# High Performance Communication Support for Sockets-based Applications over High-speed Networks

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Pavan Balaji, B.Tech.

* * * * *

The Ohio State University

2006

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. M. Lauria

Approved by

Adviser
Department of Computer
Science and Engineering

# ABSTRACT

In the past decade several high-speed networks have been introduced, each superseding the others with respect to raw performance, communication features and capabilities. However, such aggressive initiative is accompanied by an increasing divergence in the communication interface or “language” used by each network. Accordingly, portability for applications across these various network languages has recently been a topic of extensive research. Programming models such as the Sockets Interface, Message Passing Interface (MPI), Shared memory models, etc., have been widely accepted as the primary means for achieving such portability.

This dissertation investigates the different design choices for implementing one such programming model, i.e., Sockets, in various high-speed network environments (e.g., InfiniBand and 10-Gigabit Ethernet). Specifically, the dissertation targets three important sub-problems: (a) designing efficient sockets implementations to allow existing applications to be directly and transparently deployed on to clusters connected with high-speed networks; (b) analyzing the limitations of the sockets interface in various domains and extending it with features that applications need but are currently missing; and (c) designing a communication substrate to allow compatibility between various kinds of protocol stacks belonging to a common network family (e.g., Ethernet). The proposed stack comprising of the above mentioned three components, allows development of applications and other upper layers in an efficient, seamless and globally compatible manner.

# ACKNOWLEDGMENTS

I would like to thank my adviser Prof. D. K. Panda and Prof. P. Sadayappan for their guidance, friendship and encouragement throughout my Ph.D. study. I am grateful for their effort and patience and the time that they dedicated in making me a better researcher, teacher and mentor. Without their help I would not be where I am at this point.

I would also like to thank Prof. M. Lauria and Dr. P. Wyckoff for agreeing to be in my dissertation and candidacy committees, their valuable comments and suggestions.

I am grateful to Dr. Hemal Shah and Dr. Scott Hahn of Intel Corporation and Dr. Wu-chun Feng of Los Alamos National Laboratory (LANL) for their guidance and support during my summer internships and afterwards. I am especially grateful to Dr. Feng, who not only was my team lead at LANL, but also a good friend and mentor during the course of my Ph.D.

I would also like to thank my many friends at OSU: Seetha, Jin, Karthik, Amith, Sayantan, Abhinav, Gopal, Ranjit, Weikuan, Wei, Qi, Lei, Prachi, Matt, Shuang, Jerry, Kamrul, Gaurav, Naga, Bala, Sandhya, Vijay, Raj, Chubbs and many many others for the numerous technical discussions and feedback about my work. I would especially like to thank Seetha who has been a very dear friend and taught me to be passionate about my work, and Jin who taught me to be a good "senior student" in the group and added a huge amount of rigor to my research.

Finally, I would like to thank my fiance, Rinku Gupta, who was also a former member of NOWLAB at OSU, my parents, my brother and my sister-in-law for their love and support.

# VITA

| | |
|---|---|
| May 14, 1980 | Born - Hyderabad, India |
| August 1997 - August 2001 | B.Tech, Computer Science and Engg., IITM, India |
| September 2001 - December 2001 | Graduate Teaching Associate, The Ohio State University, Ohio |
| June 2002 - September 2002 | Summer Intern, Intel Corporation, Texas |
| June 2003 - September 2003 | Summer Intern, Intel Corporation, Texas |
| March 2005 - August 2005 | Summer Intern, Los Alamos National Laboratory, New Mexico |
| January 2002 - June 2006 | Graduate Research Associate, The Ohio State University, Ohio |

# PUBLICATIONS

M. Islam, P. Balaji, P. Sadayappan and D. K. Panda "Towards Quality of Service Guarantees in Job Scheduling (extended journal version)." *International Journal of High Performance Computing and Networking (JHPCN)*

P. Balaji, G. Sabin and P. Sadayappan "Opportune Job Shredding: An Efficient Approach for Scheduling Parameter Sweep Applications (extended journal version)." *International Journal for High Performance Computing Applications (JHPCA)*

P. Balaji, W. Feng and D. K. Panda "Bridging the Ethernet-Ethernot Performance Gap." *IEEE Micro Journal*, 2006.

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." *Special Issue of the Journal of Parallel and Distributed Computing (JPDC)*, 2005.

M. Islam, P. Balaji, P. Sadayappan and D. K. Panda "QoPS: A QoS based scheme for Parallel Job Scheduling (extended journal version)." *IEEE Springer LNCS Journal Series*, 2003.

P. Balaji, S. Bhagvat, H. -W. Jin and D. K. Panda "Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol (SDP) over InfiniBand." *Workshop on Communication Architecture for Clusters (CAC); in conjunction with the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

P. Balaji, K. Vaidyanathan, S. Narravula, H. -W. Jin and D. K. Panda "Designing Next Generation Data-centers with Advanced Communication Protocols and Systems Services." *Workshop on the National Science Foundation Next Generation Software (NSFNGS) Program; in conjunction with the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

V. Viswanathan, P. Balaji, W. Feng, J. Leigh, D. K. Panda "A Case for UDP Offload Engines in LambdaGrids." *Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2006.

P. Balaji, W. Feng, Q. Gao, R. Noronha, W. Yu and D. K. Panda "Head-to-TOE Comparison for High Performance Sockets over Protocol Offload Engines." *Proceedings of IEEE International Conference on Cluster Computing (Cluster)*, 2005.

P. Balaji, H. -W. Jin, K. Vaidyanathan and D. K. Panda "Supporting RDMA capable network compatibility and features for regular network adapters." *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations and Techniques (RAIT); in conjunction with IEEE International conference on Cluster Computing (Cluster)*, 2005.

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." *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW); in conjunction with IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2005.

W. Feng, P. Balaji, C. Baron, L. N. Bhuyan and D. K. Panda "Performance Characterization of a 10-Gigabit Ethernet TOE." *Proceedings of IEEE International Symposium on High Performance Interconnects (HotI)*, 2005.

H. -W. Jin, S. Narravula, G. Brown, K. Vaidyanathan, P. Balaji and D. K. Panda "RDMA over IP Networks: A Study with the Ammasso Gigabit Ethernet NIC." *Workshop on High Performance Interconnects for Distributed Computing (HPI-DC); in conjunction with IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2005.

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

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." *Proceedings of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2005.

M. Islam, P. Balaji, P. Sadayappan and D. K. Panda "Towards Provision of Quality of Service Guarantees in Job Scheduling." *Proceedings of IEEE International Conference on Cluster Computing (Cluster)*, 2004.

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." *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations and Technologies (RAIT); in conjunction with IEEE International Conference on Cluster Computing (Cluster)*, 2004.

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." *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations and Technologies (RAIT); in conjunction with IEEE International Conference on Cluster Computing (Cluster)*, 2004.

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

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." *Workshop on System Area Networks (SAN); in conjunction with IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2004.

R. Kurian, P. Balaji and P. Sadayappan "Opportune Job Shredding: An Efficient Approach for Scheduling Parameter Sweep Applications." *Proceedings of Los Alamos Computer Science Institute (LACSI) Symposium*, 2003.

P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda and J. Saltz "Impact of High Performance Sockets on Data Intensive Applications." *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2003.

M. Islam, P. Balaji, P. Sadayappan and D. K. Panda "QoPS: A QoS based scheme for Parallel Job Scheduling." *Job Scheduling Strategies for Parallel Processing (JSSPP) workshop; in conjunction with IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2003.

R. Gupta, P. Balaji, J. Nieplocha and D. K. Panda "Efficient Collective Operations using Remote Memory Operations on VIA-based Clusters." *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

P. Balaji, P. Shivam, P. Wyckoff and D. K. Panda "High Performance User-level Sockets over Gigabit Ethernet." *Proceedings of IEEE International Conference on Cluster Computing (Cluster)*, 2002.

# FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

| | | |
|---|---|---|
| Computer Architecture | : | Prof. D.K. Panda |
| Computer Networks | : | Prof. Dong Xuan |
| Software Engineering | : | Prof. Mario Lauria |

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

**Figure** **Page**

# CHAPTER 1

# INTRODUCTION

The last couple of decades have seen great advances in every part of machine and computer technology. Computers have become ubiquitous in every area, whether industry or research, and now impact every aspect of human life. In early 70s, when standard computers were gaining popularity among researchers, it was realized that there was a need for more powerful machines that could solve problems that were too complex and massive for standard computers. This realization led to the development of 'Supercomputers' – advanced and powerful machines consisting of multiple processing units. Cray-1, developed by Cray Research was one such powerful supercomputer. As supercomputers and computer technology evolved, there has been an exponential growth in the demands by applications. The high cost for designing, developing and maintaining these supercomputers, for meeting the high performance application demands, led researchers to seek an alternative to these supercomputers in the form of cluster-based systems or clusters in short.

Clusters consist of cheap commodity-off-the-shelf (COTS) PCs connected together with network interconnects. These PCs interact with each other over the network to project themselves as fast 'Supercomputers' whose aggregate capability is much higher than any of the individual PCs. Such clusters are becoming increasingly popular in various application domains mainly due to their high performance-to-cost ratio. These systems can now be designed for 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 [51].

Since a cluster-based system relies on multiple inexpensive PCs interacting with each other over the network, the capability of the network (the hardware as well as the associated communication software) forms a critical component in its efficiency and scalability. Figure 1.1 shows typical environments used by parallel and distributed computing applications. Environments can range from localized clusters, to multiple assorted clusters connected over WANs or High-speed backbone networks.

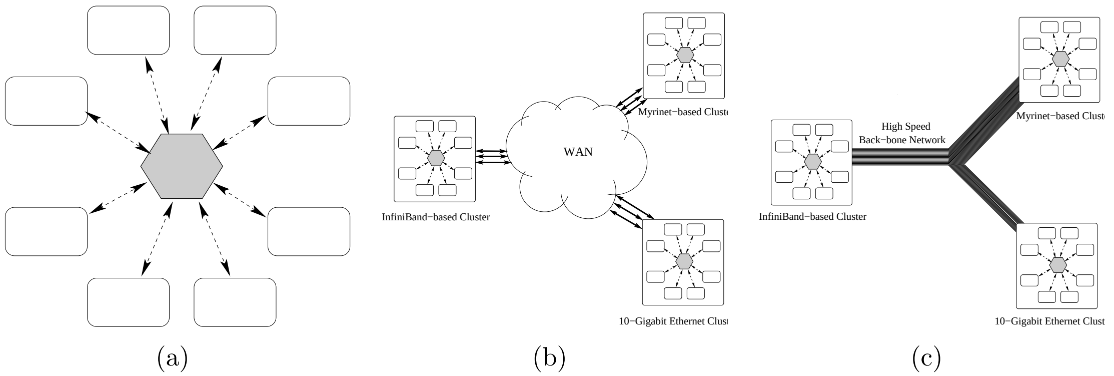

Figure 1.1: Typical Environments for Distributed and Parallel Applications:(a) Communication within the cluster environment, (b) Inter-Cluster Communication over a WAN and (c) Inter-Cluster Communication over a High-Speed Backbone Network

In the past few years, a number of high-speed networks including Gigabit [46] and 10-Gigabit Ethernet [52, 45], GigaNet cLAN [54], InfiniBand (IB) [10], Myrinet [24], Quadrics [4], etc., have been introduced in the market. With the advent of such multi-gigabit per second speed networks, 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 [78, 81] 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 [68] and GM [42] for Myrinet, EMP [76, 77] for Gigabit Ethernet, etc. To standardize these efforts, in the late 90s, the Virtual Interface Architecture (VIA) [27, 3, 54] was proposed; but it was able to achieve only limited success. The industry has recently standardized the InfiniBand Architecture to design next generation high-end clusters. All these developments are reducing the gap between the performance capabilities of the physical network and that obtained by the end users.

However, each of these new networks and user-level protocols exposes a new 'Application Programing Interface (API)' or "language" for the user to interact with. While these new APIs can be efficiently used to develop new applications, they may not be beneficial for the already existing applications which were developed over a span of several years. Application developers while writing applications aim for portability across various future platforms and networks. Programming models such as the Message Passing Interface (MPI) [60] and the Sockets Interface have been widely accepted as a feasible approach to achieve such portability.

For the last several years, MPI has been the de facto standard for scientific applications. The Sockets Interface, on the other hand, has been the most widely used programming model for traditional scientific applications, commercial applications, file and storage systems, etc.

## 1.1 The Sockets Interface: 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. High-speed networks on the other hand, together with raw network performance, provide a lot of additional features such as offloaded protocol stacks, one-sided communication operations, atomic operations and so on.

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?

- Networks such as Ethernet provide multiple flavors each differing from the other with respect to the wire-protocol or language they speak on the wire. Is it possible for a sockets implementation to allow transparent wire compatibility for such different flavors of the networks while retaining the highest possible performance?

## 1.2 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. Hardware-offloaded protocol stacks or Protocol Offload Engines (POEs) try to alleviate this problem by offloading either TCP/IP or some other protocol stack on to the network adapter. Sockets protocol stack implementations allow applications to access these offloaded protocol stacks without requiring any modifications. Figure 1.2 demonstrates the traditional approaches and our proposed sockets protocol stack approach for allowing compatibility for sockets based applications.

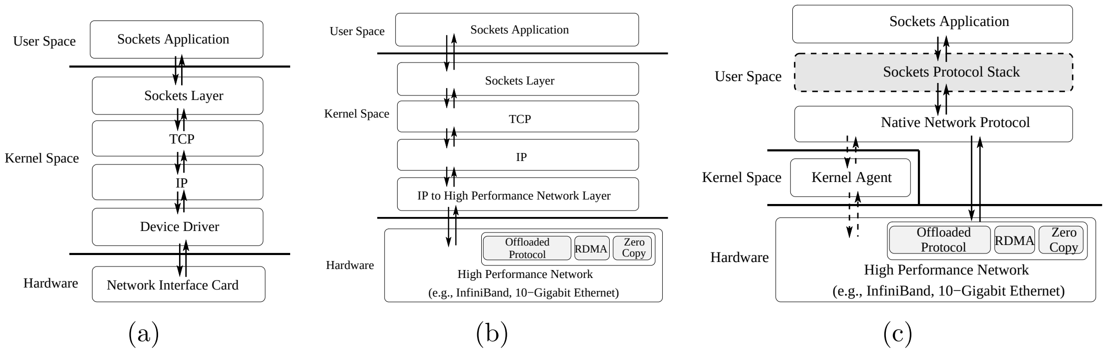

Figure 1.2: Approaches to the Sockets Interface: (a) Traditional, (b) Mapping IP to the offloaded protocol layers (such as VIA and IBA), and (c) Mapping the Sockets layer to the User-level protocol

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 1.2(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 [54] (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 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) which maps IP communications onto the NIC (Figure 1.2(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 1.2(a). Although this approach gives us the required compatibility with existing sockets implementations, it can not be expected to give any performance improvement.

The sockets protocol stack based solution creates an intermediate layer which maps the sockets library onto the offloaded protocol stack provided by the network adapter. This layer ensures the highest possible performance without necessitating any changes. Figure 1.2(c) provides an overview of the proposed Sockets Protocol Stack architecture.

## 1.3 Dissertation Overview

Keeping in mind the issues associated with the basic high performance sockets implementations, we propose a multi-component integrated framework with the following capabilities:

1. The framework should allow applications to not only run directly on the high-speed networks without any modifications, but also be able to extract the best performance from the network.

2. Instead of blindly following the sockets interface, the framework should encompass the key features provided by high-speed networks into 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 maintain wire-protocol compatibility between different networks belonging to a common family (e.g., different Ethernet networks). This ensures that applications can directly and transparently execute on a cluster which has multiple networks while retaining most of the performance of the networks.

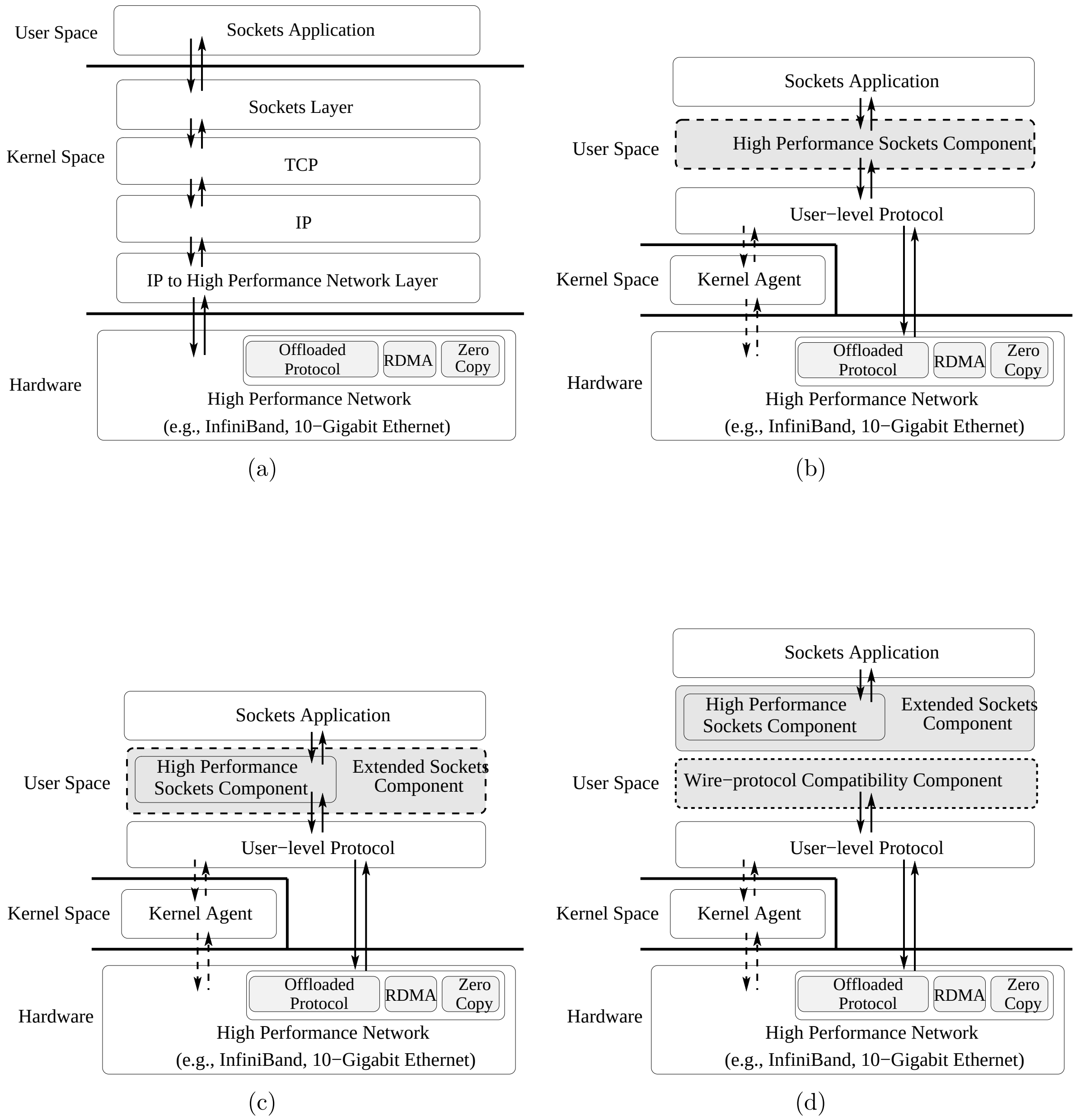

Figure 1.3: The Proposed Framework: (a) Existing Infrastructure for sockets based applications, (b) High Performance sockets component to take advantage of the offloaded protocol stacks, (c) Extended sockets component to encompass the most relevant features provided by high-speed networks into the sockets interface and (d) Wire-protocol compatibility component to achieve transparent interaction between different flavors of a common network such as Ethernet.

The focus of this dissertation is to provide an integrated framework known as the sockets protocol stack, which provides a high-performance, feature rich and globally compatible middleware for existing sockets-based applications to take advantage of. Figure 1.3 shows the existing sockets implementation (Figure 1.3(a)) together with the proposed stack comprising of three major components: (i) the high performance sockets component, (ii) the extended sockets component and (iii) the wire protocol compatibility component.

The High-performance sockets component (Figure 1.3(b)) forms the core of the sockets protocol stack. This component has two major goals: (i) to provide transparent compatibility for existing sockets-based applications over high-speed networks and (ii) to achieve such compatibility by retaining most of the performance provided by the networks by utilizing hardware-offloaded protocol stacks. Chapters 2 to 8 deal with the various design aspects and evaluations associated with this component.

The extended sockets component (Figure 1.3(c)) is primarily an extension of the high-performance sockets component. It questions the semantics of the existing sockets interface with respect to its capabilities in several scientific as well as commercial applications. Specifically, we perform detailed analysis in different environments and extend the sockets interface appropriately to be most beneficial in these environments. Chapters 9 and 10 deal with this component.

While achieving the best performance is highly desired, this has to be done in a globally compatible manner, i.e., all networks should be able to transparently take advantage of the proposed performance enhancements while interacting with each other. This, of course, is an open problem. In the wire-protocol compatibility component (Figure 1.3(d)), we pick a subset of this problem to provide such compatibility within the Ethernet family while trying to maintain most of the performance of the networks. Chapters 11 and 12 will deal with this component.

Conclusions and future research directions are indicated in Chapter 13.

# CHAPTER 2

# HIGH-PERFORMANCE USER-LEVEL SOCKETS OVER GIGABIT ETHERNET

Ethernet Message Passing (EMP) protocol [76, 77] is a high-performance protocol over Gigabit Ethernet (GigE) [46] that was developed by the Ohio Supercomputer Center and the Ohio State University. It provides a low overhead user-level protocol similar to VIA [27, 3, 54] to allow applications to utilize the capability and performance of the network. While such low-overhead protocols are good for writing new applications, it might not be so beneficial for the already existing applications written over standard interfaces such as sockets, that were developed over a span of several years.

In this research, we take on a challenge of developing a low overhead, user-level high-performance sockets interface on Gigabit Ethernet which uses EMP as the underlying protocol. There is no exact parallel between EMP and TCP/IP or UDP/IP. We analyze the semantic mismatches between the two protocols like connection management and unexpected message arrival to name a few. To capture these differences, we suggest various approaches for two commonly used options with sockets, namely data streaming and datagram. Finally, we suggest several performance enhancement techniques while providing these options and analyze each of them in detail.

Using our approach one will be able to transport the benefits of Gigabit Ethernet to the existing sockets application without necessitating changes in the user application itself. Our sockets interface is able to achieve a latency of 28.5 $\mu$s for the Datagram sockets and 37 $\mu$s for Data Streaming sockets compared to a latency of 120 $\mu$s obtained by TCP for 4-byte messages. We also attained a peak bandwidth of around 840 Mbps using our interface. In addition we tested our implementation on real applications like the File Transfer Protocol (FTP) and the Apache Web Server [6]. For FTP we got almost twice the performance benefit as TCP while Apache showed as much as six times performance enhancement.

## 2.1 Overview of EMP

In the past few years, a large number of user-level protocols have been developed to reduce the gap between the performance capabilities of the physical network and that achievable by an application programmer. The Ethernet Message Passing (EMP) protocol specifications have been developed at the Ohio Supercomputing Center and the Ohio State University to fully exploit the benefits of GigE.

EMP is a complete zero-copy, OS-bypass, NIC-level messaging system for GigE (Figure 2.2). This is the first protocol of its kind on GigE. It has been implemented on a network interface chip-set based around a general purpose embedded microprocessor design called the Tigon2 [65] (produced by Alteon Web Systems, now owned by Nortel Networks). This is a fully programmable NIC, whose novelty lies in its two CPUs on the NIC. Figure 2.1 provides an overview of the Alteon NIC architecture.

In EMP, message transmission follows a sequence of steps (Figure 2.2). First the host posts a transmit descriptor to the NIC (T1), which contains the location and

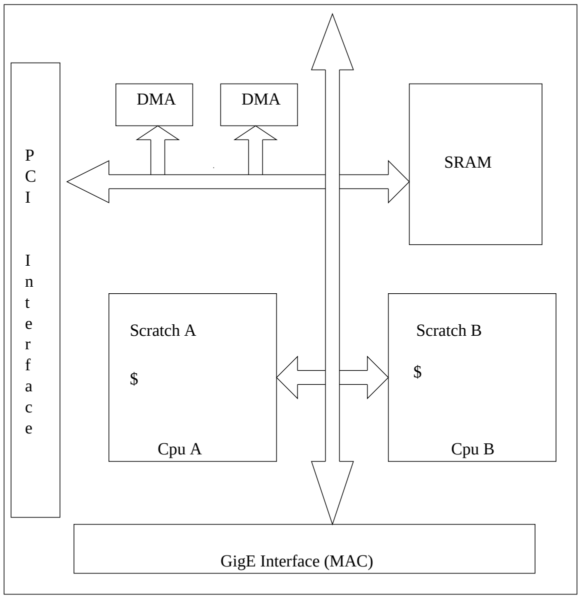

Figure 2.1: The Alteon NIC Architecture

length of the message in the host address space, destination node, and an application 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 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). One of the main features of this protocol is that it is a complete NIC

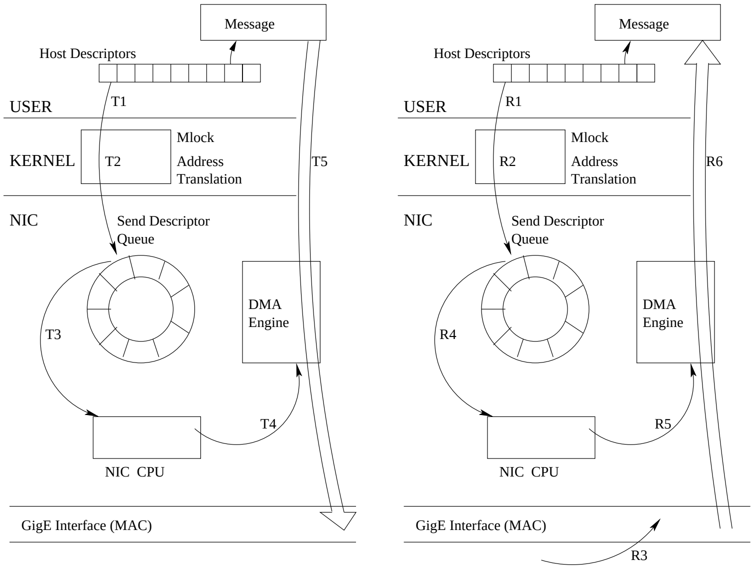

Figure 2.2: EMP protocol architecture showing operation for transmit (left), and receive (right).

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 of the NIC and the other utilizing both the CPUs of the Alteon NIC. In this chapter we have evaluated the performance of the Sockets programming model over EMP for the second implementation of EMP (using both the CPUs on the NIC).

## 2.2 Current Approaches

The traditional communication architecture involves just the application and the libraries in user space, while protocol implementations such as TCP/IP, UDP/IP, etc., reside in kernel space (Figure 2.3(a)). The kernel-based protocol stacks interact with network-specific device drivers (which also reside in the kernel) to communicate with the appropriate network adapter. Most of the protocol processing is handled

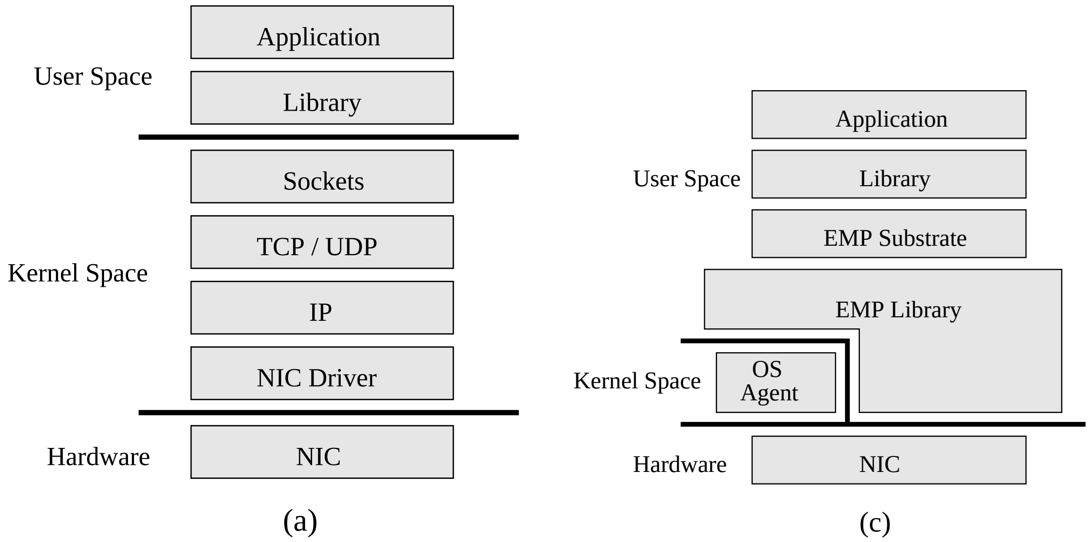

Figure 2.3: Approaches to the Sockets Interface: (a) Traditional, and (b) Mapping the Sockets layer to the User-level protocol

by the TCP/IP or UDP/IP stacks; the device drivers typically perform the minimal functionality of transmitting and receiving segments smaller than the maximum transmission unit (MTU) of the network in an unreliable fashion.

This approach is simple and can be easily ported to different network adapters using various device drivers. However, this approach does not fully utilize the capabilities of high-speed networks. In particular, it does not take advantage of hardware-offloaded protocol stack and advanced features provided by the network. Hence its performance is restricted by the implementation of the TCP/IP and UDP/IP stacks.

Most current networks, including the Alteon NICs, use this style of architecture. The device driver these NICs use is known as the Acenic driver.

The motivation for our work is to provide a high performance sockets layer over Gigabit Ethernet given the advantages associated with Gigabit Ethernet.

To be able to take advantage of the high performance offered by GigE, two important changes are required from the traditional sockets implementation. First, the TCP and IP layers must be removed to avoid message copies, which requires implementing a sockets library directly on top of a high-performance protocol for GigE (e.g., EMP, M-VIA [3]). Second, the entire interface library must exist in user space, to avoid the additional context switch to the kernel for every communication, in essence removing the kernel from the critical path.

M-VIA, while providing a high-performance VIA interface over Gigabit Ethernet, is a kernel-based protocol. Hence, due to kernel context switches on every data transfer event, it will not be able to exploit the complete benefits of Gigabit Ethernet. To the best of our knowledge EMP is the only complete OS-bypass, zero-copy and NIC-driven protocol over Gigabit Ethernet. Thus, we focus our research on the EMP protocol.

The solution proposed in this chapter creates an intermediate layer which maps the sockets library onto EMP. This layer ensures that no change is required to the application itself. This intermediate layer will be referred to as the "EMP Substrate". Figure 2.3(b) provides an overview of the proposed Sockets-over-EMP architecture.

## 2.3 Design Challenges

While implementing the substrate to support sockets applications on EMP, we faced a number of challenges. In this section, we mention a few of them, discuss the possible alternatives, the pros and cons of each of the alternatives and the justifications behind the solutions.

### 2.3.1 API Mismatches

The mismatches between TCP/IP and EMP are not limited to the syntax alone. The motivation for developing TCP/IP 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 the EMP substrate to support applications written using the sockets interface (on TCP/IP and UDP/IP), it must be kept in mind that the application was designed around the semantics of TCP/IP. We have identified the following significant mismatches in these two protocols and given solutions so as to maintain the semantics for each of the mismatches with regard to TCP/IP. More importantly, this has been done without compensating much on the performance given by EMP.

#### Connection Management

TCP/IP is a connection based protocol, unlike EMP, i.e., in TCP/IP, when a connection request is sent to the server, it contains important information about the client requesting the connection. Thus, in our approach, though explicit connection establishment is not required, we still need to carry out a three-way handshake in order to ensure that all the relevant information has been appropriately exchanged.

However, this puts an additional requirement on the substrate to post descriptors for the connection management messages too. When the application calls the listen() call, the substrate posts 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 substrate 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. In our solution, we explicitly handle unexpected messages at the substrate, and avoid these retransmissions. 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. This approach was evaluated and found to be too costly. With both threads polling, the synchronization cost of the threads themselves comes to about 20 $\mu$s. Also, the effective percentage of CPU cycles the main thread can utilize would go down to about 50%, assuming equal priority threads. In case of a blocking thread, the Operating System scheduling granularity makes the response time too coarse (order of milliseconds) for any performance benefit.

**Rendezvous Approach:** The second approach (similar to the approach indicated by [58]) is through rendezvous communication with the receiver as shown in Figure 2.4. 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.

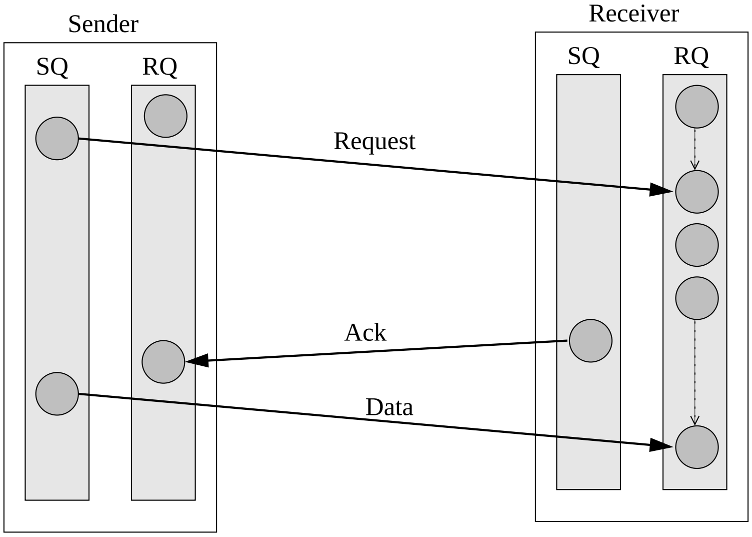

Figure 2.4: Rendezvous approach

**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 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 2.5 illustrates the eager approach with flow control.

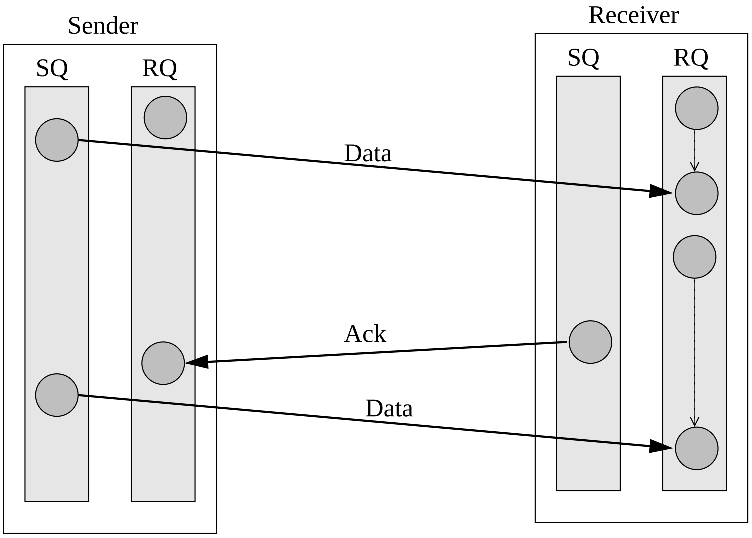

Figure 2.5: Eager with Flow Control

In Section 2.4.1, we propose an extension of this idea (with additional credits) to enhance its performance.

The first solution, using a separate communication thread, was not found to give any significant benefit in performance. However, the second and third approaches, namely the rendezvous and eager with flow control respectively, were found to give significant benefit in latency and bandwidth. Both these approaches have been implemented in the substrate, giving the user an option of choosing either one of them.

### 2.3.2 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).

In our approaches, we override the *libc* library to redirect these calls to the EMP substrate. For most cases, such overriding can be done using environment variables (e.g., LD_PRELOAD). This is the most non-intrusive approach for using the EMP substrate and can provide transparent high performance even for applications available only in binary format. For cases where the relevant symbols in the *libc* library are statically compiled into the application executable, the environment variable based approach will not be applicable. In this case, the application will have to be recompiled with the EMP substrate.

It is to be noted that in both these approaches, the application is not modified at all. However, the second approach requires recompilation of the application while the first approach does not require any recompilation either.

## 2.4 Performance Enhancement

While implementing the substrate, the functionality of the calls was taken into account so that the application does not have to suffer due to the changes. However, these adjustments do affect the performance the substrate is able to deliver. In order to improve the performance given by the substrate, we have come up with some

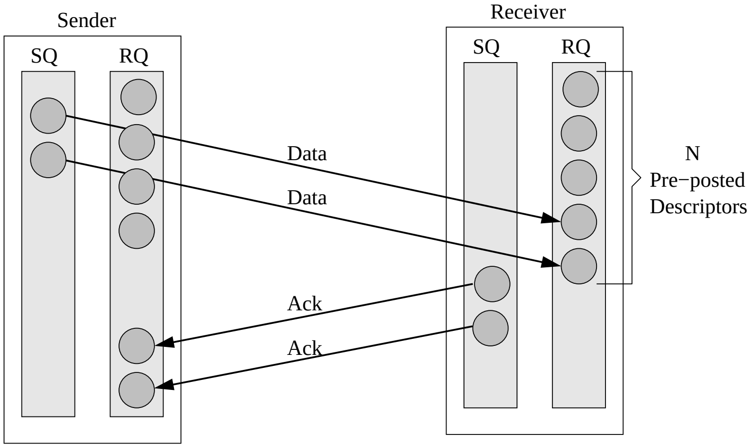

Figure 2.6: The Credit Based Approach

techniques, which are summarized below. More details on these techniques are are included in [16].

### 2.4.1 Credit-based flow control

As mentioned earlier (Section 2.3.1), the scheme we have chosen for handling unexpected messages can be extended to enhance its performance.

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 (Figure 2.6).

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 proposed 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.

**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 $2N$. On the basis of the same, this approach was used in the substrate.

### 2.4.2 Disabling Data Streaming

As mentioned earlier, 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 (e.g., those using UDP/IP datagram sockets) which do not need this option. To improve the performance of these applications, we have provided an option in the substrate 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.

### 2.4.3 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.

These delayed acknowledgments 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. However, with the increase in the number of credits given, the fraction of acknowledgment descriptors decreases, and thus reducing the effect of the time required for tag matching.

### 2.4.4 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 substrate, we 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.

These enhancements have been incorporated in the substrate and are found to give a significant improvement in the performance.

## 2.5 Performance Results

The experimental test-bed included 4 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.

### 2.5.1 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 2.7 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 2.4). 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 $\mu$s (an overhead of as low as 1 $\mu$s over EMP) for 4-bytes messages. The Data Streaming option with all enhancements turned on, is able to provide a latency of 37 $\mu$s for 4-byte messages.

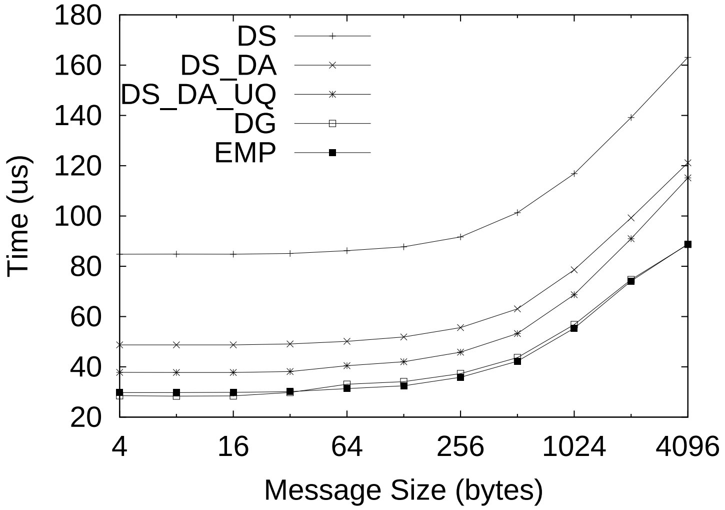

Figure 2.7: Micro-Benchmarks: Latency

Figure 2.8 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.

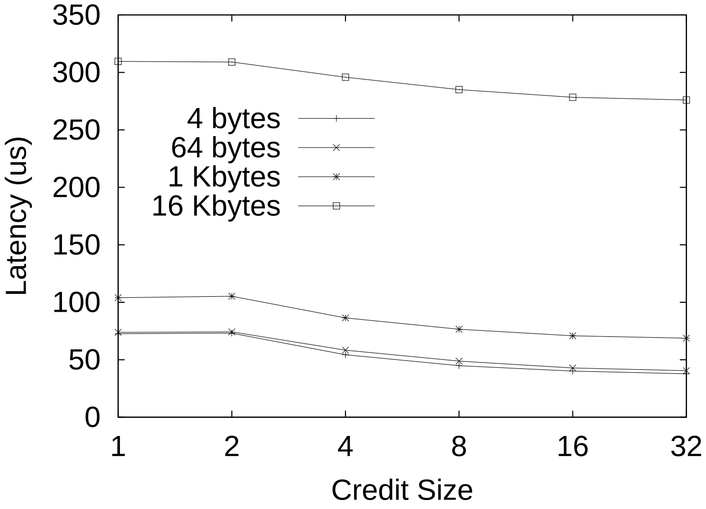

Figure 2.8: Micro-Benchmarks: Latency variation for Delayed Acknowledgments with Credit Size

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

### 2.5.2 Latency and Bandwidth

Figure 9.3 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).

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

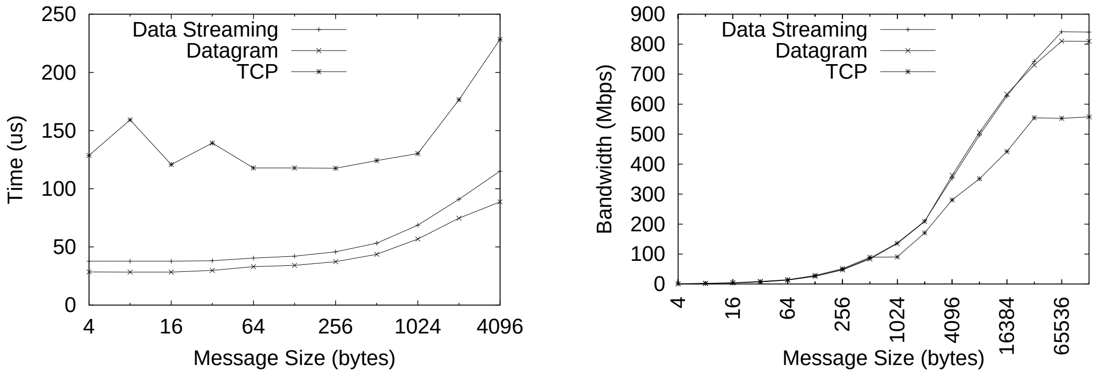

Figure 2.9: Micro-Benchmark Results: Latency (left) and Bandwidth (right)

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 $\mu$s for Datagram sockets and 37 $\mu$s 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.

### 2.5.3 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 $\mu$s for Data Streaming and Datagram sockets, compared to the 25.6 $\mu$s taken by TCP.

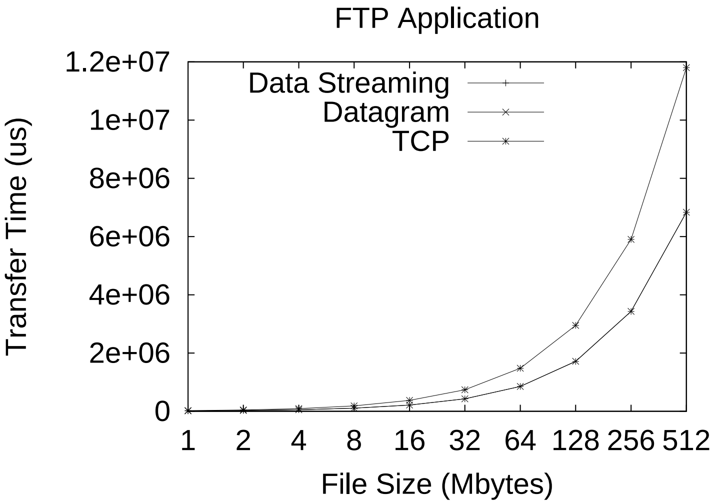

Figure 2.10: FTP Performance

The application is not able to achieve the peak bandwidth illustrated in Section 2.5.2, 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 the ftp application, is also attributed to the file system overhead.

### 2.5.4 Web Server Application

We have measured the performance obtained by the Apache Web Server application for a 4 node cluster (with one server and three clients).

The server keeps accepting requests from the clients. The clients connect to the server and send in a HTTP request message. The server accepts the connection and sends back a file 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

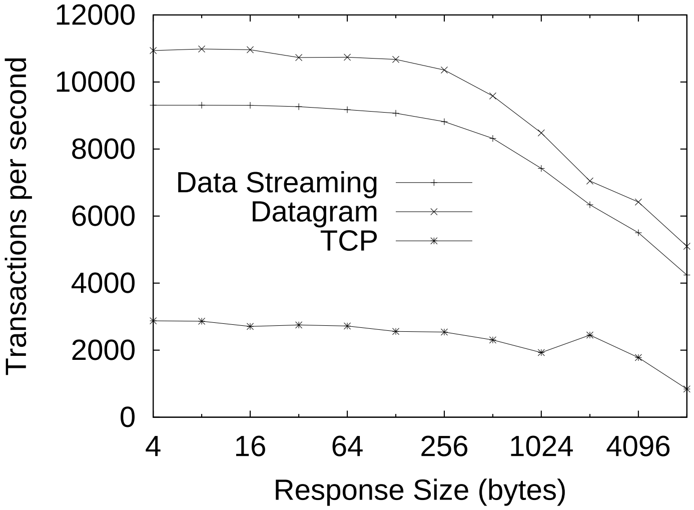

Figure 2.11: Web Server (HTTP/1.0)

per HTTP/1.0 specification). However, this was slightly modified in the HTTP/1.1 specification, 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 connection establishment is hidden.

Figure 2.11 gives the results obtained by the Web Server application following the HTTP/1.0 specifications.

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/IP, the connection time requires intervention by the kernel and is typically about 200 to

250 $\mu$s. 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 TCP/IP.

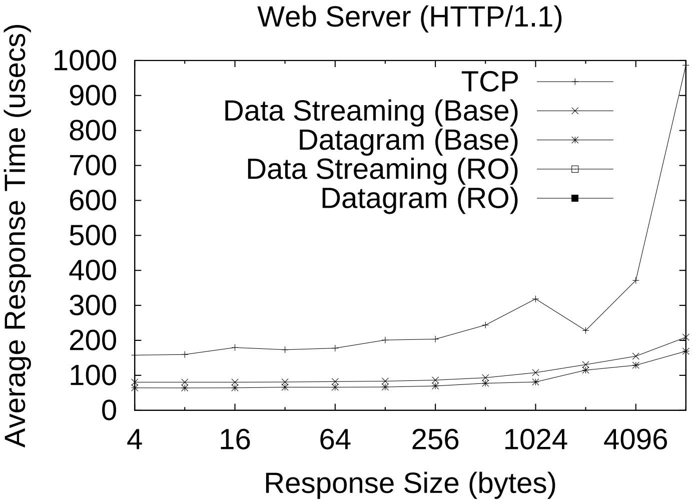

Figure 2.12: Web Server Average Response Time (HTTP/1.1)

## 2.6 Summary

Ethernet forms a major portion of the world's networks. Applications written using the sockets library have not been able to take advantage of the high performance provided by Gigabit Ethernet due to the traditional implementation of sockets on kernel based protocols.

