Rethinking the TCP Nagle Algorithm

Jeffrey C. Mogul
Compaq Computer Corp. Western Research Lab.
250 University Ave.
Palo Alto, CA 94301
mogul@pa.dec.com

Greg Minshall
Redback Networks
350 Holger Way
San Jose, CA 95134
minshall@redback.com

ABSTRACT

Modern TCP implementations include a mechanism, known as the Nagle algorithm, which prevents the unnecessary transmission of a large number of small packets. This algorithm has proved useful in protecting the Internet against excessive packet loads. However, many applications suffer performance problems as a result of the traditional implementation of the Nagle algorithm. An interaction between the Nagle algorithm and TCP's delayed acknowledgment policy can create an especially severe problem, through a temporary "deadlock." These flaws in the Nagle algorithm have prompted many application implementors to disable it, even in cases where this is neither necessary nor wise.

We categorize the applications that should and should not disable the Nagle algorithm, and we show that for some applications that often disable the Nagle algorithm, equivalent performance can be obtained through an improved implementation of the algorithm. We describe five possible modifications, including one novel proposal, and analyze their performance on benchmark tests. We also describe a receiver-side modification that can help in some circumstances.

1. INTRODUCTION

If an individual user of a shared, large, but finite resource with no explicit limits on consumption, increases his or her demand by $N\%$, he or she stands to gain nearly $N\%$ more of the resource. Yet if all users increase their demands by $N\%$, the total demand may well exceed the carrying capacity of the resource, resulting in little net gain, or even a collapse. This is known as a "tragedy of the commons" [6]. A user's perceived self-interest conflicts with the collective interest of all users, and might even be in conflict with the user's actual self-interest.

The Internet, as we have known it since its inception, is a commons, and many people recognize its vulnerability to a tragedy of the commons. This has led to numerous proposals for technical mechanisms to limit consumption (e.g., admission control), or economic mechanisms to force users to internalize costs. However, none of these mechanisms are in widespread use. Internet protocols are also used in isolated networks (intranets), with the potential for excessive demand, but where administrative or other constraints prevent the use of charging or admission controls.

Fortunately, enlighted self-interest can promote good consumption patterns. The primary such mechanisms now used in the Internet are Jacobson's "slow start" and "congestion avoidance" algorithms for TCP[8]. While the primary motivation for these algorithms was to avoid congestive collapse of a shared network, Jacobson showed that they also improved performance for lengthy TCP connections without competing traffic. That is, for most users, their own self-interest (in employing these algorithms) coincides with the interest of the network as a whole.

Even before Jacobson's work explicitly addressing congestion via feedback mechanisms, several TCP algorithms had been devised to limit the number of unnecessary packets injected into the network. (These can be viewed as open-loop congestion avoidance mechanisms.) In 1984, Nagle showed that a protocol such as Telnet could generate lots of tiny packets, even though the user's needs could be met equally well by sending fewer, larger packets. He also proposed a simple algorithm, for use by the TCP sender, to automatically limit the transmission of unnecessarily small packets[13]. This, now known as the Nagle algorithm (or "Nagle's algorithm"), is a standard requirement for TCP implementations.

The Nagle algorithm applies when a TCP sender is deciding whether to transmit a packet of data over a connection. If it has only a "small" amount of data to send, then the Nagle algorithm says to send the packet only if all previously transmitted data has been acknowledged by the TCP receiver at the other end of the connection. In this situation, "small" is defined as less data than the TCP Maximum Segment Size (MSS) for the connection, the largest amount of data that can be sent in one datagram.

Standard TCP includes another algorithm for limiting the transmission of small packets, dating back to 1982. This is the delayed acknowledgment ("delayed ACK") policy[2], which prevents the TCP receiver from sending too many acknowledgment packets. The traditional receiver implementation delays sending an acknowledgment until it has data to send on the reverse path (allowing it to "piggyback" the ACK on the reverse-path data). However, the specifications require a receiver to generate an acknowledgment at least as often as every second full-sized segment (that is, $2 * MSS$ bytes). A timer, typically set to 200 msec (but allowed to be as high as 500 msec), prevents unlimited delays. The assumption behind the delayed ACK policy is that the sender is transmitting as fast as it can (consistent with flow control and congestion control), and so we can avoid sending a superfluous ACK packet by simply waiting for the next data packet.

Unfortunately, circumstances that occur quite often in practice can lead to a temporary "deadlock" between the sender's Nagle algorithm and the receiver's delayed ACK policy: the Nagle algorithm prevents the sender from transmitting more data until it receives an outstanding ACK, while the delayed ACK policy prevents the receiver from transmitting an ACK until more data arrives. Sooner or later, the delayed-ACK timeout breaks the deadlock, but the result is to add delays on the order of hundreds of msec to operations that should complete much faster. These delays are especially visible on LANs or regional networks.

