NETBLT: A High Throughput Transport Protocol

David D. Clark
Mark L. Lambert
Lixia Zhang

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

1. Introduction

Bulk data transmission is now finding more and more application in various fields. The major performance concern of a bulk data transfer protocol is high throughput. Theoretically, a packet switched network allows any single user an unlimited share of the network resources. In the absence of other traffic, therefore, a user should be able to transmit data at the raw bandwidth of a network channel. In reality, achievable end-to-end throughputs over high bandwidth channels are often an order of magnitude lower than the provided bandwidth. Experience shows that the throughput is often limited by the transport protocol and its flow control mechanism. It is especially difficult to achieve high throughput, reliable data transmissions across long delay, unreliable network paths.

In this paper we introduce a new transport protocol, NETBLT [2], which was designed for high throughput, bulk data transmission applications. We first analyze the impact of network unreliability and delay on the end-to-end transport protocol; we then summarize previous experience; next we show the design and implementation of NETBLT, followed by our initial experience. Generally speaking, errors and variable delays are two barriers to high performance for all transport protocols. The NETBLT design and experience explores general principles for overcoming these barriers.

2. Impact of Network Unreliability and Delay

2.1. Network Unreliability

If a network were perfectly reliable, an end-to-end transport protocol would have little to do: it would only need to mark the start of a transmission, and then dump all data through the channel. Unfortunately, no real network offers such perfect reliability. The network unreliability manifests itself as packet losses, duplicates, and out-of-order deliveries.

A reliable transport protocol must then detect and recover from all transmission errors. It does so by numbering the data units, and keeping transmission state information at the two communicating ends. The state information keeps track of the status of data under transmission. It also permits recovery from errors whenever they are detected. It is this transmission state that regulates the data flow.

The state at the two ends is constantly synchronized by the arrival of new data and control information (e.g. acknowledgments). The receiving end can easily check and correct duplicate or out-of-order packets. If data or control packets are lost, the most common recovery technique is for the transmitting end to wait for the retransmission timeout and then retry, until a successful retransmission resynchronizes the end state. As will be shown in the next section, however, the detection of lost packets by timers takes a relatively long time, and may easily cause false alarms which in turn trigger superfluous retransmissions.

2.2. Transmission Delay

If there were no communication delays between the transmitter and the receiver, the transmission state at the two ends could be perfectly synchronized, i.e. each end could have a consistent view of the status of every data packet all the time. The transmitter could then retransmit any packet in error immediately after it is detected by the receiver. In this case the throughput would be limited only by the channel bandwidth between the two ends.

In reality, the network round trip delay (RTD) is usually non-negligible and varies randomly. The presence of the RTD causes the control state at the two ends to be out of synchronization: each will have a different view of the status of the data under transmission from time to time, and will not know the other's state changes until some time period later. For instance, after the transmitter sends a packet P, it will not know whether P was successfully received until at least an RTD period; in the meantime, it must keep sending subsequent packets to achieve a high throughput. A transport protocol must tolerate a certain *state out of synchronization* (SOS) between the two ends; the SOS bounds the quantity of data the transmitter is allowed to send before it must stop to resynchronize the state with the receiver.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Nos. N00014-75-C-0661 and N00014-83-K-0125.

© 1988 ACM 0-89791-245-4/88/0001/0353 $1.50

All transport protocols set a limit on the SOS region in order to bound the state information that must be maintained and the system resources that must be allocated for incoming data. Whenever the transmitter reaches the SOS boundary, the transmission stops and waits for the state resynchronization between the two ends. For instance, a TCP transmitter stops at the window boundary and waits for a new acknowledgment before it can continue. This wait for synchronization can often cause loss of performance.

A small SOS region means a simpler protocol design and implementation. The Trivial File Transfer Protocol (TFTP) [3], for example, takes a lock-step approach and synchronizes the state at every packet transmission[1]. Since the end state synchronization always takes at least an RTD, TFTP can send only one data packet each RTD, and thus performs very poorly over long delay links.