In this chapter, we have discussed the design of a low-overhead substrate to support socket based applications on EMP. For short messages, this substrate delivers a latency of 28.5 $\mu$s for Datagram sockets and 37 $\mu$s for Data Streaming sockets compared to a latency of 28 $\mu$s achieved by raw EMP. Compared to the basic TCP, latency obtained by this substrate shows performance improvement up to 4 times. A peak bandwidth of over 840 Mbps is obtained by this substrate, compared to 550 Mbps achieved by the basic TCP, a performance improvement by a percentage of up to 53%. For the FTP and Apache Web server applications, compared to the basic TCP implementation, the new substrate shows performance improvement by a factor of 2 and 6, respectively. These results demonstrate that applications written using TCP can be directly run on Gigabit Ethernet-connected cluster with this substrate.

# CHAPTER 3

# IMPACT OF HIGH-PERFORMANCE SOCKETS ON DATA INTENSIVE APPLICATIONS

Quite a number of research projects in high-end computing focus on development of methods for solving challenging compute intensive applications in science, engineering and medicine. These applications are generally run in batch mode and can generate very large datasets. Advanced sensor technologies also enable acquisition of high resolution multi-dimensional datasets. As a result, there is an increasing interest in developing applications that interactively explore, synthesize and analyze large scientific datasets [49]. In this chapter, we refer to these applications as data intensive applications.

A challenging issue in supporting data intensive applications 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 [22, 32, 66, 70] 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 [22]. 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.

A number of such frameworks have been developed using the sockets interface. To support such applications on high performance user-level protocols without any changes to the application itself, researchers have come up with various high performance sockets implementations [16, 58, 74]. Applications written using kernel-based sockets layers are often developed keeping the communication performance of TCP/IP in mind. High performance substrates, on the other hand, have different performance characteristics compared to kernel-based sockets layers. This becomes a fundamental bottleneck in the performance such high performance substrates are able to deliver. However, changing some components of an application, such as the size of the data chunks that make up the dataset, allows the applications to take advantage of the performance characteristics of high performance substrates making them more scalable and adaptable.

In this chapter, we study the efficiency and limitations of such a high performance sockets implementation over the Virtual Interface Architecture (VIA), referred to here as $S$ocketVIA, in terms of performance and the flexibility it allows, in the context of a component framework designed to provide runtime support for data intensive applications, called DataCutter [22]. In particular, we investigate answers to the following questions:

- *Can high performance sockets allow the implementation of a scalable interactive data-intensive application with performance guarantees to the end user?*
- *Can high performance sockets improve the adaptability of data-intensive applications to heterogeneous environments?*

Our experimental results show that by reorganizing certain components of the applications, significant improvements in performance can be obtained. This leads to higher scalability of applications with performance guarantees. It also enables fine grained load balancing, thus making applications more adaptable to heterogeneous environments and varying resource availability.

## 3.1 Background

In this chapter, a brief overview of the Virtual Interface Architecture (VIA) [27, 3, 54], a widely used user level protocol is provided.

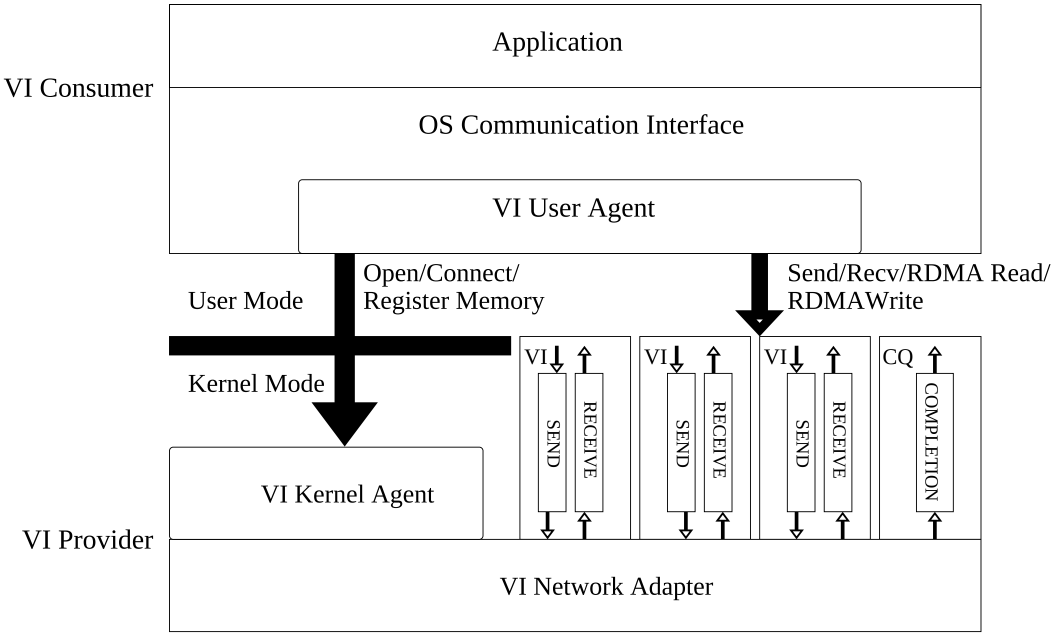

Figure 3.1: VI Architectural Model

### 3.1.1 Virtual Interface Architecture

The Virtual Interface Architecture (VIA) has been standardized as a low latency and high bandwidth user-level protocol for System Area Networks(SANs). A System Area Network interconnects various nodes of a distributed computer system.

The VIA architecture 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 path of the message. This is achieved by providing every consumer process a protected and directly accessible interface to the network named as a Virtual Interface(VI). Figure 3.1 illustrates the Virtual Interface Architecture model.

Each VI is a communication endpoint. Two VIs on different nodes can be connected to each other to form a logical bi-directional communication channel. An application can have multiple VIs. Each VI has a Work queue consisting of send and a receive Queue. A doorbell is also associated with each VI. Applications post requests to these queues in the form of VIA descriptors. The posting of the request is followed by ringing of the doorbell associated with the VI to inform the VI provider about the new request. Each VI can be associated with a completion queue (CQ). A completion queue can be associated with many VIs. Notification of the completed request on a VI can optionally be directed to the completion queue associated with it. Hence, an application can poll a single CQ instead of multiple work queues to check for completion of a request.

A VIA descriptor is a data structure which contains all the information needed by the VIA provider to process the request. Each VIA descriptor contains a Control Segment (CS), zero or more Data Segments (DS) and possibly an Address Segment (AS). When a request is completed, the Status field on the CS is marked complete. Applications can check the completion of their requests by verifying this field. On completion, these descriptors can be removed from the queues and reused for further requests.

The Data segment of the descriptor contains a user buffer virtual address. The descriptor gives necessary information including the data buffer address and length. VIA requires that the memory buffers used in the data transfer be registered. This allows the VI provider to pin down the virtual memory pages in physical memory and avoid their swapping, thus allowing the network interface to directly access them without the intervention of the operating system. For each contiguous region of memory registered, the application (VI consumer) gets an opaque handle. The registered memory can be referenced by the virtual address and the associated memory handle.

The VIA specifies two types of data transfer facilities: the traditional send and receive messaging model and the Remote Direct Memory Access (RDMA) model.

In the send and receive model, each send descriptor on the local node has to be matched with a receive descriptor on the remote node. Thus there is a one-to-one correspondence between every send and receive operation. Failure to post a receive descriptor on the remote node results in the message being dropped and if the connection is a reliable connection, it might even result in the breaking of the connection.

In the RDMA model, the initiator specifies both the virtual address of the local user buffer and that of the remote user buffer. In this model, a descriptor does not have to be posted on the receiver side corresponding to every message. The exception to this case is when the RDMA Write is used in conjunction with immediate data, a receive descriptor is consumed at the receiver end.

The VIA specification does not provide different primitives for Send and RDMA. It is the VIA descriptor that distinguishes between the Send and RDMA. The Send descriptor contains the CS and DS. In case of RDMA, the VI Send descriptor also contains the AS. In the AS, the user specifies the address of the buffer at the destination node and the memory handle associated with that registered destination buffer address.

There are two types of RDMA operations: RDMA Write and RDMA Read. In the RDMA Write operation, the initiator specifies both the virtual address of the locally registered source user buffer and that of the remote destination user buffer. In the RDMA Read operation, the initiator specifies the source of the data transfer at the remote and the destination of the data transfer within a locally registered contiguous memory location. In both cases, the initiator should know the remote address and should have the memory handle for that address beforehand. Also, VIA does not support scatter of data, hence the destination buffer in the case of RDMA Write and RDMA Read has to be contiguously registered buffer. The RDMA Write is a required feature of the VIA specification whereas the RDMA Read operation is optional. Hence, the work done in this thesis exploits only the RDMA Write feature of the VIA.

Since the introduction of VIA, many software and hardware implementations of VIA have become available. The Berkeley VIA [27], Firm VIA [19], M-VIA [3], Server Net VIA [50], GigaNet VIA [54] are among these implementations. In this chapter, we use GigaNet VIA, a hardware implementation of VIA for experimental evaluation.

## 3.2 Overview of Data Intensive Applications

As processing power and capacity of disks continue to increase, the potential for applications to create and store multi-gigabyte and multi-terabyte datasets is becoming more feasible. Increased understanding is achieved through running analysis and visualization codes on the stored data. For example, interactive visualization relies on our ability to gain insight from looking at a complex system. Thus, both data analysis and visual exploration of large datasets play an increasingly important role in many domains of scientific research. We refer here to applications that interactively query and analyze large scientific datasets as data-intensive applications.

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 in this chapter. 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 [8, 31]. 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 [21, 22]. 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 3.2 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.

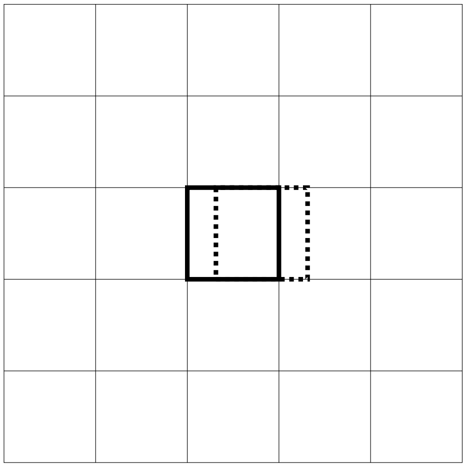

Figure 3.2: Partitioning of a complete image into blocks. A partial query (rectangle with dotted lines) requires only a part of a block

## 3.3 Performance Issues in Runtime Support for Data Intensive Applications

### 3.3.1 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.

### 3.3.2 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 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 substrates have other interesting features as demonstrated in Figures 3.3(a) and 3.3(b). Figure 3.3(a) shows that high performance substrates 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 substrates require a lower message size of U2 bytes. Using this information in Figure 3.3(b), we observe that high performance substrates result in lower message latency (from L1 to L2) at a message size of U1 bytes. We also observe that high performance substrates can use a message size of U2 bytes (from Figure 3.3(a)), hence further reducing the latency (from L2 to L3) and resulting in better performance.

### 3.3.3 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

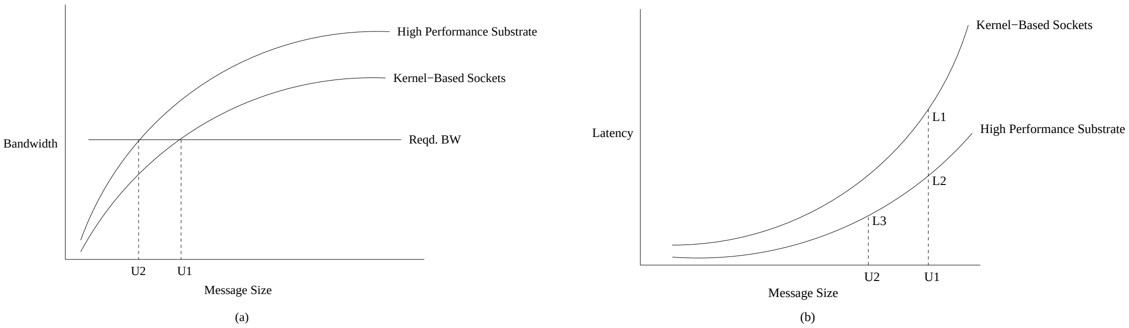

Figure 3.3: (a) High Performance Substrates achieve a given bandwidth for a lower message size compared to Kernel-Based Sockets, (b) High Performance Substrates can achieve a direct and indirect improvement in the performance based on the application characteristics

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 (see Section 8.4.1) 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.

## 3.4 Software Infrastructure used for Evaluation

In terms of application development and runtime support, component-based frameworks [22, 32, 66, 70] can provide an effective environment to address performance issues in data intensive applications. Components can be placed onto different computational resources, and task and data-parallelism can be achieved by pipelined execution of multiple copies of these components. Therefore, we use a component-based infrastructure, called DataCutter, which is designed to support data intensive applications in distributed environments. We also employ a high performance sockets interface, referred to here as SocketVIA, designed for applications written using TCP/IP to take advantage of the performance capabilities of VIA.

### 3.4.1 DataCutter

In this section we briefly describe the DataCutter framework. DataCutter implements a filter-stream programming model for developing data-intensive applications. In this model, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. The interface for a *filter*, consists of three functions: (1) an initialization function (*init*), in which any required resources such as memory for data structures are allocated and initialized, (2) a processing function (*process*), in which user-defined operations are applied on data elements, and (3) a finalization function (*finalize*), in which the resources allocated in *init* are released.

Filters are connected via *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only. We define a *data buffer* as an array of data elements transferred from one filter to another. The original implementation of the logical stream delivers data in fixed size buffers, and uses TCP for point-to-point stream communication.

The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. When a filter group is instantiated to process an application query, the runtime system establishes socket connections between filters placed on different hosts before starting the execution of the application query. Filters placed on the same host execute as separate threads. An application query is handled as a *unit of work* (UOW) by the filter group. An example is a visualization of a dataset from a viewing angle. The processing of a UOW can be done in a pipelined fashion; different filters can work on different data elements simultaneously. Processing of a UOW starts when the filtering service calls the filter **init** function, which is where any required resources such as memory can be pre-allocated. Next the **process** function is called to read from any input streams, work on the data buffers received, and write to any output streams. A special marker is sent by the runtime system after the last buffer to mark the end for the current UOW (see Figure 8.6(a)). The **finalize** function is called after all processing is completed for the current UOW, to allow release of allocated resources such as scratch space. The interface functions *may* be called again to process another UOW.

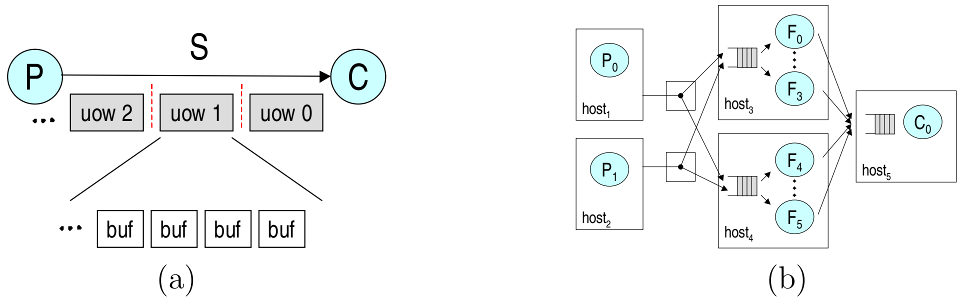

Figure 3.4: DataCutter stream abstraction and support for copies. (a) Data buffers and end-of-work markers on a stream. (b) P,F,C filter group instantiated using transparent copies.

The programming model provides several abstractions to facilitate performance optimizations. A *transparent filter copy* is a copy of a filter in a filter group (see Figure 8.6(b)). The filter copy is transparent in the sense that it shares the same *logical* input and output streams of the original filter. A transparent copy of a filter can be made if the semantics of the filter group are not affected. That is, the output of a unit of work should be the same, regardless of the number of transparent copies. The transparent copies enable data-parallelism for execution of a single query, while multiple filter groups allow concurrency among multiple queries.

The filter runtime system maintains the illusion of a single logical point-to-point stream for communication between a logical producer filter and a logical consumer filter. It is responsible for scheduling elements (or buffers) in a data stream among the transparent copies of a filter. For example, in Figure 8.6(b), if copy $P_1$ issues a buffer write operation to the logical stream that connects filter $P$ to filter $F$, the buffer can be sent to the copies on $host_3$ or $host_4$. For distribution between transparent copies, the runtime system supports a Round-Robin (RR) mechanism and a Demand Driven (DD) mechanism based on the buffer consumption rate. DD aims at sending buffers to the filter that would process them fastest. When a consumer filter starts processing of a buffer received from a producer filter, it sends an acknowledgment message to the producer filter to indicate that the buffer is being processed. A producer filter chooses the consumer filter with the minimum number of unacknowledged buffers to send a data buffer to, thus achieving a better balancing of the load.

### 3.4.2 SocketVIA

Inspite of the development of low-latency and high-bandwidth user-level protocols, a large number of applications have been developed previously on kernel-based protocols such as TCP and UDP. Some of these applications took years to develop. Trying to rewrite these applications on user-level protocols is highly time-consuming and impractical. On the other hand, the sockets interface is widely used by a variety of applications written on protocols such as TCP and UDP.

The cLAN network is a hardware implementation of the Virtual Interface Architecture (VIA). There are two typical socket implementations on the cLAN network. One is to keep the legacy socket, TCP/UDP and IP layers unchanged, while one additional layer is introduced to bridge the IP layer and the kernel level VI layer. The LANE (LAN Emulator) implementation of the socket layer is such an implementation using an IP-to-VI layer [54]. Due to the system call overhead (including the kernel-context switch, flushing of the cache, flushing of the TLB, bottom-half handlers, etc) and multiple copies involved in this implementation, applications using LANE have not been able to take complete advantage of the high performance provided by the underlying network. Another type of socket implementation on the cLAN network is to provide socket interface using a user-level library based on the user-level VIA primitives. Our implementation falls into this category. We refer to our sockets layer as *SocketVIA* in the rest of this dissertation. Since the implementation of SocketVIA is not the main focus of this chapter, we just present the micro-benchmark results for our sockets layer in the next section. For other details related to the design and implementation of SocketVIA, we refer the reader to [18].

## 3.5 Performance Evaluation

In this chapter, we present two groups of results. First, we look at the peak performance delivered by SocketVIA in the form of latency and bandwidth micro-benchmarks. Second, we examine the direct and indirect impacts on the performance delivered by the substrate on applications implemented using DataCutter in order to evaluate both latency and bandwidth aspects in a controlled way. 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.

### 3.5.1 Micro-Benchmarks

Figure 3.5(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 9.5$\mu$s, 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 3.5(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%.

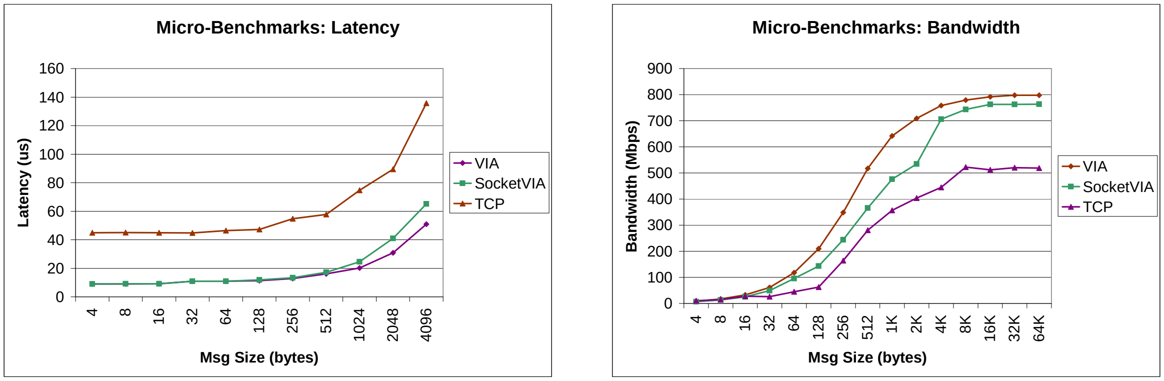

Figure 3.5: Micro-Benchmarks (a) Latency, (b) Bandwidth

### 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 3.6). 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.

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 3.2). 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.

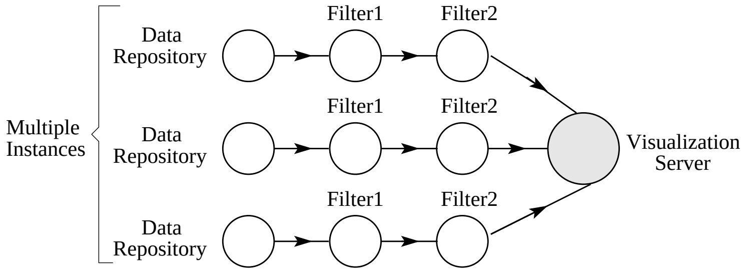

Figure 3.6: Guarantee Based Performance Evaluation: Experimental Setup

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 3.7). Performance impact with such heterogeneity is also studied in this section.

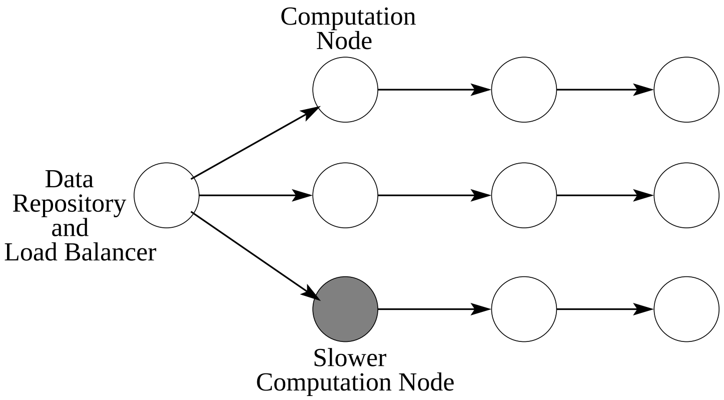

Figure 3.7: Effect of Heterogeneous Clusters: Experimental Setup

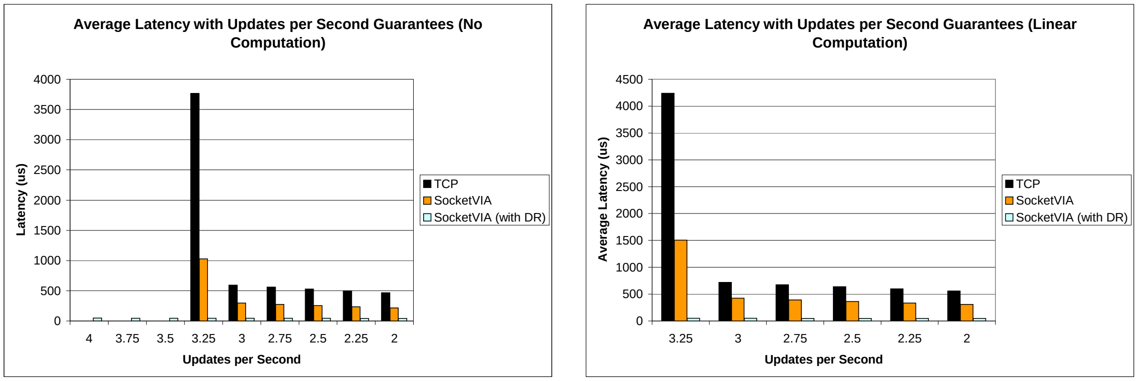

Figure 3.8: Effect of High Performance Sockets on Average Latency with guarantees on Updates per Second for (a) No Computation Cost and (b) Linear Computation Cost

### 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 3.8(a) and 3.8(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 3.8(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 3.8(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 [31], 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 chapter.

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

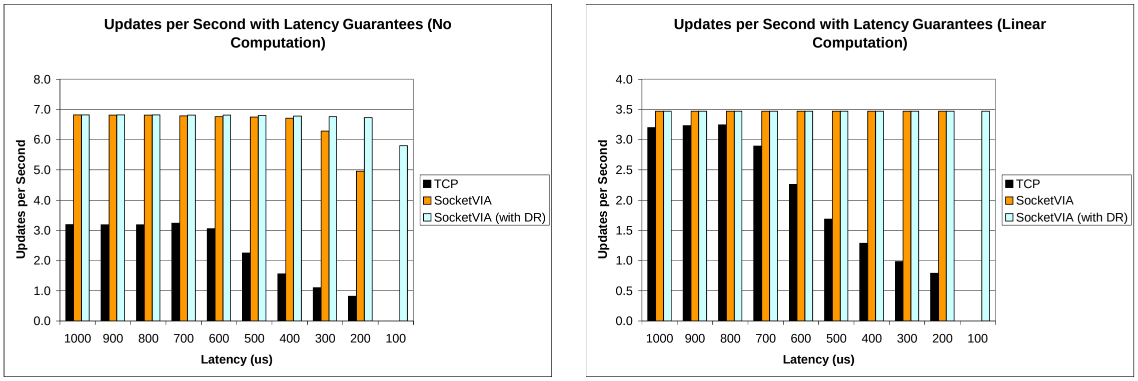

Figure 3.9: Effect of High Performance Sockets on Updates per Second with Latency Guarantees for (a) No Computation Cost and (b) Linear Computation Cost

a particular latency is targeted for a partial update query. Figures 3.9(a) and 3.9(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 3.9(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 3.9(b). With no computation cost, as the latency constraint becomes as low as 100$\mu$s, 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.

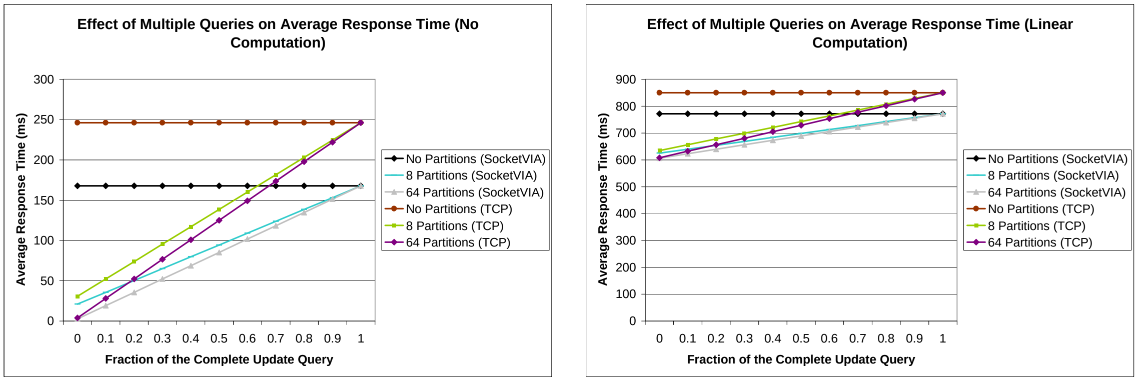

Figure 3.10: Effect of High Performance Sockets on the Average Response Time of Queries for (a) No Computation Cost and (b) Linear Computation Cost

**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 3.10(a) and 3.10(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 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.

### 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 3.7). 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.

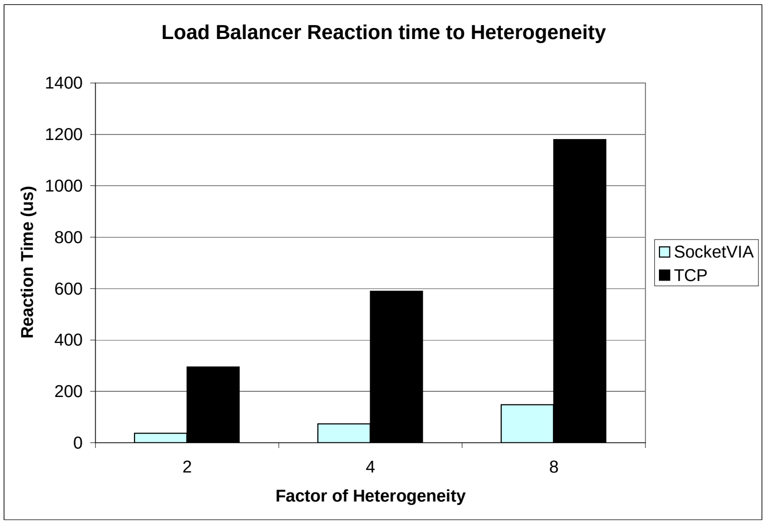

Figure 3.11: Effect of Heterogeneity in Processing Speed on Load Balancing using the Round-Robin Scheduling Scheme

Figure 3.11 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.

**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 3.12 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 3.12, 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.

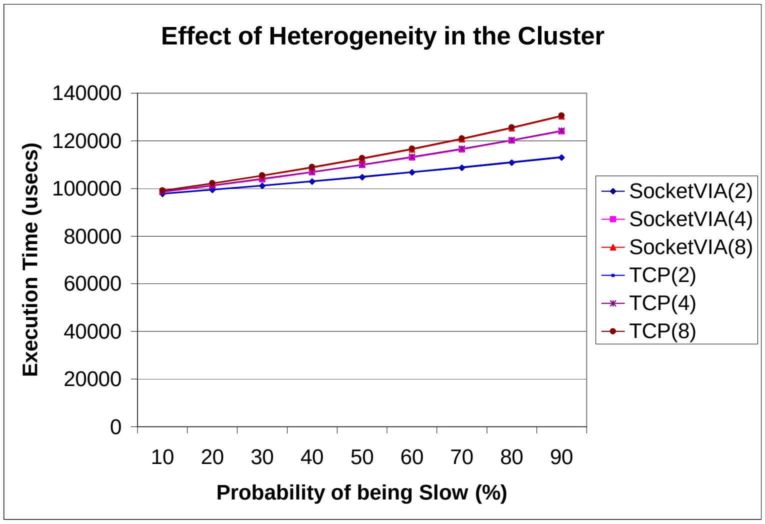

Figure 3.12: Effect of Heterogeneity in Processing Speed on Load Balancing using the Demand-Driven Scheduling Scheme

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.

## 3.6 Summary

Together with a pure performance requirements, data intensive applications have other requirements such as guarantees in performance, scalability with these guarantees and adaptability to heterogeneous networks. Typically such applications are written using the kernel-based sockets interface over TCP/IP. To allow such applications take advantage of the high performance protocols, researchers have come up with a number of techniques including High Performance Sockets layers over User-level protocols such as Virtual Interface Architecture and the emerging InfiniBand Architecture. However, these sockets layers are fundamentally limited by the fact that the applications using them had been written keeping the communication performance of TCP/IP in mind.

In this chapter, we have studied the capabilities and limitations of such a substrate, termed *S*ocketVIA, in performance, with respect to a component framework designed to provide runtime support for data intensive applications, termed as *DataCutter*. The experimental results show that by reorganizing certain components of the applications, we can make significant improvements in the performance, leading to a higher scalability of the applications with performance guarantees and fine grained load balancing making them more adaptable to heterogeneous networks. The experimental results also show that the different performance characteristics of SocketVIA allow a more efficient partitioning of data at the source nodes, thus improving the performance up to an order of magnitude in some cases. This shows that together with high performance, low-overhead substrates provide the ability to applications to simultaneously meet quality requirements along multiple dimensions. These results have strong implications on designing, developing, and implementing next generation data intensive applications on modern clusters.

# CHAPTER 4

# SOCKETS DIRECT PROTOCOL OVER INFINIBAND IN CLUSTERS: IS IT BENEFICIAL?

Sockets Direct Protocol (SDP) [5] is an industry standard specification for high performance sockets implementations over InfiniBand (IB) and the Internet Wide Area RDMA Protocol (iWARP) [71]. SDP was proposed along the same lines as the user-level sockets layers; to allow a smooth transition to deploy existing sockets based applications on to clusters connected with InfiniBand while sustaining most of the performance provided by the base network.

In this chapter, we study the benefits and limitations of an implementation of SDP. We first analyze the performance of SDP based on a detailed suite of micro-benchmarks. Next, we evaluate it on two real application domains: (a) a Multi-tier Data-Center and (b) a Parallel Virtual File System (PVFS). Our micro-benchmark results show that SDP is able to provide up to 2.7 times better bandwidth as compared to the native TCP/IP sockets implementation over InfiniBand (IPoIB) and significantly better latency for large message sizes. Our experimental results also show that SDP is able to achieve a considerably high performance (improvement of up to a factor of 2.4) compared to the native sockets implementation in the PVFS environment. In the data-center environment, SDP outperforms IPoIB for large file transfers in-spite of currently being limited by a high connection setup time. However, as the InfiniBand software and hardware products are rapidly maturing, we expect this limitation to be overcome soon. Based on this, we have shown that the projected performance for SDP, without the connection setup time, can outperform IPoIB for small message transfers as well.

## 4.1 Background

In this section we provide a brief background about InfiniBand and the Sockets Direct Protocol (SDP).

### 4.1.1 InfiniBand Architecture

The InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. In a typical IBA cluster, switched serial links connect the processing nodes and the I/O nodes. The compute nodes are connected to the IBA fabric by means of Host Channel Adapters (HCAs). IBA defines a semantic interface called as Verbs for the consumer applications to communicate with the HCAs. VAPI is one such interface developed by Mellanox Technologies.

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. This is achieved by providing the consumer applications direct and protected access to the HCA. The specification for Verbs includes a queue-based interface, known as a Queue Pair (QP), to issue requests to the HCA. Figure 4.1 illustrates the InfiniBand Architecture model.

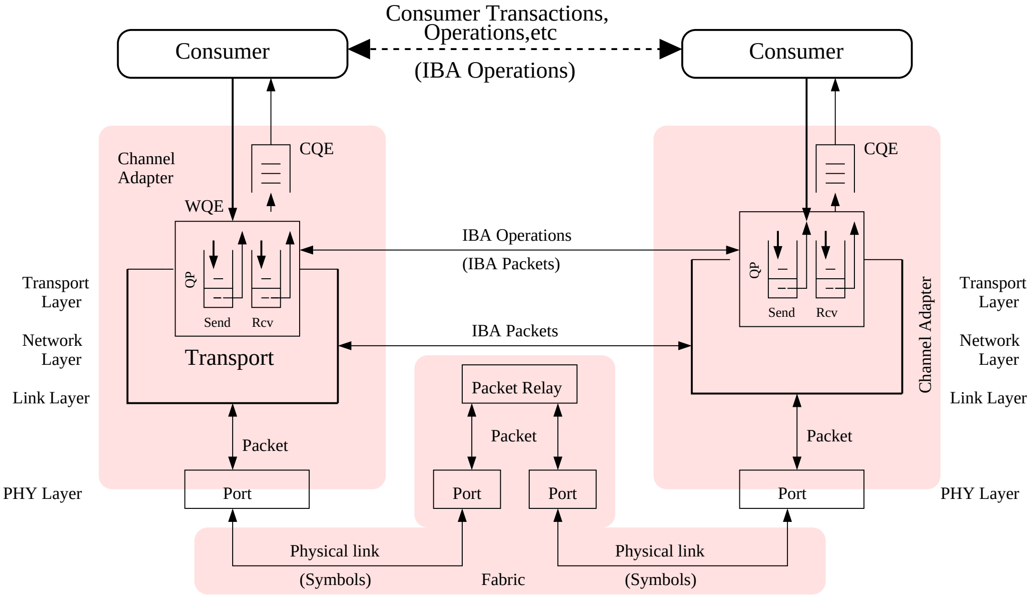

Figure 4.1: InfiniBand Architecture (Courtesy InfiniBand Specifications)

Each Queue Pair is a communication endpoint. A Queue Pair (QP) consists of the send queue and the 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). The application can either choose the request to be a Signaled (SG) request or an Un-Signaled request (USG). When the HCA completes the processing of a signaled request, it places an entry called as the Completion Queue Entry (CQE) in the Completion Queue (CQ). 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 un-signaled requests, no kind of 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 supports two types of communication semantics: channel semantics (send-receive communication model) and memory semantics (RDMA communication model).

In channel semantics, every send request has a corresponding receive request at the remote end. Thus there is one-to-one correspondence between every send and receive operation. Failure to post a receive 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.

Most 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 which has to be specified for RDMA operations. There are two kinds of RDMA operations: RDMA Write and RDMA Read. In an RDMA write operation, the initiator directly writes data into the remote node's user buffer. Similarly, in an RDMA Read operation, the initiator reads data from the remote node's user buffer.

In addition to RDMA, the reliable communication classes also optionally atomic operations directly against the memory at the end node. Atomic operations are posted as descriptors at the sender side as in any other type of communication. However, the operation is completely handled by the NIC and involves very little host intervention and resource consumption.

The atomic operations supported are Fetch-and-Add and Compare-and-Swap, both on 64-bit data. The Fetch and Add operation performs an atomic addition at the remote end. The Compare and Swap is use to compare two 64-bit values and swap the remote value with the data provided if the comparison succeeds.

Atomics are effectively a variation on RDMA: a combined write and read RDMA, carrying the data involved as immediate data. Two different levels of atomicity are optionally supported: atomic with respect to other operations on a target CA; and atomic with respect to all memory operation of the target host and all CAs on that host.

### 4.1.2 Sockets Direct Protocol

Sockets Direct Protocol (SDP) is a protocol defined by the Software Working Group (SWG) of the InfiniBand Trade Association [10]. The design of SDP is mainly based on two architectural goals: (a) Maintain traditional sockets SOCK_STREAM semantics as commonly implemented over TCP/IP and (b) Support for byte-streaming over a message passing protocol, including kernel bypass data transfers and zero-copy data transfers. Figure 4.2 illustrates the SDP architecture.

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 (using a buffer copy) or directly between ULP buffers

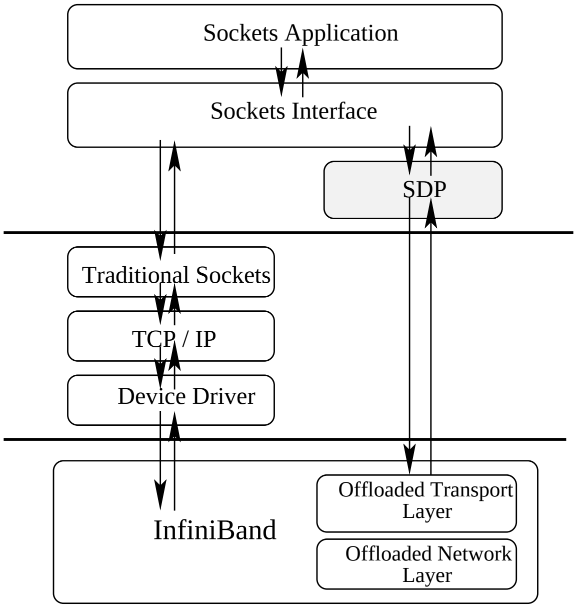

Figure 4.2: Sockets Direct Protocol

(zero copy). A mix of InfiniBand Send and RDMA mechanisms are used to transfer ULP data.

SDP specifications also specify two additional control messages known as “Buffer Availability Notification” messages.

**Sink Avail Message:** If the data sink has already posted a receive buffer and the data source has not sent the data message yet, the data sink does the following steps: (1) Registers the receive user-buffer (for large message reads) and (2) Sends a “Sink Avail” message containing the receive buffer handle to the source. The Data Source on a data transmit call, uses this receive buffer handle to directly RDMA write the data into the receive buffer.

**Source Avail Message:** If the data source has already posted a send buffer and the available SDP window is not large enough to contain the buffer, it does the following two steps: (1) Registers the transmit user-buffer (for large message sends) and (2) Sends a "Source Avail" message containing the transmit buffer handle to the data sink. The Data Sink on a data receive call, uses this transmit buffer handle to directly RDMA read the data into the receive buffer.

The current implementation of SDP follows most of the specifications provided above. There are two major deviations from the specifications in this implementation. First, it does not support "Source Avail" and "Sink Avail" messages. Second, it does not support a zero-copy data transfer between user buffers. All data transfer is done through the buffer copy mechanism. This limitation can also be considered as part of the previous ("Source Avail"/"Sink Avail") limitation, since they are always used together.

## 4.2 Software Infrastructure

We have carried out the evaluation of SDP on two different software infrastructures: Multi-Tier Data Center environment and the Parallel Virtual File System (PVFS). In this section, we discuss each of these in more detail.

### 4.2.1 Multi-Tier Data Center environment

A typical Multi-tier Data-center has as its first tier, a cluster of nodes known as the edge nodes. These nodes can be thought of as switches (up to the 7th layer) providing load balancing, security, caching etc. The main purpose of this tier is to help increase the performance of the inner tiers. The next tier usually contains the web-servers and application servers. These nodes apart from serving static content, can fetch dynamic data from other sources and serve that data in presentable form. The last tier of the Data-Center is the database tier. It is used to store persistent data. This tier is usually I/O intensive.

A request from a client is received by the edge servers. If this request can be serviced from the cache, it is. Otherwise, it is forwarded to the Web/Application servers. Static requests are serviced by the web servers by just returning the requested file to the client via the edge server. This content may be cached at the edge server so that subsequent requests to the same static content may be served from the cache. The Application tier nodes handle the Dynamic content. The type of applications this tier includes range from mail servers to directory services to ERP software. Any request that needs a value to be computed, searched, analyzed or stored uses this tier. The back end database servers are responsible for storing data persistently and responding to queries. These nodes are connected to persistent storage systems. Queries to the database systems can be anything ranging from a simple seek of required data to performing joins, aggregation and select operations on the data. A more detailed explanation of the typical data-center environment can be obtained in [14].

### 4.2.2 Parallel Virtual File System (PVFS)

Parallel Virtual File System (PVFS) [30] is one of the leading parallel file systems for Linux cluster systems today. It was designed to meet the increasing I/O demands of parallel applications in cluster systems. Typically, a number of nodes in the cluster system are configured as I/O servers and one of them (either an I/O server or an different node) as a metadata manager. It is possible for a node to host computations while serving as an I/O node.

PVFS achieves high performance by striping files across a set of I/O server nodes allowing parallel accesses to the data. It uses the native file system on the I/O servers to store individual file stripes. An I/O daemon runs on each I/O node and services requests from the compute nodes, in particular the read and write requests. Thus, data is transferred directly between the I/O servers and the compute nodes. A manager daemon runs on a metadata manager node. It handles metadata operations involving file permissions, truncation, file stripe characteristics, and so on. Metadata is also stored on the local file system. The metadata manager provides a cluster-wide consistent name space to applications. In PVFS, the metadata manager does not participate in read/write operations. PVFS supports a set of feature-rich interfaces, including support for both contiguous and non-contiguous accesses to both memory and files [35]. PVFS can be used with multiple APIs: a native API, the UNIX/POSIX API, MPI-IO [79], and an array I/O interface called Multi-Dimensional Block Interface (MDBI). The presence of multiple popular interfaces contributes to the wide success of PVFS in the industry.

## 4.3 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:

**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 DualPort 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.

### 4.3.1 Latency and Bandwidth

Figure 4.3a 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 28$\mu$s for 2 byte messages compared to a 30$\mu$s achieved by IPoIB and 7$\mu$s and 5.5$\mu$s 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 4.3b 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.

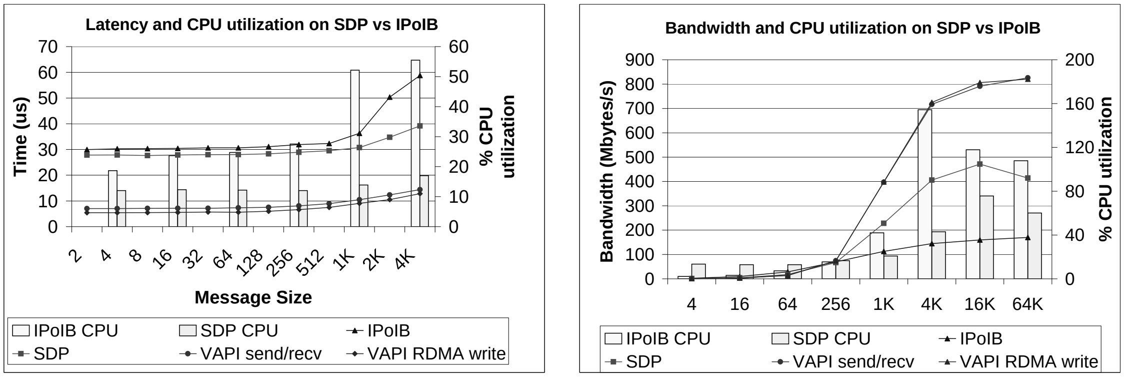

Figure 4.3: Micro-Benchmarks: (a) Latency, (b) Bandwidth

### 4.3.2 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 4.4a. 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.

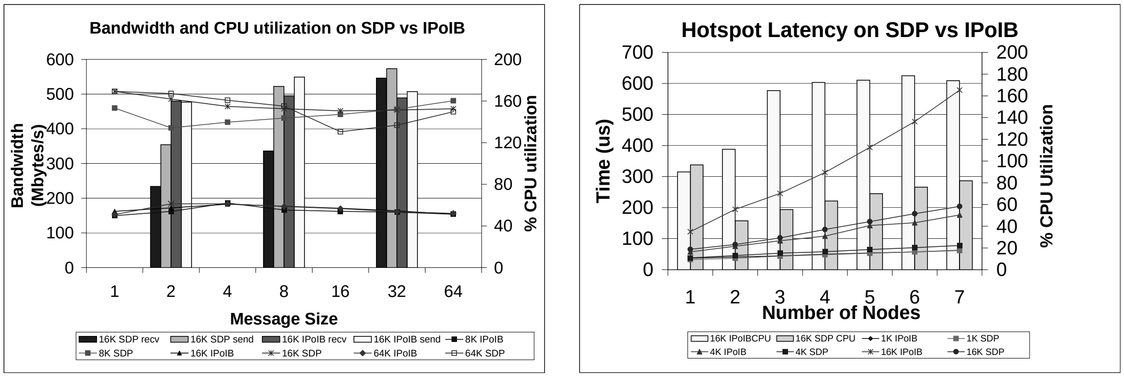

Figure 4.4: (a) Multi-Stream Bandwidth, (b) Hot-Spot Latency

### 4.3.3 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 4.4b 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 a 138$\mu$s compared to 456$\mu$s 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.

### 4.3.4 Fan-in and Fan-out

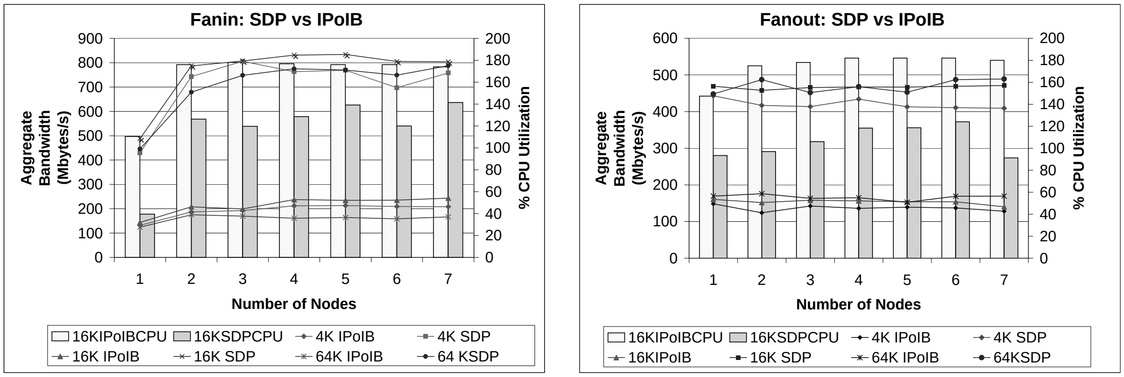

Figure 4.5: Micro-Benchmarks: (a) Fan-in, (b) Fan-out

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 4.5a and 4.5b 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.

