UNIVERSITY of CALIFORNIA
SANTA CRUZ

NETWORKING FOR OBJECT-BASED STORAGE DEVICES

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Garima Tripathi

December 2004

The thesis of Garima Tripathi is approved:

Professor Katia Obraczka, Chair

Professor Scott A. Brandt

Professor J. J. Garcia-Luna

Robert C. Miller
Vice Chancellor for Research and
Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

Networking for Object-based Storage Devices

by

Garima Tripathi

Object-Based Storage Devices (OSDs) are geared towards applications where scientific data is required to be stored over the network. The files tend to be rather large, sometimes even going up to several gigabytes in size, and thus each OSD is required to have large storage capacities - scaling up to several petabytes. Having these devices available on the network entails that they should be able to act reasonably fast as servers without spending much time in search and lookup. At the same time, since the data size being transferred is fairly large, the network also needs to be able to support them well. Because of their role as servers, such networked storage devices place stringent requirements on the network infrastructure. Essentially they require large amounts of data to be transferred efficiently incurring minimum delay. Therefore, what they require is a high-speed, reliable network with minimum or no congestion. In this thesis, we have tried to quantify the performance requirements of a network of OSDs with a focus on LANs. In doing so, we have analysed the existing network protocols vis-a-vis these requirements using the network simulator "ns-2" to simulate a typical OSD testbed. We analyzed TCP-Reno, and several TCP-variants like High-Speed TCP, BI-TCP and TCP with large send window and large packet size. Our results showed that under default window size, 1500 bytes packet size, and no background traffic, BI-TCP stood out. But when we loaded the network with background TCP traffic, both BI-TCP and default TCP performed very poorly, and High-Speed TCP stood out. Using large packet size for all TCP-variants and large send window size for TCP-Reno, we found that TCP-Reno stands out under no background load, but performs very poorly under

moderate background load. Based on these observations, we created a reliable version of UDP which incorporated the desirable properties of TCP (flow-control, flexible congestion control, and of course, reliability, but no punitive AIMD) with the light-weight and minimum overheads features of UDP. We compared Reliable UDP with the TCP-variations, and found that with large packet sizes and large send window, Reliable UDP performs better than High-Speed TCP and BI-TCP, but TCP-Reno with large packet size and large send window is still better. However, in the presence of moderate background traffic, Reliable UDP out-performs all the others. It might be worth mentioning here that the background traffic did not result in any packet drop in any scenario.

This thesis is dedicated to my husband Sunay, whose support and encouragement not only helped me get through the tough years of school, but who also made me realize my own potential.

This thesis is also a tribute to my parents, who instilled in me the ambition which was so crucial for achievement of my goals.

Acknowledgements

I would like to thank Dr. Katia Obraczka, my advisor, who stuck with me and guided me throughout the course of this work. I'll always be grateful for her patience with me while I took my time getting this thesis together. I would also like to thank Prof. Scott Brandt, who appreciated that my interest was more in networking, and allowed me to work on this topic, although I had been assigned to the Systems group.

Chapter 1

The network of OSDs

Object-Based Storage devices (OSDs) represent emerging network storage technology that is cross-platform, interoperable, and has created higher-performance devices with more intelligence and flexibility than traditional servers and storage devices. Much of the novelty of the OSD approach lies in its architecture. An OSD system consists of three different abstract components: the server, the client, and the MetaData Server. These may or may not co-exist on a single machine. In fact, the network can easily be a heterogenous one with a mixture of commodity machines as well as server-class machines. In any configuration, however, the MetaData Server is the only component that has an overall view of the system ([3]) Usually present as a cluster (for redundancy and efficiency reasons), the MetaData Servers manage the namespace and directory hierarchy, file and directory permissions, and the mapping from files to objects. The MetaData Server cluster is otherwise not involved in the storage and retrieval of data, allowing for very efficient concurrent data transfers between large numbers of clients and OSDs. The client, which handles user requests and provides a user interface, contacts the MetaData Server on 'opens' and 'closes' to get the information it needs to access the objects it requires. Objects are stored on one or more servers. Each server is independent and has no knowledge of any other

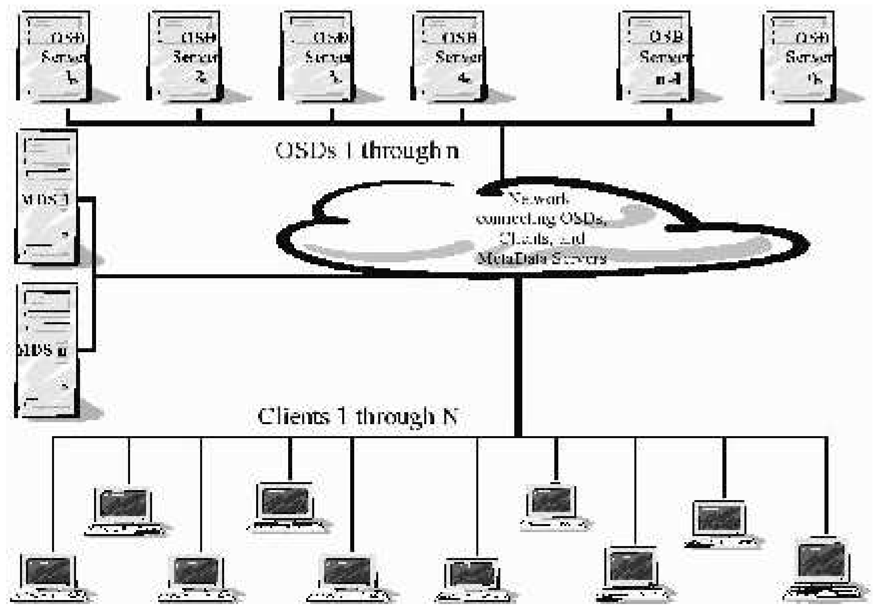


Figure 1.1: A typical OSD network: 1-10 MetaData Servers manage the requests, 1000-10,000 OSDs store the data, and 10,000-100,000 clients can connect at to the network

component of the system, including other servers. It would make sense to have each commodity machine consist of both a client and a server, while the MetaData Server, which is the potential bottleneck of the system, could be a server-class machine. Figure 1.1 shows a high-level picture of the entire system, including permitted communications between each component. The entire system is designed to exploit parallelism to improve the handling of very large files without penalizing small files as many striped file systems do. A slightly more detailed description of each component follows:

- **Clients** - Functionally the client nodes are similar to the typical client nodes that are seen in any network. The clients open and close files, read data and also write data to the OSD servers.In a typical setup, we expect there to be anywhere from 10,000 to a 100,000 clients. Out of these around 10% will be I/O nodes - performing read/write requests on the servers.

- **MetaData Servers** - An OSD network becomes different from traditional networks because of its MetaData Servers. Traditionally, metadata and data are managed by the same file system, on the same machine, and stored on the same device. For efficiency, metadata is often stored physically close to the data it describes. Unlike a typical client-server network, where the server receives all the client requests and then processes and services them, in this setup, the server functionality is divided between the MetaData Servers and the OSD servers. Here's how it works - all requests sent by the client are received by the MetaData Server, which maintains all the meta-data - i.e. all the information about the actual data. It acts like an index for all the data - where it is placed, how it is striped over multiple OSDs (if at all), and what is the sequence of the striping. When a client sends a read/write request for some file, the MetaData Server looks it up, and locates the required file on the OSD it is residing on. It then sends a request to the OSD in question to respond to the client's request. This saves a significant amount of overheads which would otherwise affect the performance of the OSDs. Since the request sizes from the clients are assumed to be rather small (128b on an average), and the size of metadata is not considered to be very big, we are assuming that a typical set-up will have from 1-10 MetaData Servers.

- **OSD Servers** - The OSDs are the repositories of data. They have large storage capacities - reaching upto several petabytes. They store the files as sets of objects striped over multiple OSDs at times and demonstrate little locality of reference[19]. As explained in by Long et al

  > A key advantage of OSDs in a high-performance environment is the ability to delegate low-level block allocation and synchronization for a given segment of data to the device on which it is stored, leaving the file system to decide only on which OSD a given segment should be placed. Since this decision is quite simple and allows massive parallelism, each OSD need only manage concurrency locally, allowing a file system built from thousands of OSDs to achieve massively parallel data transfers. Additionally, OSDs can each manage their own storage

consistency, removing the need to run a system-wide consistency check that could take days on a petabyte-scale traditional file system. [13]

In a typical setup, there are 1000-10,000 OSDs which are servicing client requests.

The three components - clients, MetaData Servers and OSDs in the setup will be connected via high-bandwidth ethernet. We assume for the purposes of the experiments that there is no communication between OSDs themselves as also none between the MetaData Servers. A client would typically make only open/close requests to MetaData Servers and upon receiving informartion about the target OSD which would be able to provide it service, it would make only read/write requests to the OSD. There isn't also much communication expected between the MetaData Servers and the OSDs. The exceptions to this are at system setup and system teardown time. When the network is freshly established, the MetaData Servers would be without any information about OSDs and their respective contents. As a result the first few requests from the clients would result in cache misses as the MetaData Server tries to locate the right information by communicating with the OSDs. Once the MetaData Servers get populated with information about the data, we expect these cache misses to subside. Similarly, when the setup is being dismantled, there would be some amount of log cleaning which would again result in larger amount of communication between the MetaData Servers and OSDs. For our experiments, we ignore these scenarios.

The requirements of an OSD network are largely derived from the fact that very large amounts of data will be transferred over the network very frequently. Some key properties of such a network are

1. **Reliabilty** - The most striking feature of an OSD network is the large quantities of data transfer involved. The systems can contain several petabytes of data, and filesizes can be upto several gigabytes large. With large quantities of data being transferred, it is easy

to see that any data loss will lead to lot of problems. While data loss in any network is undesirable, in an OSD network, it is especially unwelcome. In the event of data loss in the network, the overheads involved in re-transmitting can be very punishing for system preformance since it faces a drop in goodput. At the same time, the possibility of data loss also requires receivers (both clients as well as OSDs) to maintain large buffers in order to supply contiguous data to the applications. Such large buffers might be undesirable since they tend to escalate the system cost.

2. **High Throughput** - High throughput is a desirable network property in any network, but much moreso in an OSD network because a low throughput can be indicative of congestion, or some other network problem. Since we are dealing with scientific applications in an OSD network, a high throughput can be crucial to the correctness of the solution provided by the application.

3. **Congestion Control** - Since the amounts of data being transferred in an OSD network are very large, it is but obvious that any circumstance that leads to data loss is highly undesirable. Due to this reason, good congestion control policies are very important for OSD networks. The system should be geared to first of all avoid any possible occurrance of congestion, and if, for some reason, congestion is unavoidable, its congestion control should be fast and efficient without compromising on the throughput rate.

4. **Low latency** - Low latency, like high throughput is another required feature for the OSD network. High latency in data transfer can result in congestion, data loss and even incorrectness of the application.

5. **Flow Control** - Large data transfers also lead to concerns about flow-control on the server side. There should be enough data to keep an optimal throughput, but it should also not

be so much that the client gets overwhelmed with it and is not able to keep up with it. There should only be so much data sent, as the client can process.

Based on the above requirements, we have examined a variety of existing solutions to high-bandwidth networking, and then have proposed our recommended solution to networking in the OSD systems.

Chapter 2

Related Work

Large data transfers between high-performance computing systems is not a new problem. On the contary, several efforts have been made in this field to achieve a high throughput for similar scenarios. Among the noteworthy attempts are HTPNET [5], DCCP [12], RDMA, and RDDP [2]. In addition to these, several variations of TCP have also been suggested which deal with modifying the original protocol to accomodate the various changes in technology. We will first take a look at some of these TCP-variations.

2.1 High-Speed TCP

This version of TCP has been proposed by Sally Floyd[8], and is relatively new. The major premise behind proposing this version is the fact that it takes a long time for a TCP connection to make a full recovery from packet loss for high-bandwidth long-distance connections. To illustrate this, the authors use the standard TCP average send rate equation

$$W = \frac{1.2}{\sqrt{p}} \tag{2.1}$$

where w is the congestion window size in MSS sized segments and p is the steady state packet drop rate.

So, for example, a Standard TCP connection with 1500-byte packets, 100 ms round-trip time, and a steady-state throughput of 10 Gbps would require an average congestion window of 83,333 segments and at most one drop (or mark) every 5,000,000,000 packets (or equivalently, at most one drop every 1 2/3 hours). This is not realistic. High-Speed TCP aims to rectify this by modifying TCP congestion control. In a low congestion scenario, High-Speed TCP behaves just like Standard TCP, but it becomes more aggressive than Standard TCP when cwnd is high. For this purpose, it uses a modified TCP response function. High-Speed TCP implements three parameters - Low_Window, High_Window and High_P, and provides a threshold by which TCP will perform differently from standard TCP. The Low_Window parameter indicates the threshold of which response function to use - highspeed or standard; the High_Window parameter is the average congestion window to be achieved; and the High_P parameter is the tcp packet drop rate.When the congestion window of a TCP connection is at most Low_Window, High Speed TCP proposes to use the normal Additive Increase Multiplicative Decrease (AIMD) algorithms; when it is greater than Low_Window, it proposes the use of the HighSpeed response function with alternate values of AIMD. As such, when performing high throughput transfer with HighSpeed TCP, it should be less prone to losses and hence able to maintain a high speed connection. This is directly related to the reduced number of rtts required to keep the size of the congestion window constant. The direct constants involved with the change in AIMD are related to the values of High_Window and High_P.

With all the suggested enhancements, High-Speed TCP seems to be a very strong contender for use in the OSD networks. However, there are several reasons that it does not jump out as the obvious choice for OSD systems.

- **long-distance vs LAN/WAN** HSTCP is basically intended for high-speed long distance networks. In our case, however, OSDs are setup on a LAN or WAN at the most, therefor the bandwidth-delay product is not as high as desired to deploy HSTCP.

- **Fairness** Another issue with HighSpeed TCP is that it may impose a certain degree of unfairness as it does not reduce its transfer rate as much as Standard TCP. As suggested by Xu et al in [22], High-Speed TCP has

  > .. a serious RTT unfairness problem when multiple flows with different RTT delays are competing for the same bottleneck bandwidth.

  In fact, it has been demonstrated that High-Speed TCP has RTT unfairness proportional to $\frac{RTT_2}{RTT_1}^{4.56}$ if there are two flows with RTT_1 and RTT_2 present.

- **Stability** This is a very important factor in the deployment of any protocol in a network. Bullot et al [4] have defined a stability index which is the standard deviation normalized by the average throughput index. With this parameter, High-Speed TCP has been shown to have a very poor stability index.

- **Congestion Control** High-Speed TCP is designed to respond the same way to congestion as TCP.Its bigger focus, as a protocol, is to deploy a limited slow start policy, which is basically a mechanism for limiting the number of segments by which the congestion window is increased for one window of data during slow-start. This policy is designed to improve performance for TCP connections with large congestion windows since the current slow-start procedure in TCP can result in increasing the congestion window by thousands of segments in a single round-trip time. Such an increase can easily result in thousands of packets being dropped in one round-trip time. However, in OSD networks, where we already expect the congestion windows to be extremely large, the possibility of data loss due to packet drops is less likely than performance decrease due to congestion.

What is needed is a more customised solution for OSDs since the TCP congestion control mechanism is very inadequate for this type of scenario.

2.2 BI-TCP

BI-TCP is a new protocol that ensures linear RTT fairness under large windows while offering both scalability and bounded TCP-friendliness. The protocol combines two schemes called additive increase and binary search increase. When the congestion window is large, additive increase with a large increment ensures linear RTT fairness as well as good scalability. Under small congestion windows, binary search increase is designed to provide TCP friendliness. The protocol views congestion control as a searching problem in which the system can give a yes/no feedback through packet loss as to whether the curtrent sending rate is higher than the network's capacity. The current minimum window is assumed to be the window size where there is no packet loss incurred. Once the maximum window size is known, a binary search technique is applied to set the target window size at the midpoint between minimum and maximum window size. Binary search allows for aggressive network probing when there is a large difference between the current window size and target window size, and this probing tapers as the current window size approaches the target window size. The unique feature about the binary increase is that it is logarithmic, and therefore its increase rate reduces as the current window size approaches saturation point. The additive increase feature works in conjunction with the binary increase to achieve the target window size. A maximum increment factor called S_{max} is identified by the protocol. The authors realise that when the distance to the midpoint from the current window is too large, increasing the window size to the midpoint can result in a lot of stress for the network. At this point, the S_{max} comes in handy. The window size is increased linearly by S_{max} till it is viable to logarithmically increase it. While offering these features, BI-TCP is also scalable, being able to scale its bandwidth share to 10Gbps around 3.5e-8 loss rates (comparable to HS-TCP which reaches 10Gbps at 1^{-7}).

The main drawback of BI-TCP is that as the loss rate reduces further below 1^{-8}, its sending rate does not grow fast enough. While the authors contend that this is not the case and that there is no trade-off between RTT fairness and scalability, and show that network utilisation in BI-TCP is almost 94%, the fact is that a high network utilisation is not the only criterion - the goodput is ultimately what counts.

Another issue is that just like High-Speed TCP, BI-TCP also has very poor stability in a system under noload. Under load, however, it definitely improves and shows better stability than HS-TCP. It remains doubtfull, however, that we can adopt it for OSD systems, based solely on its performance under load.

As for Fairness, BI-TCP is shown to be highly aggressive as compared to other protocols. This actually works out in its favor since we are not really expecting any competition between protocols in a dedicated OSD system. Also, with other BI-TCP flows in the same system, there is adequate fairness that there is no contention issues, so that works for us as well.

