SOCKETS DIRECT PROTOCOL OVER INFINIBAND FOR HIGH PERFORMANCE & FAULT-TOLERANT SCALE OUT NAS SYSTEM

A Dissertation Submitted in Partial Fulfillment of the Requirement
For the Award of the Degree of

MASTER OF ENGINEERING

(Computer Technology & Application**)**

Submitted By:
THUMMAR BHAVINKUMAR VRUJLAL
College Roll No: 15/CTA/09
University Roll No: 8554

Under the esteemed guidance of:

Mr. MANOJ KUMAR

ASST. PROFESSOR

DEPARTMENT OF COMPUTER ENGINEERING

DELHI COLLEGE OF ENGINEERING

UNIVERSITY OF DELHI

(2009-2011)

CERTIFICATE

DELHI COLLEGE OF ENGINEERING
(Govt. of NationalCapitalTerritory of Delhi)
BAWANA ROAD, DELHI – 110042
Date: ___________

This is certified that the major project report entitled **Sockets Direct Protocol over Infiniband for High Performance and Fault-Tolerant Scale Out NAS System** is a work of **THUMMAR BHAVINKUMAR VRUJLAL (University Roll No-8554)** a student of Delhi College of Engineering. This work is completed under my direct supervision and guidance and forms a part of master of engineering (Computer Technology and Application) course and curriculum. He has completed his work with utmost sincerity and diligence.

The work embodied in this major project has not been submitted for the award of any other degree to the best of my knowledge.

Mr. MANOJ KUMAR
ASST. PROFESSOR & PROJECT GUIDE
Department of Computer Engineering
DelhiCollege of Engineering,
University of Delhi, India

ACKNOWLEDGEMENT

This thesis has been a brain stimulating experience for me. The knowledge gained by me during the course of this thesis will stand me in good stead in the future. I would like to thank my guide **Mr. Manoj Kumar, Asst. Professor, COE Dept.** for his constant support, encouragement and advice. Without his help and guidance, this dissertation would have been impossible. He remained a pillar of help throughout the project. Also I would like to express my gratitude to my teachers **Dr. Daya Gupta, Mrs. Rajni Jindal, Mr. Vinod Kumar and other staff of COE department** for providing me any time access to the resources and guidance.

My thanks are due to **Dr. Ranjit Noronha, IBM (India Storage Lab)** forgiving opportunity to work on this challenging project. I appreciate the time and effort he invested in steering my research work. Also I would like to express my gratitude to **Mr.AshishChaurasia** for providing excellent research environment in the organization. Special thanks to Rohit, Praveen, Deepak, Tushar, Malik, Anuj and Faiz of the IBM (ISL) for making my work at IBM a pleasant one.

I feel deeply indebted to my parents, sisters and my entire family for their love, affection and confidence in me.

Last but not least, I thank to the crowd who are active in various Infinibandlists and forum.

THUMMAR BHAVINKUMAR VRUJLAL
Master of Engineering(Computer Technology & Application)
College Roll No. - 15/CTA/09
University Roll No. - 8554
Department of Computer Engineering (DCE)

ABSTRACT

Conventional network protocols such as TCP/IP have traditionally been implemented in kernel space and have not been able to scale with increasing network speeds. Accordingly, they form the primary communication bottleneck in current high-speed networks like 10G Ethernet and Infiniband. In order to allow existing TCP/IP applications thathad been written on top of the sockets interface to take advantage of high-speednetworks, researchers have come up with a number of solutions including high performance sockets. The primary idea of high-performance sockets is to build apseudo sockets-like implementation which utilizes the advanced features of high-speednetworks while maintaining the TCP/IP sockets interface. This allows existing TCP/IPsockets based applications to transparently achieve a high performance. The Sockets Direct Protocol (SDP) is an industry standard for such high performance sockets over the InniBand (IB) and Ethernet networks.

In this dissertation, we focus on designing and enhancing SDP over IB for storage systems like Network Attached Storage (NAS) and Storage Area Network (SAN). Specifically we divide the research performed into two parts: (i) Enhancing performance of network communication by using SDP over IB. (ii) Ensuring high availability of the system by designing failover mechanisms for SDP over IB. In this dissertation, we propose application aware failover as well as application transparent failover for SDP over IB.

Table of Contents

Chapters:

Table of Figures

Chapter 1

INTRODUCTION

Cluster systems are becoming increasingly popular in various application domains mainly due to their high performance-to-cost ratio. Cluster systems are now presentat all levels of performance, due to the increasing capability of commodity processors, memory and the network communication stack.

Since the nodes in a clustersystem rely on the network communication stack in order to coordinate and communicate with each other, it forms a critical component in the efficiency and scalabilityof the system. Therefore, it is of particular interest. The network communicationstack itself comprises of two components:

(i) The network hardware
(ii) Thecommunication protocol and the associated software stack.

With respect to the first component, during the last few years, the research andindustry communities have been proposing and implementing high-speed networking hardware such as InfiniBand (IB) [4], 10-Gigabit Ethernet (10GigE) [20, 27, 28, 21]and Myrinet [15], in order to provide efficient support for such cluster systemsamongst others. For the second component (communication protocol stack), however, there has not been as much success.

Earlier generation communication protocols such as TCP/IP [38, 42] relied uponthe kernel for processing the messages. This caused multiple copies and kernelcontext switches in the critical message passing path. Thus, the communicationperformance was low. Researchers have been looking at alternatives to increase communication performance delivered by clusters in form of low-latency and high-bandwidth ULPs such as FM [33] and GM [18] for Myrinet [15] &EMP [37, 36] for Gigabit Ethernet [23].

These developments are reducing the gap between the performance capabilitiesof the physical network and that obtained by the end users. While this approachis good for developing new applications, it might not be so beneficial for existingapplications. A number of applications have been developed on kernel-based protocols such as TCP/IP or UDP/IP using the sockets interface. To support suchapplications on high performance user-level protocols without any changes to theapplication itself, researchers have come up with different techniques including high-performance sockets implementations [12, 30, 35, 14]. High-performance sockets arepseudo sockets-like implementations to meet two primary goals:

(i) To directly andtransparently allow existing sockets applications to be deployed on to clusters connected with modern networks such as IB and iWARP and

(ii) Allow such deploymentwhile retaining most of the raw performance provided by the networks.

In an attempt to standardize these efforts towards high-performance socketsimplementations, the Remote Direct Memory Access (RDMA) Consortium brought out a new standard known as theSockets Direct Protocol (SDP) [2]. Figure 1.1 shows the traditional IP over InfiniBand (IPoIB)stackand the SDP stack over IB.

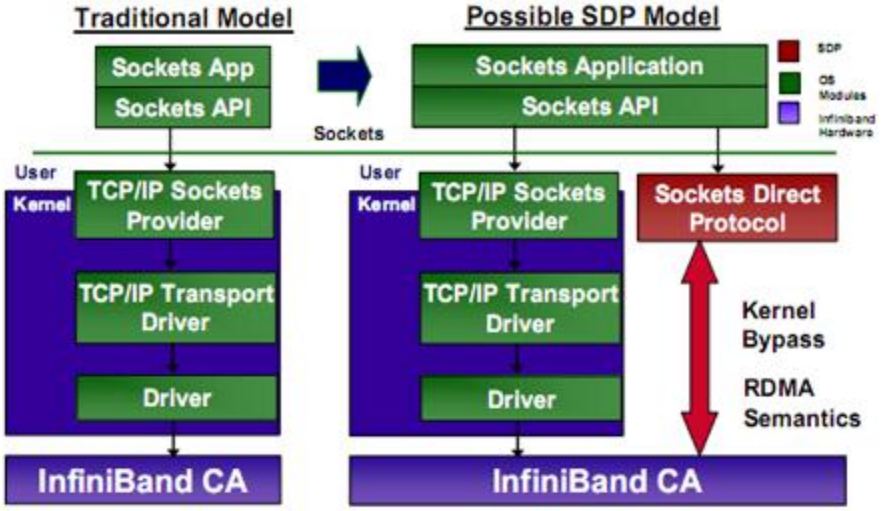

Figure 1-1: Traditional IPoIB (TCP/IP) vs. SDP over IB

Modern storage systems such as Scale Out Network Attached Storage (SONAS) are designed around hybrid architectures like Network Attached Storage (NAS) and Storage Area Network (SAN). In such systems, storage devices are connected through high performance network components and form a clustered system area network. Clients are connected to the interface nodes which host file systems to operate on multiple storage nodes connected through a high performance network. Interface nodes read and write data to/from storage nodes through socket applications. As data is move around the system area network, the network overhead is the main bottleneck in the performance of any storage systems.

Performance is not only the issue while designing any scalable storage system. System should be highly available too. Failure may occur in many ways such as network link failure, system hardware failure, disk failure etc. These failures can be handled by having redundancy in almost each and every component of the system such as redundant disks, redundant network links, redundant nodes etc. The mechanism involved in switching over to the redundant component at the time of failure is called Failover. Failover should be carried out transparently.

As a result of tradeoff between performance and availability, most of the modern storage systems use conventional TCP/IP based network communication for data transfer between interface nodes and storage nodes. IPoIB is the Upper Layer Protocol (ULP) which is being used for communication over IB to ensure the network link failover while achieving comparatively higher performance than Ethernet.

