Dissertation Proposal

Designing Middleware and Network Architectures to allow High Performance Communication for TCP/IP based applications

Advisor: D. K. Panda

Pavan Balaji

Contents

List of Figures

1 Introduction

Cluster systems are becoming increasingly popular in various application domains mainly due to their high performance-to-cost ratio. These cluster systems essentially consist of commodity-off-the-shelf (COTS) PCs networked with high-speed network interconnects. Cluster systems are now present at all levels of performance, due to the increasing performance of commodity processors, memory and network technologies. Out of the current Top 500 Supercomputers, 149 systems are clusters [9].

As interconnect architectures networking cluster systems approach multi-gigabit per second speeds, the communication overhead in cluster systems is shifting from the network itself to the networking protocols on the sender and the receiver sides. Earlier generation networking protocols such as TCP/IP [56, 58] relied upon the kernel for processing the messages. This caused multiple copies and kernel context switches in the critical message passing path. Thus, the communication overhead was high. During the last few years, researchers have been looking at alternatives to increase the communication performance delivered by clusters in the form of low-latency and high-bandwidth user-level protocols such as FM [47] and GM [36] for Myrinet [21], EMP [54, 55] for Gigabit Ethernet [39], etc. The Virtual Interface Architecture (VIA) [25, 4, 33] was proposed earlier to standardize these efforts. InfiniBand Architecture (IBA) [11] has been recently standardized by the industry to design next generation high-end clusters. These developments are reducing the gap between the performance capabilities of the physical network and that obtained by the end users.

While this approach is good for developing new applications, it might not be so beneficial for the already existing applications which were developed over a span of several years. These applications aim at portability across various platforms. Message Passing Interface (MPI) [43] and the Sockets Interface have been popular choices for achieving such portability. MPI has been the de facto standard for scientific applications while Sockets has been the most widely used programming model for commercial applications. In this proposal, we only focus on the sockets interface.

Figure 1 shows typical environments used by such applications. Figure 1(a) shows the communication infrastructure within a cluster environment connected with a high speed interconnect such as InfiniBand, Myrinet or 10-Gigabit Ethernet [40, 37].

Figures 1(b) and 1(c) show multiple clusters connected either over a WAN or a high-speed back-bone network. These figures illustrate scenarios where the application not only communicates with nodes within the cluster, but also with nodes on other clusters separated over the WAN or a back-bone network. In such scenarios, TCP/IP based sockets can achieve the required compatibility but will not be able to leverage the performance obtainable especially in the high speed backbone networks (Figure 1(c)).

1.1 Advent of High Performance Networks and Protocols

As mentioned earlier, several high performance networks have been introduced in the market in the past few years. Most of these networks provide a number of interesting features. One of the most important features

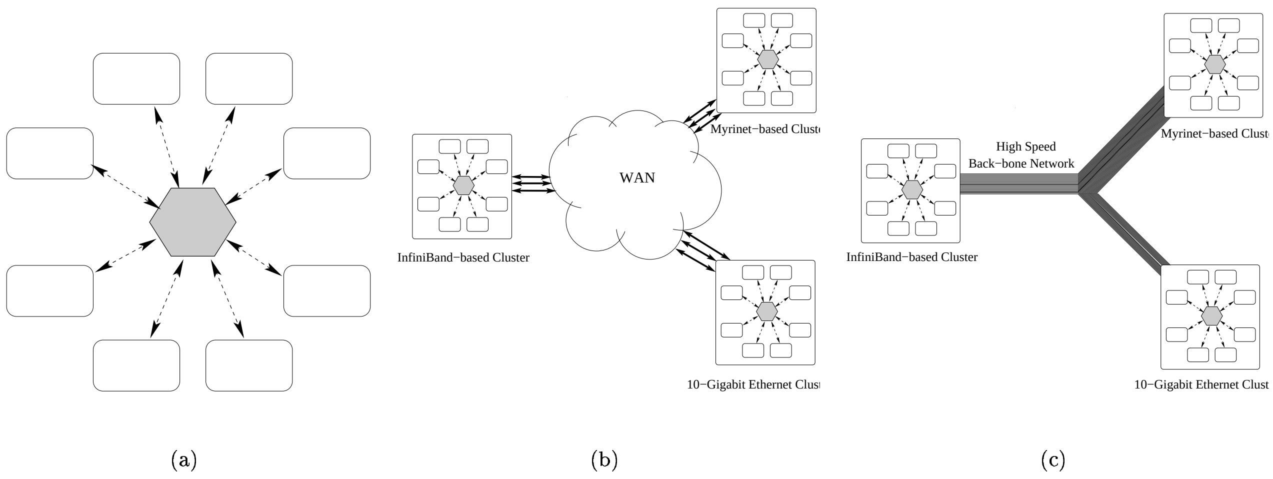


Figure 1: Typical Environments for Sockets Applications: (a) Communication within the cluster environment, (b) Inter-Cluster Communication over a WAN and (c) Inter-Cluster Communication over a High-Speed Backbone Network

provided by these networks is an offloaded protocol stack, i.e., these networks implement a protocol stack similar to TCP/IP on the hardware or firmware on the network adapter itself. Such network adapters are typically referred to as Protocol Offload Engines (POE) [60]. Other features provided by modern networks include zero-copy data transfer capabilities, one-sided communication, etc.

These networks also provide user-level protocol layers which expose these features to the user applications. Thus, new applications built on top of these user-level protocols can directly take advantage of the features provided by the network. However, current sockets layers are implemented on top of the host based TCP/IP stack, due to which sockets based applications are not able to take advantage of such features. Figure 2 illustrates this issue with high performance networks.

It needs to be noted that since the network adapters offload their own protocol stack and not any IP-based protocol such as TCP, they are not compatible with the IP-aware Wide Area Network (WAN). A new initiative by IETF called Remote Direct Data Placement (RDDP) [52, 50] was started to tackle this limitation with existing high performance networks. The Direct Data Placement standard was developed to serve two purposes. First, the protocol should be able to provide high performance in System Area Network (SAN) and other controlled environments by utilizing an offloaded protocol stack and zero-copy data transfer between host memories. Second, the protocol should maintain compatibility with the existing IP infrastructure using an implementation over IP based reliable transport layer stack. DDP uses many of the interesting features (e.g., one-sided communication) in high performance networks, and layers it on top of IP.

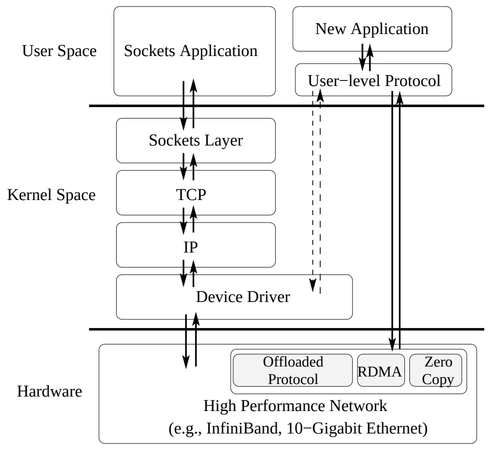


Figure 2: Traditional Implementation of Sockets on High Performance Networks

The DDP specification extends the capabilities provided by the traditional IP-based reliable protocol stacks such as TCP/IP, SCTP/IP, etc., in multiple ways. Some of them are listed below:

- Zero-Copy Data Transfer: A true and complete implementation of DDP allows the network adapter to directly access data from the user-buffers, thus allowing a zero-copy transfer of data between host memories.
- One-Sided Communication: Together with the traditional two-sided communication model, the DDP standard also allows one-sided communication. In a two-sided communication model, when the sender sends out some data, the receiver needs to explicitly perform a receive operation before the data can be received. However, in one-sided communication operations such as Remote Direct Memory Access (RDMA) operations, the sender can directly read or write data into the receiver nodes memory without any intervention from the receiver process.

Since DDP extends the capability of IP-based protocols, it supports most of the advanced features in other high performance networks while maintaining backward compatibility with the IP-based WAN environment.

1.2 Open Challenges and Issues

While several networks provide interesting features, traditionally the sockets layer had been built on top of the host based TCP/IP protocol stack; thus sockets based applications have not been able to take advantage of the performance provided by the high speed networks. One approach to allow such applications to take advantage of the high performance provided by the modern networks, is to implement a *High Performance Sockets* layer directly on top of the user-level protocols provided by the networks, thus utilizing the offloaded protocol stack present on them. These sockets layers are mainly designed to serve two purposes: (a) to provide a smooth transition to deploy existing sockets-based applications on to clusters connected with high performance networks and (b) to sustain most of the performance provided by the high performance protocols by utilizing the offloaded protocol stack they provide.

At this point, the following open questions arise:

- What are the design challenges involved in mapping the offloaded protocols offered by the high performance networks to the requirements of the sockets semantics?
- What are the issues associated with the high performance sockets layers over high performance networks? What impact can they have at the applications? What is missing in the sockets API?
- High Performance protocols such as GM, EMP, DDP, etc., offer high performance but require the applications to be completely rewritten to utilize them. On the other hand, the high performance sockets layers do not require any modifications to the applications, but the performance they can deliver is an open issue. Which approach should the next generation applications take in order to maximize performance? Is a two-folded solution possible, where (i) the application can directly run on the high performance network with high performance using a pseudo sockets-like layer and (ii) at the same time have the flexibility to modify minor portions to utilize the modern features provided by high performance networks such as zero-copy data transfer, one-sided communication, etc.?

2 Problem Statement

As indicated earlier, the traditional protocol stacks such as TCP/IP have not been able to meet the high speeds provided by the current and the upcoming multi-gigabit per second networks. This is mainly associated with the high overhead implementation of the protocol stack itself together with the multiple copies and kernel context switches in the critical message passing path. Protocol Offload Engines (POEs) try to alleviate this problem by offloading either TCP/IP or some other protocol stack on to the network adapter. High performance sockets implementations allow applications to access these offloaded protocol stacks without requiring any modifications. Figure 3 demonstrates the traditional approaches and our proposed high performance sockets approach for allowing compatibility for sockets based applications.

The traditional communication architecture involves just the application and the libraries in user space, while protocol implementations such as TCP/UDP, IP, etc reside in kernel space (Figure 3(a)). This approach not only entails multiple copies for each message, but also requires a context switch to the kernel for every communication step, thus adding a significant overhead. Most of the network adapters which do not feature any offloaded protocol stack implementations (also known as dumb NICs) use this style of architecture.

For high performance network adapters which have protocol stacks offloaded on hardware, researchers have been coming out with different approaches for providing the sockets interface. One such approach was used by GigaNet Incorporation [33] (now known as Emulex) to develop their LAN Emulator (LANE) driver to support the TCP stack over their VIA-aware cLAN cards. Similarly, Mellanox Corporation uses an IP

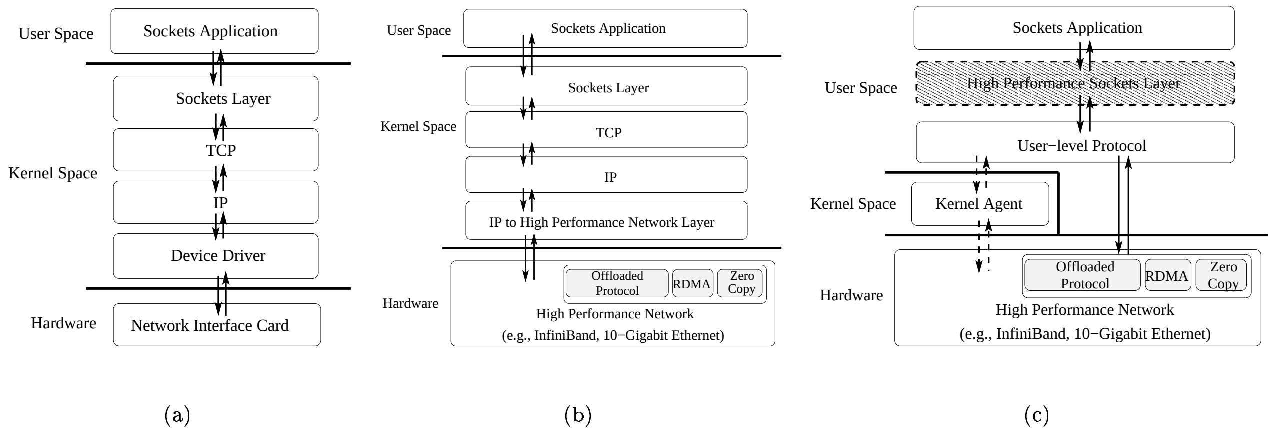


Figure 3: Approaches to the Sockets Interface: (a) Traditional, (b) Mapping IP to the offloaded protocol layers (such as VIA and IBA), and (c) Mapping the Sockets layer to the User-level protocol

over InfiniBand (IPoIB) [2] driver to support the TCP stack on their InfiniBand aware network adapters. These drivers use a simple approach. They provide an IP to offloaded protocol layer (e.g., IP-to-VI layer for VI NICs) layer which maps IP communications onto the NIC (Figure 3(b)). However, TCP is still required for reliable communications, multiple copies are necessary, and the entire setup is in the kernel as with the traditional architecture outlined in Figure 3(a). Although this approach gives us the required compatibility with existing sockets implementations, it can not be expected to give any performance improvement.

The high performance sockets based solution proposed creates an intermediate layer which maps the sockets library onto the offloaded protocol stack provided by the network adapter. This layer ensures that no change is required to the application itself. Figure 3(c) provides an overview of the proposed High Performance Sockets architecture.

2.1 Issues with High Performance Sockets over Protocol Offload Engines

While high performance sockets would be beneficial for several applications built using the sockets interface, it has several issues that need to be dealt with. Some of them are listed below:

1. A number of applications have been written keeping the performance of the traditional sockets layer in mind and optimized for these layers. Blindly running such applications on a high performance sockets layer would not provide any benefit. As we have seen in our previous work with the Data-Cutter [19, 17] runtime environment, directly utilizing the high performance sockets layer can give some benefit in certain scenarios, but the performance gain is inherently limited by the communication characteristics of the application. For example, some applications we have studied can try to balance the computation to communication ratio keeping the TCP/IP communication overhead in mind. Blindly running such

applications on a high performance sockets interface might give minimal to no advantage of such socket layers. On the other hand, redesigning small components of the application such as the computation to communication breakup ratio can increase the performance gain by several orders of magnitude.

2. User-level protocols have been developed and optimized to provide benefits to two kinds of semantics: (i) the semantics provided by the native API supported by the protocol and (ii) the semantics specified by the standard Message Passing Interface (MPI). For example, the connection time for several high performance protocols such as VAPI over InfiniBand is significantly higher as compared to the data transfer time. Also, some protocols such as GM over Myrinet limit the number of processes communicating simultaneously to a low number (such as eight). However, such semantics are several times ill-suited for standard sockets based applications. These limitations have significant impact on the capabilities and utility of these sockets layers in real environments.

3. The sockets API mandates the use of buffering on the end nodes. This limitation is especially evident in synchronous sockets interfaces. Several applications such as Apache, etc., utilize the synchronous sockets interface for communication and are forced to perform buffering of data on the end nodes. We have implemented a zero-copy high performance sockets mechanism where these copies can be avoided in some scenarios; however, this mechanism is not generic and does not guarantee a zero-copy data transfer in several scenarios. While this is not a critical disadvantage for micro-benchmarks and a few applications which have high cache hit rates, as we have seen in our previous work [13], it can limit the performance of some applications on 10-Gigabit networks to less than a third of the peak throughput achievable.

4. The sockets API requires intervention from both communication end-points. For environments where one end is computationally stressed, a two-sided communication API might have significant degradation in the performance. We have shown the disadvantages of such two-sided communication required in the sockets API on the performance of the data-center environment [45].

5. The high performance sockets layers are based on non-IP aware transport layers. This impacts the scalability of these layers outside the cluster environment, forcing a homogeneous network test-bed.

These issues need to be dealt with before a high performance sockets implementation can be used in practice.

2.2 Proposed Research Framework and Challenges

Keeping in mind the issues associated with the basic high performance sockets implementations and upcoming standards for Direct Data Placement and RDMA over IP, we propose to design and implement a multi-stage integrated framework with the following capabilities:

1. The framework should be based on high performance sockets implementations. This would allow applications to utilize the benefits of the high performance sockets based solutions proposed above. In particular, the applications can not only directly run on the network without any modifications, but also be able to extract reasonable performance from the network.

2. The framework should encompass DDP and RDMAP but only as an extended sockets interface. This would allow users to make changes to their applications as and when required and not place it as a primary requirement for them to run, i.e., the application writers will need to modify only the segments of the code which they feel are critical for performance without having to rewrite the entire application.

3. The framework should try to either utilize a completely offloaded TCP/IP protocol stack or an emulation of the same. This would allow the implementation to be compatible with the IP-based Wide Area Network (WAN) environment and at the same time capable of being used in a distributed cluster environment with a high performance.

Figure 4 illustrates the proposed integrated framework in a stage-wise format detailing the implementation steps.

Stage 0 in the figure shows the existing architecture with kernel-based implementations of TCP/IP and the sockets interface. Stage 1 illustrates the framework with high performance sockets implementations over the networks with offloaded protocol stacks. This stage focuses on the issues related to mapping the requirements of the sockets semantics (e.g., data streaming, connection management, etc.) with the features provided by the network (e.g., zero-copy data transfer, one-sided communication). The framework in this stage will be able to maintain compatibility with the existing sockets based applications while retaining the performance provided by the networks.

Stage 2 depicts an intermediate stage where we extend the sockets API to provide DDP extensions. The framework in this stage allows users to either utilize the sockets API directly, or modify minor portions in the application as and when required to utilize the features provided by the DDP and RDMAP standards.

Stage 3 illustrates the final stage of the framework where TCP/IP together with DDP and RDMAP are offloaded onto the network adapter itself. The framework in this stage retains all the benefits of high performance sockets implementations, allows applications to utilize the features provided by the DDP standards and maintains compatibility with the IP-based WAN infrastructure.

There are many challenges involved in designing, developing, and evaluating the proposed framework. Specifically, we plan to carry out research along the following directions:

- Designing, developing and analyzing the various design choices for implementing high performance sockets implementations.

- Analyzing the issues related to high performance sockets implementations. In particular, the impact on data-intensive applications which have tuned their communication to computation ratios based on the

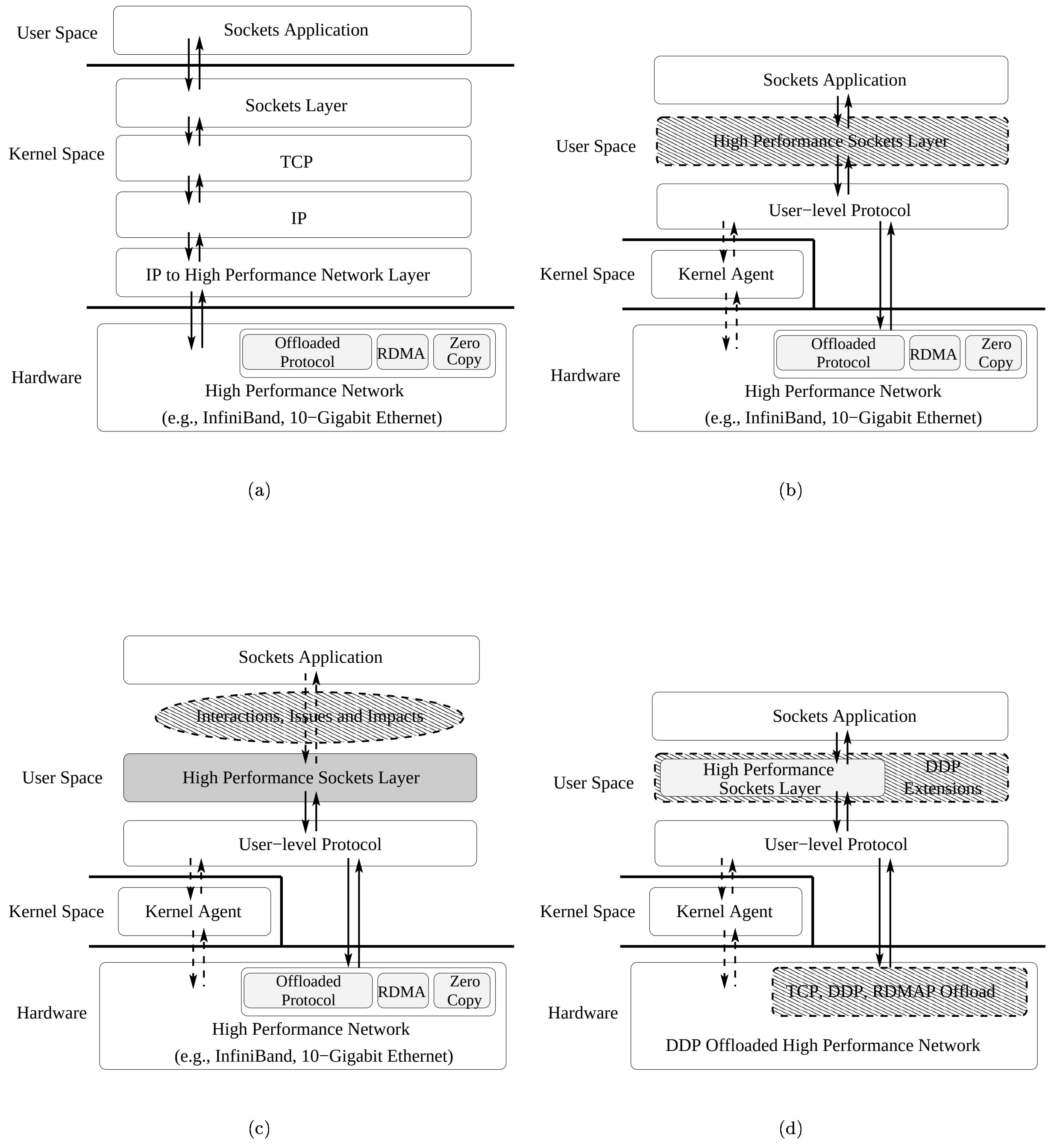


Figure 4: The Proposed Framework: (a) Stage 0: Existing Intrastructure for sockets based applications, (b) Stage 1: High Performance sockets implementation to take advantage of the offloaded protocol stacks, (c) Stage 2: Analyzing the interactions, impacts and issues associated with various application environments on the performance of high performance sockets and (d) Stage 3: Forming an integrated framework which encompasses high performance sockets and DDP/RDMAP as extended sockets API.

performance of TCP/IP, memory traffic patterns for sockets implementations during L2-cache misses, impact on computationally stressed environments, etc.

- Designing and integrating DDP and RDMAP protocol stacks with the high performance sockets framework.

- Application level study to understand the impact of the various design choices in the context of different application environments.

3 Background

In this section, we provide a brief background about different networks such as Gigabit Ethernet, GigaNet cLAN and InfiniBand and the corresponding user-level protocols developed on top of these networks.

3.1 The Ethernet Message Passing (EMP) Protocol over Gigabit Ethernet

Since its inception in the 1970s, Ethernet [39, 40, 37] has been an important networking protocol, with the highest installation base. Gigabit Ethernet builds on top of Ethernet but increases speed multiple times (Gb/s). Since this technology is fully compatible with Ethernet, one can use the existing Ethernet infrastructure to build gigabit per second networks. A Gigabit Ethernet infrastructure has the potential to be relatively inexpensive, assuming the industry continues with the Ethernet price trends. Given so many benefits of Gigabit Ethernet it is imperative that there exist a low latency messaging system for Gigabit Ethernet.

The Ethernet Message Passing (EMP) [54, 55] protocol is one such low latency messaging system developed over Gigabit Ethernet. The EMP protocol specifications have been developed at The Ohio Supercomputing Center and The Ohio State University to fully exploit the benefits of Gigabit Ethernet. EMP is a complete zero-copy, OS-bypass, NIC-level messaging system for Gigabit Ethernet. This is the first protocol of its kind on Gigabit Ethernet. It has been implemented on a Gigabit Ethernet network interface chip-set based around a general purpose embedded microprocessor design called the Tigon2 (produced by Alteon Web Systems, now owned by Nortel Networks). This is a fully programmable NIC, whose novelty lies in its two CPUs. Figure 5 provides an overview of the Alteon NIC architecture.