2.3 RDDP

The Remote Direct Data Placement [2] protocol provides information to place the incoming data directly into an upper layer protocol's receive buffer without intermediate buffers. This removes excess CPU and memory utilization associated with transferring data through the intermediate buffers. Direct Data Placement Protocol (DDP) enables an Upper Layer Protocol (ULP) to send data to a Data Sink without requiring the Data Sink to place the data in an intermediate buffer - thus when the data arrives at the Data Sink, the network interface can place the data directly into the ULP's buffer. This can enable the Data Sink to consume substantially less memory bandwidth than a buffered model because the Data Sink is not required to move the data from the intermediate buffer to the final destination. Additionally, this can also enable the network protocol to consume substantially fewer CPU cycles than if the CPU was used to move the data, and removes the bandwidth limitation of only being able to move data as fast as the CPU can copy the data. DDP preserves ULP record boundaries (messages) while providing a variety of data transfer mechanisms and completion mechanisms to be used to transfer ULP messages.

2.4 RDMA

Traditional hardware and software architecture imposes a significant load on a server's CPU and memory because data must be copied between the kernel and application. Memory bottlenecks become more severe as connection speeds exceed the processing power and memory bandwidth of servers.

RDMA gets around this by implementing a reliable transport protocol in hardware on the NIC and by supporting zero-copy networking with kernel bypass. Remote Direct Memory Access is a network interface card (NIC) feature that lets one node directly place information into the memory of another node. The technology reduces latency by minimizing demands on bandwidth and processing overhead. While zero-copy networking lets the NIC transfer data directly to or from application memory on the one hand, eliminating the need to copy data between application memory and the kernel; kernel bypass, on the other hand, lets applications issue commands to the NIC without having to execute a kernel call. The RDMA request is issued from user space to the local NIC and over the network to the remote NIC without requiring any kernel involvement. This reduces the number of context switches between kernel space and user space while handling network traffic. When an application performs an RDMA Read or Write request, no data copying is performed. The RDMA request is issued from an application running in user space to the local NIC and then carried over the network to the remote NIC without requiring any kernel involvement. Request completions might be processed either entirely in user space (by polling a user-level completion queue) or through the kernel in cases where the application wishes to sleep until a completion occurs. RDMA operations let an application read from or write to the memory of a remote application. The remote virtual memory address for the operation is carried within the RDMA message. There is no need for the remote application to do anything other than register the relevant memory buffer with its local NIC. The CPUs in the remote node are not at all involved in the incoming RDMA operation, and they incur no load.

RDMA presents a very viable solution to our problem space - it is fast, efficient, and manages to utilise the bandwidth in the system. However, this technology is still pretty nascent, and is still evolving. Another problem is that the NICs required to make the RDMA work are pretty expensive, and even now they are hard to find. This does affect its utility for our purposes. This is not meant to say that RDMA is not suitable - just that it is not ready yet to be deployed on such a large scale.

2.5 HTPNET

The High-performance Transport Protocol[5] was designed to overcome protocol processing bottleneck, based on a highly parallel architecture and to exploit the evolving characteristics of high speed networks. It used an out-of-band signaling system based upon transmitter-paced periodic exchanges of state information between end systems. This mechanism exhibited several attractive properties which have been demonstrated to perform efficiently in a simulated high-speed environment with high bandwidth delay product. Experiments using HTPNET revealed promising packets transfer rate of 12,000 to 13,000 packets/sec with a packet size of 2 Kilobytes. As a result, a data transfer rate of more than 200 Mb/s could be achieved.

The biggest issue with deploying HTPNET in an OSD network is the fact that HTPNET is designed to exploit highly parallel computing architectures, in which

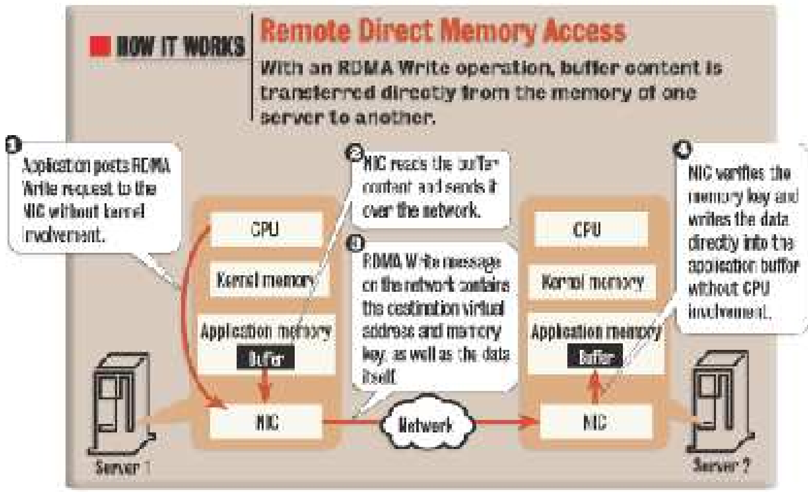


Figure 2.1: RDMA- How it works (Diagram courtesy Ariel Cohen [6])

> the protocol processing is decomposed into several concurrently operating finite state machines. Each of these finite state machines is responsible for different control functions, and the finite state machines act collectively by synchronising periodically using control messages. [5]

As is clear from the statement above, HTPNET is meant more for parallel architectures than the kind of machines we are seeking to employ in OSD networks. Also, HTPNET banks very heavily on using an additional multiprocessor-based subsystem to decouple protocol processing from the main operating system. Also, it aims to use several finite-state machines (like transputers) which are designed to exploit a parallel implementation and achieve high throughput. This design allows for very limited deployment and makes scaling very difficult. The use transputers also does not work out for clients since clients themselves need large processing capabilities.

2.6 DCCP

The Datagram Congestion Control Protocol implements a congestion-controlled, bidirectional stream of unreliable datagrams. Specifically, DCCP provides:

- An unreliable flow of datagrams, with acknowledgements.
- Reliable handshakes for connection setup and teardown.
- Reliable negotiation of options, including negotiation of a suitable congestion control mechanism.
- Mechanisms allowing a server to avoid holding any state for unacknowledged connection attempts or already-finished connections.
- Congestion control incorporating Explicit Congestion Notification (ECN) and the ECN Nonce, as per [RFC 3168] and [RFC 3540].
- Acknowledgement mechanisms communicating packet loss and ECN mark information. Acks are transmitted as reliably as the relevant congestion control mechanism requires, possibly completely reliably.
- Optional mechanisms that tell the sending application, with high reliability, which data packets reached the receiver, and whether those packets were ECN marked, corrupted, or dropped in the receive buffer.
- Path Maximum Transfer Unit (PMTU) discovery, as per [RFC 1191].[12]

DCCP is intended for applications, such as streaming media and Internet telephony, where reliable in-order delivery, combined with congestion control, can result in some information arriving at the receiver after it is no longer of use. DCCP provides standard congestion control mechanisms for such applications. It enables the use of ECN, along with conformant end-to-end congestion control, for applications that would otherwise be using UDP. In addition, DCCP implements reliable connection setup, teardown, and feature negotiation. DCCP's target applications require the flow-based semantics of TCP, but do not want TCP's in-order delivery and reliability, or would like different congestion control dynamics than TCP. Although this works very well for the streaming media kind of applications, one can easily guess that it wouldn't work for the OSD kind of systems. In a network where the majority of data flow comprises of scientific data, in-order delivery cannot be dispensed with, and reliability is a must. Hence although DCCP does tackle the problem of congestion control and large data transfer, it still does not work for our situation.

2.7 Infiniband

The InfiniBand specification defines the architecture of the interconnect that will pull together the I/O subsystems of the next generation of servers and will ultimately even move to the powerful desktops of the future. The architecture is based on a serial, switched fabric that in addition to defining link bandwidths between 2.5 and 30 Gbits/sec, resolves the scalability, expandability, and fault tolerance limitations of the shared bus architecture through the use of switches and routers in the construction of its fabric. Infiniband is promising to be the next panacea for high-performance computing. However, it is still in its nascent phase, and while more and more commercial vendors are pledging support for it, it is yet to be seen whether it finally will deliver all that it is promising.

Chapter 3

Experiments with existing transport protocols

3.1 The Testbed

Since the OSD network needs reliable data transfer with minimum congestion, while providing high throughput, the problem space can easily be narrowed down to the transport-layer. The ideal transport protocol should be able to support the large file sizes as well as the high bandwidth required in an OSD (or any high performance computing) network. Our first step was to analyze existing transport protocols. For our experiments, we used the network simulator, ns-2 [7]. Apart from providing a common baseline for all the protocols, using the simulator also allowed us to play with resources like bandwidth and network load. For example, we could try out high hypothetical bandwidths which are not even available right now.

Using the simulator, we wanted to create a network which would mimic realistic OSD scenarios as discussed in Chapter 1. So we created a basic testbed where a single OSD catered to 10 client nodes. (Since the communication with MetaData Servers was considered minimal, we left MetaData Servers out of the testbed). All the links had a bandwidth of 10Gbps with a latency of 1 millsec. The Figure 3.1 illustrates the testbed. The OSD is connected to an edge router R1 which is connected to another edge router R2 which feeds a cluster of 10 modes

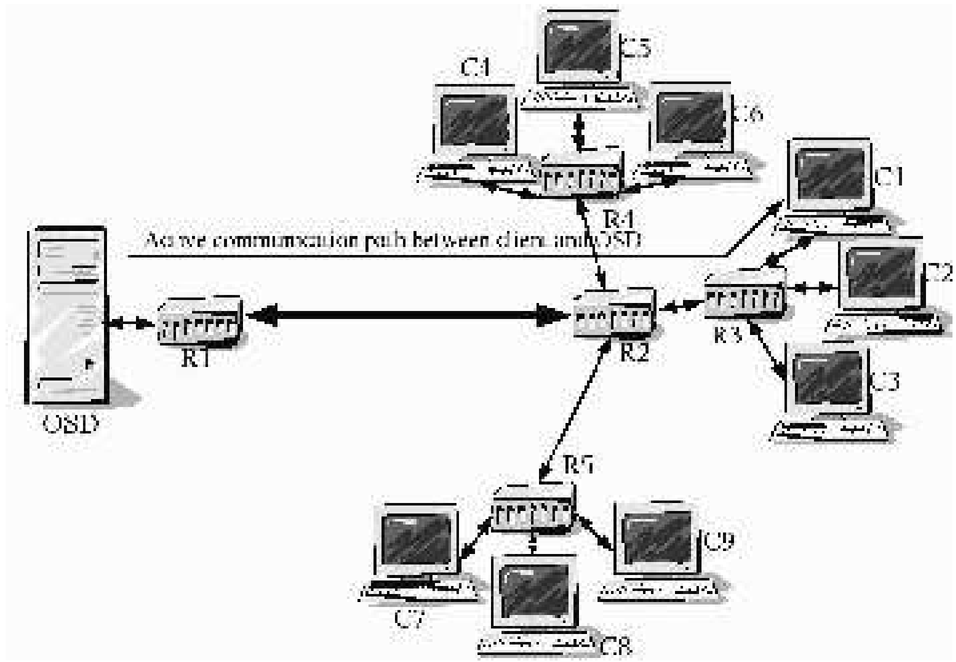


Figure 3.1: The Testbed

connected to it via other routers R3, R4, and R5 (connected to R2). The latency of each link was 1 millisec except for the link between edge router R2 and cluster Routers R3, R4, and R5 which have 0 millisec latency. We expect that the node clusters will be connected to edge routers via a high speed switch which will have no latency. We observed the read/write behaviour between the OSD and node C9 while nodes C1 to C6 were used to generate background traffic when needed. The read/write traffic under observation between the OSD and node C9 would share the 10Gb link between R1 and R2 when background traffic was needed. Since we knew that the OSD network would be a heterogenous one comprising of various operating system platforms, we decided to try out the experiments with the TCP send window sizes of two popular operating systems - Linux and Solaris. Linux TCP window size is 32KB (32768 bytes), and that of Solaris is 48KB (49152 bytes). We ran the experiments with each of these window sizes while keeping the packet size to be the ns-2 default 1500 bytes. Thus, Linux had a send window of 22 packets, and Solaris had a send window of 33 packets. The traffic simulated was in the form of FTP applications transferring files from the clients to the OSDs and vice-versa. Since the links were full-duplex, and there was equal amount of traffic being written to by both OSDs and clients, we decided to focus on the traffic read by the client C9 from the OSD since that was our primary interest in the whole set-up. Any background traffic was generated by running FTP over TCP (with default parameters) between nodes C1-C6 and OSD. Each background connection added 10-50Mbps load depending on the congestion.

3.2 Results

3.2.1 TCP with default packet size and window size

System without background load

For the first experiment, we only activated a single connection, i.e. only one client was reading data from the OSD. So the OSD was just writing to one client and all the other clients were silent and made no contribution at all to the network load. Although this type of scenario is not typical in networks, it was crucial since it allowed us to determine the most effective throughput that we could achieve from an unloaded link.

The findings from the first experiment were quite surprising. Although the bandwidth available for the link under test was 10Gb, with 10Gb links from the connected switches to the clients itself, the maximum throughput that TCP in Linux was able to provide was approximately 45 Mbps only. As can be imagined, this was a rather dismal result when viewed in light of the available resource a.k.a. the 10Gb link. Similarly, with Solaris' send window size of 48 KB, the throughput achieved was just under 70Mbps. Figure 3.2 displays the throughput graphs for both the window sizes. A quick look at the figure gave us several insights into the TCP

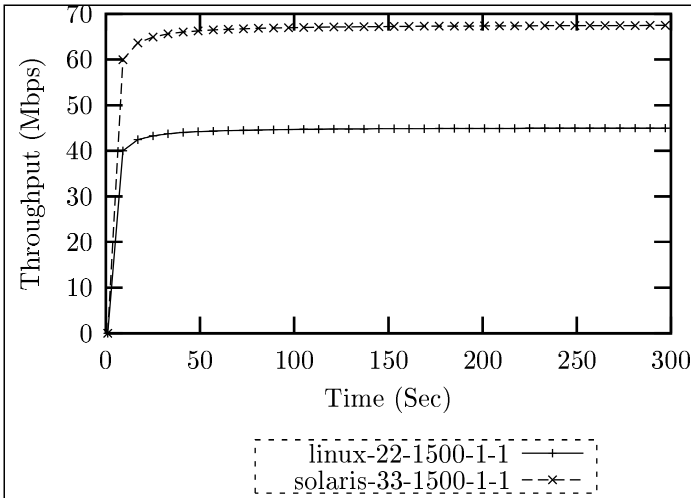


Figure 3.2: TCP throughput for OSD systems without any background noise. Two operating systems, Linux and Solaris were evaluated for their TCP throughput with their default send window sizes and packet size of 1500Bytes

performance.

- The first noticeable fact about the graphs was the steep, but steady rise in the throughput in the first few seconds itself. This initial burst of throughput is a characteristic of most TCP graphs, and was brought on by the slow-start phase of TCP. However, as we could see, despite the what the name might indicate, slow start isn't really all that slow. Infact, for Linux, achieving a thoughput of almost 40Mbps within 10 seconds was quite remarkable. Similarly, for Solaris, the throughput was even higher at almost 60Mbps in the first 10 seconds or so. This phase showed an exponential increase in the send window size, and also re-affirmed the idea that window size directly affects throughput (hence the higher throughput of Solaris as opposed to Linux).

- As the send window stabilized (when we reached the knee of the graph), we noticed the throughput levelling out. This was not really a surprise. We could see that at this time TCP's congestion control had come into play. The sliding window had stabilized, and the throughput continued to be stable hereafter. From simple back of the envelope calculations, we could see that the throughput achieved was much less that what would actually be expected from such a high bandwidth connection.

$$\begin{aligned} Throughput &= \frac{TransferSize}{TransferTime} && (3.1) \\ TransferTime &= RTT + \frac{1}{Bandwidth} \times TransferSize[16] && (3.2) \end{aligned}$$

An important fact that deserves notice here is that although the throughput is so low, there are actually no packet drops in the scenario. This leads us to the conclusion that even in an uncongested link, TCP will not be able to give us a decent throughput.

System with background traffic

In the second experiment, we wanted to determine throughput on a loaded link. For this, we ran 25 FTP connections with default TCP parameters between other nodes and OSD causing the link under observation to become congested. Each connection generating background traffic added approx 10-50 Mbps of traffic each depending on the congestion. We used TCP as the background traffic generator because we expect the OSD network to mostly contain TCP (or its variant) traffic and very small to non-existant UDP traffic.

The results of the second experiment were even more interesting. We noticed the TCP throughput appeared as a saw-toothed graph (Fig 3.3) where TCP's congestion control policies came into effect. This behaviour was demonstrated by both Linux as well as Solaris. The initial rise in the graph indicated the slow-start phase of TCP congestion window. The immediate drop indicated the multiplicative decrease that it went through as it first encountered congestion.

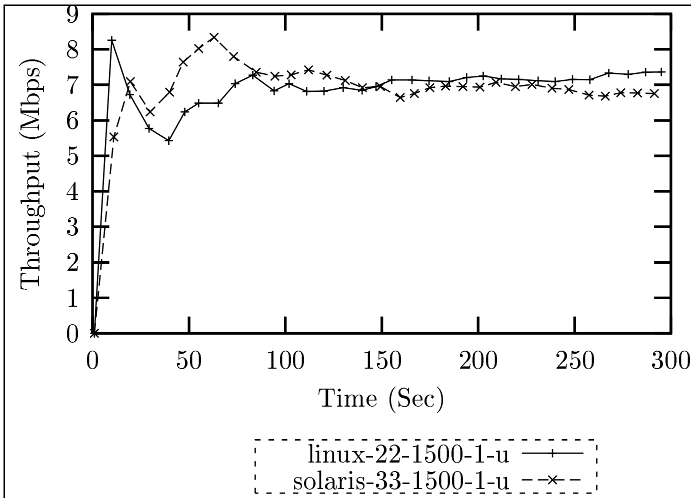


Figure 3.3: TCP throughput for OSD systems with high background noise. Two operating systems, Linux and Solaris are evaluated for their TCP throughput with their default send window sizes and packet size of 1500 bytes.

