A Performance Study of Application Level Acknowledgment and Retransmission Mechanism

Vinod Kannan
Illinois Institute of Technology
Chicago, IL 60616

Phillip M. Dickens
Illinois Institute of Technology
Chicago, IL 60616

William Gropp
Argonne National Laboratory
Argonne, IL 60439

June 13, 2003

Abstract

FOBS is a highly efficient, lightweight application level file transfer protocol utilizing UDP for sending data and TCP for control messages. Since FOBS is a UDP-based mechanism it must provide its own acknowledgment and retransmission mechanism to guarantee reliability. This research studies the impact on performance and data-loss as a function of the techniques used to implement the reliability mechanism. We show that the implementation strategy has a significant impact on the performance and data loss experienced during a large-scale data transfer. We discuss the approaches we have implemented and tested, quantify their impact on performance and show that one implementation technique provides the best performance across all network connections and transfers we have tested on.

Keywords: High Performance Networking, Internetworking, Computer Networks Protocols, Grid Computing

1 Introduction

A significant amount of current research is aimed at the development, implementation and testing of the cutting-edge networking infrastructure of the future (e.g. the Internet2 initiative [6] ,VBNS [8] and the I-WIRE initiative). These high-bandwidth, high-delay networks are capable of delivering data at speeds of 40 gigabits per second and have made possible large-scale distributed applications that were heretofore infeasible.

At the heart of any such computational grid environment (i.e. geographically distributed computational resources connected by very high speed networks) is the ability to transfer very large amounts of data in a highly efficient manner. All of the developing and envisioned advanced distributed applications are predicated upon this fundamental ability. It has been well established that in practice the actual bandwidth achieved by distributed applications executing in a Grid environment (e.g., the Internet2 infrastructure) represents only a very small fraction of the available bandwidth [10, 1].

The reason is that TCP, the data transfer mechanism of choice for widearea data transfers, has been shown to perform very poorly in a high-bandwidth, high-delay network environment [12, 13, 14, 15]. Given the performance problems inherent in the TCP protocol, a significant amount of research is aimed at developing more effective techniques for delivering data across high-performance computational Grids.

To address these issues, we have developed FOBS. FOBS overcomes the inherent limitations of the existing protocols and utilizes bandwidth available efficiently. It combines various techniques to get the best performance across all networks.

2 Related Work

The need to increase the utilization of the emerging high-performance networks is an important area of current research. There have been two approaches to solving the problem. In one, research has focused on improving the performance of the TCP protocol for the new high-bandwidth high-latency research networks. In the other approach, researchers are developing application-level techniques to circumvent the performance problems associated with TCP.

In the former approach, the size of the TCP window is the single most important factor in achieving good performance over high-bandwidth, high-delay networks[4, 19, 2]. Such "fat" pipes are kept full by increasing the TCP window size to at least the product of the bandwidth and the round-trip delay. Thus,

research has focused on automatically tuning the size of the TCP socket buffers at runtime [3]. Commercial TCP implementations have been developed that allow the system administrator to significantly increase the size of the TCP window to achieve better performance [4]. Another area of active research is the use of a selective acknowledgment mechanism [17, 7, 4] rather than the standard cumulative acknowledgment scheme. Here, a selective acknowledgment (SACK) packet that specifies exactly those packets that have been received is sent from the receiver to the sender, allowing the latter to retransmit only those segments that are missing. The Pittsburgh Supercomputing Center is an excellent source of information on the commercial and experimental versions of TCP that support various TCP extensions [7].

At the user level, the most common approach to circumventing the performance flaws in TCP is to allocate multiple TCP streams for a given data flow. This is the approach taken by PSockets [19], the GridFTP protocol [9] (developed by the Globus^{TM}Project [5]) and discussed in [18]. This approach can provide significant performance enhancement because the limitations on TCP window sizes are on a per socket basis, and thus striping the data across multiple sockets provides an aggregate TCP buffer size that is closer to the (ideal size) of the bandwidth times round-trip delay. Moreover, this approach essentially circumvents the congestion control mechanisms of TCP. That is, while some TCP streams may be blocked as a result of the congestion control mechanism, it is likely that some other streams are ready to fire. The larger the number of TCP streams, the lower the probability that all such streams will be blocked resulting in a higher probability that some TCP stream will always be ready to fire.

The most closely related user-level approaches are the Reliable Blast UDP Protocol (RUDP [16]) and SABUL [20]. In RUDP, all of the data is blasted across the network without any communication between the data sender and receiver. A message is sent to the receiver at the end of the transmission of the data packets. Then, after some timeout period, the receiver sends a list of all missing packets to the sender. The data sender then retransmits all of the lost packets. This process is iterated until all of the data has been successfully transferred. RUDP is designed for high-performance quality-of-service (QoS)-enabled networks with a very low probability of packet loss. SABUL employs a single UDP stream for data transmission, and a (single) TCP stream for control information related to the state of the data transfer.

3 FOBS (Fast Object Based System)

The design of the FOBS file transfer protocol has three main components: the sending algorithm, the receiving algorithm, and the acknowledgment scheme. The choice of these three components would depends on factors such as the type of network, latency in the network and contention for the CPU and the network resources.

