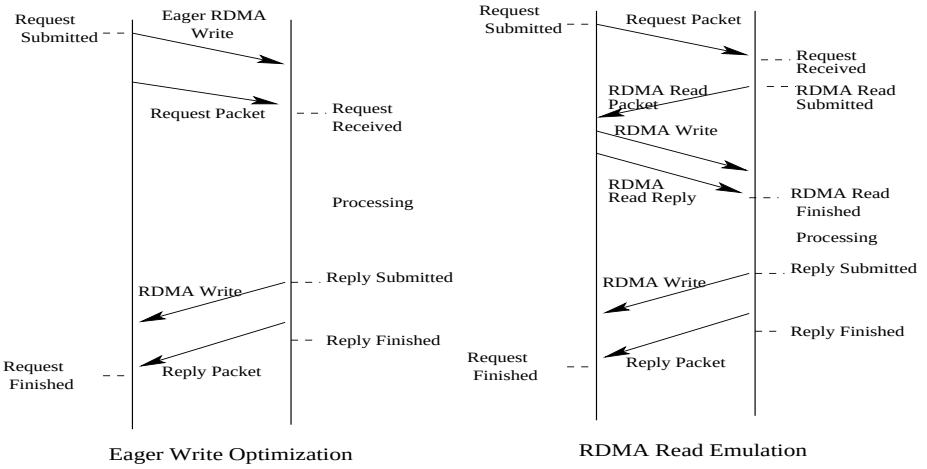

Fig. 4. Eager Write Optimization and RDMA Read Emulation

get enough CPU cycles to make progress. SAN networks make communication progress easier by offloading some of operations to the NIC. However, host processing is still needed. Since our target environments are client/server environments, communication progress is even more important. Because it not only affects the local host, but also has an impact on the performance of the remote nodes that depend on the local host as the server node. There are different ways to ensure communication progress, and different application may prefer one of them over others. To provide enough flexibility to the upper layer software, we don't enforce a single way to make communication progress. Instead, we provide a mechanism upon which multiple methods to make communication progress can be easily built. We encapsulate all the processing needed for communication into a single function. This progress function basically polls for events from multiple event sources as in Figure 5. Based on this progress function, different approaches can be used to make communication progress:

- Single thread polling approach. In this approach, the application is single threaded and it periodically calls the progress function to make progress.
- Dedicated communication thread approach. In this approach, the application is multithreaded. One of the threads is dedicated to calling the progress function.
- Multi-thread polling approach. In this approach, the application is multi-threaded. All threads can call the progress function to make communication progress.

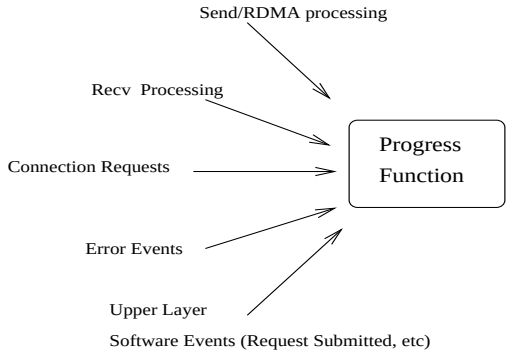

Fig. 5. Progress Function

Later in the performance section we will evaluate the impact of different progress approaches.

C. Other Issues

In this section we talk about a couple of other issues involved in our design, including thread safety, queue management, flow control and connection management.

1) Thread Safety and Locking: To take advantage of SMP machines, many server applications are multi- threaded. Thus, it is important to provide thread safety in our Message Processing Layer. In our design, how this is handled depends on the method used to make communication progress. If single thread polling approach is used, we do not need to worry about thread safety because there is only one thread. In the dedicated communication thread approach, the communication thread and other threads interact through Primary Queues and Unexpected Queues. So these queues are protected by locks. In the multi-thread polling approach, Primary Queues and Unexpected Queues are also protected by locks. Besides that, the progress function is also wrapped by a lock because this function may modify some global communication data structure which must be protected from simultaneous accesses.

We should note that locking is provided by the Message Processing Layer. The upper layer software does not need to worry about this when using our communication service.

