PROTOCOLS FOR LARGE DATA TRANSFERS OVER LOCAL NETWORKS

Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, TX 77251

Abstract

In this paper we analyze protocols for transmitting large amounts of data over a local area network. The data transfers analyzed in this paper are different from most other forms of large-scale data transfer protocols for three reasons: (1) The definition of the protocol requires the recipient to have sufficient buffers available to receive the data before the transfer takes place; (2) We assume that the source and the destination machine are more or less matched in speed; (3) The protocol is implemented at the network interrupt level and therefore not slowed down by process scheduling delays.

We consider three classes of protocols: *stop-and-wait, sliding window* and *blast* protocols. We show that the expected time of blast and sliding window protocols is significantly lower than the expected time for the stop-and-wait protocol, with blast outperforming sliding window by some small amount. Although the network error rate is sufficiently low for blast with full retransmission on error to be acceptable, the frequency of errors in the network interfaces makes it desirable to use a more sophisticated retransmission protocol. A go-back-n strategy is shown to be only marginally inferior to selective retransmission and is, given its simplicity, the retransmission strategy of choice.

Our results are based on measurements collected on SUN workstations connected to a 10 megabit Ethernet network using 3-Com interfaces. The derivation of the elapsed time in terms of the network packet error rate is based on the assumption of statistically independent errors.

1. Introduction

Recent studies have shown the importance of using large page sizes in order to achieve high performance file access, both locally as well as over a network [10,12,15]. This is due to economies in accessing the disk in large quantities as well as to economies in accessing the network in large quantities. In this paper we study the latter phenomenon. In particular, we study the performance of protocols for transmitting large amounts of data across a local network characterized by a low error rate, low propagation delay and high bandwidth.

By *large* amounts of data, we denote amounts that are one or two orders of magnitude bigger than the maximum network packet size. We show how our analysis can be extended to larger sizes such as those involved, for instance, in remote file system dumps. We study three classes of protocols: *stop-and-wait, sliding window* and *blast* protocols (See Figure 1). With *stop-and-wait* protocols, the source refrains from sending a packet until it has received an acknowledgement for the previous packet. With a *blast* protocol all data packets are transmitted in sequence, with only a single acknowledgement for the entire packet sequence. Different protocols within the category of blast protocols are distinguished by their retransmission strategies (e.g. all packets can be retransmitted, or some form of selective retransmission can be used). With *sliding window* protocols every packet is individually acknowledged but the sender continues to transmit data without waiting for an acknowledgement. In typical sliding window protocols, the sender is silenced when the window "closes". Here we assume that the window is large enough so that it never gets closed.

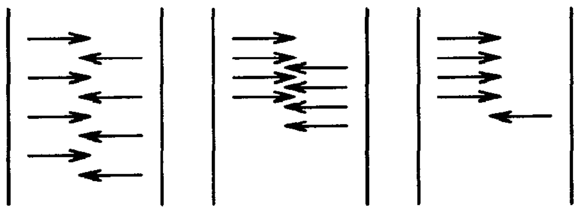

Figure 1: Stop-and-Wait, Sliding Window and Blast Protocols

For error-free transmissions, stop-and-wait protocols do not perform as well as sliding window or blast protocols, because of delays in waiting for the acknowledgement for every packet. Given the low latency and the high bandwidth of local networks, one would expect the difference in performance to be rather small. However, experimental evidence shows that the penalty for using a stop-and-wait protocol on a local network is substantially higher than expected, because of significant extra delays in generating and receiving packets. Sliding window protocols are slightly inferior to blast protocols, because of the overhead involved in handling the extra acknowledgements.

We then consider the performance of the protocols in the presence of transmission errors on the network. Given typical error rates on a local network, the expected elapsed time of a given transmission is almost identical to the error-free transmission time. As a result, under normal local network operating conditions, blast and sliding window protocols outperform stop-and-wait protocols. In fact, ***network error*** rates are sufficiently rare to make it possible to use full retransmission on error in conjunction with a blast protocol without significant degradation in the expected elapsed times. However, frequent ***interface errors*** force a more sophisticated retransmission strategy in order to maintain a near-optimal expected time and a small standard deviation.

Our results are based on measurements of error-free transmission times between SUN workstations connected to a 10 megabit Ethernet network using a 3-COM Multibus interface, and on a probabilistic analysis of the performance in the presence of errors. Since our measurements are done in the absence of any substantial network load, contention delays are all but absent from the results. Our conclusions are therefore valid only under low load conditions. Fortunately, such conditions are typical of most local network based systems. We also use delay under low load as a measure of performance, rather than throughput under high load, because low load conditions are so prevalent. In the error analysis, we assume that packet transmissions are statistically independent events with a constant failure probability. In practice, this assumption is a reasonable approximation of reality, although burst errors occasionally occur.

The outline of the rest of this paper is as follows. Section 2 describes measurements of error-free transmissions using each of the three protocols considered. In Section 3, we compare the performance of these three protocols in function of the error rate of the network. In Section 4, we study different retransmission strategies that can be used in conjunction with blast protocols. Related work is covered in Section 5 and conclusions are drawn in Section 6.