## 4.4 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 (described in Section 4.3) for the data-center tiers. For the client nodes, we used the nodes in Cluster 2 for most experiments. We will notify the readers at appropriate points in this chapter when other nodes are used as clients.

### 4.4.1 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 20,000 request subset of the world-cup trace [9] 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 research 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 [20] for generating our dynamic request access pattern (further details about the database used can be obtained in [14]).

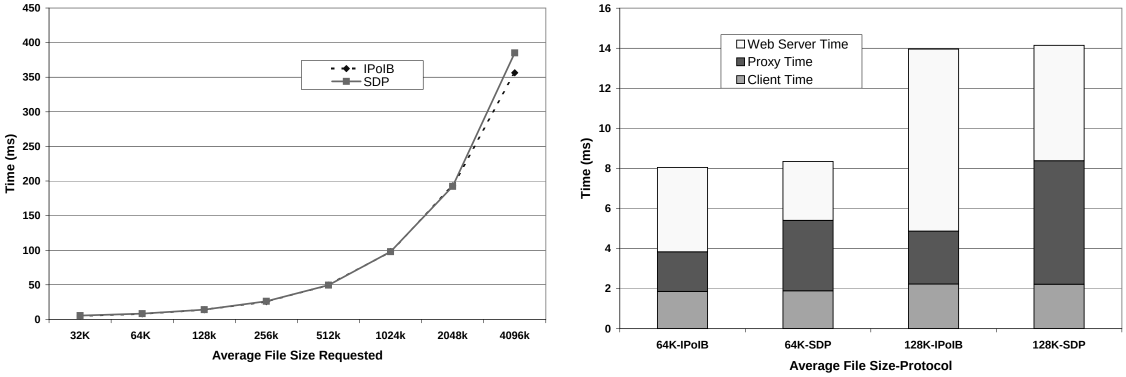

Figure 4.6: Client over Fast Ethernet: (a) Response Time and (b) Response Time Split-up

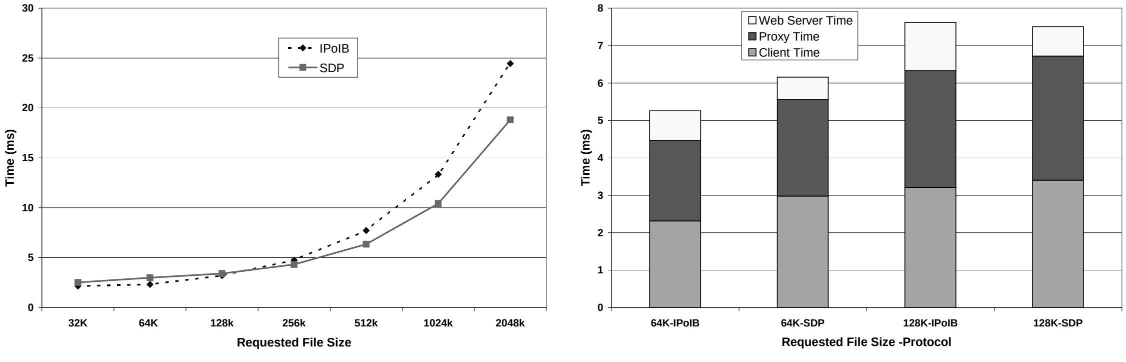

Figure 4.7: Client over IPoIB: (a) Response Time and (b) Response Time Split-up

### 4.4.2 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 4.6a 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 4.6b 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.

From the break-up graph (Figure 4.6b), 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 4.7a 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 4.7b shows the splitup 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.

Figures 4.9a and 4.9b show a comprehensive breakup of the time spent at the proxy over IPoIB and SDP respectively. A comparison of this splitup 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 500$\mu$s higher than IPoIB, makes the data-transfer related benefits of SDP imperceivable for low file size transfers.

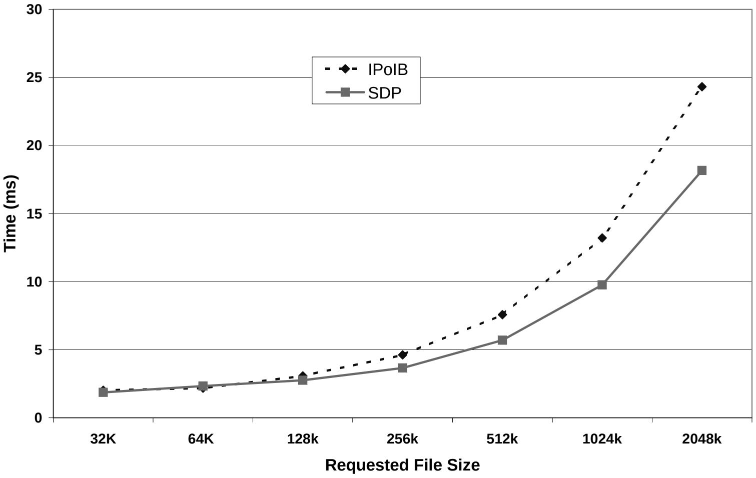

Figure 4.8: Fast Client Response Time without Connection Time

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 4.8 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 [14].

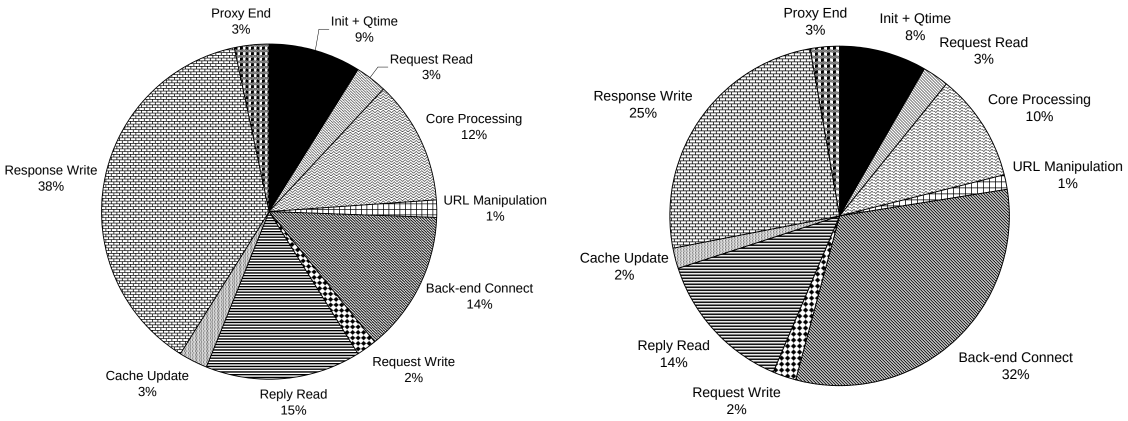

Figure 4.9: Proxy Split-up times: (a) IPoIB, (b) SDP

## 4.5 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 [30]. We also compare the performance of PVFS on the above two protocols with the performance of our previous implementation of PVFS over InfiniBand [82]. All experiments in this section have been performed on Cluster 1 (mentioned in Section 4.3).

### 4.5.1 Evaluation Methodology

There is a large difference between the bandwidth realized by the InfiniBand network (Figure 4.3b) 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 [30], 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.

### 4.5.2 PVFS Concurrent Read and Write on ramfs

Figure 4.10 shows the read performance with the original implementation of PVFS over IPoIB and SDP and an implementation of PVFS over VAPI [82], 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 4.11 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.

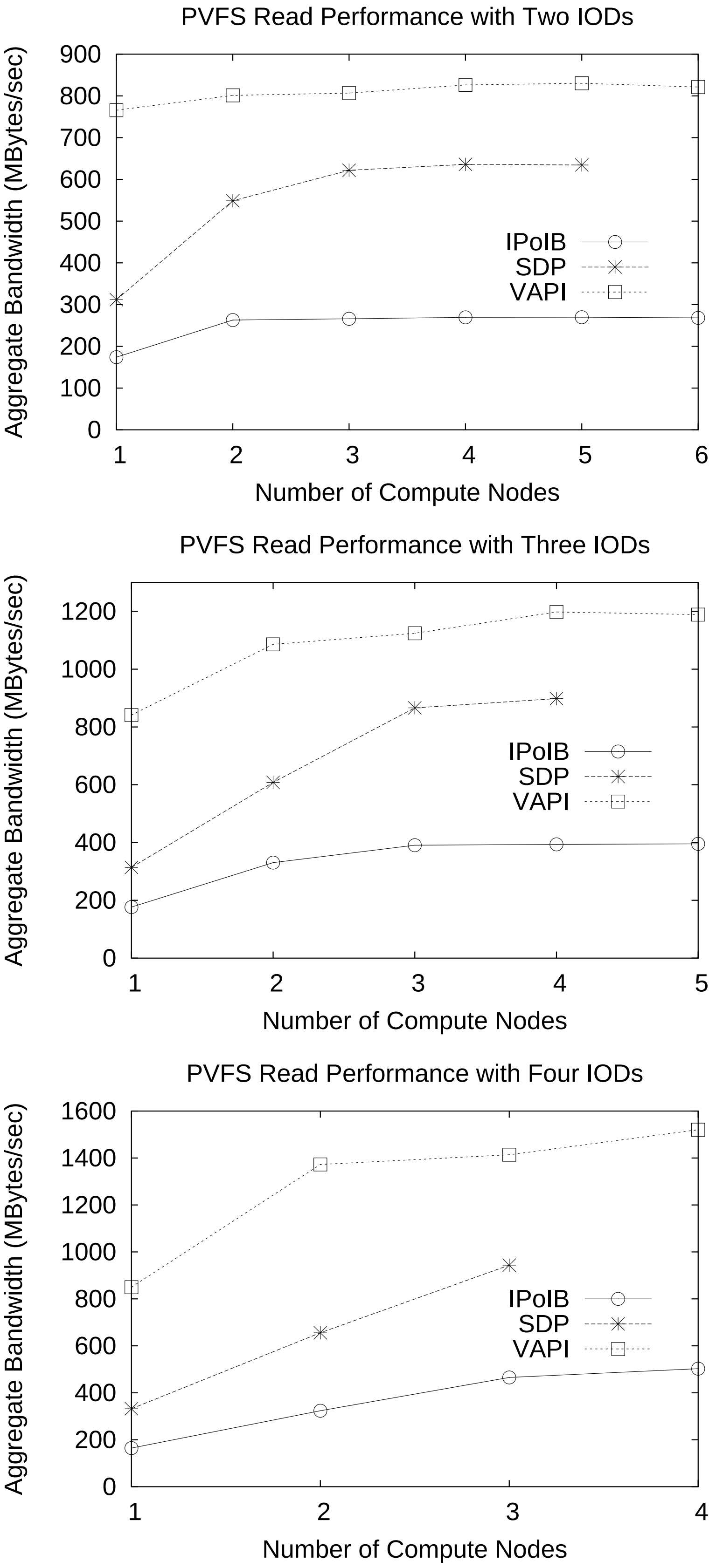

Figure 4.10: PVFS Read Performance Comparison

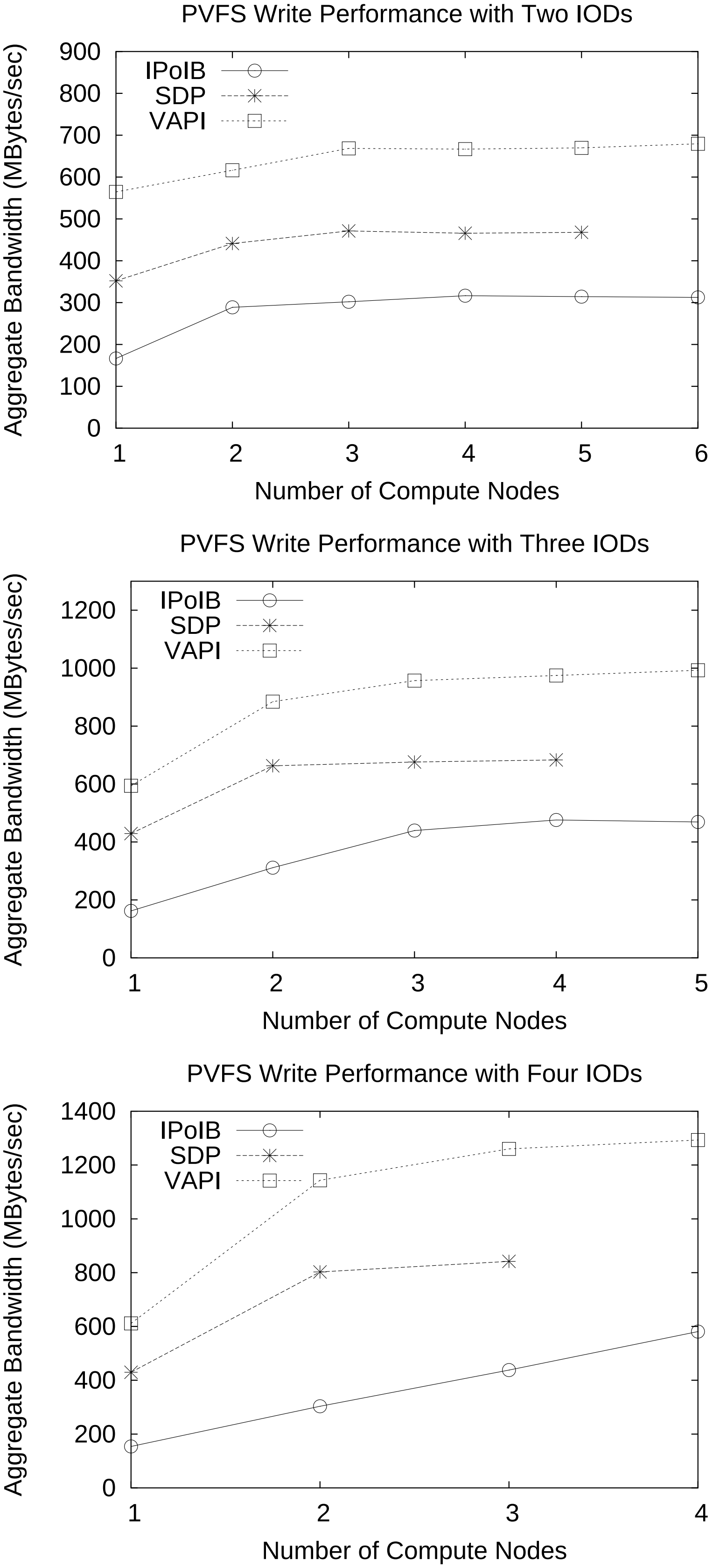

Figure 4.11: PVFS Write Performance Comparison

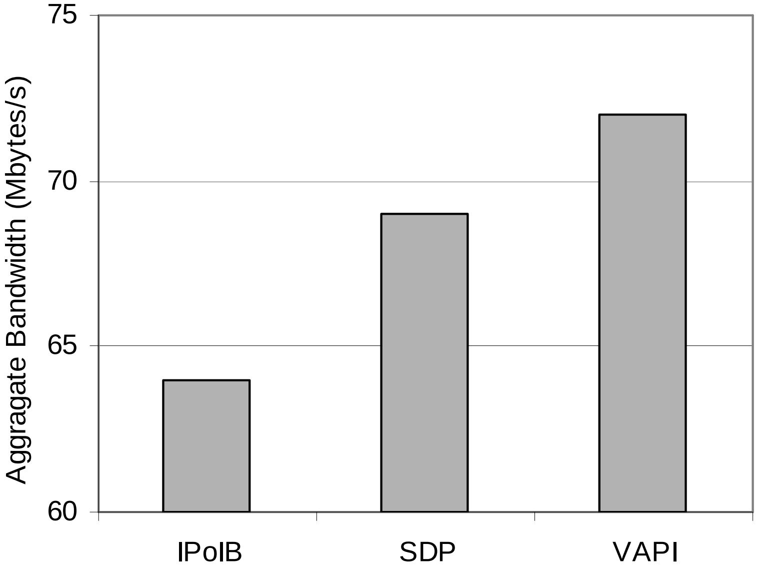

Figure 4.12: Performance of PVFS Write with Sync on ext3fs

### 4.5.3 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 4.12 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 4.3b 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.

## 4.6 Summary

The Sockets Direct Protocol had been proposed recently in order to enable traditional sockets based applications to take advantage of the enhanced features provided by the InfiniBand Architecture. In this chapter, we have studied the benefits and limitations of an implementation of SDP. We first analyzed the performance of SDP based on a detailed suite of micro-benchmarks. Next, we evaluated it on two real application domains: (1) A multi-tier Data-Center environment and (2) A Parallel Virtual File System (PVFS). Our micro-benchmark results show that SDP is able to provide up to 2.7 times better bandwidth as compared to the native sockets implementation over InfiniBand (IPoIB) and significantly better latency for large message sizes. Our results also show that SDP is able to achieve a considerably higher performance (improvement of up to 2.4 times) as compared to IPoIB in the PVFS environment. In the data-center environment, SDP outperforms IPoIB for large file transfers in spite of currently being limited by a high connection setup time. However, as the InfiniBand software and hardware products are rapidly maturing, we expect this limitation to be overcome rapidly. Based on this, we have shown that the projected performance for SDP can perform significantly better than IPoIB in all cases. These results provide profound insights into the efficiencies and bottlenecks associated with High Performance socket layers for 10-Gigabit networks. These insights have strong implications on the design and implementation of the next generation high performance applications.

# CHAPTER 5

# ASYNCHRONOUS ZERO-COPY COMMUNICATION FOR SYNCHRONOUS SOCKETS IN SDP OVER INFINIBAND

The SDP standard supports two kinds of sockets semantics, viz., Synchronous sockets (e.g., used by Linux, BSD, Windows) and Asynchronous sockets (e.g., used by Windows, upcoming support in Linux). In the synchronous sockets interface, the application has to *block* for every data transfer operation, i.e., if an application wants to send a 1 MB message, it has to wait till either the data is transferred to the remote node or is copied to a local communication buffer and scheduled for communication. In the asynchronous sockets interface, on the other hand, the application can *initiate* a data transfer and check whether the transfer is complete at a later time; thus providing a better overlap of the communication with the other on-going computation in the application. Due to the inherent benefits of asynchronous sockets, the SDP standard allows several intelligent approaches such as *source-avail and sink-avail based zero-copy* for these sockets. However, most of these approaches that work well for the asynchronous sockets interface are not as beneficial for the synchronous sockets interface. Added to this is the fact that the synchronous sockets interface is the one used by most sockets applications today due to its portability, ease of use and support on a wider set of platforms. Thus, a mechanism by which the approaches proposed for asynchronous sockets can be used for synchronous sockets is highly desirable.

In this chapter, we propose one such mechanism, termed as *AZ-SDP (Asynchronous Zero-Copy SDP)* which allows the approaches proposed for asynchronous sockets to be used for synchronous sockets while maintaining the synchronous sockets semantics. The basic idea of this mechanism is to protect application buffers from memory access during a data transfer event and carry out communication asynchronously. Once the data transfer is completed, the protection is removed and the application is allowed to touch the buffer again. It is to be noted that this entire scheme is completely transparent to the end application. We present our detailed design in this chapter and evaluate the stack with an extensive set of micro-benchmarks. The experimental results demonstrate that our approach can provide an improvement of close to 35% for medium-message uni-directional throughput and up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks.

## 5.1 Related Work

The concept of high performance sockets (such as SDP) has existed for quite some time. Several researchers, including ourselves, have performed significant amount of research on such implementations over various networks. Shah, et. al., from Intel, were the first to demonstrate such an implementation for VIA over the GigaNet cLAN network [74]. This was soon followed by other implementations over VIA [58, 18] as well as other networks such as Myrinet [63] and Gigabit Ethernet [16].

There has also been some amount of previous research for the high performance sockets implementations over IBA, i.e., SDP. Balaji et. al., were the first to show the benefits of SDP over IBA in [14] using a buffer copy based implementation of SDP. Goldenberg et. al., recently proposed a zero-copy implementation of SDP using a restricted version of the *source-avail* scheme [48, 47]. In particular, the scheme allows zero-copy communication by restricting the number of outstanding data communication requests on the network to just one. This, however, significantly affects the performance achieved by the zero-copy communication. Our design, on the other hand, carries out zero-copy communication while not being restricted to just one communication request, thus allowing for a significant improvement in the performance.

To optimize the TCP/IP and UDP/IP protocol stacks itself, many researchers have suggested several zero-copy schemes [55, 84, 34, 36]. However, most of these approaches are for asynchronous sockets and all of them require modifications to the kernel and even the NIC firmware in some cases. In addition, these approaches still suffer from the heavy packet processing overheads of TCP/IP and UDP/IP. On the other hand, our work benefits the more widely used synchronous sockets interface, it does not require any kernel or firmware modifications at all and can achieve very low packet processing overhead (due to the thin native protocol layers of the high-speed interconnects).

In summary, AZ-SDP is a novel and unique design for high performance sockets over IBA.

## 5.2 Design and Implementation Issues

As described in the previous chapters, to achieve zero-copy communication, *buffer availability notification* messages need to be implemented. In this chapter, we focus on a design that uses *source-avail* messages to implement zero-copy communication. In this section, we detail our mechanism to take advantage of asynchronous communication for synchronous zero-copy sockets.

### 5.2.1 Application Transparent Asynchronism

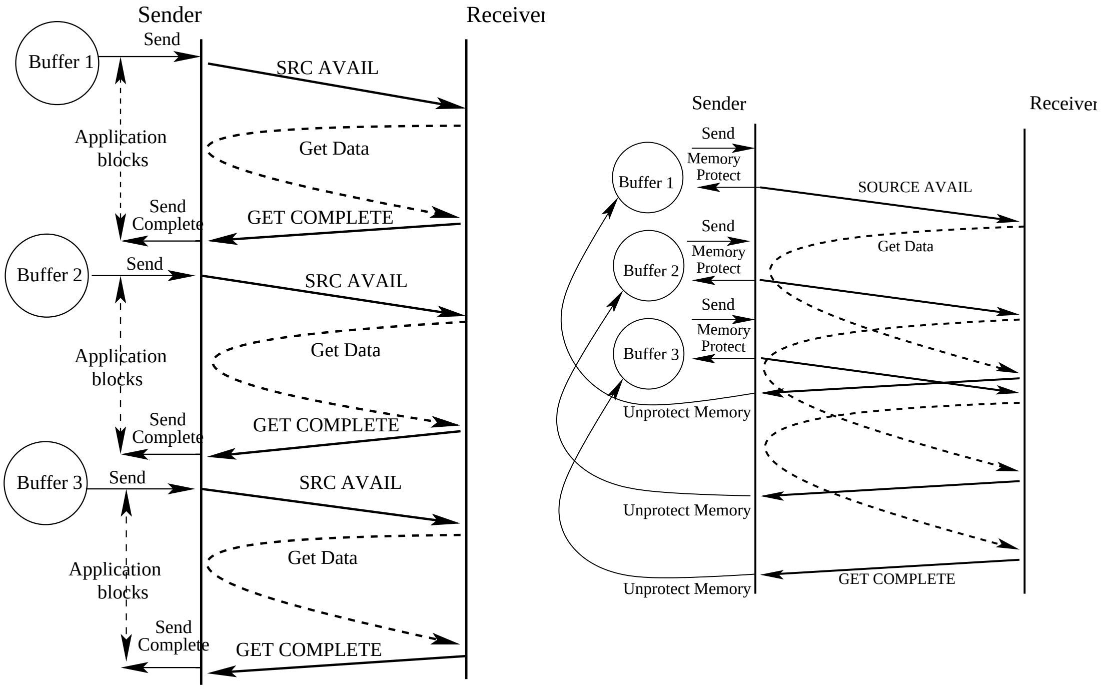

Figure 5.1: (a) Synchronous Zero-copy SDP (ZSDP) and (b) Asynchronous Zero-copy SDP (AZ-SDP)

The main idea of asynchronism is to avoid blocking the application while waiting for the communication to be completed, i.e, as soon as the data transmission is initiated, the control is returned to the application. With the asynchronous sockets interface, the application is provided with additional socket calls through which it can initiate data transfer in one call and wait for its completion in another. In the synchronous sockets interface, however, there are no such separate calls; there is just one call which initiates the data transfer *and* waits for its completion. Thus, the application cannot initiate multiple communications requests at the same time. Further, the semantics of synchronous sockets assumes that when the control is returned from the communication call, the buffer is free to be used (e.g., read from or write to). Thus, returning from a synchronous call asynchronously means that the application can assume that the data has been sent or received and try to write or read from the buffer irrespective of the completion of the operation. Accordingly, a scheme which asynchronously returns control from the communication call after initiating the communication, might result in data corruption for synchronous sockets.

To transparently provide asynchronous capabilities for synchronous sockets, two goals need to be met: (i) the interface should not change; the application can still use the same interface as earlier, i.e., the synchronous sockets interface and (ii) the application can assume the synchronous sockets semantics, i.e., once the control returns from the communication call, it can read from or write to the communication buffer. In our approach, the key idea in meeting these design goals is to memory-protect the user buffer (thus disallow the application from accessing it) and to carry out communication asynchronously from this buffer, while *tricking* the application into believing that we are carrying out data communication in a synchronous manner.

Figure 5.1 illustrates the designs of the synchronous zero-copy SDP (ZSDP) scheme and our asynchronous zero-copy SDP (AZ-SDP) scheme. As shown in Figure 5.1(a), in the ZSDP scheme, on a data transmission event, a *source-avail* message containing information about the source buffer is sent to the receiver side. The receiver, on seeing this request, initiates a *GET* on the source data to be fetched into the final destination buffer using an IBA RDMA read request. Once the *GET* has completed, the receiver sends a *GET_COMPLETE* message to the sender indicating that the communication has completed. The sender on receiving this *GET_COMPLETE* message, returns control to the application.

Figure 5.1(b) shows the design of the AZ-SDP scheme. This scheme is similar to the ZSDP scheme, except that it memory-protects the transmission application buffers and sends out several outstanding *source-avail* messages to the receiver. The receiver, on receiving these *source-avail* messages, memory-protects the receive application buffers and initiates several *GET* requests using multiple IBA RDMA read requests. On the completion of each of these *GET* requests, the receiver sends back *GET_COMPLETE* messages to the sender. Finally, on receiving the *GET_COMPLETE* requests, the sender unprotects the corresponding memory buffers. Thus, this approach allows for a better pipelining in the data communication providing a potential for much higher performance as compared to ZSDP.

### 5.2.2 Buffer Protection Mechanisms

As described in Section 5.2.1, our asynchronous communication mechanism uses memory-protect operations to disallow the application from accessing the communication buffer. If the application tries to access the buffer, a *page fault* is generated; our scheme needs to appropriately handle this, such that the semantics of synchronous sockets is maintained.

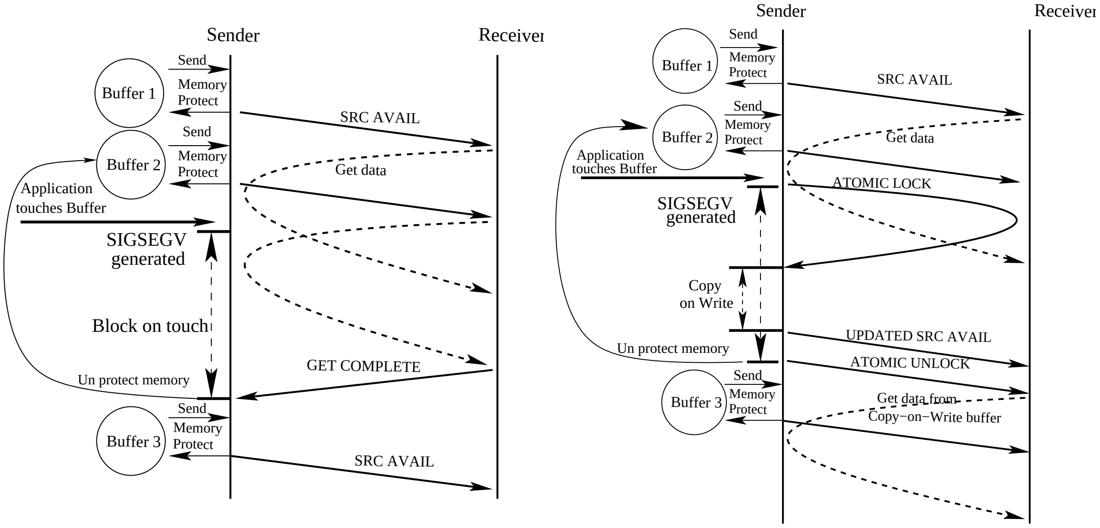

Figure 5.2: Buffer Protection Schemes for AZ-SDP: (a) Block-on-Write based buffer protection and (b) Copy-on-Write based buffer protection

As we will see in Section 8.3.1, if the application touches the communication buffer very frequently (thus generating *page faults* very frequently), it might impact the performance of AZ-SDP. However, the actual number of *page faults* that the application would generate depends closely on the kind of application we are trying to support. For example, if a middleware that supports non-blocking semantics is built on top of the sockets interface, we expect the number of *page faults* to be quite low. Considering MPI to be one example of such a middleware, whenever the end application calls a non-blocking communication call, MPI will have to implement this using the blocking semantics of sockets. However, till the application actually checks for completion, the data will remain untouched, thus reducing the number of *page faults* that might occur. Another example, is applications which perform data prefetching. As network throughput is increasing at a much faster rate as compared to the decrease in point-to-point latency, several applications today attempt to intelligently prefetch data that they might use in the future. This, essentially implies that though the prefetched data is transferred, it might be used at a much later time, if at all it is used. Again, in such scenarios, we expect the number of *page faults* occurring to be quite less. In this section, we describe generic approaches for handling the *page faults*. The performance, though, would depend on the actual number of *page faults* that the application would generate (which is further discussed in Section 8.3.1).

On the receiver side, we use a simple approach for ensuring the synchronous sockets semantics. Specifically, if the application calls a *recv()* call, the buffer to which the data is arriving is protected and the control is returned to the application. Now, if the receiver tries to read from this buffer before the data has actually arrived, our scheme blocks the application in the *page fault* until the data arrives. From the application's perspective, this operation is completely transparent except that the memory access would seem to take a longer time. On the sender side, however, we can consider two different approaches to handle this and guarantee the synchronous communication semantics: (i) *block-on-write* and (ii) *copy-on-write*. We discuss these alternatives in Sections 5.2.2 and Sections 5.2.2, respectively.

### Block on Write

This approach is similar to the approach used on the receiver side, i.e., if the application tries to access the communication buffer before the communication completes, we force the application to block (Figure 5.2(a)). The advantage of this approach is that we always achieve zero-copy communication (saving on CPU cycles by avoiding memory copies). The disadvantage of this approach is that it is not skew tolerant, i.e., if the receiver process is delayed because of some computation and cannot post a receive for the communication, the sender has to block waiting for the receiver to arrive.

### Copy on Write

The idea of this approach is to perform a copy-on-write operation from the communication buffer to a temporary internal buffer when a *page fault* is generated. However, before control is returned to the user, the AZ-SDP layer needs to ensure that the receiver has not already started the *GET* operation; otherwise, it could result in data corruption.

This scheme performs the following steps to maintain the synchronous sockets semantics (illustrated in Figure 5.2(b)):

1. The AZ-SDP layer maintains a lock at the receiver side for each *source-avail* message.
2. Once the receiver calls a *recv()* and sees this *source-avail* message, it sets the lock and initiates the *GET* operation for the data using the IBA RDMA read operation.
3. On the sender side, if a *page fault* occurs (due to the application trying to touch the buffer), the AZ-SDP layer attempts to obtain the lock on the receiver side using an IBA *compare-and-swap* atomic operation. Depending on whether the sender gets a *page fault* first or the receiver calls the *recv()* operation first, one of them will get the lock.
4. If the sender gets the lock, it means that the receiver has not called a *recv()* for the data yet. In this case, the sender copies the data into a *copy-on-write buffer*, sends an *updated-source-avail* message pointing to the *copy-on-write buffer* and

returns the lock. During this time, if the receiver attempts to get the lock and fails, it just blocks waiting for the *updated-source-avail* message.

5. If the sender does not get the lock, it means that the receiver has already called the *recv()* call and is in the process of transferring data. In this case, the sender just blocks waiting for the receiver to complete the data transfer and send it a *GET_COMPLETE* message.

The advantage of this approach is that it is more skew tolerant as compared to the *block-on-write* approach, i.e., if the receiver is delayed because of some computation and does not call a *recv()* soon, the sender does not have to block. The disadvantages of this approach, on the other hand, are: (i) it requires an additional copy operation, so it consumes more CPU cycles as compared to the ZSDP scheme and (ii) it has an additional lock management phase which adds more overhead in the communication. Thus, this approach may result in higher overhead than even the copy-based scheme (BSDP) when there is no skew.

### 5.2.3 Handling Buffer Sharing

Several applications perform buffer sharing using approaches such as memory-mapping two different buffers (e.g., *mmap()* operation). Let us consider a scenario where buffer *B1* and buffer *B2* are memory-mapped to each other. In this case, it is possible that the application can perform a *send()* operation from *B1* and try to access *B2*. In our approach, we memory-protect *B1* and disallow all accesses to it. However, if the application writes to *B2*, this newly written data is reflected in *B1* as well (due to the memory-mapping); this can potentially take place before the data is actually transmitted from *B1* and can cause data corruption.

In order to handle this, we override the *mmap()* call from *libc* to call our own *mmap()* call. The new *mmap()* call, internally maintains a mapping of all memory-mapped buffers. Now, if any communication is initiated from one buffer, all buffers memory-mapped to this buffer are protected. Similarly, if a *page fault* occurs, memory access is blocked (or *copy-on-write* performed) till all communication for this and its associated memory-mapped buffers has completed.

### 5.2.4 Handling Unaligned Buffers

The *mprotect()* operation used to memory-protect buffers in Linux, performs memory-protects in a granularity of a physical page size, i.e., if a buffer is protected, all physical pages on which it resides are protected. However, when the application is performing communication from a buffer, it is not necessary that this buffer is aligned so that it starts on a physical page.

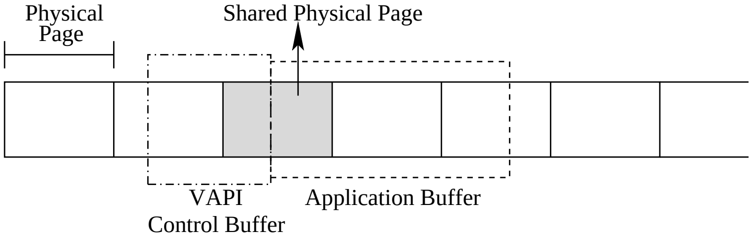

Figure 5.3: Physical Page Sharing Between Two Buffers

Let us consider the case depicted in Figure 5.3. In this case, the application buffer shares the same physical page with a control buffer used by the protocol layer, e.g, VAPI. Here, if we protect the application buffer disallowing any access to it, the protocol's internal control buffer is also protected. Now, suppose the protocol layer needs to access this control buffer to carry out the data transmission; this would result in a deadlock. In this section, we present two approaches for handling this: (i) Malloc Hook and (ii) Hybrid approach with BSDP.

### Malloc Hook

In this approach, we provide a hook for the *malloc()* and *free()* calls, i.e., we override the *malloc()* and *free()* calls to be redirected to our own memory allocation and freeing functions. Now, in the new memory allocation function, if an allocation for $N$ bytes is requested, we allocate $N$ + $PAGE\_SIZE$ bytes and return a pointer to a portion of this large buffer such that the start address is aligned to a physical page boundary.

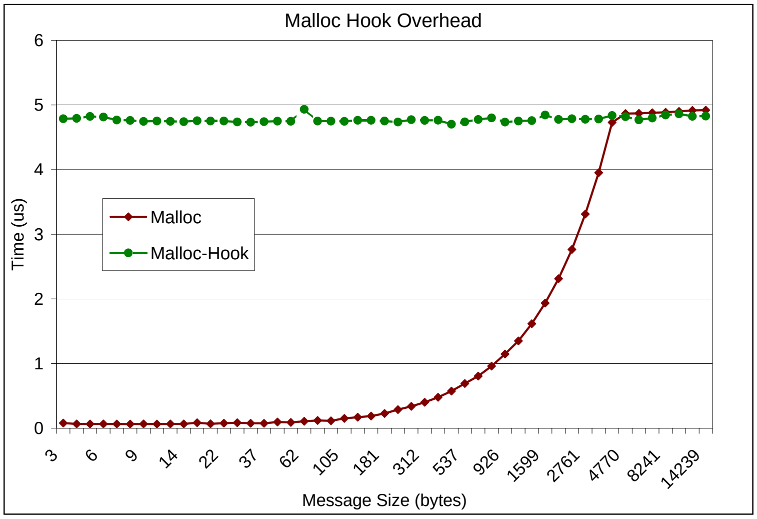

Figure 5.4: Overhead of the Malloc Hook

While this approach is simple, it has several disadvantages. First, if the application calls several small buffer allocations, for each call atleast a $PAGE\_SIZE$ amount of buffer is allocated. This might result in a lot of wastage. Second, as shown in Figure 5.4, the amount of time taken to perform a memory allocation operation increases significantly from a small buffer allocation to a PAGE_SIZE amount of buffer allocation. Thus, if we use a malloc hook, even a 40 byte memory allocation would take the amount of time equivalent to that of a complete physical page size, i.e., instead of 0.1$\mu$s, a 40 byte memory allocation would take about 4.8$\mu$s.

Table 5.1: Transmission Initiation Overhead

| Operation | w/ Malloc ($\mu$s) | w/ Malloc Hook ($\mu$s) |
|---|---|---|
| Registration Check | 1.4 | 1.4 |
| Memory-Protect | 1.4 | 1.4 |
| Memory Copy | 0.3 | 0.3 |
| **Malloc** | **0.1** | **4.8** |
| Descriptor Post | 1.6 | 1.6 |
| Other | 1.1 | 1.1 |

To understand the impact of the additional memory allocation time, we show the break up of the message transmission initiation phase in Table 5.1. As shown in the table, there are several steps involved in initiating a data transfer. Of these, the memory allocation overhead is of primary interest to us. For small message communication (e.g., source- and sink-avail messages), VAPI allocates a small buffer (40 bytes), copies the data into the buffer together with the descriptor describing the buffer itself and its protection attributes. This allows the network adapter to fetch both the descriptor as well as the actual buffer in a single DMA operation. Here, we calculate the memory allocation portion for the small buffer (40 bytes) as the fourth overhead. As we can see in the table, by adding our malloc hook, all the overheads remain the same, except for the memory allocation overhead which increases to 4.8$\mu$s, i.e., its portion in the entire transmission initiation overhead increases to about 45% from 1.5% making it the dominant overhead in the data transmission initiation part.

### Hybrid Approach with Buffered SDP (BSDP)

In this approach, we use a hybrid mechanism between AZ-SDP and BSDP. Specifically, if the buffer is not page-aligned, we transmit the page-aligned portions of the buffer using AZ-SDP and the remaining portions of the buffer using BSDP. The beginning and end portions of the communication buffer are thus sent through BSDP while the intermediate portion over AZ-SDP.

This approach does not have any of the disadvantages mentioned for the previous malloc-hook based scheme. The only disadvantage is that a single message communication might need to be carried out in multiple communication operations (at most three). This might add some overhead when the communication buffers are not page-aligned. In our preliminary results, we noticed that this approach gives about 5% to 10% better throughput as compared to the malloc-hook based scheme. Hence, we went ahead with this approach.

## 5.3 Experimental Evaluation

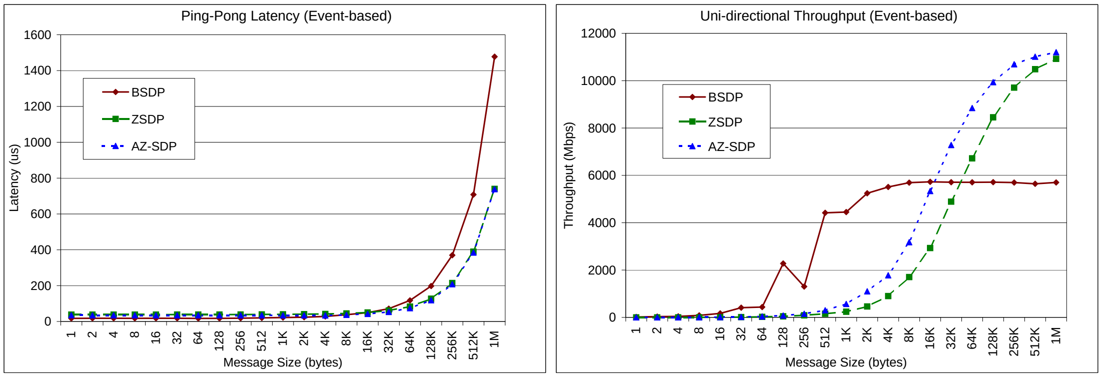

Figure 5.5: Micro-Benchmarks: (a) Ping-Pong Latency and (b) Unidirectional Throughput

In this section, we evaluate the AZ-SDP implementation and compare it with the other two implementations of SDP, i.e., BSDP and ZSDP. We perform two sets of evaluations. In the first set (section 8.3.1), we use single connection benchmarks such as ping-pong latency, uni-directional throughput, computation-communication overlap capabilities and effect of page faults. In the second set (section 8.3.2), we use multi-connection benchmarks such as hot-spot latency, multi-stream throughput and multi-client throughput tests. For AZ-SDP, our results are based on the *block-on-write* technique for page faults.

The experimental test-bed consists of four nodes with dual 3.6 GHz Intel Xeon EM64T processors. Each node has a 2 MB L2 cache and 512 MB of 333 MHz DDR SDRAM. The nodes are equipped with Mellanox MT25208 InfiniHost III DDR PCI-Express adapters (capable of a link-rate of 20 Gbps) and are connected to a Mellanox MTS-2400, 24-port fully non-blocking DDR switch.

### 5.3.1 Single Connection Micro-Benchmarks

In this section, we evaluate the three stacks with a suite of single connection micro-benchmarks.

**Ping-Pong Latency:** Figure 5.5(a) shows the point-to-point latency achieved by the three stacks. In this test, the sender node first sends a message to the receiver; the receiver receives this message and sends back another message to the sender. Such exchange is carried out for several iterations, the total time calculated and averaged over the number of iterations. This gives the time for a complete message exchange. The ping-pong latency demonstrated in the figure is half of this amount (one-way communication).

As shown in the figure, both zero-copy schemes (ZSDP and AZ-SDP) achieve a superior ping-pong latency as compared to BSDP. However, there is no significant difference in the performance of ZSDP and AZ-SDP. This is due to the way the ping-pong latency test is designed. In this test, only one message is sent at a time and the node has to wait for a reply from its peer before it can send the next message, i.e., the test itself is completely synchronous and cannot utilize the capability of AZ-SDP with respect to allowing multiple outstanding requests on the network at any given time, resulting in no performance difference between the two schemes.

**Uni-directional Throughput:** Figure 5.5(b) shows the uni-directional throughput achieved by the three stacks. In this test, the sender node keeps streaming data and the receiver keeps receiving it. Once the data transfer completes, the time is measured and the data rate is calculated as the number of bytes sent out per unit time. We used the *ttcp* benchmark [80] version 1.4.7 for this experiment.

As shown in the figure, for small messages BSDP performs the best. The reason for this is two fold: (i) Both ZSDP and AZ-SDP rely on control message exchange for every message to be transferred. This causes an additional overhead for each data transfer which is significant for small messages and (ii) Our BSDP implementation uses an optimization technique known as reverse packetization to improve the throughput for small messages. More details about this can be found in [12].

For medium and large messages, on the other hand, AZ-SDP and ZSDP outperform BSDP because of the zero-copy communication. Also, for medium messages, AZ-SDP performs the best with about 35% improvement compared to ZSDP.

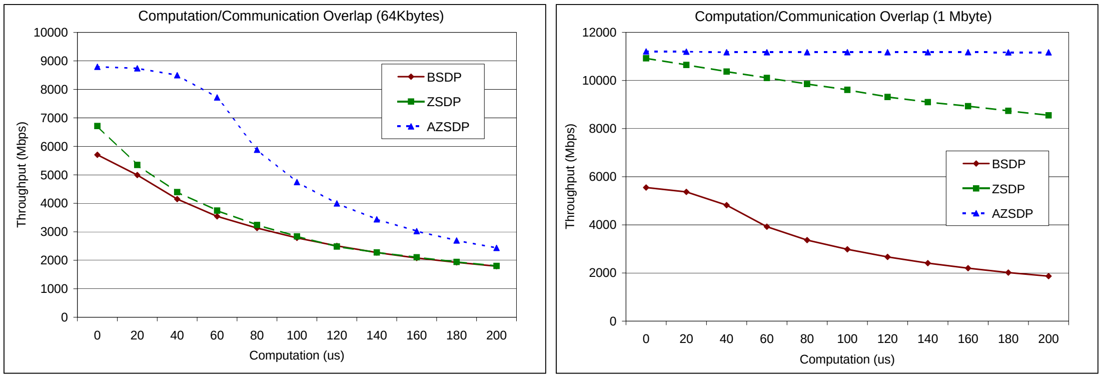

Figure 5.6: Computation and Communication Overlap Micro-Benchmark: (a) 64Kbyte message and (b) 1Mbyte message

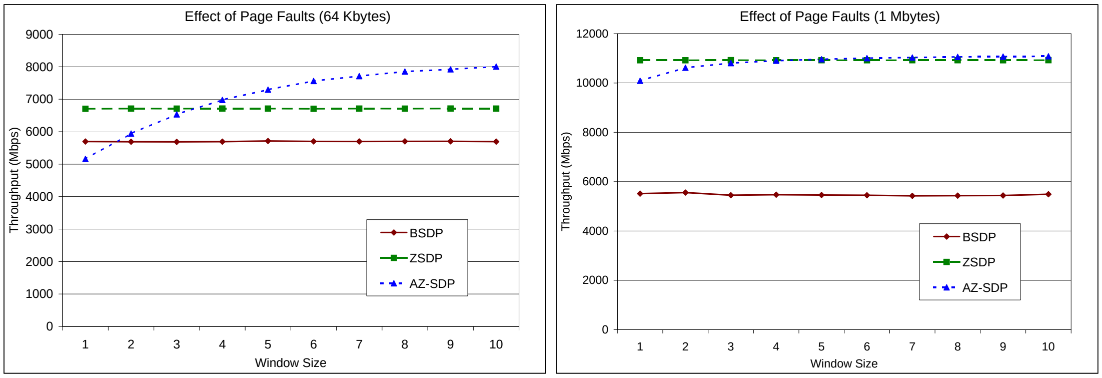

Figure 5.7: Impact of Page Faults: (a) 64Kbyte message and (b) 1Mbyte message

**Computation-Communication Overlap:** As mentioned earlier, IBA provides hardware offloaded network and transport layers to allow high performance communication. This also implies that the host CPU now has to do lesser amount of work for carrying out the communication, i.e., once the data transfer is initiated, the host is free to carry out its own computation while the actual communication is carried out by the network adapter. However, the amount of such overlap between the computation and communication that each of the schemes can exploit varies. In this experiment, we measure the capability of each scheme with respect to overlapping application computation with the network communication. Specifically, we modify the *ttcp* benchmark to add additional computation between every data transmission. We vary the amount of this computation and report the throughput delivered.