FOBS is designed for currently available (although non-QoS-enabled) high-performance networks. We have aimed to implement FOBS framework such that it has the ability to act as any of the other types of protocols, if desirable. This approach allows the application to switch between the different congestion control and retransmit algorithms based on which is the most appropriate at any given time in the transfer.

In the initial implementations [11, 10] (byte acknowledgment and bit acknowledgment), there were two UDP and one TCP stream between the sender and receiver 1. Files were broken up into chunks, and an acknowledgment for the packets received was sent for a complete chunk at a specified frequency of a certain number of packets. All the socket connections were unidirectional. The data packets were sent from the sender to receiver on one UDP socket, while the acknowledgment (byte and bit) were sent on another UDP socket from the receiver to the sender. Messages indicating the complete receiving of a chunk were sent on the TCP stream from the receiver to the sender. That design had several drawbacks :

- Overlapping acknowledgments was being sent for the entire chunk. Hence, unnecessarily large acknowledgment packets were sent, thereby limiting the size of each chunk.
- Acknowledgment was sent on UDP. Hence some of the packets were lost, and a lot of duplicates were sent.
- There was no rate control for sending the data packets. This severely affected the performance when the sender was pumping out the data from a fat to a thin pipe (for eg. Sender sending on a GigE interface while receiver received on a Fast Ethernet Interface). Hence the receiving host was overwhelmed as the it was unable to suck in the packets as fast as they were arriving.
- There was no congestion control, leading to massive packet losses.

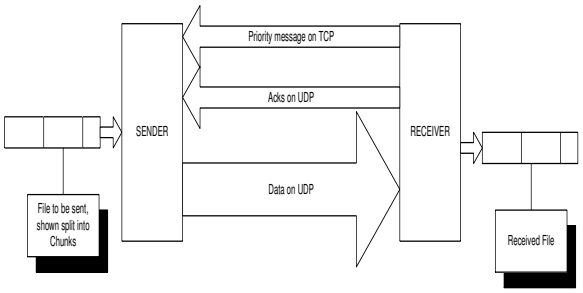

Figure 1: Initial implementation

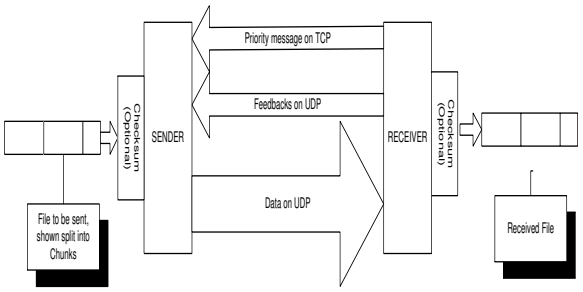

Figure 2: Improved implementation

To address these issues, we redesigned FOBS. In the new FOBS, a file is broken up into chunks, each of which is completely transferred consecutively. The chunk is measured in megabytes. For the purposes of acknowledgment, a chunk is broken up into segments of a fixed number of packets. Thus, an acknowledgment arrives for one segment at a time, thereby accumulating it. For example a file that is 220 MB in size might be broken up into three chunks of 100 MB, 100 MB, and 20 MB each. If the MTU for a link is 1,500 bytes, the first two chunks would comprise of 69906 packets, the last of which hold only 100 bytes. The last chunk would comprise 13,981 1500-byte packets and the last packet 20 bytes.

FOBS has a UDP data stream (data socket) and two TCP (feedback, completed socket) connections between the sender and the receiver 2. All the data is sent from the sender to receiver on the UDP socket. The two TCP sockets are provided for messages of different priority levels. The acknowledgment is sent on the feedback TCP stream, while a higher-priority message (e.g., the message to notifying the sender that a chunk has been completely received) is sent on the other TCP stream. The UDP is unidirectional; that is, messages are passed only from the sender to the receiver. The acknowledgment packets are kept as small as possible.

Any data transferring application or protocol can be said to move between different states. The state of the protocol should reflect what it needs to do to successfully transfer. The state should also be a function of outside variables, like congestion in the network and contention for the CPU and other resources. Thus, the sender and receiver algorithms are implemented as state machines. File I/O is accomplished by a driver that reads a chunk size of the file into the sending buffer and calls the FOBS sender state machine to execute the transfer. The state machines return to the driver when a chunk has been completely transferred. The driver then works on the next chunk. This process is done iteratively until the entire file has been transferred.

3.1 Bit and Byte Acknowledgment

The implementation began as a simple byte acknowledgment scheme.

3.1.1 Sending Algorithm

Here, the sender reads one chunk of the file from the file system into its application memory. It then allocates the necessary buffer, initializes its sockets, and waits for a connection from the receiving application on the TCP sockets. On successfully accepting the connection from the receiver, the sender checks for the lowest packet number that has not been acknowledged and prepares it for transmission by putting the packet number in the header. The sender also copies the payload size of data from the buffer to the packets payload area. This prepared UDP packet is then sent to the receiver, and the sender continues on to the next packet until the chunk boundary has been reached or an acknowledgment received. When an acknowledgment is received it is translated into an character array reflecting which packets have and have not been received. If the sender hits the chunk end, it starts at the beginning of the chunk and repeats the above process until the receiver acknowledges that all the packets in the chunk have been received. The sender then moves on to the next chunk of the file and the process continues as above until all the chunks, and hence the complete file has been transferred.