In order to keep transmitting while waiting for the end state to be synchronized, the SOS region must be set larger for longer RTD paths, so that there are always data ready to go. How big an SOS region will be sufficient? Theoretically, there does not exist an upper bound that is absolutely sufficient, because if N errors occur, the ends might need a time period of (N * RTD) to recover, where N ranges from 1 to infinity. In practice, we must assume a bound on the number of errors beyond which performance will suffer.

In summary, it is the requirement for reliable data transmission that make the transport protocol maintain a transmission state at the two ends. Communication delays cause each end's state to be out of synchronization with the other's. Finally, this SOS region must be sufficiently large to achieve a high throughput over long RTD paths.

3. Previous Experience

Previous experience with the performance of transport protocols showed two major problems: restrictions in throughput which arise from the use of windows as a flow control mechanism, and the difficulty in handling timers. The problem with windows is that they are both a data flow control and an error recovery mechanism. The problem with timers is estimating an appropriate timer value. These issues are considered in turn.

By definition, the goal of flow control in a transport protocol is to match the data transmission *rate* with the receiver's data consumption *rate*. Windows control the flow of data by bounding the number of data units which the transmitter may send to the receiver. In terms of the state information stored at each end, the window size is the SOS region boundary. As we showed earlier, a large window must be opened to achieve high throughput over long delay channels; however, this does not imply a full window of data should go all at once, but only that buffering for the data is ready. The transmission ought to be evenly distributed over an RTD time period to match the receiver's consumption speed. Unfortunately, windows only convey the former information - *how much* data can be buffered, rather than the latter - *how fast* the transmission should go. Using windows both as the flow control and the SOS bound often leads to a conflict -- the size is either too small to achieve high throughputs over long delay networks, or too large to have any control effect on the instant data rate.

To achieve reliable delivery of the window authorization, the authorization is coupled to the SOS synchronization message, the acknowledgment. Although the transmission state could include the status of every packet within the window region, a window scheme (in most definitions) takes a simplified approach to state synchronization in which the synchronization message is a single number, the number of the largest data item below which all units have arrived at the receiver. This restriction permits a simple, but perhaps inefficient form of synchronization. An acknowledgment with a sequence number N only tells the transmitter that all packets up to N have been successfully received, even in a case when the transmitter has sent N+W packets (where W is the window size) and the receiver has received all but the N+1th packet. This form of acknowledgment is often called a cumulative acknowledgment, as opposed to a selective acknowledgment (as used in NETBLT).

When a cumulative acknowledgment for N is returned, a window authorization is returned as an offset relative to N. That is, an authorization to send W packets means that the next W after N may be sent. But this means that, since a single lost packet prevents the cumulative acknowledgment from being advanced, a lost packet prevents the window from being opened until the error is recovered. An acknowledgment can be returned only after a correct reception of data. The window mechanism, by its nature, ties the flow control and error control together, and therefore becomes vulnerable in the face of data losses. The throughput is controlled by how quickly the recovery can be performed.

Unfortunately, existing mechanisms for error recovery do not operate quickly. When packets are lost, most reliable transport protocols use timers to trigger the transmission state resynchronization. If the RTD were constant, the retransmission timer could detect a loss promptly after an exact RTD time period. In practice, however, the RTD varies. This variation means that the timer must be set to longer than one RTD, otherwise it may cause excessive false alarms. Setting the timer is based on an unknown statistical distribution of the RTD and the exact causes of packet losses are usually unknown to the host (e.g. whether the network has a high-loss channel or whether the net is congested). Finding a good balance between a timer value that is too short and one that is too long can therefore be very difficult.

This problem is an intrinsic limitation of using timers, rather than a result of any specific timer algorithm being used [6]. The performance loss due to timers is particularly high when the round trip time is long (such as across a satellite link, where every loss costs a long wait), or when the channel is noisy and the error rate is high (the transmission will stop and wait too often)

These timer problems interact poorly with windows, since both false alarms and long waits cause window performance degradation. The effect of long timer delays is obvious; a lost packet prevents any acknowledgment, and thus stops data flow until the timer expires and the packet is retransmitted. A short timer has a different effect. The window controls the number of new packets which can be sent; it specifies nothing about how retransmissions are to be sent. Needless retransmission occurring at an unregulated rate can easily congest the network and the receiver. In this respect, windows do not really control the flow, but only control a parameter (the number of packets outstanding) which indirectly relates to flow.