Figure 5.6 shows the overlap capability for the different schemes with the amount of computation added represented on the x-axis and the throughput measured, on the y-axis. Figure 5.6(a) shows the overlap capability for a message size of 64Kbytes and Figure 5.6(b) shows that for a message size of 1Mbyte. As shown in the figures, AZ-SDP achieves much higher computation-communication overlap as compared to the other schemes, as illustrated by its capability to retain high performance even for a large amount of intermediate computation. For example, for a message size of 64Kbytes, AZ-SDP achieves an improvement of up to a factor of 2. Also, for a message size of 1Mbyte, we see absolutely no drop in the performance of AZ-SDP even with a computation amount of 200$\mu$s while the other schemes see a huge degradation in the performance.

The reason for this better performance of AZ-SDP is its capability to utilize the hardware offloaded protocol stack more efficiently, i.e., once the data buffer is protected and the transmission initiated, AZ-SDP returns control to the application allowing it to perform its computation while the network hardware takes care of the data transmission. ZSDP on the other hand waits for the actual data to be transmitted before returning control to the application, i.e., it takes absolutely no advantage of the network adapter's capability to carry out data transmission independently.

**Impact of Page Faults:** As described earlier, the AZ-SDP scheme protects memory locations and carries out communication from or to these memory locations asynchronously. If the application tries to touch the data buffer before the communication completes, a *page fault* is generated. The AZ-SDP implementation traps this event, blocks to make sure that the data communication completes and returns the control to the application (allowing it to touch the buffer). Thus, in the case where the application very frequently touches the data buffer immediately after communication event, the AZ-SDP scheme has to handle several page faults adding some amount of overhead to this scheme. To characterize this overhead, we have modified the *ttcp* benchmark to touch data occasionally. We define a variable known as the *Window Size (W)* for this. The sender side first calls *W* data transmission calls and then writes a pattern into the transmission buffer. Similarly, the receiver calls *W* data reception calls and then reads the pattern from the reception buffer. This obviously does not impact the BSDP and ZSDP schemes since they do not perform any kind of protection of the application buffers. However, for the AZ-SDP scheme, each time the sender tries to write to the buffer or the receiver tries to read from the buffer, a *page fault* is generated, adding additional overhead.

Figure 5.7 shows the impact of *page faults* on the three schemes for message sizes 64Kbytes and 1Mbyte respectively. As shown in the figure, for small window sizes, the performance of AZ-SDP is poor. Though this degradation is lesser for larger message sizes (Figure 5.7(b)), there is still some amount of drop. There are two reasons for this: (i) When a *page fault* is generated, no additional data transmission or reception requests are initiated; thus, during this time, the behavior of ZSDP and AZ-SDP would be similar and (ii) Each *page fault* adds about 6$\mu$s overhead. Thus, though AZ-SDP falls back to the ZSDP scheme, it still has to deal with the *page faults* for previous protected buffers causing worse performance than ZSDP[1].

### 5.3.2 Multi-Connection Micro-Benchmarks

In this section, we present the evaluation of the stacks with several benchmarks which carry out communication over multiple connections simultaneously.

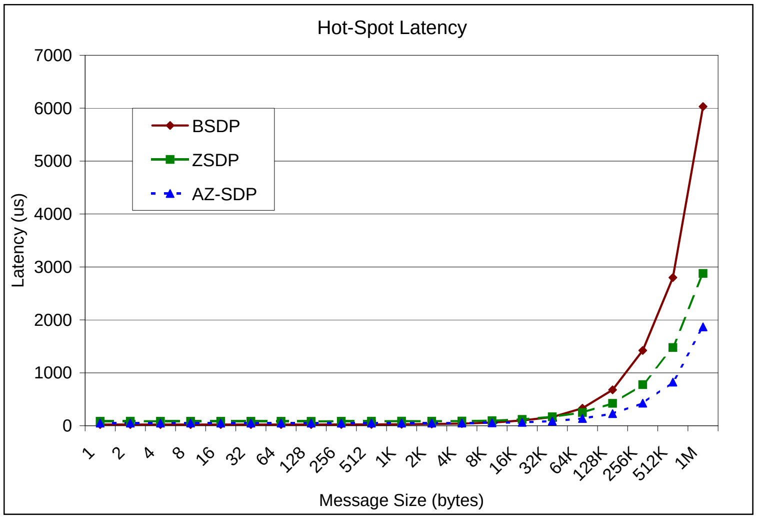

Figure 5.8: Hot-Spot Latency Test

[1] We tackle this problem by allowing AZ-SDP to completely fall back to ZSDP if the application has generated more *page faults* than a certain threshold. However, to avoid diluting the results, we set this threshold to a very high number so that it is never triggered in the experiments.

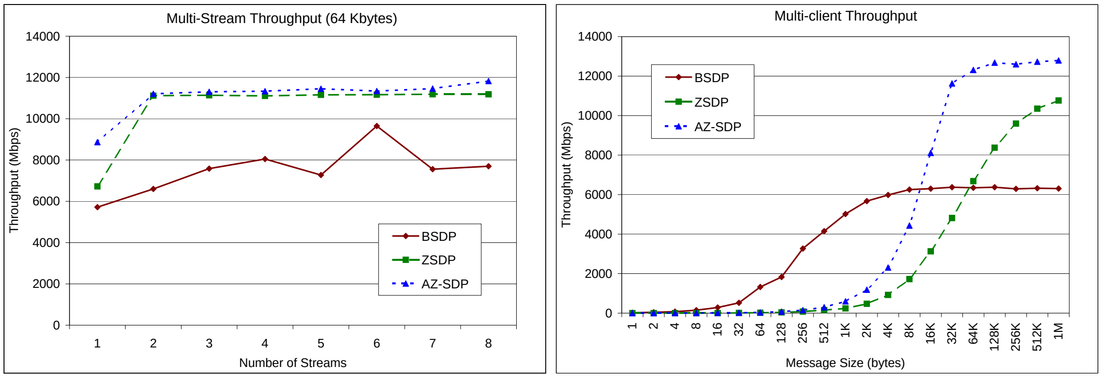

Figure 5.9: Multi-Connection Micro-Benchmarks: (a) Multi-Stream Throughput test and (b) Multi-Client Throughput test

**Hot-Spot Latency Test:** Figure 5.8 shows the impact of multiple connections in message transaction kind of environments. In this experiment, a number of client nodes perform point-to-point latency test with the same server, forming a hot-spot on the server. We perform this experiment with one node acting as a server node and three other dual-processor nodes hosting a total of 6 client processes and report the average of the latencies observed by each client process. As shown in the figure, AZ-SDP significantly outperforms the other two schemes especially for large messages. The key to the performance difference in this experiment lies in the usage of multiple connections for the test. Since the server has to deal with several connections at the same time, it can initiate a request to the first client and handle the other connections while the first data transfer is taking place. Thus, though each connection is synchronous, the overall experiment provides some asynchronism with respect to the number of clients the server has to deal with. Further, we expect this benefit to grow with the number of clients allowing a better scalability for the AZ-SDP scheme.

**Multi-Stream Throughput Test:** The multi-stream throughput test is similar to the uni-directional throughput test, except that multiple threads on the same pair of physical nodes carry out uni-directional communication separately. We measure the aggregate throughput of all the threads together and report it in Figure 5.9(a). The message size used for the test is 64Kbytes; the x-axis gives the number of threads used and the y-axis gives the throughput achieved. As shown in the figure, when the number of streams is *one*, the test behaves similar to the uni-directional throughput test with AZ-SDP outperforming the other schemes. However, when we have more streams performing communication as well, the performance of ZSDP is also similar to what AZ-SDP can achieve. To understand this behavior, we briefly reiterate on the way the ZSDP scheme works. In the ZSDP scheme, when a process tries to send the data out to a remote process, it sends the *buffer availability notification* message and *waits* till the remote process completes the data communication and informs it about the completion. Now, in a multi-threaded environment, while the first process is waiting, the remaining processes can go ahead and send out messages. Thus, though each thread is blocking for progress in ZSDP, the network is not left unutilized due to several threads accessing it simultaneously. This results in ZSDP achieving a similar performance as AZ-SDP in this environment.

**Multi-client Throughput Test:** In the multi-client throughput test, similar to the hot-spot test, we use one server and 6 clients (spread over three dual-processor physical nodes). In this setup, we perform the streaming throughput test between each of the clients and the same server. As shown in Figure 5.9(b), AZ-SDP performs significantly better than both ZSDP and BSDP in this test. Like the hot-spot test, the improvement in the performance of AZ-SDP is attributed to its ability to perform communication over the different connections simultaneously while ZSDP and BSDP perform communication one connection at a time.

## 5.4 Summary

In this chapter we proposed a mechanism, termed as *AZ-SDP (Asynchronous Zero-Copy SDP)*, which allows the approaches proposed for asynchronous sockets to be used for synchronous sockets, while maintaining the synchronous sockets semantics. We presented our detailed design in this chapter and evaluated the stack with an extensive set of micro-benchmarks. The experimental results demonstrate that our approach can provide an improvement of close to 35% for medium-message unidirectional throughput and up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks.

# CHAPTER 6

# RDMA SUPPORTED PACKETIZED FLOW CONTROL FOR THE SOCKETS DIRECT PROTOCOL (SDP) OVER INFINIBAND

Most high-speed networks have strict requirements on their upper layers that the receiver has to post a receive descriptor informing the network adapter about where to place the incoming message before the message is actually sent by the sender. Not doing so might result in limited retransmissions of the message (increasing the network load) and/or the connection being dropped or terminated. Thus, most programming models and upper layers use different schemes to handle this requirement. Like several other programming models, SDP uses a credit-based flow-control approach to handle this.

However, as we will discuss in the later sections, though the credit-based flow control is a simple and generic approach for communication, it decouples the send and receive buffers by forcing the sender to manage the send buffers and the receiver to manage the receive buffers. Thus, explicit synchronization is required within the critical path of communication in order to make sure that the sender does not overrun the receiver. Further, since the receiver is not aware of the messages that the sender is going to send, it statically and conservatively allocates buffers. Thus, when the sender sends data, the buffers might not be utilized completely.

In this chapter, we present a new flow-control approach for SDP known as the packetized flow control, which utilizes RDMA-based one-sided communication operations to perform completely sender-side buffer management for both the sender as well as the receiver buffers. This allows us to remove the “communication gap” that is formed between the sender and the receiver for managing buffers and helps us in improving the buffer usage as well as the performance achievable by SDP.

Our results demonstrate that our scheme can significantly improve the buffer usage of SDP’s flow control to close to the ideal 100% which is several orders of magnitude higher than existing schemes. Further, our scheme also allows SDP to utilize the network in a more effective manner by coalescing many small packets into a few large packets, thus improving the overall performance by close to 10X for medium message sizes.

## 6.1 Overview of Credit-based Flow-control in SDP

As mentioned earlier, most programming models and upper layers used the credit-based flow control mechanism for communication. We have presented the basic credit-based flow control approach in Section 2.4.1. Here we provide a brief overview of the same.

In this approach, the sender is given a certain number of credits (tokens). It loses a credit for every message sent and gains a credit 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 (Figure 6.1).

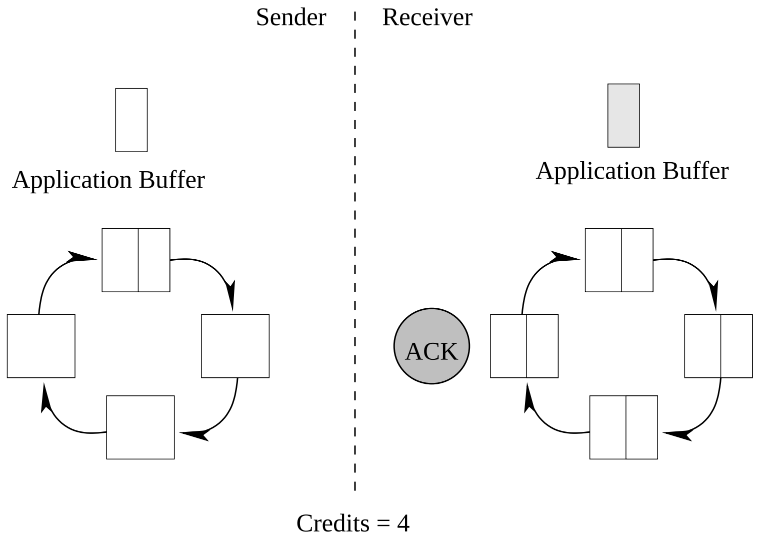

Figure 6.1: The Credit Based Approach

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 potential live lock. To avoid this problem, we have proposed the following solutions:

1. **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.

2. **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.

3. **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 $2N$. On the basis of the same, this approach was used in the substrate.

In the credit-based flow control approach, the receiver is "blind" to the sizes of the incoming messages. Accordingly, statically sized buffers are allocated in circular fashion, both on the sender and the receiver side. There are as many intermediate buffers, of size $S$, as the number of credits. So, if there are credits available, the sender can directly copy the outstanding message into the intermediate buffer and send it out to the next available receive buffer. If the message is $S$ bytes or smaller in size, it is copied to the intermediate buffer on the sender side and sent to the receiver side intermediate buffer. If the message is larger than $S$ bytes in size, it is broken up into $S$ byte chunks and copied into as many intermediate buffers as available. Data from each of these buffers is sent out as soon as a credit is available.

## 6.2 Packetized Flow Control: Design and Implementation

Each of the credit-based flow control mechanisms discussed in the previous section (Section 6.1), have their own disadvantages. Packetized flow-control has been designed to solve these problems. In this section, we provide the design and implementation details of the same.

The main disadvantage of the credit-based flow control scheme is based on the way it handles the communication of small messages, i.e., when the sender is transmitting small messages, each message uses up an entire buffer on the receiver side, thus

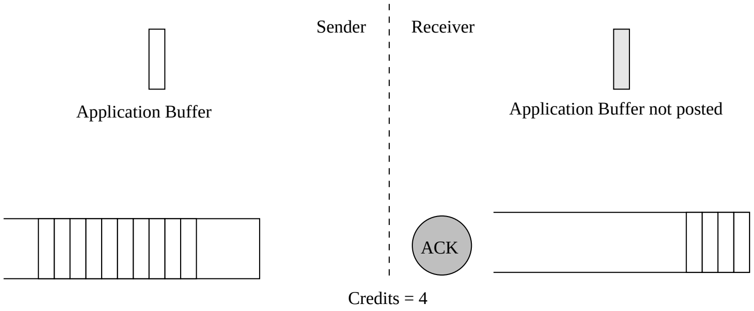

Figure 6.2: Packetized Flow Control Approach

wasting the buffer space available. For example if each message is only 1 byte and each buffer is 8 KB, effectively 99.8% of the buffer space is completely under-utilized. This wastage of buffers also reflects on the number of messages that are sent out, i.e., excessive under-utilization of buffer space might result in the SDP layer to “believe” that it has used up its resources in spite of having free resources in reality.

Another disadvantage of the credit-based flow control mechanism is its network utilization. Since this approach directly sends out data as soon as the sender has requested for transmission, it might result in very small messages being posted to the network. This, of course, results in the under-utilization of the network and hence in degradation in performance.

Packetized flow-control utilizes advanced network features such as RDMA to solve these problems with credit based flow control. In this scheme, the entire intermediate buffer is one continuous buffer instead of several buffers connected in a circular list. Or in other words, the intermediate buffer is packetized into buffers of 1 byte size. The entire buffer management, for both the sender as well as the receiver, is carried out on the sender side alone using RDMA operation. Since the sender knows exactly what size messages it is sending, this allows the sender to manage the receiver buffer in a better manner. When the new message has to be sent out, the sender knows the address of the next free location on the receiver side and can place the new message in the appropriate position using an RDMA write operation. Thus the wastage of buffers is minimal in this approach and close to the ideal 100% in most cases.

Further, if the application posts a new message to be sent after all the credits have been used up, the message can be copied to the intermediate buffer (where there is more space available due to better buffer management) and sent at a later time. The application returns assuming that the data has been sent and carries on with computation. This allows the SDP layer to coalesce multiple small messages into one larger message, thus improving the network utilization and hence the performance.

In summary, the advantages of Packetized flow control are two fold. Firstly, it avoids buffer wastage for small and medium sized messages. Server side buffer management assures that there is no buffer wastage. Secondly, it increases throughput for small and medium sized messages. When the sender is out of remote credits, it can buffer all the incoming messages until the intermediate buffer is full and thus coalesce a number of small messages into one large message. When a credit is available, one large message is sent out. This results in a higher throughput than sending each small message individually and achieves better throughput.

## 6.3 Experimental Results

In this section, we evaluate the performance of the packetized flow control as compared to the regular credit-based flow control approach. We present ping-pong latency and uni-directional throughput as well as the temporary buffer utilization measurements for both these approaches.

The experimental test-bed consists of four nodes with dual 3.6 GHz Intel Xeon EM64T processors. Each node has a 2 MB L2 cache and 512 MB of 333 MHz DDR SDRAM. The nodes are equipped with Mellanox MT25208 InfiniHost III DDR PCI-Express adapters (capable of a link-rate of 20 Gbps) and are connected to a Mellanox MTS-2400, 24-port fully non-blocking DDR switch.

### 6.3.1 Latency and Bandwidth

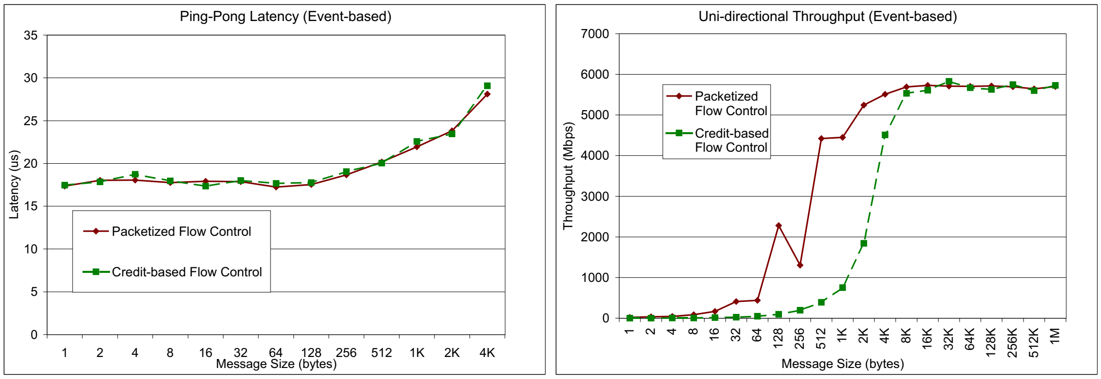

Figure 6.3: Micro-Benchmarks: (a) Ping-Pong Latency and (b) Unidirectional Throughput

Figure 6.3(a) shows the ping-pong latency comparison of the packetized flow-control approach with that of the credit-based flow control approach. As shown in the figure, there is no difference in the latency for both these approaches. The reason for this indifference is that the packetized flow control approach essentially improves the resource utilization when multiple back-to-back messages are transmitted. The ping-pong latency test, however, is completely synchronous in nature and does not transmit any back-to-back messages.

Figure 6.3(b) shows the unidirectional throughput comparison of the two approaches. As shown in the figure, packetized flow control achieves around 10X improvement in the throughput for small to medium messages. As mentioned earlier, this improvement is attributed to the better network utilization and buffer management associated with the packetized flow control approach.

### 6.3.2 Temporary Buffer Utilization

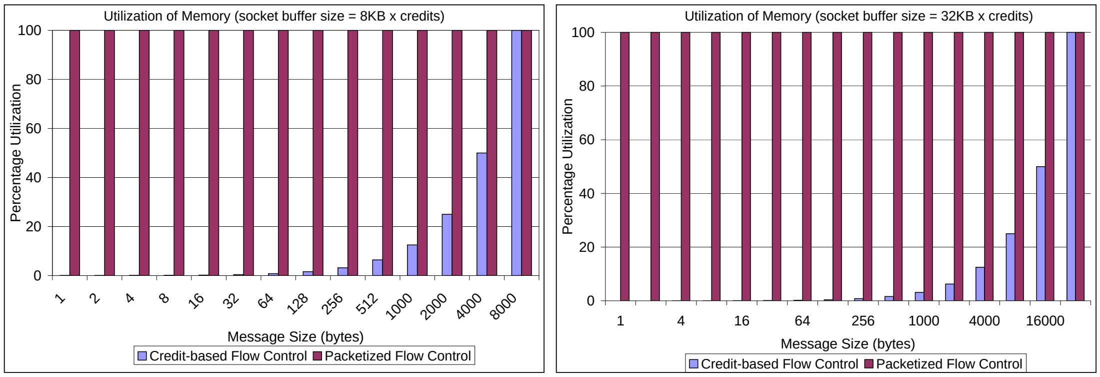

Figure 6.4: Temporary Buffer Utilization: (a) Socket buffer size = 8KB x credits, (b) Socket buffer size = 32KB x credits

Figures 6.4(a) and 6.4(b) show the way the amount of the temporary socket buffer each scheme utilizes (for different socket buffer sizes). As shown in the figures, for very small message sizes, the credit-based flow control approach is extremely wasteful utilizing close to 0% of the entire buffer space available. As the message size increases, however, this approach starts using a larger fraction of the buffer. Packetized flow control, on the other hand, always utilizes close to 100% of the allotted temporary buffer space.

## 6.4 Summary

In this chapter, we have proposed an enhanced flow control mechanism, termed as the packetized flow control, for the Sockets Direct Protocol over InfiniBand. This approach tries to meet the limitations of the existing flow control mechanisms in terms of resource usage and performance. We have presented some preliminary performance results in both these aspects and showed that our approach can achieve a throughput improvement of close to an order of magnitude for small and medium message sizes. Further, the resource usage of our approach is several orders of magnitude better in some cases.

# CHAPTER 7

# PERFORMANCE CHARACTERIZATION OF A 10-GIGABIT ETHERNET TCP OFFLOAD ENGINE (TOE)

Despite the performance criticisms of Ethernet for high-performance computing (HPC), the Top500 Supercomputer List [7] continues to move towards more commodity-based Ethernet clusters. Just three years ago, there were *zero* Gigabit Ethernet-based clusters in the Top500 list; now, Gigabit Ethernet-based clusters make up 176 (or 35.2%) of these. The primary drivers of this Ethernet trend are ease of deployment and cost. So, even though the end-to-end throughput and latency of Gigabit Ethernet (GigE) lags exotic high-speed networks such as Quadrics [69], Myrinet [24], and InfiniBand [10] by as much as ten-fold, the current trend indicates that GigE-based clusters will soon make up over half of the Top500 (as early as November 2005). Further, Ethernet is already the ubiquitous interconnect technology for commodity grid computing because it leverages the legacy Ethernet/IP infrastructure whose roots date back to the mid-1970s. Its ubiquity will become even more widespread as long-haul network providers move towards 10-Gigabit Ethernet (10GigE) [52, 45] backbones, as recently demonstrated by the longest continuous 10GigE connection between Tokyo, Japan and Geneva, Switzerland via Canada and the United States [41]. Specifically, in late 2004, researchers from Japan, Canada, the United States, and Europe completed an 18,500-km 10GigE connection between the Japanese Data Reservoir project in Tokyo and the CERN particle physical laboratory in Geneva; a connection that used 10GigE WAN PHY technology to set-up a *local-area network* at the University of Tokyo that appeared to include systems at CERN, which were 17 time zones away.

Given that GigE is so far behind the curve with respect to network performance, can 10GigE bridge the performance divide while achieving the ease of deployment and cost of GigE? Arguably yes. The IEEE 802.3-ae 10-Gb/s standard, supported by the 10GigE Alliance, already ensures interoperability with existing Ethernet/IP infrastructures, and the manufacturing volume of 10GigE is already driving costs down exponentially, just as it did for Fast Ethernet and Gigabit Ethernet[2]. This leaves us with the "performance divide" between 10GigE and the more exotic network technologies.

In a distributed grid environment, the performance difference is a non-issue mainly because of the ubiquity of Ethernet and IP as the routing language of choice for local-, metropolitan, and wide-area networks in support of grid computing. Ethernet has become synonymous with IP for these environments, allowing complete compatibility for clusters using Ethernet to communicate over these environments. On the other hand, networks such as Quadrics, Myrinet, and InfiniBand are unusable in such environments due to their incompatibility with Ethernet and due to their limitations against using the IP stack in order to maintain a high performance.

[2] Per-port costs for 10GigE have dropped nearly ten-fold in two years.

With respect to the cluster environment, Gigabit Ethernet suffers from an order-of-magnitude performance penalty when compared to networks such as Quadrics and InfiniBand. In our previous work [52, 45, 15], we had demonstrated the capabilities of the basic 10GigE adapters in bridging this gap. In this chapter, we take the next step by demonstrating the capabilities of the Chelsio T110 10GigE adapter with TCP Offload Engine (TOE). We present performance evaluations in three broad categories: (i) detailed micro-benchmark performance evaluation at the sockets layer, (ii) performance evaluation of the MPI stack atop the sockets interface, and (iii) application-level evaluation using the Apache web server [6]. Our experimental results demonstrate latency as low as 8.9 $\mu$s and throughput of nearly 7.6 Gbps for these adapters. Further, we see an order-of-magnitude improvement in the performance of the Apache web server while utilizing the TOE as compared to a 10GigE adapter without TOE.

## 7.1 Background

In this section, we briefly discuss the TOE architecture and provide an overview of the Chelsio T110 10GigE adapter.

### 7.1.1 Overview of TCP Offload Engines (TOEs)

The processing of traditional protocols such as TCP/IP and UDP/IP is accomplished by software running on the central processor, CPU or microprocessor, of the server. As network connections scale beyond GigE speeds, the CPU becomes burdened with the large amount of protocol processing required. Resource-intensive memory copies, checksum computation, interrupts, and reassembling of out-of-order packets put a tremendous amount of load on the host CPU. In high-speed networks,

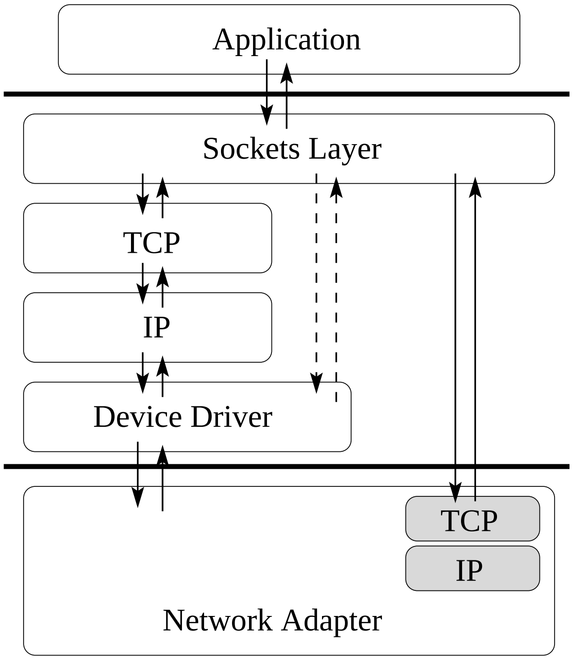

Figure 7.1: TCP Offload Engines

the CPU has to dedicate more processing to handle the network traffic than to the applications it is running. TCP Offload Engines (TOEs) are emerging as a solution to limit the processing required by CPUs for networking.

The basic idea of a TOE is to offload the processing of protocols from the host processor to the hardware on the adapter or in the system (Figure 7.1). A TOE can be implemented with a network processor and firmware, specialized ASICs, or a combination of both. Most TOE implementations available in the market concentrate on offloading the TCP and IP processing, while a few of them focus on other protocols such as UDP/IP.

As a precursor to complete protocol offloading, some operating systems started incorporating support for features to offload some compute-intensive features from the host to the underlying adapter, e.g., TCP/UDP and IP checksum offload. But as Ethernet speeds increased beyond 100 Mbps, the need for further protocol processing offload became a clear requirement. Some GigE adapters complemented this requirement by offloading TCP/IP and UDP/IP segmentation or even the whole protocol stack onto the network adapter [53, 40].

### 7.1.2 Chelsio 10-Gigabit Ethernet TOE

The Chelsio T110 is a PCI-X network adapter capable of supporting full TCP/IP offloading from a host system at line speeds of 10 Gbps. The adapter consists of multiple components: the Terminator which provides the basis for offloading, separate memory systems each designed for holding particular types of data, and a MAC and XPAC Optical Transceiver for physically transferring data over the line. An overview of the T110's architecture can be seen in Figure 7.2.

Context (CM) and Packet (PM) memory are available on-board as well as a 64 KB EEPROM. A 4.5 MB TCAM is used to store a Layer 3 routing table and can filter out invalid segments for non-offloaded connections. The T110 is a Terminator ASIC, which is the core of the offload engine, capable of handling 64,000 connections at once, with a setup and tear-down rate of about 3 million connections per second.

**Memory Layout:** Two types of on-board memory are available to the Terminator. 256 MB of EFF FCRAM Context Memory stores TCP state information for each offloaded and protected non-offloaded connection as well as a Layer 3 routing table and its associated structures. Each connection uses 128 bytes of memory to store state information in a TCP Control Block. For payload (packets), standard ECC SDRAM (PC2700) can be used, ranging from 128 MB to 4 GB.

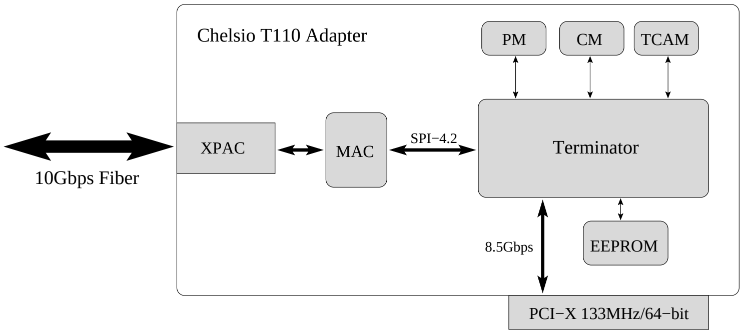

Figure 7.2: Chelsio T110 Adapter Architecture

**Terminator Core:** The Terminator sits between a systems host and its Ethernet interface. When offloading a TCP/IP connection, it can handle such tasks as connection management, checksums, route lookup from the TCAM, congestion control, and most other TCP/IP processing. When offloading is not desired, a connection can be tunneled directly to the host's TCP/IP stack. In most cases, the PCI-X interface is used to send both data and control messages between the host, but an SPI-4.2 interface can be used to pass data to and from a network processor (NPU) for further processing.

## 7.2 Interfacing with the TOE

Since the Linux kernel does not currently support TCP Offload Engines (TOEs), there are various approaches researchers have taken in order to allow applications to interface with TOEs. The two predominant approaches are High Performance Sockets (HPS) [74, 57, 58, 16, 18, 14, 55] and TCP Stack Override. The Chelsio T110 adapter uses the latter approach.

In this approach, the kernel-based sockets layer is retained and used by the applications. However, the TCP/IP stack is overridden, and the data is pushed directly to the offloaded protocol stack, bypassing the host TCP/IP stack implementation. One of Chelsio's goals in constructing a TOE was to keep it from being too invasive to the current structure of the system. By adding kernel hooks inside the TCP/IP stack and avoiding actual code changes, the current TCP/IP stack remains usable for all other network interfaces, including loopback.

The architecture used by Chelsio essentially has two software components: the TCP Offload Module and the Offload driver.

**TCP Offload Module:** As mentioned earlier, the Linux operating system lacks support for TOE devices. Chelsio provides a framework of a TCP offload module (TOM) and a thin layer known as the *toedev* which decides whether a connection needs to be handed over to the TOM or to the traditional host-based TCP/IP stack. The TOM can be thought of as the upper layer of the TOE stack. It is responsible for implementing portions of TCP processing that cannot be done on the TOE (e.g., TCP TIME_WAIT processing). The state of all offloaded connections is also maintained by the TOM. Not all of the Linux network API calls (e.g., tcp_sendmsg, tcp_recvmsg) are compatible with offloading to the TOE. Such a requirement would result in extensive changes in the TCP/IP stack. To avoid this, the TOM implements its own subset of the transport layer API. TCP connections that are offloaded have certain function pointers redirected to the TOM's functions. Thus, non-offloaded connections can continue through the network stack normally.

**Offload Driver:** The offload driver is the lower layer of the TOE stack. It is directly responsible for manipulating the Terminator and its associated resources. TOEs have a many-to-one relationship with a TOM. A TOM can support multiple TOEs as long as it provides all functionality required by each. Each TOE can only be assigned one TOM. More than one driver may be associated with a single TOE device. If a TOE wishes to act as a normal Ethernet device (capable of only inputting/outputting Layer 2 level packets), a separate device driver may be required.

## 7.3 Experimental Evaluation

In this section, we evaluate the performance achieved by the Chelsio T110 10GigE adapter with TOE. In Section 7.3.1, we perform evaluations on the native sockets layer; in Section 7.3.2, we perform evaluations of the Message Passing Interface (MPI) stack atop the sockets interface; and in Section 7.3.3, we evaluate the Apache web server as an end application.

We used two clusters for the experimental evaluation. **Cluster 1** consists of two Opteron 248 nodes, each with a 2.2-GHz CPU along with 1 GB of 400-MHz DDR SDRAM and 1 MB of L2-Cache. These nodes are connected back-to-back with Chelsio T110 10GigE adapters with TOEs. **Cluster 2** consists of four Opteron 846 nodes, each with four 2.0-GHz CPUs (quad systems) along with 4 GB of 333-MHz DDR SDRAM and 1 MB of L2-Cache. It is connected with similar network adapters (Chelsio T110 10GigE-based TOEs) but via a 12-port Fujitsu XG1200 10GigE switch (with a latency of approximately 450 ns and capable of up to 240 Gbps of aggregate throughput). The experiments on both the clusters were performed with the SuSE Linux distribution installed with kernel.org kernel 2.6.6 (patched with Chelsio TCP Offload modules). In general, we have used Cluster 1 for all experiments requiring only two nodes and Cluster 2 for all experiments requiring more nodes. We will be pointing out the cluster used for each experiment throughout this section.

For optimizing the performance of the network adapters, we have modified several settings on the hardware as well as the software systems, e.g., (i) increased PCI burst size to 2 KB, (ii) increased send and receive socket buffer sizes to 512 KB each, and (iii) increased window size to 10 MB. Detailed descriptions about these optimizations and their impacts can be found in our previous work [52, 45, 15].

### 7.3.1 Sockets-level Evaluation

In this section, we evaluate the performance of the native sockets layer atop the TOEs as compared to the native host-based TCP/IP stack. We perform micro-benchmark level evaluations in two sub-categories. First, we perform evaluations based on a single connection measuring the point-to-point latency and uni-directional throughput together with the CPU utilization. Second, we perform evaluations based on multiple connections using the multi-stream, hot-spot, fan-in and fan-out tests.

#### Single Connection Micro-Benchmarks

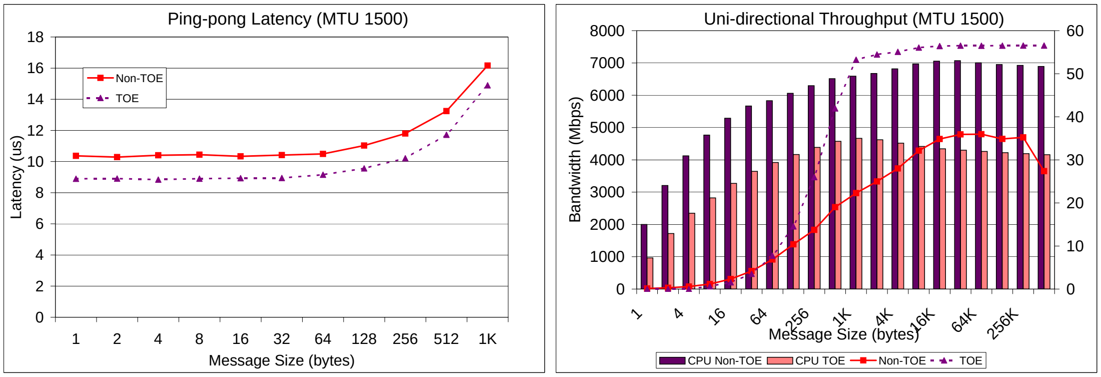

Figure 7.3: Sockets-level Micro-Benchmarks (MTU 1500): (a) Latency and (b) Throughput

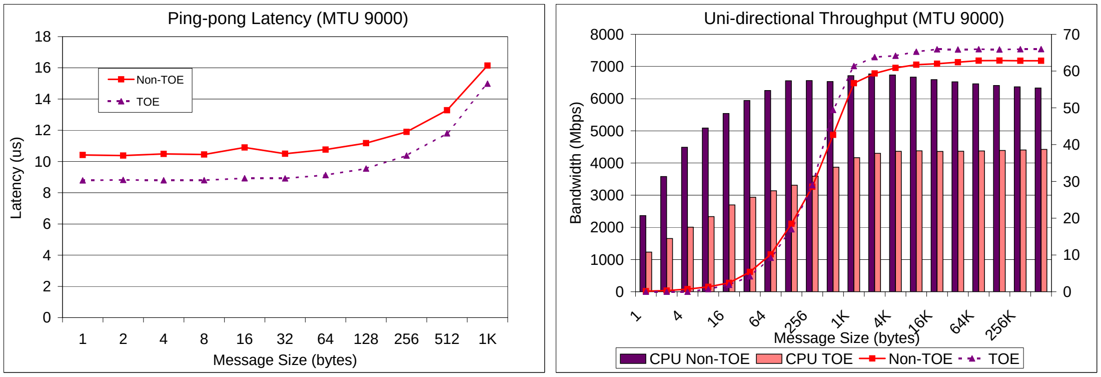

Figure 7.4: Sockets-level Micro-Benchmarks (MTU 9000): (a) Latency and (b) Throughput

Figures 7.3 and 7.4 show the basic single-stream performance of the 10GigE TOE as compared to the traditional host-based TCP/IP stack. All experiments in this section have been performed on Cluster 1 (described in Section 7.3).

Figure 7.3a shows that the TCP Offload Engines (TOE) can achieve a point-to-point latency of about 8.9 $\mu$s as compared to the 10.37 $\mu$s achievable by the host-based TCP/IP stack (non-TOE); an improvement of about 14.2%. Figure 7.3b shows the uni-directional throughput achieved by the TOE as compared to the non-TOE. As shown in the figure, the TOE achieves a throughput of up to 7.6 Gbps as compared to the 5 Gbps achievable by the non-TOE stack (improvement of about 52%). Throughput results presented throughout this chapter refer to the application data transferred per second and do not include the TCP/IP/Ethernet headers.

Increasing the MTU size of the network adapter to 9 KB (Jumbo frames) improves the performance of the non-TOE stack to 7.2 Gbps (Figure 7.4b). There is no additional improvement for the TOE due to the way it handles the message transmission.

For the TOE, the device driver hands over large message chunks (16 KB) to be sent out. The actual segmentation of the message chunk to MTU-sized frames is carried out by the network adapter. Thus, the TOE shields the host from the overheads associated with smaller MTU sizes. On the other hand, for the host-based TCP/IP stack (non-TOE), an MTU of 1500 bytes results in more segments and correspondingly more interrupts to be handled for every message causing a lower performance as compared to Jumbo frames.

We also show the CPU utilization for the different stacks. For TOE, the CPU remains close to 35% for large messages. However, for the non-TOE, the CPU utilization increases slightly on using jumbo frames. To understand this behavior, we reiterate on the implementation of these stacks. When the application calls a `write()` call, the host CPU copies the data into the socket buffer. If there is no space in the socket buffer, the CPU waits for the network adapter to complete sending out the existing data and creating space for the new data to be copied. Once the data is copied, the underlying TCP/IP stack handles the actual data transmission. Now, if the network adapter pushes the data out faster, space is created in the socket buffer faster and the host CPU spends a larger fraction of its time in copying data to the socket buffer than waiting for space to be created in the socket buffer. Thus, in general when the performance increases, we expect the host CPU to be spending a larger fraction of time copying data and burning CPU cycles. However, the usage of Jumbo frames reduces the CPU overhead for the host-based TCP/IP stack due to reduced number of interrupts. With these two conditions, on the whole, we see about a 10% increase in the CPU usage with Jumbo frames.

### Multiple Connection Micro-Benchmarks

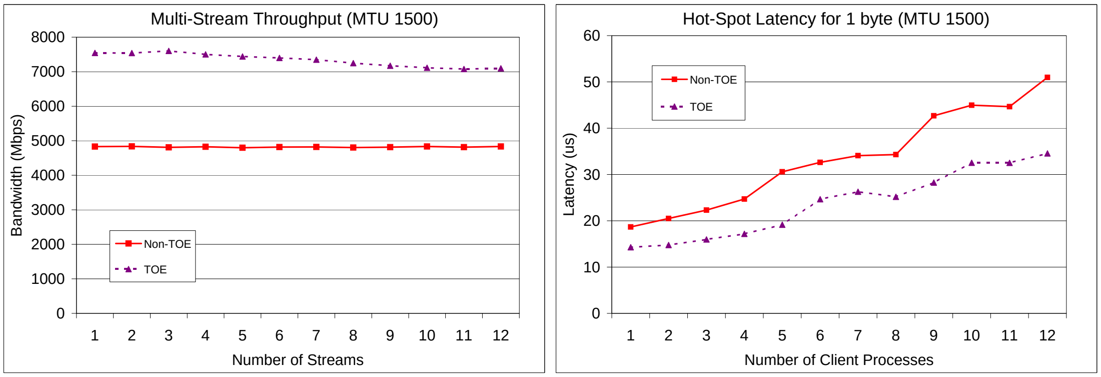

Figure 7.5: (a) Multi-stream Throughput and (b) Hot-Spot Latency

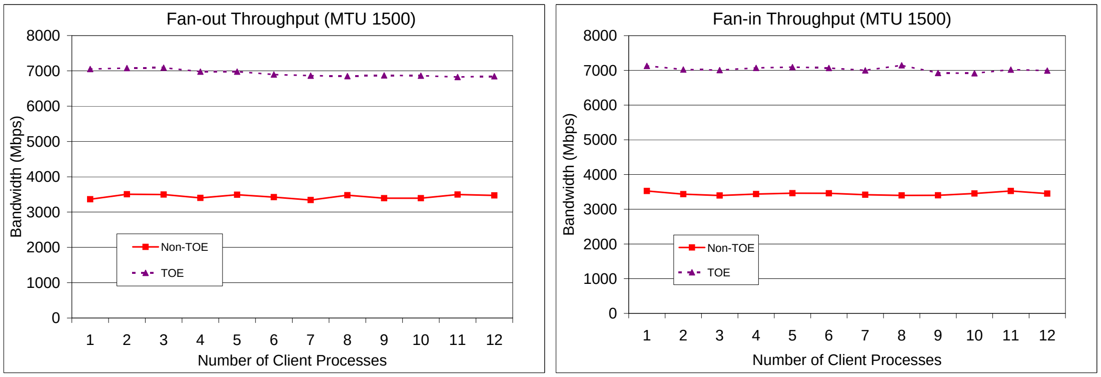

Figure 7.6: (a) Fan-out Test and (b) Fan-in Test

Here we evaluate the TOE and non-TOE stacks with micro-benchmarks utilizing multiple simultaneous connections. For all experiments in this section, we utilize an MTU of 1500 bytes in order to stick to the standard Ethernet frame size.

**Multi-stream Throughput Test:** Figure 8.4a shows the aggregate throughput achieved by two nodes (in Cluster 1) performing multiple instances of uni-directional throughput tests. We see that the TOE achieves a throughput of 7.1 to 7.6 Gbps. The non-TOE stack gets saturated at about 4.9 Gbps. These results are similar to the single stream results; thus using multiple simultaneous streams to transfer data does not seem to make much difference.

**Hot-Spot Latency Test:** Figure 8.4b shows the impact of multiple connections on small message transactions. In this experiment, a number of client nodes perform a point-to-point latency test with the same server forming a hot-spot on the server. We performed this experiment on Cluster 2 with one node acting as a server node and each of the other three 4-processor nodes hosting totally 12 client processes. The clients are alloted in a cyclic manner, so 3 clients refers to 1 client on each node, 6 clients refers to 2 clients on each node and so on. As seen in the figure, both the non-TOE as well as the TOE stacks show similar scalability with increasing number of clients, i.e., the performance difference seen with just one client continues with increasing number of clients. This shows that the look-up time for connection related data-structures is performed efficiently enough on the TOE and does not form a significant bottleneck.

**Fan-out and Fan-in Tests:** With the hot-spot test, we have shown that the lookup time for connection related data-structures is quite efficient on the TOE.

However, the hot-spot test does not stress the other resources on the network adapter such as management of memory regions for buffering data during transmission and reception. In order to stress such resources, we have designed two other tests namely fan-out and fan-in. In both these tests, one server process carries out uni-directional throughput tests simultaneously with a number of client threads (performed on Cluster 2). The difference being that in a fan-out test the server pushes data to the different clients (stressing the transmission path on the network adapter) and in a fan-in test the clients push data to the server process (stressing the receive path on the network adapter). Figure 8.5 shows the performance of the TOE stack as compared to the non-TOE stack for both these tests. As seen in the figure, the performance for both the fan-out and the fan-in tests is quite consistent with increasing number of clients suggesting an efficient transmission and receive path implementation.

### 7.3.2 MPI-level Evaluation

In this section, we evaluate the Message Passing Interface (MPI) stack written using the sockets interface on the TOE and non-TOE stacks. MPI is considered the *de facto* standard programming model for scientific applications; thus this evaluation would allow us to understand the implications of the TOE stack for such applications. We used the LAM [28] implementation of MPI for this evaluation.

Figure 7.7 illustrates the point-to-point latency and uni-directional throughput achievable with the TOE and non-TOE stacks for an MTU size of 1500 bytes. As shown in Figure 7.7a, MPI over the TOE stack achieves a latency of about 10.2 $\mu$s compared to the 12.2 $\mu$s latency achieved by the non-TOE stack. The increased point-to-point latency of the MPI stack as compared to that of the native sockets layer (8.9 $\mu$s) is attributed to the overhead of the MPI implementation. Figure 7.7b shows the uni-directional throughput achieved by the two stacks. TOE achieves a throughput of about 6.9 Gbps as compared to the 3.1 Gbps achieved by the non-TOE stack.