In EMP, message transmission follows a sequence of steps (Figure 6). First the host posts a transmit descriptor to the NIC (T1), which contains the location and length of the message in the host address space, destination node, and an MPI specified tag. Once the NIC gets this information (T2-T3), it DMAs this message from the host (T4-T5), one frame at a time, and sends the frames on the network. Message reception follows a similar sequence of steps (R1-R6) with the difference that the target memory location in the host for incoming messages is determined by performing tag matching at the NIC (R4). Both the source index of

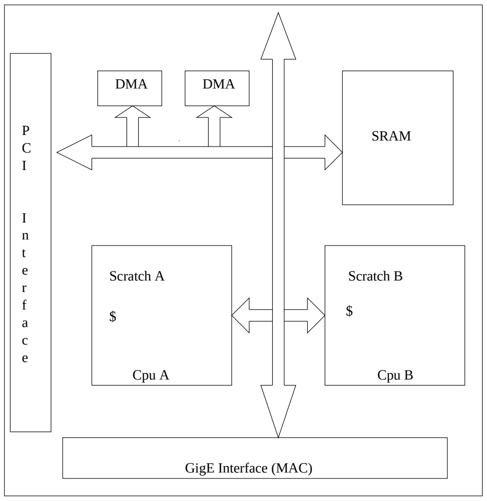


Figure 5: The Alteon NIC Architecture

the sender and an arbitrary user-provided 16-bit tag are used by the NIC to perform this matching, which allows EMP to make progress on all messages without host intervention.

EMP is a reliable protocol. This mandates that for each message being sent, a transmission record be maintained (T3). This record keeps track of the state of the message including the number of frames sent, a pointer to the host data, the sent frames, the acknowledged frames, the message recipient and so on.

Similarly, on the receive side, the host pre-posts a receive descriptor at the NIC for the message which it expects to receive (R1). Here, the state information which is necessary for matching an incoming frame is stored (R4). Once the frame arrives (R3), it is first classified as a data, header, acknowledgment or a negative acknowledgment frame. Then it is matched to the pre-posted receive by going through all the pre-posted records (R4). If the frame does not match any pre-posted descriptor, it is dropped. Once the frame has been correctly identified the information in the frame header is stored in the receive data structures for reliability and other bookkeeping purposes (R4). For performance reasons, acknowledgments are sent for a certain window size of frames. In our current implementation, this was chosen to be four. Once the receive records are updated, the frame is scheduled for DMA to the host using the DMA engine of the NIC (R6).

EMP is a zero-copy protocol as there is no buffering of the message at either the NIC or the host, in both the send and receive operations. It is OS bypass in that the kernel is not involved in the bulk of the operations. However, to ensure correctness, each transmit or receive descriptor post must make a call to the operating system for two reasons. First, the NIC accesses host memory using physical addresses, unlike the virtual addresses which are used by application programs. Only the operating system can make this translation. Second, the pages to be accessed by the NIC must be pinned in physical memory to protect against the corruption that would occur if the NIC wrote to a physical address which no longer contained the application page due to kernel paging activity. We do both operations in a single system call (T2/R2).

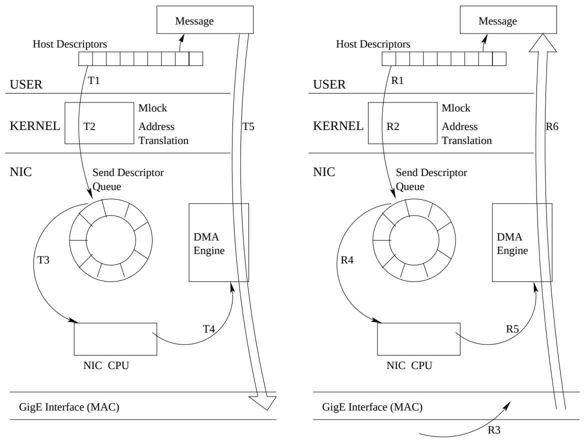


Figure 6: EMP protocol architecture showing operation for transmit (left), and receive (right).

One of the main features of this protocol is that it is a complete NIC based implementation. This gives maximum benefit to the host in terms of not just bandwidth and latency but also CPU utilization.

EMP has two implementations. One which has been implemented on the single CPU [54] of the NIC and the other utilizing both the CPUs of the Alteon NIC [55].

3.2 GigaNet cLAN: A hardware implementation of VIA

Virtual Interface Architecture (VIA) [14, 15, 24, 25, 34, 35], as an industry standard, had been developed and standardized mainly by Compaq, Intel and Microsoft. Since VIA has a very low level API which provides only the minimal communication primitives to the programmer, developing applications on VIA is considered a challenge in itself.

VIA is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers and VI Consumers. The organization of these components is illustrated in Figure 7.

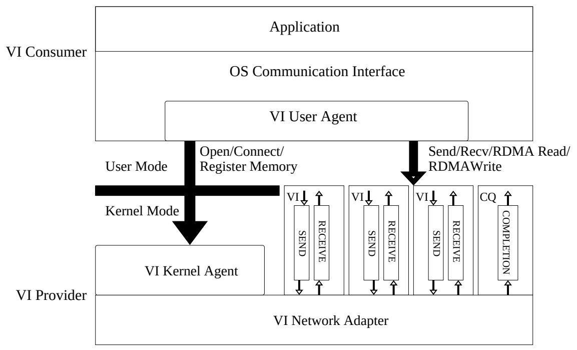


Figure 7: The Virtual Interface (VI) Architectural Model

The VI provider is composed of a physical network adapter and a software kernel agent. The VI consumer is generally composed of an application program and an operating system communication facility. A Virtual

Interface consists of a pair of Work Queues: a send queue and a receive queue. VI Consumers post requests, in the form of descriptors, on the work queues to send or receive data. A descriptor is a memory structure that contains all the information that the VI provider needs to process the request, such as pointers to data buffers. VI providers asynchronously process the posted descriptors and mark them with a status value when completed. VI consumers remove completed descriptors from the work queues and use them for subsequent requests. Each work queue has an associated doorbell that is used to notify the VI network adapter that a new descriptor has been posted to a work queue. The doorbell is directly implemented by the adapter and requires no Operating System intervention to operate. A Completion Queue allows a VI Consumer to coalesce notification of descriptor completions from the work queues of multiple VIs in a single location.

VIA has been designed to provide high-bandwidth and low-latency support over a System Area Network (SAN). Since its introduction, implementations of VIA have been made available on a variety of platforms. M-VIA (Modular VIA) [4] emulates the VIA specification by software for legacy Fast Ethernet and Gigabit Ethernet adapters. B-VIA (Berkeley VIA) [24, 25] implementation supports the VIA specification on Myrinet, by modifying its firmware. Finally, GigaNet Incorporation (now called Emulex Corporation) [34] has developed a proprietary VI-aware NIC called GigaNet cLAN. This network adapter implements the VIA specifications on the NIC itself in hardware allowing the best performance amongst the various implementations of VIA.

3.3 InfiniBand Architecture (IBA)

InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. A typical IBA cluster consists of switched serial links for interconnecting both the processing nodes and the I/O nodes. The IBA specification defines a communication and management infrastructure for both inter-processor communication as well as inter and intra node I/O. IBA also defines built-in QoS mechanisms which provide virtual lanes on each link and define service levels for individual packets.

In an InfiniBand network, processing nodes and the I/O nodes are connected to the fabric by Host Channel Adapters (HCA) and Target Channel Adapters (TCA). HCAs are associated with processing nodes and their semantic interface to consumers is specified in the form of InfiniBand Verbs. TCAs connect I/O nodes to the fabric and have interfaces to consumers that are implementation specific and are not defined in the IBA specifications. Channel Adapters usually have programmable DMA engines with protection features.

IBA mainly aims at reducing the system processing overhead by decreasing the number of copies associated with a message transfer and removing the kernel from the critical message passing path. The InfiniBand communication stack consists of different layers. The interface presented by Channel Adapters to consumers belongs to the transport layer. A Queue Pair (QP) based model is used in this interface. Figure 8 illustrates the InfiniBand Architecture.

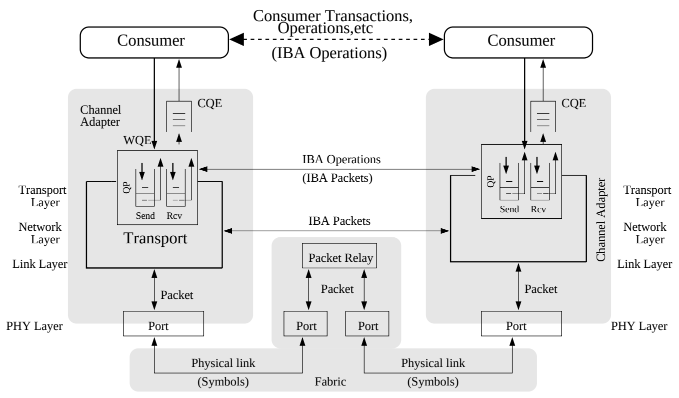


Figure 8: The InfiniBand Architectural Model

Each Queue Pair is a communication endpoint. A Queue Pair consists of a send queue and a receive queue. Two QPs on different nodes can be connected to each other to form a logical bi-directional communication channel. An application can have multiple QPs. Communication requests are initiated by posting Work Queue Requests (WQRs) to these queues. Each WQR is associated with one or more pre-registered buffers from which data is either transferred (for a send WQR) or received (receive WQR). Further, the application can either choose to be signaled on the completion of a WQR using the Signaled (SG) request or alternatively choose an Unsignaled (USG) request. When the HCA completes the processing of a signaled request, it places an entry in the Completion Queue (CQ) called as the Completion Queue Entry (CQE).

The consumer application can poll on the CQ associated with the work request to check for completion. There is also the feature of triggering event handlers whenever a completion occurs. For Unsignaled requests, no completion event is returned to the user. However, depending on the implementation, the driver cleans up the Work Queue Request from the appropriate Queue Pair on completion.

IBA Communication Models: IBA supports two types of communication semantics: Channel Semantics (Send-Receive communication model) and Memory Semantics (RDMA communication model). In channel semantics, each send request has a corresponding receive request at the remote end. Thus there is a one-to-one correspondence between every send and receive operation. Failure to post a descriptor on the remote node results in the message being dropped and if the connection is reliable, it might even result in the breaking of the connection. In memory semantics, Remote Direct Memory Access (RDMA) operations are used. These operations are transparent at the remote end since they do not require a receive descriptor to be posted. In this semantics, the send request itself contains both the virtual address for the local transmit buffer as well as that for the receive buffer on the remote end. The RDMA operations are available with the Reliable Connection (RC) service type. The IBA specifications does not provide different primitives for the channel semantics and the memory semantics. It is the "opcode" entry in the WQR, which distinguishes between the channel semantics and the memory semantics. Most other entries in the WQR are common for both the Send-Receive model as well as the RDMA model, except an additional remote buffer virtual address (and other related entries) which has to be specified for RDMA operations. There are two kinds of RDMA operations: RDMA Write and RDMA Read. In the RDMA Write model, the initiator directly writes data into the remote node's memory. Similarly, in the RDMA Read model, the initiator directly reads data from the remote node's memory.

IBA Communication Protocols: There are two main types of traffic over IBA fabrics. The first type of traffic is IBA native protocols such as VAPI [5], IBAL (InfiniBand Access Layer) [1], uDAPL (User-Level Direct Access Transport APIs) [7] and kDAPL (Kernel-Level Direct Access Transport APIs) [8]. These interfaces are low-level APIs for IBA. Applications can be programmed using these APIs to take full advantage of IBA user level networking and RDMA features. IP is the other type of traffic (and a very important one) that could use this interconnect. InfiniBand would benefit greatly from a standardized method of handling IP traffic. IPoIB provides standardized IP encapsulation over IBA fabrics as defined by the IETF Internet Area IPoIB working group [2]. The IPoIB project [3] implements this proposed standard as a layer-2 Linux network driver. The primary responsibilities of the driver are performing address resolution to map IPv4 and IPv6 addresses to InfiniBand Unreliable Datagram (UD) address vectors, the management of multicast membership, and the transmission and reception of IPoIB protocol frames.

4 Detailed Problem Description and Preliminary Studies

The problems mentioned in Section 2 have many intrinsic issues and non-trivial challenges which need to be addressed. We break up the challenges involved into three broad categories:

1. Identifying the basic design challenges involved in implementing high performance sockets layers on top of user-level protocols. These include protocol mismatches, connection management mechanisms, unexpected messages, etc. More details about these in the context of high performance sockets over Gigabit Ethernet, GigaNet cLAN and InfiniBand are provided in Section 4.2.

2. Analyzing the issues involved in high performance sockets implementations. In particular we analyze: (i) impact of such sockets implementations on data intensive applications which tune the communication to computation ratio in order to reduce the impact of network performance, (ii) impact of the memory copies associated with the sockets layer on the memory traffic generated in 10-gigabit networks and (iii) impact of the two-sided communication primitives of the sockets API in computationally stressed environments. More details about the same are provided in Section 4.3.

3. Designing an integrated framework involving the features of both high performance sockets as well as DDP and RDMAP implementations. These include efficiently extending the sockets API to encompass the DDP and RDMAP standards allowing applications to smoothly transit from the sockets to the

DDP or RDMAP interface as and when required with minimal modifications. More details about this are provided in Section 4.4.

While working on the above mentioned issues, we have published several papers as listed in Section 4.1. In the remaining part of the section, we discuss the pertinent issues and challenges involved in our proposed framework with respect to the three broad categories mentioned above.

4.1 Selected Publications

A list of publications most directly related to this proposal are provided in Section 4.1.1. Other publications which are broadly related to my research direction are listed in Section 4.1.2.

4.1.1 Related Publications

1. **P. Balaji**, S. Narravula, K. Vaidyanathan, H. -W. Jin and D. K. Panda. *On the Provision of Prioritization and Soft QoS in Dynamically Reconfigurable Shared Data-Centers over InfiniBand.* In the proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Mar 20-22, 2005, Austin, Texas.

2. **P. Balaji**, H. Shah and D. K. Panda. *Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck.* In the proceedings of the Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT); held in conjunction with the IEEE International Conference on Cluster Computing, Sep 20th, 2004, San Diego, California.

3. **P. Balaji**, K. Vaidyanathan, S. Narravula, S. Krishnamoorthy, H. -W. Jin and D. K. Panda. *Exploiting Remote Memory Operations to Design Efficient Reconfiguration for Shared Data-Centers over InfiniBand.* In the proceedings of the workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT); held in conjunction with the IEEE International Conference on Cluster Computing, Sep 20th, 2004, San Diego, California.

4. S. Narravula, **P. Balaji**, K. Vaidyanathan, S. Krishnamoorthy, J. Wu and D. K. Panda. *Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand.* In the proceedings of the workshop on System Area Networks (SAN); held in conjunction with the IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb 14th, 2004, Madrid, Spain.

5. **P. Balaji**, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu and D. K. Panda. *Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial?.* In the proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Mar 10-12, 2004, Austin, Texas.

6. **P. Balaji**, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda and J. Saltz. *Impact of High Performance Sockets on Data Intensive Applications.* In the proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC), Jun 22-24, 2003, Seattle, WA.

7. **P. Balaji**, P. Shivam, P. Wyckoff and D. K. Panda. *High Performance User-Level Sockets over Gigabit Ethernet.* In the proceedings of the IEEE International Conference on Cluster Computing, Sept 23-26, 2002, Chicago, IL.

4.1.2 Other Publications

1. H. -W. Jin, **P. Balaji**, C. Yoo, J . Y. Choi and D. K. Panda. *Exploiting NIC Architectural Support for Enhancing IP based Protocols on High Performance Networks.* Accepted for publication at the Special Issue of the Journal of Parallel and Distributed Computing (JPDC) on Design and Performance of Networks for Super-, Cluster- and Grid-Computing, 2005.

2. S. Narravula, **P. Balaji**, K. Vaidyanathan, H. -W. Jin and D. K. Panda. *Architecture for Caching Responses with Multiple Dynamic Dependencies in Multi-Tier Data-Centers over InfiniBand.* In the proceedings of the IEEE International Conference on Cluster Computing and the Grid (CCGrid), May 9-12, 2005, Cardiff, UK.

3. K. Vaidyanathan, **P. Balaji**, H. -W. Jin and D. K. Panda, Workload-driven Analysis of File Systems in Shared Multi-tier Data-Centers over InfiniBand. In the proceedings of the Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-8); to be held in conjunction with the 11th International Symposium on High Performance Computer Architecture (HPCA-11), Feb 12, 2005, San Francisco, CA.

4. M. Islam, **P. Balaji**, P. Sadayappan and D. K. Panda, Towards Provision of Quality of Service Guarantees in Job Scheduling. In the proceedings of the IEEE International Conference on Cluster Computing, Sep 20-23, 2004, San Diego, California.

5. R. Kurian, **P. Balaji**, P. Sadayappan, Opportune Job Shredding: An Effective approach for scheduling Parameter Sweep Applications. In the proceedings of the Los Alamos Computer Science Institute Symposium (LACSI), Oct 12-14, 2003, Santa Fe, New Mexico.

6. M. Islam, **P. Balaji**, P. Sadayappan and D. K. Panda, QoPS: A QoS based scheme for Parallel Job Scheduling. In the proceedings of the Job Scheduling Strategies for Parallel Processing Workshop (JSSPP); held in conjunction with the IEEE International Symposium on High Performance Distributed Computing (HPDC), Jun 24th, 2003, Seattle, WA. Also Published in the IEEE Springer Lecture Notes in Computer Science (LNCS).

7. R. Gupta, **P. Balaji**, J. Nieplocha and D. K. Panda. *Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters.* In the proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), Apr 22-26, 2003, Nice, France.

4.2 High Performance Sockets over User-level Protocols

In this section, we discuss the design issues involved in implementing high performance sockets layers directly on top of user-level protocols allowing sockets based applications to take advantage of the offloaded protocol stacks. We discuss implementations of such high performance sockets layers on three different networks: Gigabit Ethernet, GigaNet cLAN and InfiniBand.

4.2.1 High Performance Sockets over Gigabit Ethernet

While implementing a high performance sockets layer to support sockets based applications directly on top of the Ethernet Message Passing (EMP) Protocol over Gigabit Ethernet, several issues need to be considered. In this section, we mention a few of them, discuss the possible alternatives and the pros and cons of each of the alternatives.

The mismatches between TCP and EMP are not limited to the syntax alone. The motivation for developing TCP was to obtain a reliable, secure and fault tolerant protocol. However, EMP was developed to obtain a low-overhead protocol to support high performance applications on Gigabit Ethernet. While developing a high performance sockets layer over EMP, it must be kept in mind that the application was designed around the semantics of TCP; so these semantics need to be maintained.

Connection Management: TCP is a connection based protocol, unlike EMP. In TCP, when a connection request is sent to the server, it contains information about the client requesting the connection. In our approach, the client needs to send an explicit message to the server containing similar information about the client requesting the connection. However, this puts an additional requirement on the high performance sockets layer to post descriptors for the connection management messages too. To handle this issue, when the application calls the listen() call, the high performance sockets layer can postx a number of descriptors equal to the usual sockets parameter of a backlog which limits the number of connections that can be simultaneously waiting for an acceptance. When the application calls accept(), the high performance sockets layer blocks on the completion of the descriptor at the head of the backlog queue.

Unexpected Message Arrivals: Like most other user-level protocols, EMP has a constraint that before a message arrives, a descriptor must have been posted so that the NIC knows where to DMA the arriving message. However, EMP is a reliable protocol. So, when a message arrives, if a descriptor is not posted, the message is dropped by the receiver and eventually retransmitted by the sender. This facility relaxes the descriptor posting constraint to some extent. However, allowing the nodes to retransmit packets indefinitely might congest the network and harm performance. Posting a descriptor before the message arrives is not essential for the functionality, but is crucial for performance issues. We examined three separate mechanisms to deal with this.

- *Separate Communication Thread:* In the first approach, we post a number of descriptors on the receiver side and have a separate communication thread which watches for descriptors being used and reposts them. However, with both threads polling (EMP does not support event based completion notification), the synchronization cost of the threads themselves comes to about 20 μs. Also, the effective percentage of CPU cycles the main thread can utilize would go down to about 50%, assuming equal priority threads.

- *Rendezvous Approach:* The second approach (similar to the approach indicated by [41]) is through rendezvous communication with the receiver as shown in Figure 9. Initially, the receive side posts a descriptor for a request message, not for a data message. Once the sender sends the request, it blocks until it receives an acknowledgment. The receiver on the other hand, checks for the request when it encounters a read() call, and posts two descriptors – one for the expected data message and the other for the next request, and sends back an acknowledgment to the sender. The sender then sends the data message. Effectively, the sender is blocked till the receiver has synchronized and once this is done, it is allowed to send the actual data message. This adds an additional synchronization cost in the latency. Though this approach is straight-forward, it has a few disadvantages. One of the high points of TCP is its data-streaming option. In this option, the message boundaries supplied by the transmitter are not enforced at the receiver. When a message arrives, the receiving node has the option of reading any number of bytes at any time. For example, if the sender sends 10 bytes of data, TCP allows the user to read it as two sets of 5 bytes each, potentially into different user buffers. In EMP, when a message arrives, the data is directly transferred to the user space, and thus this option is disabled. Another disadvantage is a possibility of deadlocks. When the sender wants to send a data message, it first sends a request and waits for the acknowledgment, which is sent only when the receiver encounters a read() call. Consider the case when both the nodes want to send data to each other. Both might call write() and then read(), in that order. Sockets over a normal TCP implementation allows this to some extent as the system provides temporary buffer space, generally 64 Kbytes, which can be used to satisfy the write operations and allow the matching reads to proceed. Using this rendezvous approach, however, both nodes would block waiting for acknowledgment and deadlock (Figure 10). However, rendezvous is a standard technique used by most message passing layers (including MPI), which do not give any guarantees about deadlocks, putting the onus of keeping the application deadlock free on the end user.

- *Eager with Flow Control:* This approach is similar to the rendezvous approach. The receiver initially posts a descriptor. When the sender wants to send a data message, it goes ahead and sends the message. However, for the next data message, it waits for an acknowledgment from the receiver confirming that another descriptor has been posted. Once this acknowledgment has been received, the sender can send the next message. On the receiver side, when a data message comes in, it uses up the

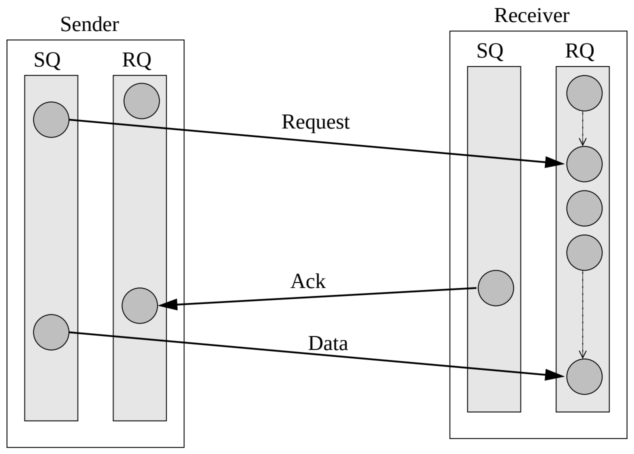


Figure 9: High Performance Sockets over EMP: Rendezvous Approach

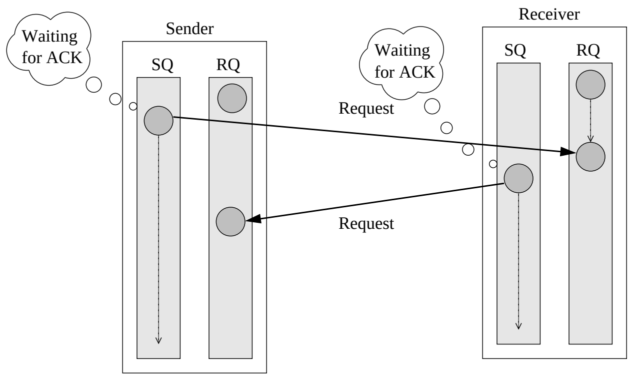


Figure 10: High Performance Sockets over EMP: Possibility of a deadlock in the rendezvous approach

pre-posted descriptor. Since this descriptor was posted without synchronization with the read() call in the application, the descriptor does not point to the user buffer address, but to some temporary memory location. Once the receiver calls the read() call, the data is copied into the user buffer, another descriptor is posted and an acknowledgment is sent back to the sender. This involves an extra copy on the receiver side. Figure 11 illustrates the eager approach with flow control. Note that even this approach is not completely free from deadlocks. However, we have proposed an extension to this idea termed as *credit based flow control* which reduces the possibilities of one and enhances its performance.

Overloading function name-space: Applications built using the sockets interface use a number of standard UNIX system calls including specialized ones such as listen(), accept() and connect(), and generic overloaded calls such as open(), read() and write(). The generic functions are used for a variety of external communication operations including local files, named pipes and other devices. In the substrate, these calls were mapped to the corresponding EMP calls (sets of calls). This mapping can be done in a number of ways.