[1] Strictly speaking, TFTP is an application protocol. Because it uses UDP, an unreliable protocol at the transport layer, TFTP faces the same network delay and unreliability problems to ensure the transmission reliability itself.

We draw the following conclusions from the previous experience:

- Window and timers perform poorly in synchronizing the end state. To achieve fast resynchronization of the end state, we need better mechanisms than the cumulative acknowledgment, and we must reduce reliance on timers.
- Flow control must be independent from error control. Mixing the two in one mechanism can only make the flow control vulnerable to transmission errors and delays.
- The goal of flow control is to match the speed at the two ends; the SOS region is a side effect of reliable data transmission. It is a mistake that window mechanism uses the SOS region as the flow control parameter. As a flow control mechanism, windows are too vulnerable to errors; as an SOS mechanism, it does not carry enough information for good performance; being both, it faces conflicts when a large SOS region is needed over a long delay network path while a small window size is desired to restrict the data flow rate.

4. NETBLT Design

4.1. Design Goals

We want a high throughput protocol that is robust in face of a network's long delay and high loss of the network. Seeing the problems with window flow control and timers, we decided that our goal was achievable only by employing a new flow control mechanism. A goal of NETBLT was to check the validity of the rate-based flow control mechanism, and to gain implementation and testing experience with rate control.

4.1.1. Flow Control by Rate

If NETBLT is to provide high throughput for its application, it must be able to transmit constantly, no matter what the status of previously-transmitted data. Provided that there is sufficient buffering in the protocol module, rate control provides this functionality.

Unlike window-based flow control, rate control works independently of the network round-trip delay (an exception to this is of course that changes to the rate take one round-trip delay to take effect). Thus no matter what the network state, the NETBLT sender of data simply transmits at the current rate. Of course if the network is congested, the rate must decrease, and the synchronization of this rate change between sending and receiving NETBLTs takes one RTD.

Rate control is also designed to work independently of error recovery. It places data to be retransmitted in the same queue with new data; all data leave the queue at the current transmission rate. Thus unlike window flow control mechanisms, there is no notion of error recovery working outside of the standard flow-control mechanism, and the load placed on the network does not change when data are retransmitted.

Rate control reduces reliance on timers and timer setting algorithms. Since retransmission is not RTD-based (as it is in TCP), an incorrect RTD estimate does not result in unneeded retransmissions. Retransmission timer estimation is instead based on the current transmission rate, which is fixed within any one RTD and known by both sender and receiver.

Currently most data loss in the network is due to congestion. Rate-based flow control can reduce this congestion in a number of ways. First, since packet retransmissions occur "in-band", they cause no extra load in the network. Second, since retransmission timers are based on the packet rate (or better yet on the packet inter-arrival time) rather than the network RTD, timers can be estimated more accurately, resulting in fewer unnecessary retransmissions. Finally, the rate can be adjusted to reflect the network's current ability to transport data.

4.1.2. Error Recovery/End State Synchronization

NETBLT's design goals include the ability to continuously transmit data in the face of long-delay networks. This means NETBLT connections must be able to maintain a large SOS region, and require efficient mechanisms to quickly synchronize the end state. NETBLT has several ways to speed up the synchronization.

NETBLT uses selective acknowledgment to convey as much information as possible to the sending NETBLT. By providing the status of each packet in the transmission, unneeded retransmissions are avoided. Obviously, the overhead incurred with a packet-for-packet ACKing scheme is too high; NETBLT therefore splits its out-of-synchronization region into "buffers", which become the synchronization point and recovery block. The sending and receiving NETBLTs synchronize state either upon successful transmission of a buffer or upon determination by the receiver that information is missing from a particular buffer. In the first case, a single message ACKs all packets contained in a particular buffer. In the second case, a single message tells the sender of data exactly which packets to retransmit.

Thus, by aggregating information into large units, NETBLT solves the ACK dilemma in window flow control mechanisms: do you incur high overhead by sending a separate ACK for each packet, or do you incur unneeded retransmissions by failing to provide complete information to the sender of data?