2. Error Free Data Transmissions

The large data transfers discussed in this paper occur as part of the interprocess communication functions provided by the V kernel [4,6]. The V kernel is a distributed operating systems kernel, currently implemented on SUN workstations [2] connected to a 10 megabit Ethernet [8] by a 3-Com interface [1]. As part of its interprocess communication facilities, the V kernel provides two operations — **MoveTo** and **MoveFrom** — which allow one process to move an arbitrary amount of data from its address space into the address space of another process, or vice versa. Both operations are ***network transparent:*** the destination process may or may not be on the same machine as the source process. By definition of the V interprocess communication primitives (see [4,6]), the recipient has sufficient buffers allocated to receive the data prior to the transfer. For instance, when a process wants to read an entire file into its address space, it first allocates a buffer big enough to contain that file. It then sends a message to the file server indicating the starting address of the buffer and its length. If necessary, the file server reads the file from disk, and then uses **MoveTo** to move the file from its address space into that of the client. In the local case, the fact that the client's buffer is already allocated allows the kernel to move the data from the source to the destination address space without an intermediate copy. In the remote case, it allows the kernel(s) to move data from the source address into the network interface of the sending machine, and from the network interface of the receiving machine into the destination address space, again without an intermediate copy.[1]

The total time necessary to execute a **MoveTo** or a **MoveFrom** is the sum of the cost of network communication and the cost of kernel overhead. In Section 2.1, we describe a set of experiments to quantify the cost of network communication. Measurements with the V kernel implementation of **MoveTo** and **MoveFrom**, including both network communication and kernel overhead, are discussed in Section 2.2.

2.1. Network Communication Overhead

On a local network, one would expect a blast or a sliding window protocol to perform only marginally better than a stop-and-wait protocol, because of the low propagation delay and the high bandwidth of the network. For instance, assume that $N = 64$ kilobytes of data have to be transferred over a 10 megabit Ethernet. Assume that we transmit the data in 1 kilobyte packets and that the acknowledgement packets are 64 bytes in size. Based on the 10 megabit data rate of the network, the transmission of a data packet takes $T = 820$ microseconds and the transmission of an acknowledgement packet takes $T_a = 51$ microseconds. The latency of the network τ can be estimated to be below 10 microseconds. If a stop-and-wait protocol is used, and no errors occur, the transfer is complete (including the receipt of the last acknowledgement at the source) after

$$N \times (T + T_a + 2 \times \tau) \text{ or } 57024 \textit{ microseconds}$$

For a sliding window protocol, with a window large

[1] At least with suitable network interfaces.

enough so that it never closes during transmission, the elapsed time becomes

$$N \times (T + T_a) + 2 \times \tau \text{ or } 55764 \textit{ microseconds}$$

If the data is transmitted by a blast protocol, then the resulting elapsed time is

$$N \times T + T_a + 2 \times \tau \text{ or } 52551 \textit{ microseconds}$$

None of these results differ from each other by more than 10 percent. The experiment described below contradicts the above line of reasoning: the stop-and-wait protocol takes about twice as much time as either the sliding window or the blast protocol. We first describe the experiment and its results, and then explain this somewhat counterintuitive outcome.

2.1.1. Experimental Method and Results

In order to quantify the cost of network communication, two standalone programs are run on two different machines connected to the network. One program acts as the source of data and the other as the destination. Data is transmitted from the source, and acknowledgements are returned from the destination as appropriate, according to which protocol is used. Data packets are 1024 bytes in length, while acknowledgements are 64 bytes. For statistical accuracy, the experiment is repeated a number of times and the results are averaged. In all measurements, the network is essentially idle, so no significant contention delay is experienced. The transfers are implemented at the data link layer and device level so that no protocol or process switching overhead appears in the results. In particular, no header (other than the Ethernet data link header) is added to the data, and no provisions are made for demultiplexing packets, or for retransmission. If a transmission error occurs, the experiment halts and is restarted. Also, each of the two programs simply busy-waits on the completion of its current operation, in order to avoid interrupt handling overhead. The experiment therefore provides an accurate approximation of the communication cost involved in the data transfer. Measurement results for the elapsed time of multi-packet transfers in stop-and-wait (SAW), sliding window (SW) and blast (B) mode are reported in Table 1.

Size	SAW	SW	B
1 Kb	4.1 ms.	4.1 ms.	4.1 ms.
8 Kb	32.7 ms.	21.7 ms.	19.8 ms.
64 Kb	261.6 ms.	161.5 ms.	141.1 ms.
512 Kb	2,093.0 ms.	1,284.0 ms.	1,149.0 ms.

Table 1: Standalone Measurements of Error Free Transmissions

2.1.2. Interpretation

Let us first consider data transfers that fit within a single network packet, for instance a 1 kilobyte transfer.[2] The reliable transfer of 1 kilobyte of data, using a 64 byte acknowledgement, takes 4.1 milliseconds (See Table 1). This is significantly more than the transmission time of the data and the acknowledgement packet, which would be 0.87 milliseconds, when computed at the 10 megabit data rate of the network. The difference between the network time, computed at the network data rate, and the measured elapsed time is accounted for almost exclusively by the time necessary for the processors to copy the packets into and out of the interface (See Figure 2).

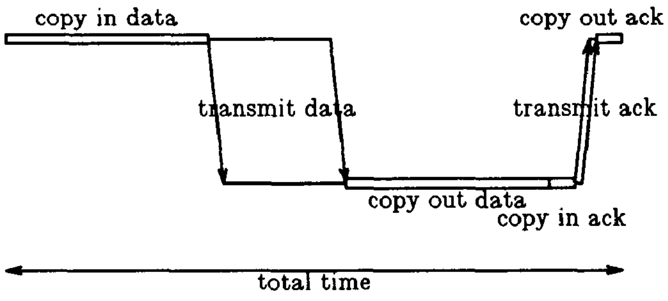

Figure 2: Network Packet Transmission

Table 2 shows a breakdown of the total elapsed time over its various components. As can be seen, of the 4.1 milliseconds total elapsed time, only 21 percent is network transmission time, while 75 percent is copying overhead, the rest (presumably) being network and device latency.