Because many applications cannot tolerate these delays, TCP implementations provide a means to disable the Nagle algorithm, and many implementors take advantage of this. We will show that, while this is appropriate for a class of applications, disabling the Nagle algorithm is inappropriate for several other classes. That can lead to severe network stress when done by an application with faulty output buffer management. However, because of the well-known potential for deadlock between the Nagle algorithm and the delayed ACK policy, many implementors are forced to disable the Nagle algorithm in circumstances where this should not be required. This creates a risk where the Nagle algorithm, a mechanism to protect the Internet against excess packets sent by buggy applications, is not employed when necessary.

Modifications to the Nagle algorithm have therefore been proposed, with the goal of eliminating this potential temporary deadlock, and therefore making the TCP sender's self-interest consistent with the interest of the entire Internet. In this paper, we evaluate several proposed modifications, including a novel mechanism that directly attacks the deadlock (by treating it as a form of priority inversion). Our evaluations are done in the context of a specific BSD-derived TCP implementation, although the concepts should be applicable to wide range of implementations.

Our goal is to require no changes to existing applications. We therefore would like to find a single algorithm variant that minimizes the likelihood of unnecessary delay, rather than requiring application programmers to choose from a menu of variants, especially if the choice would depend on quantitative parameters of the application or the environment. We show that while all of the proposals have strengths and weaknesses, only the algorithm based on deadlock detection fully eliminates the problem in the benchmark we used (unfortunately, it is not appropriate to every application). We also show that a simple change to the receiver's delayed acknowledgment mechanism can sometimes (not always) improve the situation even with unmodified senders.

Nagle's algorithm was designed in the age of shared low-bandwidth backbone links. One could argue that it is irrelevant to today's Internet, and that floods of small packets are no longer a problem worth solving. This argument fails on at least three grounds: (1) many people connect to the network over wireless links, which usually are both slow and shared; (2) even on fast links, excessive use of small packets makes inefficient use of expensive resources, such as routers; and (3) Nagle's algorithm is a useful firewall against sloppy applications or complex bugs that would otherwise send too many tiny packets. It would be a mistake to stop using it.

2. RELATED WORK

When Nagle first proposed his algorithm, he recognized the importance of analyzing the algorithm for deadlocks, since the algorithm has "no timers," but he apparently failed to realize the possibility of temporary deadlock with the delayed ACK policy. Perhaps this was because delayed ACKs had not been widely implemented at the time, or perhaps the existing applications did not trigger the deadlock.

However, the problem was recognized shortly thereafter. Early examples included the use of multi-byte "function keys" in interactive terminal sessions [19], and the interactive X window system [17]. In both of these cases, implementors quickly realized the need to disable the Nagle algorithm [5]. The BSD series of UNIX systems included a TCP_NODELAY socket option, for this purpose, dating from 4.3BSD (1986).

2.1 Previous analyses of problems with the Nagle algorithm

Crowcroft *et al.*[3], while investigating performance anomalies with a TCP-based RPC system, discovered that the Nagle algorithm interacted badly with the delayed ACK policy. They showed that, for their experiments, the problem could be traced to the way in which the BSD-based network implementation moves data from an application buffer to a socket buffer. In essence, the original implementation suffered from a problem analogous to the TCP "Silly Window Syndrome" (SWS)[2]. They proposed fixing the socket buffer management (in the kernel's sosend() function) mechanism, rather than fixing Nagle's algorithm per se. Most modern BSD-based systems now include the improved sosend() code from Crowcroft *et al.*; unfortunately, problems with the Nagle algorithm persist.

More recently, the introduction in HTTP of "persistent connections" (the use of a single TCP connection for several HTTP requests) has led several researchers to similar conclusions.

Heidemann analyzed persistent-connection HTTP (P-HTTP) [7], and showed that unlike request-per-connection HTTP, P-HTTP servers could suffer from the temporary deadlock problem. He showed that this arose whenever a response message requires an odd number of full-sized TCP segments, plus an additional partial segment. In this case, the receiver will delay its ACK (because it has sent ACKs for an even number of segments, but is holding onto its ACK for the last full-sized segment), and the sender's Nagle algorithm will delay sending the partial segment, because is waiting for an ACK for an earlier segment. Following his terminology, we will refer to this as the "odd-full+short-final-segment" (or OF+SFS) problem. Heidemann solved the problem for his server implementation by disabling the Nagle algorithm.