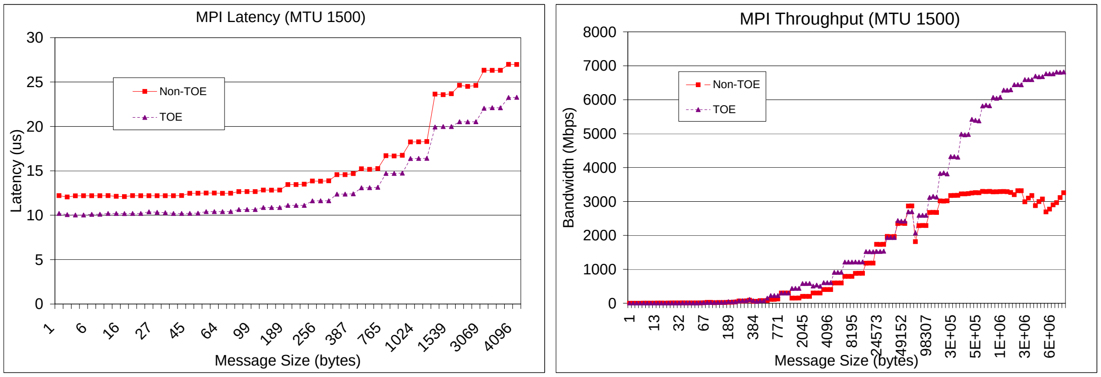

Figure 7.7: MPI-level Micro-Benchmarks (MTU 1500): (a) Latency and (b) Throughput

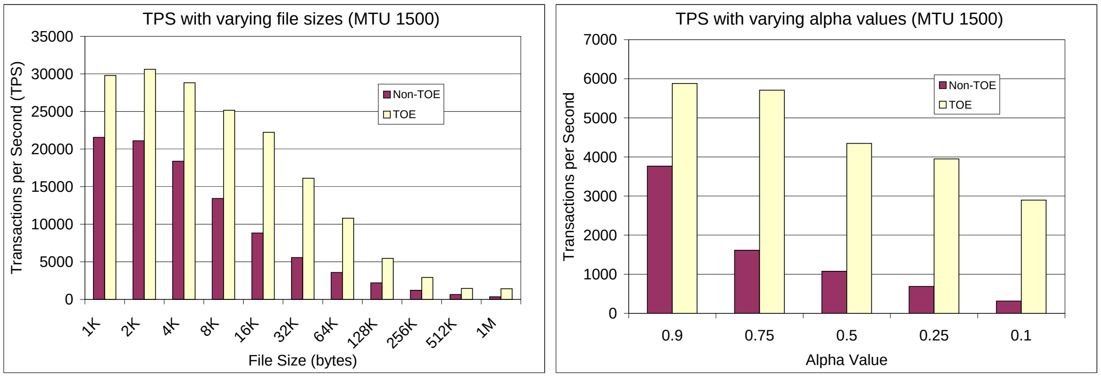

Figure 7.8: Apache Web-Server Performance: (a) Single File Traces and (b) Zipf Traces

### 7.3.3 Application-level Evaluation

In this section, we evaluate the performance of the stacks using a real application, namely the Apache web server. One node is used as a web-server and three nodes to host up to 24 client processes.

In the first experiment (Figure 7.8a), we use a simulated trace consisting of only one file. Evaluating the stacks with various sizes for this file lets us understand their performance without being diluted by other system parameters. As seen in the figure, the TOE achieves a significantly better performance as compared to the non-TOE especially for large files. In the next experiment (Figure 7.8b), we build a trace based on the popular Zipf [85] file request distribution. The Zipf distribution states that the probability of requesting the $I^{th}$ most popular document is inversely proportional to a constant power $\alpha$ of $I$. $\alpha$ denotes the temporal locality in the trace (close to *one* represents a high temporal locality). We used the World-Cup trace [9] to associate file sizes with the Zipf pattern; like several other traces, this trace associates small files to be the most popular ones while larger files tend to be less popular. Thus, when the $\alpha$ value is very close to *one*, a lot of small files tend to be accessed and when the $\alpha$ value becomes smaller, the requests are more spread out to the larger files as well. Accordingly, the percentage improvement in performance for the TOE seems to be lesser for high $\alpha$ values as compared to small $\alpha$ values.

## 7.4 Summary

In this chapter, we presented a detailed performance evaluation of the Chelsio T110 10GigE adapter with TOE. We have performed evaluations in three categories:

(i) detailed micro-benchmark level evaluation of the native sockets layer, (ii) evaluation of the Message Passing Interface (MPI) stack over the sockets interface, and (iii) application-level evaluation of the Apache web server. These experimental evaluations provide several useful insights into the effectiveness of the TOE stack in scientific as well as commercial domains.

# CHAPTER 8

# HEAD-TO-TOE EVALUATION OF HIGH-PERFORMANCE SOCKETS OVER PROTOCOL OFFLOAD ENGINES

Many researchers, including ourselves, have evaluated the benefits of sockets over offloaded protocol stacks on various networks including IBA and Myrinet. However, to our best knowledge, there has been no work that compares and contrasts the capabilities and limitations of these technologies with the recently introduced 10GigE TOEs on a homogeneous experimental testbed. In this chapter, we perform several evaluations to enable a coherent comparison between 10GigE, IBA and Myrinet with respect to the sockets interface. In particular, we evaluate the networks at two levels: (i) a detailed micro-benchmark evaluation and (ii) an application-level evaluation with sample applications from multiple domains, including a bio-medical image visualization tool known as the Virtual Microscope [8], an iso-surface oil reservoir simulator called Iso-Surface [23], a cluster file-system known as the Parallel Virtual File-System (PVFS) [67], and a popular cluster management tool named Ganglia [1]. In addition to 10GigE's advantage with respect to compatibility to wide-area network infrastructures, e.g., in support of grids, our results show that 10GigE also delivers performance that is comparable to traditional high-speed network technologies such as IBA and Myrinet in a system-area network environment to support clusters and that 10GigE is particularly well-suited for sockets-based applications.

## 8.1 Interfacing with POEs

Since the Linux kernel does not currently support Protocol Offload Engines (POEs), researchers have taken a number of approaches to enable applications to interface with POEs. The two predominant approaches are high-performance sockets implementations such as the Sockets Direct Protocol (SDP) and TCP Stack Override. In this section, we will discuss the TCP stack override approach. The high-performance sockets approach is discussed in the previous chapters and is skipped here.

### 8.1.1 TCP Stack Override

This approach retains the kernel-based sockets layer. However, the TCP/IP stack is overridden and the data is pushed directly to the offloaded protocol stack in order to bypass the host TCP/IP stack implementation (see Figure 8.1b). The Chelsio T110 adapter studied in this chapter follows this approach. The software architecture used by Chelsio essentially has two components: the TCP offload module (TOM) and the offload driver.

**TCP Offload Module:** As mentioned earlier, the Linux operating system lacks support for TOE devices. Chelsio provides a framework of a TCP offload module (TOM) and a thin layer known as the *toedev* which decides whether a connection needs to be handed over to the TOM or to the traditional host-based TCP/IP stack. The TOM can be thought of as the upper layer of the TOE stack. It is responsible for implementing portions of TCP processing that cannot be done on the TOE. The state of all offloaded connections is also maintained by the TOM. Not all of the Linux

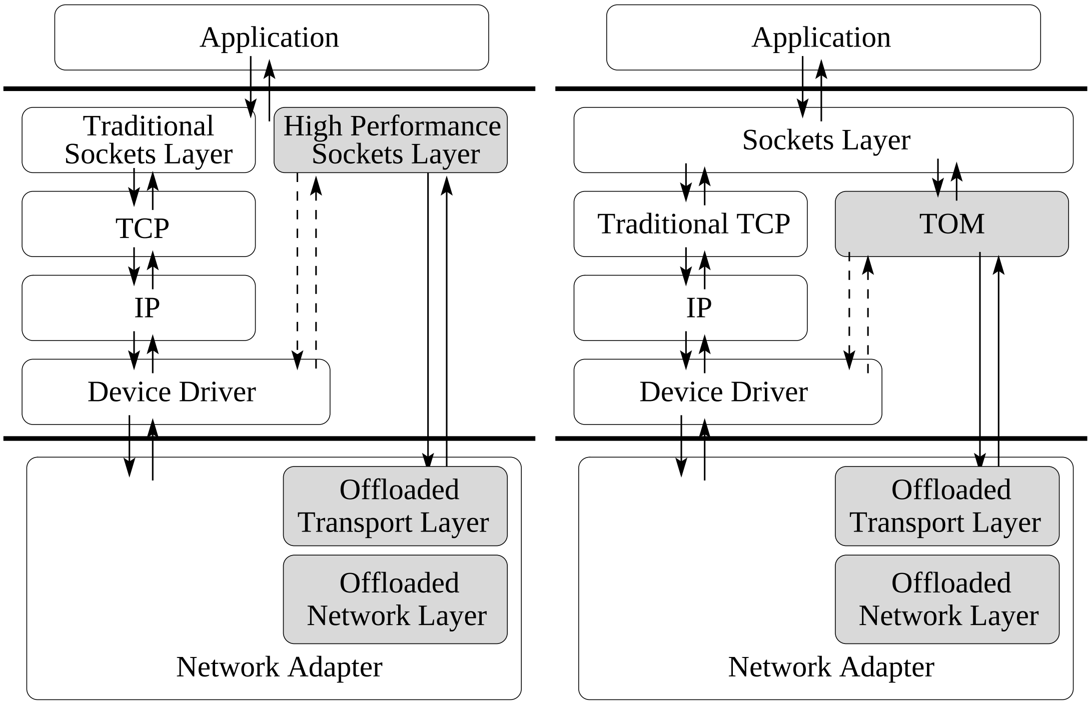

Figure 8.1: Interfacing with POEs: (a) High Performance Sockets and (b) TCP Stack Override

network API calls (e.g., tcp_sendmsg, tcp_recvmsg) are compatible with the TOE. Modifying these would result in extensive changes in the TCP/IP stack. To avoid this, the TOM implements its own subset of the transport-layer API. TCP connections that are offloaded have certain function pointers redirected to the TOM's functions. Thus, non-offloaded connections can continue through the network stack normally.

**Offload Driver:** The offload driver is the lower layer of the TOE stack. It is directly responsible for manipulating the terminator and its associated resources. TOEs have a many-to-one relationship with a TOM. A TOM can support multiple TOEs as long as it provides all the functionality required by each. Each TOE can only be assigned one TOM. More than one driver may be associated with a single

TOE device. If a TOE wishes to act as a normal Ethernet device (capable of handling only Layer 2 packets), a separate device driver may be required.

## 8.2 Experimental Testbed

For experimentally evaluating the performance of the three networks, we used the following testbed: a cluster of four nodes built around SuperMicro SUPER X5DL8-GG motherboards with ServerWorks GC LE chipsets, which include 64-bit, 133-MHz PCI-X interfaces. Each node has two Intel Xeon 3.0 GHz processors with a 512-kB L2 cache and a 533-MHz front-side bus and 2 GB of 266-MHz DDR SDRAM. We used the RedHat 9.0 Linux distribution and the Linux-2.4.25smp kernel.org kernel. Each node was equipped with the 10GigE, IBA and Myrinet networks. The 32-bit Xeon processors and the 2.4 kernel used in the testbed represent a large installation base; thus, the results described here would be most relevant for researchers using such testbeds to weigh the pros and cons of each network before adopting them.

**10GigE:** The 10GigE network was based on Chelsio T110 10GigE adapters with TOEs connected to a 16-port SuperX Foundry switch. The driver version used on the network adapters is 1.2.0, and the firmware on the switch is version 2.2.0. For optimizing the performance of the 10GigE network, we have modified several settings on the hardware as well as the software systems, e.g., (i) increased PCI burst size to 2 KB, (ii) increased send and receive socket buffer sizes to 512 KB each, (iii) increased window size to 10 MB and (iv) enabled hardware flow control to minimize packet drops on the switch. Detailed descriptions about these optimizations and their impact can be found in our previous work [52, 45, 15].

**InfiniBand:** The InfiniBand (IBA) network was based on Mellanox InfiniHost MT23108 dual-port 4x HCA adapters through an InfiniScale MT43132 twenty-four port completely non-blocking switch. The adapter firmware version is fw-23108-rel-3_2_0-rc4-build-001 and the software stack was based on the Voltaire IBHost-3.0.0-16 stack.

**Myrinet:** The Myrinet network was based on Myrinet-2000 'E' (dual-port) adapters connected by a Myrinet-2000 wormhole router crossbar switch. Each adapter is capable of a 4Gbps theoretical bandwidth in each direction. For SDP/Myrinet, we performed evaluations with two different implementations. The first implementation is using the GM/Myrinet drivers (SDP/GM v1.7.9 over GM v2.1.9). The second implementation is over the newly released MX/Myrinet drivers (SDP/MX v1.0.2 over MX v1.0.0). The SDP/MX implementation is a very recent release by Myricom (the vendor for Myrinet) and achieves a significantly better performance than the older SDP/GM. However, as a part-and-parcel of being a bleeding-edge implementation, SDP/MX comes with its share of stability issues; due to this, we had to restrict the evaluation of some of the experiments to SDP/GM alone. Specifically, we present the ping-pong latency, uni-directional and bi-directional bandwidth results (in Section 8.3.1) for both SDP/MX as well as SDP/GM and the rest of the results for SDP/GM alone. With the current active effort from Myricom towards SDP/MX, we expect these stability issues to be resolved very soon and the numbers for Myrinet presented in this section to further improve.

## 8.3 Micro-Benchmark Evaluation

In this section, we perform micro-benchmark evaluations of the three networks over the sockets interface. We perform evaluations in two sub-categories. First, we perform evaluations based on a single connection measuring the point-to-point latency, uni-directional bandwidth, and the bi-directional bandwidth. Second, we perform evaluations based on multiple connections using the multi-stream bandwidth test, hot-spot test, and fan-in and fan-out tests. In Section 8.4 we extend this evaluation to real-life applications from various domains.

### 8.3.1 Single Connection Micro-Benchmarks

Figures 8.2 and 8.3 show the basic single-connection performance of the 10GigE TOE as compared to SDP/IBA and SDP/Myrinet (both SDP/MX/Myrinet and SDP/GM/Myrinet).

**Ping-Pong Latency Micro-Benchmark:** Figures 8.2a and 8.2b show the comparison of the ping-pong latency for the different network stacks.

IBA and Myrinet provide two kinds of mechanisms to inform the user about the completion of data transmission or reception, namely polling and event-based. In the polling approach, the sockets implementation has to continuously poll on a predefined location to check whether the data transmission or reception has completed. This approach is good for performance but requires the sockets implementation to continuously monitor the data-transfer completions, thus requiring a huge amount of CPU resources. In the event-based approach, the sockets implementation requests the network adapter to inform it on a completion and sleeps. On a completion event,

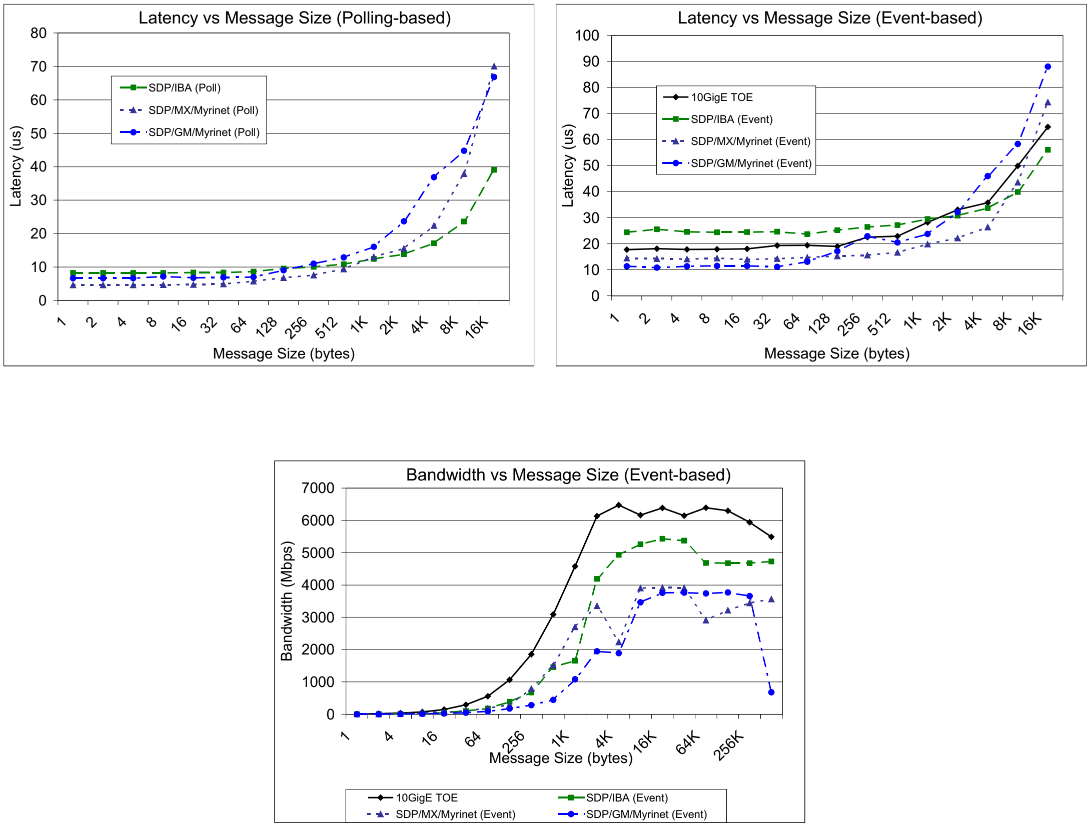

Figure 8.2: Single Connection Micro-Benchmarks: (a) Latency (polling-based), (b) Latency (event-based) and (c) Uni-directional Bandwidth (event-based)

the network adapter wakes this process up through an interrupt. While this approach is more efficient in terms of the CPU required since the application does not have to continuously monitor the data transfer completions, it incurs an additional cost of the interrupt. In general, for single-threaded applications the polling approach is the most efficient while for most multi-threaded applications the event-based approach turns out to perform better. Based on this, we show two implementations of the SDP/IBA and SDP/Myrinet stacks, viz., event-based (Figure 8.2a) and polling-based (Figure 8.2b); the 10GigE TOE supports only the event-based approach.

As shown in the figures, SDP/Myrinet achieves the lowest small-message latency for both the polling as well as event-based models. For the polling-based models, SDP/MX/Myrinet and SDP/GM/Myrinet achieve latencies of 4.64$\mu$s and 6.68$\mu$s respectively, compared to a 8.25$\mu$s achieved by SDP/IBA. For the event-based models, SDP/MX/Myrinet and SDP/GM/Myrinet achieve latencies of 14.47$\mu$s and 11.33$\mu$s, compared to the 17.7$\mu$s and 24.4$\mu$s achieved by 10GigE and SDP/IBA, respectively. However, as shown in the figure, for medium-sized messages (larger than 2 kB for event-based and 4 kB for polling-based), the performance of SDP/Myrinet deteriorates. For messages in this range, SDP/IBA performs the best followed by the 10GigE TOE, and the two SDP/Myrinet implementations, respectively. We should note that the Foundry SuperX 10GigE switch that we used has approximately a 4.5-$\mu$s flow-through latency, which is amazing for a store-and-forward switch. For the virtual cut-through based Fujitsu XG1200 switch, however, the flow-through latency is only 0.5 $\mu$s, resulting in a 10GigE end-to-end latency of only 13.7 $\mu$s.

**Unidirectional Bandwidth Micro-Benchmark:** For the uni-directional bandwidth test, the 10GigE TOE achieves the highest bandwidth at close to 6.4 Gbps compared to the 5.4 Gbps achieved by SDP/IBA and the 3.9 Gbps achieved by the SDP/Myrinet implementations[3]. The results for both event- and polling-based approaches are similar; thus, we only present the event-based numbers here. The drop in the bandwidth for SDP/GM/Myrinet at 512-kB message size, is attributed to the high dependency of the implementation of SDP/GM/Myrinet on L2-cache activity. Even 10GigE TOE shows a slight drop in performance for very large messages, but not as drastically as SDP/GM/Myrinet. Our systems use a 512-KB L2-cache and a relatively slow memory (266-MHz DDR SDRAM) which causes the drop to be significant. For systems with larger L2-caches, L3-caches, faster memory speeds or better memory architectures (e.g., NUMA), this drop can be expected to be smaller. Further, it is to be noted that the bandwidth for all networks is the same irrespective of whether a switch is used or not; thus the switches do not appear to be a bottleneck for single-stream data transfers.

**Bidirectional Bandwidth Micro-Benchmark:** Similar to the unidirectional bandwidth test, the 10GigE TOE achieves the highest bandwidth (close to 7 Gbps) followed by SDP/IBA at 6.4 Gbps and both SDP/Myrinet implementations at about 3.5 Gbps. 10GigE TOE and SDP/IBA seem to perform quite poorly with respect to the theoretical peak throughput achievable (20Gbps bidirectional). This is attributed to the PCI-X buses to which these network adapters are connected. The PCI-X bus (133 MHz/64 bit) is a shared network I/O bus that allows only a theoretical peak of 8.5 Gbps for traffic in both directions. Further, as mentioned earlier, the memory used in our systems is relatively slow (266-MHz DDR SDRAM). These, coupled with

[3] On the Opteron platform, 10GigE achieves up to 7.6Gbps; we expect an improved performance for the other networks as well. However, due to limitations in our current test-bed, we could not perform this comparison on the Opteron platform. Further, with 32-bit Xeons being the largest installation base today, we feel that the presented numbers might be more relevant to the community.

the header and other traffic overheads, causes these networks to be saturated much below the theoretical bandwidth that the network can provide. For SDP/Myrinet, we noticed that both the implementations are quite unstable and have not provided us with much success in getting performance numbers for message sizes larger than 64KB. Also, the peak bandwidth achievable is only 3.5 Gbps which is actually less than the unidirectional bandwidth that these implementations provide.

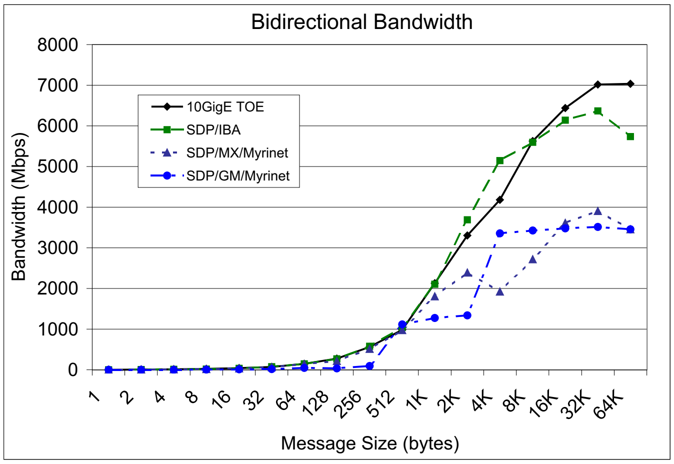

Figure 8.3: Bi-directional Bandwidth

### 8.3.2 Multiple Connection Micro-Benchmarks

As mentioned earlier, due to stability reasons, we have not been able to evaluate the performance of several benchmarks with SDP/MX/Myrinet. Hence, for the benchmarks and applications presented in this Section and Section 8.4, we present evaluations only with SDP/GM for the Myrinet network.

Figures 8.4 and 8.5 show the multi-connection experiments performed with the three networks. These experiments demonstrate scenarios where either a single process or multiple processes on the same physical node open a number of connections. These tests are designed to understand the performance of the three networks in scenarios where the network has to handle several connections simultaneously.

It is to be noted that for multi-threaded applications the polling-based approach performs very badly due to its high CPU usage; therefore these results are not shown in this chapter, and we stick to only the event-based approach for these applications.

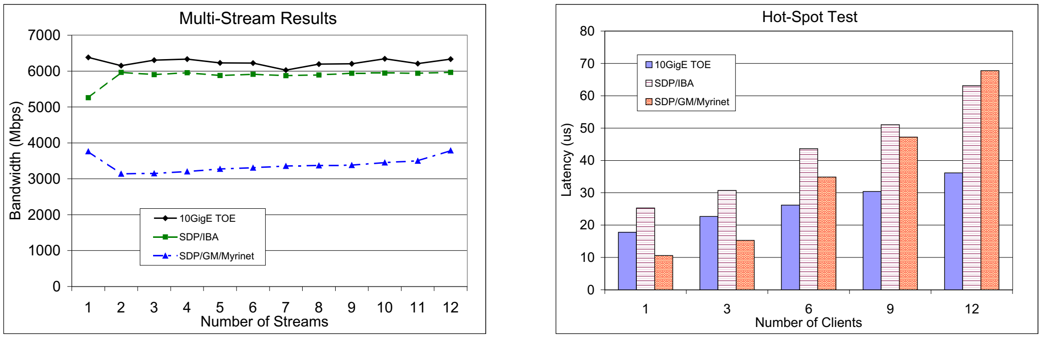

Figure 8.4: Multi-Connection Micro-Benchmarks: (a) Multi-Stream Bandwidth and (b) Hot-Spot Latency

**Multi-Stream Bandwidth:** Figure 8.4a illustrates the aggregate throughput achieved by two nodes performing multiple instances of uni-directional throughput tests. Because the performance of SDP/GM/Myrinet seems to be a little inconsistent, it is difficult to characterize the performance of Myrinet with respect to the other networks, but we have observed that SDP/GM/Myrinet generally achieves a throughput of about 3.15 to 3.75 Gbps. 10GigE TOE and SDP/IBA, on the other hand, quite consistently achieve throughputs around 5.9 to 6.2 Gbps with 10GigE performing slightly better most of the time.

**Hot-Spot Latency:** Figure 8.4b shows the impact of multiple connections on small-message transactions. In this experiment, a number of client nodes perform a point-to-point latency test with the same server forming a hot-spot on the server. We performed this experiment with one node acting as a server node and the other three dual-processor nodes hosting a total of 12 client processes. The clients are alloted in a cyclic manner, so three clients refers to having one client process on each of the three nodes, six clients refers to having two client processes on each of the three nodes, and so on. As shown in the figure, SDP/GM/Myrinet performs the best when there is just one client followed by 10GigE TOE and SDP/IBA, respectively. However, as the number of clients increase, 10GigE TOE and SDP/IBA scale quite well while the performance of SDP/GM/Myrinet deteriorates significantly; for 12 clients, for example, SDP/GM/Myrinet provides the worst performance of the three while the 10GigE TOE performs significantly better than the other two. This shows that the lookup time for connection-related data structures is performed efficiently enough on the 10GigE TOE and SDP/IBA implementations and that they scale quite well with an increasing number of connections.

**Fan-Out and Fan-In tests:** With the hot-spot test, we have shown that the lookup time for connection-related data structures is quite efficient on the 10GigE TOE and SDP/IBA implementations. However, the hot-spot test does not stress the other resources on the network adapter such as management of memory regions for buffering data during transmission and reception. In order to stress such resources, we have designed two other tests, namely fan-out and fan-in. In both these tests, one server process carries out unidirectional throughput tests simultaneously with a number of client threads. The difference being that in a fan-out test, the server pushes data to the different clients (stressing the transmission path in the implementation), and in a fan-in test, the clients push data to the server process (stressing the receive path in the implementation). Figure 8.5 shows the performance of the three networks for both these tests. As shown in the figure, for both the tests, SDP/IBA and SDP/GM/Myrinet scale quite well with increasing number of clients. 10GigE TOE, on the other hand, performs quite well for the fan-in test; however, we see a slight drop in its performance for the fan-out test with increasing clients.

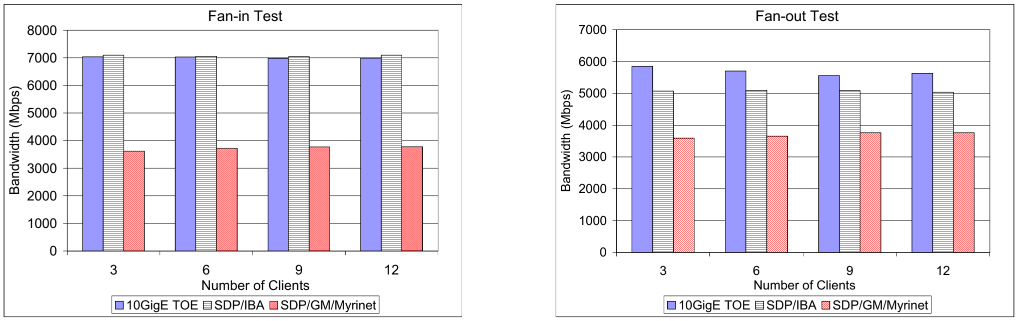

Figure 8.5: Multi-Connection Micro-Benchmarks: (a) Fan-in and (b) Fan-out

## 8.4 Application-Level Evaluation

In this section, we evaluate the performance of different applications across the three network technologies. Specifically, we evaluate a bio-medical image visualization tool known as the Virtual Microscope, an iso-surface oil reservoir simulator called Iso-Surface, a cluster file-system known as the Parallel Virtual File-System (PVFS), and a popular cluster management tool named Ganglia.

### 8.4.1 Data-Cutter Overview and Evaluation

Data-Cutter is a component-based framework [22, 32, 66, 70] that has been developed by the University of Maryland in order to provide a flexible and efficient run-time environment for data-intensive applications on distributed platforms. The Data-Cutter framework implements a filter-stream programming model for developing data-intensive applications. In this model, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. Filters are connected via *logical streams.* A *stream* denotes a unidirectional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only. The implementation of the logical stream uses the sockets interface for point-to-point stream communication. The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. When a filter group is instantiated to process an application query, the run-time system establishes socket connections between filters placed on different hosts before starting the execution of the application query. Filters placed on the same host execute as separate threads. An application query is handled as a *unit of work* (UOW) by the filter group. An example is a visualization of a dataset from a viewing angle. The processing of a UOW can be done in a pipelined fashion; different filters can work on different data elements simultaneously, as shown in Figure 8.6.

Several data-intensive applications have been designed and developed using the data-cutter run-time framework. In this chapter, we use two such applications, namely the Virtual Microscope (VM) and the Iso-Surface oil-reservoir simulation (ISO) application, for evaluation purposes.

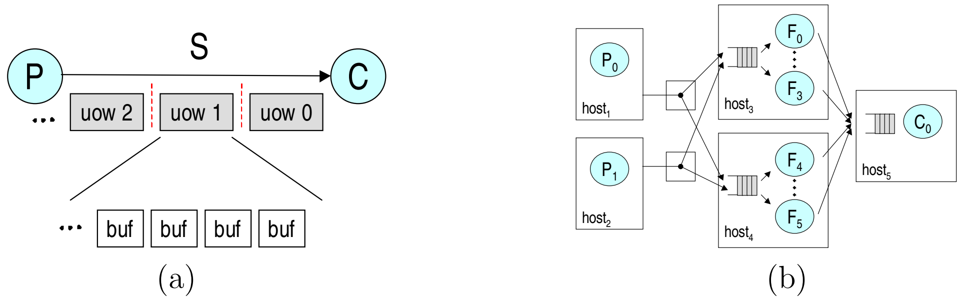

Figure 8.6: Data-Cutter stream abstraction and support for copies. (a) Data buffers and end-of-work markers on a stream. (b) P,F,C filter group instantiated using transparent copies.

*Virtual Microscope (VM):* VM is a data-intensive digitized microscopy application. 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 [8, 31]. The main difficulty stems from the handling of large volumes of image data, which can range from a few hundreds of megabytes (MB) to several gigabytes (GB) 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.

*Iso-Surface Oil-Reservoir Simulation (ISO):* Computational models for seismic analysis of oil reservoirs simulate the seismic properties of a reservoir by using output from oil-reservoir simulations. The main objective of oil-reservoir modeling is to understand the reservoir properties and predict oil production to optimize return on investment from a given reservoir, while minimizing environmental effects. This application demonstrates a dynamic, data-driven approach to solve optimization problems in oil-reservoir management. Output from seismic simulations are analyzed to investigate the change in geological characteristics of reservoirs. The output is also processed to guide future oil-reservoir simulations. Seismic simulations produce output that represents the traces of sound waves generated by sound sources and recorded by receivers on a three-dimensional grid over many time steps. One analysis of seismic datasets involves mapping and aggregating traces onto a 3-dimensional volume through a process called seismic imaging. The resulting three-dimensional volume can be used for visualization or to generate input for reservoir simulations.

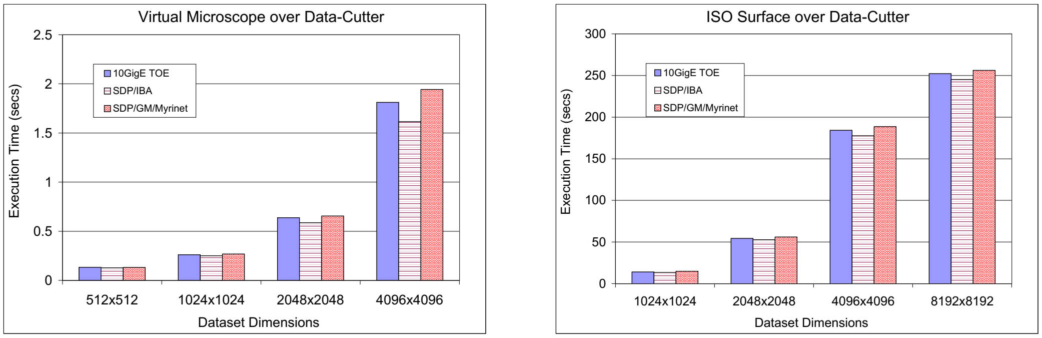

Figure 8.7: Data-Cutter Applications: (a) Virtual Microscope (VM) and (b) ISO-Surface (ISO)

**Evaluating Data-Cutter:** Figure 8.7a compares the performance of the VM application over each of the three networks (10GigE, IBA, Myrinet). As shown in the figure, SDP/IBA outperforms the other two networks. This is primarily attributed to the worse latency for medium-sized messages for 10GigE TOE and SDP/GM/Myrinet (shown in Figure 8.2a). Though the VM application deals with large datasets (each image was about 16MB), the dataset is broken down into small Unit of Work (UOW) segments that are processed in a pipelined manner. This makes the application sensitive to the latency of medium-sized messages resulting in better performance for SDP/IBA compared to 10GigE TOE and SDP/GM/Myrinet.

Figure 8.7b compares the performance of the ISO application for the three networks. The dataset used was about 64 MB in size. Again, the trend with respect to the performance of the networks remains the same with SDP/IBA outperforming the other two networks.

### 8.4.2 PVFS Overview and Evaluation

Parallel Virtual File System (PVFS) [67], is one of the leading parallel file systems for Linux cluster systems today, developed jointly by Clemson University and Argonne National Lab. It was designed to meet the increasing I/O demands of parallel applications in cluster systems. Typically, a number of nodes in the cluster system are configured as I/O servers and one of them (either an I/O server or a different node) as a metadata manager. Figure 8.8 illustrates a typical PVFS environment.

PVFS achieves high performance by striping files across a set of I/O server nodes, allowing parallel accesses to the data. It uses the native file system on the I/O servers to store individual file stripes. An I/O daemon runs on each I/O node and services requests from the compute nodes, in particular the read and write requests. Thus, data is transferred directly between the I/O servers and the compute nodes. A manager daemon runs on a metadata manager node. It handles metadata operations involving file permissions, truncation, file stripe characteristics, and so on. Metadata is also stored on the local file system. The metadata manager provides a cluster-wide consistent name space to applications. In PVFS, the metadata manager does not

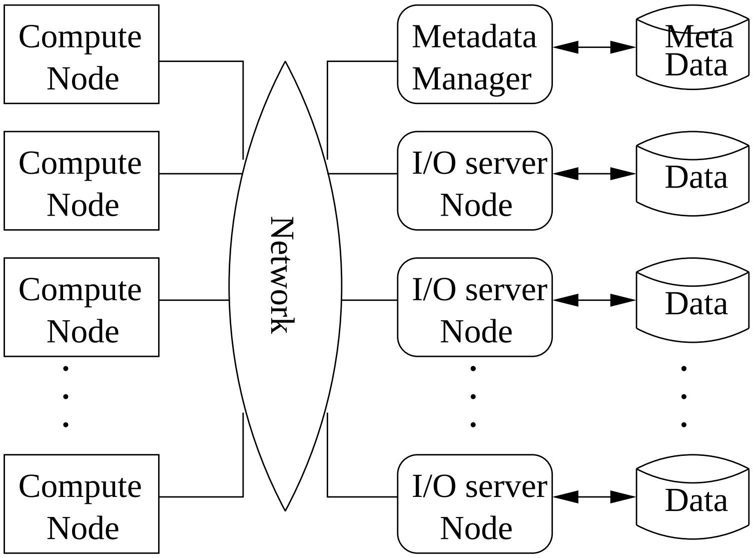

Figure 8.8: A Typical PVFS Setup

participate in read/write operations. PVFS supports a set of feature-rich interfaces, including support for both contiguous and noncontiguous accesses to both memory and files. PVFS can be used with multiple APIs: a native API, the UNIX/POSIX API, MPI-IO, and an array I/O interface called Multi- Dimensional Block Interface (MDBI). The presence of multiple popular interfaces contributes to the wide success of PVFS in the industry.

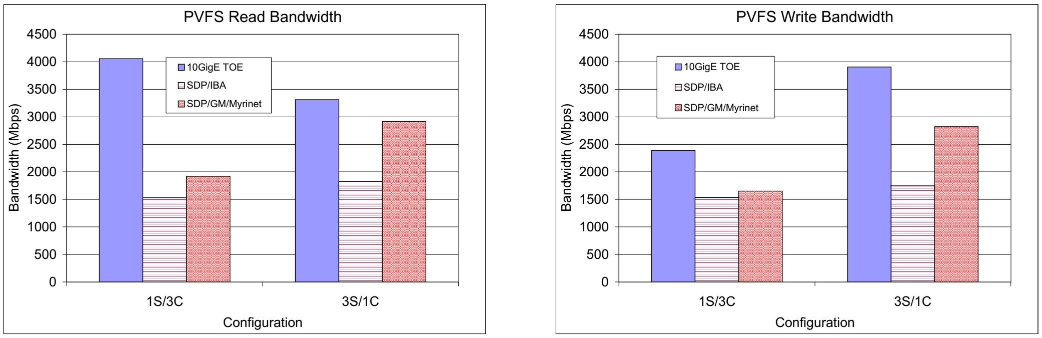

Figure 8.9: Concurrent PVFS Read/Write

**Performance of Concurrent File I/O:** In this test, we evaluate the performance of PVFS concurrent read/write operations using the *pvfs-test* program from the standard PVFS releases. For this test, an MPI program is used to parallelize file write/read access of contiguous 2-MB data buffers from each compute node. The native PVFS library interface is used in this test, more details of this program can be found in [67].

Figure 8.9 shows PVFS file read and write performance on the different networks. We perform two kinds of tests for both read and write. In the first test, we use just one server; three clients simultaneously read or write a file from/to this server. In the second test, we use three servers and stripe the file across all three servers; a single client reads or writes the stripes from all three servers simultaneously. These two tests are represented as legends "1S/3C" (representing one server and three clients) and "3S/1C" (representing three servers and one client), respectively. As shown in the figure, the 10GigE TOE considerably outperforms the other two networks in both the tests for read as well as write. This follows the same trend as shown by the basic bandwidth and fan-in/fan-out micro-benchmark results in Figures 8.2b and 8.5. SDP/IBA, however, seems to achieve considerably lower performance as compared to even SDP/GM/Myrinet (which has a much lower theoretical bandwidth: 4 Gbps compared to the 10 Gbps of IBA).

**Performance of MPI-Tile-IO:** MPI-Tile-IO [72] is a tile-reading MPI-IO application. It tests the performance of tiled access to a two-dimensional dense dataset, simulating the type of workload that exists in some visualization applications and numerical applications. In our experiments, two nodes are used as server nodes and the other two as client nodes running MPI-tile-IO processes. Each process renders a

1 × 2 array of displays, each with 1024 × 768 pixels. The size of each element is 32 bytes, leading to a file size of 48 MB.

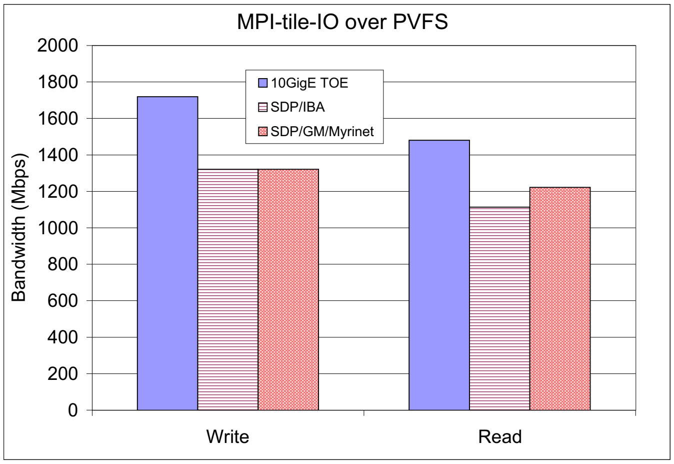

Figure 8.10: MPI-Tile-IO over PVFS

We evaluate both the read and write performance of MPI-Tile-IO over PVFS. As shown in Figure 8.10, the 10GigE TOE provides considerably better performance than the other two networks in terms of both read and write bandwidth. Another interesting point to be noted is that the performance of all the networks is considerably worse in this test versus the concurrent file I/O test; this is due to the non-contiguous data access pattern of the MPI-tile-IO benchmark which adds significant overhead.

### 8.4.3 Ganglia Overview and Evaluation

Ganglia [1] is an open-source project that grew out of the UC-Berkeley Millennium Project. It is a scalable distributed monitoring system for high-performance computing systems such as clusters and grids. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and RRDtool for data storage and visualization. It uses carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency.

The Ganglia system comprises of two portions. The first portion comprises of a server monitoring daemon which runs on each node of the cluster and occasionally monitors the various system parameters including CPU load, disk space, memory usage and several others. The second portion of the Ganglia system is a client tool which contacts the servers in the clusters and collects the relevant information. Ganglia supports two forms of global data collection for the cluster. In the first method, the servers can communicate with each other to share their respective state information, and the client can communicate with any one server to collect the global information. In the second method, the servers just collect their local information without communication with other server nodes, while the client communicates with each of the server nodes to obtain the global cluster information. In our experiments, we used the second approach.

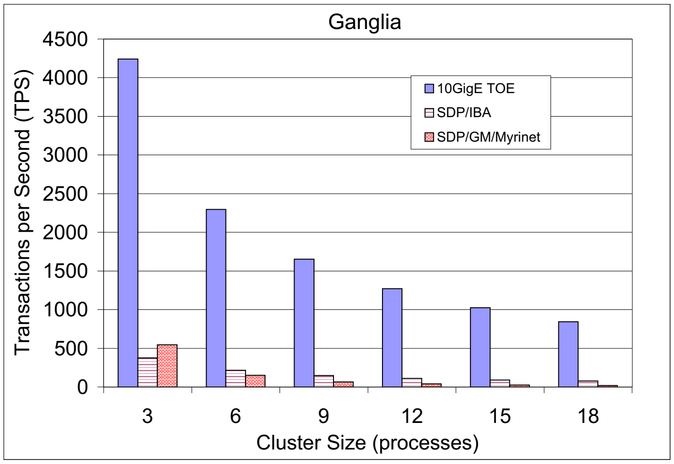

Figure 8.11: Ganglia: Cluster Management Tool

**Evaluating Ganglia:** Figure 8.11 shows the performance of Ganglia for the different networks. As shown in the figure, the 10GigE TOE considerably outperforms the other two networks by up to a factor of 11 in some cases. To understand this performance difference, we first describe the pattern in which Ganglia works. The client node is an end node which gathers all the information about all the servers in the cluster and displays it to the end user. In order to collect this information, the client opens a connection with each node in the cluster and obtains the relevant information (ranging from 2 KB to 10 KB) from the nodes. Thus, Ganglia is quite sensitive to the connection time and medium-message latency.

As we had seen in Figures 8.2a and 8.2b, 10GigE TOE and SDP/GM/Myrinet do not perform very well for medium-sized messages. However, the connection time for 10GigE is only about 60$\mu$s as compared to the *millisecond range* connection times for SDP/GM/Myrinet and SDP/IBA. During connection setup, SDP/GM/Myrinet and SDP/IBA pre-register a set of buffers in order to carry out the required communication; this operation is quite expensive for the Myrinet and IBA networks since it involves informing the network adapters about each of these buffers and the corresponding protection information. This coupled with other overheads, e.g., state transitions (INIT to RTR to RTS) that are required during connection setup for IBA, increase the connection time tremendously for SDP/IBA and SDP/GM/Myrinet. All in all, the connection setup time dominates the performance of Ganglia in our experiments, resulting in much better performance for the 10GigE TOE.

## 8.5 Summary

Traditional Ethernet-based network architectures such as Gigabit Ethernet (GigE) have delivered significantly worse performance than other high-performance networks [e.g, InfiniBand (IBA), Myrinet]. In spite of this performance difference, the low cost of the network components and their backward compatibility with the existing Ethernet infrastructure have allowed GigE-based clusters to corner 42% of the Top500 Supercomputer List. With the advent of 10GigE and TCP Offload Engines (TOEs), we demonstrated that the aforementioned performance gap can largely be bridged between 10GigE, IBA, and Myrinet via the sockets interface. Our evaluations show that in most experimental scenarios, 10GigE provides comparable (or better) performance than IBA and Myrinet. Further, for grid environments, where legacy TCP/IP/Ethernet is dominant in the wide-area network, IBA and Myrinet have been practically *no shows* because of lack of compatibility of these networks with Ethernet. However, this may soon change with the recent announcement of the Myri-10G PCI-Express network adapter by Myricom.

# CHAPTER 9

# SOCKETS VS RDMA INTERFACE OVER 10-GIGABIT NETWORKS: AN IN-DEPTH ANALYSIS OF THE MEMORY TRAFFIC BOTTLENECK

The introduction of gigabit speed networks a few years back had challenged the traditional TCP/IP implementation in two aspects, namely performance and CPU requirements. The advent of 10-Gigabit networks such as 10-Gigabit Ethernet and InfiniBand has added a new dimension of complexity to this problem, *Memory Traffic*. While there have been previous studies which show the implications of the memory traffic bottleneck, to the best of our knowledge, there has been no study which shows the actual impact of the memory accesses generated by TCP/IP for 10-Gigabit networks.

In this chapter, we evaluate the various aspects of the TCP/IP protocol suite for 10-Gigabit networks including the memory traffic and CPU requirements, and compare these with RDMA capable network adapters, using 10-Gigabit Ethernet and InfiniBand as example networks. Our measurements show that while the host based TCP/IP stack has a high CPU requirement, up to about 80% of this overhead is associated with the core protocol implementation especially for large messages and is potentially offloadable using the recently proposed TCP Offload Engines or user-level sockets layers.

Further, our studies reveal that for 10-Gigabit networks, the sockets layer itself becomes a significant bottleneck for memory traffic. Especially when the data is not present in the L2-cache, network transactions generate significant amounts of memory bus traffic for the TCP protocol stack. As we will see in the later sections, each byte transferred on the network can generate up to 4 bytes of data traffic on the memory bus. With the current moderately fast memory buses (e.g., 64bit/333MHz) and low memory efficiencies (e.g., 65%), this amount of memory traffic limits the peak throughput applications can achieve to less than 35% of the network's capability. Further, the memory bus and CPU speeds have not been scaling with the network bandwidth [15], pointing to the fact that this problem is only going to worsen in the future.

We also evaluate the RDMA interface of the InfiniBand architecture to understand the implications of having RDMA-based extensions to sockets in two aspects: (a) the CPU requirement for the TCP stack usage and the copies associated with the sockets interface, (b) the difference in the amounts of memory traffic generated by RDMA compared to that of the traditional sockets API. Our measurements show that the RDMA interface requires up to four times lesser memory traffic and has almost zero CPU requirement for the data sink. These measurements show the potential impacts of having RDMA-based extensions to sockets on 10-Gigabit networks.

## 9.1 Background

In this section, we provide a brief background about the TCP protocol suite.