3.1.2 Receiving Algorithm

At the receiving end, the receiver allocates the memory needed, initializes the necessary sockets and connects with the sender. The receiver looks for data packets on the UDP and if any is available, moves it into the allocated application buffer. This data packet is stripped

of its header to get the packet number. A check is made to see whether this is a duplicate of a previously received packet and whether it is of the currently active chunk. If affirmative, the receiver copies the payload into the chunk receiving buffer with the offset determined by the packet number. Otherwise the packet is discarded. At a predetermined frequency of a certain number of packets, a byte or bit acknowledgment for the entire chunk is sent. When all the packets of a chunk are completely received, the receiver notifies the sender on the TCP control stream. It then writes out the received chunk. This process is iterated until all the chunks that make up the file are received.

3.2 Segmented Bit and Composite Acknowledgment on TCP

The implementation then progressed from the byte/bit acknowledgment of the entire chunk to the segmented bit/composite acknowledgment for reasons previously explained.

3.2.1 Sending Algorithm

At the sender, the driver reads the first chunk size of the file into the application buffer. The sender then calls the sender state machine to transfer the data to the receiver. The state machines switches between various states to complete the transfer. Initially, it is in the initialization or re-initialization states wherein all the memory for the packets and receiving buffers are allocated. The necessary sockets are set up, and the sender waits for connection on the TCP sockets from the receiver. On successfully accepting connections on the TCP sockets, the sender moves into the send state. In this state, the data packet containing the payload and the packet number in the header is prepared. This packet is sent on the UDP. As the sender crosses the segment boundaries, it sends an end-of-segment (EOS) packet, containing the packet type, and number of the segment just completed over the feedback TCP stream. Similarly, when it hits the chunk boundary a DONE packet is sent to indicate to the receiver that it will now start retransmitting unacknowledged packets. The sender also checks for any acknowledgments after sending every data packet. If any are available, it moves into the process-feedback state, where the type of acknowledgment is determined by peeking into the packet header and the size of the acknowledgment is calculated. This size s used to suck in the acknowledgment into a feedback receiving buffer and is processed appropriately depending on the type of acknowledgment. Loss percentage is also calculated in this state. It then moves into the control state. This size is used to suck the acknowledgment into a feedback receiving buffer and is processed appropriately depending on the type of acknowledgment. Loss percentage is also calculated in this state. When the sender gets an acknowledgment from the receiver that all the packets in the chunk have been completely received, it moves into the end state. Control is passed to the driver and the process is repeated until all of the chunks have been completely transferred.

3.2.2 Receiving Algorithm

At the receiver, the driver allocates the chunk size receiving buffer into which the data is to be read in and arranged before being written out to the file system. The receivers state machine is then called. The receiver is initially in the receiver-initialize and re-initialize states where, the sockets are first initialized and the receiver connects up to the sender on the TCP streams. the UDP sockets are then set up. The receiver moves into the Receive state. In this state, the receiver sucks in the data packets as they come in. The packet is stripped of its header and placed in the appropriate place in the receiving buffer depending on the packet number in the header. When the receiver gets an EOS packet, it fires off an acknowledgment for that segment on the feedback socket.When all the packets in a chunk have been completely received, the receiver sends a high priority chunk completed packet on the completed socket. After the receiver has fired off the completed packet to notify the sender of the receiving of the entire chunk, it moves into this state. The receiver moves into this state if an error has occurred in this transfer and exits.

3.3 Acknowledgment Implementation

Several acknowledgment schemes were considered and implemented. Their effectiveness in improving performance was compared.

3.3.1 Byte Acknowledgment on UDP

The byte acknowledgment is the simplest acknowledgment scheme. The acknowledgment packet is an array of characters of size equal to the total number of data packets. Each element in the array represents one data packet, and the character element is set to either "R" or "N" signifying received and not-received respectively. The acknowledgment is sent at a preset frequency that is a certain number of data packets received.

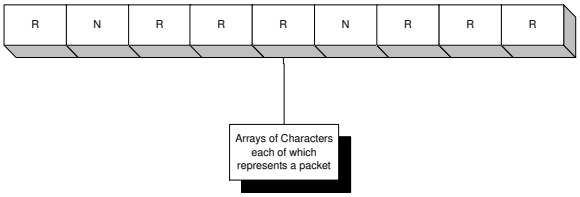

Figure 3: Byte Acknowledgment Packet

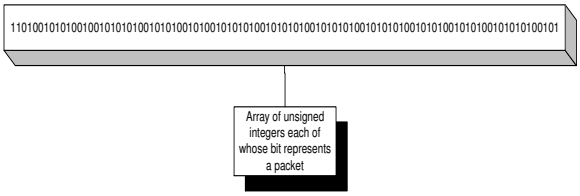

Figure 4: Bit Acknowledgment Packet

This approach, albeit simple, is the least efficient as the size of the acknowledgment packet is equal to a byte for each data packet. For example, a chunk of 10 MB, sent across in 7,153 packets would need an acknowledgment packet of size 7,153 bytes at least. There comes a point when the acknowledgment size is too large to be sent across the UDP as the UDP send buffer overflows. Thus there is a limit on how large a chunk can be, depending on how large an UDP packet can be transmitted. This is an important parameter in performance, as shown in the experimental results and takes on even more significance as the network gets faster and the latency increases because the sender needs to keep working on the new first-time sending of the data packets before starting retransmission.