NETBLT also makes error recovery more efficient by placing the data retransmission timer at the receiving end. At any given moment, the receiver knows which packets have arrived and which have not. When a timeout occurs, the receiver can eliminate unnecessary retransmissions because it knows exactly which packets need retransmission. In addition, the timer value is easy for the receiver to estimate, since it is based on the transmission rate and the number of packets expected in a particular buffer. Error recovery and state synchronization both occur on the same (receiving) end; it therefore never matters when the receiver sends ACKs to the sender. Of course long waits between a timeout and a subsequent state resynchronization mean lowered performance, but there are never spurious retransmissions because a retransmit timer expires before an ACK is received.

Even with the above ACKing schemes, the minimum time of an error recovery by requesting retransmission remains an RTD period. In order to insure that error recovery and state synchronization be kept as close as possible to a single RTD, control messages containing ACK information must be transmitted with high reliability.

An affordable reliability in control transmissions is particularly necessary in NETBLT, because single control messages can ACK many data packets (depending on the number of packets per buffer). NETBLT uses such redundancy, at the same time reducing to almost nil the overhead incurred by processing duplicate control messages.

Even with efficient error recovery, continuous transmission is not possible over long-delay networks without a large SOS region. For this reason, NETBLT implementations must make the SOS region as large as resources permit. This is not possible with a window flow-control mechanism; in order to enjoy a large SOS region using window-based flow control, a protocol must in effect eliminate flow control by using extremely large window sizes. NETBLT uses rate control to meter the transmission rate; SOS region management operates independently of flow control.

4.1.3. Coping with Delay

Transmission delay is one of the most difficult problems that a transport protocol faces, because it causes the SOS problem. Resynchronization and error recovery are performed via timers, but in other transport protocols, the timer itself suffers from the delay variations since it is delay-based. This causes a positive feedback cycle of the delay-dependency, lowering performance.

NETBLT's solution to this problem has several parts. First, it uses a large SOS region, to make it unlikely that the transmission is forced to stop due to reaching the region boundary. Second, NETBLT speeds up the state synchronization using the mechanisms discussed above. This allows the SOS region to move forward quickly along the transmission stream, so that the transmission will not be forced to stop due to reaching the region boundary. Third, NETBLT reduces reliance on timers, limiting their use to a last resort in ensuring reliability. Because of the large SOS region, timer values can be set loosely and the protocol can transmit continuously while awaiting a retransmission timeout. Of course, if the SOS region cannot be large (due perhaps to system limitations), then more care must be taken in estimating retransmission timer values.

4.2. NETBLT Protocol Design

Having discussed some of the design philosophies behind NETBLT, we move on to a slightly more detailed discussion of the protocol. NETBLT works by opening a connection between two "clients" (the sender and the receiver), transferring data in a series of large numbered blocks (buffers), and then closing the connection. Because the amount of data to be transferred can be very large, the client is not required to provide at once all the data to the protocol module. Instead, the data is provided by the client in buffers. The NETBLT layer transfers each buffer as a sequence of packets; each buffer is composed of a large number of packets, so the per-buffer interaction between NETBLT and its client is far more efficient than a per-packet interaction would be.

In its simplest form, a NETBLT transfer works as follows: the sending client provides a buffer of data for the NETBLT layer to transfer. NETBLT breaks the buffer up into packets and sends the packets across the network in Internet datagrams. The receiving NETBLT layer loads these packets into a matching buffer provided by the receiving client. When the last packet in the buffer has arrived, the receiving NETBLT checks to see that all packets in that buffer have been correctly received. If some packets are missing, the receiving NETBLT requests that they be resent. When the buffer has been completely transmitted, the receiving client is notified by its NETBLT layer. The receiving client disposes of the buffer and provides a new buffer to receive more data. The receiving NETBLT notifies the sender that the new buffer is ready, and the sender prepares and sends the next buffer in the same manner. This continues until all the data has been sent; at that time the sender notifies the receiver that the transmission has been completed. The connection is then closed.