Thereafter, the pattern continued till the size of the window stabilized to take into account the congestion on the link.

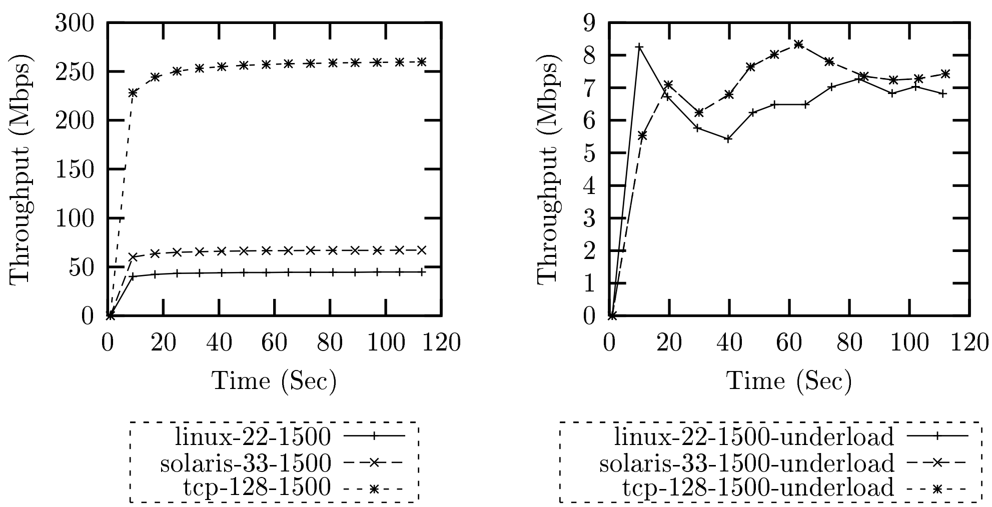
Again, we noticed that although the links experienced congestion, there was no packet drop that caused data loss. This led us to believe that the low throughput was somehow connected to maximum send window of default TCP.

3.2.2 TCP with large send window and large packet size

The next step after the conclusion we drew from the previous experiment was to try out TCP with a larger send window. This decision was also influenced by the fact that the Internet Engineering Task Force has standardized the large window option in TCP with the RFC 3390 [1]. With this RFC they have increased

> the permitted upper bound for TCP's initial window from one or two segment(s) to between two and four segments. In most cases, this change results in an upper bound on the initial window of roughly 4K bytes (although given a large segment size, the permitted initial window of two segments may be significantly larger than 4K bytes). [1].

Although this increase in send window size by IETF was pretty significant, we wanted to try out something even larger than this. Given the fact that our OSD servers and clients are expected to have large memories available, we decided to play with the window sizes, and finally settled on trying out a send window of 128 packets. This was almost 6 times as large as the send window that the default parameters for TCP provided by ns-2. Figure 3.4 illustrates the difference between the throughputs obtained for different versions of TCP. Linux, with a send window of 22 packets had the lowest of the three throughputs, Solaris, with a send window of 33 packets had slightly better throughput, while the "enhanced" version of TCP with the send-window increased to 128 packets had the best throughput of all.

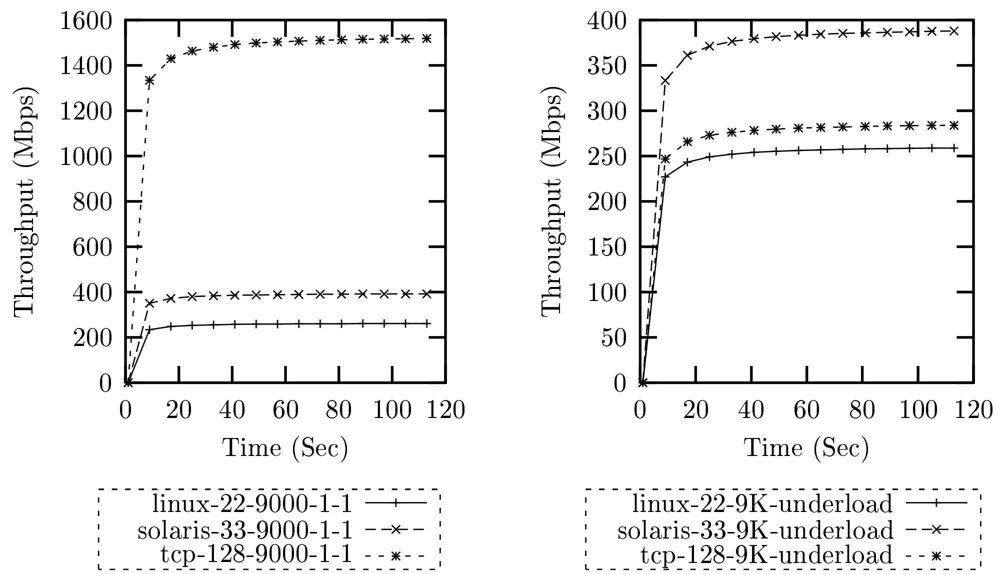


(a.No background traffic) **(b.With background traffic)**

Figure 3.4: TCP throughput for different send-window sizes, with and without background traffic. Packet size remained constant at 1500 Bytes.All three scenarios were tried first with no background traffic and then in presence of background traffic.

We also wanted to see how this large send window performed in the presence of background traffic, so we employed the same method of producing background traffic as Section 3.2.1. Again, we tried this for Linux (window size 22 packets), Solaris (window size 33 packets)

, and "enhanced" TCP (window size 128 packets). The packet sizes were kept contsnt for all at 1500 bytes. The results were quite intersting. Although, we expected that "enhanced" TCP would demonstrate the best performance, since it had the largest send window, in fact, Solaris with a window size of 33 packets performed at par with it. This re-affirmed our belief that TCP's congestion control was too aggressive, and was affecting the throughput negatively. At this point we decided to measure the throughput achieved by increasing the packet size to 9KB. We used the same testbed as described in Section 3.2.1 and tried out for both the window sizes, 22 packets and 33 packets. Finally, we also tried it with our "enhanced" TCP send-window size of 128 packets, and the results, as seen from Figure 3.5, were quite encouraging.



(a.No background traffic) **(b.With background traffic)**

Figure 3.5: TCP throughput for OSD systems with packet size 9KB. Window sizes are 22 pkts (Linux), 33 pkts (Solaris) and 128 pkts (enhanced TCP). All three scenarios were tried first with no background traffic and then in presence of background traffic.

The throughput obtained by the enhaced TCP surpassed that of TCP with default packet size by more than 4 orders of magnitude. This definitely was a big improvement.

For a final experiment, we used the same large packet size of 9KB, and simulated all the three scenarios once again, this time with background traffic. This time, Solaris with a window size of 33 packets actually out-performed "enhanced" TCP with window size 128 packets. This was ample proof of TCP's highly punitive congestion control.

3.2.3 High-Speed TCP and BI-TCP

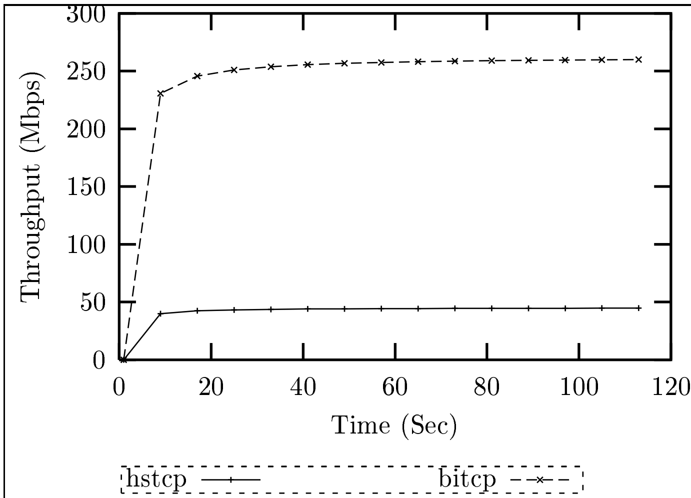


Figure 3.6: TCP throughput for HSTCP and BITCP in the OSD setup. The packet size for both protocols was kept at 1500 bytes

As we mentioned ealier in Chapter 2, several new versions of TCP have been proposed recently for high-speed networks. These re-workings of TCP are trying to create a faster, more productive TCP. Among the many TCP variations that have been proposed so far, we decided to examine the throughput achieved by High-Speed TCP and BI-TCP.

As can be seen from the Figure 3.6, BI-TCP achieves a much higher throughput than High-Speed TCP. This can be mostly attributed to the fact that High Speed TCP is designed to behave like TCP Reno for small values of the *cwnd*. In this scenario since we were only transferring packets of size 1500 bytes, High Speed TCP's aggressive response function never really got used so it lost out to BI-TCP which had slightly better response function. In addition BI-TCP also shows a much smoother curve for its throughput (not very clearly visible in the graph) thus indicating lower packet loss due to congestion. This definitely seems to make BI-TCP as a more viable choice as a possible candidate for the OSD networks.

3.3 Discussion

From the various experiments conducted in the previous section, as well as our survey of existing studies on TCP, we realised that there are several problems with TCP that affect its performance in the OSD scenario.

- **Congestion Control mechanism** The biggest single drawback of TCP in a heavy-load network is its very punishing congestion control policy. The reduction of the transmission window by half every time there is a packet loss is detected, makes the TCP throughput drop drastically. It then only gradually increases the window size, till again the next loss occurs, and we see the window reduced again. The stability of TCP suffers a great deal as a result of the reduction of send window by half.

- **Packet Size** Another area that could be improved upon is the packet size. A typical TCP packet is 1500 bytes in length. However, in a system like the OSDs, small packet size would lead to a high number of packets being transmitted, which in turn is a perfect set-up for congestion. As we mentioned, there is an effort currently underway which proposes increasing the packet sizes to 9KB [14], and as we deduced from our experiments, the larger packet sizes do indeed increase the throughput significantly.

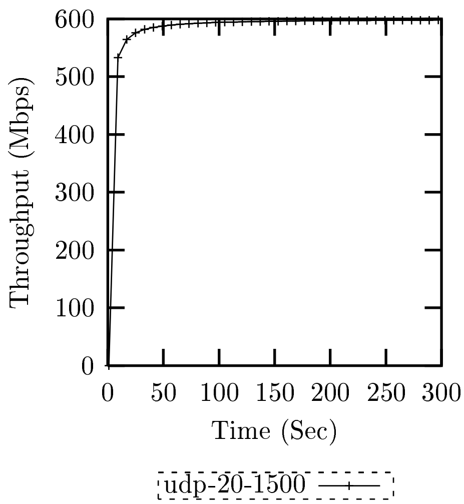
- **Send window** As demonstrated by both BI-TCP as well as High-Speed TCP, a larger send window is definitely more advantageous for better throughput. In TCP's case, the send-window is increased very conservatively in case of successful transmission, but reduced rather drastically in case of congestion. This results in a drop in throughput, and also an unstable performance, both of which are undesirable effects.

- **Complex code** The TCP protocol is rather heavy-weight, especially in view of all the optimisations done on it to make it a successful general purpose transport protocol. While that definitely has contributed to its wide deployment on the Internet, it makes TCP unsuitable for specific applications like OSDs.

3.4 UDP

By now, it was apparent that TCP, as it exists today is not entirely suitable for generating the high throughput for transferring the large amounts of data that are intrinsic to the OSD systems. As an immediate first alternative to TCP, UDP comes to mind because it has no punitive congestion control like TCP, and also it is a light-weight simple protocol which does not carry the overheads TCP does. So we tried out the same experiments with UDP. The testbed, and all the other network parameters remained the same, except for the fact that the load generators were now running UDP with a constant bit rate traffic generator. In the first scenario (Figure 3.7(a)) where there was no extra load on the system, UDP's performance was beyond expectation. It provided a throughput which utilised the bandwidth more than 60 percent. Even with load (Figure 3.7(b)), it provided almost similar throughput. With just this much information, it did seem like UDP was the perfect protocol to be deployed for the OSD networks.

However, UDP also has several major drawbacks. First and foremost, UDP is unreliable. In a network of OSDs, this could be a big concern since any data loss (especially that of scientific data) is unacceptable. Apart from this obvious flaw, UDP also suffers from the lack of any congestion-control and flow control. All these factors combine to make even UDP unsuited



(a. UDP no background traffic) **(b. UDP with background traffic)**

Figure 3.7: UDP throughput for packetsize 1500 and window 20

for the task at hand. However, UDP is also at the same time, a very light-weight and simple protocol, which makes it easy to modify in order to achieve the desired features of reliability, and congestion control. Thus, a new version of UDP could be considered which adds the characteristics like reliability and congestion control to UDP and might come close to a protocol that would suit the OSD system network.

Chapter 4

Reliable-UDP - Policies and Implementation

With the OSD networks' specific requirements, we designed a modified UDP protocol, (hereafter referred to as Reliable-UDP) which ensures a reliable service, along with congestion control, while still providing us with a better throughput than TCP. We wanted to use UDP's light-weight character, and without changing that aspect of it, add congestion-control and reliability to it. Since we wanted to generate a high throughput right from the very start, we dispensed with TCP's slow-start mechanism, and started using the maximum window capacity right away. At the same time, we wanted this throughput to be a "good" one, without too many repeat transmissions, so for this purpose, we decided to use the selective acknowledgement mechanism of TFRC. In the following sections, the Reliable-UDP policies and their implementation is discussed in greater detail.

4.1 Implementing reliability

In Reliable-UDP, we follow the simple policy that any packet sent on the wire should get acknowledged by the receiver. This ensures the sender that the transmission has been successful. To implement this, we maintain a linked list of all packets sent on the sender side. A packet is removed from this list of only if its corresponding ack has been received. The receiver always sends ack for contiguous data so the ack received for a particular packet signifies that all preceding packets have been received by the receiver. Another important thing to note is that sequence number for packets in Reliable-UDP is designated per packet instead of per data byte as in TCP. This allows us to lower the overheads somewhat, besides keeping the granularity a bit larger. Additionally, Reliable-UDP also deploys flow control based on the window size negotiated at connection setup time..

On the receiver end also, there is a linked list of packets present, except these are the packets received from the network. Upon receiving a packet, the receiver checks the sequence number of the packet against the last acknowledgement it has sent. One of several things can happen at this point.

- The packet could be the immediate next in sequence as the one previously acknowledged. In this case, we immediately deliver the packet to the application and send an acknowledgement.

- The packet could be the immediate next in sequence and there are previously queued packets. If this packet was the only missing packet to make some data contiguous, we deliver that to application and send one ack for the highest sequence number.

- The packet could be several sequence numbers ahead of the last acknowledged packet. In this case the packet is received out of order and is queued on the receiver. A immediate selective acknowledgement is sent for this packet indicating to the sender that packets are missing in between. The premise here is that in a LAN environment, packets are rarely delivered out of order. The sender immediately retransmits the missing packets. This policy is more agressive than TCP's fast retransmit but is the key in keeping the throughput going and not burdening the sender and the receiver with huge buffer requirements.

- The packet could be a duplicate of an already queued out-of-order packet. If a selective acknowledgement has already has not been sent, we send one (this is similar to TCP fast retransmit).
- The packet could have a sequence number less than the last acknowledged packet, which could mean that the acknowledgement itself was lost in the network. The receiver just sends the ack again, and ignores the actual packet.

In another version of this acknowledgement policy, the sender and receiver can negotiate at set-up time, a fixed interval for which to identify which sequence numbers have been received. For example, if we decide that this interval be eight packets, then for every eigth packet received, if all data is contiguous, we send a regular ACK else we send a "selective ACK" which lists the packets received and the sender can quickly retransmit the missing packets without waiting for a timeout. To counter the case of a missed "selective ACK", we resend it for each duplicate packet received out of order. Notice that "selective ACK" for Reliable-UDP is very different from TCP SACK which does not allow for identifying multiple missing packets. We need to come up with a better name for this mechanism.

Figure 4.1 displays the initial throughput achieved by Reliable-UDP in its most vanilla flavour. For the purpose of this experiment, the window size of the send window was kept at the ns-simulator default of 20 packets, and the packet size was 1500 bytes. As we can see, reliable UDP reaches a throughput of 40 Mbps quite quickly, and then stabilizes there. This was quite a positive result in view of the configuration, and gave us motivation to try implementing a congestion control policy for Reliable-UDP. Even in the presence of background traffic, Reliable UDP's performance does not deteriorate a lot from its previous performance.