Operation	Time
Copy data into sender's interface	1.35 ms.
Transmit data	0.82 ms.
Copy data out of receiver's interface	1.35 ms.
Copy ack into receiver's interface	0.17 ms.
Transmit ack	0.05 ms.
Copy ack out of sender's interface	0.17 ms.
Total	3.91 ms.
Observed elapsed time	4.08 ms.

Table 2: Breakdown of Transmission Cost over its Various Components

Let us now consider the case where the data transfer requires N packets to be sent from the source to the destination. The reason for the superior performance of the blast and sliding window protocols is explained in Figure 3. Figure 3.a corresponds to the transmission in stop-and-wait mode, Figure 3.b corresponds to the blast transmission, and Figure 3.c depicts the sliding window protocol. The time axis runs horizontally from left to right and the example is for the case of N=3.

[2] The maximum packet size on the 10 megabit Ethernet is 1536 bytes.

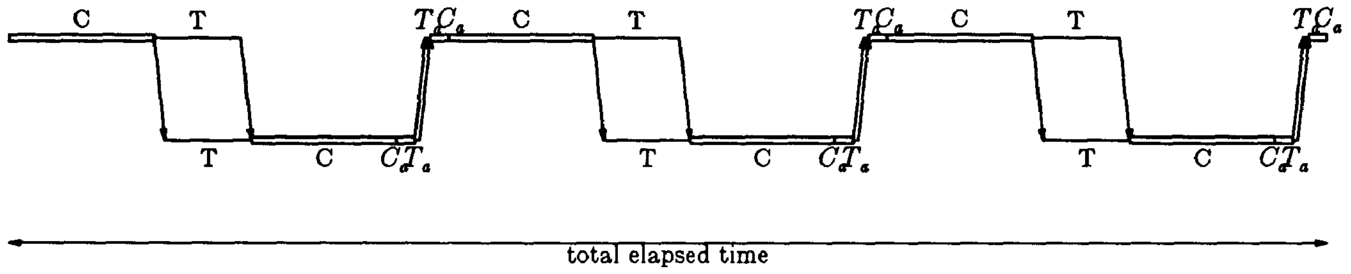

Figure 3.a: Stop-and-Wait

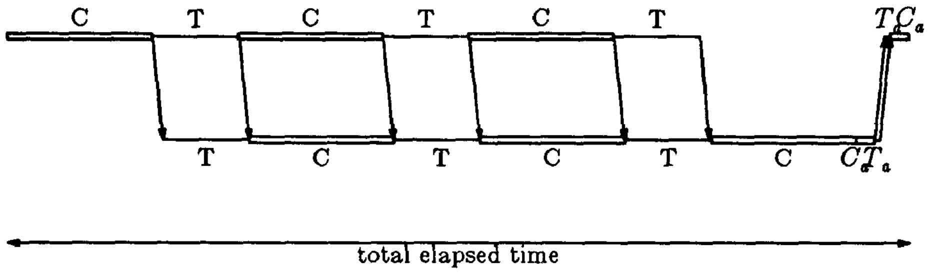

Figure 3.b: Blast Protocol

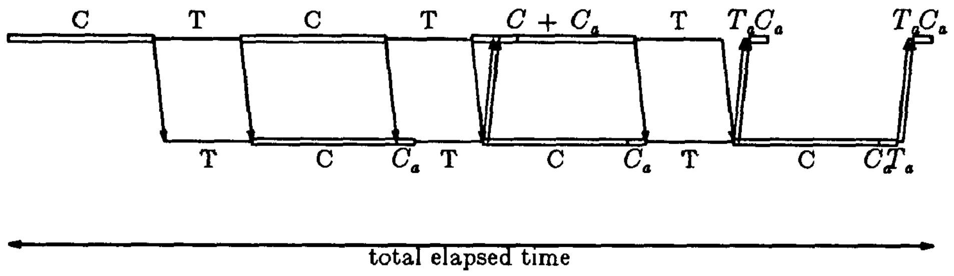

Figure 3.c: Sliding Window Protocol

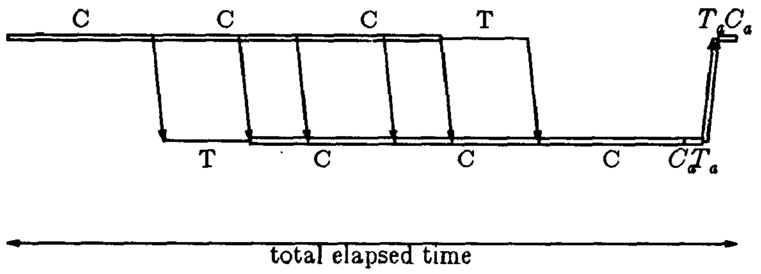

Figure 3.d: Double Buffered Interface with Blast Protocol

Consider first the sequence of events in the case of a stop-and-wait protocol. The sending processor copies a packet from main memory to its interface and then the interface puts the packet on the network. After a time period equal to the network propagation delay the packet arrives at the receiver's interface and then it is copied from the receiver's interface into the receiver's memory by the receiving processor. This process repeats itself in the reverse direction for the acknowledgement packet, and then again for the next packet, and so forth. Note that the two processors are never active in parallel. This is not the case when the transfer is done in blast mode, as shown in Figure 3.b. Due to the very low propagation delay of a local network, the packet is received in the receiver's interface almost completely concurrently with the sender's interface transferring it over the network.[3] Therefore the processor on the sending machine can start copying the next packet from memory to its interface in parallel with the processor on the receiving machine copying the previous packet out of its interface into its memory. Due to the fact that these copies happen in parallel, and, as we saw before, the copy times contribute significantly towards the overall elapsed time, blast transfer results in elapsed times that are substantially lower than those obtained for stop-and-wait transfers, much lower than one would be lead to expect if only transmission time for acknowledgements were taken into account.