The above transfer model has an SOS region of one buffer; the model operates in lock-step fashion. Because of network delay, NETBLT typically maintains a large SOS region of a number of buffers (the exact number depends on the network error rate and RTD). This allows the sender to transmit new data while acknowledgments and error recovery for old data take place, and improves performance markedly.

Obviously, the above style of transmission also does not address issues of error recovery or flow control. The following sections detail NETBLTs implementation of the rate control and error recovery schemes described earlier.

4.2.1. Flow Control

NETBLT uses two strategies for flow control--one internal and one at the client level. The sending and receiving NETBLTs transmit data in buffers; client flow control is therefore at a buffer level. Before a buffer can be transmitted, NETBLT confirms that both clients have set up matching buffers, that one is ready to send data, and that the other is ready to receive data. Either client can therefore control the flow of data by not providing a new buffer. Clients cannot stop a buffer transfer once it is in progress.

Since buffers can be quite large, there has to be another method for flow control that is used during a buffer transfer. The NETBLT layer provides this form of flow control. As discussed above, the flow control method used is rate control. The transmission rate is negotiated by the sending and receiving NETBLTs during connection setup and after each buffer transmission. The sender uses timers, instead of messages from the receiver, to maintain the negotiated rate.

In its simplest form, rate control specifies a minimum time period per packet transmission. This can cause performance problems for several reasons. First, the transmission time for a single packet is very small, frequently smaller than the granularity of the timing mechanism. Also, the overhead required to maintain timing mechanisms on a per packet basis is relatively high and lowers performance.

The solution is to control the transmission rate of groups of packets, rather than single packets. The sender transmits a burst of packets over a negotiated time interval, then sends another burst. In this way, the overhead decreases by a factor of the burst size, and the per-burst transmission time is long enough that timing mechanisms will work properly. NETBLT's rate control therefore has two parts, a burst size and a burst rate, with (burst size)/(burst rate) equal to the average transmission time per packet. The burst interval is the number of milliseconds between the start of one burst transmission and the start of the next. The burst size is the number of data packets in a burst.

These two values reflect more accurately than a window the resources available to the receiver of data. They can reflect, for instance, a slow machine with little buffer space (long burst interval, small burst size), or a faster machine with a high process-scheduling overhead (short burst interval, large burst size). Note that the rate control parameters only deal with buffering at a very low level (the burst size can be based on the number of packet buffers available to the protocol module). Higher level buffering is decoupled from flow control, unlike TCP's window flow control strategy.

4.2.2. State Synchronization and Error Recovery

The receiving and sending NETBLTs synchronize their connection states via three different control messages: GO, RESEND, and OK. When the receiving NETBLT has a buffer N ready to receive data, it transmits a GO[N] message to the sender; this message tells the sender to begin transmission of buffer N. As soon as the first data packet in N arrives at the receiver, the receiver sets the "data timer"[2] belonging to N. The timer estimates how long it will take for the remainder of the buffer to arrive. The timer is fairly easy to estimate, because of NETBLT's "priority pipe" method of transmission: all buffers are transmitted in order by buffer number (buffers are given monotonically increasing numbers, from first buffer through last buffer). Once the sending NETBLT begins transmission of a buffer, it guarantees to transmit the entire buffer at the current rate before transmitting any later buffers. The remaining packets are therefore expected to arrive at the determined rate unless they are lost or delayed by the network. The timer value can be based either on the negotiated rate or the inter-packet arrival time.

The receiver now waits for one of two events. Either all packets for the buffer arrive, or the data timer expires. In the first case, the receiver clears the buffer's data timer and sends an OK[N] message, telling the sender that it can release buffer N. It may follow this with a GO[N+M] message, starting the transfer of another buffer. In the second case, the receiver looks at which packets in buffer N have arrived, and sends a RESEND[N] message containing a list of the missing packets. The sender retransmits these packets along with possible new data from subsequent buffers, all at the negotiated rate. Because all outgoing data are ordered by buffer number, retransmitted data are sent before new data. Assuming all the retransmitted packets arrive safely, the receiver sends an OK[N] message and deactivates N's data timer.