1.1 SDP: State-of-the-Art and Limitations

As indicated earlier, the SDP standard attempts to transparently provide high performance for existing sockets-based applications over high-speed networking stacks such as IB and iWARP. While, there are several implementations of the SDP standard [10, 25, 24, 7], these lack in several aspects. Some of these aspects correspond to designs proposed in the SDP standard which might not be optimal in all scenarios, while the others are specific to existing SDP implementations where the current designs have scope for improvement in multiple dimensions.

In this dissertation, we work on replacing IPoIB with SDP over Infiniband for the storage systems like SONAS. We propose prototypes to overcome following two limitations:

(i) Failover
(ii) Performance Tuning

1.1.1 Failover

Network Link failure of any node may lead to failure of the node itself. To avoid such failure, link redundancy is used. At the time of link failure, switching over to the redundant link should be performed without affecting the normal operation of the system. IPoIB uses the Linux Bonding driver to tackle this problem but there is no such mechanism exists for the SDP. This failover can be performed in two ways as follows:

(i) Application aware failover
(ii) Application transparent failover

1.1.2 Performance Tuning

Proper performance tuning is required to ensure highest performance enhancements of SDP. Replacing IPoIB with SDP enables the use of Zero Copy (ZCopy) and Buffered Copy (BCopy). Setting of proper thresh-hold value is needed in order to specify that when to use BCopy and when to use ZCopy in order to gain highest possible performance. Further to this, variable configuration components like message size, Naggle’s switch etc. are also need to be set properly in order to achieve highest possible performance.

1.2 SONAS

Scale Out Network Attached Storage (SONAS) is a highly scalable Network Attached Storage system for large scale as deployments requiring very large storage capacities - from 100’s Terabytes to multiple Peta-bytes, independent capacity and performance scaling, support for very large file systems and parallel access to data. Figure 1-2 shows the architecture of the SONAS from the project’s perspective.

On application side, there are various file systems such as CIFS, NFS, RSYNC etc. through which clients access the storage of the system. Clustered Trivial Database Daemon (CTDB) is a cluster implementation of the TDB database used by Samba and other projects to store temporary data.CTDB provides HA features such as node monitoring, node failover, and IP takeover, like controlling “public” IP addresses – distribute public addresses across nodes, movement of addresses to “healthy” nodes in case a node having an address transitions from healthy to unhealthy.

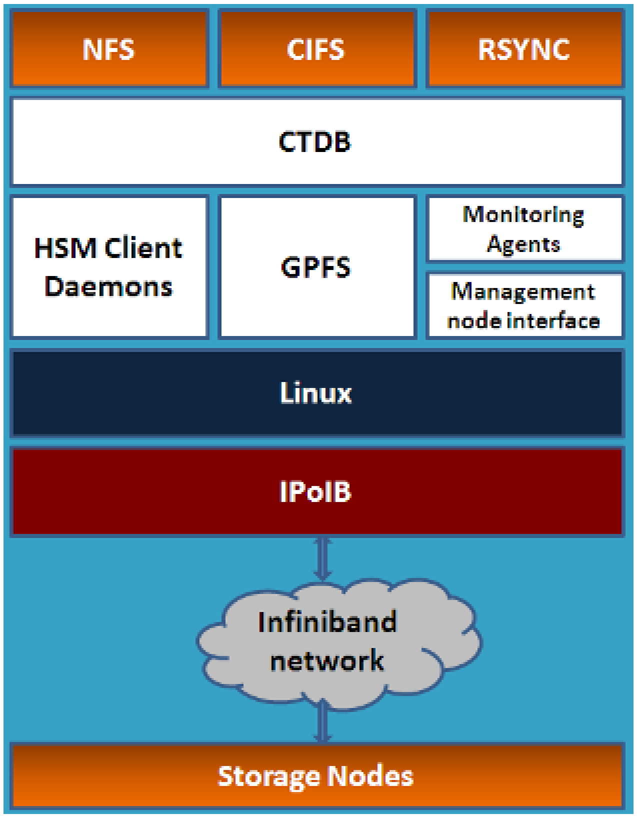

Figure 1-2: SONAS Architecture

The General Parallel File System (GPFS) is a high-performance shared-disk clustered file system developed by IBM. It is used by many of the supercomputers that populate the Top 500 List of the most powerful supercomputers on the planet. In SONAS, GPFS is an embedded component managed by the SONAS Management stack for Setup, Configuration and Monitoring. There are also some other management utilities such as management node interface.

In SONAS, multiple interface nodes hosting GPFS are connected to the storage nodes through the underlying Infiniband network. At present, the communication between interface and storage nodes over Infiniband uses the IPoIB Upper Layer Protocol. Primary objective of this research is to replace IPoIB with the more optimized SDP upper layer protocol.

1.3 ORGANIZATION OF DISSERTATION

This dissertation begins with quick introduction of the Sockets Direct Protocol (SDP) and limitations with the SDP. Chapter 1 is dedicated to the quick introduction about this dissertation having overview of SDP, failover and performance.

Chapter 2 is all about the background of this project. This chapter starts with the introduction of Infiniband which includes Communication mechanisms, various configurations and Remote Direct Memory Access (RDMA) Model. Further to Infiniband, this chapter also introduces the SDP and the OFED software stack.

Chapter 3 covers the review of some of the research literatures from which ideas has been taken and used in the proposed architectures.

Chapter 4 presents the experimental setup used in design and implementation of the proposed solutions.

Chapter 5 discusses proposed solutions in detail. It start with the detailed description of the Failover and problems in implementing it. Later in the chapter both the proposed failover solutions are discussed.

Chapter 6 is dedicated to the Performance Tuning to gain the optimal performance enhancement from SDP over Infiniband.

Chapter 7 presents all the results taken to prove the performance enhancement in terms of throughput and latency.

Chapter 8concludes the dissertation and also presents ideas about future planned works.

At the end, Bibliography shows all references which is used throughout this dissertation.

Chapter 2

BACKGROUND AND MOTIVATION

In this chapter, we start with an overview of Infiniband and subset of its features and various configurations in Section 2.1. The SDP is described in Section 2.2. Next, in Section 2.3, we provide a brief overview on Open Fabrics Enterprise Distribution (OFED) stack.

2.1 OVERVIEW OF INFINIBAND

The InfiniBand Architecture (IB) is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. The compute nodes are connected to the IB fabric by means of Host Channel Adapters (HCAs). IB 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 [1].

IB 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 the verbs interface includes a queue-based interface, known as a Queue Pair (QP), to issue requests to the HCA. Figure 2-1 illustrates the InfiniBand Architecture model.

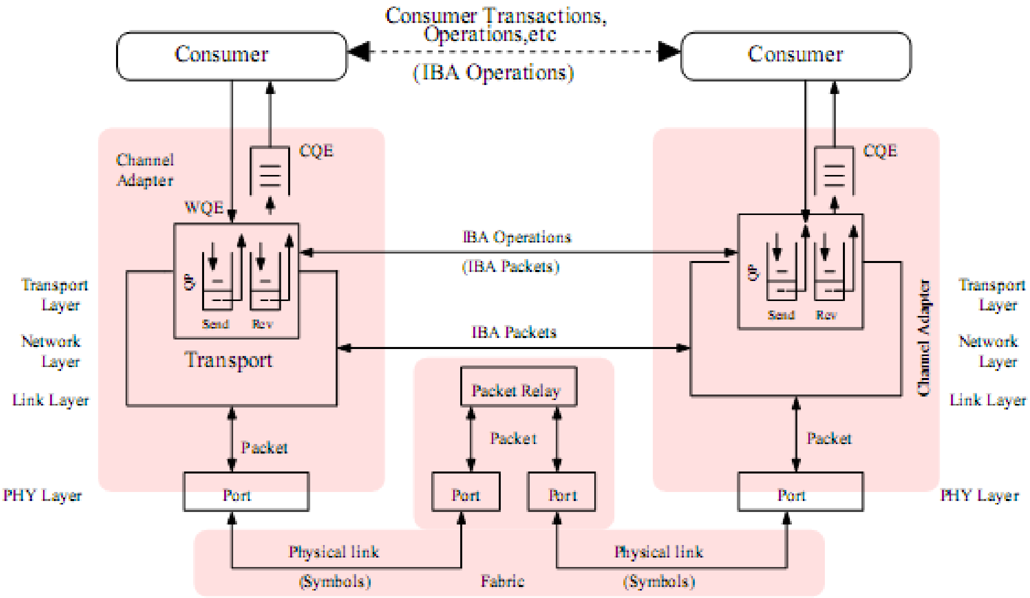

Figure 2-2: Infiniband Architecture (Courtesy Infiniband Specifications)

2.1.1 IB Communication

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 Entries (WQEs) to these queues. Each WQE is associated with one or more pre-registered buffers from which data is either transferred (for a send WQE) or received (receive WQE). 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.

2.1.2 IB Configurations

Various IB configurations have been evolved for different applications and requirements such as IPoIB, SDP over IB, RDMA enabled IB application, Message Passing Interface (MPI), iSCSI enabled RDMA etc. In this dissertation, we focus mainly on three of them.