Finally, consider the operation of the sliding window protocol (Figure 3.c). Again, the copy operations in and out of the interface happen in parallel on the sender and on the receiver. The reason for the slightly inferior performance of the sliding window protocol vs. a blast protocol is that for each packet an acknowledgement has to be copied in and out of the interfaces, while for the blast protocol, there is only an acknowledgement for the last packet.

2.1.3. Formulas for Error Free Transmissions

From Figure 3, we can derive the following formulas for the elapsed times of multi-packet data transfers in the absence of errors. First, in the case of a stop-and-wait protocol, the total elapsed time T_{SAW} for a transmission requiring N data packets, ignoring device latency, is (See Figure 3.a)

$$T_{SAW} = N \times (2C + T + 2C_a + T_a)$$

where C stands for the time necessary to make a copy of a data packet into interface, C_a for the time necessary to make a copy of an acknowledgement packet into the interface, T for the network transmission time of a data packet, and T_a for the network transmission time of an acknowledgement packet. If a blast protocol is used, then due to concurrent operation of the two processors, the elapsed time T_B becomes (see Figure 3.b)

$$T_B = N \times (C + T) + C + 2C_a + T_a$$

For a sliding window protocol, the resulting elapsed time T_{SW} equals (see Figure 3.c)

$$T_{SW} = N \times (C + C_a + T) + C + T_a$$

Note that the utilization of the network U_n, even when using a blast protocol, is still significantly below 100 percent.

$$U_n = \frac{N \times T + T_a}{N \times T + T_a + N \times C + C + 2C_a}$$

For instance, for the 64 kilobyte transfer shown in Table 2, the network utilization is only 38 percent. Better elapsed times and better network utilization can be obtained if a double buffered interface is used. In that case, the processor can start copying a packet into the second buffer in the interface while the interface is transmitting the previous packet over the network, and similarly on the receiving machine . Note that having a third transmission buffer does not provide any further improvement over double buffering, since we assume that both C and T are constant. The value of T is constant as long as there is no significant contention delay. The value of C is also constant since we assume the network transfer is the only activity occupying the processor, and therefore there is no delay in performing the copy operation. The elapsed time $T_{d/b}$ becomes (See Figure 3.d)

$$T_{d/b} = N \times C + T + C + 2C_a + T_a \quad (T \leq C)$$

$$T_{d/b} = N \times T + 2C + 2C_a + T_a \quad (T \geq C)$$

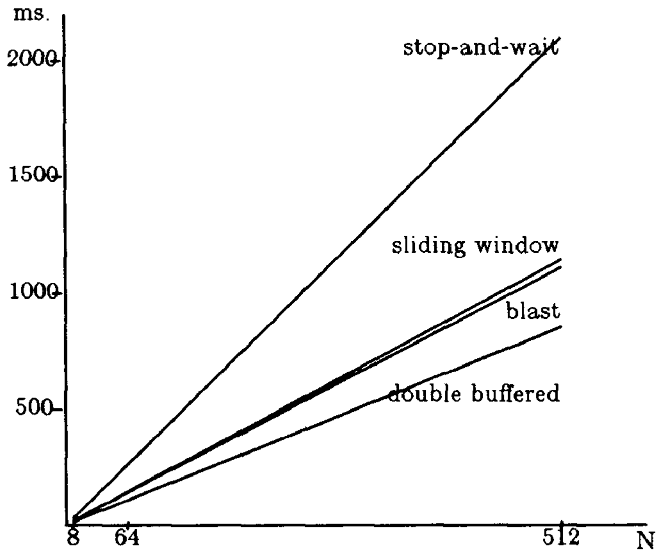

Figure 4: Comparison of Different Protocols

[3] In fact, the propagation delay is far exaggerated in Figures 2 and 3 to make it visible at all: typical propagation delays on a local network are on the order of 10 microseconds while the copy and transmission times depicted in Figure 2 and 3 are on the order of 1 millisecond.

Figure 4 compares the performance of the different protocols in terms of N for values of C, C_a, T and T_a as on the SUN workstation with a 10 megabit Ethernet (See Table 2).

One might wonder whether it is possible to get rid of the copy into the interface altogether, by simply moving the data from main memory onto the network. An interesting design that allows network access without an intermediate copy appears in the Xerox Alto personal computer, where network access is incorporated as an independent task in the processor's micro-engine [20]. The copy can also be avoided by virtual memory techniques (if the origin and the destination of the data are aligned on a page boundary). The Apollo Domain architecture supports this feature [11]. For more conventional architectures, one would like a DMA interface to copy the data from main memory directly onto the network. Most DMA interfaces do not allow such a direct copy. For instance, the Excelan DMA interface first copies the data into on-board buffers before it transmits it on the network [9]. The CMC interface allows the programmer to define the host-board interface [7]. However, the manual strongly recommends that data be copied first into on-board buffers before it is transmitted onto the network, due to limited bus bandwidth and due to the possibility of the host processor timing out on memory access over the bus, while the DMA processor is accessing memory [7,16]. Both interfaces seem to require a copy, albeit that the copy is performed by the interface processor rather than by the host processor. The formulas derived above for the elapsed time therefore remain valid, provided that C and C_a are no longer the time required for the host processor to make the copies, but rather the time required for the DMA processor to make the copies. Our experience with DMA interfaces has been mixed in this respect. With the Excelan board, the copy performed by the 8088 interface processor is much slower than the copy performed by the 68000 host processor into the 3-Com interface. We have no experience to date with the performance of the CMC interface (which contains a 68000 processor).