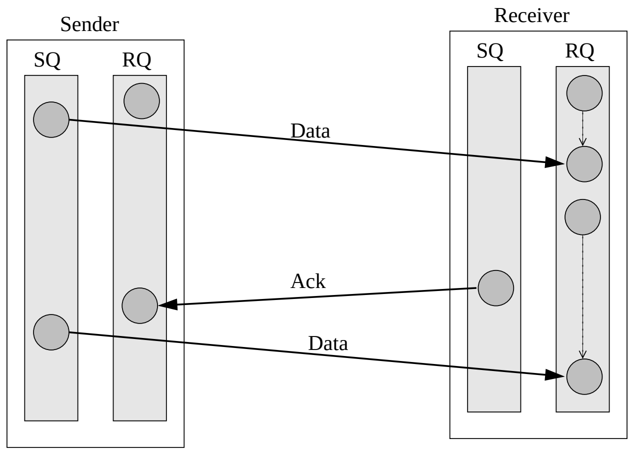


Figure 11: High Performance Sockets over EMP: Eager with Flow Control approach

- *Function Overriding:* In this approach, the TCP function calls are directly mapped to the corresponding EMP function calls by overriding them. This approach works for calls such as listen(), which have just one interpretation. But, for calls such as read() and write(), this approach does not work, as the read can be on a socket or on a file. Overriding cannot distinguish between these two interpretations.

- *Application changes:* In this approach, minor changes are made to the application by adding a parameter which allows the substrate to distinguish between a call to the EMP library and one to the libc library. This approach gives the flexibility of using both sockets over EMP as well as over TCP. However, since the aim of the substrate was to avoid any changes to the application, this approach is not suitable.

- *File descriptor tracking:* In this approach, the high performance sockets implementation functions are caused to be loaded into the application before the standard C library, and the library calls which change the state of file descriptors are monitored, including open(), close() and socket(). In this way, on a read() or write(), for instance, our functions can decide whether to call into the EMP substrate or to pass the parameters on to the standard system function of the same name.

While implementing the high performance sockets layer, the functionality of the calls needs to be taken into account so that the application does not have to suffer due to the changes. However, these adjustments do affect the performance the sockets layer is able to deliver. Several techniques are possible to improve the performance given by the sockets layer some of which are summarized below.

Credit-based flow control: As mentioned earlier, the eager with flow control scheme for handling unexpected messages is not free from the possibility of a deadlock. However, an extension of the idea is possible to reduce the possibility of its occurrence. The sender is given a certain number of credits (tokens). It loses a token for every message sent and gains a token for every acknowledgment received. If the sender is given N credits, the substrate has to make sure that there are enough descriptors and buffers pre-posted for N unexpected message arrivals on the receiver side. In this way, the substrate can tolerate up to N outstanding write() calls before the corresponding read() for the first write() is called without the occurrence of a deadlock.

One problem with applying this algorithm directly is that the acknowledgment messages also use up a descriptor and there is no way the receiver would know when it is reposted, unless the sender sends back another acknowledgment, thus forming a cycle. To avoid this problem, we have propose the following solutions:

- *Blocking the send:* In this approach, the write() call is blocked until an acknowledgment is received from the receiver, which would increase the time taken for a send to a round-trip latency. Further, this scheme is unable to exploit the benefits obtained with respect to avoiding deadlock occurrences.

- *Piggy-back acknowledgment:* In this approach, the acknowledgment is sent along with the next data message from the receiver node to the sender node. This approach again requires synchronization between both the nodes. Though this approach is used in the substrate when a message is available to be sent, we cannot always rely on this approach and need an explicit acknowledgment mechanism too.

- *Post more descriptors:* In this approach, $2N$ number of descriptors are posted where N is the number of credits given. It can be proved that at any point of time, the number of unattended data and acknowledgment messages will not exceed $2N$. A potential problem with this approach is the number of buffers registered. Since the arrival of a data message or the acknowledgment cannot be predicted, a buffer has to be allocated corresponding to each of the descriptors, effectively wasting half the buffer space. However, EMP provides a NIC based tag matching feature. Each descriptor can be associated with a tag, and the incoming message is matched with the tags to find the appropriate descriptor. This feature can help avoid this buffer usage problem.

Disabling Data Streaming: TCP supports the data streaming option, which allows the user to read any number of bytes from the socket at any time (assuming that at-least so many bytes have been sent). To support this option, we use a temporary buffer to contain the message as soon as it arrives and copy it into the user buffer as and when the read() call is called. Thus, there would be an additional memory copy in this case.

However, there are a number of applications which do not need this option. To improve the performance of these applications, we can provide an option in the high performance sockets implementation which allows the user to disable this option. In this case, we can avoid the memory copy for larger message sizes by switching to the rendezvous approach to synchronize with the receiver and DMA the message directly to the user buffer space. This approach is referred to as Datagram sockets. In this approach, however, the responsibility to avoid a deadlock lies on the user.

Delayed Acknowledgments: To improve performance, we delay the acknowledgments so that an acknowledgment message is sent only after half the credits have been used up, rather than after every message. This reduces the overhead per byte transmitted and improves the overall throughput. However, these delayed acknowledgments can bring about an improvement in the latency too. When the number of credits given is small, half of the total descriptors posted are acknowledgment descriptors. So, when the message arrives, the tag matching at the NIC takes extra time to walk through the list that includes all the acknowledgment descriptors. This time was calculated to be about 550 ns per descriptor. With the increase in the number of credits given, the fraction of acknowledgment descriptors decreases, thus reducing the effect of the time required for tag matching in the critical path.

EMP Unexpected Queue: EMP supports a facility for unexpected messages. The user can post a certain number of unexpected queue descriptors, and when the message comes in, if a descriptor is not posted, the message is put in the unexpected queue and when the actual receive descriptor is posted, the data is copied from this temporary memory location to the user buffer. The advantage of this unexpected queue is that the descriptors posted in this queue are the last to be checked during tag matching, which means that access to the more time-critical pre-posted descriptors is faster.

The only disadvantage with this queue is the additional memory copy which occurs from the temporary buffer to the user buffer. In our implementation, we can use this unexpected queue to accommodate the acknowledgment buffers. The memory copy cost is not a concern, since the acknowledgment messages do not carry data payload. Further, there is the additional advantage of removing the acknowledgment messages from the critical path.

4.2.2 High Performance Sockets over Virtual Interface Architecture

In this section, we present the design issues involved in the implementation of a high performance sockets layer over VIA. Several of the design challenges are common to those seen in Section 4.2.1. In order to avoid repetition, we present only the ones for which either the problems or the solutions are significantly different from Section 4.2.1.

Connection Management: For connection management, applications built on TCP use the `listen()` and `accept()` calls. The `listen()` call semantics dictate that, the Operating System be informed when a connection request arrives, the request be kept in a queue, and the client be informed of the status. The function returns as soon as the Operating System is informed to do so. The `accept()` call checks if a request has arrived and if it has not, waits (blocks) for one. On the other hand, VIPL (VI Provider Library) supports two primitive calls – `VipConnectWait()` and `VipConnectAccept()`. `VipConnectWait()` call blocks till the connection request arrives and the control is returned to the user only after the request arrives. Since the number of connections cannot be known before hand, asynchronous connection requests cannot be dealt with this blocking function. As we can see, there is no clear correspondence between the API supported by TCP and VIA. A number of approaches are possible to bridge this mismatch between the two APIs.

- *Peer-to-Peer Connections:* One approach would be to use peer-to-peer connection primitives specific to the GigaNet cLAN implementation of VIA such as `VipPeerConnectRequest()`. The advantage with these calls is that, on the arrival of connection requests, they are queued by the GigaNet cLAN NIC, similar to that done by the Operating System in the `listen()` call. However, though these calls are non-blocking, they have been developed for the peer-to-peer communication model, where the accepting node knows exactly from whom the connection request is going to come. This does not fit in exactly with the client-server model of TCP, since in the client-server model, the server might not know the client from which the connection request is "expected". This difference is apparent in the wild-card `accept()` call, where the server can accept a connection from any client.

- *Connection Thread:* In this approach, we use an additional connection thread to deal with this mismatch (Figure 12). As soon as a `listen()` call is encountered, a separate thread (*Connection Thread*) is spawned. When this thread is created, it calls the `VipConnectWait()` function and waits for the connection request. On the other hand, the main thread can carry on with its computation. The issues to be noted in this approach are the CPU cycles used by the connection thread and the connection time. One approach is to make the connection thread poll for connections, which would result in a high CPU usage (about 50%) for the connection thread together with the synchronization cost between the two threads (up to 20μsecs). Alternatively, making the connection thread sleep would decrease the CPU cycles used, but would increase the connection time significantly.

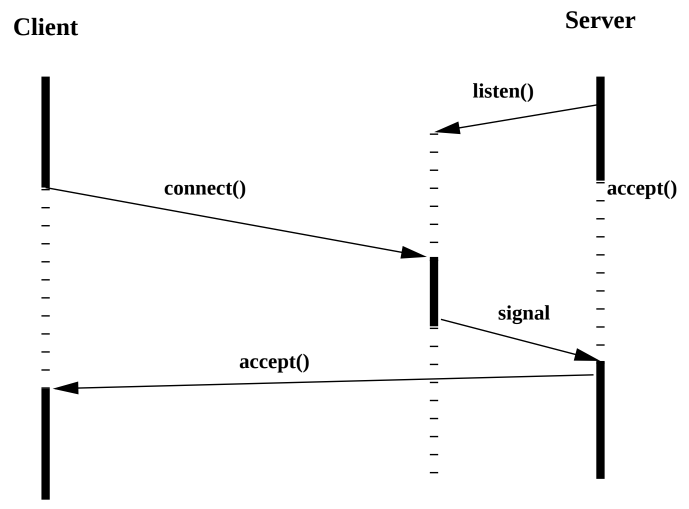


Figure 12: High Performance sockets over VIA: Separate connection thread approach

Problems with *fork()*: The *fork()* system call has an inherent property called copy-on-write. When a process forks to form another process, both of them share the same physical address till one of them attempts to write to it. As soon as one of them attempts to write to it, a copy of the physical address page is created. Now, the new physical address space is not registered, so this would return an error (Figure 13).

This problem requires us to make sure that a buffer is registered every time we try to send some data from it. However, the problem is not just restricted to this case. It is possible that the sender can modify the data during the actual sending. It is too restrictive if the user applications are not allowed to create child processes while they are using VIA.

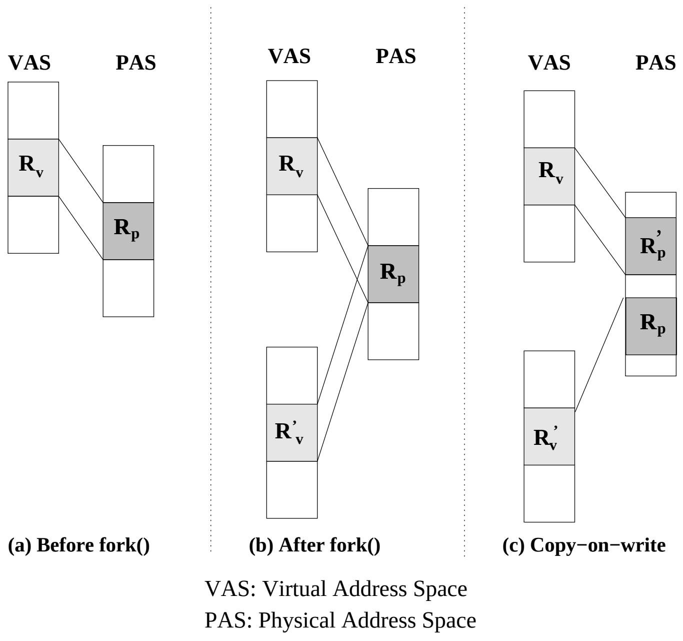


Figure 13: The Copy-on-Write problem with `fork()`

- *Copying Data:* In this approach, the data is copied into a temporary buffer before transmitting it. Since this buffer belongs to the sockets interface and is not written to by the process, the copy-on-write problem does not arise in this approach.

- *Blocking the* `send()` *call:* In this approach, the `send()` call is blocked till the entire data is transmitted by the sockets layer. In this approach, since the thread calling the `send()` call is blocked, it cannot write to the buffer through which the data is being sent. On the other hand, if the forked thread tries to write to the buffer from which the data is being sent, a new copy of the physical page is created and the unregistered copy is given to the forked thread. Therefore this does not effect the data transfer in progress. However, we have to make sure that each time we try to send data, the buffer we are sending from is registered.

Unlike the EMP protocol discussed in Section 4.2.1, high performance sockets over VIA require the memory buffers to be registered during data transmission or reception. This and several other such requirements with VIA might cause some amount of performance degradation in the high performance sockets layer. Next, we discuss some of the techniques that can be used to improve the performance of the sockets layer.

Registration/Copy Hybrid: VIA requires that data be sent from pinned buffers so that the pages are not swapped out during the course of the data transfer. Similar is the case on the receiver side. We have examined two potential solutions to address this problem.

- *Temporary Registered Buffer:* In this solution, a temporary buffer is registered initially. When `send()` is called, the data is copied into this temporary buffer and the data transfer is carried out from this buffer. Once the acknowledgment is received, the buffer is free for reuse and can be used for the next transfer.

- *User Buffer Copy:* In this solution, the user buffer itself is pinned and the data transfer done directly from the user buffer. Once the acknowledgment for this transfer is received, the buffer is free to be deregistered. In this case, the `send()` call is blocked till the data transfer completes, so that the buffer is not modified before the transfer completes.

Both these solutions have their own advantages and disadvantages. Memory registration is very costly for small messages, whereas memory copy becomes costlier for larger messages. On the receiver side, a copy is inevitable in order to support data streaming, but on the sender side a hybrid of these approaches is possible.

Lazy Memory Deregistration: Since the user buffer is being registered (for large message sizes) before sending the data, there is a possibility that the next time the user sends a message, it might be in the same buffer (or a subset of the buffer). This is quite common in applications. Based on this observation, certain enhancements for the implementation are possible. For large message sizes, when the `send()` is called, we register the user buffer and send the data, but do not deregister the buffer immediately. When the next `send()` is called, we check if this buffer is the same (or a subset) as the previously registered buffer. If it is, we continue with sending the data without registering the buffer. This enhancement does not affect the worst case performance, but improves the best case performance.

4.2.3 High Performance Sockets over InfiniBand

Due to the various similarities between VIA and InfiniBand, most of the design issues between the two high performance sockets layer implementations over these are similar. However, there are two broad changes for such a sockets implementation over InfiniBand as compared to VIA.

1. The High Performance Sockets implementation over InfiniBand has been standardized by the industry and termed as the Sockets Direct Protocol (SDP) [6]. This standard not only specifies implementations for the standard synchronous sockets API supported by the 2.4 and early 2.6 kernel versions of linux, but also an asynchronous sockets implementation. The asynchronous sockets API is used with the Windows Sockets library (WinSock) as well as some of the later 2.6 kernel implementations of linux. The interesting issue with asynchronous sockets is that the application would hand over the data

buffer to the network protocol and guarantee that it would not be touched till the data transfer is complete; this provides the protocol layer the flexibility to perform zero-copy data transmission efficiently. However, an issue to be noted here is that most applications have been written using the synchronous sockets API and need to be modified to utilize the asynchronous sockets API.

2. InfiniBand offers a richer functionality set than VIA. For example, RDMA Read or Remote Memory atomic operations supported by InfiniBand were not supported by VIA. While these are not critical for the basic high performance sockets implementation, these features open up interesting regions that can be efficiently tackled using such functionality. For example, RDMA read capabilities allow zero-copy data transmission for asynchronous sockets as we will see in the overview of SDP. Further, in Section 4.2.4, we will discuss some approaches for performing zero-copy data transfer for synchronous sockets using the RDMA Write, RDMA Read and Atomic capabilities of InfiniBand.

Sockets Direct Protocol (SDP) Overview: The SDP specifications focuses specifically on the wire protocol, finite state machine and packet semantics. Operating system issues, etc., can be implementation specific. It is to be noted that SDP supports only SOCK_STREAM or Streaming sockets semantics and not SOCK_DGRAM (datagram) or other socket semantics. SDP's Upper Layer Protocol (ULP) interface is a byte-stream that is layered on top of InfiniBand's Reliable Connection (RC) message-oriented transfer model. The mapping of the byte stream protocol to InfiniBand message-oriented semantics was designed to enable ULP data to be transfered by one of two methods: through intermediate private buffers (Bcopy) or directly between ULP buffers (Zcopy). A mix of InfiniBand Send and RDMA mechanisms are used to transfer ULP data. Zcopy uses RDMA reads or writes, transferring data between RDMA buffers (which typically belong to the ULP). Bcopy uses InfiniBand sends, transferring data between send and receive private buffers. SDP has two types of buffers:

- *Private Buffers:* Used for transmission of all SDP messages and ULP data that is to be copied into the receive ULP buffer. The Bcopy data transfer mechanism is used for this traffic.
- *RDMA Buffers:* Used when performing Zcopy data transfer. ULP data is intended to be RDMAed directly from the Data Source's ULP buffer to the Data Sink's ULP buffer.

4.2.4 Zero Copy Sockets over IBA: Emulating Asynchronous Sockets with Synchronous Sockets

As mentioned in Section 4.2.3, asynchronous sockets allow applications to present the data buffer to the protocol layer and get completion notifications asynchronously. This capability provides several advantages for asynchronous sockets. First, this allows the protocol layer to try to perform zero-copy data transfer from the user buffer and give a notification to the user only after the data transfer has complete. Second, this allows the protocol layer to post multiple outstanding data transfer requests to the network adapter allowing significant improvements in the data transfer throughput.

However, synchronous sockets do not have any such benefits. Further, some operating systems such as Linux only supported synchronous sockets till very recently. Due to this, several applications (e.g., those in the data-center domain such as Apache, MySQL, etc.) have been implemented with synchronous sockets. This mandates the requirement for an efficient implementation of synchronous I/O API in high performance sockets.

In this section, we present a new design for high performance sockets over InfiniBand, termed as zero-copy SDP (ZSDP), to allow synchronous sockets emulate the various capabilities of asynchronous sockets such as allowing zero-copy data transfer, posting multiple outstanding data transfer requests, etc. The basic idea of this design is to write protect the memory region from which the data transmission request has been provided and send it to the receiver end. The receiver on seeing this request performs an RDMA read of the data from the original location to the destination buffer.

However, there are several issues that need to be handled before this scheme can be utilized. Some of the most important issues are as follows.

1. *Handling application writes:* As mentioned earlier, in our scheme, we protect the memory region for which the data transmission request was provided. However, if the application tries to write to the buffer before the data transmission is complete, this needs to be handled. In order to handle this, we pre-allocate a large central memory region. If the application tries to write to the send buffer, since the page is protected, a page fault interrupt is generated. The sockets library handles this interrupt, copies the data to the pre-registered buffer and returns the control back to the application. The actual data transfer now is carried out from the pre-registered temporary buffer in this case. It is to be noted that in this case, the data copy cannot be avoided. So, we would perform similar to the previous implementation, except for the additional time required to handle the page fault interrupt.

2. *Synchronization between the sender and the receiver:* As soon as the receiver process receives a notification from the sender about the buffer from which the data transmission is to be carried out, it initiates an RDMA read from this buffer to the destination buffer. However, in the meanwhile if the application on the sender side tries to write to the send buffer, this data needs to be copied to a pre-registered temporary buffer and data transmission carried out from this. This mandates the need from some kind of a synchronization between the sender and the receiver processes. In our approach, we combine the synchronization phase with the receiver notification phase by using remote memory atomic operations supported by InfiniBand. The receiver initially sets a NULL value in a pre-decided location. To inform the receiver that the send buffer is available, the sender just performs an atomic compare-and-swap on this location with the send buffer address. If the atomic is successful, the sender just waits for the receiver to complete the data transmission (using RDMA read) and inform about it. If, however, the

atomic fails, it means that the receiver had arrived earlier and has swapped the memory location with the address of the receive buffer. In this case, the sender just transfers the data by doing an RDMA write to this location.

4.3 Analyzing High Performance Sockets in various environments: Issues and Impacts

In this section, we analyze the impacts of high performance sockets in various environments and the limitations associated with the high performance sockets implementations. In particular, we analyze: (i) impact on data-intensive applications which tune the communication to computation ratio in order to reduce the impact of network performance, (ii) impact of memory copies associated with the sockets layer on the memory traffic generated on 10-gigabit networks and (iii) impact of the two-sided communication of the sockets API in computationally stressed environments.

4.3.1 Impact of High Performance Sockets on Data Intensive Applications

Overview of Data-Intensive Applications: A challenging issue in supporting data intensive applications on cluster environments is that large volumes of data should be efficiently moved between processor memories. Data movement and processing operations should also be efficiently coordinated by a runtime support to achieve high performance. Together with a requirement in terms of good performance, such applications also require guarantees in performance, scalability with these guarantees, and adaptability to heterogeneous environments and varying resource availability.

Component-based frameworks [20, 29, 46, 49] have been able to provide a flexible and efficient environment for data intensive applications on distributed platforms. In these frameworks, an application is developed from a set of interacting software components. Placement of components onto computational resources represents an important degree of flexibility in optimizing application performance. Data-parallelism can be achieved by executing multiple copies of a component across a cluster of storage and processing nodes [20]. Pipelining is another possible mechanism for performance improvement. In many data intensive applications, a dataset can be partitioned into *user-defined data chunks.* Processing of the chunks can be pipelined. While computation and communication can be overlapped in this manner, the performance gain also depends on the granularity of computation and the size of data messages (data chunks). Small chunks would likely result in better load balance and pipelining, but a lot of messages are generated with small chunk sizes. Although large chunks would reduce the number of messages and achieve higher communication bandwidth, they would likely suffer from load imbalance and less pipelining.

An example of data-intensive applications is digitized microscopy. We use the salient characteristics of this application as a motivating scenario and a case study. The software support required to store, retrieve, and process digitized slides to provide interactive response times for the standard behavior of a physical microscope is a challenging issue [10, 28]. The main difficulty stems from handling of large volumes of image data, which can range from a few hundreds of Megabytes to several Gigabytes per image. At a basic level, the software system should emulate the use of a physical microscope, including continuously moving the stage and changing magnification. The processing of client queries requires projecting high resolution data onto a grid of suitable resolution and appropriately composing pixels mapping onto a single grid point.

Consider a visualization server for digitized microscopy. The client to this server can generate a number of different types of requests. The most common ones are *complete update queries*, by which a completely new image is requested, and *partial update query*, by which the image being viewed is moved slightly or zoomed into. The server should be designed to handle both types of queries.

Processing of data in applications that query and manipulate scientific datasets can often be represented as an acyclic, coarse grain data flow, from one or more data sources (e.g., one or more datasets distributed across storage systems) to processing nodes to the client. For a given query, first the data of interest is retrieved from the corresponding datasets. The data is then processed via a sequence of operations on the processing nodes. For example, in the digitized microscopy application, the data of interest is processed through *Clipping*, *Subsampling*, *Viewing* operations [18, 20]. Finally, the processed data is sent to the client.

Data forming parts of the image are stored in the form of blocks or data chunks for indexing reasons, requiring the entire block to be fetched even when only a part of the block is required. Figure 14 shows a complete image being made up of a number of blocks. As seen in the figure, a partial update query (the rectangle with dotted lines in the figure) may require only part of a block. Therefore, the size and extent of a block affect the amount of unnecessary data retrieved and communicated for queries.

Performance Issues in Runtime Support for Data Intensive Applications:

- *Basic Performance Considerations:* For data-intensive applications, performance can be improved in several ways. First, datasets can be declustered across the system to achieve parallelism in I/O when retrieving the data of interest for a query. With good declustering, a query will hit as many disks as possible. Second, the computational power of the system can be efficiently used if the application can be designed to exploit data parallelism for processing the data. Another factor that can improve the performance, especially in interactive exploration of datasets, is *pipelined* execution. By dividing the data into chunks and pipelining the processing of these chunks, the overall execution time of the application can be decreased. In many applications, pipelining also provides a mechanism to gradually create the output data product. In other words, the user does not have to wait for the processing of the query to be completed; partial results can be gradually generated. Although this may not actually reduce the overall response time, such a feature is very effective in an interactive setting, especially if the region of interest moves continuously.