1. **IPoIB:**

IPoIB is specified by the IETF (RFC 4931/4932). IP over Infiniband provides TCP/UDP interface for Infiniband. It uses IB as a transport for IP. There is no RDMA available in IPoIB. IPoIB is also used for address resolution for other ULPs such as SDP, iSER and RDS. IPoIB has the highest level of application compatibility which means that there is no change at application side required to use IPoIB. Figure 2-2 shows the IPoIB protocol stack.

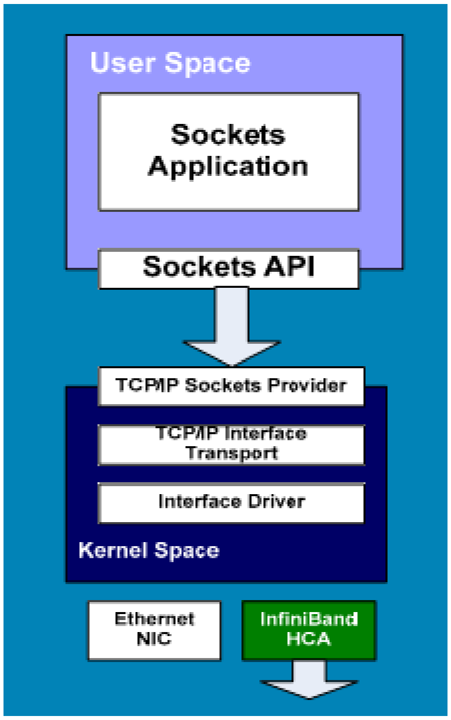

Figure 2-2: IPoIB Stack

2. SDP Over IB:

SDP also provides a compatible socket interface although some minor configurations are needed in the host machine in order to use SDP. SDP provides RDMA mechanism through ZCopy.

3. RDMA Enabled Application:

As SDP module is implemented in the kernel space, SDP over IB is not the optimal solution for performing RDMA. To overcome this problem, Infiniband also provides necessary APIs (verbs) to implement RDMA enabled applications which give optimal performance due to no kernel intervention at all.

Figure 2-3 shows SDP over IB and RDMA enabled application stacks.

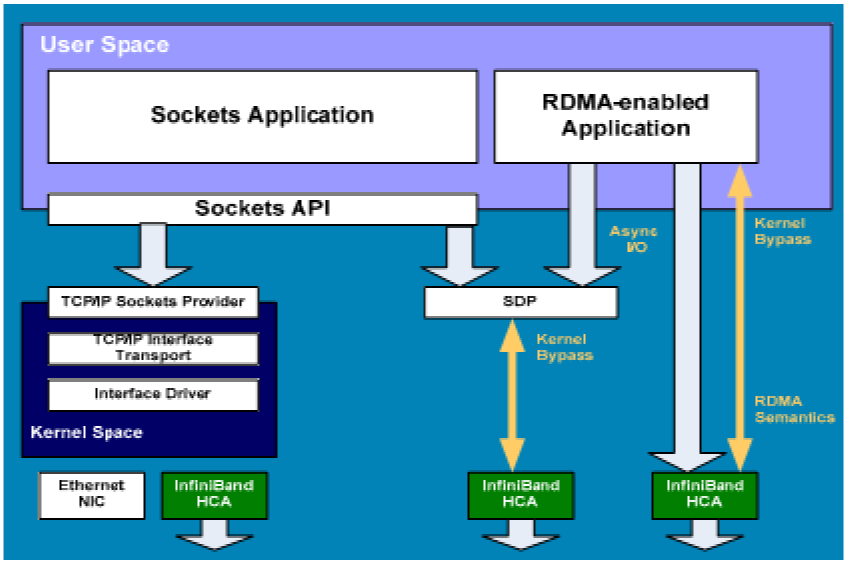

Figure 2-3: SDP & RDMA Stacks

2.1.3 RDMA Communication Model

IB 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 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 retransmitted for a user specified amount of time. In the memory semantics, Remote Direct Memory Access (RDMA) operations are used. These operations are transparent at the remote end since they do not require the remote end to involve in the communication. Therefore, an RDMA operation has to specify both the memory address for the local buffer as well as that for the remote buffer. 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 directly reads data from the remote node's user buffer. Most entries in the WQE 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.

2.2 SOCKETS DIRECT PROTOCOL (SDP)

The SDP standard focuses specifically on the wire protocol, finite state machine and packet semantics. Operating system issues, etc., can be implementation specific. It is to be noted that SDP supports only SOCK STREAM or streaming sockets semantics and not SOCK DGRAM (datagram) or other socket semantics.

SDP enables existing socket based applications to transparently utilize the IB capabilities and achieve superior performance. As SDP enables direct data transfer between two applications running on two different nodes without any intervention of kernel from either side, performance of the data transfer gets improved significantly.

SDP's Upper Layer Protocol (ULP) interface is a byte-stream protocol that is layered on top of IB's message-oriented transfer model. The mapping of the byte stream protocol to the underlying message-oriented semantics was designed to enable ULP data to be transferred by one of two methods:

(i) Through intermediate private buffers (using a buffer copy)

(ii) Directly between ULP buffers (zero copy).

A mix of send/receive and RDMA mechanisms are used to transfer ULP data. The SDP specification also suggests two additional control messages known as Buffer Availability Notification messages, viz., source-avail and sink-avail messages for performing zero-copy data transfer.

Figure 2-4 shows the data transfer modes over SDP.

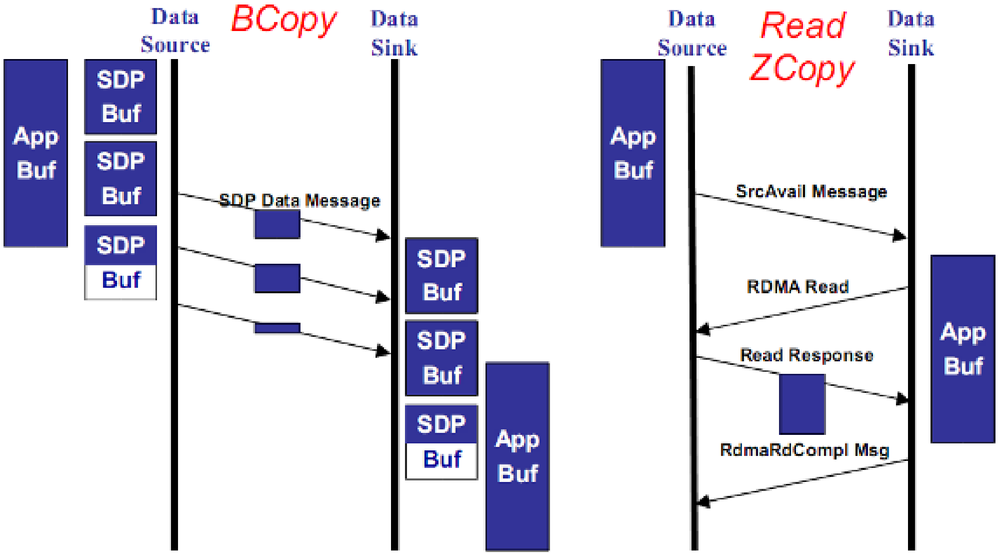

Figure 3-4: SDP Data Transfer Modes (BCopy&ZCopy)

Sink-avail Message: If the data sink has already posted a receive buffer andthe data source has not sent the data message yet, the data sink does the followingsteps:

(i) Registers the receive user buffer (for large message reads)

(ii) Sendsa sink-avail message containing the receive buffer handle to the source.

The datasource on a data transmit call, uses this receive buffer handle to directly RDMAwrite the data into the receive buffer.

Source-avail Message: If data source has already posted a send bufferand the available SDP window is not large enough to contain buffer, it does thefollowing 2 steps:

(i) Registers the transmit user buffer (for large message sends)

(ii) Sends a source-avail message containing the transmit buffer handle to the 16data sink.

The data sink on a data receive call, uses this transmit buffer handle to directly RDMA read the data into the receive buffer.Figure 2-5 shows the performance improvements by replacing IPoIB with SDP for sockets applications.

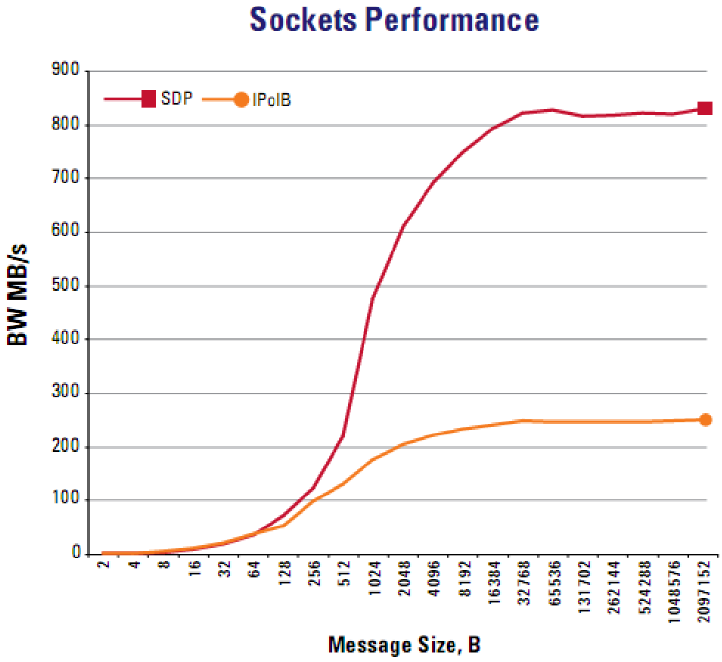

Figure 2-5: Performance Comparision for SDP vs. IPoIB

2.3 OFED STACK

OFED is high performance server and storage connectivity software for field-proven RDMA and Transport Offload hardware solutions. The OFED from OpenFabrics alliance has been hardened through collaborative development and testing by all major InfniBand vendors. OFED is supported by Mellanox and major InfniBand vendors to enable OEMs to meet the needs of HPC applications.