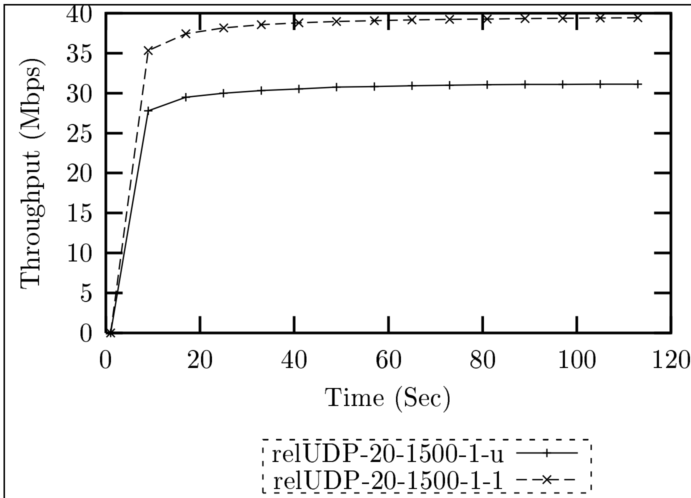


Figure 4.1: RelUDP throughput for window-size:20, packet-size:1500bytes, with and without background traffic

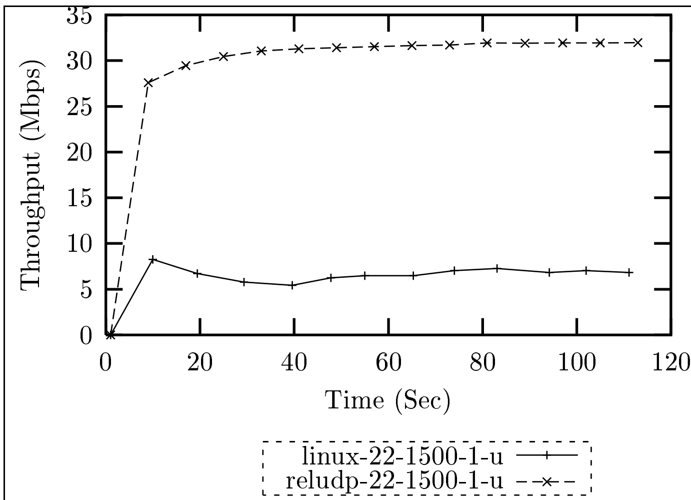


Figure 4.2: Throughput for OSD systems with high background noise. Two transport protocols TCP with Linux's window size and RelUDP with window size of 22 packets are evaluated

4.2 Implementing Congestion Control

It was obvious from the graphs in Fig 3.3 that the biggest drawback with TCP was its multiplicative decrease policy in the event of encountering congestion. We therefore, decided to implement Reliable-UDP's congestion control with TCP-like congestion control poilicy, but without the punitive multiplicative decrease of the send window.

The biggest indication of congestion control is non-receipt of packets. They could be either data packets or acknowledgements - but if a packet is not received before its allocated time, then it is assumed that the link is congested, and congestion control policies are put into effect. Reliable-UDP assumes that it is quite possible that just like the acknowledgement, the actual packet might also get dropped in the network, in which case, it will never get acknowledged. To find out whether this happened, the sender maintains a timer which tracks the estimated RTT of the first packet in the send-queue. This timer tracks whether the acknowledgement has been received in time or not. We follow the TCP's initial RTT estimate algorithm to fire the time-out if the acknowledgement has not been received. If the timer fires, it means that the packet could have been lost in the network. In such a case, the sender retransmits the packet for which the timer has been fired, and resets the timer but unlike TCP, we do not decrease the send window.

From Figure 4.2, we can see that just implementing a congestion control policy without any multiplicative decrease actually benefitted our throughput achieved. The throughput obtained from Reliable-UDP with a smaller send window size was actually almost four times higher than that obtained by TCP (in this case Linux) with a slighly higher window size.

4.3 Reliable-UDP Performance Analysis

In a final comparison, we simulated several scenarios where we plotted the throughputs of different TCP variations alongside that of Reliable-UDP. The first scenario was using ns-2 defaults for window size and packet size for TCP-variations and the same for Reliable-UDP. In the second scenario, we kept the default packet size and window size, but added background traffic to the link under observation. In the third scenario, we used the enhanced TCP parameters of packet size 9KB and window size 128 packets (where applicable). In the fourth and final scenario, we used the same parameters, but added background traffic to the network. The performance of Reliable-UDP as compared to the TCP variations is described in the subsections below

4.3.1 Default parameters, No Background Traffic

In this scenario we simuated the TCP-variations like TCP with default packet size (1500B) and window size(22 packets), High-Speed TCP with packet size 1500 Bytes, BI-TCPwith packet size 1500 bytes, and then also Reliable-UDP with default packet size 1500 bytes and window size 22 packets. The figure 4.3 illustrates the performance of all four. As we can see, the best throughput out of all the protocols was achieved by BI-TCP. In fact, all the other three protocols were almost similar in their performance, while BI-TCP surpassed them by almost 5 times as much. As before, we can guess that this was due to BI-TCP's more aggressive response function in presence of smaller *cwnd*.

4.3.2 Default parameters, with Background Traffic

In this scenario, we again simulated the four protocols under observation. This time, however, we added background traffic, which gave rise to congestion on the network. As we can see from Figure 4.4, High-Speed TCP performance left all the others far behind. In this scenario, Reliable-UDP performed better than both BI-TCP and enhanced TCP, but even so its performance was no comparison against High-Speed TCP. The interesting fact to note here is that BI-TCP performance completely broke down under the congestion. At the same time, however, High-Speed TCP, although performing very well, took awhile to become stable, which indicates

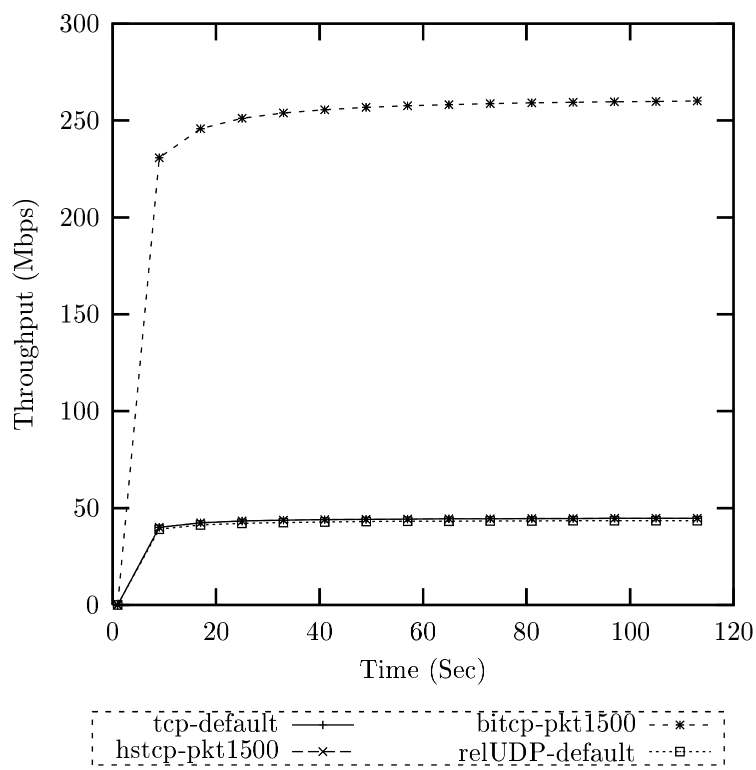


Figure 4.3: Throughput for TCP variations and Reliable-UDP with default packet size and window size, and no background load

possibility of packet loss during congestion.

The conflicting performances of both BI-TCP and High-Speed TCP make it a very difficult to take an absolute decision at this point regarding the more suitable protocol for OSD networks.

4.3.3 Large Packet Size, Large Window size, no background traffic

Since we were having a tough time deciding between High-Speed TCP and BI-TCP, and also since we had already seen that enhanced packet sizes and window sizes were definitely instrumental in producing higher throughputs, we decided to simulate another scenario where we used the same four protocols, but this time with packet size 9KB and window size 128 packets. Here

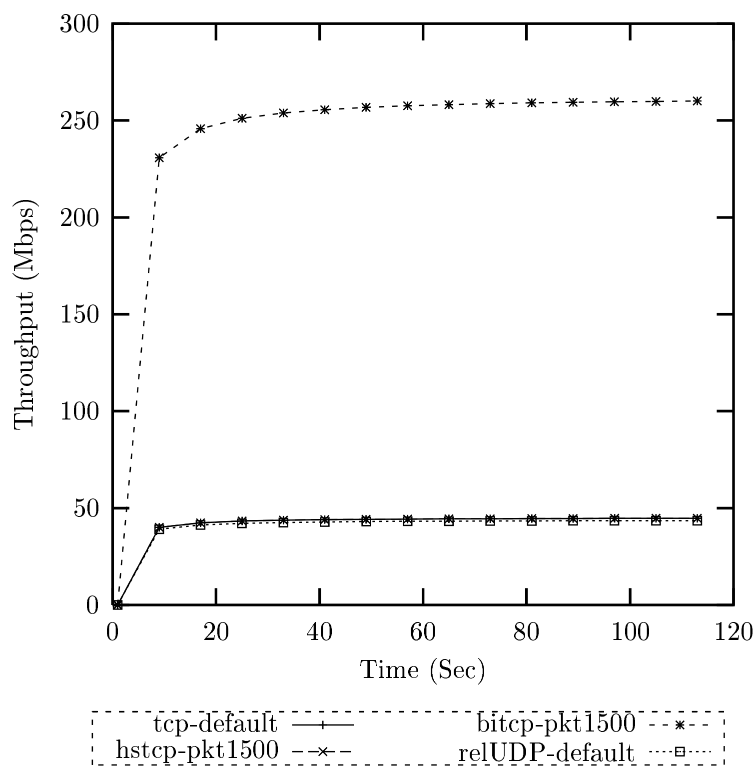


Figure 4.4: Throughput for TCP variations and Reliable-UDP with default packet size and window size, and background load

we should remind the reader that both BI-TCP and High-Speed TCP modulate the window size based on traffic load, so we had given 128 packets as the upper limit they could go to when increasing the window size. The results are depicted in Figure 4.5.

As we can observe, enhanced TCP and BI-TCP both did very well in this scenario. Reliable-UDP was again second in performance, but like before there was significant gap between the performance of the top two and Reliable-UDP.

4.3.4 Large Packet Size, Large Window size, with background traffic

From the previous scenario, we could assume that just enhancing TCP itself might actually give us a solution, but mentioned before, a scenario with no background load was a more ideal

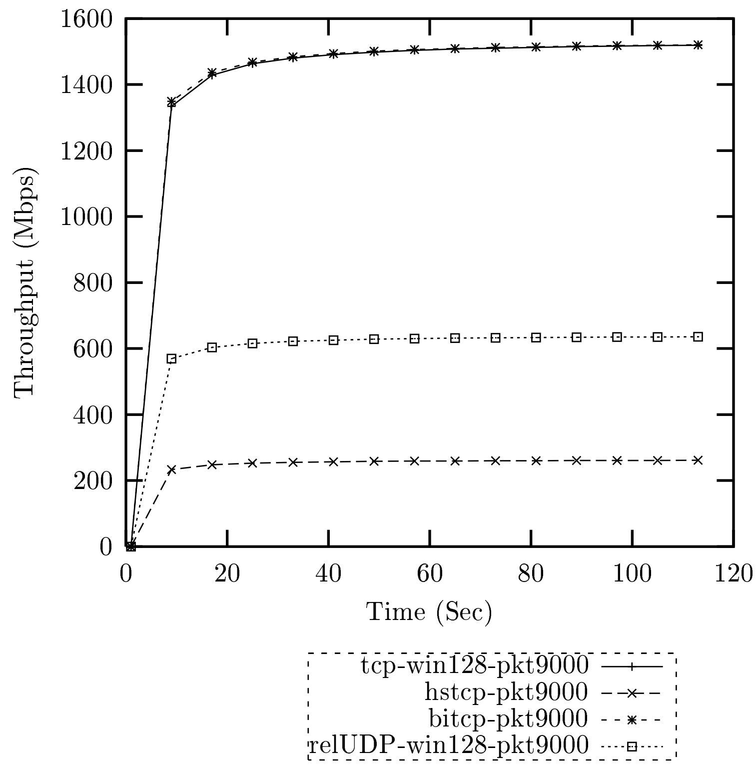


Figure 4.5: Throughput for TCP variations and Reliable-UDP with packet size 9KB and window size 128 packets, and no background load

one, and a less practical one. So, to make it more realistic, we re-ran the scenario, this time adding background traffic to each scenario. The results of these simulations are depicted in the Figure 4.6. As we can see here, the enhanced TCP just breaks down when the background load becomes very heavy. Reliable-UDP, however, gives almost comparable performance to the one it gave without any heavy load. We also noticed that Reliable-UDP, had been providing the most consistent throughputs in all the scenarios, with background traffic, or without. This definitely made it the protocol of choice for the OSD networks.

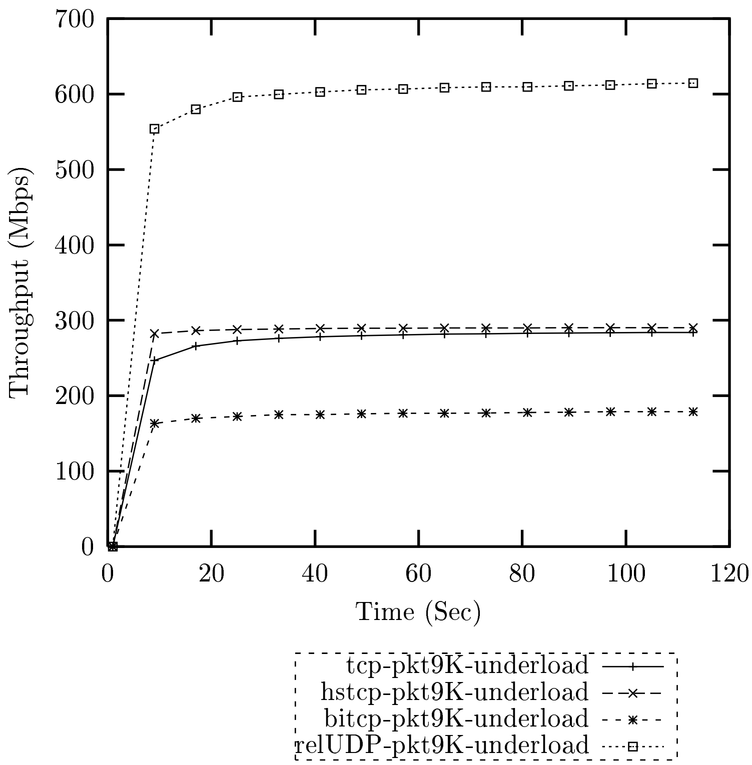


Figure 4.6: Throughput for TCP variations and Reliable-UDP with 9KB packets and 128 packet windows, under background traffic

4.3.5 Other aspects for comparison

Apart from providing just a more stable throughput, Reliable-UDP has several other advantages over enhanced TCP

1. **Congestion Control mechanism** As mentioned before, TCP's biggest drawback was its highly punitive congestion control. In Reliable-UDP, however, this kind of a punishing policy is not applied. We maintain a fixed window size, and while the re-transmission does take away some of our good throughput (original transmissions as opposed to duplicate transmission), we do not see any reduction in transmission rate. This keeps the throughput up, and gives a much better performance.

2. **Acknowledgement Policy** While TCP does have the Fast-Retransmit policy, it still does not suffice in case of packet loss. As we saw, Reliable-UDP adopts this policy and also improvises on it to achieve a better response in case of packet losses. This policy avoids a lot of extra re-transmits and is definitely a value-add when it comes to reducing extra load from the network. Also, Reliable-UDP decreases the buffering requirements on both the sender and receiver since the missing packets are requested more proactively.

3. **Connection setup**A very fundamental improvement comes during connection setup where reliable UDP can negotiate the congestion control, the packet size, the packets to ACK ratio etc. upfront. In comparison, TCP variations like High-Speed TCP and BI-TCP are more generic, and do not allow any negotiations at setup time.

4. **Stability** Reliable-UDP is shown to be stable under no-congestion, as well as under small-to-medium congestion. It starts becoming unstable under high-congestion and packet loss. However, this factor does not appear to be very critical since we are looking at a dedicated network where we will be doing a lot of congestion avoidance to begin with. TCP's default versions (TCP Reno, for example), is shown to be more unstable under a scenario of no congestion by [4]. However, our experiments indicate that it is pretty stable. Under high load, however, it is initially unstable but quickly reaches stability.

5. **Fairness** Apart from stability, fairness is also an important criterion for selecting one protocol over any other. In the OSD scenario we need to pick a protocol that would be fair to other competing flows on the network. However, we can definitely have the concession that the protocol does not have to show fairness to any other kind of protocol, since we are concerned with a closed network here. We therefore restricted our test of Reliable UDP's fairness to that against other competing Reliable UDP flows only. To test

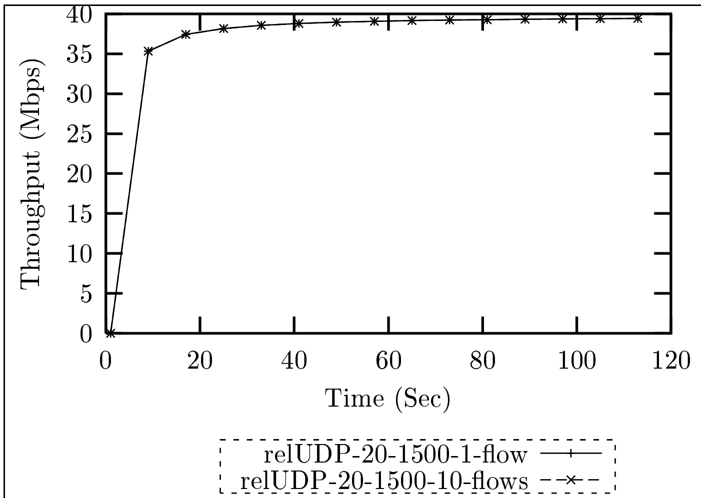


Figure 4.7: Throughput for Reliable-UDP with 1500B packets and 20 packet send window, when 10 flows were competing on the same link.

for fairness, we ran upto 10 Reliable UDP flows on the same network, and observed the performance. All the flows had default window size of 20 packets and default packet size of 1500 bytes, and they were using FTP applications to transfer data. The results, as seen from Figure 4.7, were rather encouraging. The performance of Reliable UDP dropped a little, but remained pretty stable. Based on these initial results, we can safely state that Reliable UDP's performance is both stable as well as fair.

Chapter 5

Possible topologies for the OSD network

One of the areas that we wanted to investigate for networking in OSDs was the kind of topologies that we could apply towards them. For this, we wanted to try out different kinds of level 2 options - buses, repeaters, several buses connected with repeaters etc. However, the network simulator does not understand link level details and therefore any experiments dealing with this layer of the networking stack were not possible. Instead, we tried out different switched ethernet topologies using our basic configuration of 10 clients per OSD.