- *Message Granularity vs. Performance Guarantee:* The granularity of the work and the size of data chunks affects the performance of pipelined execution. The chunk size should be carefully selected by

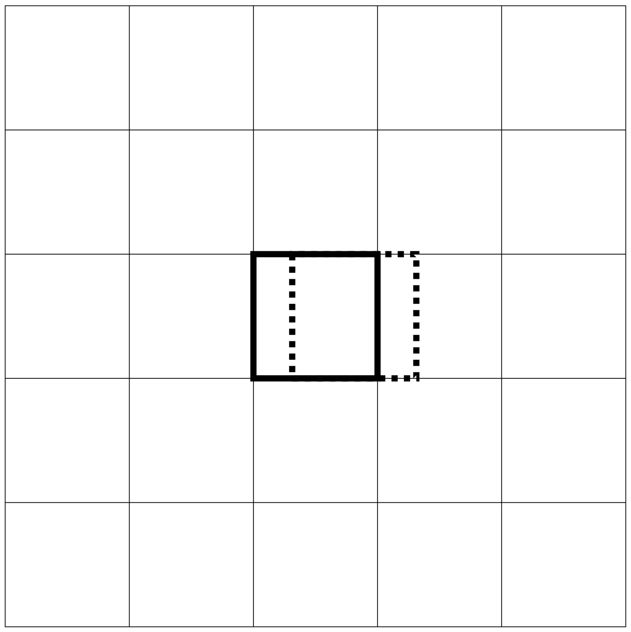


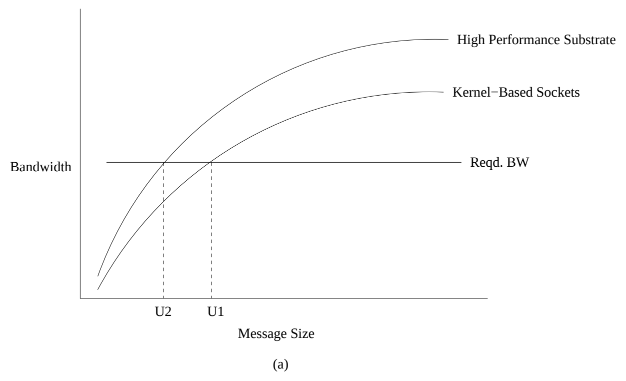
Figure 14: Data Intensive Applications: Partial and Complete update queries in digital image visualization applications

taking into account the network bandwidth and latency (the time taken for the transfer of a message including the protocol processing overheads at the sender and the receiver ends). As the chunk size increases, the number of messages required to transfer the data of interest decreases. In this case, bandwidth becomes more important than latency. However, with a bigger chunk size, processing time per chunk also increases. As a result, the system becomes less responsive, i.e., the frequency of partial/gradual updates decreases. On the other hand, if the chunk size is small, the number of messages increases. As a result, latency may become a dominant factor in the overall efficiency of the application. Also, smaller chunks can result in better load balance among the copies of application components, but communication overheads may offset the performance gain. Having large blocks allows a better response time for a complete update query due to improved bandwidth. However, during a partial update query, this would result in more data being fetched and eventually being wasted. On the other hand, with small block size, a partial update query would not retrieve a lot of unnecessary data, but a complete update query would suffer due to reduced bandwidth. In addition to providing a higher bandwidth and lower latency, high performance sockets layers have other interesting features as demonstrated in Figures 15(a) and 15(b). Figure 15(a) shows that high performance sockets achieve a required bandwidth at a much lower message size compared to kernel-based sockets layers such as TCP/IP. For instance, for attaining bandwidth 'B', kernel-based sockets need a message size of U1 bytes, whereas high performance sockets require a lower message size of U2 bytes. Using this information in Figure 15(b), we observe that high performance sockets result in lower message latency (from L1 to L2) at a message size of U1 bytes. We also observe that high performance sockets can use a message size of U2 bytes (from Figure 15(a)), hence further reducing the latency (from L2 to L3) and resulting in better performance. While this can be interpreted as an encouraging result for high performance sockets implementations, this also means that several applications such as these, which have been designed keeping the communication characteristics of TCP/IP in mind, might not be able to take advantage of high performance sockets implementations fully. For example, if the application does not make modifications to its chunk size with respect to the performance of the high performance sockets and continues to use the chunk sizes designed with respect to the performance of TCP/IP, the performance gain might be minimal.

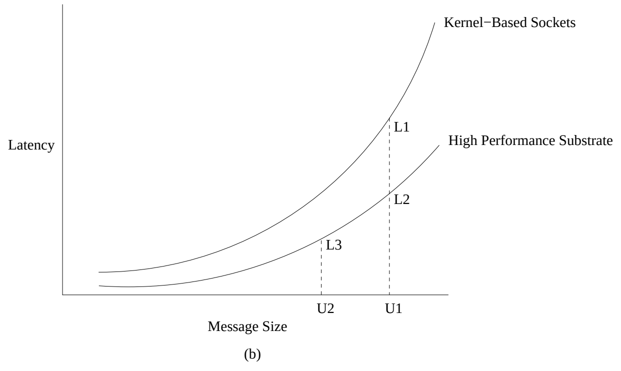
- *Heterogeneity and Load Balancing:* Heterogeneity arises in several situations. First, the hardware environment may consist of machines with different processing power and memory capacity. Second, the resources can be shared by other applications. As a result, the availability of resources such as CPU and memory can vary dynamically. In such cases, the application should be structured to accommodate the heterogeneous nature of the environment. The application should be optimized in its use of shared resources and be adaptive to the changes in the availability of the resources. This requires the application to employ adaptive mechanisms to balance the workload among processing nodes depending on the computation capabilities of each of them. A possible approach is to adaptively schedule data and application computations among processing nodes. The data can be broken up into chunks so as to allow pipelining of computation and communication. In addition, assignment of data chunks to processing units can be done using a demand-driven scheme so that faster nodes can get more data to process. If a fast node becomes slower (e.g., due to processes of other applications), the underlying load balancing mechanism should be able to detect the change in resource availability quickly. For such applications, which adapt themselves based on the system behavior, the benefits high performance sockets can provide are an open issue. As we will see in Section 5.4, in some scenarios applications such as these have been designed to completely overlap computation with communication in order to not be affected by the performance of TCP/IP. For such applications, the performance gain is expected to minimal or none.

4.3.2 Sockets vs RDMA: Memory Traffic Analysis

As discussed in Section 4.2.4, several zero-copy techniques have been developed by researchers to develop high performance sockets layers. However, these are mostly relevant to asynchronous sockets. On the other hand, most applications have been written using the synchronous sockets interface and will not be able to utilize such zero-copy techniques. Further, though some techniques (Section 4.2.4) are possible to avoid copies for synchronous sockets in certain scenarios, this cannot ensure zero copy data transfer in all cases.



(a)



(b)

Figure 15: Latency and Bandwidth tradeoffs in data communication for data intensive applications

Based on this, in this section, we analyze the impact of memory copies associated with the sockets layer on the memory traffic generated in 10-gigabit and faster networks. We study the impact of cache misses on the amount of memory traffic generated; we estimate the amount of memory traffic for a typical throughput test.

Memory traffic comprises of two components: Front Side Bus (FSB) reads and writes generated by the CPU(s) and DMA traffic generated through the I/O bus by other devices (NIC in our case). We study the memory traffic associated with the transmit path and the receive paths separately. Further, we break up each of these paths into two cases: (a) Application buffer fits in cache and (b) Application buffer does not fit in cache. These two cases lead to very different memory traffic analyses, which we will study in this section. Figures 16a and 16b illustrate the memory accesses associated with network communication.

Transmit Path (application buffer fits into L2-cache): As mentioned earlier, in the transmit path, TCP copies the data from the application buffer to the socket buffer. The NIC then DMAs the data from the socket buffer and transmits it. For the case when the application buffer fits in the cache, the following are the steps involved on the transmission side:

- *CPU reads the application buffer:* Most micro-benchmark tests are written such that the application buffer is reused on every iteration. The buffer is fetched to cache once and it remains valid throughout the experiment. Subsequent reading of the buffer gets its copy from the cache. So, there is no data traffic on the FSB for this step.

- *CPU writes to the socket buffer:* The default socket buffer size for most kernels including Linux and Windows Server 2003 is 64KB, which fits in cache (on most systems). In the first iteration, the socket buffer is fetched to cache and the application buffer is copied into it. In the subsequent iterations, the socket buffer stays in one of *Exclusive*, *Modified* or *Shared* states, i.e., it never becomes *Invalid.*

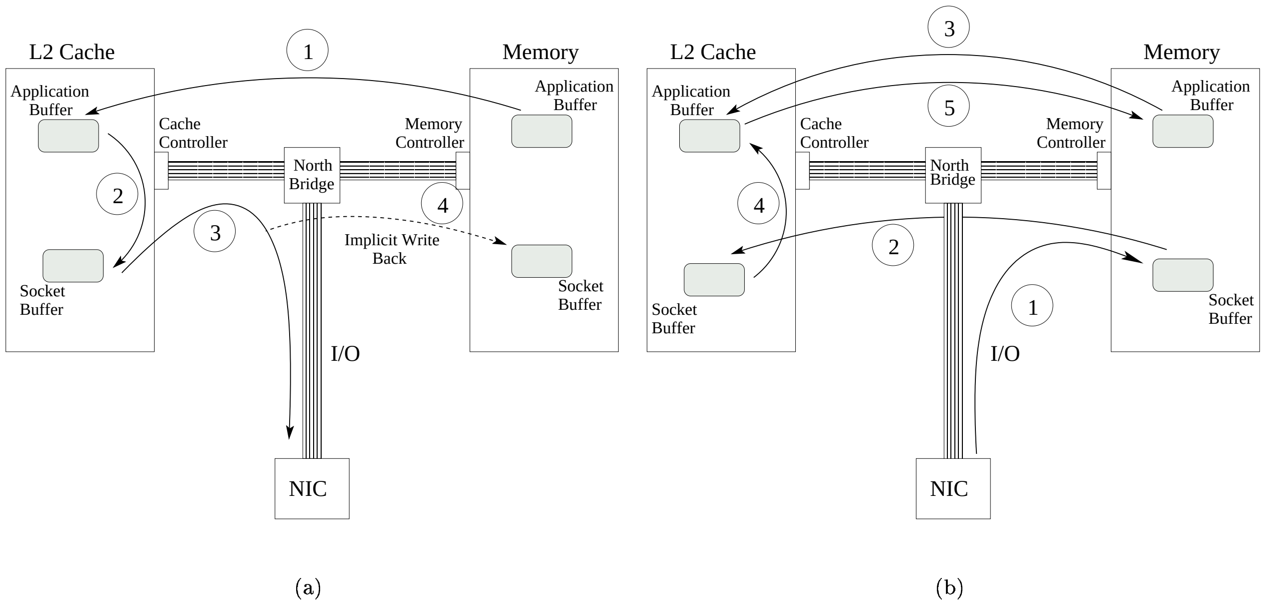


Figure 16: Memory traffic Associated with a typical network message traffic: (a) Transmit path and (b) Receive path

Further, any change of the socket buffer state from one to another of these three states just requires a notification transaction or a Bus Upgrade from the cache controller and generates no memory traffic.

- *NIC does a DMA read of the socket buffer:* Most current memory I/O controllers allow DMA reads to proceed from cache. Also, as an optimization, most controllers do an *implicit write back* of dirty cache lines to memory during a DMA read. Since the socket buffer is dirty at this stage, this generates one byte of memory traffic during the DMA operation.

Based on these three steps, in the case where the application buffer fits in the cache, there is one byte of memory traffic for every byte of network data transmitted.

However, due to the set associative nature of some caches, it is possible that some of the segments corresponding to the application and socket buffers be mapped to the same cache line. This requires that these parts of the socket buffer be fetched from memory and written back to memory on every iteration. In the worst case, this might sum up to as many as three additional memory transactions (one additional explicit write back and one fetch of the socket and the application buffers to cache). It is to be noted that, even if a cache line corresponding to the socket buffer is evicted to accommodate another cache line, the amount of memory traffic due to the NIC DMA does not change; the only difference would be that the traffic would be a memory read instead of an implicit write back. Summarizing, in the case where the application buffer fits in cache, in theory there can be between 1-4 bytes of data transferred to or from memory for every byte of data transferred on the network. However, we assume that the cache mapping and implementation are efficient enough to avoid such a scenario and do not expect this to add any additional memory traffic.

Transmit Path (application buffer does not fit into L2-cache): For the case when the application buffer does not fit into the cache, the following are the steps involved on the transmission side:

- *CPU reads the application buffer:* The application buffer has to be fetched every time to cache since it does not completely fit into it. However, it does not have to be written back to memory each time since it is only used for copying into the socket buffer and is never dirtied. Hence, this operation requires a byte of data to be transferred from memory for every byte transferred over the network.

- *CPU writes to the socket buffer:* Again, we assume that the socket buffer is small enough to fit into cache. So, once the socket buffer is fetched to cache, it should stay valid throughout the experiment and require no additional memory traffic. However, the large application buffer size can force the socket buffer to be pushed out of cache. This can cause up to 2 bytes of memory traffic per network byte (one transaction to push the socket buffer out of cache and one to fetch it back).

- *NIC does a DMA read of the socket buffer:* Similar to the case where the application buffer fits into the cache, the socket buffer is dirty at this point. When a DMA request from the NIC arrives, the segment of the socket buffer corresponding to the request, can be either in cache (dirtied) or in memory. In the first case, during the DMA, the memory controller does an implicit write back of the cache lines to memory. In the second case, the DMA takes place from memory. So, in either case, there would be one byte of data transferred either to or from memory for every byte of data transferred on the network. Based on these, we can expect the memory traffic required for this case to be between 2 to 4 bytes for every byte of data transferred over the network. Also, we can expect this value to move closer to 4 as the size of the application buffer increases (forcing more cache misses for the socket buffer).

As discussed earlier, due to the set-associative nature of the cache, it might be required that the socket buffer be fetched from and written back to memory on every iteration. Since the socket buffer being pushed out of cache and fetched back to cache is already accounted for (due to the large application buffer), the estimated amount of memory traffic does not change in this case.

Receive Path (application buffer fits into L2-cache): The memory traffic associated with the receive path is simpler compared to that of the transmit path. We again consider 2 cases for the receive path: (a) Application buffer fits into cache and (b) Application buffer does not fit into cache. For the case when the application buffer fits into cache, the following are steps involved on the receive path:

- *NIC does a DMA write into the socket buffer:* When the data arrives at the NIC, it does a DMA write of this data into the socket buffer. During the first iteration, if the socket buffer is present in cache and is dirty, it is flushed back to memory by the cache controller. Only after the buffer is flushed out of the cache is the DMA write request allowed to proceed. We'll see in the next couple of steps that the socket buffer will be fetched to the cache so that the data be copied into the application buffer. So for all subsequent iterations, during the NIC DMA write, the socket buffer can be expected to be in the cache. Also, since it is only being used to copy data into the application buffer, it will not be dirtied. Thus, the DMA write request would be allowed to proceed as soon as the socket buffer in the cache is invalidated by the memory I/O controller (Figure 16), i.e., the socket buffer does not need to be flushed out of cache for the subsequent iterations. This sums up to one transaction to the memory during this step.

- *CPU reads the socket buffer:* At this point, the socket buffer is not present in cache (even if the buffer was present in the cache before the iteration, it has to be evicted for the previous step). So, the CPU needs to read the socket buffer into memory. This requires one transaction to the memory during this step.

- *CPU writes to application buffer:* Since the application buffer fits into cache, there is no additional memory traffic during this operation (again, we assume that the cache policy is efficient enough to avoid cache misses due to cache line mappings in set associative caches).

Based on these three steps, we can expect 2 bytes of memory traffic for every byte transferred over the network.

Receive Path (application buffer does not fit into L2-cache): For the case when the application buffer does not fit into the cache, the following steps are involved on the receive path:

- *NIC does a DMA write into the socket buffer:* Since this step involves writing data directly to memory, it is not affected by the cache size or policy and would be similar to the case when the application buffer fits into cache. Thus, this step would create one transaction to the memory.

- *CPU reads the socket buffer:* Again, at this point the socket buffer is not present in cache, and has to be fetched, requiring one transaction from the memory.

- *CPU writes to application buffer:* Since the application buffer does not fit into cache entirely, it has to be fetched in parts, data copied to it, and written back to memory to make room for the rest of the application buffer. Thus, there would be two transactions to and from the memory for this step (one to fetch the application buffer from memory and one to write it back).

This sums up to 4 bytes of memory transactions for every byte transferred on the network for this case. It is to be noted that for this case, the number of memory transactions does not depend on the cache policy. Table 1 gives a summary of the memory transactions expected for each of the above described cases. *Theoretical* refers to the possibility of cache misses due to inefficiencies in the cache policy, set associativity, etc. *Practical* assumes that the cache policy is efficient enough to avoid cache misses due to memory to cache mappings.

Table 1: Memory to Network traffic ratio

	fits in cache	does not fit in cache
Transmit (Theoretical)	1-4	2-4
Transmit (Practical)	**1**	**2-4**
Receive (Theoretical)	2-4	4
Receive (Practical)	**2**	**4**

4.3.3 Impact of High Performance Sockets in Computationally Stressed Environments

In this section, we analyze the impact of the two-sided communication primitives of the sockets API in computationally stressed environments. We used multi-tier data-center as an example of such environments for our studies.

Overview of Cluster-based Data-Centers and their Requirements: With increasing adoption of the Internet as primary means of electronic interaction and communication, E-portal and E-commerce, highly scalable, highly available and high performance web servers, have become critical for companies to reach, attract, and keep customers. Cluster-based data-centers, consisting of multiple tightly interacting layers known as tiers, have become a central requirement to providing such services.

Each tier in a multi-tier data-center can contain multiple physical nodes. Requests from clients are load-balanced on to the nodes in the proxy tier. This tier mainly does caching of content generated by the other back-end tiers. The other functionalities of this tier include embedding inputs from various application servers into a single HTML document (for framed documents for example), balancing the requests sent to the back-end based on certain pre-defined algorithms such as the load on the different nodes and other such services.

The second tier consists of two kinds of servers. First, those which host static content such as documents, images, music files and others which do not change with time. These servers are typically referred to as web-servers. Second, those which compute results based on the query itself and return the computed data in the form of a static document to the users. These servers, referred to as application servers, usually handle compute intensive queries which involve transaction processing and implement the data-center business logic.

The last tier consists of database servers. These servers hold a persistent state of the databases and other data repositories. These servers could either be compute intensive or I/O intensive based on the query format. For simple queries, such as search queries, etc., these servers tend to be more I/O intensive requiring a number of fields in the database to be fetched into memory for the search to be performed. For more complex queries, such as those which involve joins or sorting of tables, these servers tend to be more compute intensive.

Other than these three tiers, various data-center models specify multiple other tiers which either play a supporting role to these tiers or provide new functionalities to the data-center. For example, the CSP architecture [51] specifies an additional edge service tier which handles security, caching, SAN enclosure of packets for TCP termination and several others. Figure 17 represents a typical multi-tier data-center.

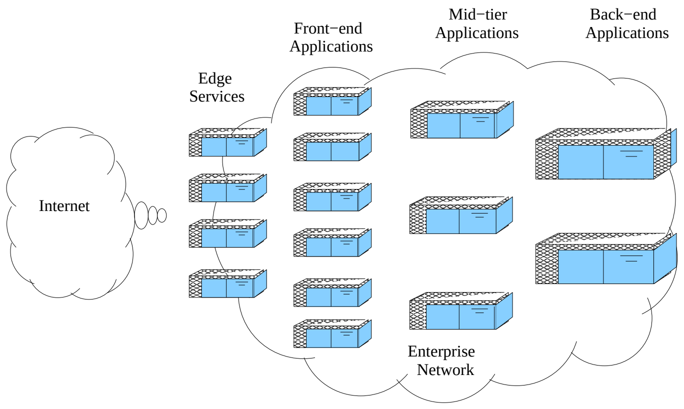


Figure 17: Traditional Cluster-based Data-Center (courtesy CSP Architecture)

Shared Data-Centers: In the past few years several researchers have proposed and configured data-centers providing multiple independent services, known as shared data-centers [30, 31]. For example, several ISPs and other web service providers (e.g., HP, Yahoo, IBM, etc.) host multiple unrelated web-sites on their data-centers allowing potential differentiation in the service provided to each of them.

Modern day data-centers have several requirements from the network interconnects they use as well as the protocol support offered by these interconnects. For this proposal, we concentrate on mainly two requirements within the data-center environment, namely, caching dynamic web content and efficient resource management for shared data-centers using dynamic reconfiguration of resources.

Caching Dynamic Web Content: Caching dynamic content, typically known as *Active Caching* [26] at various tiers of a multi-tier data-center is a well known method to reduce the computation and communication overheads. However, it has its own challenges: issues such as cache consistency and cache coherence become more prominent. In the state-of-art data-center environment, these issues are handled based on the type of data being cached. For dynamic data, for which relaxed consistency or coherency is permissible, several methods like TTL [38], Adaptive TTL [32], and Invalidation [42] have been proposed. However, for data like stock quotes or airline reservation, where old quotes or old airline availability values are not acceptable, strong consistency and coherency is essential.

- *Web Cache Consistency:* Cache consistency refers to a property of the responses produced by a single logical cache, such that no response served from the cache will reflect older state of the server than that reflected by previously served responses, i.e., a consistent cache provides its clients with non-decreasing views of the server's state. In a multi-tier data-center environment many nodes can access data at the same time (*concurrency*). Data consistency provides each user with a consistent view of the data, including all visible (committed) changes made by the user's own updates and the updates of other users. That is, either all the nodes see a completed update or no node sees an update. Hence, for strong consistency, stale view of data is permissible, but partially updated view is not. Several different levels of consistency are used based on the nature of data being used and its consistency requirements. For example, for a web site that reports football scores, it may be acceptable for one user to see a score, different from the scores as seen by some other users, within some frame of time. There are a number of methods to implement this kind of weak or lazy consistency models. The *Time-to-Live (TTL)* approach, also known as the Δ-*consistency* approach, proposed with the HTTP/1.1 specification, is a popular weak consistency (and weak coherence) model currently being used. This approach associates a *TTL* period with each cached document. On a request for this document from the client, the front-end node is allowed to reply back from their cache as long as they are within this *TTL* period, i.e., before the *TTL* period expires. This guarantees that document cannot be more *stale* than that specified by the *TTL* period, i.e., this approach guarantees that staleness of the documents is bounded by the *TTL* value specified. Researchers have proposed several variations of the *TTL* approach including *Adaptive TTL* [32] and *MONARCH* [44] to allow either dynamically varying *TTL* values (as in *Adaptive TTL*) or document category based *TTL* classification (as in *MONARCH*). There has also been considerable amount of work on *Strong Consistency* algorithms [23, 22].

- *Web Cache Coherence:* Cache coherence refers to the average *staleness* of the documents present in the cache, i.e., the time elapsed between the current time and the time of the last update of the document in the back-end. A cache is said to be strong coherent if its average *staleness* is *zero*, i.e., a client would get the same response whether a request is answered from cache or from the back-end. Typically, when a request reaches the proxy node, the cache is checked for the file. If the file was previously requested and cached, it is considered a cache hit and the user is served with the cached file. Otherwise the request is forwarded to its corresponding server in the back-end of the data-center. The maximal hit ratio in proxy caches is about 50% [53]. Majority of the cache misses are primarily due to the dynamic nature of web requests. Caching dynamic content is much more challenging than static content because the cached object is related to data at the back-end tiers. This data may change, thus invalidating the cached object and resulting in a cache miss. The problem providing consistent caching for dynamic content has been well studied and researchers have proposed several weak as well as strong cache consistency algorithms [23, 22, 61]. However, the problem of maintaining cache coherence has not been studied as much. The two popular coherency models used in the current web are *immediate or strong coherence* and *bounded staleness*. The *bounded staleness* approach is similar to the previously discussed *TTL* based approach. Though this approach is efficient with respect to the number of cache hits, etc., it only provides a weak cache coherence model. On the other hand, *immediate coherence* provides a strong cache coherence. With *immediate coherence*, caches are forbidden from returning a response other than that which would be returned were the origin server contacted. This guarantees semantic transparency, provides *Strong Cache Coherence*, and as a side-effect also guarantees *Strong Cache Consistency.* There are two widely used approaches to support *immediate coherence.* The first approach is pre-expiring all entities (forcing all caches to re-validate with the origin server on every request). This scheme is similar to a no-cache scheme. The second approach, known as *client-polling*, requires the front-end nodes to inquire from the back-end server if its cache is valid on every cache hit. The no-caching approach to maintain *immediate coherence* has several disadvantages:

– Each request has to be processed at the home node tier, ruling out any caching at the other tiers

– Propagation of these requests to the back-end nodes over traditional protocols can be very expensive

– For data which does not change frequently, the amount of computation and communication overhead incurred to maintain strong coherence could be very high, requiring more resources

These disadvantages are overcome to some extent by the *client-polling* mechanism. In this approach, the proxy server, on getting a request, checks its local cache for the availability of the required document. If it is not found, the request is forwarded to the appropriate application server in the inner tier and there is no cache coherence issue involved at this tier. If the data is found in the cache, the proxy server checks the *coherence status* of the cached object by contacting the back-end server(s). If there were updates made to the dependent data, the cached document is discarded and the request is forwarded to the application server tier for processing. The updated object is now cached for future use. Even though this method involves contacting the back-end for every request, it benefits from the fact that the actual data processing and data transfer is only required when the data is updated at the back-end. This scheme can potentially have significant benefits when the back-end data is not updated very frequently.