OFED includes kernel-level drivers, channel-oriented RDMA and send/receive operations, kernel bypasses of the operating system, both kernel and user-level application programming interface (API) and services for parallel message passing (MPI), sockets data exchange (e.g., RDS, SDP), NAS and SAN storage (e.g. iSER, NFS-RDMA, SRP) and file system/database systems.

The network and fabric technologies that provide RDMA performance with OFED include: legacy 10 Gigabit Ethernet, iWARP for Ethernet, RDMA over Converged Ethernet (RoCE), and 10/20/40 Gigabit InfiniBand.

OFED is available for many Linux and Windows distributions, including: Red Hat Enterprise Linux (RHEL), Novell SUSE Linux Enterprise Distribution (SLES), Oracle Enterprise Linux (OEL) and Microsoft Windows Server operating systems. Some of these distributions ship OFED in-box. This makes OFED easily accessible and usable by OEMs and end users facilitating quick adoption in multiple market verticals in the high performance computing, enterprise data centre and storage sectors. The entire set of OpenFabrics Software – from which modules and patches are selected to form OFED releases resides on the OpenFabrics servers and is available for download.Figure 2-6 shows the complete OFED stack.

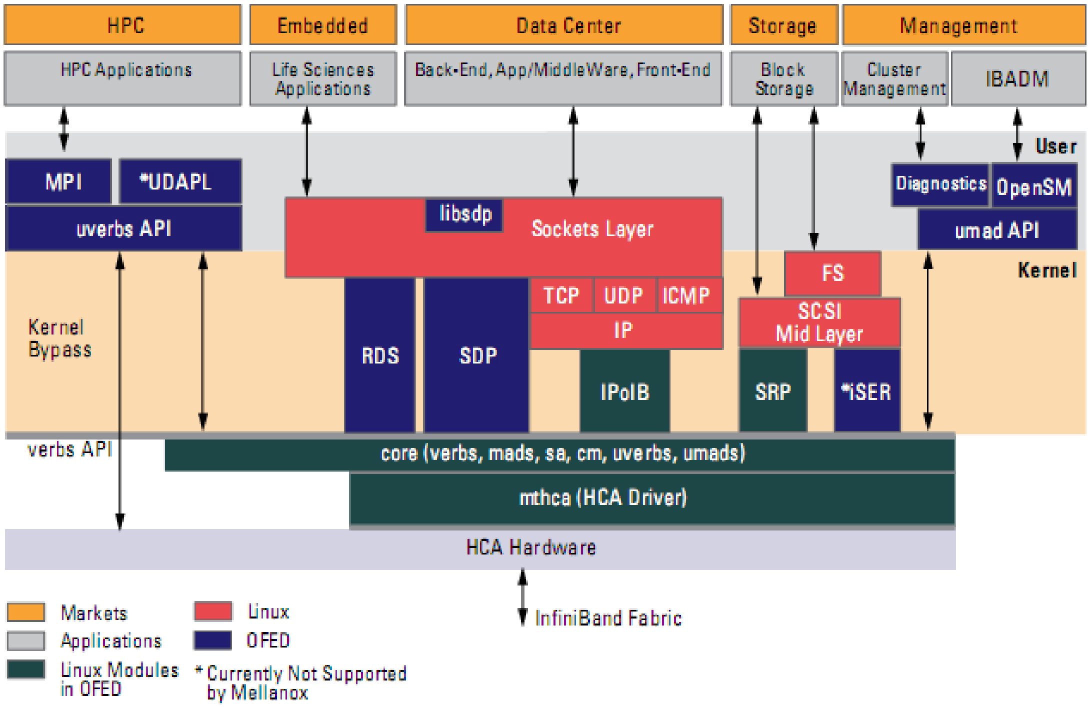

Figure 2-6: OFED Stack (Courtesy Mellanox)

Chapter 3

REVIEW OF LITERATURES

In this chapter, we presented related research literatures which have been published or presented earlier on similar issues.

3.1 SDP

BZcopy and Zcopy are the results of earlier research work from various researchers. Earlier researchers have already proved the performance enhancement over SDP compare to IPoIB [10]. Researchers have proved that by using SDP instead of IPoIB improves the Bandwidth and Latency while ZCopy [25] actually lowers the CPU utilization of the host machine. Figure 3-1 shows the SDP stack components for ZCopy.

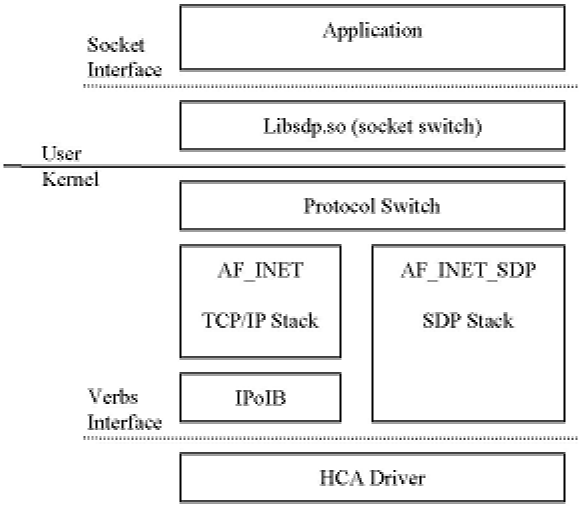

Figure 3-1: SDP Stack Components

3.2 FAILOVER

The primary focus of this dissertation is on solving the problem of failover. We have discussed two approaches; SDP Bonding and Socket Duplication. SDP bonding is proposed around the idea of extending the Linux bonding mechanism for IPoIB to work with SDP.

There is another approach proposed by researchers called Automatic Path Migration (APM) [45].

Researchers have proposed Automatic Path Migration (APM), which allows user transparent detection and recovery from network fault(s), without application restart. In this paper, they designed a set of modules; which work together for providing network fault tolerance for user level applications leveraging the APM feature. Performance evaluation at the MPI Layer shows that APMincurs negligible overhead in the absence of faults in the system. In the presence of network faults, APM incurs negligible overhead for reasonably long running applications.

In this paper, they addressed challenges regarding the failover. They designed a set of modules; alternate path specification module, path loading request module and path migration module, which work together for providing network fault tolerance for user level applications. They evaluated these modules with simple micro-benchmarks at the Verbs Layer, the user access layer for InfiniBand, and study the impact of different state transitions associated with APM. They have also integrated these modules at the MPI (Message Passing Interface) layer to provide network fault tolerance for MPI applications. Performance evaluation at the MPI Layer shows that APM incurs negligible overhead in the absence of faults in the system. In the presence of network faults, APM incurs negligible overhead for reasonably long running applications. For Class B FT and LU NAS Parallel Benchmarks [46] with 8 processes, the degradation is around 5-7% in the presence of network faults.

This mechanism was proposed for the Message Passing Interface. As MPI uses IB verbs at application layer to communicate to the OFED stack components, this mechanism can’t be used with the socket based applications directly. Due to the various design issues with the use of APM for socket based applications this approach was never taken as the solution of the failover problem.

Another approach we proposed in this dissertation is Application Transparent Failover through Socket Duplication. Socket Duplication (Socket Cloning) is primarily designed and used for the closeted web servers [47].

To solve the caching problems in dispatcher based systems researchers have proposed a novel idea called socket cloning. In this paper, they presented a newnetwork support mechanism, called Socket Cloning (SC), inwhich an opened socket can be migrated efficiently betweencluster nodes. With SC, the processing of HTTP requestscan be moved to the node that has a cached copy of therequested document, thus bypassing any object transfer between peer servers. A prototype has been implemented andtests shown that SC incurs less overhead than all the mentioned approaches. In trace-driven benchmark tests, their system outperforms these approaches by more than 30%with a cluster of twelve web server nodes.

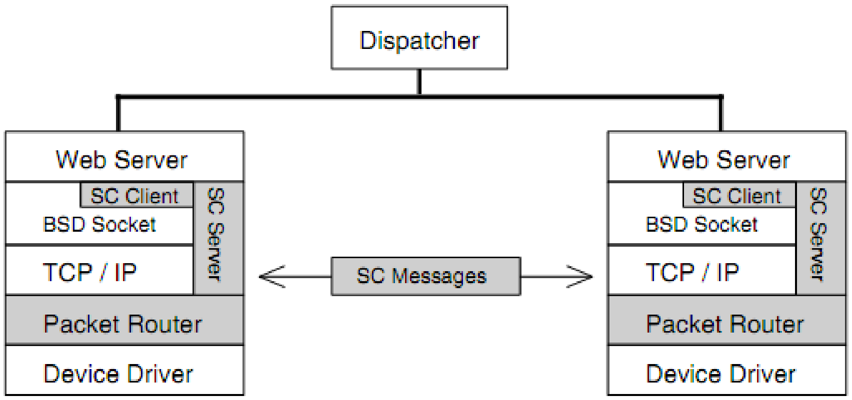

Figure 4-2:System Architecture of Socket Cloning (SC)

To design an application transparent failover for SDP over Infiniband, we have taken this idea of socket cloning and used it for the single system. Instead of cloning sockets across two different machines, in this dissertation we propose a duplication (cloning) of the socket from broken link interface to the redundant link interface. As in this case the cloning is across the same system the IP address of the socket would remain same while just the port address might need to change.