5.1 Dumb-bell

This was the initial topology which we used throughout the course of this work in order to get the baseline numbers for the OSD network. As can be seen from Figure 5.1, the maximum throughput settles at around 70Mbps, which, given the bandwidth available, is not very efficient. The biggest problem with the dumb-bell topology is, quite obviously, the bottleneck that it creates. In a scenario where all sum of all the bandwidths of the connections feeding main link is less than or equal to the bandwidth of the main link, this topology might barely work. However, in a practical set-up such a topology does not really work. PMTUD problems are also often seen in a dumbbell network topology. If PMTUD fails due to ICMP filtering or host stack

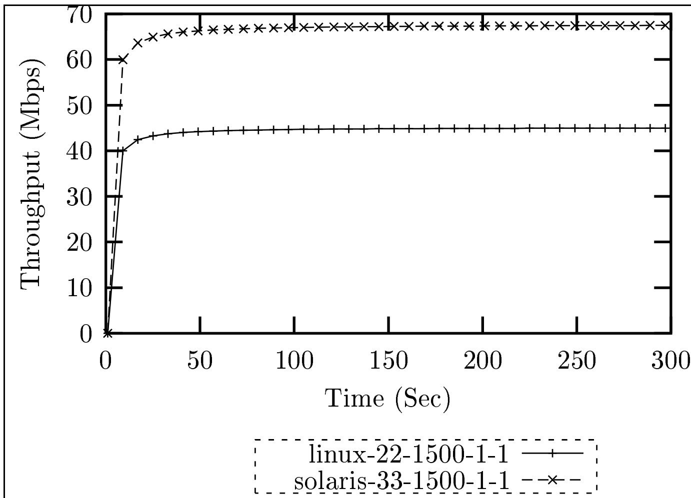


Figure 5.1: TCP throughput for OSD systems in a dumb-bell topology. Two operating systems, Linux and Solaris are evaluated for their TCP throughput with their default send window sizes and packet size of 1500 bytes

problems, then large packets will be unable to traverse the tunnel. For these various reasons, the dumb-bell topology was not considered as a viable option for the OSD scenario.

5.2 STAR

The star topology is pretty simple. The main idea is that a distinguished node is present in the center, and is connected to every other node. For the purpose of our experiment, we made a switch as the distinguished node, and all the clients as well as the OSD were connected to this switch. We ran this experiment with both the TCP window sizes(Linux and Solaris), and as can be seen from Figure 5.4, we obtained a much higher throughput than the dumb-bell topology. The biggest advantage of the Star topology is that it makes troubleshooting really simple since the nodes can be disconnected from the switch one at a time until the problem is isolated. On the flip side, it requires more cable to wire a star. Also, any switch failure can disrupt the entire network. Cost can also be a factor with the Star topology since the cost of a reliable switch can

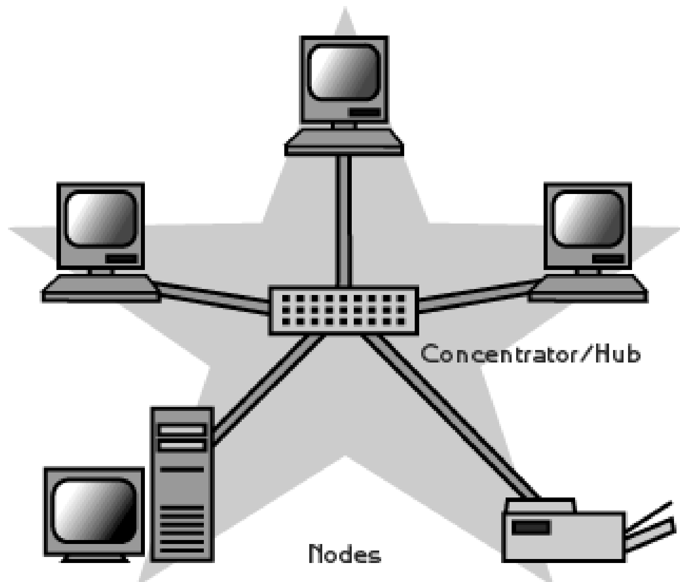


Figure 5.2: A typical STAR topology

go extremely high.

5.3 Cross-Bar

We tried out the standard TCP's preformance in the cross-bar topology where every node can be connected to every other node by one link and there was only one hop. The cross-bar topology was chosen since it offers the facility of being fully connected. In a typical application there will be a column of clients on the left edge and a row of OSDs on the bottom edge. The switch configures itself dynamically to connect a client to an OSD. As long as each client wants to communicate with a different OSD there will be no contention. If two or more clients need to access the same OSD, however, one will be blocked until the switch reconfigures itself. A crossbar has a short diameter - information needs to pass through only one switching element on a path from one edge to another - but poor scalability.

For the purpose of this experiment, we assumed that the link had ethernet character-

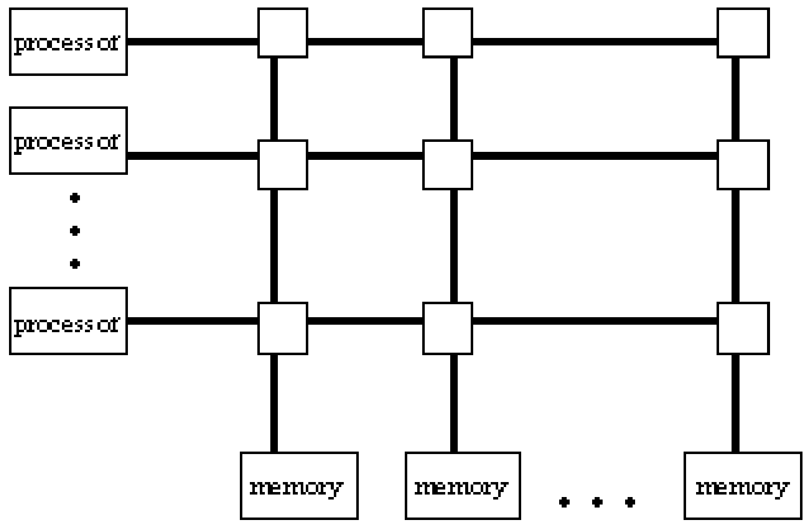


Figure 5.3: A typical Crossbar topology

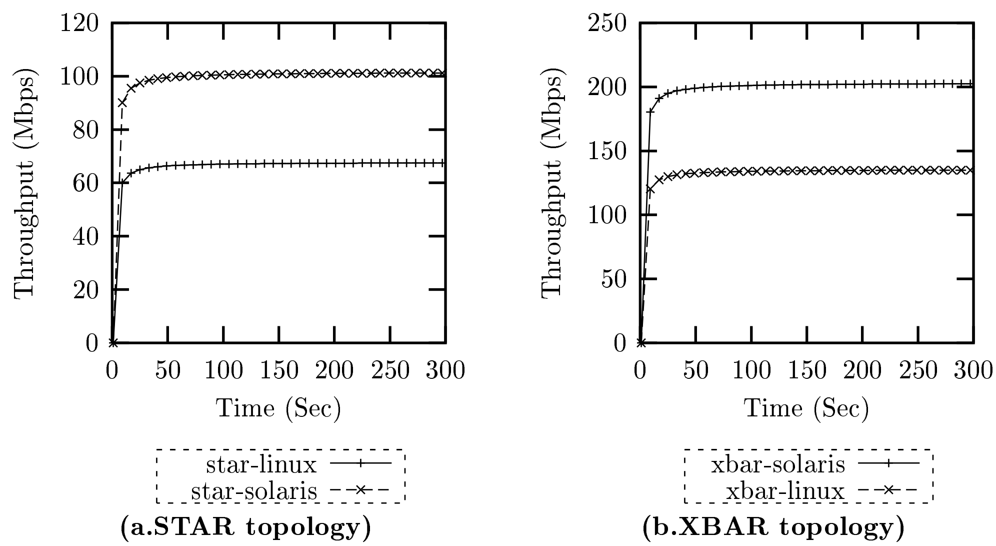


Figure 5.4: TCP throughput Star and Cross-bar topology. Throughput is obtained for Linux and Solaris using their default send window sizes and packet size of 1500 bytes

istics, and had a delay of 1ms. The cross-bar switches themselves were assumed to have zero latency. A typical cross-bar topology looks like the Figure 5.3. From our experiments, we deduced that this particular topology might actually work well for our scenario. As can be seen from Figure 5.4, we can see that its performance was almost twice that of the Star topology. Although in its native form, the cross-bar topology does not scale well, there are numerous variations for it which might work better. Trying out all these variations is in itself a big excercise which we have left for now as it is outside the scope of this work.

5.4 Network Parameters for the OSD network

5.4.1 Simple mathematical calculations

Now that we have narrowed down what kind of protocols and what kind of topologies to deploy for the OSD network, it would also be interesting to see that based on different bandwidths available, what kind of performance can be expected from the network. We did some simple back-of-the-envelope-calculations to find out the otimum number of clients that can be supported for a network which links with bandwidth 1Gbps, and another one which has links with bandwidth 10Gbps. Table 5.1 sums up our calculations. One might notice that some of the entries in the table are marked N/A. These indicate the values where the total throughput calculated was greater than the available bandwidth.

On perusing the table, we can see that we can get almost 100% utilisation of the 10Gb link with various configurations, but it is highly unlikely that all such configurations will be very practical. Table 5.1 summarises the results of these calculations. We can see that without extra background load, the OSD network gives optimum throughput when 5 clients are reading from

Table 5.1: Calculated throughputs for different configurations - Packetsize is constant at 9000 for all scenarios

clients per OSD	Windowsize (in pkts)	1Gb Bandwidth	10 Gb Bandwidth
5	**64**	0.09 Gbps	0.45 Gbps
	128	0.18 Gbps	0.89 Gbps
	256	0.36 Gbps	1.78 Gbps
10	**64**	0.09 Gbps	0.89 Gbps
	128	0.18 Gbps	1.78 Gbps
	256	0.36 Gbps	3.56 Gbps
15	**64**	0.09 Gbps	1.34 Gbps
	128	0.18 Gbps	2.68 Gbps
	256	0.36 Gbps	5.34 Gbps
20	**64**	0.09 Gbps	1.79 Gbps
	128	0.18 Gbps	3.57 Gbps
	256	0.36 Gbps	7.12 Gbps
25	**64**	0.09 Gbps	2.23 Gbps
	128	0.18 Gbps	4.46 Gbps
	256	0.36 Gbps	8.90 Gbps
30	**64**	0.09 Gbps	2.68 Gbps
	128	0.18 Gbps	5.35 Gbps
	256	0.36 Gbps	N/A
35	**64**	0.09 Gbps	3.13 Gbps
	128	0.18 Gbps	7.14 Gbps
	256	0.36 Gbps	N/A
40	**64**	0.09 Gbps	3.57 Gbps
	128	0.18 Gbps	7.14 Gbps
	256	0.36 Gbps	N/A
45	**64**	0.09 Gbps	4.02 Gbps
	128	0.18 Gbps	8.03 Gbps
	256	0.36 Gbps	N/A
50	**64**	0.09 Gbps	4.47 Gbps
	128	0.18 Gbps	8.92 Gbps
	256	0.36 Gbps	N/A
100	**64**	0.09 Gbps	8.93 Gbps
	128	0.18 Gbps	N/A
	256	0.36 Gbps	N/A

the OSD and 5 are writing to it. It is noteworthy here that these simulations adopted the larger packet size and larger window size of the "enhanced" TCP, and the throughputs obtained are based on the same.

5.4.2 Simulating different loads per OSD

Apart from simple calculations, we also simulated the no background load scenario for different permutations of clients per OSD. To find out the throughput achieved in an OSD network, we used the same testbed (Figure 3.1) as Chapter 3.

For the purpose of this experiment, we used the "enhanced" version of TCP as the transport protocol.As per the results obtained from Chapter 3.2.2, using window-size of 128 packets, and packet-size of 9K seemed more reasonable than using ns-2's default parameters. On this, we ran FTP applications to simulate the data transfer. Since the link connecting the clients to their respective switch is 1Gbps in bandwidth, and at any point at least 10 clients are supported by a single OSD, we provisioned a 10Gbps link connecting all the clients to the OSD. We tried various configurations to find out how many simultaneous connections will load the 10Gbps link enough to cause packet loss. In every configuration, we had an equal number of clients reading from and writing to the OSD. So, if there were total 16 active clients, 8 were writing to the OSD and another 8 were being written to by the OSD. Gradually we kept increasing the load as more and more clients wrote to the OSD, and also more and more clients were written to by the OSD. The basic idea behind trying this increasing load on the system was to try and figure out an optimum configuration where the OSDs could service the largest number of clients possible while maintaining a high throughput.

The results, as shown in Figure 5.5 were very encouraging. As we can see, with an increased window size of 128 packets and an increased packet size of 9000bytes, TCP is able to support as many as 12 clients without showing any congestion. The throughput is also around

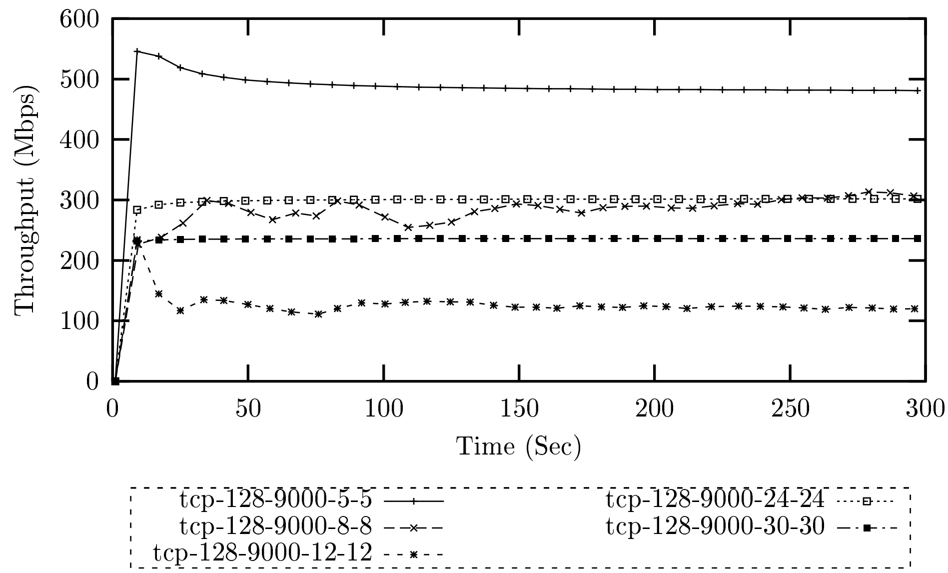


Figure 5.5: TCP throughput for different loads; window-size:128, packet size:9000bytes

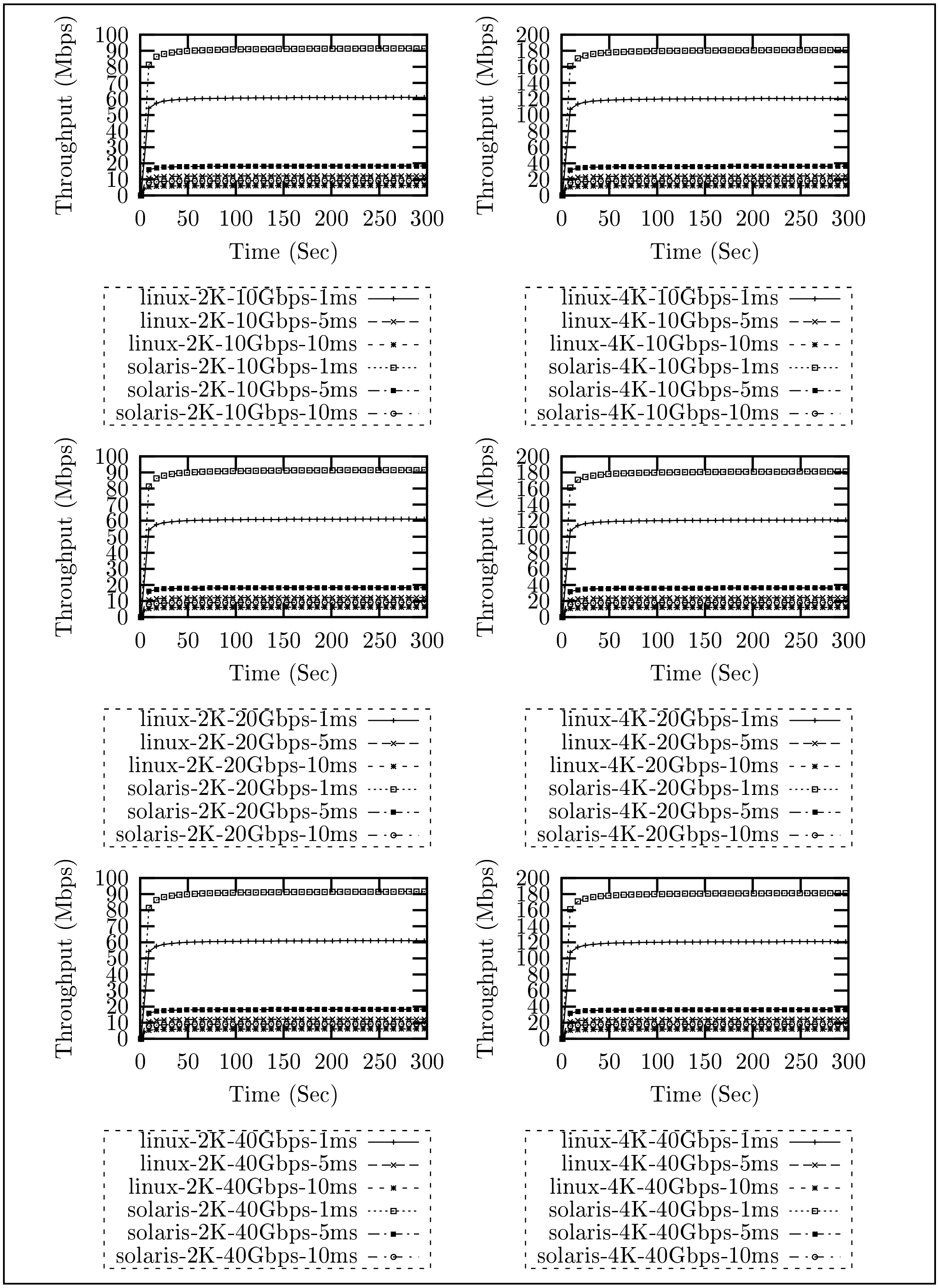
500Mbps which is more than 10 times as high as what we were getting with just 1 client using packets of 1500bytes and window of 20 packets. Given that the client-to-switch link has a bandwidth of 1Gb, a throughput of 500Mbps does look rather promising. And once we consider the fact that there are at least 10 clients per OSD, the amount of throughput for the 10Gb link is at least 5Gbps. Given that we measured the throughput for only one-way, the total traffic can easily be higher than 5Gbps. We could, of course, go even higher than a window size of 128 packets, but then the number of clients which can be supported by the 10Gb link would have to go down. This ratio can be played with till we reach a configuration that seems optimal, but at least for now, it provides us with a good starting point.