While cache coherence can be maintained using the *client-polling* mechanism, an efficient implementation of this scheme would depend on the performance as well as the features provided by the underlying middleware or protocol layer. As mentioned earlier, by their very nature, application server nodes are compute intensive. Execution of CGI-Scripts, business-logic, servlets, database processing, etc., are typically very taxing on the server CPUs. In this case, high performance sockets lies on a clear disadvantage due to its two-sided communication model. Though the amount of communication needed by the server is very little, the communicating process might be starved for CPU in the presence of several other compute intensive applications. On the other hand, protocols providing one-sided communication primitives such as RDMA Read, etc., can perform such tasks with far little overhead.

Efficient Resource Management for shared data-centers using Dynamic Reconfiguration of Resources: The increase in the number of unrelated services or websites hosted by a data-center results in a growing fragmentation of resources available and ultimately in the degradation of the performance provided by the data-center. Over-provisioning of nodes in the data-center for each service provided is a widely used approach. In this approach, nodes are alloted to each service depending on the worst case estimates of the load expected and the nodes available in the data-center. For example, if a data-center hosts two web-sites, each web-site is provided with a fixed subset of nodes in the data-center based on the traffic expected for that web-site. It is easy to see that though this approach gives the best possible performance, it might incur severe under utilization of resources especially when the traffic is bursty and directed to a single web-site. In this context, we present a novel design to provide dynamic reconfigurability of the nodes in the data-center environment. This technique enables the nodes in the data-center environment to efficiently adapt their functionality based on the system load and traffic pattern. Dynamic reconfigurability attempts to provide benefits in several directions: (i) cutting down the time needed for configuring and assigning the resources available by dynamically transferring the traffic load to the best available node/server, (ii) improving the performance achievable by the data-center by reassigning under-utilized nodes to loaded services, (iii) cutting down the cost of the data-center by reducing over-provisioning of nodes and improving the utilization of the resources available inside the data-center and several others.

While reconfigurability is a widely used technique for clusters, the data-center environment poses several interesting challenges for the design and implementation of such a scheme. While there are several approaches for implementing dynamic reconfigurability and issues related to each of those approaches, we would like to point out only the ones which are most relevant to the context of this proposal, i.e., the issues related to the two-sided communication of the sockets API in this scenario. Some of the major design challenges in dynamic reconfigurability with regard to the communication mechanism are listed below (other design challenges can be found in [48]):

- *Load-Balancer based Reconfiguration:* Two different approaches could be taken for reconfiguring the nodes: Server-based reconfiguration and Load-balancer based reconfiguration. In server-based reconfiguration, when a particular server detects a significant load on itself, it tries to reconfigure a relatively free node that is currently serving some other web-site content. Though intuitively the loaded server itself is the best node to perform the reconfiguration (based on its closeness to the required data and the number of messages required), performing reconfiguration on this node adds a significant amount of load to an already loaded server. Due to this reason, reconfiguration does not happen in a timely manner and the overall performance is affected adversely. On the other hand, in load-balancer based reconfiguration, the edge servers (functioning as load-balancers) detect the load on the servers, find a free server to alleviate the load on the loaded server and perform the reconfiguration themselves. Since the shared information like load, server state, etc., is closer to the servers, this approach incurs the cost of requiring more network transactions for its operations.

- *System Wide Shared State:* The various nodes in the data-center participate in the dynamic reconfiguration scheme in a distributed manner. However, the decision each node needs to make is pertinent to the global state of the system and cannot be made based on the view of a single node. So, these nodes need to communicate with each other to share such information regarding the system load, current configuration of the system, etc. Further, these communications tend to be asynchronous in nature. For example, the server nodes are not aware about when a particular load-balancer might require their state information.

An interesting point to note in this communication pattern is the amount of replication in the information exchanged between the nodes. For example, let us consider a case where the information is being shared between the web-site 'A' and the load-balancers in the shared data-center. Here, each node serving web-site 'A' provides its state information to each one of the load-balancing nodes every time they need it, i.e., the same information needs to be communicated with every node that needs it.

Based on these communication patterns, intuitively a global shared state seems to be the ideal environment for efficient distribution of data amongst all the nodes. In this architecture each node can write its relevant information into the shared state and the other nodes can asynchronously read this information without interrupting the source node. This architecture essentially depicts a producer-consumer scenario for non-consumable resources.

One approach for implementing such a shared state, is by distributing the data across the physical memories of various nodes and allowing the nodes in the data-center to read or write into these memory locations. While an implementation of such a logical shared state is possible using a high performance sockets interface (with the modules explicitly reading and communicating the data upon request from other nodes), such an implementation would lose out on all the benefits a shared state could provide. In particular: (i) All communication needs to be explicitly performed by the server nodes by sending (replicated) information to each of the load-balancers and (ii) Asynchronous requests from the nodes need to be handled by either using a signal based mechanism (using the SIGIO signal handler) or by having a separate thread block for incoming requests, both of which require the server node host intervention.

On the other hand, several networks such as InfiniBand provide several advanced features such as one-sided communication operations, including RDMA operations. In an implementation based on this, each node writes information related to itself on its local memory. Other nodes needing this information can directly read this information using an *RDMA read* operation without disturbing this node at all. This implementation of a logical shared state retains the efficiencies of the initially proposed shared state architecture, i.e., each node can write data into its shared state and the other nodes can read data asynchronously from the shared state without interrupting the source node.

- *Shared State with Concurrency Control:* The logical shared state described above is a very simplistic view of the system. The nodes use the information available and change the system to the best possible configuration. However, for using this logical shared state, several issues need to be taken into consideration.

As shown in Figure 18, each load-balancer queries the load on each server at regular intervals. On detecting a high load on one of the servers, the load-balancer selects a lightly loaded node serving a different web-site, and configures it to ease the load on the loaded server. However, to avoid multiple simultaneous transitions and hot-spot effects during reconfiguration, additional logic is needed. In our design, we propose an architecture using a two-level hierarchical locking with dual update counters to address these problems.

As shown in Figure 19, each web-site has an unique internal lock. This lock ensures that exactly one of the multiple load-balancers handling requests for the same web-site, can attempt a conversion of a server node, thus avoiding multiple simultaneous conversions. After acquiring this internal lock (through a remote atomic compare-and-swap operation), the load-balancer selects a lightly loaded server and performs a second remote atomic operation to configure that server to serve the loaded web-site. This second atomic operation (atomic compare-and-swap) also acts as a mutually exclusive lock between load-balancers that are trying to configure the free server to serve their respective web-sites.

It is to be noted that after a reconfiguration is made, some amount of time is taken for the load to get balanced. However, during this period of time other load balancers can still detect a high load on the servers and can possibly attempt to reconfigure more free nodes. To avoid this unnecessary reconfiguration of multiple nodes, each relevant load-balancer needs to be notified about any recent reconfiguration done, so that it can wait for some amount of time before it checks the system load and attempts a reconfiguration. In our design, each load-balancer keeps a *local update counter* and a *shared update counter* to keep track of all reconfigurations. Before making a reconfiguration, a check is made to see if the *local update counter* and the *shared update counter* are equal. In case they are equal, a reconfiguration is made and the *shared update counters* of all the other relevant load-balancers is incremented (using atomic fetch-and-add). Otherwise, if the *shared update counter* is more than the *local update counter*, it indicates a very recent reconfiguration, so no reconfiguration is made at this instance by this load-balancer. However, the *local update counter* is updated to the *shared update counter*. This ensures that each high load event is handled by only one load-balancer.

4.4 Direct Data Placement and RDMA over TCP/IP

In this section, we discuss some of the issues involved in implementing the Direct Data Placement (DDP) and RDMA protocol layers and integrating them with the high performance sockets layers proposed above. We propose implementing this in several steps as follows:

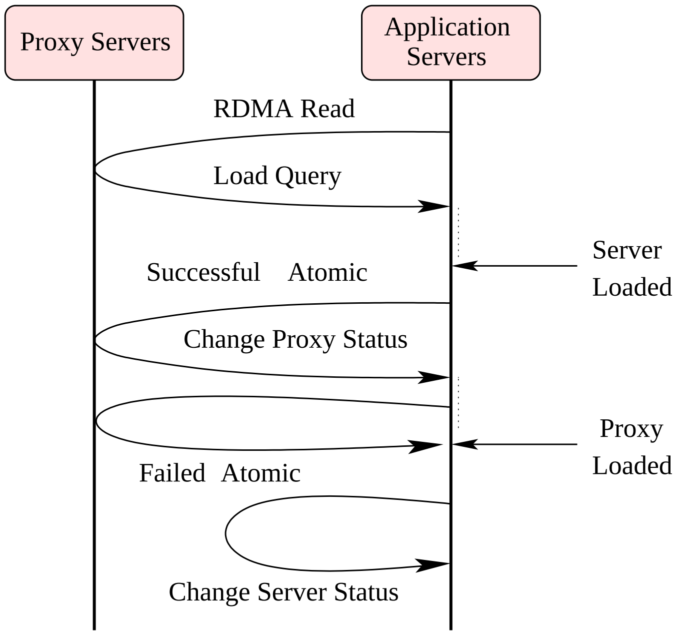


Figure 18: Concurrency controlled protocol for dynamic reconfiguration using one-sided operations

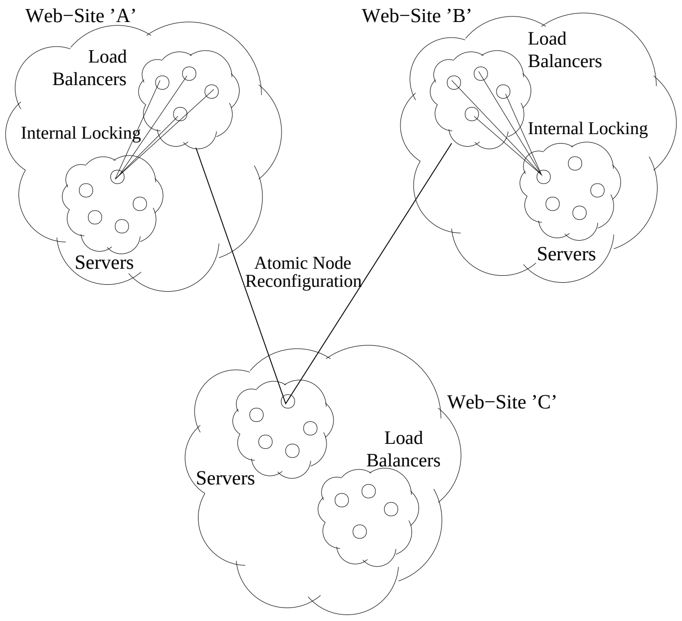


Figure 19: Hierarchical Locking with Dual Counters based concurrency control

- Step 1: Implementing a Software Prototype of DDP over TCP/IP in user space as well as kernel space. This would not only allow us to gain insights into implementation specific details about these, but also serve the purpose of a compatibility middleware for applications running on non-DDP aware networks (e.g., Fast Ethernet) when they want to interact with applications running on a DDP aware network.

- Step 2: Implementing DDP protocol on top of existing TCP Offload Engines. This approach would allow us to gain the performance anticipated for DDP with two-sided communication. However, since the TCP Offload Engines cannot be expected to offer one-sided communication primitives, this will need to be emulated in software. The rest of the DDP stack would essentially be only a thin layer mapping the DDP requests to the features provided by the TCP Offload Engine.

- Step 3: Complete implementation of DDP with the TCP/IP protocol stack as well as the DDP and RDMAP protocols offloaded onto the network adapter. We plan to perform a fast path offload of DDP, i.e., only the data communication path will be offloaded to the NIC; connection management, etc., will still be handled by the host. Also, the software DDP implementations done in Step 1 will be used as a fall back path in case of resource depletion or error conditions which the NIC might be incapable of handling.

More details about each of these are provided in the remaining part of the section.

4.4.1 Software Prototype and Simulation based Study

In this section, we will describe the issues involved in implementing a software implementation of DDP over TCP/IP. We describe two software implementations of DDP: (a) user-level DDP and (b) kernel-level DDP

User-level DDP: This implementation emulates the entire functionality of DDP in user space. This is the most portable implementation of the lot, but is expected to perform the worst. As shown in Figure 20, this approach implements the DDP protocol stack on top of the sockets layer. The following are some of the design issues that need to be dealt with in a User-level DDP implementation.

- *Compatibility with existing sockets applications:* In our proposed framework, DDP is mainly meant to be an extended API for sockets based applications so that existing sockets applications can slowly be ported to use the benefits of DDP. However, there are certain sockets based calls which need to be aware of the existence of DDP. setsockopt(), for example, can be used to set a given socket to DDPMODE. All future communication using this socket will be transferred using the DDP transport layer. Further, read(), write() and several other socket calls need to check if the socket mode is set to DDPMODE before carrying out any communication. This requires modifications to these calls, while making sure that existing sockets applications (which do not use DDP) are not hampered.

  This can be carried out by overloading the standard libc library using our own RDDP library. This library first checks whether a given socket is currently in DDP mode. If it is, it carries out the standard

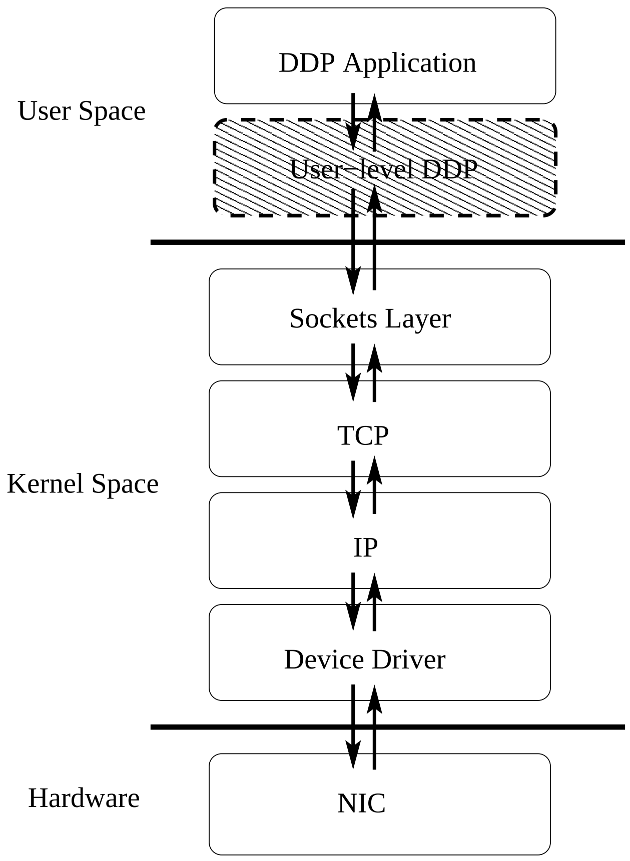


Figure 20: User-level DDP Implementation

DDP procedures to transmit the data. If it isn't, the RDDP library dynamically loads the libc library to pass on the control to libc for the particular call.

- *Gather operations supported by the DDP specifications:* The DDP specifications support gather operations for a list of data segments to be transmitted. Since, the user-level DDP is using TCP as the underlying mode of communication, there are interesting challenges to support this in a zero-copy manner.

    1. *Approach1:* The simplest approach would be to copy data into a standard buffer and send the data out from this buffer. This approach is very simple but would require an extra copy of the data.
    2. *Approach2:* Use readv(), writev() calls in the sockets API. Though in theory sockets supports scatter/gather of data using readv() and writev() calls, the actual implementation of these calls is implementation specific to the kernel. It is possible that the data in these list of buffers be sent out as different messages. While this is perfectly fine with TCP, it creates a lot of fragmentation for DDP, forcing it to have additional buffering to take care of this.
    3. *Approach3:* Use the TCP_CORK mechanism. TCP_CORK allows data to be pushed into the socket buffer. However, until the entire socket buffer is full, data is not sent onto the network. This allows us to copy all the data from the list of the application buffers directly into the TCP socket buffers before sending them out on to the network, thus saving an additional copy. We are currently using this mechanism.

- *Non-Blocking Operations Support:* The DDP specifications supports the implementation of Non-Blocking operations. This means that the application layer can just post a send descriptor; once this is done, it can carry out with its computation and check for completion at a later time.

    1. *Approach 1:* All data can be sent out in the Post_Descriptor() call itself. The Descriptor_Done() check would only be a pseudo call which always returns true. However, this differs from the non-blocking calls semantics and does not give the application the true benefit of non-blocking calls.
    2. *Approach 2:* Multi-threaded implementation using fork(). The difficulty with fork() is that once the two threads are created, they do not share the physical address space. All virtual addresses are mapped to a different physical address space (on write). So, for non-blocking communication, the application thread has to copy the data to a memory location shared by the two threads. Only after the copy is it free to carry out its computation. This approach involves an additional copy, and does not strictly follow the non-blocking call semantics either.
    3. *Approach 3:* Multi-threaded implementation using pthreads(). This approach gives the flexibility of a shared physical address space for the two threads. However, pthreads is notorious for its huge

inter-thread signaling cost. During a Post_Descriptor() or Descriptor_Done() call, there has to be explicit signaling between the threads which can be too expensive. We need to ensure that this signaling is done using IPC calls, semaphores or some other more scalable mechanism.

- *Asynchronous Communication supporting Non-Blocking Operations:* In the previous issue (Non-Blocking operations support), we had stated various options for allowing support for non-blocking operations and chose to use pthreads to allow cloning of virtual address space between the processes. Communication between the threads was intended to be carried out using IPC calls.

  The DDP specification does not allow a shared queue for the multiple sockets in an application (unless they are sockets separated due to forking of processes). Each socket has separate send and receive work queues where descriptors posted for that socket are placed.

  1. *Approach 1:* The first approach to implement this would be to spawn two new threads for each socket set to DDPMODE (one thread for asynchronous send and one for asynchronous receive). While this approach is simple, its not very scalable. As the number of sockets becomes very high, there are a large number of threads created resulting in a number of context switches and huge degradation in the performance.

  2. *Approach 2:* The second approach is to have just two threads which do asynchronous communication for all the sockets. In this approach, the asynchronous threads have to obtain the descriptors from the main thread, process them asynchronously and send back the completed descriptor to the main thread. The first socket which is set to DDPMODE creates the thread.

     While this approach is efficient in theory, the linux implementation of IPC calls makes it difficult and/or in-efficient to use this kind of an approach. The asynchronous thread has to continuously monitor all the DDPMODE sockets as well as the IPC call queue to see if there's a possibility to send or receive data. If there is, it can go ahead and send the data out. The asynchronous thread can monitor the DDPMODE sockets using the select() call. However, this scheme does not work for IPC primitives such as sendmsg() or pipe().

  3. *Approach 3:* This approach is actually just an extension of approach 2. In this approach, we use UNIX socket connections between the main thread and the asynchronous threads. The first socket set to DDPMODE opens a connection with the asynchronous threads and all subsequent sockets use this connection in a persistent manner. This option allows the main thread to post descriptors in a non-blocking manner (since the descriptor is copied to the socket buffer) and at the same time allows the asynchronous thread to use a select() call to make progress on all the DDPMODE sockets as well as the inter-process communication. It is to be noted that though the descriptor involves an additional copy by using this approach, the size of a descriptor is typically very small

(around 60 bytes in the current implementation), so this copy doesn't affect the performance too much.

- *Shared queues between asynchronous threads and application thread:* The approach used for implementing non-blocking communication (described above) allows the posting of the descriptors to be lock free. The UNIX sockets implementation handles concurrency. However, listing the completed descriptors by the asynchronous thread and polling them out by the main thread need to be explicitly synchronized by locking mechanisms in the library.

  The later versions of libc (3.2.2 and later) do not support generic atomic operations to perform locks on user-level data structures. The current version of user-level DDP implements locking using assembly level code. Though this approach is portable across all x86 architectures, its not portable across all the different architectural platforms. To enhance portability these assembly calls are currently protected by define macros. We are looking at more generic schemes such as lock-less queues, etc to make this more portable.

Kernel-level DDP: This implementation emulates the entire functionality of DDP in kernel space. This implementation is lesser portable than the previous user-space implementation (since it requires modifications to the core kernel) but is expected to give a better performance than the previous implementation. As shown in Figure 21, this approach implements the DDP protocol stack on directly top of the TCP/IP stack parallel to the sockets layer. Design wise, this is a true implementation of DDP since it allows applications to access the TCP/IP stack either through the sockets layer or the DDP extended sockets layer. The following are some of the design issues that need to be dealt with in a Kernel-level DDP implementation.

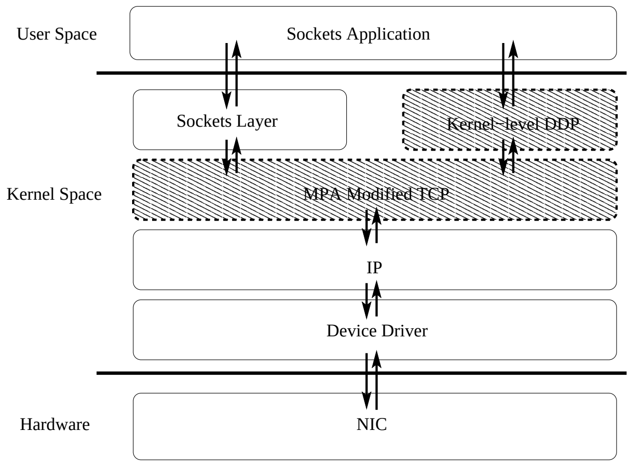


Figure 21: Kernel-level DDP Implementation

- *User buffer mapping for asynchronous I/O:* The posting of the descriptor is an asynchronous event to the actual reception and placement of the data in the appropriate user buffer. The user buffer is allocated and posted in the form of a virtual address space. However, since the kernel is the centralized agent for all the processes running, it is required that the kernel be aware of not only the user process virtual address space, but also some kind of mapping to either the actual process which owns it or to the physical pages they are mapped to.

1. *Approach 1:* The first approach (storing the process id of the user process) does not require the user buffer to be pinned. But this would require the kernel to fetch the page back to physical memory when the data arrives in case of a page swap. Though this saves on the amount of buffer registered, it could lead to a significant performance degradation.

2. *Approach 2:* The second approach (storing the physical pages corresponding to the user buffer) requires that the user buffer be pinned. However, this guarantees that user-buffer is always present in the physical memory space and doesn't require the kernel to fetch the buffer to physical memory when the data arrives. In our implementation, we went ahead with this approach.

- *Kernel access to mapped user buffers:* As mentioned in the previous issue, we pin the user buffers associated in data transfer to optimize the performance provided by the DDP implementation. The next issue we faced was to allow the kernel access the pinned physical address space associated with user buffers. The way this access was provided was by creating a kernel virtual address and mapping it to the same physical pages. Now, since both the user buffer and the kernel buffer are pinned, all data written to the kernel buffer is reflected in the user space through the physical memory pages. The issue in this kind of access is the actual mapping of user-buffer physical pages to the kernel virtual address. This is typically done using the kmap() kernel call. However, in SMP machines such a mapping is very expensive (the exact reason for this is being investigated).

  Optimization to kmap(): Since the cost of the kmap() call is high in SMP machines, as an optimization, we have designed a variant implementation where the mapping of the user-buffer is pushed out of the critical message passing path. In our implementation, we map the user buffer to a kernel virtual address during the memory registration phase. During the actual posting of the descriptor, we only store the offset of the transmission of reception buffer from the registered memory space. It is to be noted that the amount of kernel virtual address space that can be kept mapped to a user buffer is limited. So, in case the mapping of the user buffer fails, the mapping is deferred to the time when the descriptor is actually posted; this would incur the high mapping cost associated with the kmap() call. Most applications don't register significant portions of memory, so this optimization would give a significantly higher performance.