2) Queue Management: As mentioned before, in multi-threaded applications Primary Queues and Unexpected Queues are protected by locks. Contention exists when more than one thread access a single queue at the same time. To reduce this contention, the number of these queues can be chosen at compile time. For example, if the number of queues is the same as the number of application threads, there will be no contention among them if they access different queues because each queue is protected by a different lock.

Currently the Primary Queues are processed in a round-robin fashion. However, the processing can be easily changed to give priority to certain queues and thus implement some kind of QoS for message processing.

3) Flow Control: SAN networks such as VIA or InfiniBand often require that a receive be posted to a communication end-point before the other side can successfully send a message. Since RDMA operations are one-sided, they do not have this limitation. In our communication protocol, control packets are transferred through send/receive operations. Thus, we have to provide a flow control scheme so that we do not overrun the receive side with too many packets.

This task is made easy be observing the fact that for Primary Queues, each request, including emulated RDMA read reqeusts, will only generate a limited number of control packets to the server node. Thus, by limiting the number of outstanding transactions and reposting receives at the server side before sending back responses, we can make sure that buffer overruns will not happen.

4) Connection Management and Name Service: SAN networks often use connection oriented communication model. (InfiniBand also provides datagram services.) Therefore, a connection must first be established before two nodes can communicate. This work can be done statically or dynamically. In the former case, all connections are set up during the initialization phase. In the latter approach, nodes can join dynamically and connections are set up on demand.

In our design, upper layer software uses high level names (service names or server names) to specify the target node when accessing our communication services. The high level names are translated by our Message Processing Layer to low level names which can be mapped directly to the communication end points. Each node provides this service by maintaining a name translation table. However, it is possible to use existing directory service to provide

this name service.

IV. Performance

We have implemented prototypes of our design for two SAN networks connected by Giganet cLAN 1000 VIA [16] and Mellanox InfiniHost 4x InfiniBand [17] cards. Each node of our IBA cluster has 512MB memory and dual Intel Xeon 2.40 GHz processors with Hyperthreading, giving the illusion of 4 processors. Each node of our VIA cluster has 1GB memory and four Intel PIII 700 MHz processors. The Linux kernel versions are 2.4.18 for IBA and 2.2.4 for VIA.

Please note that the IBA performance results presented here are based on early microcode release for InfiniHost. The Mellanox InfiniHost HCA SDK build id is thca-x86-0.0.4-rc4-build-001. The adapter firmware build id is fw-23108-1.13.1016_2-build-001. As the InfiniBand products are rapidly maturing, we expect to see upcoming firmware releases to continue reducing latency and increasing bandwidth. This will have direct performance impact on our middleware.

A. Base Performance

Figure 6 shows the base latency (one way) for IBA and VIA. From the figure we can see that for IBA, RDMA write operation has better latency than send/receive operation. But for VIA, latencies for send/receive and RDMA write are almost the same and thus we only show one line in the figure.

Figures 7 and 8 show the base latency for our implementation and Figures 9 and 10 show the base throughput. All tests were carried out between two nodes: a client and a server. In the latency tests, the client sends out a request and wait for it to finish (until the corresponding reply has arrived). In the throughput tests, the client sends out requests as fast as it can and the server measures how fast it can handle the requests. The requests consist of one request data segment and one reply data segment. We have used two methods to make communication progress: single thread polling and dedicated communication thread. From the figures we can see that by using polling, the roundtrip latency for small request and reply messages is around 24 microseconds for IBA and around 28 microseconds for VIA. The roundtrip overhead of the implemented middleware is around 4 microseconds for IBA and around 8 microseconds for VIA. How to make communication progress also has an impact on throughput, especially for small messages. From Figures 9 and 10 we can observe that using communication thread degrades performance for small messages significantly. For large messages the performance is almost the same for both cases.

To get more insight into where the time has been spent, we provide latency timing breakdowns for requests with 8 byte request and reply data in Tables I and II. We have broken up the latency into five major parts. Base overhead refers to the IBA or VIA latency for 0 byte messages. Header overhead refers to the time added by sending packets with a control header. Protocol processing overhead is the time spent in our Message Processing Layer. This overhead should be the same for both IBA and VIA. However, the actual time is different due to the fact that the nodes in our IBA cluster testbed are much faster. SAN adaption overhead is the time spent in the SAN Adaption Layer. From the tables we notice that IBA has larger overhead. This is due to the fact the current microcode release fro IBA is prelimilary. If we use a dedicated communication thread, the extra locking and context switching overhead is around 3 microseconds for the IBA cluster and around 6 microseconds for the VIA cluster.