5.4.3 Permutations of Network parameters

In order to determine the optimal network configuration for OSDs, it was also important that we try out different permutations of the network parameters like packet size, delay per link etc.

So, we performed several experiments to analyze the same.Figure 5.6 depicts the throughput obtained in each permutation.

Figure 5.6: TCP throughput different permutations of network parameters in the OSD setup


Throughput (Mbps)
Time (Sec)
linux-2K-10Gbps-1ms
linux-2K-10Gbps-5ms
linux-2K-10Gbps-10ms
solaris-2K-10Gbps-1ms
solaris-2K-10Gbps-5ms
solaris-2K-10Gbps-10ms
Throughput (Mbps)
Time (Sec)
linux-4K-10Gbps-1ms
linux-4K-10Gbps-5ms
linux-4K-10Gbps-10ms
solaris-4K-10Gbps-1ms
solaris-4K-10Gbps-5ms
solaris-4K-10Gbps-10ms
Throughput (Mbps)
Time (Sec)
linux-2K-20Gbps-1ms
linux-2K-20Gbps-5ms
linux-2K-20Gbps-10ms
solaris-2K-20Gbps-1ms
solaris-2K-20Gbps-5ms
solaris-2K-20Gbps-10ms
Throughput (Mbps)
Time (Sec)
linux-4K-20Gbps-1ms
linux-4K-20Gbps-5ms
linux-4K-20Gbps-10ms
solaris-4K-20Gbps-1ms
solaris-4K-20Gbps-5ms
solaris-4K-20Gbps-10ms
Throughput (Mbps)
Time (Sec)
linux-2K-40Gbps-1ms
linux-2K-40Gbps-5ms
linux-2K-40Gbps-10ms
solaris-2K-40Gbps-1ms
solaris-2K-40Gbps-5ms
solaris-2K-40Gbps-10ms
Throughput (Mbps)
Time (Sec)
linux-4K-40Gbps-1ms
linux-4K-40Gbps-5ms
linux-4K-40Gbps-10ms
solaris-4K-40Gbps-1ms
solaris-4K-40Gbps-5ms
solaris-4K-40Gbps-10ms

Chapter 6

Conclusions

In this research, we set out to solve the following five main problems:

- Identify the properties of an OSD network
- Analyze the existing transport protocols for their suitability for OSD networks.
- Propose modifications to existing transport protocols if they do not fit OSD network requirements.
- Evaluate some suitable topologies for deploying OSD networks.
- Suggest possible tunable parameters for OSD networks.

As we saw from Chapter 1, the most desirable properties for OSD networks are reliability, minimal congestion, low latency and flow control. Although reliability seems to be the most important requirement, in effect all these are very closely inter-related. Low congestion demands that a high bandwidth connection be used to join the clients and OSD servers. At the same time, reliability also seems to place the demand of large buffers on both the client as well as the server side. We also identified the need for large packet sizes during the course of our investigation. These large packet sizes avoid flooding of the networks by multiple small packets, but at the same time, they re-iterate the need to avoid packet loss since losing each packet would lead to a large penalty in goodput.

At this point it was obvious that any transport protocol that would be used for the OSD networks, needs to be light-weight and provide a high-throughput. UDP was the first protocol that came to mind due to precisely these properties and it did seem to be an excellent choice initially. However, UDP offers neither reliability, nor any form of congestion control. Both these properties were highly crucial to the OSD networks, so UDP was dropped. TCP, on the other hand, was not light-weight, and also offered lower throughputs than UDP, but it did have both reliability and congestion-control. The problem was that these same properies were also making unviable, since they made it very heavy-weight. TCP's congestion-control was especially cumbersome, since it is so punitive.

We also looked at a few variations of TCP, like high-speed TCP and BI-TCP, which are specifically designed with high-speed networks in mind. However, the problem with High-Speed TCP is that it is aimed towards high-speed "long distance" networks. OSD networks, although being high-speed, are designed to be spread out over a LAN, or WAN at the most. Also, high-speed TCP's throughput varies considerably as it tries to adjust it's window-size based on the congestion in the network. BI-TCP, in comparison, has a much smoother throughput graph, but its throughput achieved is much lower than high-speed TCP. We also examined new transport protocols like RDMA and RDDP, but deploying them at this time is not very cost-effective since they require special NICs and are not yet supported fully. DCCP (which is a variation of UDP) could be an interesting candidate due to the flexibility it provides in tuning the congestion-control algorithms, but DCCP does not offer any reliability, thus making it unsuitable.

At this point it was obvious that any transport protocol that we need to use for the OSDs had to combine the salient features of UDP (light-weight and high-throughput) to those of TCP (reliability and and congestion control). We decided to create a new protocol called Reliable-UDP, which offerred these features, while keeping the overheads minimal. Reliable-UDP employed Selective Ack and flow-control thus reducing the need for congestion control. Its performance was quite impressive when deployed in the OSD scenario. In fact, it matched the throughput obtained by high-speed TCP, but without the congestion problems. In addition, Reliable-UDP was also more stable than any of the existing TCP-variations like BI-TCP or High-Speed TCP since it provided almost similar throughput in both scenarios - with background load, and without background load. Although suggesting the creation of a brand-new protocol for the OSD scenario seems a possibility, we would rather not do so at this point. The biggest reason for this is the fact that Reliable-UDP is by no means a complete protocol, and there are possibly quite a few variations of UDP already suggested that might incorporate quite a few principles of Reliable-UDP. Also, while it is easy to suggest a new protocol, its actual implementation in an operating system and its deployment can be a big hurdle to cross. Instead, it would be easier (and more practical) to tweak the existing code for TCP to incorporate the lessons learned from Reliable-UDP, and continue to use TCP itself.

Apart from the most suitable transport protocol, there was also the questions of the most optimum network toplogy as well as the tuning of the various network parameters like delay per llink, packet size and window size etc which would provide us with the highest throughput for the OSD networks. To identify these, we ran more simulations with multiple permutations of the network parameters. Based on these simulations, we concluded that in order for the throughput to be even reasonably acceptable, both the default window size and default packet size for TCP would have to be increased. As it turns out, there is already reserach going on with regards to increasing the packet size to 9K, so hopefully once that happens, OSD networks would definitely benefit from the change. As for increasing the send-window size, that is a tunable that TCP should allow the users to manipulate.

To sum up, for OSD networks, we might actually get away with just tweaking the existing protocols and tunable network parameters. It would be both a time and labour-intensive effort to create a new transport protocol which could cater to OSDs. And even then, the effort of deploying it would far outweigh the benefits obtained by it. Instead, it would be far more practical to modify the existing TCP and make it a little less complex, less punishing towards congestion, and increase the send window as well as the packet size to achieve a reasonable performance for the OSDs.

Appendices

Appendix A

Network Simulator Code for RelUDP

A.1 The Header File

```
/* -*-      Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) Xerox Corporation 1997. All rights reserved.
 *
 * License is granted to copy, to use, and to make and to use derivative
 * works for research and evaluation purposes, provided that Xerox is
 * acknowledged in all documentation pertaining to any such copy or
 * derivative work. Xerox grants no other licenses expressed or implied.
 * The Xerox trade name should not be used in any advertising without its
 * written permission.
 *
 * XEROX CORPORATION MAKES NO REPRESENTATIONS CONCERNING EITHER THE
 * MERCHANTABILITY OF THIS SOFTWARE OR THE SUITABILITY OF THIS SOFTWARE
 * FOR ANY PARTICULAR PURPOSE.  The software is provided "as is" without
 * express or implied warranty of any kind.
 *
 * These notices must be retained in any copies of any part of this
 * software.
 *
 */

#ifndef ns_reludp_h
#define ns_reludp_h

#include "agent.h"
#include "trafgen.h"
#include "packet.h"
#include "rtp.h"

//"rtp timestamp" needs the samplerate
#define SAMPLERATE 8000
#define RTP_M 0x0080 // marker for significant events

#define TH_SYN 0x0001
```

```
#define TH_FIN 0x0002
#define TH_ACK 0x0004
#define TH_TRN 0x0008
#define     TH_SACK 0x0010
#define NONE 0
#define RELUDP_TIMER_RTX 0


class relUdpAgent;

class RelUdpTimer : public TimerHandler {
public:
      RelUdpTimer(relUdpAgent *a) : TimerHandler() { a_ = a; }
protected:
      virtual void expire(Event *e);
      relUdpAgent *a_;
};

class QueueNode {
public:
      QueueNode();
      int seqNum;
      Packet *pkt;
      int rexmit;
      double sendTime;
      double timeOut;
      QueueNode* nextNode;
      QueueNode* prevNode;
};

class QueueHead {
public:
      QueueHead();
      void enqueue(QueueNode*);
      QueueNode* dequeue();
      int lastAckRcvd;
      int lastAckSent;
      int numNodes;
      QueueNode* firstNode;
      QueueNode* lastNode;
};

class relUdpAgent : public Agent {
public:
      relUdpAgent();
      relUdpAgent(packet_t);
      virtual void sendmsg(int nbytes, const char *flags = 0)
      {
```

```
            sendmsg(nbytes, NULL, flags);
      }
      virtual void sendmsg(int nbytes, AppData* data, const char *flags = 0);
      virtual void send_max_packets();
      virtual void recv(Packet* pkt, Handler*);
      virtual int command(int argc, const char*const* argv);
      void ackRcvd(int seqnum, Packet *pkt);
      void sackRcvd(int seqnum, Packet *pkt);
      void pktRcvd(hdr_rtp *rh, Packet *pkt);
      void rtt_update(double now, double sendTime);
      void timeout();
      void finish();
protected:
      int seqno_;
      int send_window_;
      int max_xmit_Q_len_;
      int last_ackno_;
      int delayed_ack_;
      int delayed_ack_cnt_;
      int maxseqno_;
      int keepsending_;
      int maxburst_;
      QueueHead sentQueue;
      QueueHead XmitQueue;
      QueueHead recvQueue;
      RelUdpTimer rtx_timer_;
};



#endif
```