### 9.1.1 TCP/IP Protocol Suite

The data processing path taken by the TCP protocol stack is broadly classified into the transmission path and the receive path. On the transmission side, the message is copied into the socket buffer, divided into MTU sized segments, data integrity ensured through checksum computation (to form the TCP checksum) and passed on to the underlying IP layer. Linux-2.4 uses a combined checksum and copy for the transmission path, a well known optimization first proposed by Jacobson, et al. [37]. The IP layer extends the checksum to include the IP header and form the IP checksum and passes on the IP datagram to the device driver. After the construction of a packet header, the device driver makes a descriptor for the packet and passes the descriptor to the NIC. The NIC performs a DMA operation to move the actual data indicated by the descriptor from the socket buffer to the NIC buffer. The NIC then ships the data with the link header to the physical network and raises an interrupt to inform the device driver that it has finished transmitting the segment.

On the receiver side, the NIC receives the IP datagrams, DMAs them to the socket buffer and raises an interrupt informing the device driver about this. The device driver strips the packet off the link header and hands it over to the IP layer. The IP layer verifies the IP checksum and if the data integrity is maintained, hands it over to the TCP layer. The TCP layer verifies the data integrity of the message and places the data into the socket buffer. When the application calls the `read()` operation, the data is copied from the socket buffer to the application buffer.

## 9.2 Understanding TCP/IP Requirements

In this section, we study the impact of cache misses not only on the performance of the TCP/IP protocol stack, but also on the amount of memory traffic associated with these cache misses; we estimate the amount of memory traffic for a typical throughput test. In Section 9.3, we validate these estimates through measured values.

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. In this section, we describe the path taken by the second case, i.e., when the application buffer does not fit in cache. We also present the final memory traffic ratio of the first case, but refer the reader to [15] for the actual data path description due to space restrictions. Figures 9.1a and 9.1b illustrate the memory accesses associated with network communication.

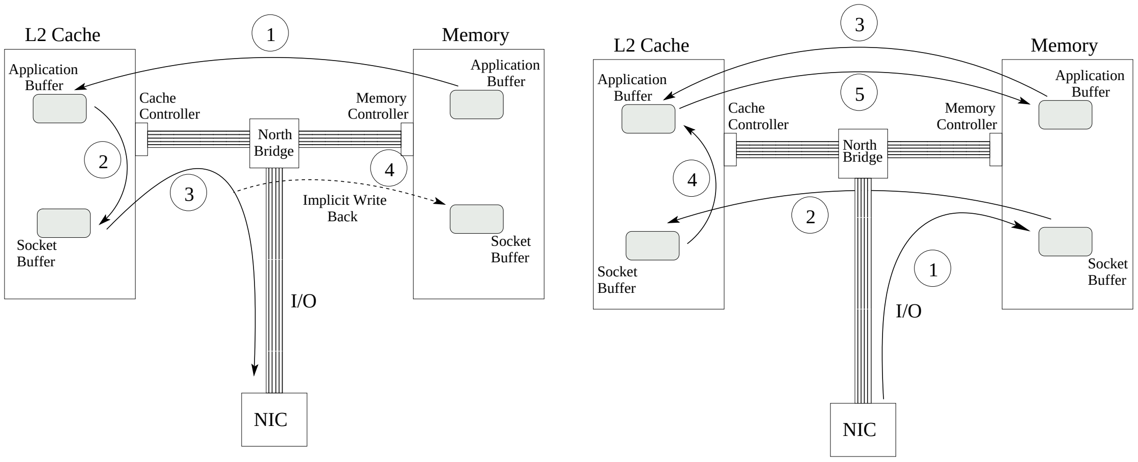

Figure 9.1: Memory Traffic for Sockets: (a) Transmit Path; (b) Receive Path

### 9.2.1 Transmit Path

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. The following are the steps involved on the transmission side:

**CPU reads the application buffer (step 1):** The application buffer has to be fetched to cache on every iteration 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 (step 2):** 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*. 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. So ideally this operation should not generate any 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). Thus, this operation can require between 0 and 2 bytes of memory traffic per network byte.

**NIC does a DMA read of the socket buffer (steps 3 and 4):** 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, most memory controllers perform 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 four steps, 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).

Further, 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. 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. 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.

### 9.2.2 Receive Path

The memory traffic associated with the receive path is simpler compared to that of the transmit path. The following are steps involved on the receive path:

**NIC does a DMA write into the socket buffer (step 1):** 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. In the subsequent iterations, even if the socket buffer is fetched to cache, it would not be in a *Modified* state (since it is only being used to copy data into the application buffer). Thus, the DMA write request would be allowed to proceed as soon as the socket buffer in the cache is invalidated by the North Bridge (Figure 9.1), 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 (step 2):** Again, at this point the socket buffer is not present in cache, and has to be fetched, requiring one transaction from the memory. It is to be noted that even if the buffer was present in the cache before the iteration, it has to be evicted or invalidated for the previous step.

**CPU writes to application buffer (steps 3, 4 and 5):** 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 9.1 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. While the actual memory access pattern is significantly more complicated than the one described above due to the pipelining of data transmission to and from the socket buffer, this model captures the bulk of the memory transactions and provides a fair enough estimate.

Table 9.1: 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** |

## 9.3 Experimental Results

In this section, we present some of the experiments we have conducted over 10 Gigabit Ethernet and InfiniBand.

The test-bed used for evaluating the 10-Gigabit Ethernet stack consisted of two clusters.

**Cluster 1:** Two Dell2600 Xeon 2.4 GHz 2-way SMP nodes, each with 1GB main memory (333MHz, DDR), Intel E7501 chipset, 32Kbyte L1-Cache, 512Kbyte L2-Cache, 400MHz/64-bit Front Side Bus, PCI-X 133MHz/64bit I/O bus, Intel 10GbE/Pro 10-Gigabit Ethernet adapters.

**Cluster 2:** Eight P4 2.4 GHz IBM xSeries 305 nodes, each with 256Kbyte main memory and connected using the Intel Pro/1000 MT Server Gigabit Ethernet adapters. We used Windows Server 2003 and Linux kernel 2.4.18-14smp for our evaluations. The multi-stream tests were conducted using a FoundryNet 10-Gigabit Ethernet switch.

The test-bed used for evaluating the InfiniBand stack consisted of the following cluster.

**Cluster 3:** Eight nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4GHz processors with a 512Kbyte L2 cache and a 400MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA 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 2.4.7-10smp kernel version.

### 9.3.1 10-Gigabit Ethernet

In this section we present the performance delivered by 10-Gigabit Ethernet, the memory traffic generated by the TCP/IP stack (including the sockets interface) and the CPU requirements of this stack.

#### Micro-Benchmark Evaluation

For the micro-benchmark tests, we have studied the impacts of varying different parameters in the system as well as the TCP/IP stack including (a) Socket Buffer Size, (b) Maximum Transmission Unit (MTU), (c) Network adapter offloads (checksum, segmentation), (d) PCI burst size (PBS), (e) Switch Behavior, (f) TCP window size, (g) Adapter Interrupt delay settings, (h) Operating System (Linux and Windows Server 2003) and several others. Most of these micro-benchmarks use the same buffer for transmission resulting in maximum cache hits presenting the ideal case performance achievable by 10-Gigabit Ethernet. Due to this reason, these results tend to hide a number of issues related to memory traffic. The main idea of this research is to study the memory bottleneck in the TCP/IP stack. Hence, we have shown only a subset of these micro-benchmarks in this chapter. The rest of the micro-benchmarks can be found in [15].

**Single Stream Tests:** Figure 9.2a shows the one-way ping-pong latency achieved by 10-Gigabit Ethernet. We can see that 10-Gigabit Ethernet is able to achieve a latency of about 37$\mu$s for a message size of 256bytes on the Windows Server 2003 platform. The figure also shows the average CPU utilization for the test. We can see that the test requires about 50% CPU on each side. We have also done a similar analysis on Linux where 10-Gigabit Ethernet achieves a latency of about 20.5$\mu$s (Figure 9.3a).

Figure 9.2b shows the throughput achieved by 10-Gigabit Ethernet on the Windows Server 2003 platform. The parameter settings used for the experiment were a socket buffer size of 64Kbytes (both send and receive on each node), MTU of 16Kbytes, checksum offloaded on to the network card and the PCI burst size set to 4Kbytes. 10-Gigabit Ethernet achieves a peak throughput of about 2.5Gbps with a CPU usage of about 110% (dual processor system). We can see that the amount of CPU used gets saturated at about 100% though we are using dual processor systems. This is attributed to the interrupt routing mechanism for the "x86" architecture. The x86 architecture routes all interrupts to the first processor. For interrupt based protocols such as TCP, this becomes a huge bottleneck, since this essentially restricts the transmission side to about one CPU. This behavior is also seen in the multi-stream transmission tests (in particular the fan-out test) which are provided in the later sections. The throughput results for the Linux platform are presented in Figure 9.3b.

**Multi-Stream Tests:** For the multi-stream results, we study the performance of the host TCP/IP stack in the presence of multiple data streams flowing from or

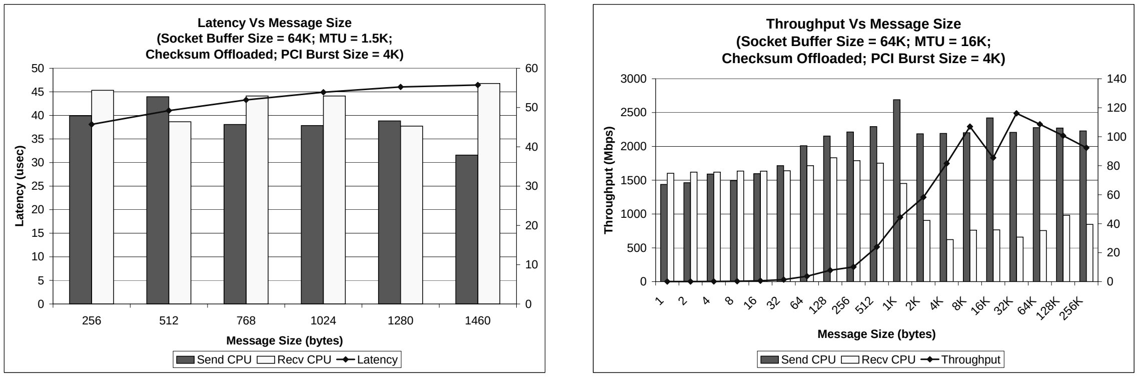

Figure 9.2: Micro-Benchmarks for the host TCP/IP stack over 10-Gigabit Ethernet on the Windows Platform: (a) One-Way Latency (MTU 1.5K); (b) Throughput (MTU 16K)

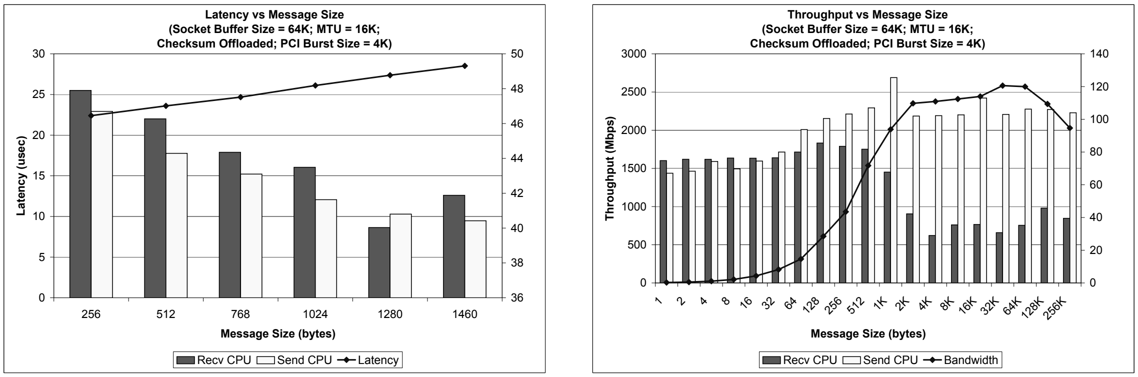

Figure 9.3: Micro-Benchmarks for the host TCP/IP stack over 10-Gigabit Ethernet on the Linux Platform: (a) One-Way Latency (MTU 1.5K); (b) Throughput (MTU 16K)

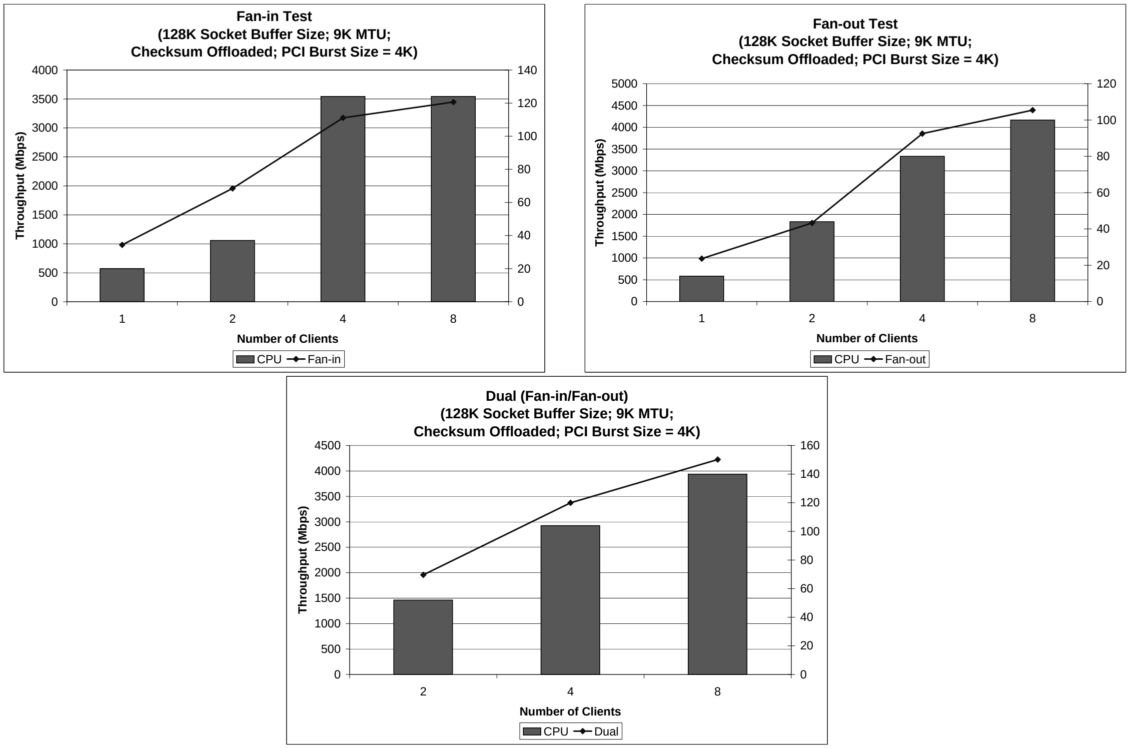

Figure 9.4: Multi-Stream Micro-Benchmarks: (a) Fan-in, (b) Fan-out, (c) Dual (Fan-in/Fan-out)

into the node. The environment used for the multi-stream tests consisted of one node with a 10-Gigabit Ethernet adapter and several other nodes connected to the same switch using a 1-Gigabit Ethernet adapter.

Three main experiments were conducted in this category. The first test was a Fan-in test, where all the 1-Gigabit Ethernet nodes push data to the 10-Gigabit Ethernet node through the common switch they are connected to. The second test was a Fan-out test, where the 10-Gigabit Ethernet node pushes data to all the 1-Gigabit Ethernet nodes through the common switch. The third test was Dual test, where the 10-Gigabit Ethernet node performs the fan-in test with half the 1-Gigabit Ethernet nodes and the fan-out test with the other half. It is to be noted that the Dual test is quite different from a multi-stream bi-directional bandwidth test where the server node (10-Gigabit Ethernet node) does both a fan-in and a fan-out test with each client node (1-Gigabit Ethernet node). The message size used for these experiments is 10Mbytes. This forces the message *not to be in L2-cache* during subsequent iterations.

Figures 9.4a and 9.4b show the performance of the host TCP/IP stack over 10-Gigabit Ethernet for the Fan-in and the Fan-out tests. We can see that we are able to achieve a throughput of about 3.5Gbps with a 120% CPU utilization (dual CPU) for the Fan-in test and about 4.5Gbps with a 100% CPU utilization (dual CPU) for the Fan-out test. Further, it is to be noted that the server gets saturated in the Fan-in test for 4 clients. However, in the fan-out test, the throughput continues to increase from 4 clients to 8 clients. This again shows that with 10-Gigabit Ethernet, the receiver is becoming a bottleneck in performance mainly due to the high CPU overhead involved on the receiver side.

Figure 9.4c shows the performance achieved by the host TCP/IP stack over 10-Gigabit Ethernet for the Dual test. The host TCP/IP stack is able to achieve a throughput of about 4.2Gbps with a 140% CPU utilization (dual CPU).

### TCP/IP CPU Pareto Analysis

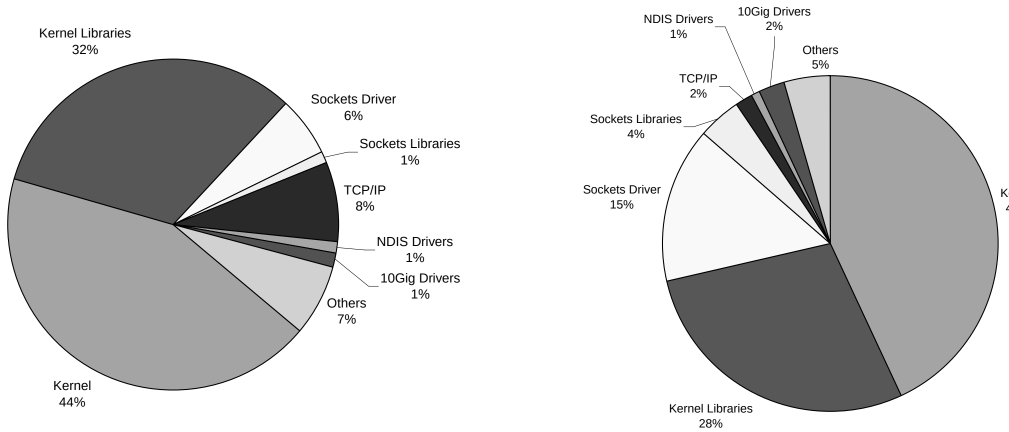

Figure 9.5: Throughput Test: CPU Pareto Analysis for small messages (64bytes): (a) Transmit Side, (b) Receive Side

In this section we present a module wise break-up (Pareto Analysis) for the CPU overhead of the host TCP/IP stack over 10-Gigabit Ethernet. We used the *NTttcp* throughput test as a benchmark program to analyze this. Like other micro-benchmarks, the NTttcp test uses the same buffer for all iterations of the data transmission. So, the pareto analysis presented here is for the ideal case with the maximum number of cache hits. For measurement of the CPU overhead, we used the *Intel VTune$^{TM}$ Performance Analyzer*. In short, the *VTune$^{TM}$ Analyzer* interrupts the processor at specified events (e.g., every 'n' clock ticks) and records its execution context at that sample. Given enough samples, the result is a statistical profile of the ratio of the time spent in a particular routine. More details about the *Intel VTune*$^{TM}$ *Performance Analyzer* can be found in [15].

Figures 9.5 and 9.6 present the CPU break-up for both the sender as well as the receiver for small messages (64bytes) and large messages (16Kbytes) respectively. It can be seen that in all the cases, the kernel and the protocol stack add up to about 80% of the CPU overhead. For small messages, the overhead is mainly due to the per-message interrupts. These interrupts are charged into the kernel usage, which accounts for the high percentage of CPU used by the kernel for small messages. For larger messages, on the other hand, the overhead is mainly due to the data touching portions in the TCP/IP protocol suite such as checksum, copy, etc.

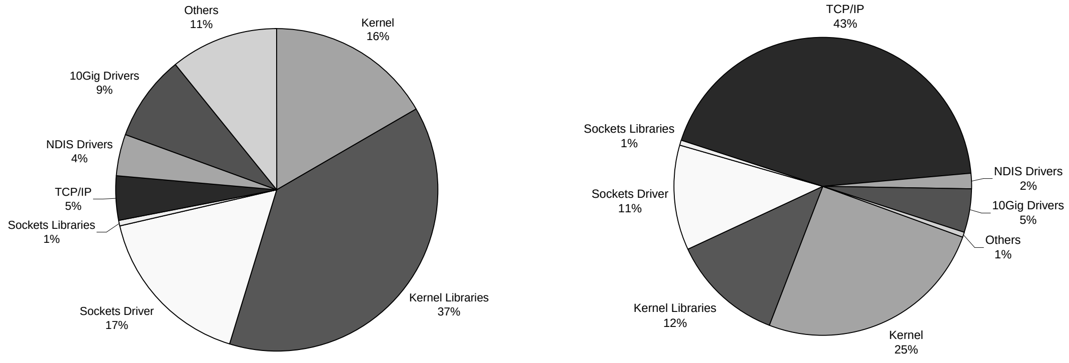

Figure 9.6: Throughput Test: CPU Pareto Analysis for large messages (16Kbytes): (a) Transmit Size, (b) Receive Side

As it can be seen in the pareto analysis, in cases where the cache hits are high, most of the overhead of TCP/IP based communication is due to the TCP/IP protocol processing itself or due to other kernel overheads. This shows the potential benefits of having TCP Offload Engines in such scenarios where these components are optimized by pushing the processing to the hardware. However, the per-packet overheads for small messages such as interrupts for sending and receiving data segments would still be present in spite of a protocol offload. Further, as we'll see in the memory traffic analysis (the next section), for cases where the cache hit rate is not very high, the memory traffic associated with the sockets layer becomes very significant forming a fundamental bottleneck for all implementations which support the sockets layer, including high performance user-level sockets as well as TCP Offload Engines.

**TCP/IP Memory Traffic**

For the memory traffic tests, we again used the *VTune*$^{TM}$ *Performance Analyzer*. We measure the number of cache lines fetched and evicted from the processor cache to calculate the data traffic on the Front Side Bus. Further, we measure the data being transferred from or to memory from the North Bridge to calculate the memory traffic (on the Memory Bus). Based on these two calculations, we evaluate the amount of traffic being transferred over the I/O bus (difference in the amount of traffic on the Front Side Bus and the Memory Bus).

**Single Stream Tests:** Figure 9.7a shows the memory traffic associated with the data being transferred on the network for the sender and the receiver sides. As discussed in Section 9.2, for small message sizes (messages which fit in the L2-cache), we can expect about 1 byte of memory traffic per network byte on the sender side and about 2 bytes of memory traffic per network byte on the receiver side. However, the amount of memory traffic seems to be large for very small messages. The reason for this is the TCP control traffic and other noise traffic on the memory bus. Such traffic would significantly affect the smaller message sizes due to the less amount of memory traffic associated with them. However, when the message size becomes moderately large (and still fits in L2-cache), we can see that the message traffic follows the trend predicted.

For large message sizes (messages which do not fit in the L2-cache), we can expect between 2 and 4 bytes of memory traffic per network byte on the sender side and about 4 bytes of memory traffic per network byte on the receiver side. We can see that the actual memory to network traffic ratio follows this trend.

These results show that even without considering the host CPU requirements for the TCP/IP protocol stack, the memory copies associated with the sockets layer can generate up to 4 bytes of memory traffic per network byte for traffic in each direction, forming what we call the *memory-traffic* bottleneck. It is to be noted that while some TCP Offload Engines try to avoid the memory copies in certain scenarios, the sockets API can not force a zero copy implementation for all cases (e.g., transactional protocols such as RPC, File I/O, etc. first read the data header and decide the size of the buffer to be posted). This forces the memory-traffic bottleneck to be associated with the sockets API.

**Multi-Stream Tests:** Figure 9.7b shows the actual memory traffic associated with the network data transfer during the multi-stream tests. It is to be noted that the message size used for the experiments is 10Mbytes, so subsequent transfers of the message need the buffer to be fetched from memory to L2-cache.

The first two legends in the figure show the amount of bytes transferred on the network and the bytes transferred on the memory bus per second respectively. The third legend shows 65% of the peak bandwidth achievable by the memory bus. 65% of the peak memory bandwidth is a general rule of thumb used by most computer companies to estimate the peak practically sustainable memory bandwidth on a memory bus when the requested physical pages are non-contiguous and are randomly placed. It is to be noted that though the virtual address space could be contiguous, this doesn't enforce any policy on the allocation of the physical address pages and they can be assumed to be randomly placed.

It can be seen that the amount of memory bandwidth required is significantly larger than the actual network bandwidth. Further, for the Dual test, it can be seen that the memory bandwidth actually reaches within 5% of the peak practically sustainable bandwidth.

### 9.3.2 InfiniBand Architecture

In this section, we present briefly the performance achieved by RDMA enabled network adapters such as InfiniBand.

Figure 9.8a shows the one-way latency achieved by the RDMA write communication model of the InfiniBand stack for the polling based approach for completion as well as an event based approach. In the polling approach, the application continuously monitors the completion of the message by checking the arrived data. This activity makes the polling based approach CPU intensive resulting in a 100% CPU utilization. In the event based approach, the application goes to sleep after posting the descriptor. The network adaptor raises an interrupt for the application once the message arrives. This results in a lesser CPU utilization for the event based scheme.

We see that RDMA write achieves a latency of about 5.5$\mu$s for both the polling based scheme as well as the event based scheme. The reason for both the event based scheme and the polling based scheme performing alike is the receiver transparency for RDMA Write operations. Since, the RDMA write operation is completely receiver transparent, the only way the receiver can know that the data has arrived into its

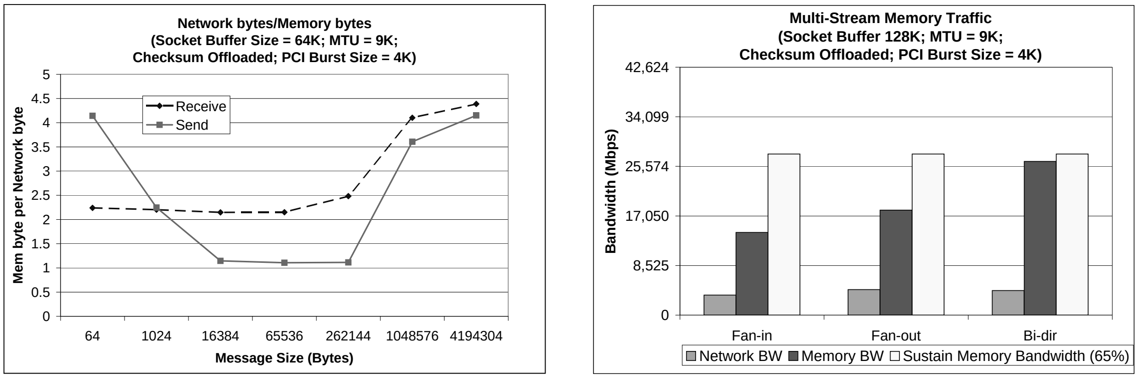

Figure 9.7: Throughput Test Memory Traffic Analysis: (a) Single Stream, (b) Multi Stream

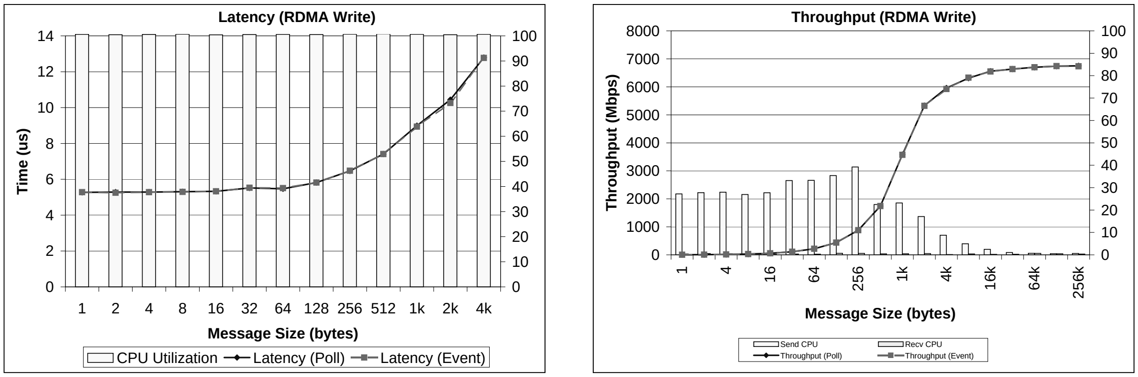

Figure 9.8: IBA Micro-Benchmarks for RDMA Write: (a) Latency and (b) Throughput

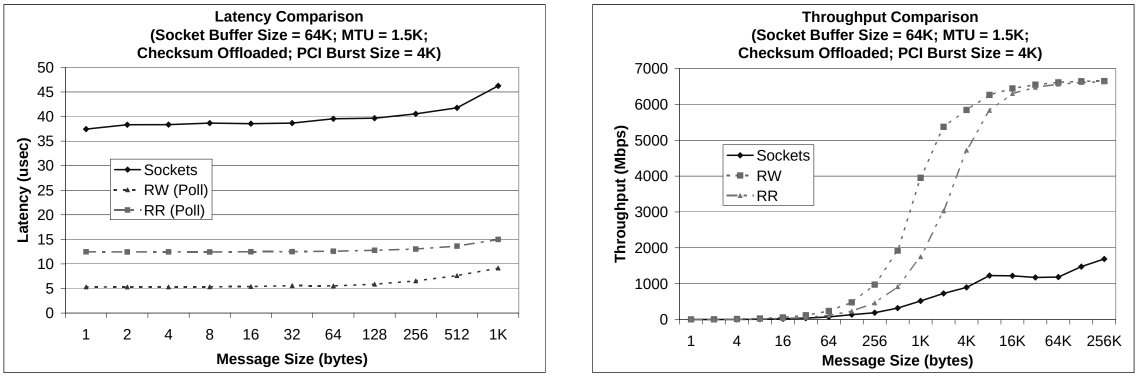

Figure 9.9: Latency and Throughput Comparison: Host TCP/IP over 10-Gigabit Ethernet Vs InfiniBand

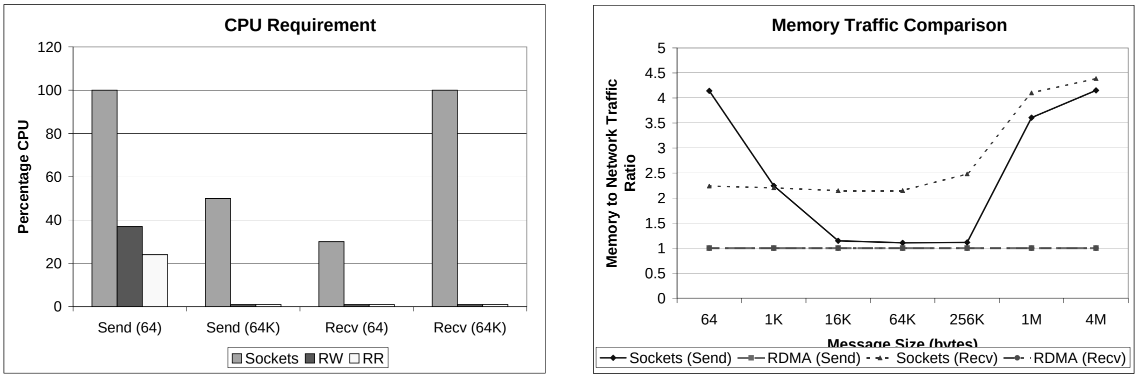

Figure 9.10: CPU Requirement and Memory Traffic Comparisons: Host TCP/IP over 10-Gigabit Ethernet Vs InfiniBand

buffer is by polling on the last byte. So, in an event-based approach only the sender would be blocked on send completion using a notification event; the notification overhead at the sender is however parallelized with the data transmission and reception. Due to this the time taken by RDMA write for the event-based approach is similar to that of the polling based approach. Due to the same reason, the CPU overhead in the event-based approach is 100% (similar to the polling based approach).

RDMA Read on the other hand achieves a latency of about 12.5$\mu$s for the polling based scheme and about 24.5$\mu$s for the event based scheme. The detailed results for RDMA read and the other communication models such as send-receive and RDMA write with immediate data can be found in [15].

Figure 9.8 shows the throughput achieved by RDMA write. Again, results for both the polling based approach as well as the event-based approach are presented.

Both approaches seem to perform very close to each other giving a peak throughput of about 6.6Gbps. The peak throughput is limited by the sustainable bandwidth on the PCI-X bus. The way the event-based scheme works is that, it first checks the completion queue for any completion entries present. If there are no completion entries present, it requests a notification from the network adapter and blocks while waiting for the data to arrive. In a throughput test, data messages are sent one after the other continuously. So, the notification overhead can be expected to be overlapped with the data transmission overhead for the consecutive messages. This results in a similar performance for the event-based approach as well as the polling based approach.

The CPU utilization values are only presented for the event-based approach; those for the polling based approach stay close to 100% and are not of any particular interest. The interesting thing to note is that for RDMA, there is nearly zero CPU utilization for the data sink especially for large messages.

### 9.3.3 10-Gigabit Ethernet/InfiniBand Comparisons

Figures 9.9a and 9.9b show the latency and throughput comparisons between IBA and 10-Gigabit Ethernet respectively. In this figure we have skipped the event based scheme and shown just the polling based scheme. The reason for this is the software stack overhead in InfiniBand. The performance of the event based scheme depends on the performance of the software stack to handle the events generated by the network adapter and hence would be specific to the implementation we are using. Hence, to get an idea of the peak performance achievable by InfiniBand, we restrict ourselves to the polling based approach.

We can see that InfiniBand is able to achieve a significantly higher performance than the host TCP/IP stack on 10-Gigabit Ethernet; a factor of three improvement in the latency and a up to a 3.5 times improvement in the throughput. This improvement in performance is mainly due to the offload of the network protocol, direct access to the NIC and direct placement of data into the memory.

Figures 9.10a and 9.10b show the CPU requirements and the memory traffic generated by the host TCP/IP stack over 10-Gigabit Ethernet and the InfiniBand stack. We can see that the memory traffic generated by the host TCP/IP stack is much higher (more than 4 times in some cases) as compared to InfiniBand; this difference is mainly attributed to the copies involved in the sockets layer for the TCP/IP stack. This result points to the fact that in spite of the possibility of an offload of the TCP/IP stack on to the 10-Gigabit Ethernet network adapter TCP's scalability would still be restricted by the sockets layer and its associated copies. On the other hand, an RDMA extended sockets interface can be expected to achieve all the advantages seen by InfiniBand.

Some of the expected benefits are (1) Low overhead interface to the network, (2) Direct Data Placement (significantly reducing intermediate buffering), (3) Support for RDMA semantics, i.e., the sender can handle the buffers allocated on the receiver node and (4) Most importantly, the amount of memory traffic generated for the network communication will be equal to the number of bytes going out to or coming in from the network, thus improving scalability.

## 9.4 Summary

The compute requirements associated with the TCP/IP protocol suite have been previously studied by a number of researchers. However, the recently developed 10 Gigabit networks such as 10-Gigabit Ethernet and InfiniBand have added a new dimension of complexity to this problem, *Memory Traffic*. While there have been previous studies which show the implications of the memory traffic bottleneck, to the best of our knowledge, there has been no study which shows the actual impact of the memory accesses generated by TCP/IP for 10-Gigabit networks.

In this chapter, we first do a detailed evaluation of various aspects of the host-based TCP/IP protocol stack over 10-Gigabit Ethernet including the memory traffic and CPU requirements. Next, we compare these with RDMA capable network adapters, using InfiniBand as an example network. Our measurements show that while the host based TCP/IP stack has a high CPU requirement, up to 80% of this overhead is associated with the core protocol implementation especially for large messages and is potentially offloadable using the recently proposed TCP Offload Engines. However, the current host based TCP/IP stack also requires multiple transactions of the data (up to a factor of four in some cases) over the current moderately fast memory buses, curbing their scalability to faster networks; for 10-Gigabit networks, the host based TCP/IP stack generates enough memory traffic to saturate a 333MHz/64bit DDR memory bandwidth even before 35% of the available network bandwidth is used.

Our evaluation of the RDMA interface over the InfiniBand network tries to nail down some of the benefits achievable by providing RDMA-based extensions to sockets. In particular, we try to compare the RDMA interface over InfiniBand not only in performance, but also in other resource requirements such as CPU usage, memory traffic, etc. Our results show that the RDMA interface requires up to four times lesser memory traffic and has almost zero CPU requirements for the data sink. These measurements show the potential impacts of having RDMA-based extensions to sockets on 10-Gigabit networks.

# CHAPTER 10

# SUPPORTING STRONG COHERENCY FOR ACTIVE CACHES IN MULTI-TIER DATA-CENTERS OVER INFINIBAND

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. Multi-tier Data-centers have become a central requirement to providing such services. Figure 10.1 represents a typical multi-tier data-center. The front tiers consist of front-end servers such as proxy servers that provide web, messaging and various other services to clients. The middle tiers usually comprise of application servers that handle transaction processing and implement data-center business logic. The back-end tiers consist of database servers that hold a persistent state of the databases and other data repositories. As mentioned in [73], a fourth tier emerges in today's data-center environment: a communication service tier between the network and the front-end server farm for providing edge services such as caching, resource monitoring and dynamic reconfiguration, load balancing, security, and others. In this chapter, we concentrate on two of these services, namely caching and resource monitoring and dynamic reconfiguration.

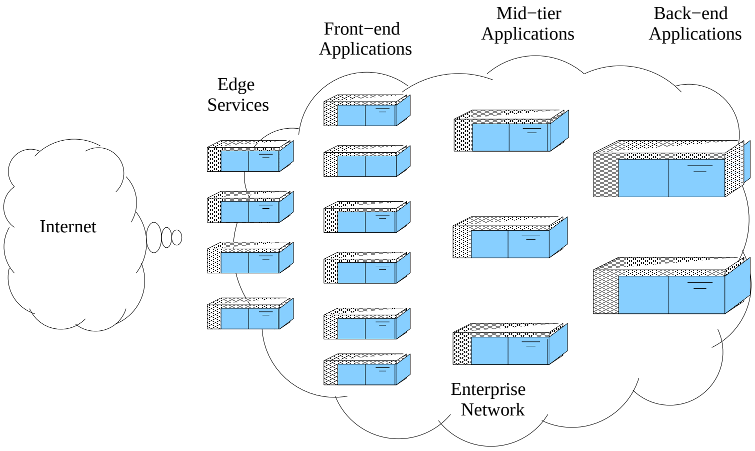

Figure 10.1: A Typical Multi-Tier Data-Center

**Caching:** With ever increasing on-line businesses and services and the growing popularity of personalized Internet services, dynamic content is becoming increasingly common [29, 83, 75]. This includes documents that change upon every access, documents that are query results, documents that embody client-specific information, etc., which involve significant computation and communication requirements. Caching is a widely accepted approach for curbing these requirements and maintaining a high performance. However, caching dynamic content, typically known as *Active Caching* [29] has its own challenges: issues such as cache consistency and cache coherence become more prominent. Efficiently handling these can allow data-centers to sustain a high performance even for data-centers serving large amounts of dynamic content.

**Resource Monitoring and Dynamic Reconfiguration:** In the past few years several researchers have proposed and configured data-centers providing multiple independent services, known as shared data-centers [33, 59]. For example, several ISPs and other web service providers host multiple unrelated web-sites on their data-centers allowing potential differentiation in the service provided to each of them. The increase in such services results in a growing fragmentation of the 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 website. 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.

While current generation high-speed networks such as InfiniBand provide a wide range of capabilities to allow efficient handling of issues such as these (both active caching as well dynamic reconfigurability), most of the data-center applications are written using the sockets interface. Rewriting them to use the native network interface is cumbersome and impractical. Using high-performance sockets implementations such as SDP over InfiniBand is a straight-forward approach for allowing such applications to benefit from the high performance of the networks without any changes. However, as we will see in the later sections, while SDP provides a high performance, it is restricted by the sockets interface itself. Specifically, for many data-center applications, the two-sided communication interface of sockets (with both the sender and receiver having to be involved for communication) hampers the performance significantly due to the non-uniform load conditions prevalent in data-centers.

In this chapter, we propose an extended sockets interface where one-sided communication features, similar to the Remote Direct Memory Access (RDMA), are incorporated into the sockets semantics. Our results demonstrate that with this new sockets interface, applications such as active caching and dynamic reconfigurability can achieve close to an order of magnitude improvement in performance while requiring minimal changes.

## 10.1 Background

In this section, we briefly describe the various schemes previously proposed by researchers to deal with caching dynamic content as well as dynamic reconfiguration in shared data-centers.

### 10.1.1 Web Cache Consistency and Coherence

Traditionally, frequently accessed static content was cached at the front tiers to allow users a quicker access to these documents. In the past few years, researchers have come up with approaches of caching certain dynamic content at the front tiers as well [29]. In the current web, many cache eviction events and uncachable resources are driven by two server application goals: First, providing clients with a *recent* or *coherent* view of the state of the application (i.e., information that is not too old); Secondly, providing clients with a *self-consistent* view of the application's state as it changes (i.e., once the client has been told that something has happened, that client should never be told anything to the contrary). Depending on the type of data being considered, it is necessary to provide certain guarantees with respect to the view of the data that each node in the data-center and the users get. These constraints on the view of data vary based on the application requiring the data.

**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.

**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.

### Web Cache Consistency

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 $\Delta$-*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* [39] and *MONARCH* [61] 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 [26, 25].

**Web Cache Coherence**

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% [75]. 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 [26, 25, 83]. 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. However, this scheme also has disadvantages, mainly based on the traditional networking protocols:

- Every data document is typically associated with a home-node in the data-center back-end. Frequent accesses to a document can result in all the front-end nodes sending in *coherence status* requests to the same nodes potentially forming a *hot-spot* at this node
- Traditional protocols require the back-end nodes to be interrupted for every cache validation event generated by the front-end

In this chapter, we focus on this model of cache coherence and analyze the various impacts of the advanced features provided by InfiniBand on this.

### 10.1.2 Shared Cluster-Based Data-Center Environment

A clustered data-center environment essentially tries to utilize the benefits of a cluster environment (e.g., high performance-to-cost ratio) to provide the services requested in a data-center environment (e.g., web hosting, transaction processing). As mentioned earlier, researchers have proposed and configured data-centers to provide multiple independent services, such as hosting multiple web-sites, forming what is known as shared data-centers.

Figure 10.2 shows a higher level layout of a shared data-center architecture hosting multiple web-sites. External clients request documents or services from the data-center over the WAN/Internet through load-balancers using higher level protocols such as *HTTP*. The load-balancers on the other hand serve the purpose of exposing a single IP address to all the clients while maintaining a list of several internal IP addresses to which they forward the incoming requests based on a pre-defined algorithm (e.g., round-robin).

While hardware load-balancers are commonly available today, they suffer from being based on a pre-defined algorithm and are difficult to be tuned based on the requirements of the data-center. On the other hand, though software load-balancers are easy to modify and tuned based on the data-center requirements, they can potentially form bottlenecks themselves for highly loaded data-centers. In the past, several researchers have proposed the use of an additional cluster of nodes (known as the edge tier) [73] to perform certain services such as intelligent load-balancing, caching, etc [59]. Requests can be forwarded to this cluster of software load-balancers either by the clients themselves by using techniques such as DNS aliasing, or by using an additional hardware load-balancer.

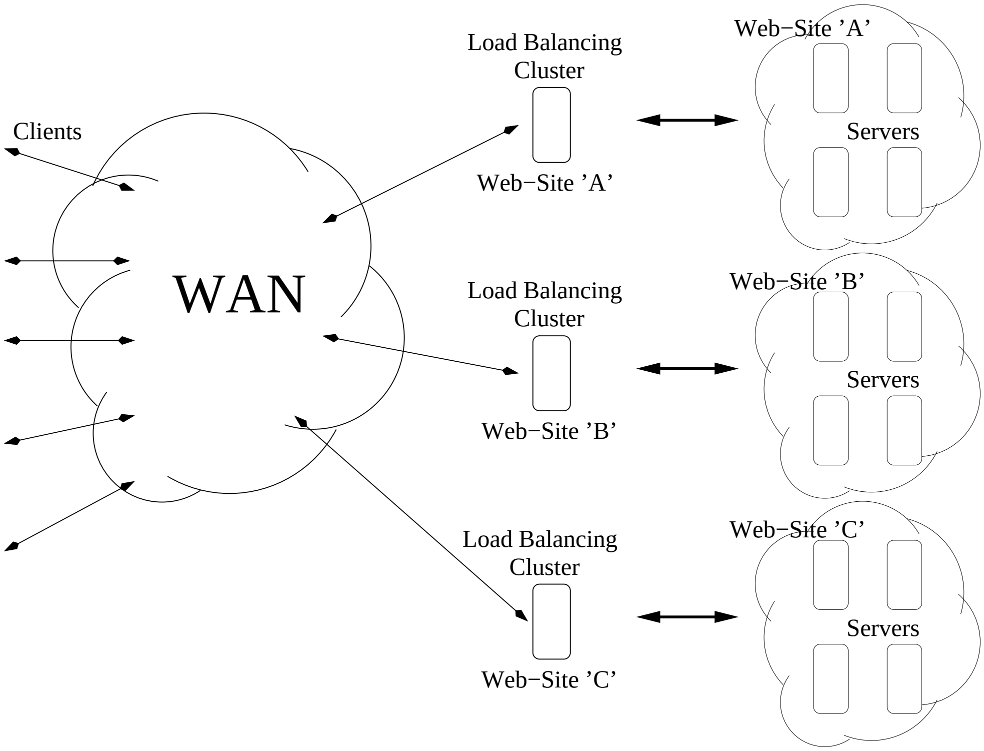

Figure 10.2: A Shared Cluster-Based Data-Center Environment

The servers inside the clustered data-center provide the actual services such as web-hosting, transaction processing, etc. Several of these services require computationally intensive processing such as CGI scripts, Java servlets and database query operations (table joins, etc). This makes the processing on the server nodes CPU intensive in nature.

## 10.2 Providing Strong Cache Coherence

In this section, we describe the architecture we use to support strong cache coherence. We first provide the basic design of the architecture for any generic protocol. Next, we point out several optimizations possible in the design using the one-sided communication and atomic operation features provided by the extended sockets interface over InfiniBand.

### 10.2.1 Basic Design

As mentioned earlier, there are two popular approaches to ensure cache coherence: *Client-Polling* and *No-Caching*. In this research, we focus on the *Client-Polling* approach to demonstrate the potential benefits of InfiniBand in supporting strong cache coherence.

While the HTTP specification allows a cache-coherent client-polling architecture (by specifying a TTL value of NULL and using the `‘‘get-if-modified-since’’` HTTP request to perform the polling operation), it has several issues: (1) This scheme is specific to sockets and cannot be used with other programming interfaces such as

InfiniBand's native Verbs layers (e.g.: VAPI), (2) In cases where persistent connections are not possible (HTTP/1.0 based requests, secure transactions, etc), connection setup time between the nodes in the data-center environment tends to take up a significant portion of the client response time, especially for small documents.