Also, if the acknowledgment packet is greater than a frame size (1,500 bytes for Ethernet), the chances of its being lost are increased. Since the acknowledgment is overlapping, there is wasted information sent in each acknowledgment packet, thus decreasing overall performance.

Sending the acknowledgment on UDP is faster but is not reliable, and hence the need for the overlapping nature of the acknowledgment. This debilitates performance especially when there is moderate to heavy congestion in the network leading to the loss of a large number of acknowledgments wherein a large number of packets are transmitted over and over, increasing the number of duplicates that arrive at the receiver.

3.3.2 Bit Acknowledgment on UDP

Bit acknowledgment is similar to the byte acknowledgment except that each data packet is now represented in a bit. The acknowledgment packet is an array of unsigned integers. The corresponding bit of the appropriate integer is set to 1 or 0 to indicate that packet has been received or not received, respectively.

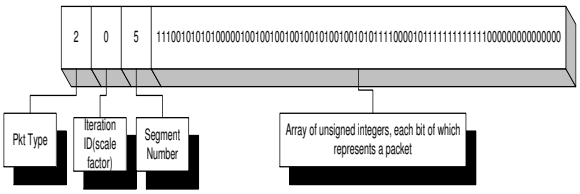

Figure 5: Segmented Bit Acknowledgment

In this case, the size of the acknowledgment packet is one -eight that of the byte acknowledgment. A 10 MB chunk of 7,153 packets could be acknowledged in as little as 895 bytes. Thus, the size of the acknowledgment is decreased, and so chunk can be much larger than in the byte acknowledgment scheme. This as explained earlier, is an important performance parameter.

The disadvantages are due to the overlapping nature of the acknowledgment, as in the byte acknowledgment scheme. Also, the time required to translate the bit acknowledgment by using bit comparisons is larger than translating the simpler byte acknowledgment, thus slowing the sender. This degrades performance as the size of the chunk and thus the bit acknowledgment increases.

As in byte acknowledgment, sending the bit acknowledgment over UDP causes a larger number of duplicates to be received because of the loss of acknowledgment packets.

3.3.3 Segmented Bit Acknowledgment on TCP

In order to fix the problems of the bit and the byte acknowledgments on UDP, it was decided that the acknowledgments had to be transmitted reliably on TCP. This meant that the acknowledgment packets would have to be as small as possible to fit into a single frame (usually 1500 bytes) so that the time taken to transfer the acknowledgment packet is kept to a minimum. Also, since transmitting the acknowledgment on TCP would make sure that each and every packet was received, the need for overlapping the packets was moot.

Specifically, in the segmented approach the chunk is split into segments of a fixed number of packets and an acknowledgment is sent for each segment separately. At the same time, there is an overhead in preparing

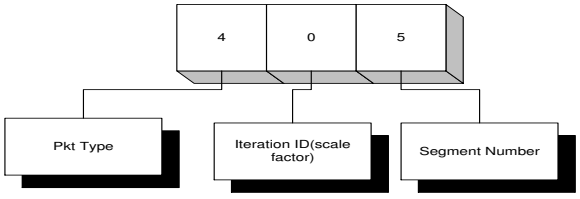

Figure 6: ACK packet type

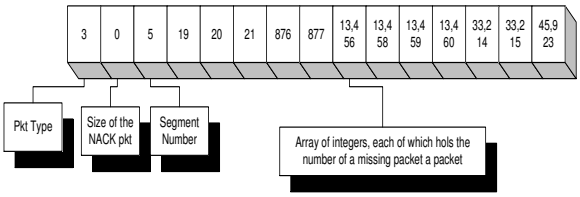

Figure 7: NACK pkt type

and transmitting the acknowledgment packet on TCP, interrupting the receiver and sender from their primary tasks of receiving data packets and sending data packets respectively. In order to adjust for this problem, an optimal frequency (segment size) was picked.

This leads to a significant performance improvement as the time for translation and transmitting are decreased as shown in the next section.

Further efforts were made to decrease the time for translation of the acknowledgment packet at the sender so that it could spend as much of the time possible in sending the data packets. This led to the composite acknowledgment scheme.

3.3.4 Segmented Composite Acknowledgment on TCP

In very high speed networks, a point is reached where in the CPU is unable to pump out packets as fast as the network can handle. Thus all efforts must be made to keep the sender and the receiver working on their primary tasks of sending and receiving the data packets. Acknowledgment preparation, transmission, and processing should take secondary importance if the maximum throughput is to be achieved. Translating the selective bit acknowledgment even on a segment basis significantly slows down the sending speed. This is especially true when the network is clear and there is little or no loss. For example, when the packets in a segment are transmitted for the first time and all of them are received at the receiver with no loss, it sends a bit acknowledgment for that segment in which all the bits are set to 1. The translation of this acknowledgment takes a longer time than, say, a cumulative packet for that segment of a type that specified that all packets in that segment had been received.