A.2 The RelUDP Code

```
/* -*-	Mode:C++; c-basic-offset:4; tab-width:4; indent-tabs-mode:t -*- */
/*
 * Copyright (C) Xerox Corporation 1997. All rights reserved.
*
* License is granted to copy, to use, and to make and to use derivative
* works for research and evaluation purposes, provided that Xerox is
* acknowledged in all documentation pertaining to any such copy or derivative
* work. Xerox grants no other licenses expressed or implied. The Xerox trade
* name should not be used in any advertising without its written permission.
*
* XEROX CORPORATION MAKES NO REPRESENTATIONS CONCERNING EITHER THE
* MERCHANTABILITY OF THIS SOFTWARE OR THE SUITABILITY OF THIS SOFTWARE
* FOR ANY PARTICULAR PURPOSE.  The software is provided "as is" without
* express or implied warranty of any kind.
*
* These notices must be retained in any copies of any part of this software.
*/

#include <iostream.h>
#include "reludp.h"
#include "random.h"
#include "address.h"
#include "ip.h"

double est_rtt_ = 1;

static class relUdpAgentClass : public TclClass {
public:
      relUdpAgentClass() : TclClass("Agent/relUDP") {}
      TclObject* create(int, const char*const*) {
            return (new relUdpAgent());
      }
} class_relUdp_agent;

relUdpAgent::relUdpAgent() : Agent(PT_RELUDP), seqno_(-1), rtx_timer_(this) {
      bind("packetSize_", &size_);
      bind("window_", &send_window_);
      bind("maxSeq_", &maxseqno_);
      bind("burst_", &maxburst_);
      bind("xmitQlength_", &max_xmit_Q_len_);
      keepsending_ = 0;
      last_ackno_ = -1;
      delayed_ack_ = 1;
      delayed_ack_cnt_= 0;
```

```
      if ((send_window_ + maxburst_) > max_xmit_Q_len_)
            printf("relUDP (warning): xmitQlength_ >= (window_ + burst_)\n");
}


relUdpAgent::relUdpAgent(packet_t type) : Agent(type), rtx_timer_(this) {
      bind("packetSize_", &size_);
      bind("window_", &send_window_);
      bind("maxSeq_", &maxseqno_);
      bind("burst_", &maxburst_);
      bind("xmitQlength_", &max_xmit_Q_len_);
      keepsending_ = 0;
      last_ackno_ = -1;
      delayed_ack_ = 1;
      delayed_ack_cnt_= 0;
      if ((send_window_ + maxburst_) > max_xmit_Q_len_)
            printf("relUDP (warning): xmitQlength_ >= (window_ + burst_)\n");
}

QueueHead::QueueHead() {
      // constructor for QueueHead
      lastAckRcvd = -1;
      lastAckSent = -1;
      numNodes = NONE;
      firstNode = NULL;
      lastNode = NULL;
}

QueueNode::QueueNode() {
      // constructor for QueueNode
      seqNum = NONE;
      rexmit = 0;
      sendTime = 0;
      timeOut = 0;
      prevNode = NULL;
      nextNode = NULL;
}

void QueueHead::enqueue(QueueNode* newNode) {
      // adds a node at the end of the Queue
      if (numNodes == NONE) {
            // no nodes in the queue yet
            firstNode = newNode;
            lastNode = newNode;
            newNode->nextNode = NULL;
            newNode->prevNode = NULL;
      } else {
            // at least one node exists
```

```
            newNode->prevNode = lastNode;
            lastNode->nextNode = newNode;
            lastNode = newNode;
      }
      numNodes++;
}

QueueNode* QueueHead::dequeue() {
      // removes one node at a time from the Queue
      // node being removed is at the head of the Queue
      QueueNode *tempNode;
      if (numNodes == NONE) {
            // nothing in the Queue to remove
            return(NULL);
      } else {
            tempNode = firstNode;
            if (tempNode->nextNode != NULL) {
                  firstNode = tempNode->nextNode;
                  firstNode->prevNode = NULL;
            } else {
                  firstNode = NULL;
                  lastNode = NULL;
                  if (numNodes != 1)
                        printf("dequeue: Error numNodes = %d\n", numNodes);
            }
            numNodes--;
            return(tempNode);
      }
}

void relUdpAgent::send_max_packets() {
      int n, seq, xmit_Q_left, unsent, usable, window_left;
      seq = seqno_;
      if (seq == -1)
            seq = 0;

      // Infinite send
      if (seq < maxseqno_) {
            // what remians to be sent
            unsent = maxseqno_ - seq;

            // How many can we send?
            window_left = send_window_ - sentQueue.numNodes;
            xmit_Q_left = max_xmit_Q_len_ - XmitQueue.numNodes;
            usable = xmit_Q_left + window_left;
            n = unsent > usable ? usable : unsent;
            //printf("send_max_packets: sending %d packets\n", n);
            sendmsg(size_ * n, NULL, 0);
```

```
	} else if (sentQueue.lastAckRcvd >= maxseqno_) {
		printf("--> Finished\n");
		finish();
	}
}
```

```
void relUdpAgent::sendmsg(int nbytes, AppData* data, const char* flags) {
	Packet *p;
	int n;
	QueueNode *qnode;
	double local_time = Scheduler::instance().clock();
	if (nbytes == -1) {
		// Infinite send, till we reach max seqno
		keepsending_ = 1;
		send_max_packets();
		return;
	}


	n = nbytes / size_;

	while (n-- > 0) {
		// packet is now ready. Have to decide whether it can be
		// sent on the wire right away, or will be queued in the
		// XmitQueue first
		if (XmitQueue.numNodes) {
			qnode = new QueueNode;
			qnode->seqNum = ++seqno_;
			XmitQueue.enqueue(qnode);
		} else {
			if (sentQueue.numNodes >= send_window_) {
				qnode = new QueueNode;
				qnode->seqNum = ++seqno_;
				XmitQueue.enqueue(qnode);
			} else {
				p = allocpkt();
				qnode = new QueueNode;
				hdr_cmn::access(p)->size() = size_;
				hdr_rtp* rh = hdr_rtp::access(p);
				if (seqno_ == -1)
					rh->flags() |= TH_SYN;
				else
					rh->flags() = TH_TRN;
				rh->seqno() = ++seqno_;
```

```
                hdr_cmn::access(p)->timestamp() =
                      (u_int32_t)(SAMPLERATE*local_time);

                // p->setdata(data);
                // additions for reliable UDP
                // qnode->pkt = p;
                qnode->seqNum = rh->seqno();
                qnode->sendTime = Scheduler::instance().clock();
                qnode->timeOut = qnode->sendTime + (2 * est_rtt_);
                if (!(rtx_timer_.status() == TIMER_PENDING)) {
                      printf("sendmsg: starting timer for %d\n",
                        qnode->seqNum);
                      rtx_timer_.resched(qnode->timeOut -
                        qnode->sendTime);
                }
                sentQueue.enqueue(qnode);
                local_time = Scheduler::instance().clock();
                hdr_cmn::access(p)->timestamp() =
                      (u_int32_t)(SAMPLERATE*local_time);
                send(p, 0);
          }
    }
}

n = nbytes % size_;
if (n > 0) {
    // packet is now ready. Have to decide whether it can be
    // sent on the wire right away, or will be queued in the
    // XmitQueue first
    if (XmitQueue.numNodes) {
          qnode = new QueueNode;
          qnode->seqNum = ++seqno_;
          XmitQueue.enqueue(qnode);
    } else {
          if (sentQueue.numNodes >= send_window_) {
                qnode = new QueueNode;
                qnode->seqNum = ++seqno_;
                XmitQueue.enqueue(qnode);
          } else {
                p = allocpkt();
                qnode = new QueueNode;
                hdr_cmn::access(p)->size() = size_;
                hdr_rtp* rh = hdr_rtp::access(p);
                if (seqno_ == -1)
                      rh->flags() |= TH_SYN;
                else
                      rh->flags() = TH_TRN;
                rh->seqno() = ++seqno_;
```

```
                        hdr_cmn::access(p)->timestamp() =
                              (u_int32_t)(SAMPLERATE*local_time);
                        qnode->seqNum = rh->seqno();
                        qnode->sendTime = Scheduler::instance().clock();
                        qnode->timeOut = qnode->sendTime + (2 * est_rtt_);
                        if (!(rtx_timer_.status() == TIMER_PENDING)) {
                              printf("sendmsg: starting timer for %d\n",
                                qnode->seqNum);
                              rtx_timer_.resched(qnode->timeOut - qnode->sendTime);
                        }
                        sentQueue.enqueue(qnode);
                        local_time = Scheduler::instance().clock();
                        hdr_cmn::access(p)->timestamp() =
                              (u_int32_t)(SAMPLERATE*local_time);
                        send(p, 0);
                  }
            }
      }

      //idle();
}

void relUdpAgent::recv(Packet* pkt, Handler*) {

      Packet *p;
      hdr_rtp *rh = hdr_rtp::access(pkt);

      if (app_ ) {
            // If an application is attached, pass the data to the app
            hdr_cmn *h = hdr_cmn::access(pkt);
            // new additions for reliable UDP
            if (rh->flags() & TH_ACK) {
                  // from receiver - transmission completed
                  // additions for reliable UDP
                  ackRcvd(rh->seqno(), pkt);
                  // end of additions
            } else if (rh->flags() & TH_SACK) {
                  sackRcvd(rh->seqno(), pkt);
            } else {
                  //double local_time = Scheduler::instance().clock();
                  // additions for reliable UDP
                  pktRcvd(rh, pkt);
                  // end of additions for reliable UDP
            }
      } else if (pkt->userdata() && pkt->userdata()->type() == PACKET_DATA) {
            // otherwise if it's just PacketData, pass it to Tcl
            //
            // Note that a Tcl procedure Agent/relUdp recv
```

```
            // {from data} needs to be defined.  For example,
            //
            // Agent/relUdp instproc recv {from data} {puts data}

            PacketData* data = (PacketData*)pkt->userdata();

            // Check if the packet is from the original
            // sender or just an ack from the sink
            // if from original sender, then acknowledge by
            // sending back otherwise, ignore
            if (rh->flags() & TH_ACK) {
                  // from receiver - transmission completed
                  ackRcvd(rh->seqno(), pkt);
            } else if (rh->flags() & TH_SACK) {
                  sackRcvd(rh->seqno(), pkt);
            } else {
                  rh->flags() |= TH_ACK;
                  pktRcvd(rh, pkt);

            }

            // end of new additions for reliable UDP
            //Tcl& tcl = Tcl::instance();
            //tcl.evalf("%s recv %x", name(), rh->flags());
      } else {
            if (rh->flags() & TH_ACK) {
                  ackRcvd(rh->seqno(), pkt);
            }  else if (rh->flags() & TH_SACK) {
                  sackRcvd(rh->seqno(), pkt);
            } else {
                  rh->flags() |= TH_ACK;
                  //double local_time = Scheduler::instance().clock();
                  pktRcvd(rh, pkt);
            }

      }
}


int relUdpAgent::command(int argc, const char*const* argv) {
      if (argc == 3) {
            printf("REDUDP: argv[0, 1, 2] = <%s> <%s> <%s>\n",
              argv[0], argv[1], argv[2]);
            if (strcmp(argv[1], "advance") == 0) {
                  //int newseq = atoi(argv[2]);
                  //PacketData* data = new PacketData(newseq);
                  //sendmsg(atoi(argv[2]), data);
                  return (TCL_OK);
```

```
        }
        if (strcmp(argv[1], "advanceby") == 0) {
            int newseq = atoi(argv[2]);
            PacketData* data = new PacketData(newseq);
            sendmsg(atoi(argv[2]), data);
            return (TCL_OK);
        }
    } else if (argc == 4) {
        if (strcmp(argv[1], "send") == 0) {
            PacketData* data = new PacketData(1 + strlen(argv[3]));
            strcpy((char*)data->data(), argv[3]);
            sendmsg(atoi(argv[2]), data);
            return (TCL_OK);
        }
    } else if (argc == 5) {
        if (strcmp(argv[1], "sendmsg") == 0) {
            PacketData* data = new PacketData(1 + strlen(argv[3]));
            strcpy((char*)data->data(), argv[3]);
            sendmsg(atoi(argv[2]), data, argv[4]);
            return (TCL_OK);
        }
    }


    return (Agent::command(argc, argv));
}

void relUdpAgent::ackRcvd(int seqnum, Packet *pkt) {

    // Client sends acks to confirm receipt of packets.
    // Upon receiving an ack, sender will check whether
    // the seqno acked is higher than what has been
    // ack'd previously. If yes, then it will remove
    // all packets upto this seqno from the buffer.
    // If not, then just ignore the ack.

    QueueNode *newNode;
    double local_time;
    double now;
    Packet *p;

    if (seqnum > sentQueue.lastAckRcvd) {
        // packet is in the sendQueue, and will have to be
        // removed. Since the acks are for contiguous packets,
        // more than one node of the queue might get removed.
        sentQueue.lastAckRcvd = seqnum;
        now = Scheduler::instance().clock();
        while (sentQueue.numNodes &&
```

```
        sentQueue.firstNode->seqNum <= seqnum) {
      newNode = sentQueue.dequeue();

      /* Update RTT if this is not a re transmitted packet */
      if (!newNode->rexmit)
            rtt_update(now, newNode->sendTime);
}


if (sentQueue.numNodes == 0) {
      if (rtx_timer_.status() == TIMER_PENDING) {
            rtx_timer_.force_cancel();
      }
} else {
      double timeout;
      if (now > sentQueue.firstNode->timeOut) {
            if (rtx_timer_.status() == TIMER_PENDING)
                  rtx_timer_.force_cancel();
            printf("ackRcvd: Already timeouts\n");
            this->timeout();
      } else {
            timeout = sentQueue.firstNode->timeOut - now;
            rtx_timer_.resched(timeout);
      }
}

if (XmitQueue.numNodes > 0) {
      while (XmitQueue.numNodes > 0 &&
        sentQueue.numNodes < send_window_) {
            newNode = XmitQueue.dequeue();
            newNode->sendTime = Scheduler::instance().clock();
            newNode->timeOut = newNode->sendTime + (2 * est_rtt_);
            if (!(rtx_timer_.status() == TIMER_PENDING)) {
                  rtx_timer_.resched(newNode->timeOut -
                    newNode->sendTime);
            }
            sentQueue.enqueue(newNode);
            p = allocpkt();
            hdr_cmn::access(p)->size() = size_;
            hdr_rtp* rh = hdr_rtp::access(p);
            rh->seqno() = newNode->seqNum;
            if (rh->seqno() == -1)
                  rh->flags() |= TH_SYN;
            else
                  rh->flags() = TH_TRN;
            local_time = Scheduler::instance().clock();
            hdr_cmn::access(p)->timestamp() =
                  (u_int32_t)(SAMPLERATE*local_time);
```

```
                        send(p, 0);
                  }
            } else {
                  printf("ackRcvd: Nothing to send\n");
            }
      }

      if (keepsending_) {
                  send_max_packets();
      }
}

void relUdpAgent::pktRcvd(hdr_rtp* rh, Packet *pkt) {

      Packet *p = allocpkt();
      double local_time = Scheduler::instance().clock();
      hdr_rtp* rh_send = hdr_rtp::access(p);
      hdr_cmn* h = hdr_cmn::access(p);
      hdr_ip *oip;
      hdr_ip *nip;

      // first check if there are any packets in the queue
      if (rh->seqno() > recvQueue.lastAckSent + send_window_ + 1) {
            // Out of sliding window. Just drop and no ACK.
            Packet::free(pkt);
            Packet::free(p);
            return;
      }

      if (rh->seqno() <= recvQueue.lastAckSent) {
            // packet has previously been rcvd, ack it
            // then ignore it
            //printf("pktRcvd: seqno %d duplicate  (last %d)\n",
            //rh->seqno(), recvQueue.lastAckSent);
            // get the ip headers
            oip = hdr_ip::access(pkt);
            nip = hdr_ip::access(p);
            // copy the flow id
            nip->flowid() = oip->flowid();
            rh_send->flags() = rh->flags() | TH_ACK;
            rh_send->seqno() = rh->seqno();
            //local_time = Scheduler::instance().clock();
            hdr_cmn::access(p)->timestamp()
                  = (u_int32_t)(SAMPLERATE*local_time);
            send(p, 0);
            Packet::free(pkt);
            return;
      }
```

```
if (rh->seqno() == recvQueue.lastAckSent + 1) {
      // In order receive
      if (recvQueue.numNodes == 0) {
            if (delayed_ack_cnt_ == delayed_ack_) {
                  // And no packets queued. Just ACK
                  // get the ip headers
                  oip = hdr_ip::access(pkt);
                  nip = hdr_ip::access(p);
                  // copy the flow id
                  nip->flowid() = oip->flowid();
                  rh_send->flags() = rh->flags() | TH_ACK;
                  rh_send->seqno() = rh->seqno();
                  //local_time = Scheduler::instance().clock();
                  hdr_cmn::access(p)->timestamp()
                    = (u_int32_t)(SAMPLERATE*local_time);
                  delayed_ack_cnt_ = 0;
                  send(p, 0);
            } else {
                  delayed_ack_cnt_++;
                  Packet::free(p);
            }
            recvQueue.lastAckSent++;
            Packet::free(pkt);
            return;
      } else {
            // In order received but packets already queued.
            // Check and ACK all contigous packets.
            QueueNode *tmpnode = recvQueue.firstNode;
            QueueNode *nextnode;
            int   ackNo = rh->seqno();
            while (tmpnode->seqNum == ackNo + 1) {
                  ackNo++;
                  nextnode = tmpnode->nextNode;
                  tmpnode = recvQueue.dequeue();
                  Packet::free(tmpnode->pkt);
                  delete tmpnode;
                  tmpnode = nextnode;
                  if (tmpnode == NULL)
                        break;
            }
            // get the ip headers
            oip = hdr_ip::access(pkt);
            nip = hdr_ip::access(p);
            // copy the flow id
            nip->flowid() = oip->flowid();

            rh_send->flags() = rh->flags() | TH_ACK;
```

```
            rh_send->seqno() = ackNo;
            recvQueue.lastAckSent = ackNo;
            hdr_cmn::access(p)->timestamp()
                  = (u_int32_t)(SAMPLERATE*local_time);
            send(p, 0);
            Packet::free(pkt);
      }
} else {
      // packet has been rcvd out-of-order, find
      // where it will go in the queue and then
      // insert it
      QueueNode *newnode = new QueueNode();
      newnode->seqNum = rh->seqno();
      newnode->pkt = pkt;
      QueueNode *tmpnode = recvQueue.firstNode;
      if (recvQueue.numNodes == 0) {
            recvQueue.enqueue(newnode);
            oip = hdr_ip::access(pkt);
            nip = hdr_ip::access(p);
            // copy the flow id
            nip->flowid() = oip->flowid();
            rh_send->flags() = TH_SACK;
            rh_send->seqno() = rh->seqno();
            //local_time = Scheduler::instance().clock();
            hdr_cmn::access(p)->timestamp()
                  = (u_int32_t)(SAMPLERATE*local_time);
            send(p, 0);
            return;
      }

      if (newnode->seqNum < tmpnode->seqNum) {
            // This node needs to be inserted at the head
            newnode->nextNode = tmpnode;
            tmpnode->prevNode = newnode;
            newnode->prevNode = NULL;
            recvQueue.firstNode = newnode;
            recvQueue.numNodes++;
            oip = hdr_ip::access(pkt);
            nip = hdr_ip::access(p);
            // copy the flow id
            nip->flowid() = oip->flowid();
            rh_send->flags() = TH_SACK;
            rh_send->seqno() = rh->seqno();
            //local_time = Scheduler::instance().clock();
            hdr_cmn::access(p)->timestamp()
                  = (u_int32_t)(SAMPLERATE*local_time);
            send(p, 0);
            return;
```

```
        }

        while(newnode->seqNum > tmpnode->seqNum) {
              tmpnode = tmpnode->nextNode;
              if (tmpnode == NULL)
                    break;
        }

        if (tmpnode == NULL) {
              recvQueue.enqueue(newnode);
              Packet::free(p);
              return;
        } else if (tmpnode->seqNum == newnode->seqNum) {
              // Duplicate packet. Send a selective acknowledgement
              // so that sender can retransmit the missing pieces.
              oip = hdr_ip::access(pkt);
              nip = hdr_ip::access(p);
              // copy the flow id
              nip->flowid() = oip->flowid();
              rh_send->flags() = TH_SACK;
              rh_send->seqno() = rh->seqno();
              //local_time = Scheduler::instance().clock();
              hdr_cmn::access(p)->timestamp()
                    = (u_int32_t)(SAMPLERATE*local_time);
              send(p, 0);
              Packet::free(pkt);
        } else {
              tmpnode = tmpnode->prevNode;
              newnode->nextNode = tmpnode->nextNode;
              tmpnode->nextNode->prevNode = newnode;

              newnode->prevNode = tmpnode;
              tmpnode->nextNode = newnode;
              // inserted, now increment recvQueue.numNodes
              recvQueue.numNodes++;
              oip = hdr_ip::access(pkt);
              nip = hdr_ip::access(p);
              // copy the flow id
              nip->flowid() = oip->flowid();
              rh_send->flags() = TH_SACK;
              rh_send->seqno() = rh->seqno();
              //local_time = Scheduler::instance().clock();
              hdr_cmn::access(p)->timestamp()
                    = (u_int32_t)(SAMPLERATE*local_time);
              send(p, 0);
        }
  }
}
```

```
void relUdpAgent::sackRcvd(int seqnum, Packet *pkt) {
      QueueNode *newNode;
      double now;
      double local_time;
      Packet *p;
      int i;

      if (sentQueue.numNodes == 0) {
            printf("SackRcvd: No nodes here. Why did the SACK come\n");
            return;
      }

      now = Scheduler::instance().clock();
      newNode = sentQueue.firstNode;
      i = sentQueue.numNodes;
      while (i > 0 && newNode->seqNum < seqnum) {
            p = allocpkt();
            hdr_cmn::access(p)->size() = size_;
            hdr_rtp* rh = hdr_rtp::access(p);
            rh->seqno() = newNode->seqNum;
            rh->flags() = TH_TRN;
            newNode->sendTime = now;
            newNode->timeOut = now + (2 * est_rtt_);
            hdr_cmn::access(p)->timestamp() =
              (u_int32_t)(SAMPLERATE*now);
            send(p, 0);
            i--;
            newNode = newNode->nextNode;
      }
      Packet::free(pkt);
      newNode = sentQueue.firstNode;
      if (!(rtx_timer_.status() == TIMER_PENDING)) {
            rtx_timer_.resched(newNode->timeOut - newNode->sendTime);
      }

}

void relUdpAgent::rtt_update(double now, double sendTime) {

      double sample_rtt;

      sample_rtt = now - sendTime;
      est_rtt_ = (0.6 * est_rtt_) + ((1 - 0.6) * sample_rtt);
}

void relUdpAgent::timeout()
{
```

```
    QueueNode *newNode;
    double now;
    double local_time;
    Packet *p;
    int i;

    if (sentQueue.numNodes == 0) {
        printf("expire: No nodes here. Why did the timer fire\n");
        return;
    }

    now = Scheduler::instance().clock();
    newNode = sentQueue.firstNode;
    i = sentQueue.numNodes;
    while (i > 0 && now >= newNode->timeOut) {
        newNode->rexmit++;
        p = allocpkt();
        hdr_cmn::access(p)->size() = size_;
        hdr_rtp* rh = hdr_rtp::access(p);
        rh->seqno() = newNode->seqNum;
        rh->flags() = TH_TRN;
        newNode->sendTime = now;
        newNode->timeOut = now + (2 * est_rtt_);
        hdr_cmn::access(p)->timestamp() =
          (u_int32_t)(SAMPLERATE*now);
        send(p, 0);
        i--;
        newNode = newNode->nextNode;
    }

    newNode = sentQueue.firstNode;
    if (!(rtx_timer_.status() == TIMER_PENDING)) {
        rtx_timer_.resched(newNode->timeOut - newNode->sendTime);
    } else {
        printf("timeout: timedout but timer running\n");
    }
}

void RelUdpTimer::expire(Event*)
{
    //printf("expire: Timeout\n");
    a_->timeout();
}

void relUdpAgent::finish()
{
    Tcl::instance().evalf("%s done", this->name());
}
```