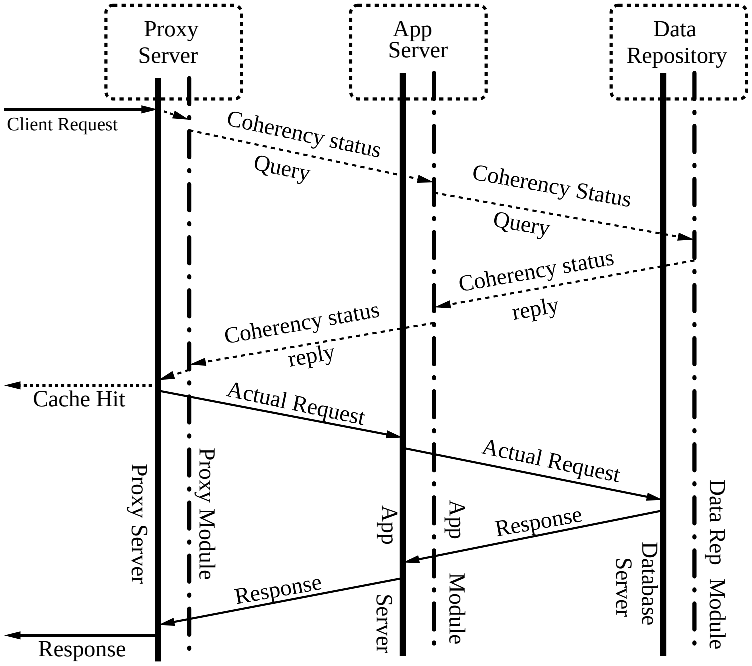

Figure 10.3: Strong Cache Coherence Protocol

In the light of these issues, we present an alternative architecture to perform *Client-Polling*. Figure 10.3 demonstrates the basic coherency architecture used in this research. The main idea of this architecture is to introduce *external helper modules* that work along with the various servers in the data-center environment to ensure cache coherence. All issues related to cache coherence are handled by these modules and are obscured from the data-center servers. It is to be noted that the data-center servers require very minimal changes to be compatible with these modules.

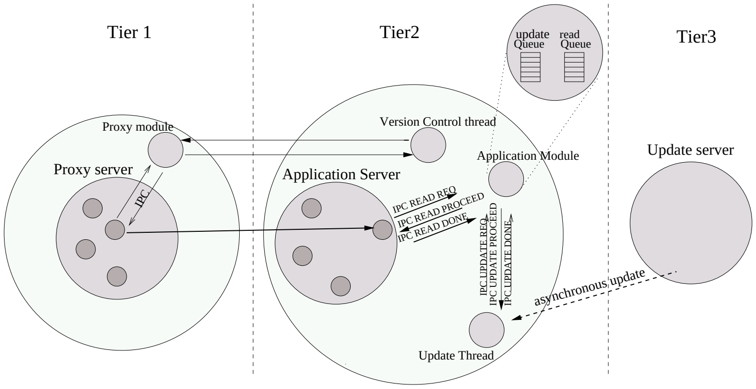

Figure 10.4: Interaction between Data-Center Servers and Modules

The design consists of a module on each physical node in the data-center environment associated with the server running on the node, i.e., each proxy node has a proxy module, each application server node has an associated application module, etc. The proxy module assists the proxy server with validation of the cache on every request. The application module, on the other hand, deals with a number of things including (a) Keeping track of all updates on the documents it owns, (b) Locking appropriate files to allow a multiple-reader-single-writer based access priority to files, (c) Updating the appropriate documents during update requests, (d) Providing the proxy module with the appropriate version number of the requested file, etc. Figure 10.4 demonstrates the functionality of the different modules and their interactions.

**Proxy Module:** On every request, the proxy server contacts the proxy module through IPC to validate the cached object(s) associated with the request. The proxy module does the actual verification of the document with the application module on the appropriate application server. If the cached value is valid, the proxy server is allowed to proceed by replying to the client's request from cache. If the cache is invalid, the proxy module simply deletes the corresponding cache entry and allows the proxy server to proceed. Since the document is now not in cache, the proxy server contacts the appropriate application server for the document. This ensures that the cache remains coherent.

**Application Module:** The application module is slightly more complicated than the proxy module. It uses multiple threads to allow both updates and read accesses on the documents in a multiple-reader-single-writer based access pattern. This is handled by having a separate thread for handling updates (referred to as the *update thread* here on). The main thread blocks for IPC requests from both the application server and the update thread. The application server requests to read a file while an update thread requests to update a file. The main thread of the application module, maintains two queues to ensure that the file is not accessed by a writer (update thread) while the application server is reading it (to transmit it to the proxy server) and vice-versa.

On receiving a request from the proxy, the application server contacts the application module through an IPC call requesting for access to the required document (IPC_READ_REQUEST). If there are no ongoing updates to the document, the application module sends back an IPC message giving it access to the document (IPC_READ_PROCEED), and queues the request ID in its *Read Queue*. Once the application server is done with reading the document, it sends the application module another IPC message informing it about the end of the access to the document (IPC_READ_DONE). The application module, then deletes the corresponding entry from its *Read Queue*.

When a document is to be updated (either due to an update server interaction or an update query from the user), the update request is handled by the *update thread*. On getting an update request, the update thread initiates an IPC message to the application module (IPC_UPDATE_REQUEST). The application module on seeing this, checks its *Read Queue*. If the *Read Queue* is empty, it immediately sends an IPC message (IPC_UPDATE_PROCEED) to the update thread and queues the request ID in its *Update Queue*. On the other hand, if the *Read Queue* is not empty, the update request is still queued in the *Update Queue*, but the IPC_UPDATE_PROCEED message is not sent back to the update thread (forcing it to hold the update), until the *Read Queue* becomes empty. In either case, no further read requests from the application server are allowed to proceed; instead the application module queues them in its *Update Queue*, after the update request. Once the update thread has completed the update, it sends an IPC_UPDATE_DONE message to the update module. At this time, the application module deletes the update request entry from its *Update Queue*, sends IPC_READ_PROCEED messages for every read request queued in the *Update Queue* and queues these read requests in the *Read Queue*, to indicate that these are the current readers of the document.

It is to be noted that if the *Update Queue* is not empty, the first request queued will be an update request and all other requests in the queue will be read requests. Further, if the *Read Queue* is empty, the update is currently in progress. Table 10.1 tries to summarize this information.

Table 10.1: IPC message rules

| IPC_TYPE | Read Queue State | Update Queue State | Rule |
|---|---|---|---|
| IPC_READ_REQUEST | Empty | Empty | 1. Send IPC_READ_PROCEED to proxy<br>2. Enqueue Read Request in Read Queue |
| IPC_READ_REQUEST | Not Empty | Empty | 1. Send IPC_READ_PROCEED to proxy<br>2. Enqueue Read Request in Read Queue |
| IPC_READ_REQUEST | Empty | Not Empty | 1. Enqueue Read Request in Update Queue |
| IPC_READ_REQUEST | Not Empty | Not Empty | Enqueue the Read Request in the Update Queue |
| IPC_READ_DONE | Empty | Not Empty | Erroneous State. Not Possible. |
| IPC_READ_DONE | Not Empty | Empty | 1. Dequeue one entry from Read Queue. |
| IPC_READ_DONE | Not Empty | Not Empty | 1. Dequeue one entry from Read Queue<br>2. If Read Queue is *now* empty, Send IPC_UPDATE_PROCEED to head of Update Queue |
| IPC_UPDATE_REQUEST | Empty | Empty | 1. Enqueue Update Request in Update Queue<br>2. Send IPC_UPDATE_PROCEED |
| IPC_UPDATE_REQUEST | Empty | Not Empty | Erroneous state. Not Possible |
| IPC_UPDATE_REQUEST | Not Empty | Empty | 1. Enqueue Update Request in Update Queue |
| IPC_UPDATE_REQUEST | Not Empty | Not Empty | Erroneous State. Not possible |
| IPC_UPDATE_DONE | Empty | Empty | Erroneous State. Not possible |
| IPC_UPDATE_DONE | Empty | Not Empty | 1. Dequeue Update Request from Update Queue<br>2. For all Read Requests in Update Queue:<br>- Dequeue Read Requests from Update Queue<br>- Send IPC_READ_PROCEED<br>- Enqueue in Read Queue |
| IPC_UPDATE_DONE | Not Empty | Not Empty | Erroneous State. Not Possible. |

### 10.2.2 Strong Coherency Model using the Extended Sockets Interface

In this section, we point out several optimizations possible in the design described, using the advanced features provided by the extended sockets interface over InfiniBand. In Section 12.2 we provide the performance achieved by the extended sockets optimized architecture.

As described earlier, on every request the proxy module needs to validate the cache corresponding to the document requested. In traditional protocols such as TCP/IP, this requires the proxy module to send a version request message to the *version thread*[4], followed by the *version thread* explicitly sending the version number back to the proxy module. This involves the overhead of the TCP/IP protocol stack for the communication in both directions. Several researchers have provided solutions such as SDP to get rid of the overhead associated with the TCP/IP protocol stack while maintaining the sockets API. However, the more important concern in this case is the processing required at the version thread (e.g. searching for the index of the requested file and returning the current version number).

Application servers typically tend to perform several computation intensive tasks including executing CGI scripts, Java applets, etc. This results in a tremendously high CPU requirement for the main application server itself. Allowing an additional version thread to satisfy version requests from the proxy modules results in a high CPU usage for the module itself. Additionally, the large amount of computation carried out on the node by the application server results in significant degradation in performance for the version thread and other application modules running on

[4] Version Thread is a separate thread spawned by the application module to handle version requests from the proxy module

the node. This results in a delay in the version verification leading to an overall degradation of the system performance.

In this scenario, it would be of great benefit to have a one-sided communication operation where the proxy module can directly check the current version number without interrupting the version thread. Extended sockets over InfiniBand provides an one-sided GET operation which allows the initiator node to directly read data from the remote node's memory. This feature of extended sockets makes it an ideal choice for this scenario. In our implementation, we rely on the GET operation for the proxy module to get information about the current version number of the required file. Figure 10.5 demonstrates the extended sockets optimized coherency architecture.

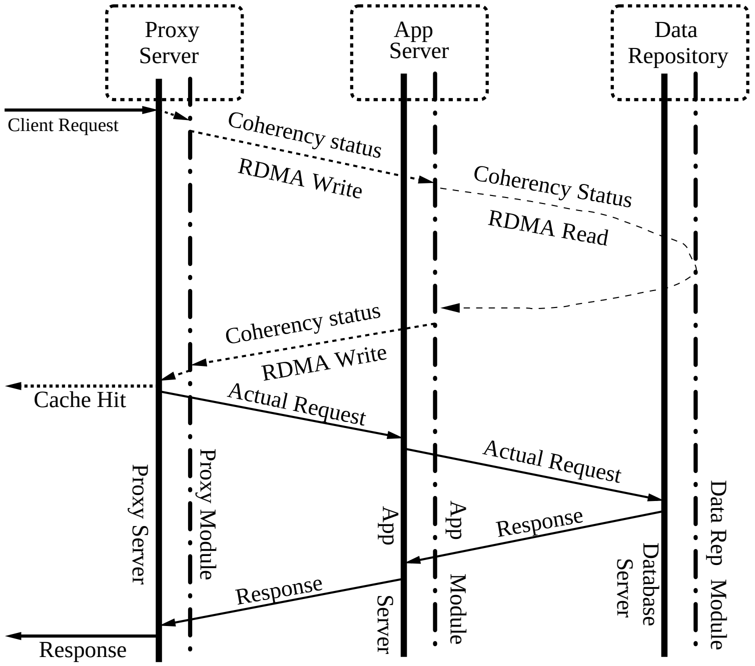

Figure 10.5: Strong Cache Coherency Protocol: Extended Sockets based Optimizations

## 10.3 Design of Reconfiguration Based on Remote Memory Operations

In this section, we describe the basic design issues in the dynamic reconfigurability scheme and the details about the implementation of this scheme using the extended sockets interface over InfiniBand.

### 10.3.1 Reconfigurability Support

Request patterns seen over a period of time, by a shared data-center, may vary significantly in terms of the ratio of requests for each co-hosted web-site. For example, interesting documents or dynamic web-pages becoming available and unavailable might trigger bursty traffic for some web-site at some time and for some other web-site at a different time. This naturally changes the resource requirements of a particular co-hosted web site from time to time. The basic idea of reconfigurability is to utilize the idle nodes of the system to satisfy the dynamically varying resource requirements of each of the individual co-hosted web-sites in the shared data-center. Dynamic reconfigurability of the system requires some extent of functional equivalence between the nodes of the data-center. We provide this equivalence by enabling software homogeneity such that each node is capable of belonging to any web-site in the shared data-center. Depending on the current demands (e.g., due to a burst of requests to one web-site), nodes reconfigure themselves to support these demands.

**Support for Existing Applications:** A number of applications have been developed in the data-center environment over the span of several years to process requests and provide services to the end user; modifying them to allow dynamic reconfigurability is impractical. To avoid making these cumbersome changes to the existing applications, our design makes use of *external helper modules* which work alongside the applications to provide effective dynamic reconfiguration. Tasks related to system load monitoring, maintaining global state information, reconfiguration, etc. are handled by these helper modules in an application transparent manner. These modules, running on each node in the shared data-center, reconfigure nodes in the data-center depending on current request and load patterns. They use the run-time configuration files of the data-center applications to reflect these changes. The servers on the other hand, just continue with the request processing, unmindful of the changes made by the modules.

**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.

**Remote Memory Operations Based Design:** As mentioned earlier, by their very nature the server nodes are compute intensive. Execution of CGI-Scripts, business-logic, servlets, database processing, etc. are typically very taxing on the server CPUs. So, the helper modules can potentially be starved for CPU on these servers. Though in theory the helper modules on the servers can be used to share the load information through explicit two-sided communication, in practice, such communication does not perform well [64].

Extended sockets over InfiniBand, on the other hand, provides one-sided remote memory operations (like RDMA and Remote Atomics) that allow access to remote memory without interrupting the remote node. In our design, we use these operations to perform load-balancer based server reconfiguration in a server transparent manner. Since the load-balancer is performing the reconfiguration with no interruptions to the server CPUs, this one-sided operation based design is highly resilient to server load.

The major design challenges and issues involved in dynamic adaptability and reconfigurability of the system are listed below.

- Providing a System Wide Shared State
- Concurrency Control to avoid Live-locks and Starvation
- Avoiding server thrashing through history aware reconfiguration
- Tuning the reconfigurability module sensitivity

We present these challenges in the following few sub-sections.

**System Wide Shared State**

As discussed earlier, the external helper modules present in the system handle various issues related to reconfigurability. However, the decision each module 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 modules 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 the traditional TCP/IP based 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.

Further, as mentioned earlier and observed in our previous work [64], due to various factors such as the skew and the load on the server nodes, even a simple two sided communication operation might lead to a significant degradation in the performance.

On the other hand, extended sockets provides several advanced features such as one-sided communication operations. In our implementation, each node writes information related to itself on its local memory. Other nodes needing this information can directly read this information using a *GET* 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 in Section 10.3.1 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 10.6, 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 10.7, 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.

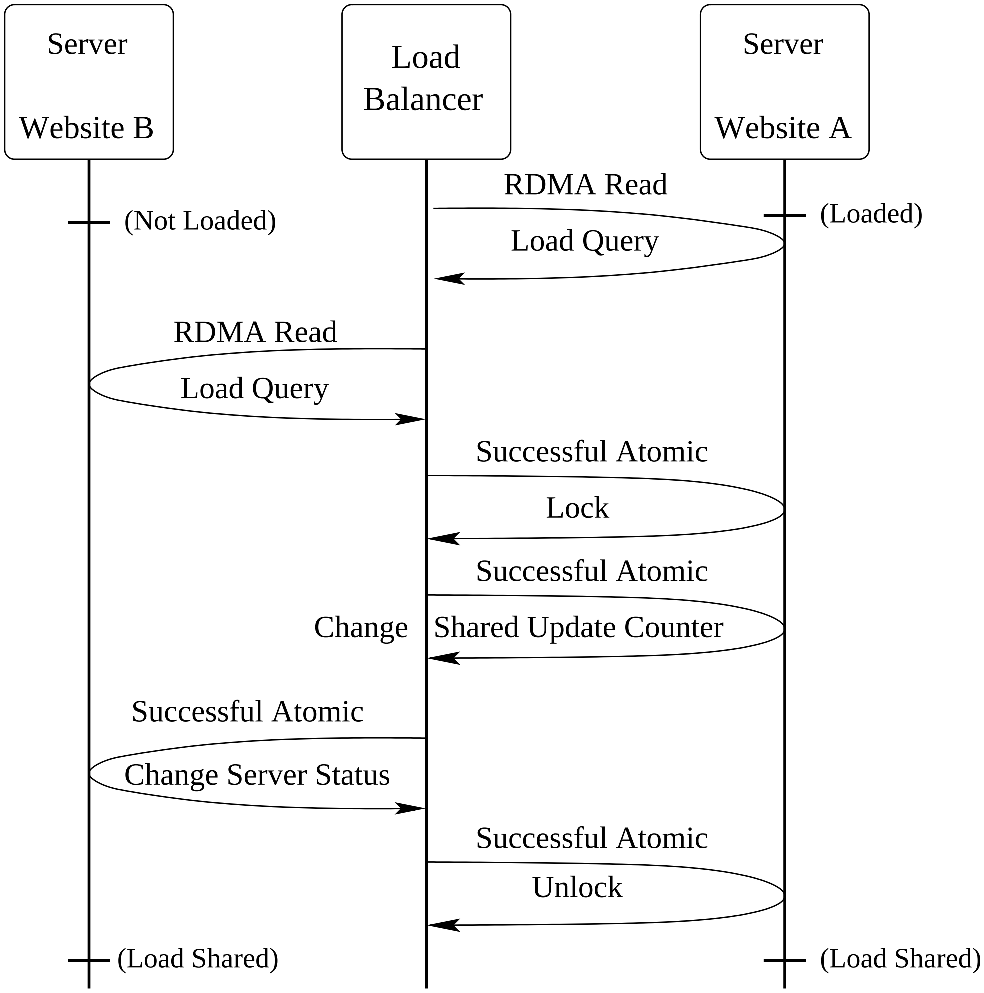

Figure 10.6: Concurrency Control for Shared State

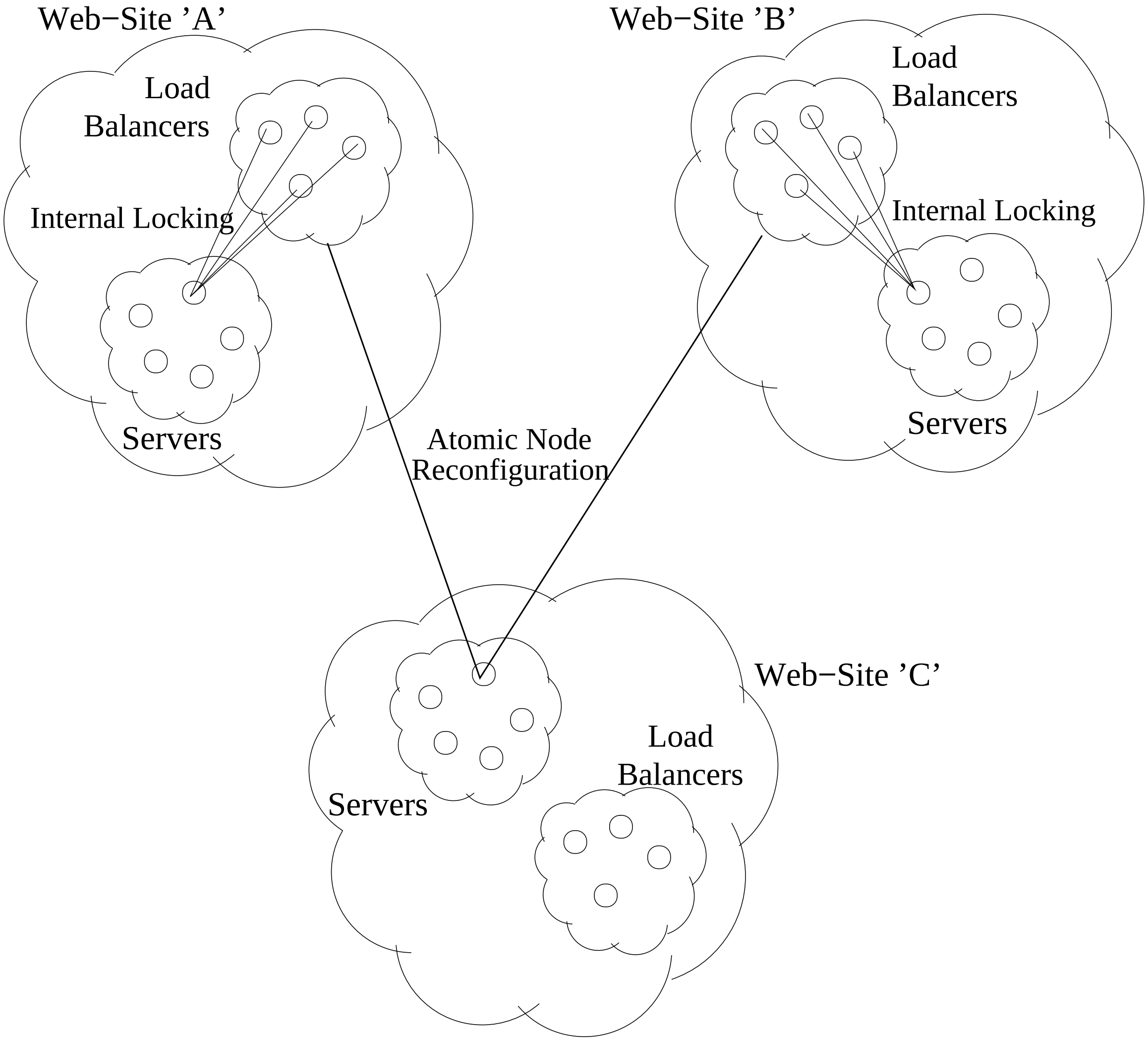

Figure 10.7: Hot-Spot Avoidance with Hierarchical Locking

### History Aware Reconfiguration

Due to the irregular nature of the incoming requests, a small burst of similar requests might potentially trigger a re-configuration in the data-center. Because of this, small bursts of similar requests can cause nodes in the shared data-center to be moved between the various co-hosted web-sites to satisfy the instantaneous load, resulting in *thrashing* in the data-center configuration.

To avoid such *thrashing*, in our scheme, we allow a history aware reconfiguration of the nodes, i.e., the nodes serving one web-site are re-allocated to a different web-site only if the load to the second web-site stays high for a pre-defined period of time $T$. However, this approach has its own trade-offs. A small value for $T$ could result in *thrashing* in the data-center environment. On the other hand, a large value of $T$ could make the approach less respondent to bursty traffic providing a similar performance as that of the non-reconfigurable or rigid system. The optimal value of $T$ depends on the kind of workload and request pattern. While we recognize the importance of the value of $T$, in this chapter, we do not concentrate on the effect of its variation and fix it to a pre-defined value for all the experiments.

**Reconfigurability Module Sensitivity**

As mentioned earlier, the modules on the load-balancers occasionally read the system information from the shared state in order to decide the best configuration at that instant of time. The time interval between two consecutive checks is a system parameter $S$ referring to the sensitivity of the external helper modules. A small value of $S$ allows a high degree of sensitivity, i.e., the system is better respondent to a variation in the workload characteristics. However, it would increase the overhead on the system due to the frequent monitoring of the state. On the other hand, a large value of $S$ allows a low degree of sensitivity, i.e., the system is less respondent to variation in the workload characteristics. At the same time, it would also result in a lower overhead on the system to monitor the state.

## 10.4 Experimental Results

In this section, we first show the micro-benchmark level performance given by VAPI, SDP and IPoIB. Next, we analyze the performance of a cache-coherent 2-tier data-center environment. Cache coherence is achieved using the *Client-Polling* based approach in the architecture described in Section 10.2.

All our experiments used a cluster system consisting of 8 nodes 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 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.2.0-build-001. The adapter firmware version is fw-23108-rel-1_18_0000. We used the Linux 2.4.7-10 kernel.

### 10.4.1 Micro-benchmarks

In this section, we compare the ideal case performance achievable by IPoIB and InfiniBand VAPI using a number of micro-benchmark tests. Since extended sockets defaults back to SDP for regular sockets function calls and to the native VAPI interface for extended calls such as one-sided GET/PUT operations, its performance is very similar to SDP or VAPI (depending on what calls are used). Thus, these numbers are not explicitly shown in this figure.

Figure 10.8 a shows the one-way latency achieved by IPoIB, VAPI Send-Receive, RDMA Write, RDMA Read and SDP for various message sizes. Send-Receive achieves a latency of around 7.5$\mu$s for 4 byte messages compared to a 30$\mu$s achieved by IPoIB, 27$\mu$s achieved by SDP and 5.5$\mu$s and 10.5$\mu$s achieved by RDMA Write and RDMA Read, respectively. Further, with increasing message sizes, the difference between the latency achieved by native VAPI, SDP and IPoIB tends to increase.

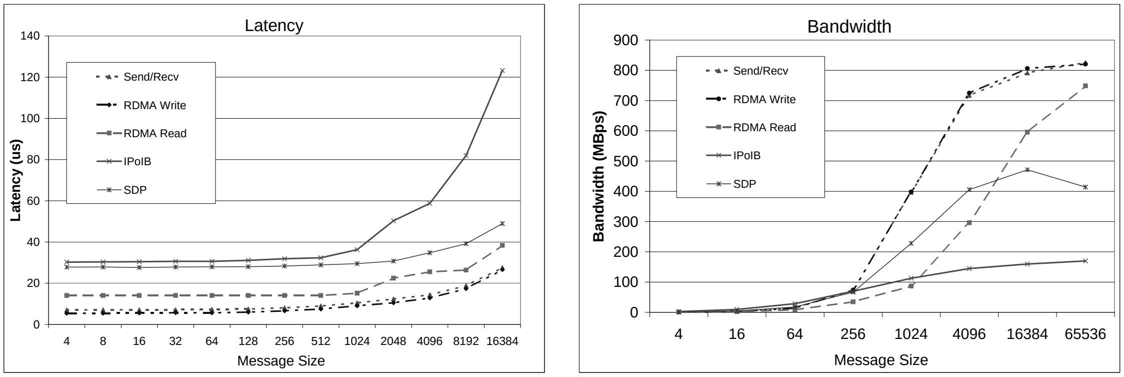

Figure 10.8: Micro-Benchmarks: (a) Latency, (b) Bandwidth

Figure 10.8b shows the uni-directional bandwidth achieved by IPoIB, VAPI Send-Receive and RDMA communication models and SDP. VAPI Send-Receive and both RDMA models perform comparably with a peak throughput of up to 840Mbytes/s compared to the 169Mbytes/s achieved by IPoIB and 500Mbytes/s achieved by SDP. We see that VAPI is able to transfer data at a much higher rate as compared to IPoIB and SDP. This improvement in both the latency and the bandwidth for VAPI compared to the other protocols is mainly attributed to the zero-copy communication in all VAPI communication models.

### 10.4.2 One-sided vs Two-sided Communication

In this section, we present performance results showing the impact of the loaded conditions in the data-center environment on the performance of RDMA Read and IPoIB. The results for the other communication models can be found in [17].

We emulate the loaded conditions in the data-center environment by performing background computation and communication operations on the server while the load-balancer performs the test with a separate thread on the server. This environment emulates a typical cluster-based shared data-center environment where multiple server nodes communicate periodically and exchange messages, while the load balancer, which is not as heavily loaded, attempts to get the load information from the heavily loaded machines.

The performance comparison of RDMA Read and IPoIB for this experiment is shown in Figure 10.9. We observe that the performance of IPoIB degrades significantly with the increase in the background load. On the other hand, one-sided communication operations such as RDMA show absolutely no degradation in the performance. These results show the capability of one-sided communication primitives in the data-center environment.

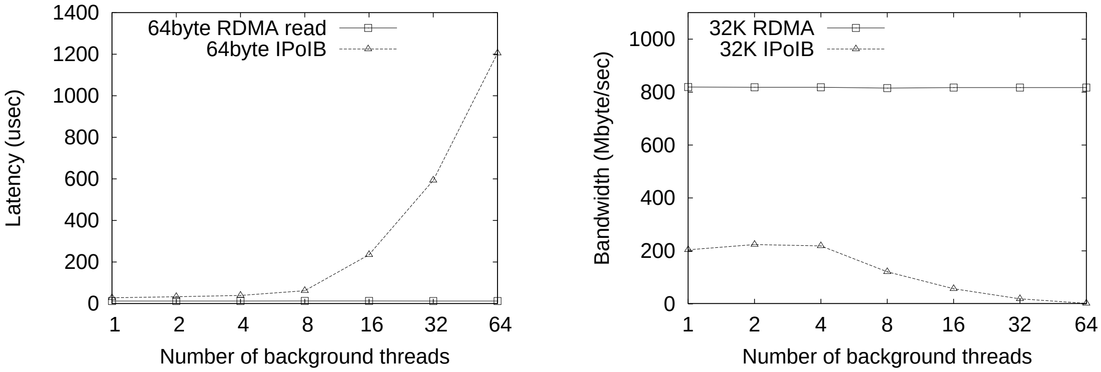

Figure 10.9: Performance of IPoIB and RDMA Read with background threads: (a) Latency and (b) Bandwidth

### 10.4.3 Strong Cache Coherence

In this section, we analyze the performance of a cache-coherent 2-tier data-center environment consisting of three proxy nodes and one application server running

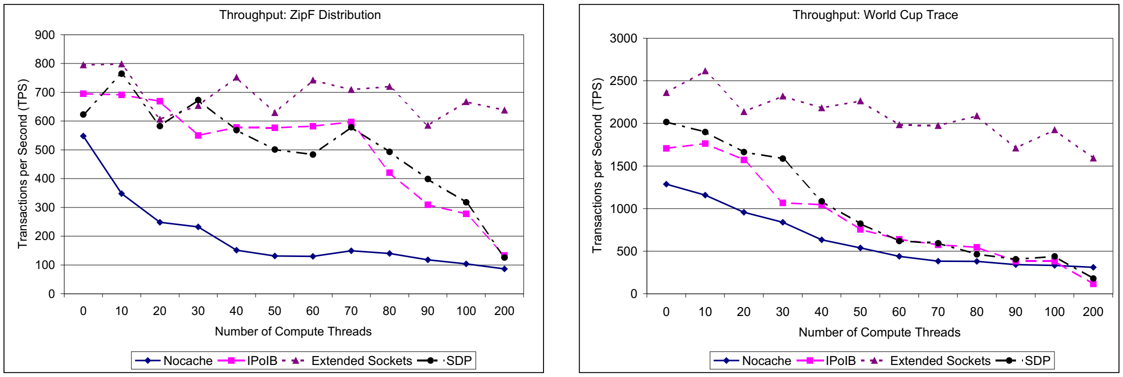

Figure 10.10: Data-Center Throughput: (a) Zipf Distribution, (b) WorldCup Trace

Apache-2.0.52. Cache coherency was achieved using the *Client-Polling* based approach described in Section 10.2. We used three client nodes, each running three threads, to fire requests to the proxy servers.

Two kinds of traces were used for the results. The first trace consists of 20 files of sizes varying from 200bytes to 1Mbytes. The access frequencies for these files follow a Zipf distribution [85]. The second trace is a 20,000 request subset of the WorldCup trace [9]. For all experiments, accessed documents were randomly updated by a separate update server with a delay of one second between the updates.

The HTTP client was implemented as a multi-threaded parallel application with each thread independently firing requests at the proxy servers. Each thread could either be executed on the same physical node or on a different physical nodes. The architecture and execution model is similar to the WebStone workload generator [62].

As mentioned earlier, application servers are typically compute intensive mainly due to their support to several compute intensive applications such as CGI script execution, Java applets, etc. This typically spawns several compute threads on the application server node using up the CPU resources. To emulate this kind of behavior, we run a number of compute threads on the application server in our experiments.

Figure 10.10(a) shows the throughput for the first trace (ZipF distribution). The x-axis shows the number of compute threads running on the application server node. The figure shows an evaluation of the proposed architecture implemented using IPoIB, SDP and Extended sockets and compares it with the response time obtained in the absence of a caching mechanism. We can see that the proposed architecture performs equally well for all three (IPoIB, SDP and Extended sockets) for a low number of compute threads; All three achieve an improvement of a factor of 1.5 over the no-cache case. This shows that two-sided communication is not a huge bottleneck in the module as such when the application server is not heavily loaded.

As the number of compute threads increases, we see a considerable degradation in the performance in the no-cache case as well as the Socket-based implementations using IPoIB and SDP. The degradation in the no-cache case is quite expected, since all the requests for documents are forwarded to the back-end. Having a high compute load on the back-end would slow down the application server's replies to the proxy requests.

The degradation in the performance for the Client-Polling architecture with IPoIB and SDP is attributed to the two sided communication of these protocols and the context switches taking place due to the large number of threads. This results in a significant amount of time being spent by the application modules just to get access to the system CPU. It is to be noted that the version thread needs to get access to the system CPU on every request in order to reply back to the proxy module's version number requests.

On the other hand, the Client-Polling architecture with extended sockets does not show any significant drop in performance. This is attributed to the one-sided RDMA operations supported by InfiniBand that are completely performed on hardware. For example, the version number retrieval from the version thread is done by the proxy module using an extended sockets GET operation. That is, the version thread does not have to get access to the system CPU; the proxy thread can retrieve the version number information for the requested document without any involvement of the version thread.

The throughput achieved by the WorldCup trace (Figure 10.10(b)) also follows the same pattern as above. With a large number of compute threads already competing for the CPU, the wait time for this remote process to acquire the CPU can be quite high, resulting in this degradation of performance.

### 10.4.4 Performance of Reconfigurability

In this section, we present the basic performance benefits achieved by the dynamic reconfigurability scheme as compared to a traditional data-center which does not have any such support.

**Performance with Burst Length**

In this section, we present the performance of the dynamic reconfigurability scheme as compared to the rigid configuration and the over-provisioning schemes in a data-center hosting three web-sites (nodes alloted in the ratio 3:3:2). For the workload, we have used a single file trace with a file size of 1 KB. Results for other file sizes, ZipF based traces [85] and a real World-Cup trace [9] can be found in [17].

Figure 10.11 shows the performance of the dynamic reconfigurability scheme as compared to the rigid configuration and over-provisioning schemes for varying burst length in the traffic. In a rigid configuration if there is a burst of traffic for the first website only three nodes are used and the remaining 5 nodes are relatively idle. The rigid configuration achieves an aggregate throughput of about 26,000 Transactions Per Second (TPS) in this scenario. In the over-provisioning scheme, a maximum of 6 nodes are assigned to the web-site being loaded. The maximum throughput achievable in this best configuration is around 51,000 TPS. However, in the reconfiguration scheme we see that the performance depends mainly on the burstiness of the traffic. If the burst length is too short, reconfiguration seems to perform comparably with the rigid scheme but for huge bursts reconfiguration achieves performance close to that of the over-provisioning scheme. The performance of reconfiguration for low burst lengths is comparable with the rigid scheme mainly due to the switching time overhead, i.e., the time required for the nodes to be reconfigured to the optimal configuration. This switching time, however, can be tuned by varying the sensitivity value addressed in Section 10.3.1. For high bursts, the reconfiguration switching time is negligible and gets amortized.

### Node Utilization

In case of shared data-centers, the logically partitioned sets of nodes serve the individual web-sites. Due to this partitioning and possible unbalanced request load, the nodes serving a particular web-site might be over-loaded even when other nodes in the system are not being utilized. These un-utilized servers could typically be used to share the load on the loaded web-site to yield better overall data-center performance.

In this section we measure the effective node utilization of our approach and compare it with the rigid and the over-provisioned cases. The effective node utilization

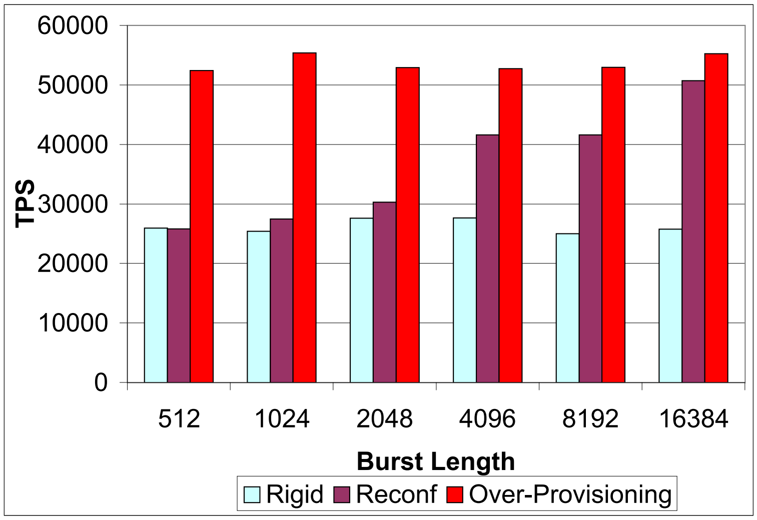

Figure 10.11: Impact of Burst Length

is measured as the total number of nodes being fully used by the data-center to serve a particular web-site.

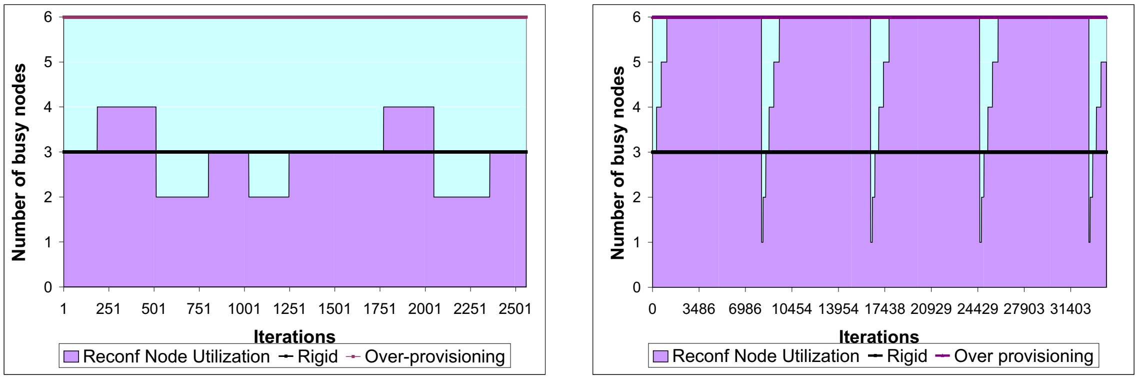

Figure 10.12: Node Utilization in a data-center hosting 3 web-sites with burst length (a) 512 (b) 8k

Figure 10.12 shows the node utilization for a shared data-center having three co-hosted web-sites. The rigid case has a constant node utilization of three nodes since it is statically configured. The over-provisioned case can use a maximum of 6 nodes (leaving 1 node for each of the other web-sites).

In figure 10.12a, we can see that for non-bursty traffic (burst length = 512), the reconfiguration scheme is not able to completely utilize the maximum available nodes because the switching time between configurations is comparable to the time required to serve the burst length of requests. It is to be noted that it performs comparably with the rigid scheme.

Further, nodes switching to one of the web-sites incurs a penalty for the requests coming to the other web-sites. For example, a burst of requests for web-site 'A' might cause all the nodes to shift accordingly. So, at the end of this burst web-site 'A' would have six nodes while the other 2 web-sites have one node each. At this time, a burst of requests to any of the other web-sites would result in a node utilization of one. This causes a drop in the number of nodes used for reconfigurability as shown in the figure.

Figure 10.12b shows the node utilization with a burst length of 8096. We can see that for large burst lengths the switching time for our reconfigurability scheme is negligible. And for large periods of time, the maximum number of nodes are fully utilized.

## 10.5 Summary

Several data-center applications and their extensions have been written using the sockets interface in the past. However, due to the uneven load conditions in the data-center environments, such an interface is not the best in several scenarios. For example, several enhancements previously proposed by researchers, such as the client-polling scheme for dynamic or active data caching and dynamic reconfigurability of resources, have suffered from mediocre performance benefits because of being throttled by the performance of the underlying sockets library and accordingly have not had much success.

In this chapter, we proposed an extended sockets interface that extends the standard sockets interface with advanced features offered by high-speed networks such as one-sided communication operations. We demonstrated that the extended sockets interface can provide close to order of magnitude benefits for several applications in the data-center environment, including active caching and dynamic reconfigurability schemes.

# CHAPTER 11

# SUPPORTING IWARP COMPATIBILITY AND FEATURES FOR REGULAR NETWORK ADAPTERS

Though TOEs have been able to handle most of the inefficiencies of the host-based TCP/IP stack, they are still plagued with some of the limitations in order to maintain backward compatibility with the existing infrastructure and applications. For example, the traditional sockets interface is often not the best interface to allow high performance communication [15, 56, 64, 17]. Several techniques used with the sockets interface (e.g., peek-and-post, where the receiver first posts a small buffer to read the header information and then decides the length of the actual data buffer to be posted) make it difficult to efficiently perform zero-copy data transfers with such an interface.

Several new initiatives by IETF such as iWARP and Remote Direct Data Placement (RDDP) [11], were started to tackle such limitations with basic TOEs and other POEs. The iWARP standard, when offloaded on to the network adapter, provides two primary extensions to the TOE stack: (i) it exposes a rich interface including zero-copy, asynchronous and one-sided communication primitives and (ii) it extends the TCP/IP implementation on the TOE to allow such communication while maintaining compatibility with the existing TCP/IP implementations.

With such aggressive initiatives in the offloading technology present on network adapters, the user market is now distributed amongst these various technology levels. Several users still use regular Ethernet network adapters (42.4% of the Top500 supercomputers use Ethernet with most, if not all, of them relying on regular Gigabit Ethernet adapters [51]) which do not perform any kind of protocol offload; then we have users who utilize the offloaded protocol stack provided with TOEs; finally with the advent of the iWARP standard, a part of the user group is also moving towards such iWARP-capable networks.

TOEs and regular Ethernet network adapters have been compatible with respect to both the data format sent out on the wire (Ethernet + IP + TCP + data payload) as well as with the interface they expose to the applications (both using the sockets interface). With iWARP capable network adapters, such compatibility is disturbed to some extent. For example, currently an iWARP-capable network adapter can only communicate with another iWARP-capable network adapter[5]. Also, the interface exposed by the iWARP-capable network is no longer sockets; it is a much richer and newer interface.

For a wide-spread usage, network architectures need to maintain compatibility with the existing and widely used network infrastructure. Thus, for a wide-spread acceptance of iWARP, two important extensions seem to be quite necessary.

1. Let us consider a scenario where a server handles requests from various client nodes (Figure 11.1). In this scenario, for performance reasons, it is desirable for the server to use iWARP for all communication (e.g., using an iWARP-capable network adapter). The client on the other hand might *NOT* be equipped with

[5] The intermediate switches, however, need not support iWARP.

an iWARP-capable network card (e.g., it might use a regular Fast Ethernet or Gigabit Ethernet adapter or even a TOE). For such and various other scenarios, it becomes quite necessary to have a software implementation of iWARP on such networks in order to maintain compatibility with the hardware offloaded iWARP implementations.

2. Though the iWARP interface provides a richer feature-set as compared to the sockets interface, it requires applications to be rewritten with this interface. While this is not a concern for new applications, it is quite cumbersome and impractical to port existing applications to use this new interface. Thus, it is desirable to have an interface which provides a two-fold benefit: (i) it allows existing applications to run directly without any modifications and (ii) it exposes the richer feature set of iWARP such as zero-copy, asynchronous and one-sided communication to the applications to be utilized with minimal modifications.

In general, we would like to have a software stack which would provide the above mentioned extensions for regular Ethernet network adapters as well as TOEs. In this chapter, however, we focus only on regular Ethernet adapters and design and implement a software stack to provide both these extensions. Specifically, (i) the software stack emulates the functionality of the iWARP stack in software to provide compatibility for regular Ethernet adapters with iWARP capable networks and (ii) it provides applications with an *extended* sockets interface that provides the traditional sockets functionality as well as functionality extended with the rich iWARP features.

## 11.1 Background

In this section, we provide a brief background about the iWARP standard.

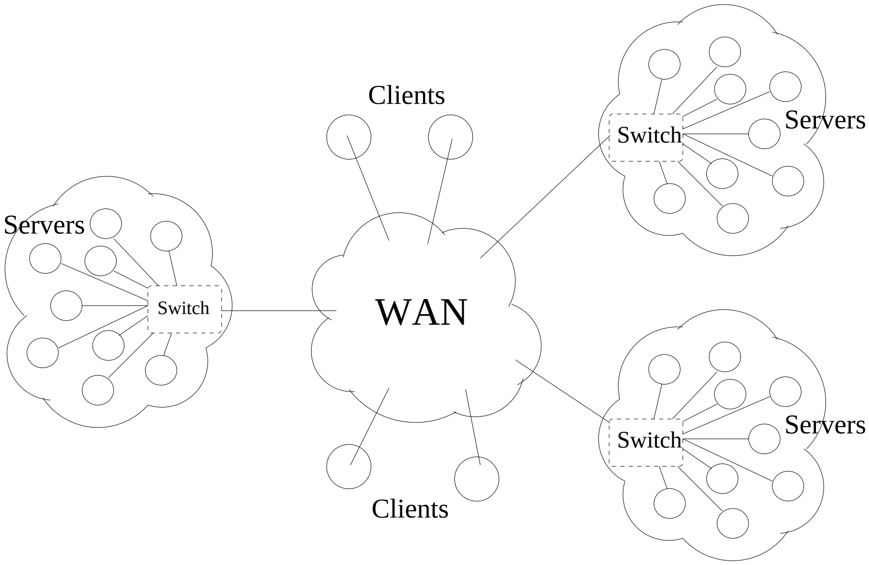

Figure 11.1: Multiple clients with regular network adapters communicating with servers using iWARP-capable network adapters.

### 11.1.1 iWARP Specification Overview

The iWARP standard comprises of up to three protocol layers on top of a reliable IP-based protocol such as TCP: (i) RDMA interface, (ii) Direct Data Placement (DDP) layer and (iii) Marker PDU Aligned (MPA) layer.

The RDMA layer is a thin interface which allows applications to interact with the DDP layer. The DDP layer uses an IP based reliable protocol stack such as TCP to perform the actual data transmission. The MPA stack is an extension to the TCP/IP stack in order to maintain backward compatibility with the existing infrastructure. Details about the DDP and MPA layers are provided in the subsequent sections.

**Direct Data Placement (DDP)**

The DDP standard was developed to serve two purposes. First, the protocol should be able to provide high performance in 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 an IP based reliable transport layer stack. Maintaining these two features involves novel designs for several aspects. We describe some of these in this section.

**In-Order Delivery and Out-of-Order Placement:** DDP relies on de-coupling of placement and delivery of messages, i.e., placing the data in the user buffer is performed in a decoupled manner with informing the application that the data has been placed in its buffer. In this approach, the sender breaks the message into multiple segments of MTU size; the receiver places each segment directly into the user buffer, performs book-keeping to keep track of the data that has already been placed and once all the data has been placed, informs the user about the arrival of the data. This approach has two benefits: (i) there are no copies involved in this approach and (ii) suppose a segment is dropped, the future segments do not need to be buffered till this segment arrives; they can directly be placed into the user buffer as and when they arrive. The approach used, however, involves two important features to be satisfied by each segment: Self-Describing and Self-Contained segments.

The Self-Describing property of segments involves adding enough information in the segment header so that each segment can individually be placed at the appropriate location without any information from the other segments. The information contained in the segment includes the Message Sequence Number (MSN), the Offset in the message buffer to which the segment has to be placed (MO) and others. Self-Containment of segments involves making sure that each segment contains either a part of a single message, or the whole of a number of messages, but not parts of more than one message.