Clearly a high performance protocol should have various types of acknowledgments and the best one for a scenario must be used. This led to the implementation of two other versions of the segmented acknowledgments. They are the positive ACK acknowledgment and the negative NACK acknowledgment.

ACK acknowledgment is used when all the packets in a segment are completely received. It is a 12-byte acknowledgment packet that is independent of the segment size. It is interpreted as three integers, as shown in 6. Its fields specify the packet type, the scale factor to authenticate that this acknowledgment is for the present working chunk, and the segment number of the segment that has been completely received. The advantages in using ACK acknowledgment are the lesser time needed to prepare this type of packet, the smaller size of the packet needed to be transmitted and the significantly reduced time in processing this packet at the data sending side.

NACK acknowledgment is a negative acknowledgment scheme involving an array of integers containing the packet type, the size of the packet and the information authenticating this packet to be of the present working chunk (scale factor) followed by the specific packet numbers that have not yet been received. NACK packets can be sent on a per segment basis or for the complete chunk. This acknowledgment comes into play when a small number of packets needed to be retransmitted. The time required to prepare this packet type, however does not warrant the sending of NACK packets on a per segment basis. Only when the total number of data packets not received at the receiver is such that all these missing packet numbers can be fit into a single TCP segment sized packet is the NACK packet used. To further reduce the size of the NACK, it is made of only that size as needed and the size of the packet is written on the packet itself for the data sender to understand where this packets boundaries are. The NACK packet is requested by the sender by setting a field in the DONE packet sent when the chunk boundary is reached. The sender, knowing that the total number of data packets outstanding to be few enough for a NACK packet size, would not send the end of segment messages.

At the data sender side, in the process feedback state, the sender peeks at all the packets received on the feedback TCP stream to determine what packet type is available to be read in, and thus reads the size of the NACK packet.

For example, if the TCP segments size/Ethernet

Ack type	Segmented Bit Feedback	Segmented ACK	NACK
Size bytes	1264	12	¡1500

Table 1: Sample Acknowledgment sizes of the various types

Host	Host type	CPUs	Net. Conn.	Op. Sys.
ANL	Linux Box	2*500 MHz Pentium III	100 Mbps	Linux
NCSA	SGI Origin 2000	64*195 MHz R10000	100 Mbps	IRIX 6.5
CACR	HP V2500	64*400 MHz PA-8500	600 Mbps	HP-UX 11.10

Table 2: Host hardware configurations

frame size is 1470 bytes, the number of packet numbers that can be put into it as integers of 4 bytes each is 367, of which a few would be needed for the packet header information. If the total number of data packets that have not been received at the receiver is 10, then the actual size of the NACK packet would be 12 bytes for the packet header information and 40 bytes for the 10 integers. The size 52 bytes would be written into the size field in the packet header.

NACK helps to improve performance by decreasing the number of EOS packets and therefore the segmented feedback packets to be sent and processed on the sending and receiving sides, respectively. Its impact is felt even more acutely when sending at very high network speeds.

Table 1 showing the sample size of the various packet types, for segments of 10,000 packets and when the MTU is 1500 bytes.

4 Experimental Design

We investigated (reasonably) large-scale memory to memory transfers on two high-performance network connections: one between Argonne National Laboratory (ANL) and the National Center for Supercomputing Applications (NCSA) at the University of Illinois, Urbana-Champaign, and one between ANL and the Center for Advanced Computing Research (CACR) at the California Institute of Technology. Memory-to-memory transfer was chosen so as to not let the file I/O speeds and performance corrupt the contributions and effects of the protocol implementation details. The sites are all connected across the Abilene network. At ANL ,the end host was a Intel Pentium3 node running Red Hat Linux 8.0 . The host used at the CACR was an HP V2500 system (with 64 440 MHz MIPS R10000 processors) running HP-UX 11.10. The end host at the NCSA was a 64 processor SGI Origin2000 running IRIX 6.5.The CACR machine was connected to the Abilene network via a OC12 600 Mbps network card. The NCSA and ANL hosts were connected to Abilene via a 100 Mbps network interface as shown in Table 2.

The round-trip delay between ANL and NCSA was measured (using traceroute) to be on the order of 8 milliseconds, and this we categorize as a short-haul network. The round-trip delay between ANL and CACR was on the order of 58 milliseconds, which we loosely categorize as a long-haul network. The transmitted data size for the experiments was varied between 20 MB to 160 MB. The experiments were performed over multiple iterations, and we tried to keep other parameters like network and CPU contention constant by performing the experiments late night on weekends.

Thus from the hardware configuration, the network link going from Modi4 to Skinner may be characterized as transfer from a thin to a fat pipe and vice versa. Similarly, sending from Modi4 to ccn38 may be characterized as a transfer from a host with significantly more power to a less powerful host when measured in terms of the raw computing capacity and physical memory available. These have an extremely significant effect on the performance of the protocol as explained in the next section.

5 Experimental Results

The various acknowledgment schemes were tested for their performance. Performance is measured as two parameters: packet loss percentage and the throughput achieved. One implementation is deemed to have "performed" better if the throughput achieved is higher. Similarly, it is said to have "performed" worse if the packet loss percentage is higher. Packet loss percentage and throughput achieved are related as increased packet loss increases the amount of retransmission thereby decreasing throughput. The relationship between throughput and packet loss percentage is non-linear.