B. Eager Write and Pin-down Cache Impact

Figures 11 and 12 show the latency for two cases: one always using eager write optimization and one always using RDMA read to get request data. Small message optimization is used in both cases. We can see that for large messages eager write optimization can reduce the latency. Figures 13 and 14 show the impact of pin-down cache on latency. Figures 15 and 16 show the impact of pin-down cache on throughput. In these tests, the request message size is such that it cannot use the small message optimization. From the figures we can see that using pin-down cache can significantly improve both latency and throughput for IBA. Buffer pinning and unpinning in cLAN VIA are quite efficient. Nevertheless, pin-down cache still brings noticeable performance improvements.

C. Impact of Service Time

In the previous tests, we have conducted the experiments in such a way that the server always replies immediately to any incoming requests. To better understand the interaction between service time (time spent processing the requests) and the way communication progress is made, we have conducted tests by simulating the service time. Two nodes were used in these tests: a server and a client. The client sends out requests as fast as it can. The server consists of one or more worker threads. Each worker thread tries to get an incoming thread, processes it for a certain time, and sends a reply. The processing is simulated by just keeping incrementing a counter for a certain period of time. The results are shown in Figures 17 and 18. We compared two methods for making communication progress: polling and dedicated communication thread. In the polling approach, there was only one thread while in the dedicated communication thread approach one communication thread and multiple worker threads were used. From the figures we can see that if service time is small, the overhead of executing multiple threads dominates and the polling approach performs better. However, as the service time increases, the multiple server threads approach is able to use multiple CPUs to process requests simultaneously and gives a better performance.

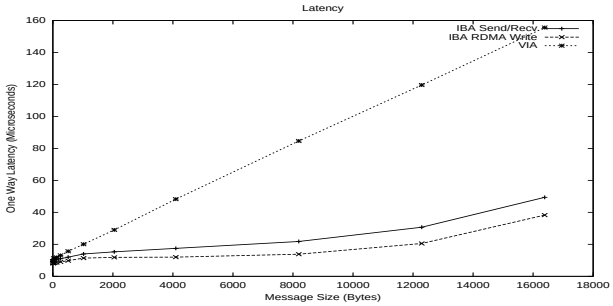

Fig. 6. Base Latency for IBA and VIA

In Figures 19 and 20, we show the results using dedicated communication threads with 3 and 4 worker threads. From the figures we notice that when the total number of threads exceeds the number of CPUs (the case with 4 worker threads and one communication thread), the throughput drops because the communication thread has to compete with worker threads for CPU time.