Nielsen *et al.*, in their analysis of the benefits of request pipelining in HTTP/1.1[15], also noted that the Nagle algorithm could cause delays, without explicitly ascribing this to a deadlock with the delayed ACK policy. They reported disabling the Nagle algorithm in both their server and their client, although they do not explain why it was necessary to disable the Nagle algorithm at the client. (We will consider this issue in section 4.) They explicitly recommended that "HTTP/1.1 implementations that buffer output disable Nagle's algorithm."

2.2 Previous proposed modifications

Minshall[10] observed that a simple modification to the Nagle algorithm should solve the OF+SFS problem: the sender delays *only* if it is waiting for acknowledgment of data that was sent in a short ($< 1MSS$) packet (but not if all unacknowledged packets were full-sized)[1]. He also observed that it might be necessary to disable the Nagle algorithm for protocols such as P-HTTP, when using pipelining.

Minshall *et al.* reported on measurements comparing a simplified implementation of Minshall's proposed modification[11]. They found that it often performed better than the original Nagle algorithm, but the results were equivocal. Also, they did not carefully analyze the performance of the modified algorithm over a wide range of packet sizes. As we shall show, the algorithm behaves poorly in some cases, particularly when the application buffer is smaller than the MSS. Minshall *et al.* stressed that application implementors should, whenever possible, use appropriate buffering mechanisms to mitigate TCP performance problems.

[1] We believe that a version of this modification was first proposed by David Mosberger[12], although Mosberger proposed testing the size only of the most recently sent unacknowledged packet.

2.3 The Linux TCP_CORK option

In BSD-based UNIX systems, the only control over the TCP sending policy explicitly available to application programs is the TCP_NODELAY socket option, which simply disables the Nagle algorithm. Linux has provided an additional socket option, TCP_CORK, which operates as a sort of "super-Nagle" mode. While this option is set, the Linux TCP will never send any segments smaller than the MSS. When the application disables TCP_CORK, all remaining pending data is sent, subject to the normal Nagle-algorithm restrictions. While the traditional Nagle algorithm will send a partial-size segment on an idle connection, Linux will not, if TCP_CORK is set. The TCP_CORK option was added to allow an application to send a message using several distinct system calls, without sending unnecessarily small packets. This option does not, as far as we know, provide any solution to the OF+SFS problem for unmodified applications.

3. INTERACTION BETWEEN THE NAGLE AND DELAYED-ACK ALGORITHMS

As we described earlier, the delays usually attributed to the Nagle algorithm are in fact the result of a temporary deadlock between the TCP sender's Nagle algorithm, and the TCP receiver's delayed-ACK policy. (The "deadlock" is temporary, because the receiver cannot delay its acknowledgment for more than 500 msec, and BSD-based systems limit this to 200 msec.) Heidemann's observation was that this deadlock occurs only when transmitting a message requiring an odd number of full-sized TCP segments, plus an additional partial segment (what we are calling the OF+SFS problem.)

The interaction is actually somewhat more complex than this, at least in BSD-based systems, because of several aspects of the TCP implementation.

We show several examples to illustrate the interaction. In these figures, time runs from top to bottom. The "client" host is shown on the left, broken down into an application ("App") and the TCP sender. The server appears on the right; only the TCP receiver is shown here. Blocks of message data are shown as rectangles labelled with letters A, B, C, etc. Although TCP sequence numbers count bytes, TCP congestion control (including the Nagle algorithm) counts in terms of MSS-length segments, and the length of a "1 MSS" segment is shown in each figure. Time is shown both in units of round-trip times (RTTs) and with additional terms. We assume that the sender's initial congestion window, cwnd, is $2 * MSS$, which is commonly the case after the client's SYN has been acknowledged (if the congestion window were larger, we would simply need to show larger application buffers in these figures). We also assume a high enough network bandwidth that its contribution to the delay may be ignored.

Figure 1 shows how a successful (non-delayed) transfer might take place. The application constructs a buffer (A, B, C, D) of exactly $4 * MSS$ bytes, and hands it to the TCP stack in one operation. (In a BSD system, this involves copying the data into a socket buffer.) At time "0 RTT," the TCP sender transmits the first two segments (because cwnd is $2 * MSS$). The receiver immediately acknowledges these two segments. After approximately 1 RTT, the sender receives the ACK and sends the remaining two segments. The receiving (server) application then processes the request data, and returns a response, which arrives at time $2 * RTT + server\_time$.