Chapter 4

EXPERIMENTAL SETUP

In this chapter, we provide information about the setup we used to perform required experiments. In section 4.1, we provide information about network system configuration needed to setup experiments.

4.1 NETWORK SYSTEM CONFIGURATION

For the experimental test-bed, we used cluster of four nodes connected through 10 Gbps DDR Infiniband link. Each node in the system has installed two Infiniband network interface cards from Mallenox. We worked on Red-Hat Enterprise (Linux) operating system RHEL 6 and OFED 5.2.1 to perform all required experiments. All systems are the System X from IBM.

Additional host side configuration is needed to enable SDP to use existing socket interface of all targeted socket based applications. There is two methods for conversing from IPoIB to SDP.

(i) Automatic Conversion
(ii) Explicit/Source code Conversion

Automatic Conversion:

- Load the ib_sdp module of OFED
- Set the environmental variable LIBSDP_CONFIG_FILE = /etc/libsdp.conf
- Set the environmental variable LD_PRELOAD=libsdp.so to preload the SDP socket library in to memory so that it can be used instead of original socket library comes with Linux kernel.

By using libsdp.conf, one may control the use of SDP. This method configures the driver to automatically translate TCP to SDP based on Source IP, Destination IP, Port Number or Application Name.

Explicit/Source code Conversion:

One has to define #define AF_INET_SDP 27 a separate protocol type in the socket application so that this constant can be used in the socket system call as follows:

- socket(AF_INET_SDP, SOCK_STREAM,0);

As this method requires change in the application for conversion from IPoIB to SDP, we haven’t used this configuration. Throughout the dissertation, all the displayed results are taken by configuring system through automatic conversion.

Chapter 5

FAILOVER MECHANISMS FOR SDP OVER IB

5.1 FAILOVER

Process of switching over the redundant link in case of active network link failure is called as Failover. Figure 5-1 shows the configuration of the system needed for any failover mechanism.

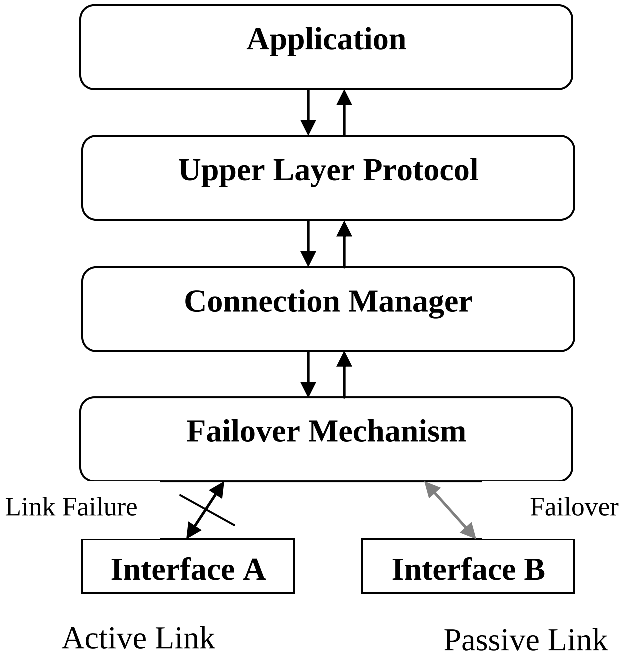

Figure 5-1: System Configuration for Failover

As shown in figure, Interface A and Interface B are two NICs. Initially Interface A is in active state so all the communication is passes through this link. Suppose that at some point of time Interface A goes down due to some technical issue, at this moment communication transfer should be switch over to the passive link B without affecting the normal operation of the system. This process of switching over is called as Failover. Failover mechanism must have two primary functionalities as follows:

(i) Link Detection

(ii) Switch over to redundant link

5.1.1 Link Detection

Two schemes have been proposed by researchers and any one of them can be used with Linux bonding driver to monitor the link status. These two methods are:

(i) ARP Monitor
(ii) MII Monitor

ARP Monitor:

The ARP monitor operates as its name suggests: it sends ARP queries to one or more designated peer systems on the network, and uses the response as an indication that the link is operating. This gives some assurance that traffic is actually flowing to and from one or more peers on the local network.

The ARP monitor relies on the device driver itself to verify that traffic is flowing. In particular, the driver must keep up to date the last receive time, dev->last_rx, and transmit start time, dev->trans_start. If these are not updated by the driver, then the ARP monitor will immediately fail any slaves using that driver, and those slaves will stay down. If networking monitoring (tcpdump, etc) shows the ARP requests and replies on the network, then it may be that your device driver is not updating last_rx and trans_start.

MII Monitor:

The MII monitor monitors only the carrier state of the local network interface. It accomplishes this in one of three ways: by depending upon the device driver to maintain its carrier state, by querying the device's MII registers, or by making an ethtool query to the device.

If the use_carrier module parameter is 1 (the default value), then the MII monitor will rely on the driver for carrier state information (via the netif_carrier subsystem). As explained in the use_carrier parameter information, above, if the MII monitor fails to detect carrier loss on the device (e.g., when the cable is physically disconnected), it may be that the driver does not support netif_carrier.

If use_carrier is 0, then the MII monitor will first query the device's (via ioctl) MII registers and check the link state. If that request fails (not just that it returns carrier down), then the MII monitor will make an ethtool ETHOOL_GLINK request to attempt to obtain the same information. If both methods fail (i.e., the driver either does not support or had some error in processing both the MII register and ethtool requests), then the MII monitor will assume the link is up.

5.1.2 Switch Over To Redundant Link

As mentioned earlier, Failover mechanism can be implemented using two methodologies:

(i) Application Aware Failover
(ii) Application Transparent Failover

Both the methods have its pros and cons in terms of configuration requirements, performance etc.

5.2 APPLICATION AWARE FAILOVER

In this dissertation we propose prototypeof SDP Bonding as an application aware failover mechanism.

5.2.1 SDP Bonding

IPoIB uses the Linux Bonding driver to perform failover at the time of network link failure. As SDP also uses the IPoIB for address resolution, bonding driver can also be used with SDP. As data communication paths for IPoIB and SDP are different, operations needed to perform at the time of failover would be different. Figure 4-2 shows the proposed prototype for the SDP Bonding.

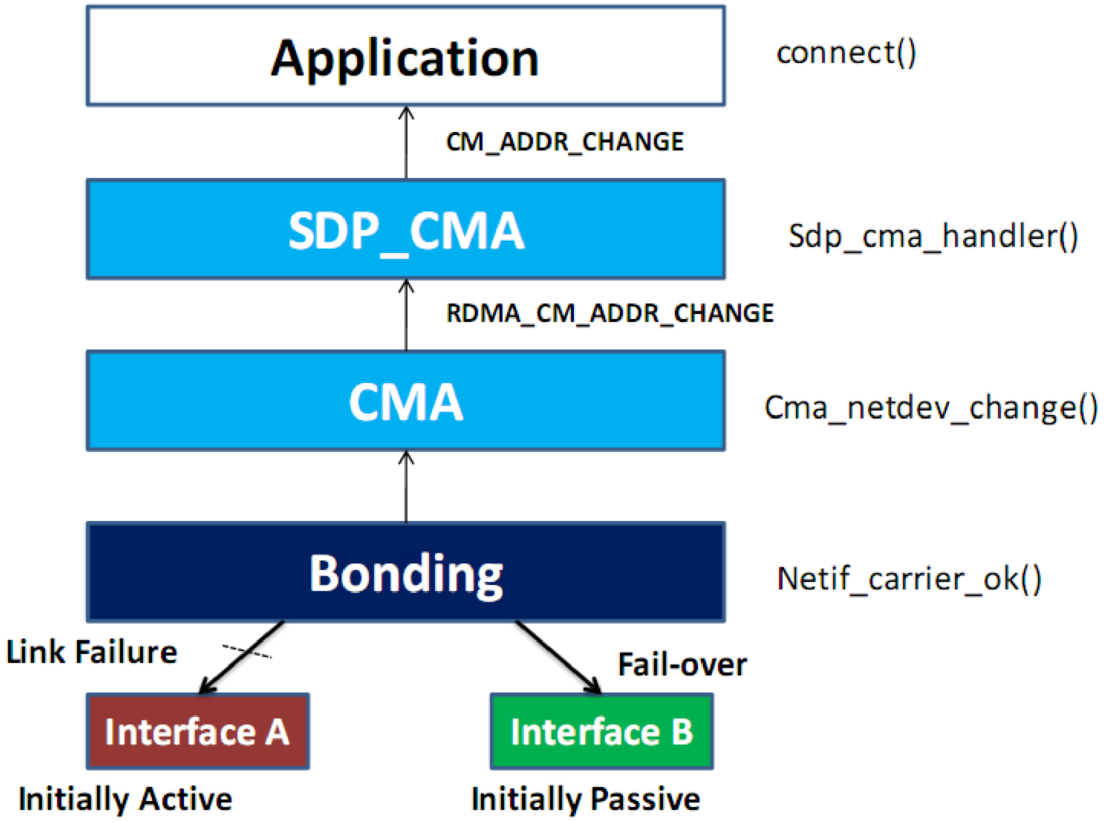

Figure 5-2: SDP Bonding

Upon detecting the broken link by the Bonding layer, Connection Manager abstract layer sends an RDMA_CM_ADDR_CHANGE event to the upper layer protocol’s connection manager.