In summary, it seems that the elapsed time is not significantly improved by using currently available DMA interfaces. The amount of host processor utilization for network access is decreased, since the interface processor can perform the copy, although bus traffic slows down the processor somewhat during DMA operation. In general, the relative importance of the copy operation indicates that memory and bus bandwidth are the critical factors. Therefore, it seems likely that a processor with a fast block move operation, accompanied by very high speed device memory is more promising than any kind of special purpose hardware on the interface.

2.2. Large Data Transfers at the Kernel Level

The protocol used for the set of measurements in the previous section assumes that the network is error-free. When an error occurs, the experiment is terminated and has to be (manually) restarted. A real protocol has to deal with the possibility of errors on the network by some form of retransmission strategy. This introduces overhead, even for error-free transmissions. Additionally, in the experiment above network access is the only task that the processor has to perform. There is no multiplexing, no access rights checking, and the processor busy-waits on the completion of a transmission in order to start the next one. In a real system, the processor has to be shared by a number of different tasks, that have to be protected from each other, and network I/O has to be performed in an interrupt-driven rather than a busy-wait fashion.

All of these requirements have been implemented as part of the V kernel's network interprocess communication. Table 3 gives the results of our measurements of the V kernel's **MoveTo** operation.[4] The results only confirm the measurements of Section 2.1. The extra overhead stems from the transmission of the headers, as well as from access right checking, demultiplexing and interrupt handling. The formulas derived for the elapsed times under various combinations of transmission protocols and network interfaces remain valid, if the extra overhead is added to C, and C_a. For instance, for the blast protocol, the modified values of C and C_a are 1.83 and 0.67 milliseconds, vs. 1.35 and 0.17 milliseconds in the standalone experiments. The relative increase of C and C_a, compared to T and T_a, makes the blast protocol even more advantageous than in the case of a standalone program. In fact, in the case of the V kernel, the extra overhead is relatively minimal compared to other protocol implementations, which suggests that the use of a blast protocol would be even more advantageous for other implementations.

Size	SAW	B
1 Kb	5.9 ms.	5.9 ms.
8 Kb	43.1 ms.	24.8 ms.
64 Kb	340.2 ms.	173.0 ms.
512 Kb	2,719.0 ms	1,370.0 ms.

Table 3: V Kernel Moveto Measurements

3. Effect of Transmission Errors

So far we have considered the performance of the various protocols for the **MoveTo** operation in the case of error-free transmissions. We now turn our attention to their performance in the presence of transmission errors on the network. We first compare stop-and-wait with retransmission of a single packet after a time interval T_r to blast with retransmission of the full sequence of packets after T_r. We show that, for typical local network error rates, the expected time of the blast transmission is

[4] Measurements for the sliding window protocol are not available at the time of writing. We would expect the elapsed times for sliding window to be slightly higher than those for blast, as suggested by the standalone measurements in Table 1.

significantly better than the expected time of the stop-and-wait transmission. We conclude therefore that under normal operating conditions, a blast protocol is superior to a stop-and-wait protocol for use on a local network. We then consider various retransmission strategies that can be used in conjunction with a blast protocol. We do not consider the sliding window protocol in any great detail in this section. Its error characteristics are similar to those of the blast protocol with selective retransmission (See Section 3.2.3).

In this analysis, we assume that packet transmissions are statistically independent events which can fail with probability p_n. This is a reasonable first order approximation of the behavior of the network. Analysis of the performance under other error distributions is beyond the scope of this paper.

3.1. Expected Time

3.1.1. Stop-and-Wait

Denoting by $T(D)$ ($T(1)$) the time necessary for a D-packet (1-packet) transfer, we obviously have

$$T(D) = D \times T(1)$$

The probability p_e of a 1-packet exchange failing is

$$p_e = 1 - (1 - p_n)^2$$

and the probabilities $s(i+1)$ of the exchange succeeding on the (i+1)-th transmission attempt form a geometric distribution with parameter p_e

$$s(i+1) = p_e^i \times (1 - p_e)$$

The expected time for a 1-packet transfer to complete is then

$$T_0(1) + (T_0(1) + T_r) \times (\frac{p_e}{1-p_e})$$

where $T_0(1)$ is the time necessary for a 1-packet exchange without any errors.[5] For D packets, we get for the expected time

$$\mu = D \times [T_0(1) + (T_0(1) + T_r) \times (\frac{p_e}{1-p_e})]$$

3.1.2. Blast

Let us next consider the performance of a blast protocol with full retransmission on error (without a negative acknowledgement). A D-packet transfer succeeds in this case if all D packets reach the destination machine and the acknowledgement packet reaches the source machine. Assuming independent transmissions, the probability of the D-packet transfer failing is then

$$p_e = 1 - (1 - p_n)^{D+1}$$

The probabilities s(i+1) that a transmission succeeds on the (i+1)-th transmission attempt form a geometric distribution with parameter p_e.