**Middle Box Fragmentation:** DDP is an end-to-end protocol. The intermediate nodes do not have to support DDP. This means that the nodes which forward the segments between two DDP nodes, do not have to follow the DDP specifications. In other words, DDP is transparent to switches with IP forwarding and routing. However, this might lead to a problem known as "Middle Box Fragmentation" for Layer-4 or above switches.

Layer-4 switches are transport protocol specific and capable of making more intelligent decisions regarding the forwarding of the arriving message segments. The forwarding in these switches takes place at the transport layer (e.g., TCP). The modern load-balancers (which fall under this category of switches) allow a hardware based forwarding of the incoming segments. They support optimization techniques such as TCP Splicing [38] in their implementation. The problem with such an implementation is that, there need not be a one-to-one correspondence between the segments coming in and the segments going out. This means that the segments coming in might be re-fragmented and/or re-assembled at the switch. This might require buffering at the receiver node, since the receiver cannot recognize the DDP headers for each segments. This mandates that the protocol not assume the self-containment property at the receiver end, and add additional information in each segment to help recognize the DDP header.

### Marker PDU Aligned (MPA)

In case of “Middle Box Fragmentation”, the self-containment property of the segments might not hold true. The solution for this problem needs to have the following properties:

- It must be independent of the segmentation algorithm used by TCP or any layer below it.
- A deterministic way of determining the segment boundaries are preferred.
- It should enable out-of-order placement of segments. In the sense, the placement of a segment must not require information from any other segment.
- It should contain a stronger data integrity check like the Cyclic Redundancy Check (CRC).

The solution to this problem involves the development of the MPA protocol [43]. Figure 11.2 illustrates the new segment format with MPA. This new segment is known as the FPDU or the Framing Protocol Data Unit. The FPDU format has three essential changes:

- Markers: Strips of data to point to the DDP header in case of middle box fragmentation
- Cyclic Redundancy Check (CRC): A Stronger Data Integrity Check
- Segment Pad Bytes

The markers placed as a part of the MPA protocol are strips of data pointing to the MPA header and spaced uniformly based on the TCP sequence number. This

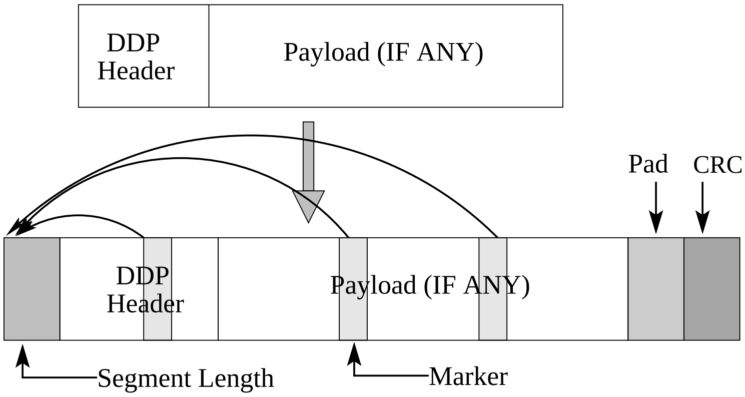

Figure 11.2: Marker PDU Aligned (MPA) protocol Segment format

provides the receiver with a deterministic way to find the markers in the received segments and eventually find the right header for the segment.

## 11.2 Designing Issues and Implementation Details

To provide compatibility for regular Ethernet network adapters with hardware offloaded iWARP implementations, we propose a software stack to be used on the various nodes. We break down the stack into two layers, namely, the *Extended sockets interface* and the *iWARP layer* as illustrated in Figure 11.3. Amongst these two layers, the *Extended sockets interface* is generic for all kinds of iWARP implementations; for example it can be used over the *software iWARP layer* for regular Ethernet networks presented in this chapter, over a *software iWARP layer* for TOEs, or over hardware offloaded iWARP implementations. Further, for the *software iWARP layer* for regular Ethernet networks, we propose two kinds of implementations: user-level iWARP and kernel-level iWARP. Applications, however, only interact with the extended sockets interface which in turn uses the appropriate iWARP stack available on the system.

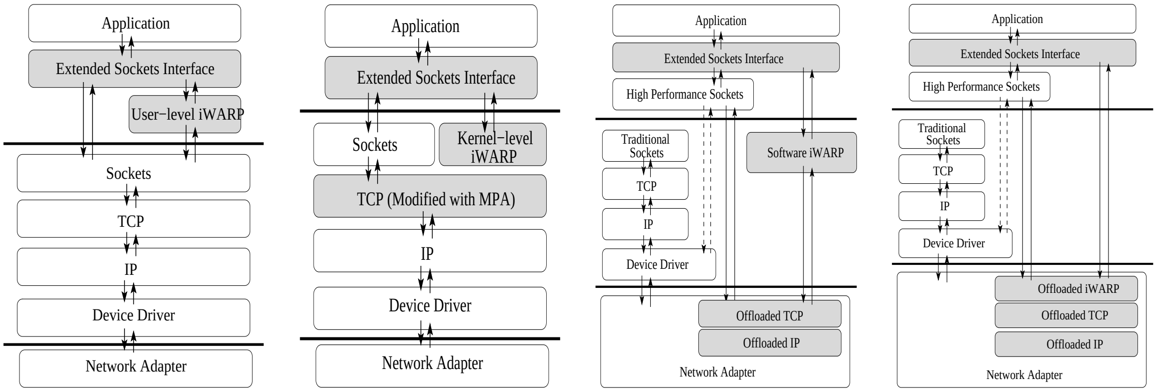

Figure 11.3: Extended sockets interface with different Implementations of iWARP: (a) User-Level iWARP (for regular Ethernet networks), (b) Kernel-Level iWARP (for regular Ethernet networks), (c) Software iWARP (for TOEs) and (d) Hardware offloaded iWARP (for iWARP-capable network adapters).

In this chapter, we only concentrate on the design and implementation of the stack on regular Ethernet network adapters (Figures 11.3a and 11.3b).

### 11.2.1 Extended Sockets Interface

The extended sockets interface is designed to serve two purposes. First, it provides a transparent compatibility for existing sockets based applications to run without any modifications. Second, it exposes the richer interface provided by iWARP such as zero-copy, asynchronous and one-sided communication to the applications to utilize as and when required with minimal modifications. For existing sockets applications (which do not use the richer extensions of the extended sockets interface), the interface just passes on the control to the underlying sockets layer. This underlying sockets layer could be the traditional host-based TCP/IP sockets for regular Ethernet networks or a High Performance Sockets layer on top of TOEs [44] or other

POEs [16, 18, 14]. For applications which *DO* use the richer extensions of the extended sockets interface, the interface maps the calls to appropriate calls provided by the underlying iWARP implementation. Again, the underlying iWARP implementation could be a software implementation (for regular Ethernet network adapters or TOEs) or a hardware implementation.

In order to extend the sockets interface to support the richer interface provided by iWARP, certain sockets based calls need to be aware of the existence of iWARP. The setsockopt() system call, for example, is a standard sockets call. But, it can be used to set a given socket to *IWARP_MODE*. All future communication using this socket will be transferred using the iWARP implementation. Further, `read()`, `write()` and several other socket calls need to check if the socket mode is set to *IWARP_MODE* before carrying out any communication. This requires modifications to these calls, while making sure that existing sockets applications (which do not use the extended sockets interface) are not hampered.

In our implementation of the extended sockets interface, we carried this out by overloading the standard *libc* library using our own extended sockets interface. This library first checks whether a given socket is currently in *IWARP_MODE*. If it is, it carries out the standard iWARP procedures to transmit the data. If it is not, the extended sockets interface dynamically loads the *libc* library to pass on the control to the traditional sockets interface for the particular call.

### 11.2.2 User-Level iWARP

In this approach, we designed and implemented the entire iWARP stack in user space above the sockets layer (Figure 11.3a). Being implemented in user-space and above the sockets layer, this implementation is very portable across various hardware and software platforms[6]. However, the performance it can deliver might not be optimal. Extracting the maximum possible performance for this implementation requires efficient solutions for several issues including (i) supporting gather operations, (ii) supporting non-blocking operations, (iii) asynchronous communication, (iv) handling shared queues during asynchronous communication and several others. In this section, we discuss some of these issues and propose various solutions to handle these issues.

**Gather operations supported by the iWARP specifications:** The iWARP specification defines gather operations for a list of data segments to be transmitted. Since, the user-level iWARP implementation uses TCP as the underlying mode of communication, there are interesting challenges to support this without any additional copy operations. Some of the approaches we considered are as follows:

1. 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. The second approach is to use the scatter-gather readv() and writev() calls provided by the traditional sockets interface. Though in theory traditional sockets supports scatter/gather of data using readv() and writev() calls, the actual implementation of these calls is specific to the kernel. It is possible (as is currently implemented in the 2.4.x linux kernels) that the data in these list

[6] Though the user-level iWARP implementation is mostly in the user-space, it requires a small patch in the kernel to extend the MPA CRC to include the TCP header too and to provide information about the TCP sequence numbers used in the connection in order to place the markers at appropriate places (this cannot be done from user-space).

of buffers be sent out as different messages and not aggregated into a single message. While this is perfectly fine with TCP, it creates a lot of fragmentation for iWARP, forcing it to have additional buffering to take care of this.

3. The third approach is to use the TCP_CORK mechanism provided by TCP/IP. The TCP_CORK socket option 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 and at the same time guaranteeing that all the segments are sent out as a single message.

**Non-blocking communication operations support:** As with iWARP, the extended sockets also supports non-blocking communication 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. In our approach, we use a multi-threaded design for user-level iWARP to allow non-blocking communication operations (Figure 11.4). As shown in the figure, the application thread posts a send and a receive to the asynchronous threads and returns control to the application; these asynchronous threads take care of the actual data transmission for send and receive, respectively. To allow the data movement between the threads, we use pthreads() rather than fork(). This approach gives the flexibility of a shared physical address space for the application and the asynchronous threads. The pthreads() specification states that all pthreads should share the same process ID (pid). Operating Systems such as Solaris follow this specification. However, due

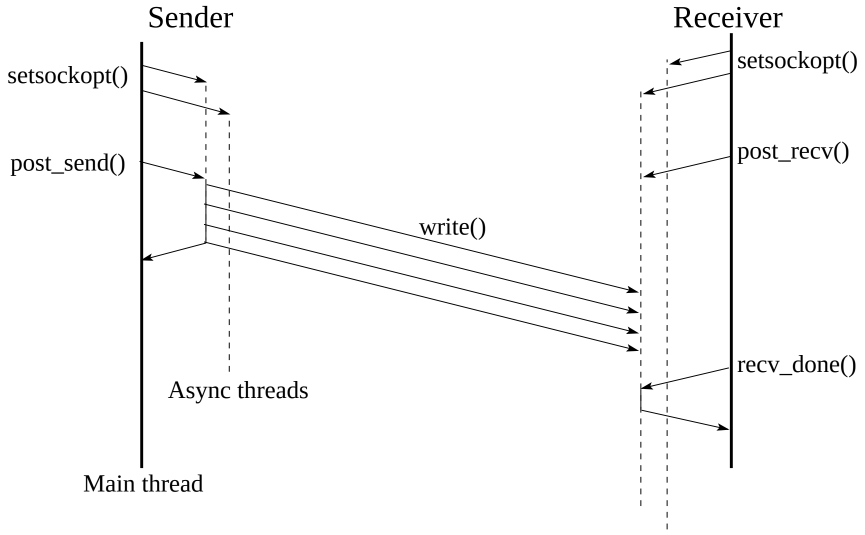

Figure 11.4: Asynchronous Threads Based Non-Blocking Operations

to the flat architecture of Linux, this specification was not followed in the Linux implementation. This means that all pthreads() have a different PID in Linux. We use this to carry out inter-thread communication using inter-process communication (IPC) primitives.

**Asynchronous communication supporting non-blocking operations:** In the previous issue (non-blocking communication operations support), we 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 iWARP specification does not require a shared queue for the multiple sockets in an application. Each socket has separate send and receive work queues where descriptors posted for that socket are placed. We use UNIX socket connections between the main thread and the asynchronous threads. The first socket set to *IWARP_MODE* 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 *IWARP_MODE* 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 does not affect the performance too much.

### 11.2.3 Kernel-Level iWARP

The kernel-level iWARP is built directly over the TCP/IP stack bypassing the traditional sockets layer as shown in Figure 11.3b. This implementation requires modifications to the kernel and hence is not as portable as the user-level implementation. However, it can deliver a better performance as compared to the user-level iWARP. The kernel-level design of iWARP has several issues and design challenges. Some of these issues and the solutions chosen for them are presented in this section.

Though most part of the iWARP implementation can be done completely above the TCP stack by just inserting modules (with appropriate symbols exported from the TCP stack), there are a number of changes that are required for the TCP stack itself. For example, ignoring the remote socket buffer size, efficiently handling out-of-order segments, etc. require direct changes in the core kernel. This forced us to recompile the linux kernel as a patched kernel. We have modified the base kernel.org kernel version 2.4.18 to the patched kernel to facilitate these changes.

**Immediate copy to user buffers:** Since iWARP provides non-blocking communication, copying the received data to the user buffers is a tricky issue. One simple solution is to copy the message to the user buffer when the application calls a completion function, i.e., when the data is received the kernel just keeps it with itself and when the application checks with the kernel if the data has arrived, the actual copy to the user buffer is performed. This approach, however, loses out on the advantages of non-blocking operations as the application has to block waiting for the data to be copied while checking for the completion of the data transfer. Further, this approach requires another kernel trap to perform the copy operation.

The approach we used in our implementation is to immediately copy the received message to the user buffer as soon as the kernel gets the message. An important issue to be noted in this approach is that since multiple processes can be running on the system at the same time, the current process scheduled can be different with the owner of the user buffer for the message; thus we need a mechanism to access the user buffer even when the process is not currently scheduled. To do this, we pin the user buffer (prevent it from being swapped out) and map it to a kernel memory area. This ensures that the kernel memory area and the user buffer point to the same physical address space. Thus, when the data arrives, it is immediately copied to the kernel memory area and is automatically reflected into the user buffer.

**User buffer registration:** The iWARP specification defines an API for the buffer registration, which performs pre-communication processes such as buffer pinning, address translation between virtual and physical addresses, etc. These operations are required mainly to achieve a zero-copy data transmission on iWARP offloaded network adapters. Though this is not critical for the kernel-level iWARP implementation as it anyway performs a copy, this can protect the buffer from being swapped out and avoid the additional overhead for page fetching. Hence, in our approach, we do pin the user-buffer.

**Efficiently handling out-of-order segments:** iWARP allows out-of-order placement of data. This means that out-of-order segments can be directly placed into the user-buffer without waiting for the intermediate data to be received. In our design, 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. Hence, 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.

## 11.3 Experimental Evaluation

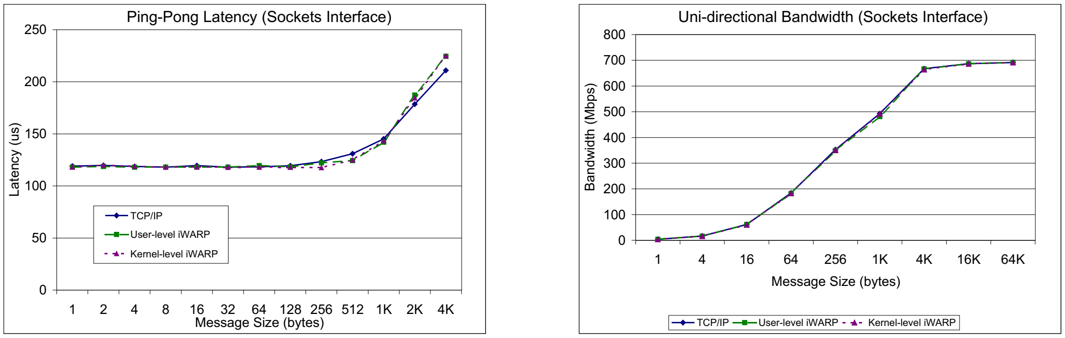

Figure 11.5: Micro-Benchmark Evaluation for applications using the standard sockets interface: (a) Ping-pong latency and (b) Uni-directional bandwidth

In this section, we perform micro-benchmark level experimental evaluations for the extended sockets interface using the user- and kernel-level iWARP implementations. Specifically, we present the ping-pong latency and uni-directional bandwidth achieved in two sets of tests. In the first set of tests, we measure the performance achieved for standard sockets based applications; for such applications, the extended sockets interface does basic processing to ensure that the applications do not want to utilize the extended interface (by checking if the *IWARP_MODE* is set) and passes on the control to the traditional sockets layer. In the second set of tests, we use applications which utilize the richer extensions provided by the extended sockets interface; for such applications, the extended sockets interface utilizes the software iWARP implementations to carry out the communication.

The latency test is carried out in a standard ping-pong fashion. The sender sends a message and waits for a reply from receiver. The time for this is recorded by the sender and it is divided by two to get the one-way latency. For measuring the bandwidth, a simple window based approach was followed. The sender sends $WindowSize$ number of messages and wait for a message from the receiver for every $WindowSize$ messages.

The experimental test-bed used is as follows: Two Pentium III 700MHz Quad machines, each with an L2-cache size of 1 MB and 1 GB of main memory. The interconnect was a Gigabit Ethernet network with Alteon NICs on each machine connected using a Packet Engine switch. We used the RedHat 9.0 linux distribution installed with the kernel.org kernel version 2.4.18.

The results for the applications with the standard unmodified sockets interface are presented in Figure 11.5. As shown in the figure, the extended sockets interface adds very minimal overhead to existing sockets applications for both the latency and the bandwidth tests.

For the applications using the extended interface, the results are shown in Figure 11.6. We can see that the user- and kernel-level iWARP implementations incur overheads of about 100$\mu$s and 5$\mu$s respectively, as compared to TCP/IP. There are several reasons for this overhead. First, the user- and kernel-level iWARP implementations are built over sockets and TCP/IP respectively; so they are not expected to give a better performance than TCP/IP itself. Second, the user-level iWARP implementation has additional threads for non-blocking operations and requires IPC between threads. Also, the user-level iWARP implementation performs locking for shared queues between threads. However, it is to be noted that the basic purpose of these implementations is to allow compatibility for regular network adapters with iWARP-capable network adapters and the performance is not the primary goal of these implementation. We can observe that both user- and kernel-level iWARP implementations can achieve a peak bandwidth of about 550Mbps. An interesting result in the figure is that the bandwidth of the user- and kernel-level iWARP implementations for small and medium message sizes is significantly lesser compared to TCP/IP. This is mainly because the iWARP implementations disable Nagle's algorithm in order to try to maintain message boundaries. For large messages, we see some degradation compared to TCP/IP due to the additional overhead of CRC data integrity performed by the iWARP implementations.

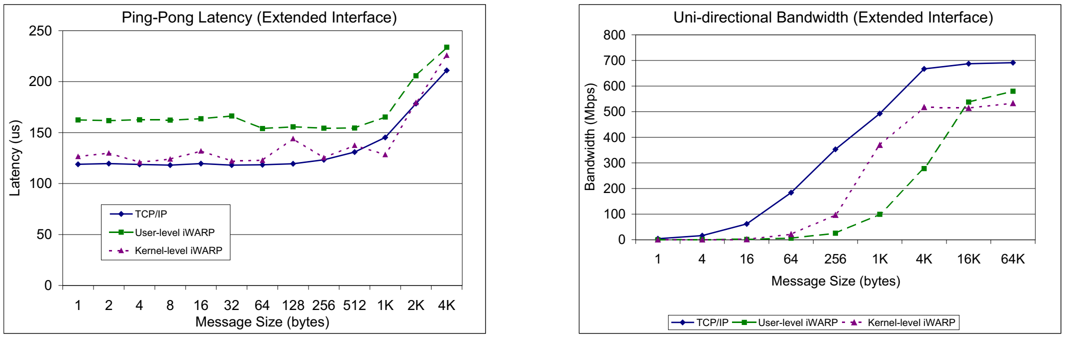

Figure 11.6: Micro-Benchmark Evaluation for applications using the extended iWARP interface: (a) Ping-pong latency and (b) Uni-directional bandwidth

## 11.4 Summary

Several new initiatives by IETF such as iWARP and Remote Direct Data Placement (RDDP) [11], were started to tackle the various limitations with TOEs while providing a completely new and feature rich interface for applications to utilize. For a wide-spread acceptance of these initiatives, however, two important issues need to be considered. First, software compatibility needs to be provided for regular network adapters (which have no offloaded protocol stack) with iWARP-capable network adapters. Second, the predecessors of iWARP-capable network adapters such as TOEs and host-based TCP/IP stacks used the sockets interface for applications to utilize them while the iWARP-capable networks provide a completely new and richer interface. Rewriting existing applications using the new iWARP interface is cumbersome and impractical. Thus, it is desirable to have an *extended* sockets interface which provides a two-fold benefit: (i) it allows existing applications to run directly without any modifications and (ii) it exposes the richer feature set of iWARP such as zero-copy, asynchronous and one-sided communication to the applications to be utilized with minimal modifications. In this research, we have designed and implemented a software stack to provide both these extensions.

# CHAPTER 12

# UNDERSTANDING THE ISSUES IN DESIGNING IWARP FOR 10-GIGABIT ETHERNET TOE ADAPTERS

The iWARP standard is designed around two primary goals: (i) to provide rich and advanced features including zero-copy, asynchronous and one-sided communication primitives to applications and (ii) to allow such communication while maintaining compatibility with the existing TCP/IP/Ethernet based infrastructure (e.g., switches, routers, load-balancers, firewalls).

In this chapter, we study the performance overheads associated with iWARP in maintaining this compatibility. In order to do this, we propose three implementations of iWARP over TCP offload engines – a software implementation (where the iWARP stack is completely implemented in software – Figure 11.3(c)), a NIC-offloaded implementation (where the iWARP stack is completely implemented on the network adapter – Figure 11.3(d)) and a hybrid hardware-software implementation (where part of the iWARP stack is implemented on the NIC and part of it on the host). It is to be noted that these three schemes only differ in where the iWARP stack is implemented; the TCP/IP stack is implemented in hardware (TOE) for all three approaches.

Our experimental results show that neither a completely software-based implementation nor a completely NIC-based implementation can provide the best performance for iWARP, mainly due to the overheads the stack introduces in order to maintain backward compatibility with the existing infrastructure. On the other hand, a hybrid hardware-software approach which utilizes both the host to handle some aspects and the NIC to handle the remaining aspects of the iWARP stack provides a better performance than either approach. Specifically, the hybrid approach achieves close to 2X better performance as compared to the hardware implementation and close to 4X better performance as compared to the software implementation.

## 12.1 Design Choices for iWARP over 10-Gigabit Ethernet

In this section, we will describe the different design choices for implementing a complete iWARP implementation over 10-Gigabit Ethernet. In particular, we describe three design choices: (i) Software iWARP, (ii) NIC-offloaded iWARP and (iii) Host-assisted iWARP.

### 12.1.1 Software iWARP Implementation

*Software iWARP* is a completely host-based implementation of iWARP, i.e., all aspects of the iWARP stack are implemented in software (Figure 12.1(a)). It is a generic implementation which can be used on any Ethernet adapter while maintaining complete compatibility with hardware iWARP implementations. There are several design aspects associated with improving the performance of this implementation, which we had described in our previous work [13]. The details of this design are provided in Chapter 11 and are skipped in this section.

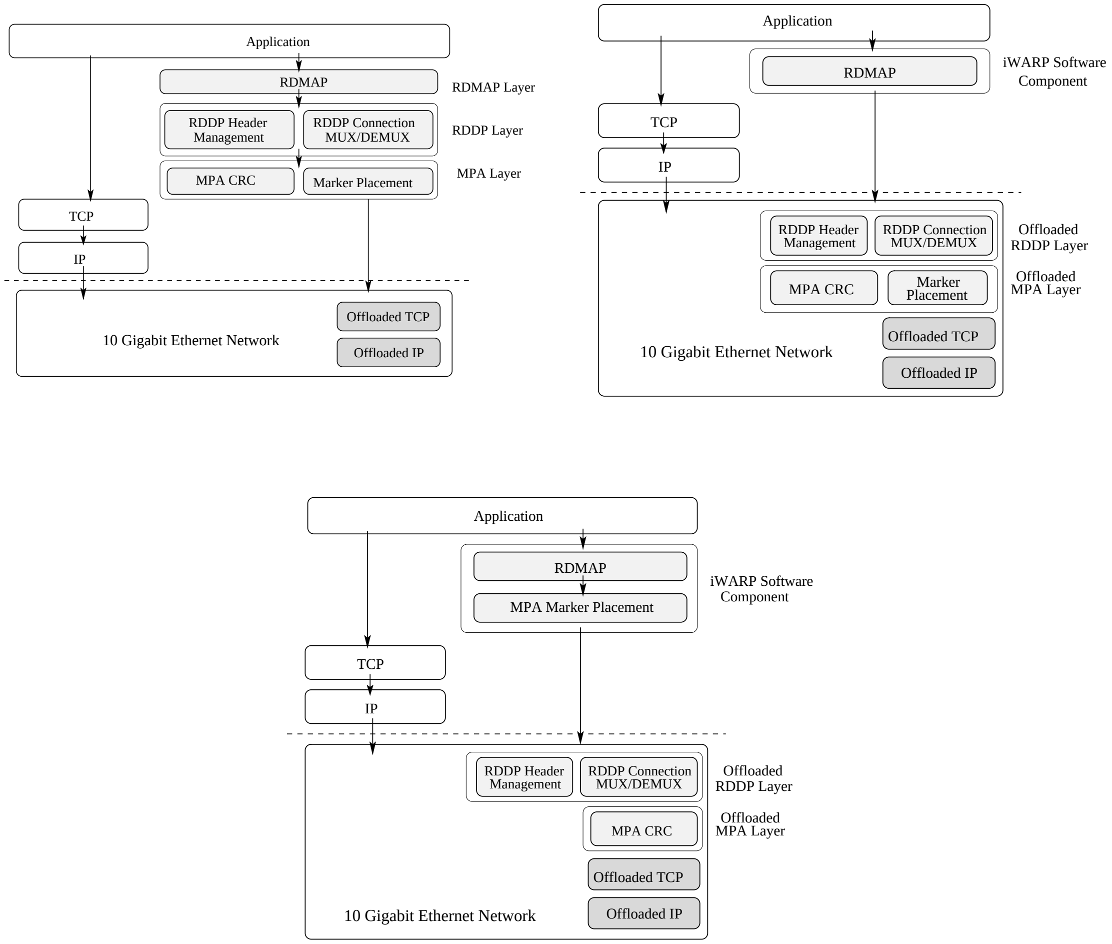

Figure 12.1: iWARP Implementations: (a) Software iWARP, (b) NIC-offloaded iWARP and (c) Host-assisted iWARP

### 12.1.2 NIC-offloaded iWARP Implementation

*NIC-offloaded iWARP* is a completely hardware-based implementation of iWARP, i.e., all aspects of the iWARP stack are implemented on the network adapter (Figure 12.1(b)). Specifically, the CRC, the marker placement, RDDP header generation and multiplexing/de-multiplexing of packets to connections is performed completely on the network adapterSince the Chelsio T110 network adapters are not completely programmable, the RDDP headers generated are actually garbage data; however, this does not change the correctness of our evaluation..

The iWARP specification states that the markers have to be placed at every 512 bytes in the data stream. Now, for the NIC-offloaded iWARP, this can be implemented in two ways:

1. In the first approach, the NIC can download large segments of data to the network adapter and move it around in NIC memory to insert the markers at appropriate locations. This approach deals with only one DMA transfer for the entire data from host memory to the NIC memory. However, for the markers to be inserted, two additional memory transactions are required on the NIC memory apart from the two memory transactions required for moving the data from host memory to the NIC and from the NIC to the wire, i.e., four memory transactions are required for transferring data in each direction on the network adapter. Thus, to saturate the uni-directional bandwidth of a 10Gbps link, a memory bandwidth of 40Gbps is required on the network adapter and several clock cycles to perform this processing itself. This makes the resource requirements on the NIC quite high.

2. In this approach, the NIC DMAs only (approximately) 512-byte segments from the host memory and adds appropriate markers at the end of each 512-byte segment. This approach does not require any additional NIC memory traffic for marker placement; however, this approach relies on DMAs of small 512-byte segments, thus increasing the number of DMA operations that are associated with a message transmission or reception.

In our design of the NIC-offloaded iWARP, we used the second approach based on multiple small DMA operations.

### 12.1.3 Hybrid Host-assisted iWARP Implementation

*Host-assisted iWARP* is a hybrid hardware-software implementation of iWARP (Figure 12.1(c)). Specifically, the CRC, RDDP header generation and multiplexing/de-multiplexing of packets to connections is performed on the network adapter. The marker placement, however, is done in host-space. We use a copy-based scheme to place the markers in the appropriate positions in this approach, i.e., the memory transactions involving placing the markers is performed at the host. In this approach, the NIC does not have to deal with any additional memory transactions or processing overhead associated with moving data to insert markers. However, the host has to perform additional processing, such as moving data in host-space in order to insert the appropriate markers.

## 12.2 Performance Results

In this section we describe our evaluation framework and compare the performance of the different designs.

### 12.2.1 Experimental testbed

We use a cluster of four nodes built around SuperMicro SUPER X5DL8-GG motherboards with ServerWorks GC LE chipsets, which include 133-MHz PCI-X interfaces. Each node has two Intel Xeon 3.0 GHz processors with a 512-KB cache and a 533 MHz front-side bus and 2 GB of 266-MHz DDR SDRAM. The nodes are connected with the Chelsio T110 10-Gigabit Ethernet TCP Offload Engines.

### 12.2.2 iWARP Evaluation

In this section, we evaluate the three different mechanisms proposed to implement iWARP. Software iWARP which is the most generic and portable approach [13], NIC offloaded iWARP described in Section 12.1.2 and Host Assisted iWARP described in Section 12.1.3. We use the standard ping-pong latency and unidirectional bandwidth microbenchmarks to compare these three different design choices.

As seen in Figure 12.2, for the small message latency as well as uni-directional throughput tests, the NIC-offloaded iWARP performs the best. The software iWARP naturally performs the worst because of the overhead of the protocol processing, as described in Section 12.1.1. However, when the message size increases, host-assisted iWARP starts to outperform the NIC-offloaded iWARP. This reversal in trend is attributed to the marker placement processing which is performed on the host for the host-assisted iWARP scheme and on the NIC for NIC-offloaded iWARP as discussed in Section 12.2.2. For large messages, the DMA operations become a significant bottleneck causing the saturation in performance as shown in Figure 12.2(b).

We also performed another experiment to evaluate the impact of the marker placement. We use the same latency, bandwidth tests described above, but vary the marker

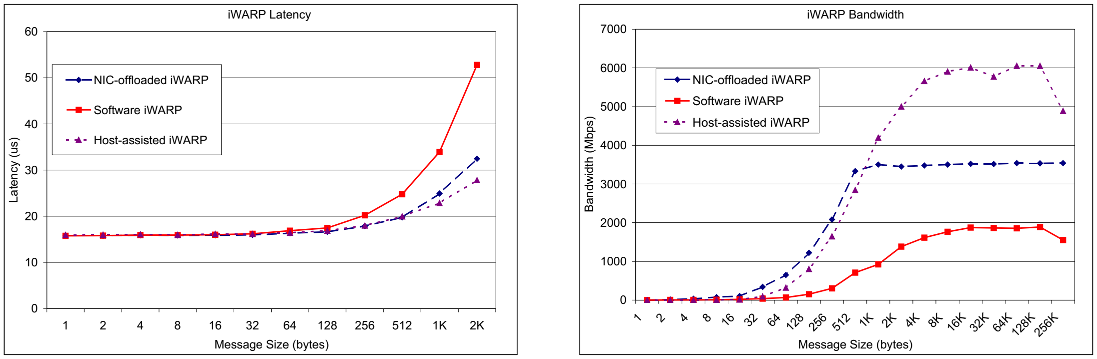

Figure 12.2: iWARP Micro-benchmarks: (a) Latency (b) Bandwidth

separation from 512 bytes (standard iWARP specification) to larger sizes (not compliant with the iWARP specification).

As shown in Figure 12.3, as the marker separation increases, NIC-offloaded iWARP starts improving in performance. We can draw the following conclusion that if we want complete compatibility with the specification, host assisted design is the best option. It is to be noted that though a larger marker separation size increases the buffering requirements on the network adapters for out-of-order arrived packets, in most SAN environments this is not a serious concern. In such environments, if complete compatibility is not an requirement, then the NIC-offloaded design with increased marker separation size is a good option.

## 12.3 Summary

The iWARP standard is designed around two primary goals: (i) to provide rich and advanced features including zero-copy, asynchronous and one-sided communication primitives to applications and (ii) to allow such communication while maintaining compatibility with the existing TCP/IP/Ethernet based infrastructure (e.g.,

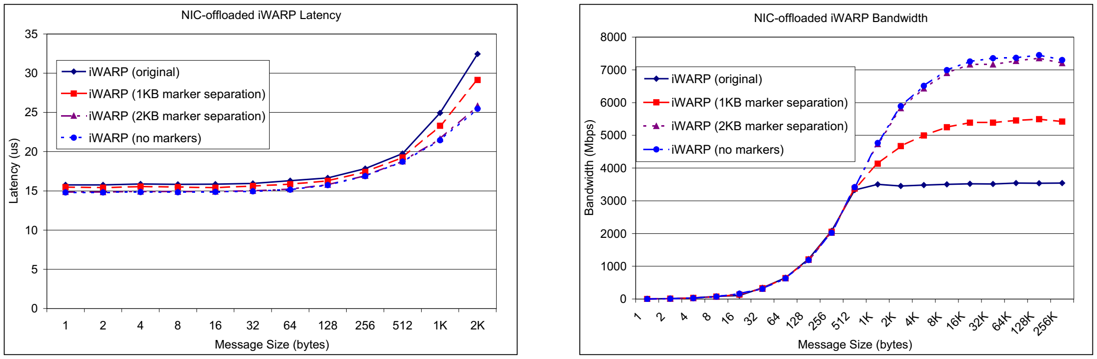

Figure 12.3: Impact of marker separation on iWARP performance: (a) Latency (b) Bandwidth

switches, routers, load-balancers, firewalls). In this chapter, we study the performance overheads associated with iWARP in maintaining this compatibility. In order to do this, we propose three implementations of iWARP – a complete software implementation (based on our previous work), a complete NIC-based implementation on the Chelsio T110 adapters and a hybrid hardware-software implementation. Our experimental results show that, when complete compatibility with the iWARP specification is required, neither a completely software-based implementation nor a completely NIC-based implementation can provide the best performance for iWARP, mainly due to the overheads the stack introduces in order to maintain backward compatibility with the existing infrastructure. On the other hand, a hybrid hardware-software approach which utilizes both the host to handle some aspects and the NIC to handle the remaining aspects of the iWARP stack provides a better performance than either approach. Specifically, the hybrid approach achieves close to 2X better performance as compared to the hardware implementation and close to 4X better performance as compared to the software implementation. On the other hand, when complete compatibility with the iWARP specification is not required, the NIC-offloaded iWARP performs the best.

# CHAPTER 13

# CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

With several high-speed networks being introduced in the market, each having its own communication interface or "language" that it exposes to end applications, portability for applications over various networks has become a topic of extensive research. Programming models such as Sockets, Message Passing Interface (MPI), Shared memory models, etc., have been widely accepted as the primary means for achieving such portability. This dissertation investigates the different design choices for implementing one such programming model, i.e., Sockets, in various high-speed network environments (e.g., InfiniBand, 10-Gigabit Ethernet).

## 13.1 Summary of Research Contributions

The dissertation targets three important sub-problems within the sockets programming model: (a) designing efficient sockets implementations to allow existing applications to be directly and transparently deployed on to clusters connected with high-speed networks; (b) analyzing the limitations of the sockets interface in various domains and extending it with features that applications need but are currently missing; and (c) designing a communication substrate to allow compatibility between various kinds of protocol stacks belonging to a common network family (e.g., Ethernet). In order to tackle these three sub-problems, we propose a framework consisting of three different components, namely the high performance sockets component, the extended sockets component and the wire-protocol compatibility component. We provide details of our findings in each of these three components in the following sub-sections.

### 13.1.1 High Performance Sockets

The high performance sockets aims at providing high performance for applications without requiring any modifications. We have designed various high performance sockets implementations for several networks including Gigabit Ethernet, GigaNet cLAN, InfiniBand and 10-Gigabit Ethernet as described in Chapters 2, 3, 4, 5, 6, 7 and 8. Our designs have shown performance improvements close to 6X as compared to traditional TCP/IP in some cases. For InfiniBand, we also proposed several extensions such as packetized flow control, asynchronous zero-copy communication, etc., to the industry standard high-performance sockets specification, Sockets Direct Protocol (SDP), to improve the performance further.

### 13.1.2 Extended Sockets Interface

In the extended sockets interface component, we challenge the sockets semantics itself and question whether the interface is rich enough to support current and next-generation applications or whether it needs to be extended. We analyzed the resource usage (e.g., CPU, memory traffic) associated with the standard sockets interface in Chapter 9 and evaluated a web data-center in Chapter 10; both studies revealed that a sockets interface extended with features like RDMA can provide significant benefits to the application performance of close to 10 times in some cases.

### 13.1.3 Wire-protocol Compatibility for Ethernet Adapters

While achieving the best performance is highly desired, this has to be done in a globally compatible manner, i.e., all networks should be able to transparently take advantage of the proposed performance enhancements while interacting with each other. This, of course, is an open problem. In the wire-protocol compatibility component, we picked a subset of this problem to provide such compatibility within the Ethernet family while trying to maintain most of the performance of the networks. We proposed different designs for achieving such compatibility and showed that with less than 5% degradation in the performance, we can achieve wire protocol compatibility between TCP offload engines and iWARP capable adapters.

## 13.2 Future Research Directions

Apart from the work done in this dissertation, there are several aspects that require further research to complete the understanding and analysis of the proposed research area. In this section, we will discuss some of these aspects.

### 13.2.1 Multi-network Sockets Direct Protocol

Though current high-speed networks provide a high-bandwidth, with the upcoming SMP and multi-core architectures where multiple processes running on a single node must share the network, bandwidth can still become the performance bottleneck for some of today's most demanding applications. Multirail networks, where nodes in a cluster are connected by multiple rails of the same network or different networks, are becoming increasingly common today. Such networks provide the potential for a high bandwidth as well as fault-tolerance if a network fails. However, current SDP implementations do not utilize any such capabilities offered by current clusters. Incorporating such capabilities into SDP could be beneficial for these clusters.

### 13.2.2 Hardware Supported Flow-control

Networks such as InfiniBand provide hardware support for end-to-end flow-control. However, currently most programming models perform flow control in software. With packetized flow control, we utilized the RDMA capability of networks to improve the flow control mechanism. However, we still rely on the application to make regular sockets calls in order to ensure communication progress. Using hardware supported flow-control, such restrictions can be relaxed or potentially completely removed.

### 13.2.3 Connection Caching in SDP

While SDP over InfiniBand has a good data transfer performance, its performance for non data-touching operations such as connection establishment is not the best. For certain applications, such as Ganglia and data-center applications, connection establishment falls in the critical path. Specifically, for every request a connection is established between the client and the server and is torn down at the end of the request. For such applications, caching the connection can improve the performance significantly.

### 13.2.4 NIC-based TCP Termination

Schemes such as SDP only provide a high performance in a cluster environment. In a SAN/WAN environment where some nodes are connected over the WAN and some over the SAN, some kind of approach is required where nodes within the cluster can communicate with SDP while the nodes outside the cluster can communicate with TCP/IP. TCP termination allows us to achieve this by converting data flowing over TCP/IP to SDP and vice versa. In the NIC-based TCP Termination approach, we propose a solution for some of the programmable network interfaces available today to perform efficient TCP termination. In this approach, each network adapter itself emulates the capabilities of a hardware offloaded TCP/IP stack (e.g., TCP offload engine, or TOE, for short). For example, the network adapter adds pre-calculated pseudo TCP/IP like headers and directly sends them out to the remote cluster. The network adapter on the receiver node matches this TCP/IP header to a pre-established native transport layer connection and places the data in the appropriate user buffer. The advantage of this approach is that it directly utilizes the core network interfaces used by the various clusters. The disadvantage, however, is that this approach is only valid for programmable network interfaces such as Myrinet and Quadrics and is not directly implementable on other network interfaces such as InfiniBand and 10GigE.

### 13.2.5 Extending SDP Designs to Other Programming Models

Several of the schemes that we proposed in this dissertation are applicable to several other programming models and upper layers such as MPI, file-systems, etc. For example, the packetized flow-control could provide several improvements for MPI applications. Similarly, the asynchronous zero-copy communication can provide improvements to other upper-layers relying on the POSIX I/O semantics such as file-systems.

### 13.2.6 Build Upper-layers based on Extended Sockets

Several middleware and programming models that are traditionally implemented on sockets are being ported to use the native interface exposed by networks in order to achieve a high performance. However, most of these upper-layers typically use only basic zero-copy communication and RDMA features of these networks; rewriting them completely to utilize such features is highly impractical and time consuming. Porting such upper-layers to use extended sockets makes this task much simpler and a good starting point to incorporate network-specific enhancements.

### 13.2.7 High Performance MPI for iWARP/Ethernet

The Message Passing Interface (MPI) is the *de facto* standard programming model for many scientific applications. However, currently there is no MPI implementation over the upcoming iWARP standard for Ethernet networks. Many of the designs proposed in this dissertation are applicable for designing MPI over iWARP.

# BIBLIOGRAPHY

[1] Ganglia Cluster Management System. http://ganglia.sourceforge.net/.

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

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

[4] Quadrics Supercomputers World Ltd. http://www.quadrics.com/.

[5] Sockets Direct Protocol. `http://www.infinibandta.com`.

[6] The Apache Web Server. http://www.apache.org/.

[7] Top500 supercomputer list. http://www.top500.org, November 2004.

[8] 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.

[9] The Internet Traffic Archive. http://ita.ee.lbl.gov/html/traces.html.

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

[11] S. Bailey and T. Talpey. Remote Direct Data Placement (RDDP), April 2005.

[12] P. Balaji, S. Bhagvat, H. W. Jin, and D. K. Panda. Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol (SDP) over InfiniBand. In *the Workshop on Communication Architecture for Clusters (CAC); held in conjunction with the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, Apr 2006.

[13] P. Balaji, H. W. Jin, K. Vaidyanathan, and D. K. Panda. Supporting iWARP Compatibility and Features for Regular Network Adapters. In *Workshop on RDMA: Applications, Implementations, and Technologies (RAIT); held in conjunction with the IEEE International Conference on Cluster Computing*, 2005.

[14] 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 and Systems Software (ISPASS)*, 2004.

[15] 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.

[16] P. Balaji, P. Shivam, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[17] P. Balaji, 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.

[18] 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)*, 2003.

[19] 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 *IPDPS '00*.

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

[21] 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.

[22] 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.

[23] M. D. Beynon, T. Kurc, U. Catalyurek, and J. Saltz. A component-based implementation of iso-surface rendering for visualizing large datasets. *Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS*, 2001.

[24] 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. *IEEE Micro '95.*

[25] 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.

[26] 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.

[27] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *the Proceedings of Supercomputing*, pages 7–13, November 1998.

[28] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium.*

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

[30] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *4th Annual Linux Showcase and Conference.* USENIX Association, 2000.

[31] 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.

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

[33] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of ACM Sigmetrics 2003, San Diego, CA*, June 2003.

[34] Jeff Chase, Andrew Gallatin, and Ken Yocum. End-System Optimizations for High-Speed TCP.

[35] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Cluster Computing*, 02.

[36] H. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings of 1996 Winter USENIX*, 1996.

[37] D. Clark, V. Jacobson, J. Romkey, and H.Salwen. An Analysis of TCP processing overhead. *IEEE Communications*, June 1989.

[38] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. On the Performance of TCP Splicing for URL-aware Redirection. In *the Proceedings of the USENIX Symposium on Internet Technologies and Systems*, October 1999.

[39] 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.

[40] Chelsio Communications. http://www.chelsio.com/.

[41] Chelsio Communications. http://www.gridtoday.com/04/1206/104373.html, December 2004.

[42] Myricom Corporations. The GM Message Passing Systems.

[43] P. Culley, U. Elzur, R. Recio, and S. Bailey. Marker PDU Aligned Framing for TCP Specification, November 2002.

[44] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *HotI*, 2005.

[45] 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 *SC '03*.

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

[47] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Transparently Achieving Superior Socket Performance using Zero Copy Socket Direct Protocol over 20 Gb/s InfiniBand Links. In *RAIT*, 2005.

[48] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Zero Copy Sockets Direct Protocol over InfiniBand - Preliminary Implementation and Performance Analysis. In *HotI*, 2005.

[49] Roch Guerin and Henning Schulzrinne. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Network Quality of Service. Morgan Kaufmann, 1999.

[50] R. Horst. ServerNet Deadlock Avoidance and Fractahedral Topologies. In *Proceedings of the International Parallel Processing Symposium*, pages 274–280, 1996.

[51] http://www.top500.org. Top 500 supercomputer sites.

[52] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro '04*.

[53] Ammasso Incorporation. http://www.ammasso.com/.

[54] GigaNet Incorporations. cLAN for Linux: Software Users' Guide.

[55] 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. *JPDC '05*.

[56] H.-W. Jin, S. Narravula, G. Brown, K. Vaidyanathan, P. Balaji, and D.K. Panda. Performance Evaluation of RDMA over IP: A Case Study with the Ammasso Gigabit Ethernet NIC. In *Workshop on High Performance Interconnects for Distributed Computing (HPI-DC); In conjunction with HPDC-14*, July 2005.

[57] J. S. Kim, K. Kim, and S. I. Jung. Building a High-Performance Communication Layer over Virtual Interface Architecture on Linux Clusters. In *ICS '01*.

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

[59] Cherkasova L. and Ponnekanti S. R. Optimizing a content-aware load balancing strategy for shared Web hosting service. In *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 492 – 499, 29 Aug - 1 Sep 2000.

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

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

[62] Inc Mindcraft. http://www.mindcraft.com/webstone.

[63] Myricom Inc. Sockets-GM Overview and Performance.

[64] 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.

[65] Netgear Incorporations. http://www.netgear.com.

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

[67] P. H. Carns and W. B. Ligon III and R. B. Ross and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

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

[69] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *HotI '01.*

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

[71] A. Romanow and S. Bailey. An Overview of RDMA over IP. In *Proceedings of International Workshop on Protocols for Long-Distance Networks (PFLDnet2003)*, 2003.

[72] Rob B. Ross. Parallel i/o benchmarking consortium. http://www-unix.mcs.anl.gov/rross/pio-benchmark/html/.

[73] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, R. S. Madukkarumukumana, and G. 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.

[74] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *the Proceedings of CANPC workshop (held in conjunction with HPCA Conference), pages 91-107*, 1999.

[75] 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.

[76] Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *the Proceedings of Supercomputing*, 2001.

[77] 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.

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

[79] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Workshop on I/O in Parallel and Distributed Systems*, 1999.

[80] USNA. TTCP: A test of TCP and UDP performance, December 1984.

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

[82] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *ICPP*, Oct. 2003.

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

[84] C. Yoo, H. W. Jin, and S. C. Kwon. Asynchronous UDP. *IEICE Transactions on Communications*, E84-B(12):3243–3251, December 2001.

[85] George Kingsley Zipf. Human Behavior and the Principle of Least Effort. Addison-Wesley Press, 1949.