Flow Chart:

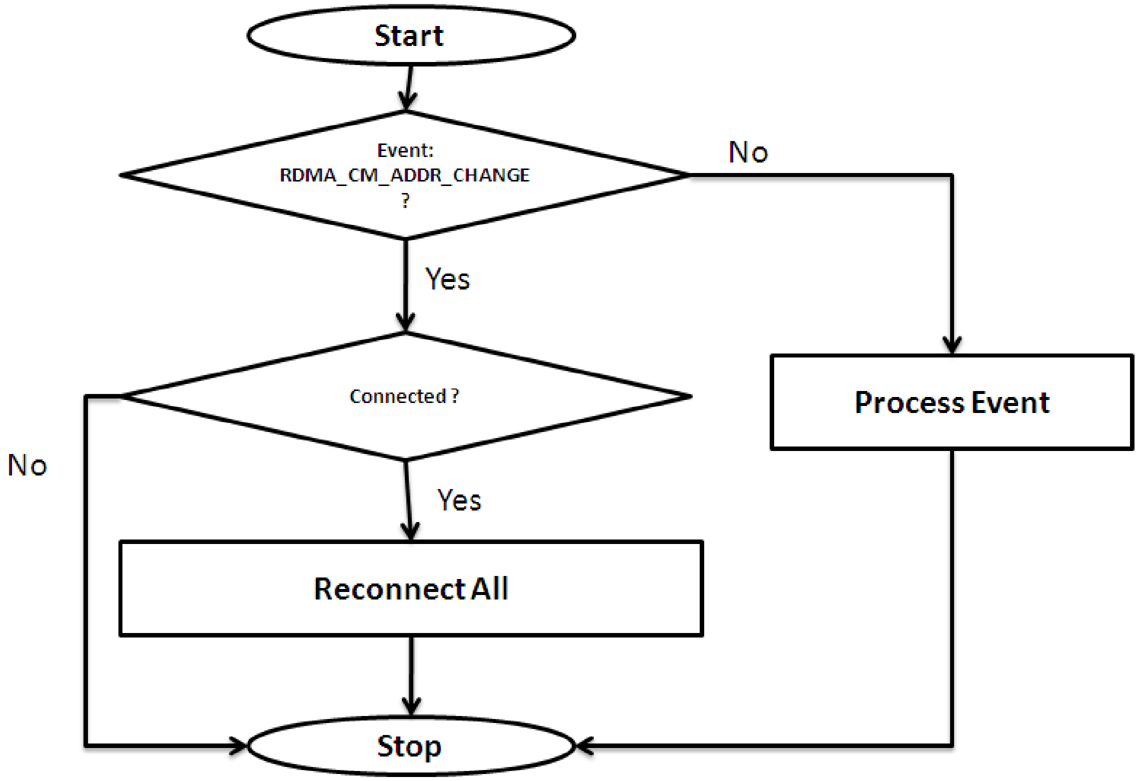

Figure 5-3: **SDP Bonding: Flow Chart**

At this time, SDP module might have been performing read or write operations through BCopy or ZCopy. As these copy operation uses the IB Access layer to access the HCA, these operations can’t be stopped or notified about the link failure. As link has got failed, the state of the SDP module performing read/write operation would be undefined. Due to this reason, it is not possible to reconnect the broken connection from the kernel itself. So instead of reconnecting from kernel level SDP module, we propose to notify upper layer socket library by sending CM_ADDR_CHANGE event. Upon receiving this event, application just need to call connect again with the saved parameters. As we discussed, application code needs to be changed in order to have failover through SDP Bonding.

As we are not performing any extra operations at the SDP kernel layer modules, SDP Bonding mechanism doesn’t imposes any kind of overhead during the normal operations of the system.

We have taken performance results by running Netperf[44] benchmarks on proposed solution. Proposed solution will be available to open source community once the thorough testing of the implementation is carried out. Throughput test results are as follows:

Message Size	Without Bonding	With Bonding
8 KB	10107 Mbps	6678 Mbps
64 KB	10133 Mbps	6703 Mbps
1 MB	9926 Mbps	9923 Mbps
10 MB	10101 Mbps	10053 Mbps
100 MB	10097 Mbps	10068 Mbps

Table 1: Throughput Test: SDP Bonding vs. Without Bonding

5.3 APPLICATION TRANSPARENT FAILOVER

In this section, we propose an application transparent failover mechanism using Socket Duplication technique. This technique is influenced from the socket cloning solution for clustered web servers' implementation [47].

As we described earlier, bonding driver notifies the Connection Manager about the link failure through sending an event RDMA_CM_ADDR_CHANGE but as both the modules (Connection Manager and SDP send/recv) are in different context, we can't perform reconnection in the kernel layer itself.

When a socket exists in one address space and is then accessed in a different address space (on the same peer), the socket needs to be duplicated into the second address space. Note that if two threads are accessing the socket in the same address space, socket duplication is not required.

Performing socket duplication in user-mode imposes certain restrictionsbecause socket state cannot be shared between the address spaces. In fact, in the context of InfiniBand networks available today, the socket can only exist in one address space at a time (since HCAs are not required to support sharing queue pairs between multiple address spaces).

Because of these restrictions, SDP allows only one address space at a time to execute operations that either transfer data or change state for an underlying shared socket. Address spaces dynamically swap control of the underlying socket, as needed, to execute requested operations. The SDP socket duplication procedure serializes operations that different address spaces request on a shared socket. The procedure waits for all In-Process operations to complete before swapping control of an underlying socket to another address space. Logically, the procedure takes control of the underlying socket away from the controlling address space as soon as a non-controlling address space requests an operation on that socket.

After control is taken away, the procedure treats the original controlling address space like a non-controlling address space if the original controlling address space requests operations on that socket. In this way a socket may transition back and forth between controlling address spaces based on ULP behavior.

We enabled socket duplication by bringing the connection to a consistent state, closing the InfiniBand connection, handing the state to the new controlling address space, and then creating a new reliable connection in the new address space. Note that after the connection is suspended and then restarted on a new InfiniBand connection, the connection by definition does not have any outstanding SinkAvail or SrcAvail advertisements. Any incomplete SinkAvail or SrcAvail advertisements were effectively canceled during the transition to a new connection.

In managed failover, the SDP connection may in fact be reestablished using different paths, ports, HCAs or hosts. The original connection in a managed failover scenario is analogous to the controlling address space in socket duplication. The new failed over connection is analogous to the non-controlling address space. Managed failover changes where one end of the connection is situated. Failing over both ends requires two managed failover operations.

The decision to attempt a managed failover must occur before the socket duplication may take place. For this purpose we rely on the link detection technique used by the Linux bonding driver for IPoIB. Bonding driver sends a notification to the connection manager at the time of link failure. This notification in turn starts the socket duplication procedure.

5.3.1 Implementation

In implementation details, the new failed over connection is analogous to the non-Controlling Address Space.

This implementation in the controlling address space waits for all In-Process data transfer operations to complete, and then it sends a SuspComm Message to the Remote Peer to request a suspension of the session. This SDP Message contains the destination TCP port number received from the non-Controlling Address Space. The Remote Peer connects to this TCP port number when resuming communication. The Local Peer doesn’t send additional SDP Messages or perform any RDMA operations from the Controlling Address Space, after sending the SuspComm Message.

Upon receiving the SuspComm Message, the Remote Peer waits for all In-Process data transfer operations to complete, then sends a SuspCommAck Message indicating that the session is suspended. After sending the SuspCommAck Message, this peer doesn’t send any more SDP Messages or perform any RDMA operations until a new connection is set up.

The Remote Peer waits for completion of the Send of the SuspCommAck Message, then close the LLP connection. The Remote Peer then initiate the new connection to the destination TCP port number received through the SuspComm Message, utilizing the same IP address specified in the prior connection setup sequence. Posting of receive Private Buffers and the contents of the header follows the same rules as connection setup.

Once the SuspCommAck Message is received, the Controlling Address Space on the Local Peer sends a signal to the non-Controlling Address Space through a new message introduced by us: AckRecv. This message may contain following data:

- Any buffered receive ULP data.
- The Remote Peer’s TCP port number (to ensure the parameter does not change when the socket is re-connected).
- The sizes of the local receive Private Buffers.
- The current values for IRD and ORD.

The non-Controlling Address Space accepts the connection request from the Remote Peer and initializes its state variables for the new connection. The Hello Message initializes SDP connection state.

The (previously) non-Controlling Address Space then sends a HelloAck Message to the Remote Peer. The receive Private Buffer size parameter in the HelloAck Message MUST be the values received from the Controlling Address Space. The IRD and ORD values MAY be the values received from the Controlling Address Space. It also makes buffered received ULP data from the Controlling Address Space available to the ULP.