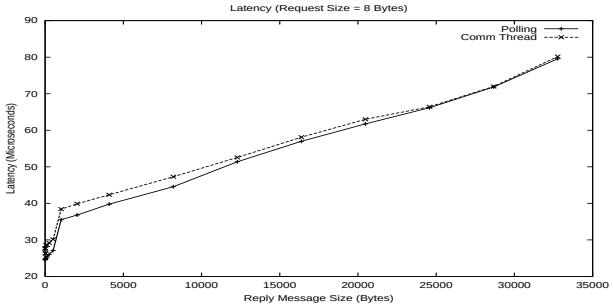

Fig. 7. Latency (Roundtrip) of the Proposed Middleware with IBA

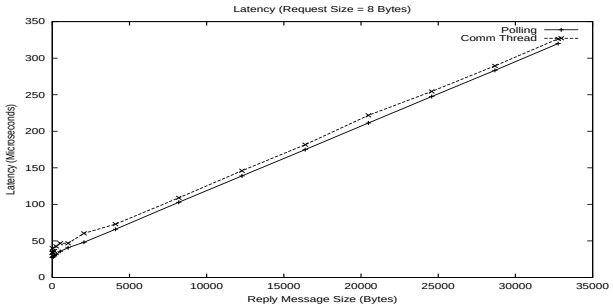

Fig. 8. Latency (Roundtrip) of the Proposed Middleware with VIA

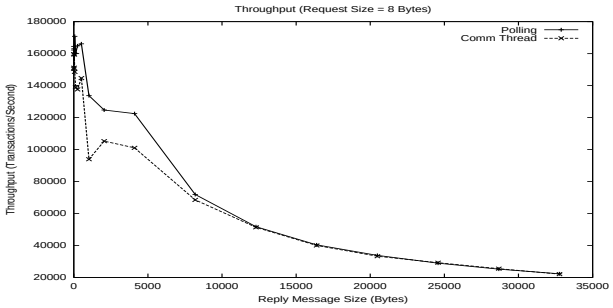

Fig. 9. Throughput of the Proposed Middleware with IBA

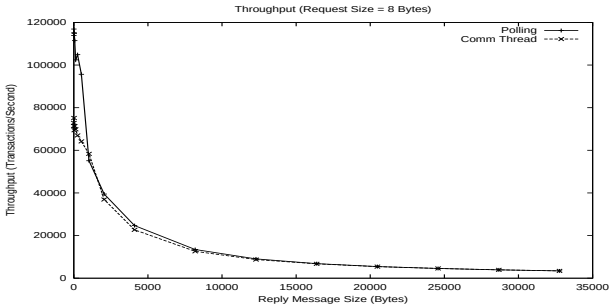

Fig. 10. Throughput of the Proposed Middleware with VIA

TABLE I

ROUNDTRIP LATENCY BREAKDOWN OF THE PROPOSED MIDDLEWARE FOR 8 BYTES REQUEST/REPLY WITH IBA

Components	Time (Microseconds)
Base Overhead	20.56
Header Overhead	0.70
SAN Adaption Overhead	2.41
Protocol Processing Overhead	0.77
Threading Overhead (Optional)	3.08

TABLE II

ROUNDTRIP LATENCY BREAKDOWN OF THE PROPOSED MIDDLEWARE FOR 8 BYTES REQUEST/REPLY WITH VIA

Components	Time (Microseconds)
Base Overhead	21.28
Header Overhead	4.86
SAN Adaption Overhead	0.41
Protocol Processing Overhead	1.30
Threading Overhead (Optional)	6.16