In order for the sending and receiving NETBLTs to synchronize their states as quickly as possible, control messages must be delivered reliably and in a timely fashion. NETBLT insures control message reliability in two ways. First, it uses a highly redundant message transmission algorithm. In NETBLT, multiple control messages can be packed into a single packet; the receiving NETBLT maintains a single long-lived control packet containing multiple messages, which is transmitted every time a group of new messages is generated by the receiver. ACKed control messages are pushed off the front of the packet, and new messages are added at the back of the packet, so a given message is transmitted as a member of the control packet until it is ACKed by the sender of data. Control message ACKs are by sequence number. Each control message has a unique sequence number which increases by one for each message sent. The sender of data receives these messages and notes the highest sequence number below which all messages have been received. It returns this "high ACK sequence number" in all packets flowing back to the receiver of data. When the receiver gets a high ACK sequence number, it pushes off the front of the control packet all messages with a sequence number less than or equal to the high ACK number.

The above message transmission algorithm gives high redundancy, which increases as the network delay increases. The sending NETBLT has been designed to throw away duplicate control messages with almost no overhead. Also, individual control messages are very short in length, so the bandwidth consumed by these messages is quite low. The cost of this redundancy is therefore fairly low and the benefits valuable.

Even with the redundant transmission algorithm described above, a control message will occasionally get lost. The receiving NETBLT therefore maintains a control message retransmit timer. The timer value is based on the network RTD, and is reset every time the control packet is transmitted. The timer is cleared whenever all messages in the control packet have been ACKed by the sender of data. Obviously, being based on the network RTD, this timer algorithm falls prey to the same problems that the TCP retransmit timer does. In NETBLT's case, however, the reliance on the control timer is so reduced (by the redundant transmission algorithm), that the retransmit timer almost never expires, and its value can be quite loose.

5. NETBLT Experience

In order to test NETBLT's effectiveness at providing high throughput over noisy or long-delay networks, extensive testing was conducted over a number of different networks. The current NETBLT test implementations run under UNIX (via the user-accessible raw IP socket, not the kernel) on a SUN-3 workstation, and under MS-DOS on an IBM PC/AT. A third implementation for the Symbolics Lisp Machine was also briefly tested[3].

Over high-bandwidth, short-delay LANs, NETBLT performed extremely well, achieving application memory-to-memory transfer rates of up to 1.75 megabits per second between two IBM PC/ATs on a 10 megabit-per-second Proteon Token Ring. Throughputs of up to 1.46 megabits per second were achieved over a heavily used 10 megabit-per-second Ethernet. When running memory-to-memory transfers over a Microvax-II-based C gateway connecting a Proteon Token Ring and an Ethernet, NETBLT achieved throughputs of up to 1.43 megabits per second. Since the above network environments are relatively hospitable, this came as no great surprise.

[2] The actual data timer mechanism is more complicated than this, but the added detail tends to obscure rather than clarify.

[3] A document describing in detail NETBLT implementation tests over a variety of networks is forthcoming.

A more accurate measure of NETBLT's abilities was in transmissions across the 3 megabit-per-second Wideband satellite network. This network provides high bandwidth with long (1.8 second) round-trip delay. In addition, the Wideband network possesses some interesting packet re-ordering and delay characteristics that resulted in substantial changes to the original NETBLT protocol design.

5.1. Coping with Delay

NETBLT's ability to handle large SOS regions proved invaluable in coping with the Wideband network's 1.8 second round-trip delay. Most tests used an SOS region ("window") of over 300,000 bytes with no performance loss due to state management overhead. Obviously not all systems will have that much available buffer space; the interesting point is that it proves large SOS regions do indeed provide high throughputs, and in NETBLT's case, the management of such a large region did not inhibit performance any. The large SOS region allowed NETBLT to transmit constantly, overlapping error recovery and ACK waits with the transmission of new data.

NETBLT was also eventually able to handle the Wideband network's tendency towards large delav variances. Under heavy load, the Wideband network can vary the delay of groups of packets by 600-700 milliseconds, due to its packet reservation scheme. The NETBLT receiver's data timers initially did not tolerate this delay well, and initially there were many spurious retransmission requests as data timers timed out due to delayed packets.