When connection setup is complete, the Local Peer resumes normal data transfer. We haven’t implemented this technique completely due to lack of time, so we don’t have any test results for this methods.

Chapter 6

PERFORMANCE TUNING FOR SDP OVER IB

In this section we propose ideal settings for Zcopy threshold value in terms of message size to gain the optimal performance. We carried out various experiments for different message sizes with combination of Zcopy threshold values to make a decision making statement for the optimal configuration.

As initiating Zcopy involves the cost of making the user space buffer to be available to the Host Channel Adapter until the data transfer is over. This preparation takes place by defining Fast Memory Regions (FMR) which can be break in to two different procedures

(i) Mapping
(ii) Locking

In order to transfer the control of any user space buffer directly to the device, first the user space virtual address must be converted into the physical address and then make sure that this memory region remains in the physical memory until the data transfer is over.

So, this process of preparing user space buffer takes some time. This time is the main decision factor in deciding the Zcopy threshold value.

All the experiments are done on system having 16 cores of 2.67 GHz CPUs and 32 GB of RAM. The experiment results are taken for the Netperf benchmarks. These are as follows:

6.1 SDP_ZCOPY_THRESH: 0

- ZCopy Disabled

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 1024
Recv    Send    Send                          Utilization      Service Demand
Socket  Socket  Message  Elapsed              Send    Recv    Send    Recv
Size     SizeSize      Time    Throughput  local    remote   local   remote
bytes    bytesbytessecs.    10^6bits/s      % S      % S      us/KB   us/KB
87380  65536    1024      10.10      2000.35     12.48     7.08      8.178   4.641
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m8192
RecvSend    SendUtilization      Service Demand
Socket Socket  Message  Elapsed              Send    RecvSend    Recv
Size   SizeSize   Time   Throughput  local   remote  local   remote
bytes  bytesbytessecs.   10^6bits/s  % S      % S      us/KB   us/KB
87380  65536   8192    10.10    9176.81   12.50    6.26     1.785   0.894
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 65536
RecvSend    SendUtilization      Service Demand
Socket Socket  Message  Elapsed           Send    Recv    Send    Recv
Size   SizeSizeTime    Throughput  local   remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536  65536     10.01    10133.04   8.18     6.63     1.058   0.858
```

```
[[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 1048576
RecvSend    SendUtilization      Service Demand
Socket Socket  Message  Elapsed           Send    Recv    Send    Recv
Size   SizeSizeTime    Throughput  local   remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536   1048576   10.01      9926.18   6.75     6.86     0.891   0.906
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C ---m 10485760
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed             Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
 87380  65536   10485760  10.02     10101.18   6.51     6.33     0.845   0.822
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 104857600
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed             Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
 87380  65536   104857600  10.05     10097.56   6.46     6.29     0.839   0.816
```

- **Result Summary for Zcopy Threshold = 0:**

Message Size	Throughput	Local CPU Utilization	Remote CPU Utilization
1 KB	2 Gbps	12.48	7.08
8 KB	9.1 Gbps	12.50	6.26
64 KB	10.1 Gbps	8.18	6.63
1 MB	9.9 Gbps	6.75	6.86
10 MB	10.1 Gbps	6.51	6.33
100 MB	10.1 Gbps	6.46	6.33

6.2 SDP_ZCOPY_THRESH: 64 KB

```
[root]# modprobeib_sdpsdp_zcopy_thresh=65536
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 1024
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536   1024    10.10     1883.85   12.52    6.24     8.710   4.344
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 8192
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size  SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536   8192    10.10     9104.75   12.49    6.26     1.798   0.902
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 65536
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     RecvSend    Recv
Size  SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536  65536    10.01     10131.13   8.39     7.03     1.085   0.910
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 131072
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
 87380  65536 131072    10.00     4901.58   2.77     7.02     0.742   1.878
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 1048576
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 1048576    10.10      6411.63   1.98     1.32     0.406   0.269
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 10485760
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
 87380  65536 10485760    10.00      6725.33   1.65     1.16     0.321   0.227
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 104857600
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 104857600  10.02      6677.41   1.54     1.44     0.303   0.282
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 1048576000
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size  SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536   1048576000  10.40      9676.80   5.90     5.69     0.799   0.771
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 2048576000
Recv   Send    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 2048576000    10.24     9602.05   5.86     5.58     0.800   0.762
```

- **Result Summary for Zcopy Threshold = 64KB:**

Message Size	**Throughput**	**Local CPU Utilization**	**Remote CPU Utilization**
1 KB	1.9Gbps	12.52	6.24
8 KB	9.1 Gbps	12.49	6.26
64 KB	10.1 Gbps	8.39	7.02
128 KB	4.9 Gbps	2.77	7.02
1 MB	6.4Gbps	1.98	1.32
10 MB	6.7Gbps	1.65	1.16
100 MB	10.1 Gbps	1.52	1.44
1 GB	9.7 Gbps	5.09	5.69
2 GB	9.6 Gbps	5.86	5.58

6.3 SDP_ZCOPY_THRESH: 1MB

```
[root]# modprobeib_sdpsdp_zcopy_thresh=1048576
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 8192
Recv   Send    Send                      Utilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536   8192    10.10      7907.50   13.71    6.47     2.272   1.073
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 65536
Recv   Send    Send                      Utilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536  65536    10.01      10129.16   8.25     6.36     1.067   0.824
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 1048576
Recv   Send    Send                      Utilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 1048576    10.01      9927.18   6.72     6.29     0.888   0.831
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 10485760
Recv   Send    Send                    Utilization      Service Demand
Socket Socket  Message  Elapsed           Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
 87380  65536 10485760    10.01      6519.21   1.65     1.75     0.332   0.353
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 104857600
Recv   Send    Send                    Utilization      Service Demand
Socket Socket  Message  Elapsed           Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 104857600    10.02      6690.25   1.60     1.21     0.313   0.238
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 1048576000
Recv   Send    Send                    Utilization      Service Demand
Socket Socket  Message  Elapsed           Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 1048576000    10.50      9586.22   5.90     5.63     0.806   0.770
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 2048576000
Recv   Send    Send                    Utilization      Service Demand
Socket Socket  Message  Elapsed           Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 2048576000    10.30      9545.88   5.88     6.02     0.807   0.826
```

- **Result Summary for Zcopy Threshold = 1MB:**

Message Size	Throughput	Local CPU Utilization	Remote CPU Utilization
8 KB	7.9 Gbps	13.71	6.47
64 KB	10.1 Gbps	8.25	6.36
1 MB	9.9Gbps	6.72	6.29
10 MB	6.5Gbps	1.65	1.16
100 MB	6.7Gbps	1.6	1.21
1 GB	9.6Gbps	5.9	5.63
2 GB	9.6 Gbps	5.88	6.02

6.4 SDP_ZCOPY_THRESH: 10 MB

```
[root]# modprobeib_sdpsdp_zcopy_thresh=10485760
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 65536
Recv   Send    Send                         Utilization       Service Demand
Socket Socket  Message  Elapsed             Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536  65536    10.01      9939.40   8.31     6.31     1.096   0.832
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C ---m 1048576
Recv   Send    Send                         Utilization       Service Demand
Socket Socket  Message  Elapsed             Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 1048576    10.01      9928.36   6.72     6.75     0.888   0.892
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C ---m 10485760
Recv   Send    Send                    Utilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSize     Time     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 10485760    10.10      6585.33   1.67     1.37     0.332   0.274
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C ---m 104857600
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 104857600    10.02      6598.41   1.50     1.29     0.298   0.257
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C ---m 1048576000
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
 87380  65536 1048576000    10.80      9320.61   5.88     5.98     0.827   0.841
```