$$s(i+1) = p_e^i \times (1 - p_e)$$

The expected time T(D) for a D-packet transfer becomes

$$\mu = T_0(D) + (T_0(D) + T_r) \times (\frac{p_e}{1-p_e})$$

3.1.3. Comparison

Figure 5 compares the two strategies for different values of T_r (The other parameters in the figure are D = 64, $T_0(1) = 5.9$ *msec.* and $T_0(D) = 173$ *msec.*, from Table 3). In addition to these curves we need some idea about the error rate on a 10 megabit Ethernet. Surprisingly enough, very little empirical data is available about the error rates on local networks. Shoch and Hupp report an observed error rate of 1 in 200,000 packets on the experimental Xerox PARC 3 megabit Ethernet [17]. Our measurements on our local 10 megabit Ethernet indicate an error rate of approximately 1 in 100,000 under normal circumstances. However, when one station transmits at full speed to another workstation, the error rates rise an order of magnitude, to approximately 1 in 10,000. We assume that most of the additional errors are due to failures in the 3-COM Ethernet interface (See also [5,13] for additional evidence of large packet losses in interfaces). We therefore operate somewhere in the region between 10^{-4} and 10^{-5} in Figure 5. In comparing the results for the stop-and-wait protocol and the blast protocol, the key observation to make is that $T_0(D)$ — the error-free transmission time for the blast protocol — is significantly smaller than $D \times T_0(1)$ — the comparable value for the stop-and-wait protocol (See Section 2). Consequently, for low error rates where this term dominates, the blast strategy performs significantly better. While the network error rate allows us to operate in the flat part of the curve, the frequency of errors in the interfaces actually forces us partly into the beginning of the knee of the curve. Nevertheless, the expected time of the blast protocol is still notably better than that of the stop-and-wait protocol.

These results also allow us to make a stronger conclusion: since the expected time for a blast protocol with the crudest retransmission strategy — full retransmission on error and no negative acknowledgement — results in a nearly optimal expected time (for the appropriate range of p_n values), no significant improvements in expected time can be achieved by more sophisticated retransmission strategies. In the next section, we show that such strategies can significantly improve the standard deviation.

[5] The second occurence of $T_0(1)$ in this formula should strictly speaking be replaced by $T_0'(1)$, the elapsed of a *failed* transmission. In practice, the difference is minor and can be subsumed by slightly adjusting the value of T_r.

At this point, an observation is in order about the size of the data transfers used in a blast protocol. Clearly as the size of the data transfer increases, errors are more likely and retransmission becomes more costly. For such very large sizes, we suggest the use of ***multiple blasts***, whereby the transfer is broken up in a number of different blasts, each of which proceeds according to the definition of the blast protocol.

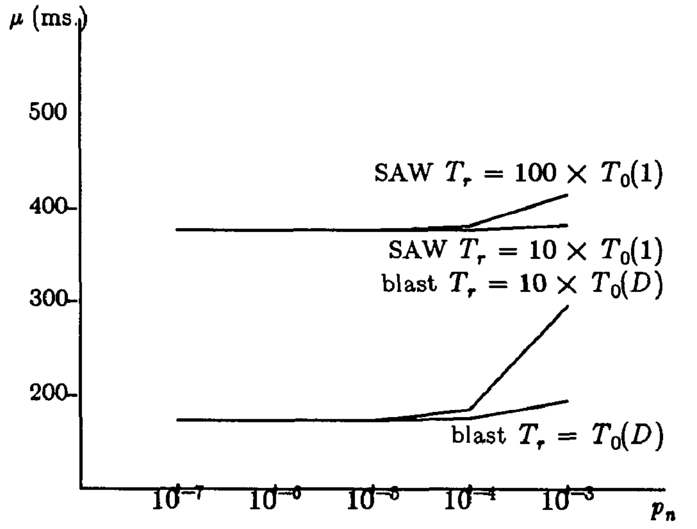

Figure 5: Expected Time for 64 kilobyte Transfers

3.2. Standard Deviation

We now analyze the standard deviation of different retransmission strategies that can be used in conjunction with a blast protocol. We assume that we operate under error conditions such that the expected time of the transfer is nearly identical to the error-free transmission time (i.e. we operate in the flat region of the curves in Figure 6).

Consider a given transmission strategy and denote by $T_0(D)$ the error-free transmission time. Furthermore, let $T_0^k(D)$ be the elapsed time for the k-th transmission attempt, let $T_r^k(D)$ be the interval between the k-th and the (k+1)-th transmission attempt, and finally let $s(i+1)$ be the probability of success on the (i+1)-th transmission attempt. Then, if the transfer succeeds on the (i+1)-th transmission attempt, the total elapsed time for this transfer is

$$\sum_{k=0}^{i} T_0^{k+1}(D) + \sum_{k=0}^{i-1} T_r^{k+1}(D)$$

Assuming we are operating under low error conditions and that thus the expected time is constant and approximately equal to $T_0(D)$, we get for the variance

$$\sigma^2 = \sum_{i=0}^{\infty} [(\sum_{k=0}^{i} T_0^{k+1}(D) + \sum_{k=0}^{i-1} T_r^{k+1}(D))^2 \times s(i+1)] - T_0^2(D)$$

This formula indicates three potentially fruitful avenues for reducing the variance:

(1) Reduce the retransmission intervals $T_r^k(D)$: this can be accomplished either by choosing a small timeout value or by providing a negative acknowledgement when the transfer fails.

(2) Reduce the transmission time $T_0^k(D)$ for retransmissions: this can be done by reducing the number of packets to be sent on retransmission. A negative acknowledgement packet can carry information as to which packets were successfully received.

(3) Reduce the probability of failure of the retransmissions: since we are assuming independent failures, this probability is only dependent on the number of packets transmitted. Thus, here also reducing the number of packets sent has a beneficial effect.

Clearly, a combination of these different approaches is optimal. However, we analyze the different methods in isolation to assess their relative benefits. In particular, we consider the following retransmission strategies:

(1) Full retransmission on error without negative acknowledgement.

(2) Full retransmission on error with a negative acknowledgement after the last packet.

(3) Retransmission from the first packet not received (indicated in a negative acknowledgement)

(4) Selective retransmission of the packets not received (indicated in a negative acknowledgement).