- *Efficiently handling out-of-order segments:* When data arrives out-of-order, the standard TCP/IP stack buffers this data in the out-of-order queue. When the data filling the out-of-order "hole" arrives, this data is transferred to the actual receive queue from where the application can read. On the other hand, DDP allows out-of-order placement of data. This means that the out-of-order segments can be directly placed into the user-buffer without waiting for the intermediate data to be received.

In our implementation, this is handled by placing the data directly and maintaining a queue of received segment sequence numbers. At this point, technically, the received data segments present in the kernel can be freed once they are copied into the user buffer. However, the actual sequence numbers of the received segments are used by TCP for acknowledgments, re-transmissions, etc. So, to allow TCP to proceed with these without any hindrance, we defer the actual freeing of these segments till their sequence numbers cross TCP's unacknowledged window.

4.4.2 DDP implementation on TCP Offload Engines

In this section, we discuss the issues involved in implementing DDP over existing TCP Offload Engines.

Several TCP Offload Engines available in the market provide different APIs in order to access their offloaded protocol stack. These network adapters are similar to the normal Protocol Offload Engines discussed in Section 1, except that the protocol offloaded onto the network adapter is TCP/IP itself. This allows the applications to achieve compatibility with the IP-based WAN. However, sockets based applications cannot directly use the offloaded protocol stack.

Implementing a DDP extended sockets layer on top of TCP Offload Engines needs to be a two phase approach. In the first phase, basic high performance sockets needs to be implemented based on the offloaded protocol stack. In the second phase, this needs to be extended to encompass the DDP functionality.

Another issue that needs to be noted is that TCP Offload Engines are not required to expose any kind of one sided communication primitives. However, the DDP API requires that one-sided communication functions be implemented. In the network adapter does not provide such functionality, this needs to be implemented in software.

4.4.3 Active Caching and Dynamic Reconfigurability using DDP

In this section, we point out some of the application level evaluations that would allow us to understand the performance capabilities of DDP extended high performance sockets as compared to basic high performance sockets. In particular, we look at the coherent caching for dynamic content (Active caching) and resource management using dynamic reconfigurability schemes.

Since DDP is WAN compatible, this implementation would allow us to evaluate the setup in a distributed data-center environment where several clusters are distributed over a WAN and are trying to cache and serve dynamic content in a distributed manner.

Similarly, for dynamic reconfigurability, DDP would allow us to consider Internet proxies performing dynamic reconfiguration of nodes inside the data-center.

4.4.4 NIC Offloaded DDP and RDMAP

In this section, we discuss the issues involved in offloading DDP and RDMAP on to the network adapter in order to have a true and complete implementation of DDP extended high performance sockets. There are several issues, however, that need to be addressed if such an offload is to be performed.

TCP/IP Offload: There are several TCP/IP offloaded network adapters in the market. We can either start with one of these, or use a programmable network interface card to have a fast path offload of TCP/IP on to the network adapter. Several network adapters today come with multiple CPUs on the NIC. If a fast path offload of TCP/IP needs to be performed, we need to analyze the breakup of different functionalities into the different CPUs on the NIC and come up with the optimal breakup for a high performance offloaded TCP/IP stack.

DDP and RDMAP Offload: Performing a DDP offload is more tricky than just performing a fast path TCP/IP offload. Together with the checksum carried out by TCP/IP, DDP requires to perform an additional CRC check. Also, to insert markers into the appropriate location (MPA protocol), data that has been DMAed to the NIC might need to be moved to different locations. This involves additional memory bandwidth requirement on the network adapter.

5 Preliminary Results

In this section, we present some of the preliminary results we obtained in some of the implementations and analyses proposed in Section 4.

5.1 High Performance Sockets over Gigabit Ethernet

In this section, we present some of the preliminary results we obtained with our implementation of high performance sockets over Gigabit Ethernet.

The experimental test-bed consisted of four Pentium III 700MHz Quads, each with a Cache Size of 1MB and 1GB main memory. The interconnect was a Gigabit Ethernet network with Alteon NICs on each machine connected using a Packet Engine switch. The linux kernel version used was 2.4.18.

5.1.1 Implementation Alternatives

This section gives the performance evaluation of the basic substrate without any performance enhancement and shows the advantage obtained incrementally with each performance enhancement technique.

In Figure 22 the basic performance given by the substrate for data streaming sockets is labeled as DS and that for datagram sockets is labeled as DG. DS_DA refers to the performance obtained by incorporating Delayed Acknowledgments as mentioned in Section 5.3. DS_DA_UQ refers to the performance obtained with both the Delayed Acknowledgments and the Unexpected Queue option turned on (Section 5.4). For this experiment, for the Data Streaming case, we have chosen a credit size of 32 with each temporary buffer of size 64KB. With all the options turned on, the substrate performs very close to raw EMP. The Datagram option performs the closest to EMP with a latency of 28.5 μs (an overhead of as low as 1 μs over EMP) for 4-bytes messages. The Data Streaming option with all enhancements turned on, is able to provide a latency of 37 μs for 4-byte messages.

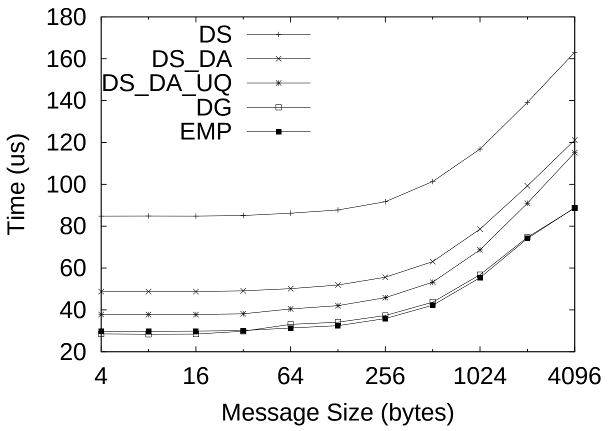


Figure 22: High Performance Sockets over EMP: Comparison of different performance optimizations

Figure 23 shows the drop in latency with delayed acknowledgment messages. The reason for this is the decrease in the amount of tag matching that needs to be done at the NIC with the reduced number of acknowledgment descriptors. For a credit size of 1, the percentage of acknowledgment descriptors would be 50%, which leads to an additional tag matching for every data descriptor. However, for a credit size of 32, the percentage of acknowledgment descriptors would be 6.25%, thus reducing the tag matching time.

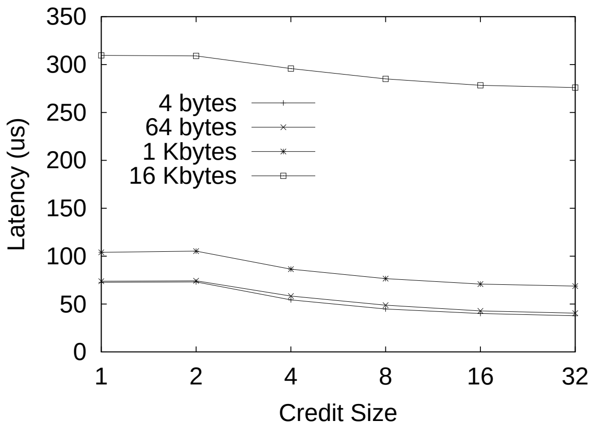
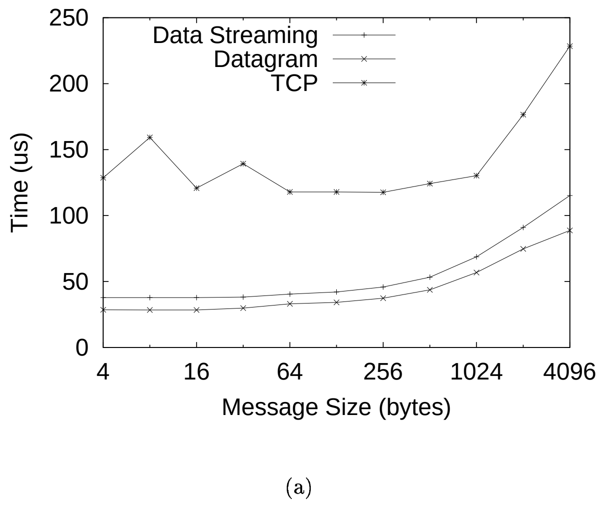


Figure 23: High Performance Sockets over EMP: Impact of Delayed Acknowledgments

The bandwidth results have been found to stay in the same range with each performance evaluation technique.

5.1.2 Latency and Bandwidth

Figure 24 shows the latency and the bandwidth achieved by the substrate compared to TCP. The Data Streaming label corresponds to DS_DA_UQ (Data Streaming sockets with all performance enhancements turned on).



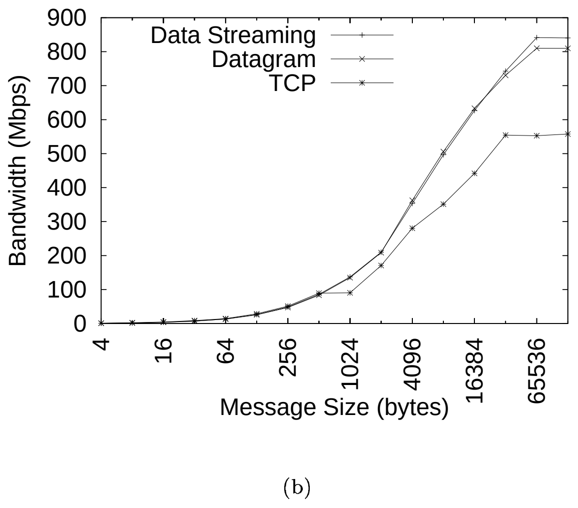


Figure 24: High Performance Sockets over EMP Micro-Benchmarks: (a) Latency and (b) Bandwidth

Again, for the data streaming case, a credit size of 32 has been chosen with each temporary buffer of size 64 Kbytes. In default, TCP allocates 64 Kbytes of kernel space for the NIC to use for communication activity. With this amount of kernel space, TCP has been found to give a bandwidth of about 340 Mbps. However, since the modern systems allow much higher memory registration, we changed the kernel space allocated by TCP for the NIC to use. With increasing buffer size in the kernel, TCP is able to achieve a bandwidth of about 550 Mbps (after which increasing the kernel space allocated does not make any difference). Further, this change in the amount of kernel buffer allocated does not affect the latency results obtained by TCP to a great extent.

The substrate is found to give a latency as low as 28.5 μs for Datagram sockets and 37 μs for Data Streaming sockets achieving a performance improvement of 4.2 and 3.4 respectively, compared to TCP. The peak bandwidth achieved was above 840Mbps with the Data Streaming option.

5.1.3 FTP Application

We have measured the performance of the standard File Transfer Protocol (ftp) given by TCP on Gigabit Ethernet and our substrate. To remove the effects of disk access and caching, we have used RAM disks for this experiment.

With our substrate, the FTP application takes 6.84 secs for Data Streaming and Datagram sockets, compared to the 11.8 secs taken by TCP for transferring a 512MB file. For small files, FTP takes 13.6 μs for Data Streaming and Datagram sockets, compared to the 25.6 μs taken by TCP. The application is not able to achieve the peak bandwidth, due to the File System overhead.

There is a minor variation in the bandwidth achieved by the data streaming and the datagram options in the standard bandwidth test. The overlapping of the performance achieved by both the options in ftp application, is also attributed to the file system overhead.

5.1.4 Web Server Application

We have measured the performance obtained by the Web Server application for a 4 node cluster (with one server and three clients). The experiment was designed in the following manner – the server keeps accepting requests from the clients. The clients connect to the server and send in a request message (which can typically be considered a file name) of size 16 bytes. The server accepts the connection and sends back a message of size S bytes to the client. We have shown results for S varying from 4 bytes to 8 Kbytes. Once the message is sent, the connection is closed (as per HTTP/1.0 specifications). However, this was slightly modified in the HTTP/1.1 specifications, which we also discuss in this section.

A number of things have to be noted about this application. First, the latency and the connection time results obtained by the substrate in the micro-benchmarks play a dominant role in this application. For connection management, we use a data message exchange scheme as mentioned earlier. This gives an inherent benefit to the Sockets-over-EMP scheme since the time for the actual request is hidden, as the connection message descriptors are pre-posted.

Figure 25 gives the results obtained by the Web Server application following the HTTP/1.0 specifications.

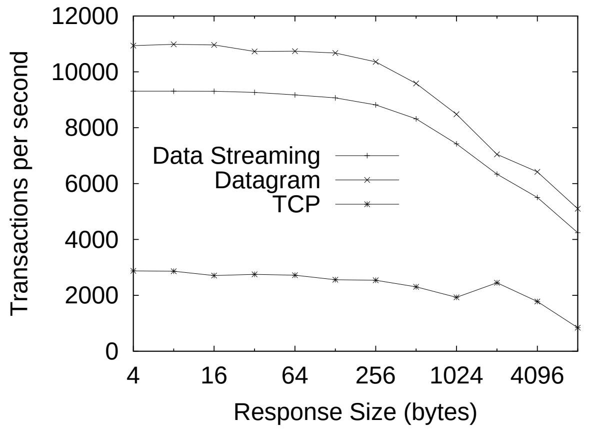


Figure 25: High Performance Sockets over EMP: Performance of a web-server (HTTP 1.0)

In the substrate, once the "connection request" message is sent by the substrate, the application can start sending the data messages. This reduces the connection time of the substrate to the time required by a message exchange. However, in TCP, the connection time requires intervention by the kernel and is typically about 200 to 250 μs. To cover this disadvantage, TCP has the following enhancements: the HTTP 1.1 specifications allow a node to make up to 8 requests on one connection. Even with this specification, our substrate was found to perform better than the base TCP application. In the worst case, if the web server allows infinite requests on a single connection, the web server application boils down to a simple latency test.

5.2 High Performance Sockets over Virtual Interface Architecture

In this section, we present the peak performance achieved by high performance sockets over VIA in the form of micro-benchmark results. Application level evaluation for this layer is presented in Section 5.4 as a part of the impact of high performance sockets on data intensive applications.

The experiments were carried out on a PC cluster which consists of 16 Dell Precision 420 nodes connected by GigaNet cLAN and Fast Ethernet. We use cLAN 1000 Host Adapters and cLAN5300 Cluster switches. Each node has two 1GHz Pentium III processors, built around the Intel 840 chipset, which has four 32-bit 33-MHz PCI slots. These nodes are equipped with 512MB of SDRAM and 256K L2-level cache. The Linux kernel version is 2.2.17.

5.2.1 Micro-Benchmarks

Figure 26(a) shows the latency achieved by our substrate compared to that achieved by the traditional implementation of sockets on top of TCP and a direct VIA implementation (base VIA). Our sockets layer gives a latency of as low as 9.5μs, which is very close to that given by VIA. Also, it is nearly a factor of five improvement over the latency given by the traditional sockets layer over TCP/IP.

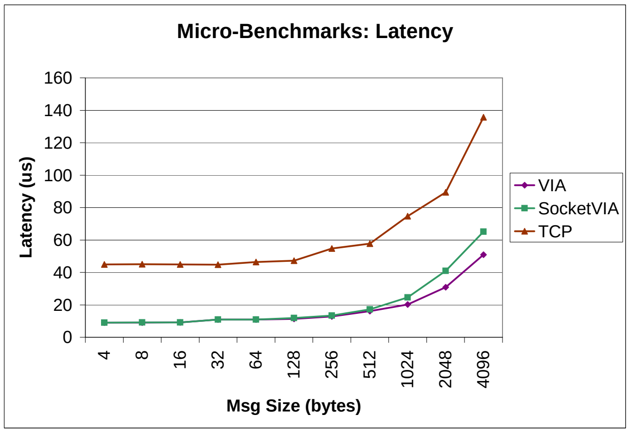
Figure 26(b) shows the bandwidth achieved by our substrate compared to that of the traditional sockets implementation and base cLAN VIA implementation. SocketVIA achieves a peak bandwidth of 763Mbps compared to 795Mbps given by VIA and 510Mbps given by the traditional TCP implementation; an improvement of nearly 50%.

5.3 High Performance Sockets over InfiniBand

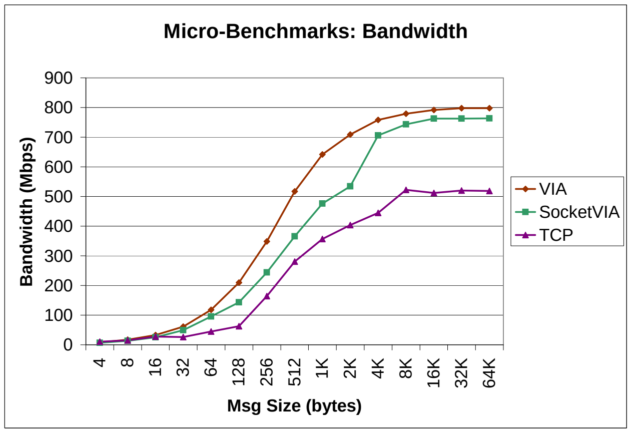
In this section, we present some of the preliminary results we obtained with our implementation of high performance sockets over InfiniBand (SDP). We have carried out three kinds of tests over this sockets layer: (i) Micro-benchmark level evaluation with a detailed micro-benchmark suite, (ii) Evaluation in the data-center environment and (iii) Evaluation of the Parallel Virtual File System (PVFS).

5.3.1 SDP Micro-Benchmark Results

In this section, we compare the micro-benchmark level performance achievable by SDP and the native sockets implementation over InfiniBand (IPoIB). For all our experiments we used 2 clusters:



(a)



(b)

Figure 26: High Performance Sockets over VIA Micro-Benchmarks: (a) Latency and (b) Bandwidth

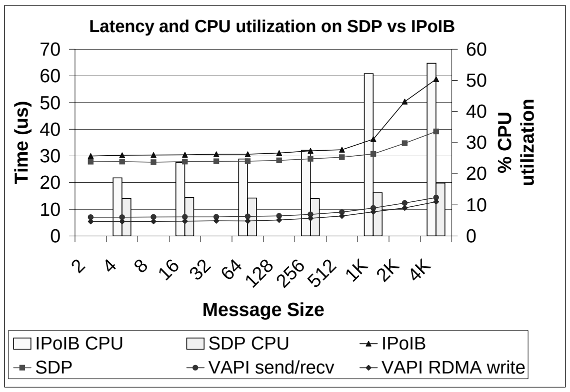
Cluster 1: An 8 node cluster built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 kB L2 cache and a 400 MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 Dual-Port 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The SDK version is thca-x86-0.2.0-build-001. The adapter firmware version is fw-23108-rel-1_17_0000-rc12-build-001. We used the Linux RedHat 7.2 operating system.

Cluster 2: A 16 Dell Precision 420 node cluster connected by Fast Ethernet. Each node has two 1GHz Pentium III processors, built around the Intel 840 chipset, which has four 32-bit 33-MHz PCI slots. These nodes are equipped with 512MB of SDRAM and 256K L2-level cache.

We used Cluster 1 for all experiments in this section.

Latency and Bandwidth: Figure 27a shows the one-way latency achieved by IPoIB, SDP and Send-Receive and RDMA Write communication models of native VAPI for various message sizes. SDP achieves a latency of around 28μs for 2 byte messages compared to a 30μs achieved by IPoIB and 7μs and 5.5μs achieved by the Send-Receive and RDMA communication models of VAPI. Further, with increasing message sizes, the difference between the latency achieved by SDP and that achieved by IPoIB tends to increase.

Figure 27b shows the uni-directional bandwidth achieved by IPoIB, SDP, VAPI Send-Receive and VAPI RDMA communication models. SDP achieves a throughput of up to 471Mbytes/s compared to a 169Mbytes/s achieved by IPoIB and 825Mbytes/s and 820Mbytes/s achieved by the Send-Receive and RDMA communication models of VAPI. We see that SDP is able to transfer data at a much higher rate as compared to IPoIB using a significantly lower portion of the host CPU. This improvement in the throughput and CPU is mainly attributed to the NIC offload of the transportation and network layers in SDP unlike that of IPoIB.



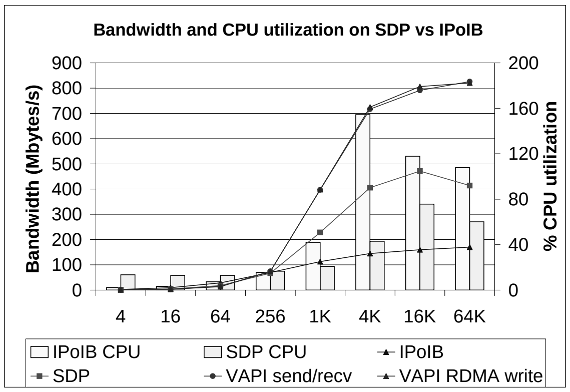


Figure 27: High Performance Sockets over InfiniBand: (a) Latency and (b) Bandwidth

Multi-Stream Bandwidth: In the Multi-Stream bandwidth test, we use two machines and N threads on each machine. Each thread on one machine has a connection to exactly one thread on the other machine and on each connection, the basic bandwidth test is performed. The aggregate bandwidth achieved by all the threads together within a period of time is calculated as the multi-stream bandwidth. Performance results with different numbers of streams are shown in Figure 28. We can see that SDP achieves a peak bandwidth of about 500Mbytes/s as compared to a 200Mbytes/s achieved by IPoIB. The CPU Utilization for a 16Kbyte message size is also presented.

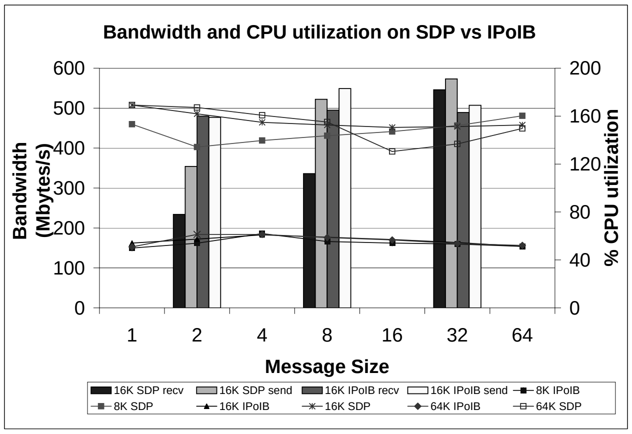


Figure 28: High Performance Sockets over InfiniBand: Multi-Stream Bandwidth

Hot-Spot Test: In the Hot-Spot test, multiple clients communicate with the same server. The communication pattern between any client and the server is the same pattern as in the basic latency test, i.e., the server needs to receive messages from all the clients and send messages to all clients as well, creating a hot-spot on the server. Figure 29 shows the one-way latency of IPoIB and SDP when communicating with a hot-spot server, for different numbers of clients. The server CPU utilization for a 16Kbyte message size is also presented. We can see that as SDP scales well with the number of clients; its latency increasing by only 138μs compared to 456μs increase with IPoIB for a message size of 16Kbytes. Further, we find that as the number of nodes increases we get an improvement of more than a factor of 2, in terms of CPU utilization for SDP over IPoIB.

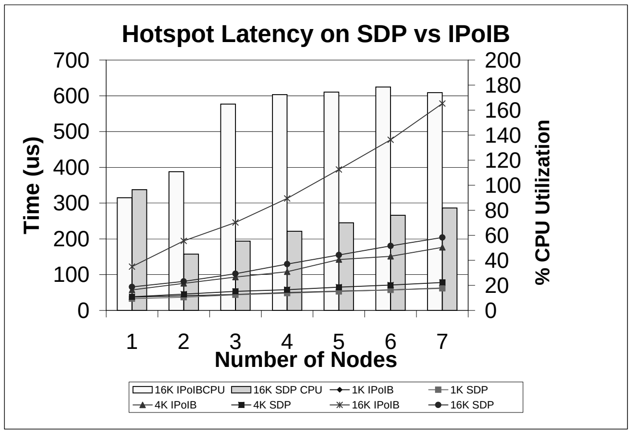


Figure 29: High Performance Sockets over InfiniBand: Hot-Spot Test

Fan-in and Fan-out: In the Fan-in test, multiple clients from different nodes stream data to the same server. Similarly, in the Fan-out test, the same server streams data out to multiple clients. Figures 30a and 30b show the aggregate bandwidth observed by the server for different number of clients for the Fan-in and Fan-out tests respectively. We can see that for the Fan-in test, SDP reaches a peak aggregate throughput of 687Mbytes/s compared to a 237Mbytes/s of IPoIB. Similarly, for the Fan-out test, SDP reaches a peak aggregate throughput of 477Mbytes/s compared to a 175Mbytes/s of IPoIB. The server CPU utilization for a 16Kbyte message size is also presented. Both figures show similar trends in CPU utilization for SDP and IPoIB as the previous tests, i.e., SDP performs about 60-70% better than IPoIB in CPU requirements.