A solution to this problem was to base the data timer value on the inter-packet arrival time, rather than the negotiated rate. The receiver could then immediately increase its data timer values to reflect packet delay caused by congestion. The result was fewer false data timeouts and spurious retransmissions. Another solution is to set the data timer very loosely and correspondingly enlarge the SOS region. Obviously, machines which take the latter step will require large amounts of memory dedicated to NETBLT.

5.2. Packet Re-ordering

During initial testing, the Wideband network re-ordered groups of packets in a range of up to 100 (i.e. packet 100 would arrive before packet 1). Eventually, NETBLT was able to deal with this problem with a minimum of performance degradation. It forced us to rely more heavily on data timers, but this seems to have had little effect on performance over the Wideband network. Initially, NETBLT attempted to make retransmit "guesses" ahead of the data timer in order to improve efficiency. One such guess involved immediately generating a retransmit request if data packets were missing at the time the last packet in a buffer was received. Because of packet re-ordering, this "last packet" would frequently arrive before the first packet, causing a spurious retransmit for the entire buffer. This guess was eliminated, forcing retransmissions only upon expiration of the data timer.

NETBLT would also generate retransmit requests if packets for a buffer N+M arrived while packets for a buffer N were still missing (because the sending NETBLT transmits buffers in sequence, lowest number first, this was a reasonable guess on networks with little packet re-ordering). Again, packet re-ordering would cause spurious retransmission requests, so this guess was eliminated in favor of a slightly more complicated guess that involved resetting buffer N's data timer rather than generating a retransmit request.

5.3. Rate Control

Because of the Wideband network's long RTD, a window-based flow control mechanism would need to use a window so large that the flow control effect was essentially nonexistent. By decoupling the SOS region and the flow control, NETBLT was able to eliminate this problem. Rate control proved to work extremely well[4]. Instead of the transmission "hanging" while its window waited to open, NETBLT transmitted constantly at the negotiated rate with no slowdowns. In fact, rate control was ideally suited to the wideband network, which transmits packets internally in the form of timed bursts. By matching NETBLT's bursts to the network's, an high percentage of the network's bandwidth was utilized.

5.4. Results

NETBLT performed extremely well over the Wideband network. Out of a maximum available bandwidth of approximately 1 megabit per second[5], NETBLT managed a consistent steady-state throughput of 920,000 to 945,000 bits per second at the network's maximum transmission rate. At slower transmission rates, NETBLT worked at close to 100% efficiency.

Although rate control provides a neat solution to many problems, two important hurdles must be crossed before rate control can be used in production protocols. First, the protocol must be able to set an initial transmission rate based on the available network bandwidth and the speed at which the receiver can process data. The protocol must be also able to dynamically change the transmission rate in the face of changing network and receiver load. No matter how congested a network is, NETBLT will currently happily transmit packets at the negotiated rate until it reaches its SOS region boundary. The protocol has the ability to re-negotiate transmission rates at buffer boundaries; unfortunately if NETBLT uses internal state information to change the rate, it must make guesses based on incomplete information, and could start oscillations in the transmission rate. The guesses would permit dynamic changes to the transmission rate, but would still require a "blind" guess for an initial transmission rate. It is therefore important that support for rate selection exist at the network level. Of course this would require modifications to network gateways; even with the necessary modifications, the assumption must be made that the network load changes slowly enough that information gotten from the gateways is meaningful by the time it arrives.

There has been some work attempting to analyze the problem of adjusting the rate of a number of simultaneous flows [1]. But the problem is difficult, and both practical and analytical experience will be needed to identify a suitable approach.

A second problem is packet loss due to the lack of flow control in the link layer. NETBLT provides high enough performance that it uncovers previously hidden low-level network problems. For

[4] The next section discusses some enhancements that must be made for rate control to work correctly all the time.

[5] Obviously far less than the network's stated raw bandwidth. Unfortunately network overhead consumed fully 2/3 of the available bandwidth at the time we were testing.

example, the current NETBLT implementation operates on a machine with a fairly unsophisticated Ethernet network interface. It is possible, using NETBLT, to exceed the network interface's receiving speed when receiving bursts of packets. On a single-wire link, this problem can be eliminated by judicious choice of burst size and interval. If there are gateways along the transmission path, however, the transport layer no longer has control over the speed at which bursts of packets arrive. Also, congestion may occur anywhere along a transmission path -- within gateways or along a single Ethernet cable. If the NETBLT layer has enough low-level knowledge of its link level's capabilities, it can cope with the lack of flow control in the link layer. This is not, however, a good solution since it violates protocol layering. This problem can only be solved by developing link-layer flow control mechanisms.