```
[root]# LD_PRELOAD=libsdp.so netperf -H 172.31.134.1 -c -C -- -m 2048576000
RecvSend    SendUtilization       Service Demand
Socket Socket  Message  Elapsed            Send     Recv     Send    Recv
Size   SizeSizeTime     Throughput  local    remote   local   remote
bytes  bytesbytessecs.    10^6bits/s  % S      % S      us/KB   us/KB
87380  65536 2048576000    10.68   9210.10   5.84     6.36     0.831   0.905
```

- **Result Summary for Zcopy Threshold = 10 MB**

Message Size	Throughput	Local CPU Utilization	Remote CPU Utilization
64 KB	9.9 Gbps	8.31	6.
1 MB	9.9Gbps	6.75	6.31
10 MB	6.5Gbps	1.67	1.37
100 MB	6.6Gbps	1.5	1.29
1GB	9.3Gbps	5.88	5.98
2 GB	9.2Gbps	5.84	6.36

As we can see in the performance measurements, for higher message size keeping Zcopy threshold value around 1MB to 8 MB gives better performance in terms of CPU utilization of local and remote machines. While CPU utilization of machines has reduced, throughput of the communication has also reduced little bit for Zcopy. While for smaller message sizes, disabling the Zcopy by setting Zcopy threshold to 0, gives higher performance in terms of throughput as well as CPU utilization.

So as we present, Zcopy operation affects mainly CPU utilization of the system while throughput and latency has minimal effects.

Chapter 7

PERFORMANCE MEASUREMENT

In this chapter we present all performance comparisons for throughput and latency over IPoIB and SDP over IB. All the results are taken by keeping Zcopy threshold value equal to 64KB and with SDP Bonding enable.

7.1 THROUGHPUT& LATENCYOVER IPoIB

Message Size	Throughput	Latency
8KB	1206 Mbps	4.1 us
64KB	897 Mbps	6.3 us
4MB	2223 Mbps	4 us
1 GB	2622 Mbps	2.7 us

7.2 THROUGHPUT & LATENCY FOR SDP OVER IB

Message Size	Throughput	Latency
8KB	9104 Mbps	1.2 us
64KB	10131 Mbps	1.02 us
4MB	7112 Mbps	0.8 us
1GB	9673 Mbps	0.8 us

Chapter 8

CONCLUDING REMARKS &FUTURE WORK

8.1 CONCLUSION

As CPU speed increases CPU copying becomes expensive unless zero copy techniques are being used. SDP with Zcopy path does a great job of increasing the CPU effectiveness for application processing. SDP allows existing applications to transparently utilize Infiniband high performance capabilities without any code changes.

Using Zcopy for whole communication won’t give the optimal performance enhancement. In this dissertation, we presented the choice of Zcopy threshold value to ensure the highest possible performance enhancement. Zcopy gives higher performance for larger messages while for short messages Bcopy should be used in order to gain higher performance. Another parameter called MTU size also plays important role in ensuring optimal performance. In this dissertation, we presented that increase in size of MTU slightly from the default one, increases the performance significantly.

Another major aspect of any system design is Availability. System uses redundant copies of resources to tackle the failure issues. In this dissertation, we Proposed architectures for application aware as well as application transparent failover mechanisms to ensure the failover in case of link failure.

Application aware failover mechanism needs the reconnection from the application side and so application code needs to be changed. This Bonding mechanism is a simple technique to tackle the link failure issue. In this dissertation, we presented performance results with and without bonding which shows that Bonding doesn’t imposes much overhead in the normal operation of the system.

Another approach we proposed in this dissertation is application transparent failover. We proposed Socket Duplication technique to tackle link failure completely transparent to the application. This technique doesn’t need any change in the application at all and can be implemented in the kernel stack completely. As this technique is implemented in the kernel it imposes the performance degradation in the system’s communication.

Selection should be done by considering the need of the system. If the system is getting developed from scratch, application aware failover (SDP Bonding) can be used and in other hand if whole system is available, application transparent technique can be used to gain the performance enhancement.

In this dissertation, we tested both mechanisms for the GPFS; a SONAS system component but in actual this solutions can be deployed to any system as they operates on socket interface.

8.2 FUTURE WORK

Testing of all the proposed techniques for failover has not been carried out thoroughly at present due to lack of available time. In future, we would like to test all techniques for many more storage and cluster configuration.

Apart from the SDP over Infiniband, there are another similar configurations have been proposed such as Direct Socket over Myrinet. In future, we look forward for studying such configurations and try to solve their limitations. By doing so, it would be very useful in designing any cluster systemfor optimal performance and high availability.

BIBLIOGRAPHY

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

[2] SDP Specification. http://www.rdmaconsortium.org/home.

[3] Universal, 64/32-bit, 66/33MHz, Myrinet/PCI interfaces. http://www.myri.com/news/99903/index.html.

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

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

[6] P. Balaji. High Performance Communication Support for Sockets-based ApplicationsOver High-speed Networks. PhD thesis, The Ohio State University,Columbus, OH, 2006.

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

[8] P. Balaji, H. W. Jin, K. Vaidyanathan, and D. K. Panda. Supporting iWARPCompatibility and Features for Regular Network Adapters. In Workshop on Remote Direct Memory Access (RDMA): Applications Implementations, and Technologies (RAIT); held in conjunction with IEEE International Conference on Cluster Computing, 2005.

[9] P. Balaji, S. Narravula, K. Vaidyanathan, H. W. Jin, and Dhabaleswar K.Panda. On the Provision of Prioritization and Soft QoS in Dynamically Reconfigurable Shared Data-Centers over InfiniBand. In the Proceedings of the IEEEInternational Symposium on Performance Analysis of Systems and Software(ISPASS), 2005.

[10] 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 IEEE International Symposium on Performance Analysis of Systems andSoftware (ISPASS), 2004.

[11] P. Balaji, H. V. Shah, and D. K. Panda. Sockets vs RDMA Interface over10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck.In Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT); held in conjunction with IEEE International Conference on Cluster Computing, San Diego, CA, Sep 20 2004.

[12] P. Balaji, P. Shivam, P. Wyckoff, and D.K. Panda. High Performance UserLevel Sockets over Gigabit Ethernet. In Proceedings of the IEEE InternationalConference on Cluster Computing, September 2002.

[13] 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 onRemote Direct Memory Access (RDMA): Applications, Implementations, andTechnologies (RAIT 2004); held in conjunction with the IEEE InternationalConference on Cluster Computing, San Diego, CA, September 2004.

[14] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact ofHigh Performance Sockets on Data Intensive Applications. In Proceedings of theIEEE International Symposium on High Performance Distributed Computing(HPDC), 2003.

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

[16] A. Cohen, S. Rangarajan, and H. Slye. On the Performance of TCP Splicingfor URL-aware Redirection. In the Proceedings of the USENIX Symposium onInternet Technologies and Systems, October 1999.

[17] Remote Direct Memory Access Consortium. http://www.rdmaconsortium.org.

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

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

[20] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. PerformanceCharacterization of a 10-Gigabit Ethernet TOE. In Proceedings of Hot Interconnects Symposium, 2005.

[21] 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 Networksof Workstations, Clusters and Grids: A Case Study. In Proceedings of theSupercomputing (SC), 2003.

[22] Internet Engineering Task Force. http://www.ietf.org.

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

[24] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Transparently Achieving Superior Socket Performance using Zero Copy Socket Direct Protocol over20 Gb/s InfiniBand Links. In Workshop on Remote Direct Memory Access(RDMA): Applications Implementations, and Technologies (RAIT); held inconjunction with IEEE International Conference on Cluster Computing, 2005.

[25] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Zero Copy SocketsDirect Protocol over InfiniBand - Preliminary Implementation and PerformanceAnalysis. In IEEE Hot Interconnects: A Symposium on High PerformanceInterconnects, 2005.

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

[27] J. Hurwitz and W. Feng. Initial End-to-End Performance Evaluation of10-Gigabit Ethernet. In IEEE Hot Interconnects: A Symposium on High-Performance Interconnects, Palo Alto, California, 2003.

[28] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet onCommodity Systems. IEEE Micro, 2004.

[29] 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 theAmmasso Gigabit Ethernet NIC. In Workshop on High Performance Interconnects for Distributed Computing (HPI-DC); In conjunction with HPDC-14,2005.

[30] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer overVirtual Interface Architecture. In Proceedings of the International Conferenceon Cluster Computing, 2001.

[31] 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-1 Workshop, held in conjunction with Int'lSymposium on High Performance Computer Architecture (HPCA-8), 2004.

[32] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand.http://nowlab.cse.ohio-state.edu/projects/mpi-iba/index.html.

[33] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Work-stations: Illinois Fast Messages (FM). In Proceedings of Supercomputing (SC),1995.

[34] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. InternetSmall Computer Systems Interface (iSCSI), RFC 3720, April 2004.

[35] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Socketsand RPC over Virtual Interface (VI) Architecture. In International Workshopon Communication and Architectural Support for Network-Based Parallel Computing (CANPC), 1999.

[36] P. Shivam, P. Wyckoff, and D. K. Panda. Can User-Level protocols take advantage of Multi-CPU NICs? In Proceedings of the IEEE International Paralleland Distributed Processing Symposium (IPDPS), 2002.

[37] P. Shivam, P. Wyckoff, and D.K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In Proceedings of Supercomputing(SC), 2001.

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

[39] J. Stone and C. Partridge. When the CRC and TCP Checksum Disagree. InACM SIGCOMM, 2000.

[40] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda. Shared Receive Queue basedScalable MPI Design for InfiniBand Clusters. In International Parallel andDistributed Processing Symposium (IPDPS), 2006.

[41] S. Sur, M. J. Koop, and Dhabaleswar K. Panda. High-Performance and ScalableMPI over InfiniBand With Reduced Memory Usage: An In-Depth PerformanceAnalysis. In Proceedings of SuperComputing, 2006.

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

[43] SithaBhagvat. Designing and Enhancing the Sockets Direct Protocol (SDP) over iWARP and InfiniBand. MS Thesis, The State Ohio University, 2006

[44] www.**netperf**.org

[45] Abhinav Vishnu,Amith R. Mamidala,SundeepNarravula andDhabaleswar K. Panda. Automatic Path Migration over InfiniBand: Early Experiences in IEEE 2007.

[46] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning,R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon,V. Venkatakrishnan, and S. K. Weeratunga. The NASParallel Benchmarks. Number 3, pages 63–73, Fall1991.

[47] Yiu-Fai Sit, Cho-Li Wang, Francis Lau. Socket Cloning for Cluster-BasedWeb Servers at Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER’02).

[48] Bob Woodruff, Sean Hefty, Roland Dreier, Hal Rosenstock. Introduction to the InfiniBandCore Software.

[49] Ariel Cohen. A Performance Analysis of the Sockets Direct Protocol (SDP) with Asynchronous I/O over 4x Infiniband in IEEE 2004.

[50] www.rdma-linuxlist.com