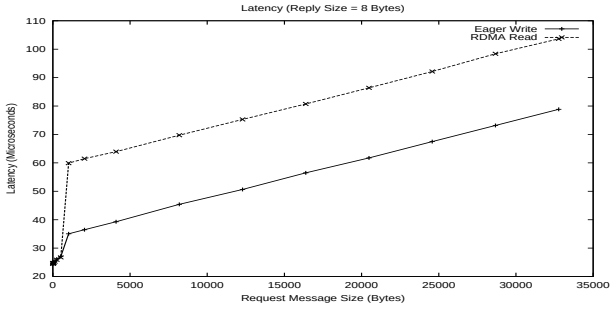

Fig. 11. Latency for Eager Write and RDMA Read with IBA

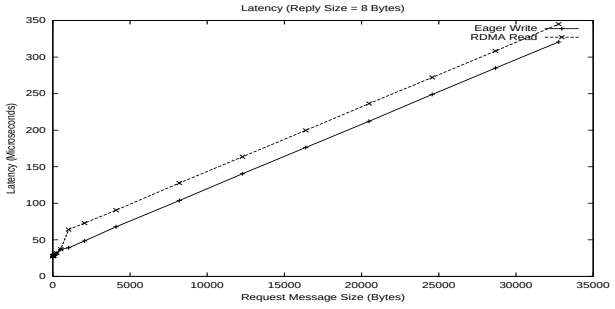

Fig. 12. Latency for Eager Write and RDMA Read with VIA

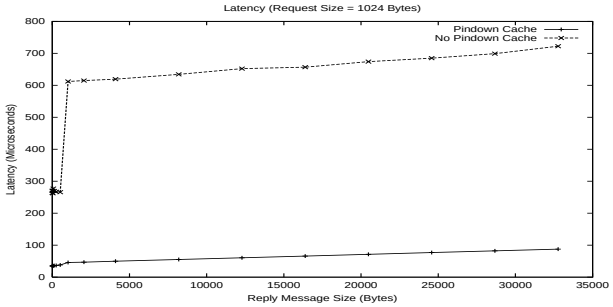

Fig. 13. Pin-down Cache Impact on Latency with IBA

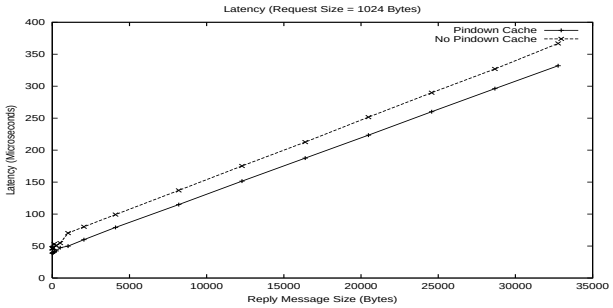

Fig. 14. Pin-down Cache Impact on Latency with VIA

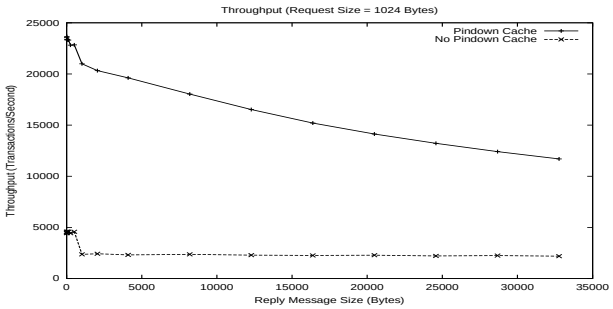

Fig. 15. Pin-down Cache Impact on Throughput with IBA

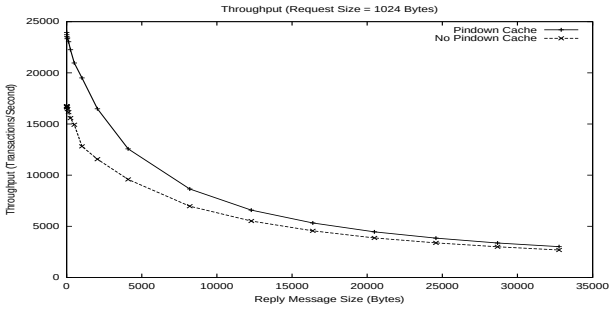

Fig. 16. Pin-down Cache Impact on Throughput with VIA

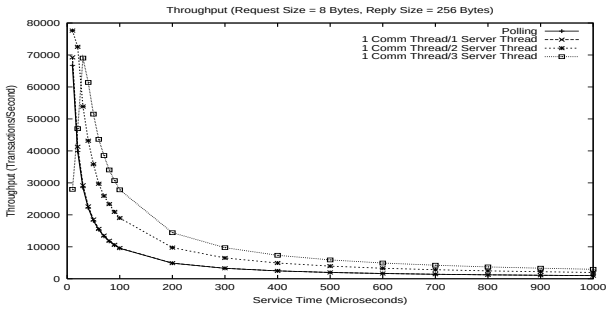

Fig. 17. Impact of Service Time on Throughput with IBA (1-3 Worker Threads)

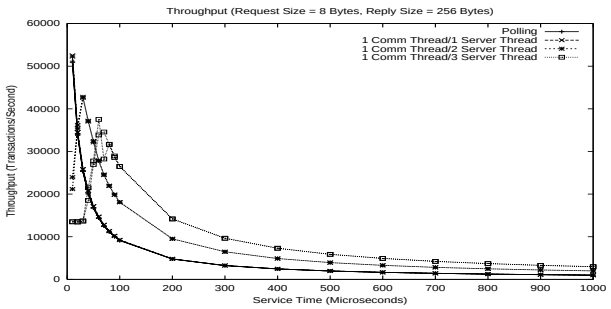

Fig. 18. Impact of Service Time on Throughput with VIA (1-3 Worker Threads)

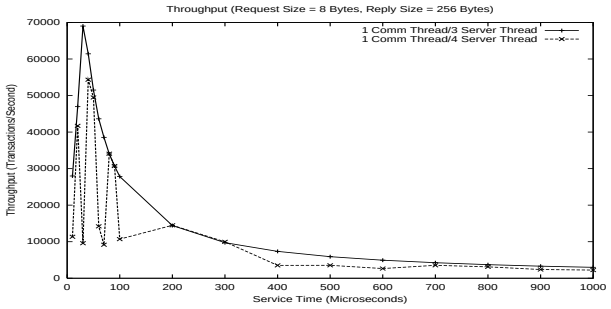

Fig. 19. Impact of Service Time on Throughput with IBA (3-4 Worker Threads)