Clearly, rate control can only offer optimal throughput if it is supported by all lower protocol layers. The network must be able to offer load information which NETBLT can use to determine an optimal initial transmission rate. It must also constantly update this information so that the transmission rate can be adjusted if the network load changes.

6. Summary

In this section we first summarize the above discussions to a few guidelines for reliable transport protocol design; we then look at a couple of existing protocols, followed by a brief conclusion of the paper.

We consider that the following can be used as guidelines in future reliable transport protocol designs:

1. Employ separate mechanisms for data flow control and error recovery.
2. Use a rate flow control to match the data transmission and consumption rates at the two communicating ends.
3. Set as large an SOS region as resources permit.
4. Explore feasible mechanisms, such as explicit information exchange between the transmitter and receiver about each other's behavior, using SACK instead of simple ACKs, and redundancy in data or control transmissions, to help speed up the end state resynchronization.
5. Coordinate with the underlying network traffic control.

Design efforts should be directed to synchronize the end state as quickly as possible. On the other hand, there is also a concern with the overhead involved in the synchronization, and we need a good judgment on balance.

Now we look at the approaches taken by a few existing protocols. As mentioned earlier, TFTP sets an SOS region of one packet, and hence loses throughput over long delay channels. At the other extreme, Blast [4] sets the SOS region to the total amount of data being transmitted. Based on the assumption of a low loss, low delay communication channel, the transmitter sends all data at once, under some rate flow control (the intention is to match incoming data with the disk operations), then recovers transmission errors according to SACKs from the receiver. Since there is no overlap between data transmission and error recovery phases, the latter can cause a long tail in the transmission process, if the channel is noisy or the delay is long, dragging the throughput value over the total transmission period to a very low average[6].

To manage the transmission state, there is a question of what basic data unit should be used in the transmission state management. The smaller the unit, the larger the state space needed to cover the same amount of data. For example, TCP uses the byte as the basic unit, while TP4 uses the packet, therefore improving TCP's performance by adding a SACK mechanism may be slightly more complex and costly than doing so to TP4.

Much work has been done on the performance issues of transport protocols. The flow control, transmission error recovery, and buffer space management issues seem to interact in a rather complicated way [5]. We showed in the above that once we understand the impact of network delays and errors on the transport protocol performance, and decouple flow control from error recovery mechanism, the problem is simplified and solutions are revealed.

NETBLT experience has helped us reach a better understanding on transport protocol performance issues. Remaining research topics are related to the flow control interactions with the network:

1. How good a performance can the end hosts achieve by tuning the control rate, under varying network conditions?
2. How does one design a network architecture to support rate-based traffic control?

References

[1] D.Bertsekas and R. Gallager.
Flow Control Schemes Based on Input Rate Adjustment.
Data Networks.
Prentice-Hall, Inc., 1987, Chapter 6.4.

[2] David Clark, Mark Lambert, and Lixia Zhang.
NETBLT: A Bulk Data Transfer Protocol.
Network Information Center RFC-998, SRI International.
March, 1987

[3] Karen Sollins.
The TFTP Protocol.
Network Information Center RFC-783, SRI International.
June, 1981

[4] Dan Theriault.
BLAST, an Experimental File Transfer Protocol.
MIT-LCS Compuer System Research Group RFC-217.
March, 1982

[5] R. Watson and S. Mamrak.
Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices.
ACM Transactions on Computer Systems 5(2):97-120, May, 1987.

[6] Lixia Zhang.
Why TCP Timers Don't Work Well.
In *Proceedings of Symposium on Communication Architectures and Protocols.* ACM SIGCOMM, 1986.

[6] A common misconception is that NETBLT and Blast look similar to each other. Actually the two are very different except that both protocols use separate flow control and error recovery mechanisms.