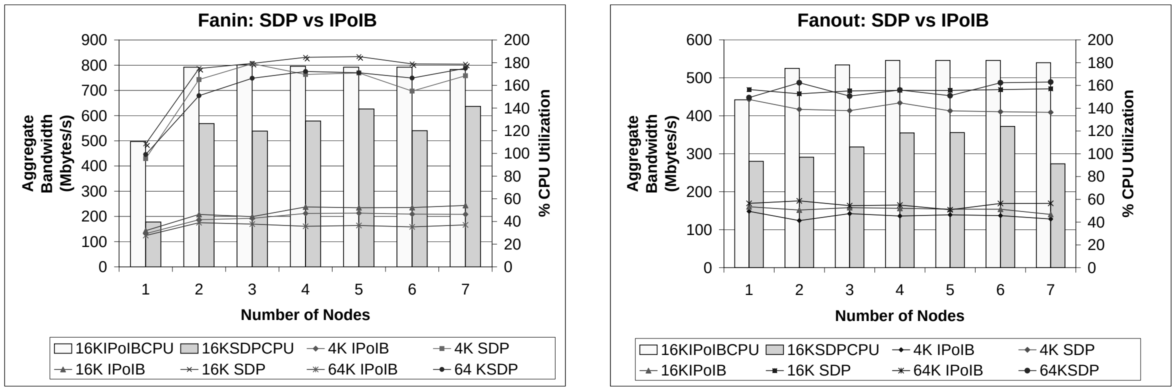


Figure 30: High Performance Sockets over InfiniBand: (a) Fan-in test and (b) Fan-out test

5.3.2 Data-Center Performance Evaluation

In this section, we analyze the performance of a 3-tier data-center environment over SDP while comparing it with the performance of IPoIB. For all experiments in this section, we used nodes in Cluster 1 for the data-center tiers. For the client nodes, we used the nodes in Cluster 2 for most experiments.

Evaluation Methodology: As mentioned earlier, we used a 3-tier data-center model. Tier 1 consists of the front-end proxies. For this we used the proxy module of *apache-1.3.12*. Tier 2 consists of the web server and PHP application server modules of Apache, in order to service static and dynamic requests respectively. Tier 3 consists of the Database servers running *MySQL* to serve dynamic database queries. All the three tiers in the data-center reside on an InfiniBand network; the clients are connected to the data-center using Fast Ethernet. We evaluate the response time of the data-center using *Openload*, an open source client workload generator. We use a 20000 request subset of the world-cup trace [57] for our experiments. To generate requests amounting to different average file sizes, we scale the file sizes in the given trace linearly, while keeping the access pattern intact.

In our experiments, we evaluate two scenarios: requests from the client consisting of 100% static content (involving only the proxy and the web server) and requests from the client consisting of 100% dynamic content (involving all the three tiers in the data-center). "Openload" allows firing a mix of static and dynamic requests. However, the main aim of this paper is the analysis of the performance achievable by IPoIB and SDP. Hence, we only focused on these two scenarios (100% static and 100% dynamic content) to avoid dilution of this analysis with other aspects of the data-center environment such as workload characteristics, etc.

For evaluating the scenario with 100% static requests, we used a test-bed with one proxy at the first tier and one web-server at the second tier. The client would fire requests one at a time, so as to evaluate the ideal case response time for the request. For evaluating the scenario with 100% dynamic page requests, we set up the data center with the following configuration: Tier 1 consists of 3 Proxies, Tier 2 contains 2 servers which act as both web servers as well as application servers (running PHP) and Tier 3 with 3 MySQL Database Servers (1 Master and 2 Slave Servers). We used the TPC-W transactional web benchmark [16] for generating our dynamic request access pattern (further details about the database used can be obtained in [12]).

Experimental Results: We used a 20,000 request subset of the world-cup trace to come up with our base trace file. As discussed earlier, to generate multiple traces with different average file sizes, we scale each file size with the ratio of the requested average file size and the weighted average (weighted by the frequency of requests made to the given file) of the base trace file.

Figure 31a shows the response times seen by the client for various average file sizes requested over IPoIB and SDP. As seen in the figure, the benefit obtained by SDP over IPoIB is quite minimal. In order to analyze the reason for this, we found the break-up of this response time in the proxy and web servers. Figure 31b shows the break-up of the response time for average file size requests of 64K and 128K. The "Web-Server Time" shown in the graph is the time duration for the back-end web-server to respond to the file request from the proxy. The "Proxy-Time" is the difference between the times spent by the proxy (from the moment it gets the request to the moment it sends back the response) and the time spent by the web-server. This value denotes the actual overhead of the proxy tier in the entire response time seen by the client. Similarly, the "Client-Time" is the difference between the times seen by the client and by the proxy.

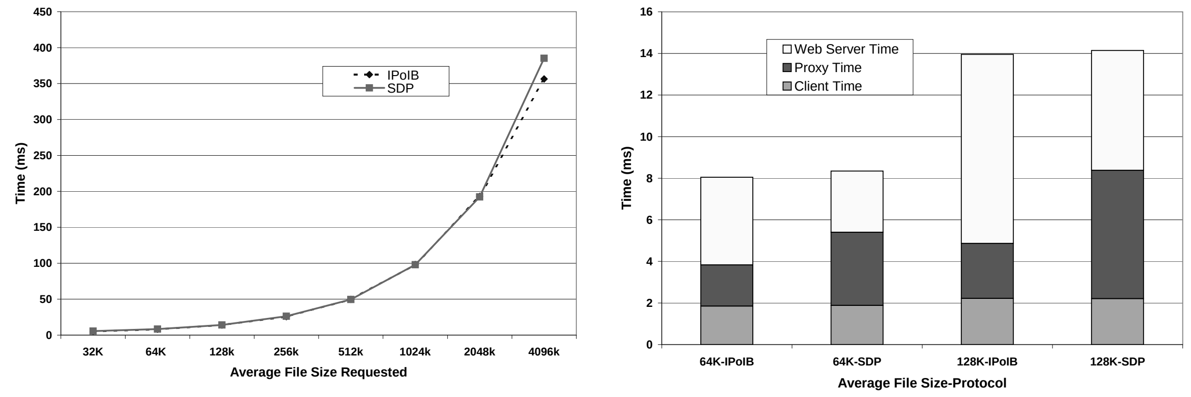


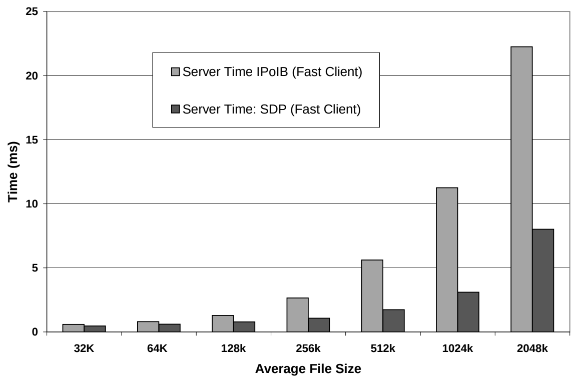
Figure 31: High Performance Sockets over InfiniBand: (a) Web-Server Client Response Time and (b) Response time breakup

From the break-up graph (Figure 31b), we can easily observe that the web server over SDP is consistently better than IPoIB, implying that the web server over SDP can deliver better throughput. Further, this also implies that SDP can handle a given server load with lesser number of back-end web-servers as compared to an IPoIB based implementation due to the reduced "per-request-time" spent at the server. In spite of this improvement in the performance in the web-server time, there's no apparent improvement in the overall response time.

A possible reason for this lack of improvement is the slow interconnect used by the clients to contact the proxy server. Since the client connects to the data-center over fast ethernet, it is possible that the client is unable to accept the response at the rate at which the server is able to send the data. To validate this hypothesis, we conducted experiments using our data-center test-bed with faster clients. Such clients may themselves be on high speed interconnects such as InfiniBand or may become available due to Internet proxies, ISPs etc.

Figure 32a shows the client response times that is achievable using SDP and IPoIB in this new scenario which we emulated by having the clients request files over IPoIB (using InfiniBand; we used nodes from cluster 1 to act as clients in this case). This figure clearly shows a better performance for SDP, as compared to IPoIB for large file transfers above 128K. However, for small file sizes, there's no significant improvement. In fact, IPoIB outperforms SDP in this case. To understand the lack of performance benefits for small file sizes, we took a similar split up of the response time perceived by the client.

Figure 32b shows the split-up of the response time seen by the faster clients. We observe the same trend as seen with clients over Fast Ethernet. The "web-server time" reduces even in this scenario. However, it's quickly apparent from the figure that the time taken at the proxy is higher for SDP as compared to IPoIB. For a clearer understanding of this observation, we further evaluated the response time within the data-center by breaking down the time taken by the proxy in servicing the request.



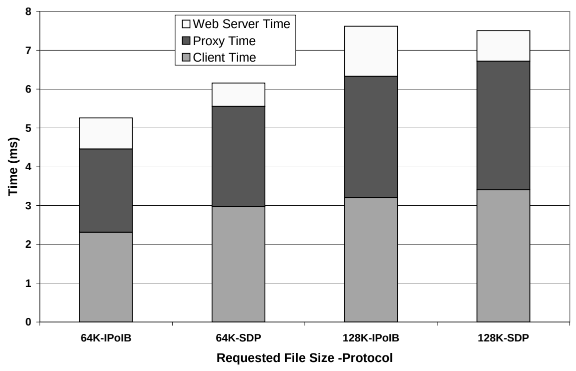
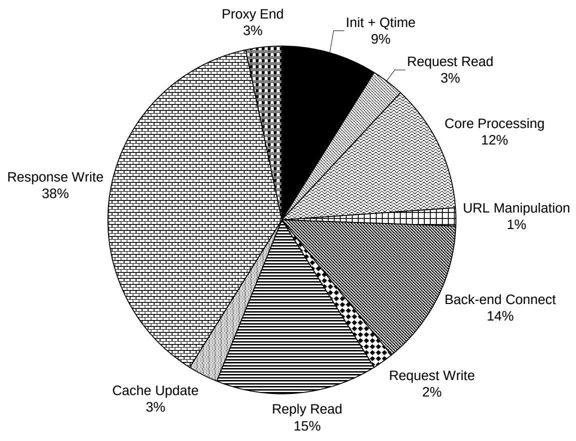


Figure 32: High Performance Sockets over InfiniBand: (a) Web-Server Client Response Time (Fast Clients) and (b) Response time breakup (Fast Clients)

Figures 33a and 33b show a comprehensive breakup of the time spent at the proxy over IPoIB and SDP respectively. A comparison of this split-up for SDP with IPoIB shows a significant difference in the time for the the proxy to connect to the back-end server. This high connection time of the current SDP implementation, about 500μs higher than IPoIB, makes the data-transfer related benefits of SDP imperceivable for low file size transfers.



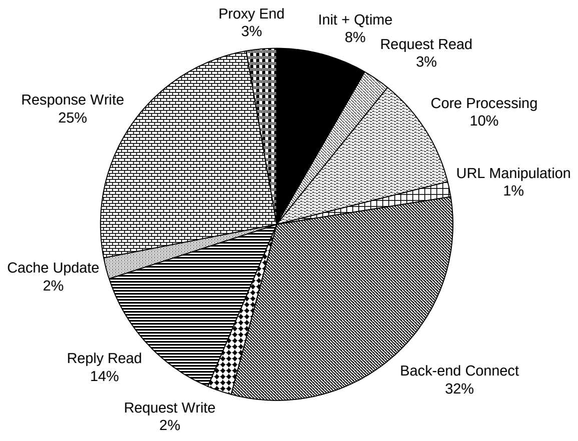


Figure 33: Web Server Response Splitup: (a) IPoIB and (b) SDP

The current implementation of SDP has inherent lower level function calls during the process of connection establishment, which form a significant portion of the connection latency. In order to hide this connection time overhead, researchers are proposing a number of techniques including persistent connections from the proxy to the back-end, allowing free connected Queue Pair (QP) pools, etc. Further, since this issue of connection setup time is completely implementation specific, we tried to estimate the (projected) performance SDP can provide if the connection time bottleneck was resolved.

Figure 34 shows the projected response times of the fast client, without the connection time overhead. Assuming a future implementation of SDP with lower connection time, we see that SDP is able to give significant response time benefits as compared to IPoIB even for small file size transfers. A similar analysis for dynamic requests can be found in [12].

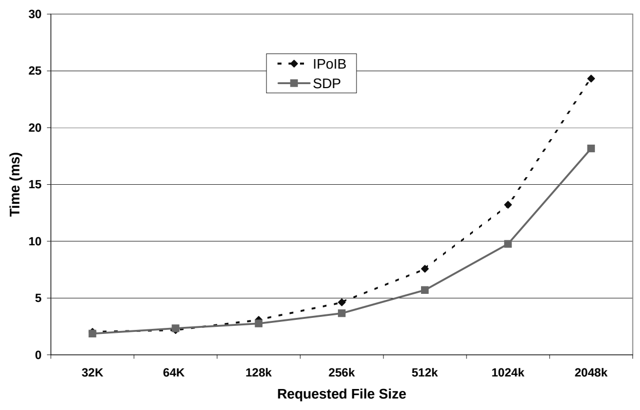


Figure 34: Response Time estimates without connection overhead

5.3.3 PVFS Performance Evaluation

In this section, we compare the performance of the Parallel Virtual File System (PVFS) over IPoIB and SDP with the original PVFS implementation [27]. We also compare the performance of PVFS on the above two protocols with the performance of our previous implementation of PVFS over InfiniBand [59]. All experiments in this section have been performed on Cluster 1.

Evaluation Methodology: There is a large difference between the bandwidth realized by the InfiniBand network and that which can be obtained on a disk-based file system in most cluster systems. However, applications can still benefit from fast networks for many reasons in spite of this disparity. Data frequently resides in server memory due to file caching and read-ahead when a request arrives. Also, in large disk array systems, the aggregate performance of many disks can approach network speeds. Caches on disk arrays and on individual disks also serve to speed up transfers. Therefore, we designed two types of experiments. The first type of experiments are based on a memory-resident file system, *ramfs*. These tests are designed to stress the network data transfer independent of any disk activity. Results of these tests are representative of workloads with sequential I/O on large disk arrays or random-access loads on servers which are capable of delivering data at network speeds. The second type of experiments are based on a regular disk file system, *ext3fs*. Results of these tests are representative of disk-bounded workloads. In these tests, we focus on how the difference in CPU utilization for these protocols can affect the PVFS performance.

We used the test program, *pvfs-test* (included in the PVFS release package), to measure the concurrent read and write performance. We followed the same test method as described in [27], i.e., each compute node simultaneously reads or writes a single contiguous region of size $2N$ Mbytes, where N is the number of I/O nodes. Each compute node accesses 2 Mbytes data from each I/O node.

PVFS Concurrent Read and Write on ramfs: Figure 35a shows the read performance with the original implementation of PVFS over IPoIB and SDP and an implementation of PVFS over VAPI [59], previously done by our group. The performance of PVFS over SDP depicts the peak performance one can achieve without making any changes to the PVFS implementation. On the other hand, PVFS over VAPI depicts the peak performance achievable by PVFS over InfiniBand. We name these three cases using the legends *IPoIB*, *SDP*, and *VAPI*, respectively. When there are sufficient compute nodes to carry the load, the bandwidth increases at a rate of approximately 140 Mbytes/s, 310 Mbytes/s and 380 Mbytes/s with each additional I/O node for IPoIB, SDP and VAPI respectively. Note that in our 8-node InfiniBand cluster system (Cluster 1), we cannot place the PVFS manager process and the I/O server process on the same physical node since the current implementation of SDP does not support socket-based communication between processes on the same physical node. So, we have one compute node lesser in all experiments with SDP.

Figure 35b shows the write performance of PVFS over IPoIB, SDP and VAPI. Again, when there are sufficient compute nodes to carry the load, the bandwidth increases at a rate of approximately 130 Mbytes/s, 210 Mbytes/s and 310 Mbytes/s with each additional I/O node for IPoIB, SDP and VAPI respectively.

Overall, compared to PVFS on IPoIB, PVFS on SDP has a factor of 2.4 improvement for concurrent reads and a factor of 1.5 improvement for concurrent writes. The cost of writes on *ramfs* is higher than that of reads, resulting in a lesser improvement for SDP as compared to IPoIB. Compared to PVFS over VAPI, PVFS over SDP has about 35% degradation. This degradation is mainly attributed to the copies on the sender and the receiver sides in the current implementation of SDP. With a future zero-copy implementation of SDP, this gap is expected to be further reduced.

PVFS Concurrent Write on ext3fs: We also performed the above mentioned test on a disk-based file system, *ext3fs* on a Seagate ST340016A, ATA 100 40 GB disk. The write bandwidth for this disk is 25 Mbytes/s. In this test, the number of I/O nodes are fixed at three, and the number of compute nodes four. We chose PVFS *write with sync*. Figure 36 shows the performance of PVFS write with sync with IPoIB, SDP and VAPI. It can be seen that, although each I/O server is disk-bound, a significant performance improvement of 9% is achieved by PVFS over SDP as compared to PVFS over IPoIB. This is because the lower overhead of SDP as shown in Figure 27 leaves more CPU cycles free for I/O servers to process concurrent requests. Due to the same reason, SDP achieves about 5% lesser performance as compared to the native VAPI implementation.

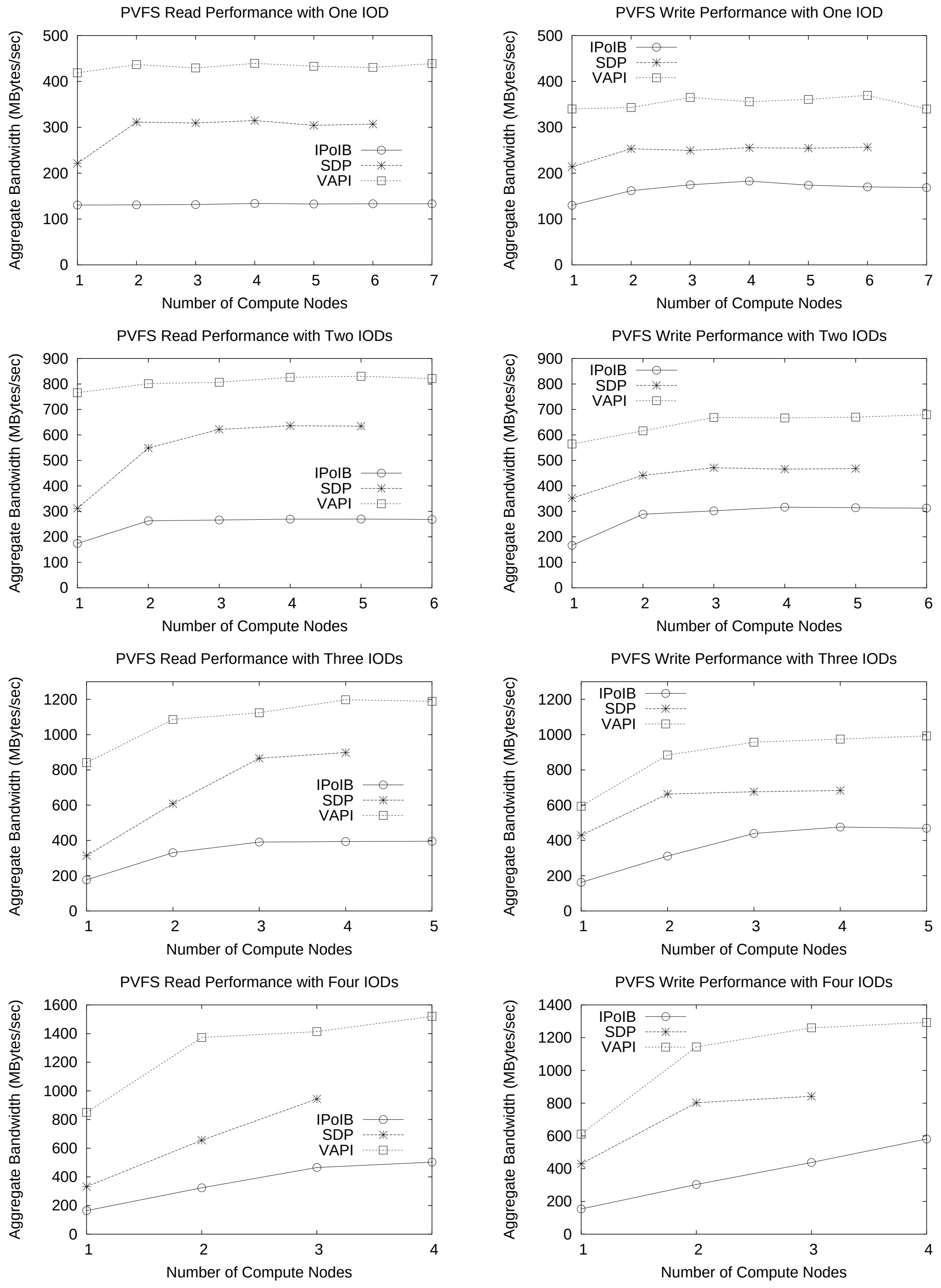


Figure 35: PVFS Read Performance Comparison

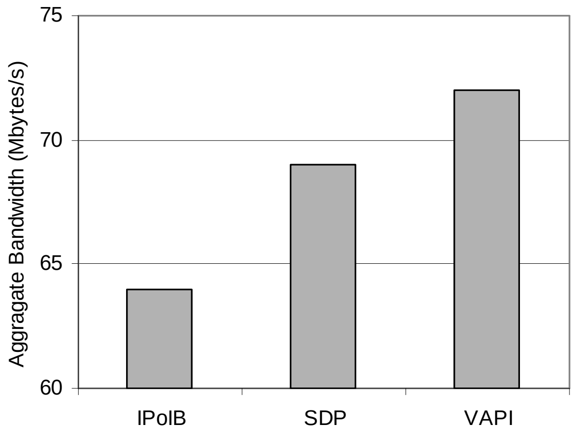


Figure 36: Performance of PVFS Write with Sync on ext3fs

5.4 Impact of High Performance Sockets on Data Intensive Applications

In this section, we present some results showing the impact of high performance sockets implementations (in particular sockets over VIA) on the performance and behavior of various data intensive applications.

5.4.1 Experimental Setup

In these experiments, we used two kinds of applications. The first application emulates a visualization server. This application uses a 4-stage pipeline with a visualization filter at the last stage. Also, we executed three copies of each filter in the pipeline to improve the end bandwidth (Figure 37). The user visualizes an image at the visualization node, on which the visualization filter is placed. The required data is fetched from a data repository and passed onto other filters, each of which is placed on a different node in the system, in the pipeline.

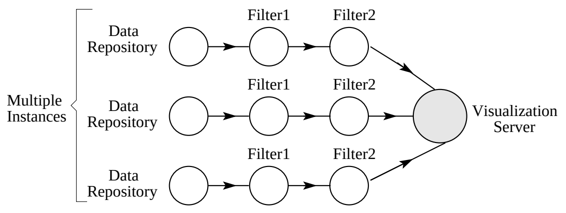


Figure 37: Guarantee Based Performance Evaluation: Experimental Setup

Each image viewed by the user requires 16MB of data to be retrieved and processed. This data is stored in the form of chunks with pre-defined size, referred to here as the distribution block size. For a typical distribution block size, a complete image is made up of several blocks (Figure 14). When the user asks for an update to an image (partial or complete), the corresponding chunks have to be fetched. Each chunk is retrieved as a whole, potentially resulting in some additional unnecessary data to be transferred over the network.

Two kinds of queries were emulated. The first query is a complete update or a request for a new image. This requires all the blocks corresponding to the query to be fetched. This kind of update is bandwidth sensitive and having a large block size would be helpful. Therefore, as discussed in the earlier sections, for allowing a certain update rate for the complete update queries, a certain block size (or larger) has to be used.

The second query is a partial update. This type of query is executed when the user moves the visualization window by a small amount, or tries to zoom into the currently viewed image. A partial update query requires only the excess blocks to be fetched, which is typically a small number compared to the number of blocks forming the complete image. This kind of update is latency sensitive. Also, the chunks are retrieved as a whole. Thus, having small blocks would be helpful.