The effect of the rate and congestion control on the performance was also reflected in the results collected.

As expected, the best performance was best for the rate and congestion controlled segmented bit and composite acknowledgments on TCP. Byte acknowledgment on UDP gives the worst performance. The results are explained below.

5.1 Short-Haul Network:

The link between ANL and NCSA is the short-haul network.

5.1.1 From ANL to NCSA

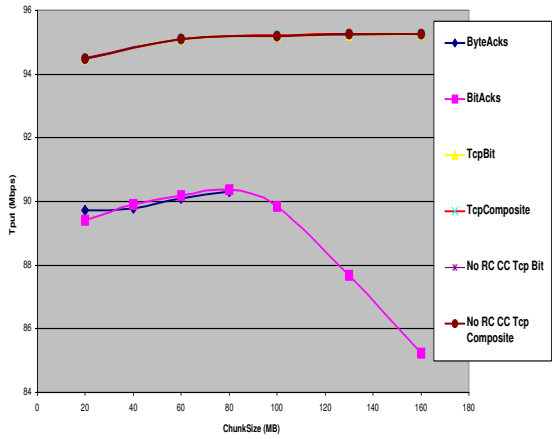

Figure 8: Throughput: ANL to NCSA

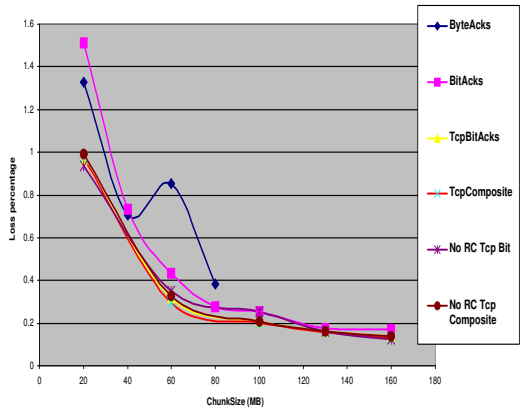

Figure 9: Loss Percentage: ANL to NCSA

As seen in figures 8 and 9, segmented bit and composite acknowledgments on TCP utilize nearly 96% of the available bandwidth with less than 2/10Th's percent of packet loss. This is due to sending from a less powerful host to a significantly more powerful one with enough processing power to handle all the various computational tasks involved in the receiving. Thus, there is no chance for the network buffer at the receiver to overflow and therefore cause packet loss at the receiving host.

The segmented bit and the segmented byte acknowledgments with the rate and congestion control turned off get a similar performance as those with the rate and congestion control since it is a short haul network. There is no packet loss in the network due to the short haul nature and there is no packet loss at the end host, since the network buffer does not overflow because of processing of the received data packets. Rate and congestion control do not kick in, effectively making all the TCP segmented-based protocols the same.

The byte acknowledgment and bit acknowledgment scheme on UDP show similar performance since the network is short haul. They achieve 91% of the available bandwidth. There is little or no loss of the acknowledgment packets and the receiving host is able to process the bit acknowledgments fast enough to negate the effects of processing the bits scheme. But as the chunk size is increased, there is a point after which the byte acknowledgment gets too big to be transferred on UDP, because it is larger than the UDP send buffer. Also, as the acknowledgment size increases, the time spent on processing the acknowledgment increases and the sender spends lesser and lesser time sending data packets. This is seen as a decrease in the loss percentage which is about twice that of the prior two implementations.

Bit acknowledgment can handle still larger sizes; but as the size of the acknowledgment packet increases, the amount of processing involved in translating the packet starts to take a debilitating effect on the performance. Thus the throughput achieved is significantly reduced although the loss percentage is seen to decrease.

TCP performed the poorest achieving only a little more than 10% of the bandwidth.

5.1.2 From NCSA to ANL

The transfer from NCSA to ANL is a short-haul transfer from a host with powerful CPU to a host with a CPU of lesser processing power. This processing power plays a key role in the performance of the protocol. As seen in figures 10 and 11, performance is worse than

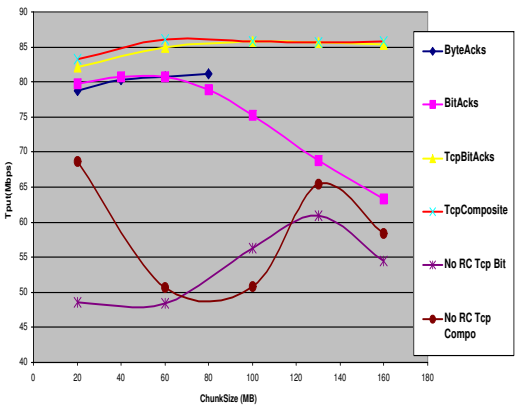

Figure 10: Throughput: NCSA to ANL

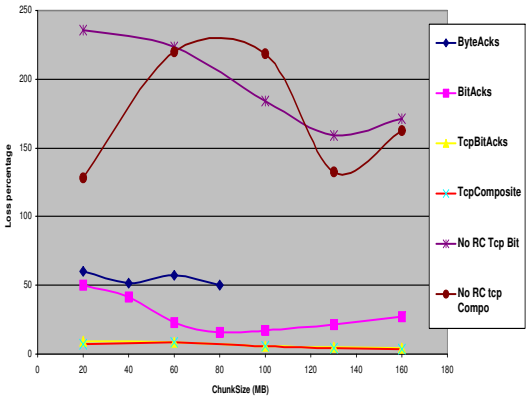

Figure 11: Loss Percentage: NCSA to ANL

when sending in the reverse direction in all the protocol implementations.