Appendix B

Some sample Tcl Scripts for NS-2

B.1 Script for generating TCP traffic for Window Size 20 packets and Packet Size 1500 Bytes

```
#For each OBSD server, there are 10 clients. In the overall setup, there
#would be 1-10 MetaData Servers, 10,000-100,000 Clients, and 1000-10,000
#OBSD servers. In this setup, we will tone down the numbers to more
#manageable ones, and will eventually scale them up mathematically to
#find out the corresponding results for the target setup.
set ns [new Simulator]

#Define different colors for data flows
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green
set old_data 0

set nf [open tcp-result.nam w]

$ns namtrace-all $nf


#Open the output files
set obsd2node_tracefd [open tcp-win20-pkt1500-obsd2node.tr w]

#Create the OBSD server node
set obsd [$ns node]

#Create the ten client nodes
for {set i 1} {$i < 11} {incr i} {
    #puts "setting client nodes: i == $i"
    set n($i) [$ns node]
}

#Create the MetaData server node
set m [$ns node]
```

```
#Create the three routers
for {set i 0} {$i < 3} {incr i} {
    set r($i) [$ns node]
}

#Create the four switches
for {set i 1} {$i < 5} {incr i} {
    set s($i) [$ns node]
}

#the links
#connect the routers r1 and r2 with a duplex link with the bandwidth
#10 Gigabit, a propogation delay of 1ms and a DropTail queue.
#Create a similar link between r1 and r0, and also between r2 and r0.
$ns duplex-link $r(1) $r(2) 10Gb 1ms DropTail
$ns duplex-link $r(1) $r(0) 10Gb 1ms DropTail
$ns duplex-link $r(2) $r(0) 10Gb 1ms DropTail

#connect the OBSD server node with router r1
$ns duplex-link $obsd $r(1) 10Gb 1ms DropTail

#connect the MetaData Server with router r0
$ns duplex-link $m $r(0) 10Gb 1ms DropTail

#connect the switches with router
$ns duplex-link $s(1) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(2) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(3) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(4) $r(2) 10Gb 0ms DropTail

#connect clients n1, n2 and n3 with switch s1
$ns duplex-link $n(1) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(2) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(3) $s(1) 10Gb 1ms DropTail

#connect clients n4, n5 and n6 with switch s2
$ns duplex-link $n(4) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(5) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(6) $s(2) 10Gb 1ms DropTail

#connect clients n7, n8 and n9 with switch s3
$ns duplex-link $n(7) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(8) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(9) $s(3) 10Gb 1ms DropTail

#connect client c10 with switch s4
```

```
$ns duplex-link $n(10) $s(4) 10Gb 1ms DropTail

#mark the queues which need to be monitored
#$ns duplex-link-op $r(1) $r(2) queuePos 0.5
$ns duplex-link-op $s(1) $n(1) queuePos 0.5
$ns duplex-link-op $r(1) $obsd queuePos 0.5

#create load generators - all ftp load generators run on OBSD
#with clients as sinks.
set client_tcp(1) [new Agent/TCP]
set client_sink1 [new Agent/TCPSink]
$ns attach-agent $obsd $client_tcp(1)
$ns attach-agent $n(1) $client_sink1
$client_tcp(1) set packetSize_ 1500

$ns connect $client_tcp(1) $client_sink1
set client_ftp(1) [$client_tcp(1) attach-source FTP]
$client_ftp(1) set produce 1000
$client_ftp(1) set producemore 200000

$client_tcp(1) set class_ 1

set client_tcp(10) [new Agent/TCP]
set client_sink10 [new Agent/TCPSink]
$ns attach-agent $n(10) $client_tcp(10)
$ns attach-agent $obsd $client_sink10
$client_tcp(10) set packetSize_ 1500

$ns connect $client_tcp(10) $client_sink10

set client_ftp(10) [new Application/FTP]
$client_ftp(10) attach-agent $client_tcp(10)
$client_ftp(10) set produce 1000
$client_ftp(10) set producemore 200000

$client_tcp(10) set class_ 2


$ns at 0.0 "$ns trace-queue $s(1) $n(1) $obsd2node_tracefd"

#Start the traffic source
$ns at 1.0 "$client_ftp(1) start"
$ns at 1.0 "$client_ftp(10) start"


#Stop the traffic sources
$ns at 300.0 "$client_ftp(1) stop"
$ns at 300.0 "$client_ftp(10) stop"
```

```
#Define a 'finish' procedure
#
#This procedure also calculates the throughput at the bottleneck link in Mbit/s
proc finish {} {
    global ns old_data obsd2node_tracefd
    $ns flush-trace

    #Close the output files
    close $obsd2node_tracefd
    close $node2obsd_tracefd
    close $nf

    exec /usr/local/bin/nam tcp-win20-pkt1500.nam &

    exec awk {
      {
          if ($1=="r" && $5=="tcp") {
            old_data=old_data + $6
            print $2, (old_data*8.0)/($2*1000000)
          }
      }
    } tcp-win20-pkt1500-obsd2node.tr > tcp-win20-pkt1500-obsd2node.xg

    # Call xgraph plotting program to plot throughput vs time
    exec xgraph -x Time(Sec) -y Throughput(Mbps) tcp-win20-pkt1500-obsd2node.xg &
    exit 0
}


#Call the finish procedure after 310 seconds simulation time
$ns at 310.0 "finish"

$ns run
```

B.2 Script for generating TCP traffic for Window Size 20 packets and Packet Size 1500 Bytes - Underload

```
#For each OBSD server, there are 10 clients. In the overall setup, there
#would be 1-10 MetaData Servers, 10,000-100,000 Clients, and 1000-10,000
#OBSD servers. In this setup, we will tone down the numbers to more
#manageable ones, and will eventually scale them up mathematically to
#find out the corresponding results for the target setup.

set ns [new Simulator]

#Define different colors for data flows
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green
set old_data 0

set nf [open tcp-result.nam w]
$ns namtrace-all $nf

#Open the output files
set tracefd [open tcp-win20-pkt1500.tr w]

#Create the OBSD server node
set obsd [$ns node]

#Create the ten client nodes
for {set i 1} {$i < 26} {incr i} {
    set n($i) [$ns node]
}

#Create the MetaData server node
set m [$ns node]

#Create the three routers
for {set i 0} {$i < 3} {incr i} {
    set r($i) [$ns node]
}

#Create the four switches
for {set i 1} {$i < 5} {incr i} {
    set s($i) [$ns node]
}

#the links
#connect the routers r1 and r2 with a duplex link with the bandwidth
#10 Gigabit, a propogation delay of 1ms and a DropTail queue.
#Create a similar link between r1 and r0, and also between r2 and r0.
```

```
$ns duplex-link $r(1) $r(2) 10Gb 1ms DropTail
$ns duplex-link $r(1) $r(0) 10Gb 1ms DropTail
$ns duplex-link $r(2) $r(0) 10Gb 1ms DropTail

#connect the OBSD server node with router r1
$ns duplex-link $obsd $r(1) 10Gb 1ms DropTail

#connect the MetaData Server with router r0
$ns duplex-link $m $r(0) 10Gb 1ms DropTail

#connect the switches with router
$ns duplex-link $s(1) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(2) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(3) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(4) $r(2) 10Gb 0ms DropTail

#connect clients n1, n2 and n3 with switch s1
$ns duplex-link $n(1) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(2) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(3) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(4) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(5) $s(1) 10Gb 1ms DropTail

#connect clients n4, n5 and n6 with switch s2
$ns duplex-link $n(6) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(7) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(8) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(9) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(10) $s(2) 10Gb 1ms DropTail

#connect clients n7, n8 and n9 with switch s3
$ns duplex-link $n(11) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(12) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(13) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(14) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(15) $s(3) 10Gb 1ms DropTail

$ns duplex-link $n(16) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(17) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(18) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(19) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(20) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(21) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(22) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(23) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(24) $s(3) 10Gb 1ms DropTail

#connect client c10 with switch s4
```

```
$ns duplex-link $n(25) $s(4) 10Gb 1ms DropTail

#mark the queues which need to be monitored
$ns duplex-link-op $s(4) $n(25) queuePos 0.5

#create load generators - all ftp load generators run on OBSD
#with clients as sinks.
set client_tcp(25) [new Agent/TCP]
set client_sink(25) [new Agent/TCPSink]
$ns attach-agent $obsd $client_tcp(25)
$ns attach-agent $n(25) $client_sink(25)
$ns connect $client_tcp(25) $client_sink(25)
set client_ftp(25) [$client_tcp(25) attach-source FTP]
$client_ftp(25) set produce 1000
$client_ftp(25) set producemore 200000
$client_tcp(25) set class_ 1

for {set i 1} {$i < 25} {incr i} {
    set client_tcp($i) [new Agent/TCP]
    set client_sink($i) [new Agent/TCPSink]
    $ns attach-agent $obsd $client_tcp($i)
    $ns attach-agent $n($i) $client_sink($i)
    $client_tcp($i) set packetSize_ 9000
    $client_tcp($i) set window_ 128
    $ns connect $client_tcp($i) $client_sink($i)
    set client_ftp($i) [$client_tcp($i) attach-source FTP]
    $client_ftp($i) set produce 1000
    $client_ftp($i) set producemore 200000
    $client_tcp($i) set class_ 1
}

$ns at 0.0 "$ns trace-queue $s(4) $n(25) $tracefd"

#Start the traffic source
for {set i 1} {$i < 26} {incr i} {
    $ns at 1.0 "$client_ftp($i) start"
}

#Stop the traffic sources
for {set i 1} {$i < 26} {incr i} {
    $ns at 300.0 "$client_ftp($i) stop"
}

#Define a 'finish' procedure
#This procedure also calculates the throughput at the bottleneck link in Mbit/s
proc finish {} {
    global ns old_data tracefd
    $ns flush-trace
```

```
    #Close the output files
    close $tracefd

    exec awk {
      {
          if ($1=="r" && $5=="tcp") {
            old_data=old_data + $6
            print $2, (old_data*8.0)/($2*1000000)
          }
      }
    } tcp-win20-pkt1500.tr > tcp-win20-pkt1500.xg

    # Call xgraph plotting program to plot throughput vs time
    exec xgraph -x Time(Sec) -y Throughput(Mbps) tcp-win20-pkt1500.xg &
    exit 0
}

#Call the finish procedure after 21 seconds simulation time
$ns at 310.0 "finish"

$ns run
```

B.3 Script for generating RELUDP traffic for Window Size 22 packets and Packet Size 1500 Bytes

```
#For each OBSD server, there are 10 clients. In the overall setup, there
#would be 1-10 MetaData Servers, 10,000-100,000 Clients, and 1000-10,000
#OBSD servers. In this setup, we will tone down the numbers to more
#manageable ones, and will eventually scale them up mathematically to
#find out the corresponding results for the target setup.

set ns [new Simulator]

#Define different colors for data flows
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green
set old_data 0

set nf [open relUDP-result.nam w]
$ns namtrace-all $nf

#Open the output files
set obsd2node_tracefd [open relUDP-win22-pkt1500.tr w]

#Create the OBSD server node
set obsd [$ns node]

#Create the ten client nodes
for {set i 1} {$i < 11} {incr i} {
    set n($i) [$ns node]
}

#Create the MetaData server node
set m [$ns node]

#Create the three routers
for {set i 0} {$i < 3} {incr i} {
    set r($i) [$ns node]
}

#Create the four switches
for {set i 1} {$i < 5} {incr i} {
    set s($i) [$ns node]
}

#the links
#connect the routers r1 and r2 with a duplex link with the bandwidth
#10 Gigabit, a propogation delay of 1ms and a DropTail queue.
#Create a similar link between r1 and r0, and also between r2 and r0.
```

```
$ns duplex-link $r(1) $r(2) 10Gb 1ms DropTail
$ns duplex-link $r(1) $r(0) 10Gb 1ms DropTail
$ns duplex-link $r(2) $r(0) 10Gb 1ms DropTail

#connect the OBSD server node with router r1
$ns duplex-link $obsd $r(1) 10Gb 1ms DropTail

#connect the MetaData Server with router r0
$ns duplex-link $m $r(0) 10Gb 1ms DropTail

#connect the switches with router
$ns duplex-link $s(1) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(2) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(3) $r(2) 10Gb 0ms DropTail
$ns duplex-link $s(4) $r(2) 10Gb 0ms DropTail

#connect clients n1, n2 and n3 with switch s1
$ns duplex-link $n(1) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(2) $s(1) 10Gb 1ms DropTail
$ns duplex-link $n(3) $s(1) 10Gb 1ms DropTail

#connect clients n4, n5 and n6 with switch s2
$ns duplex-link $n(4) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(5) $s(2) 10Gb 1ms DropTail
$ns duplex-link $n(6) $s(2) 10Gb 1ms DropTail

#connect clients n7, n8 and n9 with switch s3
$ns duplex-link $n(7) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(8) $s(3) 10Gb 1ms DropTail
$ns duplex-link $n(9) $s(3) 10Gb 1ms DropTail

#connect client c10 with switch s4
$ns duplex-link $n(10) $s(4) 10Gb 1ms DropTail

#mark the queues which need to be monitored
$ns duplex-link-op $s(1) $n(1) queuePos 0.5
$ns duplex-link-op $r(1) $obsd queuePos 0.5

#create load generators - all ftp load generators run on OBSD
#with clients as sinks.
set client_tcp(1) [new Agent/relUDP]
set client_sink1 [new Agent/relUDP]
$ns attach-agent $obsd $client_tcp(1)
$ns attach-agent $n(1) $client_sink1
$client_tcp(1) set packetSize_ 1500
$ns connect $client_tcp(1) $client_sink1
set client_ftp(1) [new Application/FTP]
$client_ftp(1) attach-agent $client_tcp(1)
```

```
$client_ftp(1) set produce 1000
$client_ftp(1) set producemore 200000
$client_tcp(1) set class_ 1

$ns at 0.0 "$ns trace-queue $s(1) $n(1) $obsd2node_tracefd"

#Start the traffic source
$ns at 1.0 "$client_ftp(1) start"

#Stop the traffic sources
$ns at 300.0 "$client_ftp(1) stop"

#Define a 'finish' procedure
#This procedure also calculates the throughput at the bottleneck link in Mbit/s
proc finish {} {
    global ns old_data obsd2node_tracefd
    $ns flush-trace
    #Close the output files
    close $obsd2node_tracefd
    close $node2obsd_tracefd
    close $nf

    exec /usr/local/bin/nam relUDP-win22-pkt1500.nam &

    exec awk {
      {
          if ($1=="r" && $5=="reludp") {
            old_data=old_data + $6
            print $2, (old_data*8.0)/($2*1000000)
          }
      }
    } relUDP-win22-pkt1500.tr > relUDP-win22-pkt1500.xg

    # Call xgraph plotting program to plot throughput vs time
    exec xgraph -x Time(Sec) -y Throughput(Mbps) relUDP-win22-pkt1500.xg &
    exit 0
}


#Call the finish procedure after 310 seconds simulation time
$ns at 300.0 "finish"

$ns run
```

Bibliography

[1] M. Allman, Sally Floyd, and C. Partridge. Increasing TCP's initial window. Technical report, IETF, October 2002.

[2] Stephen Bailey and Tom Talpey. The architecture of direct data placement and remote direct memory access in internet protocols. Internet-draft, IETF, July 2004.

[3] S. Brandt, L. Xue, E. Miller, and D. Long. Efficient metadata management in large distributed file systems. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2003.

[4] Hadrien Bullot, R. Les Cottrell, and Richard Hughes-Jones. Evaluation of advanced TCP stacks on fast long-distance production networks. *Journal of Grid Computing*, 1(4):345–359, 2003.

[5] T.S. Chan and I. Gorton. HTPNET: A high performance transport protocol. *Computer Communications Journal, Butterworth-Heinemann*, September 1995.

[6] Ariel Cohen. RDMA offers low overhead, high speed, 2003.

[7] Kevin Fall and Kannan Varadhan. *The ns Manual (formerly ns Notes and Documentation)*. The VINT Project.

[8] S. Floyd. Highspeed TCP for large congestion windows. Internet-draft, IETF, June 2002.

[9] Andy D. Hospodor and Ethen L. Miller. Interconnection architectures for petabyte-scale high-performance storage systems. *21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2004.

[10] Cheng Jin, David X. Wei, and Steven H. Low. FAST TCP for high-speed long-distance networks, June 2003.

[11] Cheng Jin, David X. Wei, and Steven H. Low. TCP FAST: motivation, architecture, algorithms, performance. July 2003.

[12] Eddie Kohler, Mark Handley, and Sally Floyd. Datagram congestion control protocol. Internet-draft, IETF, August 2004.

[13] Darrell D. E. Long, Scott A. Brandt, Ethan L. Miller, Patrick E. Mantey, Alexandre Brandwajn, and Katia Obraczka. Scalable file systems for high performance computing. June 2001.

[14] M. Mathis, J. Heffner, and K. Lahey. Path MTU discovery. Technical report, IETF, 2004.

[15] Craig Partridge. *Gigabit Networks.* Addison-Wesley, 1993.

[16] Larry L. Peterson and Bruce S. Davie. *Computer Networks.* Morgan Kaufmann, 2000.

[17] William Stallings. *High-Speed Networks and Internets - Performance and Quality of Service.* Prentice Hall, 2002.

[18] Cisco Systems. Storage networking 101, Cisco Systems Whitepaper, 2001.

[19] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. OBFS: A file system for object-based storage devices. IEEE, April 2004.

[20] Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce T. McLarty. File system workload analysis for large scale scientific computing applications. IEEE, April 2004.

[21] Ralph O. Weber. Information technology—SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, August 2002.

[22] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control for fast, long distance networks. In *Proceedings of The 23rd Conference of the IEEE Communications Society.* IEEE, March 2004.