Certain of these retransmission strategies lead themselves to exact analytical evaluation, while others are more easily evaluated by approximation or simulation. In the following, we only present the results of our study. Readers interested in the details of the derivation are referred to [21].

3.2.1. Full Retransmission on Error without Negative Acknowledgement

In the case of full retransmission on error without negative acknowledgement, the standard deviation of the elapsed time is easily shown to be

$$\sigma = (T_0(D) + T_r) \times (p_e^{1/2} \times \frac{(1+p_e)^{1/2}}{1-p_e})$$

$$p_e = 1 - (1 - p_n)^{D+1}$$

This function is set out against p_n for different values of T_r in Figure 6. It is clear from the above formula and from the figure that the value of T_r has a significant effect on σ, even for low error rates.

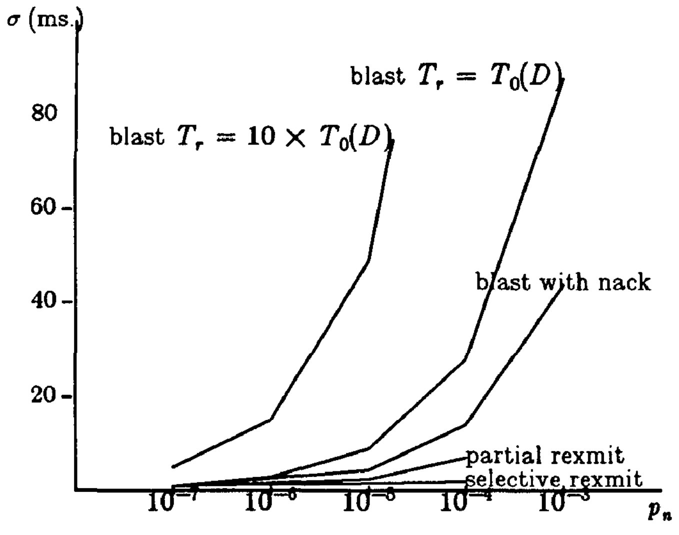

Figure 6: 64 Kilobyte MoveTo: Standard Deviation

3.2.2. Full Retransmission on Error with Negative Acknowledgement

In order to achieve a low standard deviation when using full retransmission, we need to choose T_r small w.r.t. $T_0(D)$. This can be done as shown in Figure 6 by choosing a small value of T_r. Alternatively, a small effective value of T_r can be achieved by using a negative acknowledgement, while still maintaining a much larger physical value of T_r. We use the following strategy:

(1) If the destination receives the last packet, it sends either a positive or a negative acknowledgement depending on whether or not it received all packets in the sequence.

(2) If the sender gets a negative acknowledgement, or if the sender does not receive any acknowledgement within a time interval T_r, it retransmits the whole sequence of packets.

The characteristics of this strategy can be derived by an approximative argument (See [21]). If $p_n \ll \frac{1}{D}$ and $D \gg 1$, then it can be shown that the standard deviation is approximately equal to

$$\sigma \approx T_0(D) \times (p_e^{1/2} \times \frac{(1+p_e)^{1/2}}{1-p_e})$$

This formula indicates that the standard deviation when using full retransmission with a negative acknowledgement is all but independent from the retransmission interval (for low error rates). The values of σ for different values of p_n are set out in Figure 6 for comparison with full retransmission without negative acknowledgement.

3.2.3. Partial and Selective Retransmission

By either choosing a small retransmission interval or by using negative acknowledgements, we have minimized the component of the standard deviation that is dependent on the retransmission interval. The standard deviation is also dependent on the amount of data retransmitted during retransmissions. This component can be minimized using either partial retransmission (from the first packet not received) or by selective retransmission (of the subset of packets not received).

These retransmission strategies are implemented as follows. In order to execute a D-packet transfer, (D-1) packets are transmitted without acknowledgement. The last packet is sent reliably, i.e. it is retransmitted periodically until an acknowledgement is received. The acknowledgement to the last packet indicates which is the first of the D-1 unreliably transmitted packets that was not received (in the case of partial retransmission) or which of the D-1 unreliably transmitted packets did not get to their destination (in the case of selective retransmission). If D' did not get there, they need to be retransmitted using the same method: transmit $D'-1$ packets unreliably and the last packet reliably. This procedure continues until all packets get to their destination.

The standard deviations associated with these retransmission strategies are difficult to derive analytically. We have simulated the procedures by computer and determined both the expected time and the variance from the simulation. Figure 6 shows the standard deviation observed in the simulation.

3.2.4. Standard Deviation: Summary

Figure 6 presents a comparison of the standard deviation of the four retransmission strategies that we consider. Full retransmission without a negative acknowledgement produces unacceptable variations in the elapsed times of the transfers (for realistic retransmission intervals). Use of a negative acknowledgement reduces these variations drastically. Given the presence of such a negative acknowledgement, the extension to partial retransmission starting from the first packet not successfully received is trivial and provides further reduction of the variance. We have chosen not to use selective retransmission, given the additional complexity this introduces in the protocol software and given that the improvement in performance is not very significant. If one were to consider networks with higher error rates or much larger transmissions, selective retransmission might become worthwhile.

4. Related Work

Protocols to support network interprocess communication have been the subject of a number of recent research papers. The protocol supporting the V kernel interprocess communication has been described in [4,6]. Another interesting example appears in Cedar RPC, although this design refrains from using blast protocols for large transmissions [3]. The author first became aware of the name *blast* protocol in conjunction with protocols developed at MIT for downloading screen images from a VAX to an Alto. The idea of blast protocols has been mentioned by various other authors, including Spector who calls it a multi-packet transfer and suggests using an overall software checksum on the entire data segment [18]. Needham mentions the use of a *transmit-and-pray* protocol for file transfers on the Cambridge ring: it is essentially a disk-to-disk (rather than machine-to-machine) blast protocol with full retransmission on error [14].