TCP segmented implementations again perform better. TCP segmented bit implementation achieves 85.3% of the bandwidth with 4.34% packet loss, whereas TCP segmented composite implementation achieves 85.8% of bandwidth available with 3.9% of packet loss.

Removing the rate control and congestion control, destabilizes the system as the packets sent over TCP take a longer time to be received at the host pumping out packets faster than the receiver can suck them in. Thus, we see that the loss in this case varies and is noticeably higher than all other implementations. As loss percentage increases, the amount of packets needed to be transmitted, the number of acknowledgments sent and processed all increase, thus crippling throughput realized.

Byte acknowledgment on UDP has a higher loss percentage than bit acknowledgment with about 50% of packet loss. This situation is also reflected in the throughput achieved (81% of available).

With the bit acknowledgment on UDP, the packet loss percentage is considerably lower (26%), but this is again due to the sender being slowed down in sending the data packets as it is involved in processing the acknowledgments.

Here again, TCP performed the worst only utilizing around 8.6% of installed bandwidth.

5.2 Long-Haul Network

The link between NCSA and CACR is the long-haul network.

5.2.1 From NCSA to CACR

The transfer from NCSA to CACR is between a thin to a fat pipe, with both sides having enough processing power to negate this parameters effect on performance, as shown in figures 13 and 12. Thus the critical factor determining the performance is the fact that the receiving host was connected to a much bigger pipe or network interface (with a correspondingly higher network buffers) than the sender. There was no loss at the receiving host end.

All TCP segmented acknowledgment implementations, including those without rate and congestion control, perform the same in terms of the throughput realized, utilizing nearly 95% of the bandwidth. But once again, the lack of rate and congestion control, and the fact that the acknowledgments are being sent over TCP over the high latency network, increases the

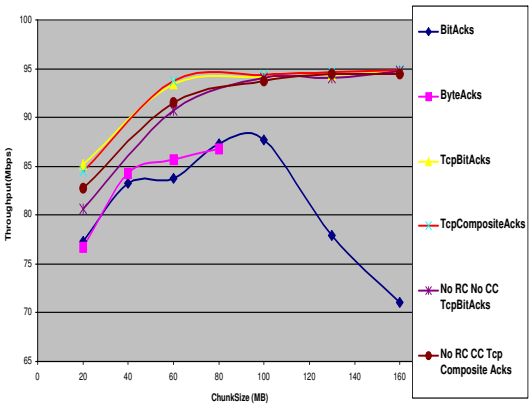

Figure 12: Throughput: NCSA to CACR

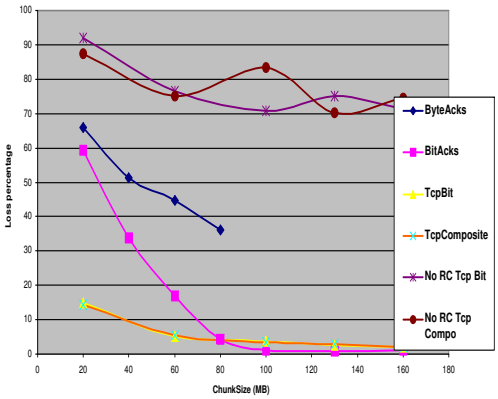

Figure 13: Loss Percentage: NCSA to CACR

time required by the acknowledgments to reach the sender. This situation is reflected in the considerably higher loss percentage for the TCP acknowledgment schemes with no rate or congestion control.

TCP was able to utilize only 2% of the bandwidth.

5.2.2 From CACR to NCSA

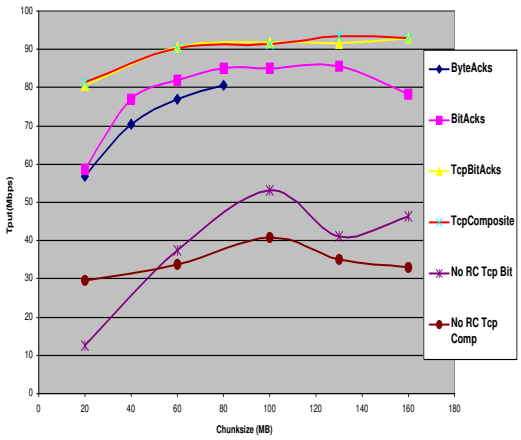

Figure 14: Throughput: CACR to NCSA

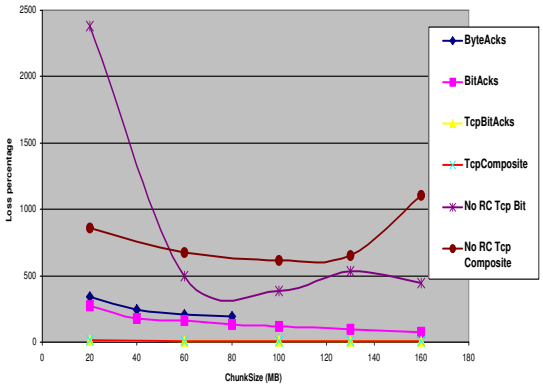

Figure 15: Fig: Loss Percentage: CACR to NCSA

This is the extreme case, in which the sender pumps out data at a much higher rate than the receiver is able to accept it. The transfer is from a fat to a thin pipe. This causes congestion in the network (at the receiving host end). High loss percentages are seen throughout, because the receiver's network buffer is overwhelmed

by getting packet at rates six times faster than its specified network capacity.