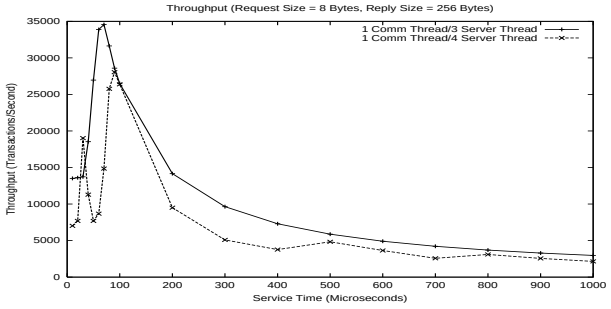

Fig. 20. Impact of Service Time on Throughput with VIA (3-4 Worker Threads)

V. Conclusions

In this paper, we proposed a middleware layer to provide efficient communication support for cluster-based applications on SANs. We used various techniques such as carefully designed interfaces, communication protocol optimization and flexible mechanisms to ensure communication progress. We have implemented our design for two platforms: VIA and InfiniBand. Our performance evaluation shows that the protocol processing overhead is low and our proposed middleware largely preserves the performance of underlying communication layer.

Although we have evaluated our system using various micro-benchmarks, many design alternatives in our system can be best evaluated using real applications. Currently, we are working along this direction, and our target applications are database applications and cluster-based service systems such as cluster-based web servers.

Acknowledgments

We would like to thank Craig Stunkel of IBM Research for valuable discussions and his support.

References

[1] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *International Symposium on Computer Architecture*, 1992, pp. 256–266.

[2] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM)," in *Proceedings of the Supercomputing*, 1995.

[3] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-level Network Interface for Parallel and Distributed Computing," in *ACM Symposium on Operating Systems Principles*, 1995.

[4] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," *IEEE Micro*, pp. 66–76, March/April 1998.

[5] InfiniBand Trade Association, "InfiniBand Architecture Specification, Release 1.0," October 24 2000.

[6] DAT Collaborative, "uDAPL and kDAPL API Specification V1.0," June 2002.

[7] The Open Group, "The Interconnect Software Consortium." [Online]. Available: http://www.opengroup.org/icsc/

[8] Lawrence Livermore National Laboratory, "MVICH: MPI for Virtual Interface Architecture," August 2001. [Online]. Available: http://www.nersc.gov/research/FTG/mvich/

[9] H. V. Shah and R. S. Madukkarumukumana, "Design and Implementation of Efficeint Communication Abstractions on the Virtual Interface Architecture: Stream Sockets and RPC Experience," *Software-Practice and Experience*, vol. 31, pp. 1043–1065, 2001.

[10] M. Banikazemi, J. Liu, D. K. Panda, and P. Sadayappan, "Implementing TreadMarks over VIA on Myrinet and Gigabit Ethernet: Challenges, Design Experience, and Performance Evaluation," in *Proceedings of International Conference on Parallel Processing*, 2001.

[11] InfiniBand Trade Association, "Socket Direct Protocol Specification V1.0," 2002.

[12] Technical Committee T10, "SCSI RDMA Protocol," 2002.

[13] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li, "Expericences with VI Communication for Database Storage," in *In Proceedings of International Symposium on Computer Architecture'02*, 2002.

[14] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini, "User-Level Communication in Cluster-Based Servers," in *Proceedings of the Eighth Symposium on High-Performance Architecture (HPCA'02)*, February 2002.

[15] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication," In Proceedings of 12th International Parallel Processing Symposium, April 1998.

[16] Emulex Corporation, "cLAN: High Performance Host Bus Adapter," September 2000. [Online]. Available: http://www.emulex.com/products/legacy.html

[17] Mellanox Technologies, "Mellanox InfiniBand InfiniHost Adapters," July 2002. [Online]. Available: http://www.mellanox.com