A large body of work is concerned with the performance of various transmissions strategies such as stop-and-wait, go-back-n and selective retransmission [19]. Most of these analyses assume that the network is a scarce resource, to be shared in an efficient way by a large number of users, and therefore use throughput under high offered load as a measure of performance. The networks studied usually have high error rates (and frequently high latency, such as satellite networks). Although some of them consider the use of cumulative acknowledgements, few consider delaying the acknowledgement altogether until the end of the transmission. Their analysis needs to be reconsidered in a local network environment, where network bandwidth is plentiful, errors are rare, and low delay under low load is more important than high throughput under high load. In fact, most of this work ignores the software cost of generating and receiving the packets, which dominates the transmission cost in a local network environment.

5. Conclusions

The V interkernel protocol has been designed to aggressively take advantage of local network characteristics such as low error rate, high bandwidth and low latency. In order to do so, the software overhead involved in dealing with network interprocess communication must be minimized. As a result, the protocol has been implemented at the network interrupt level and assuming communication partners that are more or less matched in speed. For large data transfers, client buffers are made available prior to the transfer, so that no intermediate copies need to be made.

In such a "tight" implementation of the protocol, we have shown that the overhead of copying data in and out of the network interfaces is a dominating factor in the overall elapsed time. Since blast protocols and sliding window protocols allow these copies to occur in parallel on the source and the destination machine, they perform substantially better than stop-and-wait protocols.

We have also considered the effect of transmission errors on the performance of various protocols. Experimental evidence suggests that network errors are relatively rare, but that interface errors occur more frequently, especially if the devices are driven at full speed. Given the network error rate, it would be acceptable to use full retransmission on error in conjunction with a blast protocol. The frequency of errors in the interface causes such a strategy to have unacceptable variations in the elapsed times. We have argued for a partial retransmission strategy starting from the last packet not received by the destination, since it is simple to implement and not significantly worse than more complicated strategies.

Acknowledgements

The original V kernel protocol was developed by the author at Stanford University under the direction of David Cheriton. I also wish to thank Guy Almes, David Cheriton and Allan Porterfield for their comments on an earlier draft of the paper.

References

[1] 3-COM Corporation, Multibus Ethernet (ME) Controller Model 3C4000, Reference Manual (May 1982).

[2] A. Bechtolsheim, V.R. Pratt and F. Baskett, The SUN Workstation Architecture, Technical Report 229, Computer Systems Laboratory, Stanford University (February 1982).

[3] A.D. Birrell and B.J. Nelson, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59 (February 1984).

[4] D.R. Cheriton, The V Kernel: A Software Base for Distributed Systems, IEEE Software, Vol. 1, No. 2, pp. 19-42 (April 1984).

[5] D.R. Cheriton and P.J. Roy, Performance of the V Storage Server: A Preliminary Report, Proc. of the 1985 ACM Computer Science Conference, pp. 302-308 (March 1985).

[6] D.R. Cheriton and W. Zwaenepoel, The Distributed V Kernel and its Performance for Diskless Workstations, Proc. of the 9th Symposium on Operating System Principles, pp. 129-140 (October 1983).

[7] Communication Machinery Corporation, ENP Family Product Brief (February 1984).

[8] Digital Equipment Corporation, Intel Corporation and Xerox Corporation, The Ethernet: A Local Area Network - Data Link Layer and Physical Layer, Specifications, Version 2.0.

[9] Excelan, EXOS 101 Ethernet Front-End Processor.

[10] E.D. Lazowska, J. Zahorjan, D.R. Cheriton and W. Zwaenepoel, File Access Performance of Diskless Workstations, accepted for publication in ACM Transactions on Computer Systems.

[11] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf, The Architecture of an Integrated Local Network, IEEE Journal on Selected Areas in Communications, Vol. SAC-1, No. 5, pp. 842-857, November 1983.

[12] M.K. McKusick, W.N. Joy, S.J. Leffler and R.S. Fabry, A Fast File System for UNIX, ACM

Transactions on Computer Systems, Vol. 2, No. 3, pp. 181-197 (August 1984).

[13] J. Nabielsky, Interfacing to the 10 Mbps Ethernet: Observations and Conclusions, Proc. '84 ACM SigComm Conference, pp. 124-131 (June 1984).

[14] R.M. Needham, Systems Aspects of the Cambridge Ring, Proceedings of the Seventh ACM Symposium on Operating System Principles, pp. 82-85 (December 1979).

[15] J. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, J.G. Thompson, A Trace-Driven Analysis of the Unix 4.2BSD File System, Technical Report, Computer Science Division, University of California at Berkeley.

[16] P.J. Roy, private communication.

[17] J. Shoch and J.A. Hupp, Measured Performance of an Ethernet Local Network, Communications of the ACM, Vol. 23, No. 12, pp. 711-721 (December 1980).

[18] A.Z. Spector, Multiprocessing Architectures for Local Computer Networks, Technical Report STAN-CS-81-874, Department of Computer Science, Stanford University (August 1981).

[19] A.S. Tanenbaum, Computer Networks, Prentice-Hall, 1981.

[20] C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull and D.R. Boggs, Alto: A personal computer, Computer Structures: Principles and Examples (Eds. D.P. Siewiorek, C.G. Bell and A. Newell), McGraw-Hill, pp. 549-572, 1982.

[21] W. Zwaenepoel, Protocols for Large Data Transfers on Local Networks, Technical Report TR-85-23, Department of Computer Science, Rice University.