Rate and congestion control play the key role in reducing the loss percentage in this transfer case. The TCP segmented bit acknowledgment sees 4.72% packet loss and the TCP composite acknowledgment has 4.71% loss. Throughput realized by both is very nearly 93% of the available.

The same protocols without the rate collar perform significantly worse as the packets sent on TCP are received very slowly. TCP sees the congestion in the network because the packets are pumped out at rates that are not being absorbed at the receiver fast enough. Its window based congestion control kicks in, and even the maximum segment size (MSS) or less sized acknowledgment packets do not reach in time. The number of duplicates increases enormously. Packet losses of more than 1000% are seen in this case with the throughput falling to 50% of the available.

Byte acknowledgment on UDP also has a high loss percentage since some of the acknowledgment packets are now lost because of the congestion in the network. It has a 200% loss and achieves 80.5% of bandwidth available.

Bit acknowledgment on UDP performs better than the byte because of smaller acknowledgment sizes, with 75% of loss acheiving 80% of the throughput.

TCP acheieved a throughput of nearly 15 Mbps.

6 Conclusions and Future research

We have shown that FOBS is a highly efficient high-performance data transfer protocol. It is portable and has been tested across all major platforms. The various mechanisms to ensure reliability of the transfer, have been tested and proved to perform the most efficiently.

Current research is focusing on congestion control mechanisms. We are collecting and analyzing the packet loss patterns to give us a system level understanding of the transfer dynamics. Various congestion control models, including an end-to-end system-aware congestion model, are being developed that can be used as the network conditions dictate.

We are also looking into the scalability of our approach to higher performance network connections. Multi-threaded and multi-streamed versions are being developed and studied to address issues seen in the highest-performance links.

References

[1] Abilene network.
URL: http://www.internet2.edu/abilene.

[2] Automatic tcp window tuning and applications.
URL: http://dast.nlanr.net/Articles/GettingStarted/TCP_window_size.html.

[3] Automatic tcp window tuning and applications.
URL: http://dast.nlanr.net/Projects/Autobuf_v1.0/autotcp.html.

[4] Enabling high performance data transfers on hosts: (notes for users and system administrators).
URL: http://www.psc.edu/networking/perf_tune.html#intro.

[5] Globus^{TM}project.
URL: http://www.globus.org.

[6] Internet2.
URL: http://www.internet2.org.

[7] List of sack implementations.
URL: http://www.psc.edu/networking/all_sack.html.

[8] The very high performance backbone network service.
URL: http//www.vbns.net.

[9] Allcock, B. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnet, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. Preprint ANL/MCS-P871-0201, Argonne National Laboratory, Feb 2001.

[10] P. Dickens and B.Gropp. An evaluation of object-based data transfers over high performance networks. In *11th High Performance Distributed Computing Conference.*, 2002.

[11] P. Dickens, B. Gropp, and P.Woodward. High performance wide area data transfers over high performance networks. In *The 2002 International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2002.

[12] W. Feng and P. Tinnakornsrisuphap. The failure of tcp in high-performance computational grids. In *Super Computing*, 2000.

[13] R. Hobby. Internet2 end-to-end performance initiative (or pat pipes are not enough).
URL: http//www.internet2.org.

[14] B. Irwin and M. Mathis. Web100: Facilitating high-performance network use,white paper for the internet2 end-to-end performance initiative.
URL: http://www.internet2.edu/e2epi/web02/p_web100.shtml.

[15] V. Jacobson, R. Braden, and D. Borman. Rfc 1323: Tcp extensions for high performance.
URL: http://www.ietf.org/rfc/rfc1323.txt?number=1323, May 1992.

[16] J. Leigh, O. Yu, D. Schonfeld, and R. Ansari. Adaptive networking for tele-immersion. In *In: Proceedings of the Immersive Projection Technology/Eurographics Virtual Environments Workshop (IPT/EGVE)*. National Laboratory for Advanced Networking Research, 2001.

[17] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Rfc 2018: Tcp selective acknowledgment options.
URL: http://www.ietf.org/rfc/rfc2018.txt?number=2018.

[18] S. Ostermann, M. Allman, and H. Kruse. An application-level solution to tcp's satellite inefficiencies. In *Workshop on Satellite-based Information Services (WOSBIS)*, November 1996.

[19] H. Sivakumar, S. Bailey, and R. Grossman. Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *In Proceedings of Super Computing 2000 (SC2000)*.

[20] H. Sivakumar, M. Mazzucco, Q. Zhang, and R. Grossman. Simple available bandwidth utilization library for high speed wide area networks. Journal of Supercomputing, 2003 , to appear.