In summary, if the block size is too large, the partial update will likely take long time, since the entire block is fetched even if a small portion of one block is required. However, if the block size is too small, the complete update will likely take long time, since many small blocks will need to be retrieved. Thus, for an application which allows both kinds of queries, there would be a performance tradeoff between the two types of queries. In the following experiments, we show the improved scalability of the application with socketVIA compared to that of TCP with performance guarantees for each kind of update.

The second application we look at is a software load-balancing mechanism such as the one used by DataCutter. When data is processed by a number of nodes, perfect pipelining is achieved when the time taken by the load-balancer to send one block of the message to the computing node is equal to the time taken by the computing node to process it. In this application, typically the block size is chosen so that perfect pipelining is achieved in computation and communication. However, the assumption is that the computation power of the nodes does not change during the course of the application run. In a heterogeneous, dynamic environment, this assumption does not hold. In our experiments, in a homogeneous setting, perfect pipelining is achieved at 16KB and 2KB for TCP/IP and VIA, respectively. This means that the block size required in TCP/IP is significantly larger than that in VIA. However, on heterogeneous networks, when a block size is too large, a mistake by a load balancer (sending the data block to a slow node) may become too costly (Figure 38). Performance impact with such heterogeneity is presented later in this section.

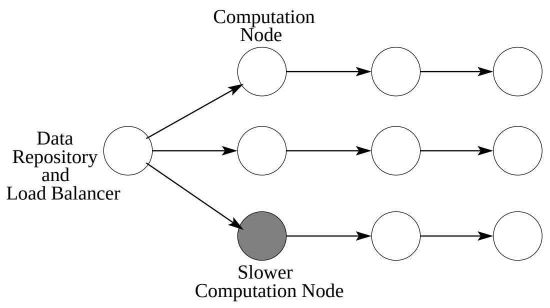


Figure 38: Guarantee Based Performance Evaluation: Experimental Setup

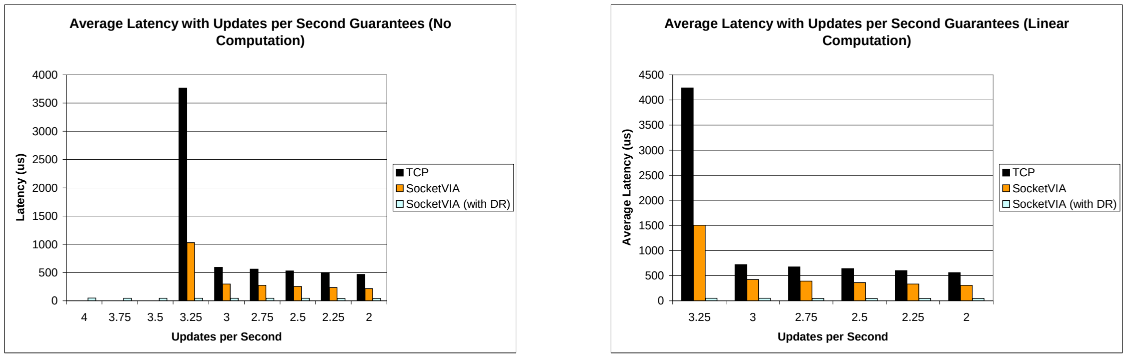


Figure 39: Effect of High Performance Sockets on Average Latency with guarantees on Updates per Second for (a) No Computation Cost and (b) Linear Computation Cost

5.4.2 Guarantee based Performance Evaluation

Effect on Average Latency with guarantees on Updates per Second: In the first set of experiments, the user wants to achieve a certain frame rate (i.e., the number of new images generated or full updates done per second). With this constraint, we look at the average latency observed when a partial update query is submitted. Figures 39(a) and 39(b) show the performance achieved by the application. For a given frame rate for new images, TCP requires a certain message size to attain the required bandwidth. With data chunking done to suit this requirement, the latency for a partial update using TCP would be the latency for this message chunk (depicted as legend 'TCP'). With the same chunk size, SocketVIA inherently achieves a higher performance (legend 'SocketVIA'). However, SocketVIA requires a much smaller message size to attain the bandwidth for full updates. Thus, by repartitioning the data by taking SocketVIA's latency and bandwidth into consideration, the latency can be further reduced (legend 'SocketVIA (with DR)'). Figure 39(a) shows the performance with no computation. This experiment emphasizes the actual benefit obtained by using SocketVIA, without being affected by the presence of computation costs at each node. We observe, here, that TCP cannot meet an update constraint greater than 3.25 full updates per second. However, SocketVIA (with DR) can still achieve this frame rate without much degradation in the performance. The results obtained in this experiment show an improvement of more than 3.5 times without any repartitioning and more than 10 times with repartitioning of data. In addition to socketVIA's inherently improving the performance of the application, reorganizing some components of the application (the block size in this case) allows the application to gain significant benefits not only in performance, but also in scalability with performance guarantees.

Figure 39(b) depicts the performance with a computation cost that is linear with message size in the experiments. We timed the computation required in the visualization part of a digitized microscopy application, called Virtual Microscope [28], on DataCutter and found it to be 18ns per byte of the message. Applications such as these involving browsing of digitized microscopy slides have such low computation costs per pixel. These are the applications that will benefit most from low latency and high bandwidth substrates. So we have focused on such applications in this paper.

In this experiment, even SocketVIA (with DR) is not able to achieve an update rate greater than 3.25, unlike the previous experiment. The reason for this is that the bandwidth given by SocketVIA is bounded by the computation costs at each node. For this experiment, we observe an improvement of more than 4 and 12 times without and with repartitioning of data, respectively.

Effect on Updates per Second with Latency Guarantees: In the second set of experiments, we try to maximize the number of full updates per second when a particular latency is targeted for a partial update query. Figures 40(a) and 40(b) depict the performance achieved by the application. For a given latency constraint, TCP cannot have a block size greater than a certain value. With data chunking done to suit this requirement, the bandwidth it can achieve is quite limited as seen in the figure under legend 'TCP'. With the same block size, SocketVIA achieves a much better performance, shown by legend 'SocketVIA'. However, a re-chunking of data that takes the latency and bandwidth of SocketVIA into consideration results in a much higher performance, as shown by the performance numbers for 'SocketVIA (with DR)'. Figure 40(a) gives the performance with no computation, while computation cost, which varies linearly with the size of the chunk, is introduced in the experiments for Figure 40(b). With no computation cost, as the latency constraint becomes as low as 100μs, TCP drops out. However, SocketVIA continues to give a performance close to the peak value. The results of this experiment show an improvement of more than 6 times without any repartitioning of data, and more than 8 times with repartitioning of data. With a computation cost, we see that for a large latency guarantee, TCP and SocketVIA perform very closely. The reason for this is the computation cost in the message path. With a computation cost of 18ns per byte, processing of data becomes a bottleneck with VIA. However, with TCP, the communication is still the bottleneck. Because of the same reason, unlike TCP, the frame rate achieved by SocketVIA does not change very much as the requested latency is decreased. The results for this experiment show a performance improvement of up to 4 times.

Effect of Multiple queries on Average Response Time: In the third set of experiments, we consider a model where there is a mixture of two kinds of queries. The first query type is a zoom or a magnification query, while the second one is a complete update query. The first query covers a small region of the image, requiring only 4 data chunks to be retrieved. However, the second query covers the entire image, hence all the data chunks should be retrieved and processed. Figures 41(a) and 41(b) display the average response time to queries. The x-axis shows the fraction of queries that correspond to the second type. The remaining fraction of queries correspond to the first type. The volume of data chunks accessed for each query depends on the partitioning of the dataset into data chunks. Since the fraction of queries of each kind may not be known a priori, we analyze the performance given by TCP and SocketVIA with different partition sizes. If the dataset is not partitioned into chunks, a query has to access the entire data, so the timings do not vary with varying fractions of the queries. The benefit we see for SocketVIA compared to TCP is just the

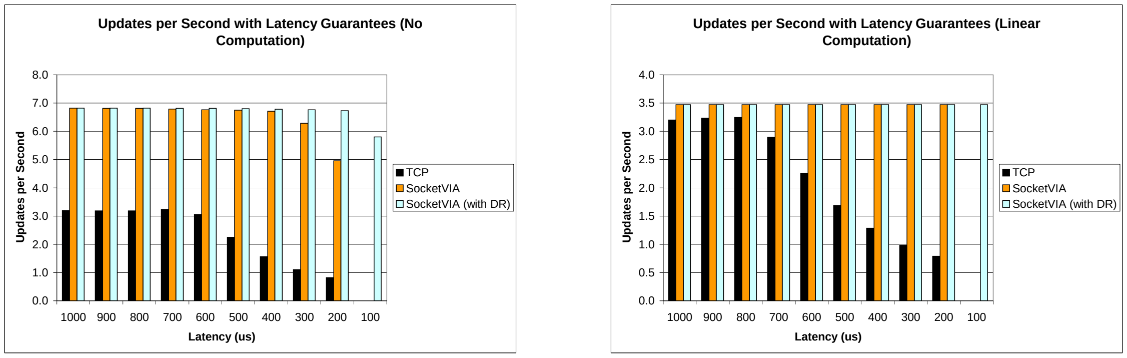


Figure 40: Effect of High Performance Sockets on Updates per Second with Latency Guarantees for (a) No Computation Cost and (b) Linear Computation Cost

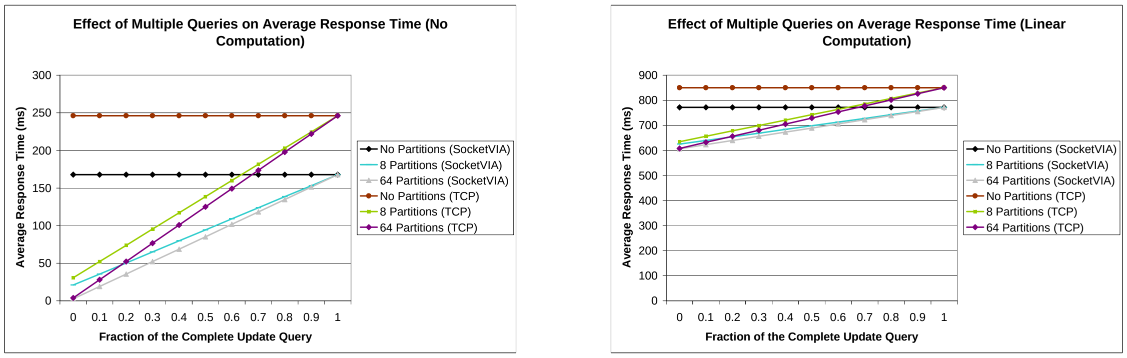


Figure 41: Effect of High Performance Sockets on the Average Response Time of Queries for (a) No Computation Cost and (b) Linear Computation Cost

inherent benefit of SocketVIA and has nothing to do with the partition sizes. However, with a partitioning of the dataset into smaller chunks, the rate of increase in the response time is very high for TCP compared to SocketVIA. Therefore, for any given average response time, SocketVIA can tolerate a higher variation in the fraction of different query types than TCP. For example, for an average response time of 150ms and 64 partitions per block, TCP can support a variation from 0% to 60% (percentage of the complete update queries), but fails after that. However, for the same constraint, SocketVIA can support a variation from 0% to 90% before failing. This shows that in cases where the block size cannot be pre-defined, or just an estimate of the block size is available, SocketVIA can do much better.

5.4.3 Effect of SocketVIA on Heterogeneous Clusters

In the next few experiments, we analyze the effect of SocketVIA on a cluster with a collection of heterogeneous compute nodes. We emulate slower nodes in the network by making some of the nodes do the processing on the data more than once. For host-based protocols like TCP, a decrease in the processing speed would result in a degradation in the communication time, together with a degradation in the computation time. However, in these experiments, we assume that communication time remains constant and only the computation time varies.

Effect of the Round-Robin scheduling scheme on Heterogeneous Clusters: For this experiment, we examine the impact on performance of the round-robin (RR) buffer scheduling in DataCutter when TCP and SocketVIA are employed. In order to achieve perfect pipelining, the time taken to transfer the data to a node should be equal to the processing time of the data on each of the nodes. For this experiment, we have considered load balancing between the filters of the Visualization Application (the first nodes in the pipeline, Figure 38). The processing time of the data in each filter is linear with message size (18ns per byte of message). With TCP, a perfect pipeline was observed to be achieved by 16KB message. But, with SocketVIA, this was achieved by 2KB messages. Thus, load balancing can be done at a much finer granularity.

Figure 42 shows the amount of time the load balancer takes to react to the heterogeneity of the nodes, with increasing factor of heterogeneity in the network. The factor of heterogeneity is the ratio of the processing speeds of the fastest and the slowest processors. With TCP, the block size is large (16KB). So, when the load balancer makes a mistake (sends a block to a slower node), it results in the slow node spending a huge amount of time on processing this block. This increases the time the load balancer takes to realize its mistake. On the other hand, with SocketVIA, the block size is small. So, when the load balancer makes a mistake, the amount of time taken by the slow node to process this block is lesser compared to that of TCP. Thus the reaction time of the load balancer is lesser. The results for this experiment show that with SocketVIA, the reaction time of the load balancer decreases by a factor of 8 compared to TCP.

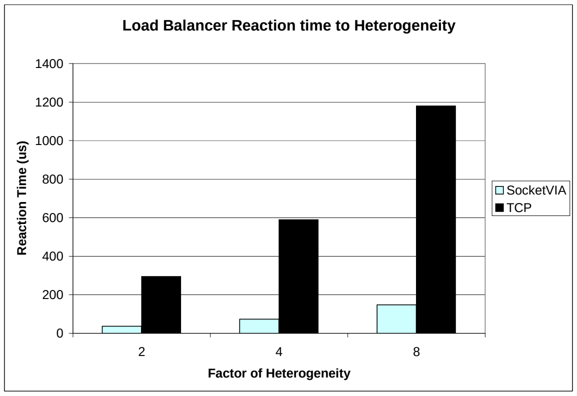


Figure 42: Effect of Heterogeneity in Processing Speed on Load Balancing using the Round-Robin Scheduling Scheme

Effect of the Demand-Driven scheduling scheme on Heterogeneous Clusters: For this experiment, we examine the impact on performance of the demand-driven (DD) buffer scheduling in DataCutter when TCP and SocketVIA are employed. Due to the same reason as the Round-Robin scheduling (mentioned in the last subsection), a block size of 2KB was chosen for socketVIA and a block size of 16KB for TCP.

Figure 43 shows the execution time of the application. The node is assumed to get slow dynamically at times. The probability of the node becoming slower is varied on the x-axis. So, a probability of 30% means that, 30% of the computation is carried out at a slower pace, and the remaining 70% is carried out at the original pace of the node. In Figure 43, the legend socketVIA(n) stands for the application running using socketVIA and a factor of heterogeneity of 'n'. The other legends are interpreted in a similar manner.

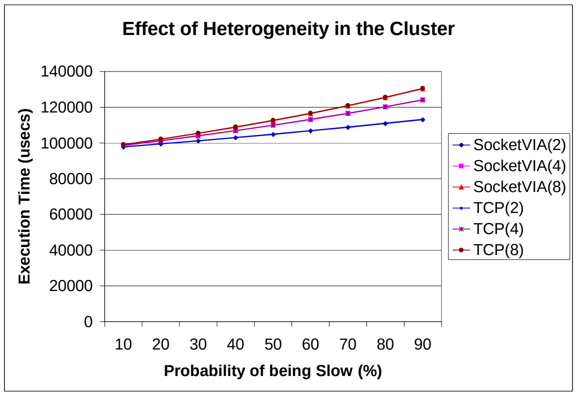


Figure 43: Effect of Heterogeneity in Processing Speed on Load Balancing using the Demand-Driven Scheduling Scheme

We observe that application performance using TCP is close to that of socketVIA. This is mainly because of the fact that demand-driven assignment of data chunks to consumers allows more work to be routed to less loaded processors. In addition, pipelining of data results in good overlap between communication and computation. Thus, our results show that if high-performance substrates are not available on a hardware configuration, applications should be structured to take advantage of pipelining of computations and dynamic scheduling of data. However, as our earlier results show, high-performance substrates are desirable for performance and latency guarantees.

6 Continuing and Future Work

In Section 4, we had mentioned several design issues that need to be dealt with in order to have an integrated framework with high performance sockets extended with DDP and RDMAP functionalities. In Section 5, we showed some of the results in some of the issues we had handled and other preliminary studies we had done. In this section, we describe the work we are currently carrying out and that needs to be performed in order to have a complete integrated framework for DDP extended high performance sockets.

1. Designing and Implementing the zero-copy high performance sockets implementation for InfiniBand described in Section 4.2.4. We have completed some initial implementation of this task; the preliminary results of this implementation show that the peak performance achievable is close to the maximum the underlying protocol can provide. We expect to complete the rest of the implementation in about two months.

2. Designing and Implementing DDP over TCP Offload Engines (described in Section 4.4.2). We plan to implement this on the TCP Offload Engine based network adapters supplied by Ammasso whose 16 node cluster is available. We have completed some initial implementation of this task and expect to complete it in one month time.

3. Implementing Active Cache and Dynamic Reconfigurability over DDP (described in Section 4.4.3). We expect this task to be complete in one month time.

4. Implementing DDP, RDMAP and potentially TCP/IP fast path on to the network adapter (described in Section 4.4.4). We expect this task to be complete in about six months time.

7 Significance and Expected Contribution

The work proposed in this proposal is to design and implement an integrated framework with the following capabilities: (i) the framework should be based on high performance sockets implementations; this would allow application to utilize the benefits of the POE based solutions proposed above, (ii) the framework should encompass the Direct Data Placement (DDP) and RDMA over IP (RDMAP) standards but only as an extended sockets interface; this would allow users to make changes to their applications as and when required and not place it as a primary requirement for them to run and (iii) the framework should try to either utilize a completely offloaded TCP/IP protocol stack or an emulation of the same; this would allow the implementation to be compatible with the IP-based Wide Area Network (WAN) environment and capable of being used in a distributed System Area Network (SAN) environment.

Since our framework is designed to allow applications to have a common protocol to achieve zero-copy high performance communication in a System Area Network (SAN) environment and TCP/IP compatibility in a WAN environment, we expect it to have significant impact in several TCP/IP based applications used either in a cluster-based SAN environment or in a SAN/WAN environment. We have demonstrated the impact it can have in some environments such as distributed cluster-based data-centers. We expect other applications such as those based on grid middleware (e.g., Globus), grid aware message passing (e.g., Grid MPI), distributed storage systems (e.g., iSCSI), distributed file management infrastructures and file systems (e.g., Parallel NFS, Lustre), etc., to be able to utilize and benefit from this significantly. Broadly, we expect this work to significantly impact the performance of existing TCP/IP based applications as well as the design of next generation network architectures.

References

[1] IBAL: InfiniBand Linux SourceForge Project. http://infiniband.sourceforge.net/IAL/Access/IBAL.

[2] IP over InfiniBand Working Group. http://www.ietf.org/html.charters/ipoib-charter.html.

[3] IPoIB: InfiniBand Linux SourceForge Project. http://infiniband.sourceforge.net/NW/IPoIB/overview.htm.

[4] M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via/.

[5] Mellanox Technologies. http://www.mellanox.com.

[6] Sockets Direct Protocol. http://www.infinibandta.org.

[7] The DAT Collaborative. http://www.datcollaborative.org/udapl.html.

[8] The DAT Collaborative. http://www.datcollaborative.org/kdapl.html.

[9] TOP 500 Supercomputer Sites. http://www.top500.org.

[10] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[11] Infiniband Trade Association. http://www.infinibandta.org.

[12] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand: Is it Beneficial? Technical Report OSU-CISRC-10/03-TR54, The Ohio State University, 2003.

[13] P. Balaji, H. V. Shah, and D. K. Panda. Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. In *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT)*, San Diego, CA, Sep 20 2004.

[14] M. Banikazemi, V. Moorthy, L. Hereger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP switch-connected NT clusters. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2000.

[15] Mohammed Banikazemi, V. Moorthy, L. Hereger, Dhabaleswar K. Panda, and Bulent Abali. Efficient Virtual Interface Architecture Support for IBM SP switch-connected NT clusters. In *the Proceedings of International Parallel and Distributed Processing Symposium*, pages 33–42, May 2000.

[16] TPC-W Benchmark. http://www.tpc.org.

[17] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a Framework for Data-Intensive Wide-Area Applications. pages 116–130, May 2000.

[18] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.

[19] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed Processing of Very Large Datasets with DataCutter. *Parallel Computing*, October 2001.

[20] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[21] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[22] Adam D. Bradley and Azer Bestavros. Basis Token Consistency: Extending and Evaluating a Novel Web Consistency Algorithm. In *the Proceedings of Workshop on Caching, Coherence, and Consistency (WC3)*, New York City, 2002.

[23] Adam D. Bradley and Azer Bestavros. Basis token consistency: Supporting strong web cache consistency. In *the Proceedings of the Global Internet Worshop*, Taipei, November 2002.

[24] P. Buonadonna, J. Coates, S. Low, and D.E. Culler. Millennium Sort: A Cluster-Based Application for Windows NT using DCOM, River Primitives and the Virtual Interface Architecture. In *the Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[25] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *Proceedings of Supercomputing*, 1998.

[26] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the Web. In *Middleware Conference*, 1998.

[27] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[28] U. Catalyurek, M. D. Beynon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The virtual microscope. *IEEE Transactions on Information Technology in Biomedicine*. To appear.

[29] Common Component Architecture Forum. *http://www.cca-forum.org.*

[30] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Sigmetrics '03*.

[31] L. Cherkasova and S. R. Ponnekanti. Optimizing a content-aware load balancing strategy for shared Web hosting service. In *the Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems '00.*

[32] Michele Colajanni and Philip S. Yu. Adaptive ttl schemes for load balancing of distributed web servers. *SIGMETRICS Perform. Eval. Rev.*, 25(2):36–42, 1997.

[33] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[34] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[35] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[36] Myricom Corporations. The GM Message Passing System.

[37] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *Proceedings of the IEEE International Conference on Supercomputing*, Phoenix, Arizona, November 2003.

[38] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP 1.1. RFC 2616. June, 1999.

[39] H. Frazier and H. Johnson. Gigabit Ethernet: From 100 to 1000Mbps.

[40] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro*, January 2004.

[41] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *Proceedings of Cluster Computing*, 2001.

[42] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol. IETF Internet Draft, November 2000.

[43] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[44] Mikhail Mikhailov and Craig E. Wills. Evaluating a New Approach to Strong Web Cache Consistency with Snapshots of Collected Content. In *WWW2003, ACM*, 2003.

[45] S. Narravula, P. Balaji, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand. In *SAN*, 2004.

[46] R. Oldfield and D. Kotz. Armada: A parallel file system for computational. In *Proceedings of CCGrid2001*, May 2001.

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

[48] P. Palaji, K. Vaidyanathan, S. Narravula, H. W. Jin K. Savitha, and D. K. Panda. Exploiting Remote Memory Operations to Design Efficient Reconfiguration for Shared Data-Centers over InfiniBand. In *Proceedings of Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT 2004)*, San Diego, CA, September 2004.

[49] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *HPDC*, August 2000.

[50] R. Recio, P. Culley, D. Garcia, and J. Hillard. IETF Draft: RDMA Protocol Specification, November 2002.

[51] Hemal V. Shah, Dave B. Minturn, Annie Foong, Gary L. McAlpine, Rajesh S. Madukkarumukumana, and Greg J. Regnier. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *the Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages pages 61–72, San Francisco, CA, March 2001.

[52] Hemal V. Shah, James Pinkerton, Renato Recio, and Paul Culley. Direct Data Placement over Reliable Transports, November 2002.

[53] Weisong Shi, Eli Collins, and Vijay Karamcheti. Modeling Object Characteristics of Dynamic Web Content. *Special Issue on scalable Internet services and architecture of Journal of Parallel and Distributed Computing (JPDC)*, Sept. 2003.

[54] Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *the Proceedings of the IEEE International Conference on Supercomputing*, pages 57–64, Denver, Colorado, November 10-16 2001.

[55] Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. Can User-Level Protocols Take Advantage of Multi-CPU NICs? In *the Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 15-19 2002.

[56] W. R. Stevens. *TCP/IP Illustrated, Volume I: The Protocols.* Addison Wesley, 2nd edition, 2000.

[57] Internet Traffic Archive Public Tools. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[58] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume II: The Implementation.* Addison Wesley, 2nd edition, 2000.

[59] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[60] Eric Yeh, Herman Chao, Venu Mannem, Joe Gervais, and Bradley Booth. Introduction to tcp/ip offload engines (toe). White Paper, May 2002.

[61] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Web Cache Consistency. *ACM Transactions on Internet Technology, 2:3,*, August. 2002.