Now consider figure 2, showing a Nagle-delayed transfer. Here, the application buffer handed to the TCP sender is just slightly shorter than $4 * MSS$ bytes. Again, at time "0 RTT," the TCP sender transmits the first two segments (because cwnd is $2*MSS$), and the receiver immediately acknowledges these two segments. At time "1 RTT" in this example, however, the sender sends full-sized segment C, but defers sending partial segment D because it has not yet received an ACK for segment C. Meanwhile, the delayed-ACK policy at the receiver defers its ACK for C in the hope that a full-sized D will soon arrive. After up to 200 msec, the receiver gives up and ACKs C, which allows the sender to transmit D. The response is received at time $3*RTT+ack\_timeout+server\_time$, which can be up to $RTT+200msec$ later than it was for the *longer* transfer in figure 1.

Note that in the scenario of figure 2, the transmission of segments C and D was delayed until the receipt of an acknowledgment, under the assumption that cwnd starts at $2 * MSS$. Other scenarios, not limited by the congestion window, can also cause Nagle-related delays, at least in the BSD implementation. This requires an short explanation of the BSD socket buffering mechanism (covered in more detail by Crowcroft *et al.*[3]).

When an application uses a system call (such as write()) to send a buffer on a TCP stream, the kernel invokes the sosend() function, which copies bytes from the user buffer to the socket buffer and then calls the tcp_output() function. If the user buffer is larger than a page "cluster" (MCLBYTES), sosend() copies it in chunks of length MCLBYTES, calling tcp_output() each time. The tcp_output() function also runs upon receipt of an acknowledgment or a timeout. Each time it is called, tcp_output() sends as many packets as it can, consistent with the TCP sending rules.

The formal Nagle algorithm defers transmission of a small segment unless the sender has no unacknowledged data outstanding (i.e., the sender is "idle"). However, the BSD tcp_output() function has a subtle quirk: it might send more than one packet per invocation, but it only tests for the idle state once per invocation. Figure 3 shows an abstracted version of tcp_output().

Since sosend() calls tcp_output() once for every MCLBYTES of data, this quirk means that messages shorter than $min(cwnd, MCLBYTES)$ will not be delayed by the Nagle algorithm if sent on an idle connection. (The formal Nagle algorithm might delay a final partially-full packet of such a message, since by the time that packet is sent, the connection could be awaiting an ACK.) This makes it somewhat difficult to predict exactly when the Nagle algorithm will cause delays, since both cwnd and the "idleness" of a connection are dynamic values.

3.1 Buffer tearing

Many buffered applications use power-of-two buffer sizes, yet typical maximum transmission units (MTUs) are not powers of two, and so typical MSS values are relatively prime with typical application write-buffer sizes. Heidemann's analysis [7] showed that small packets could be generated when sending a message of length $bufsize + epsilon$, where $bufsize = 2^N$ and $epsilon < MSS$.

For example, the Apache Web server uses $bufsize = 4096$ (regardless of the system page size); on an Ethernet, TCP would use an MSS of 1460 bytes. If Apache sends a message of length 4100 bytes, the first 4096-byte buffer is sent as a sequence of $(1460, 1460, 1176)$ bytes, and the remaining 4 bytes are then buffered to be sent. The 1176-byte packet is not delayed (because the first two full-sized segments generate an immediate acknowledgement), but it is not immediately acknowledged itself. This means that the Nagle algorithm will delay transmission of the final 4-byte segment.

We call this problem "buffer tearing": a message that could have been sent using a minimal number of TCP segments is torn into smaller pieces by a buffering mechanism. While sometimes the ex-

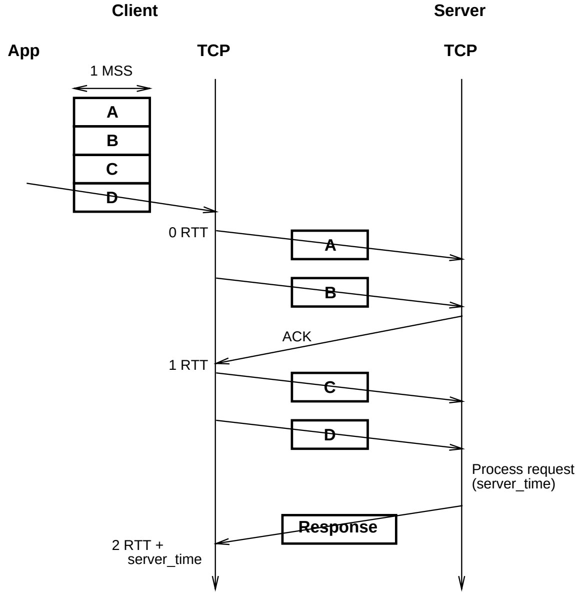


Figure 1: Example of successful transfer

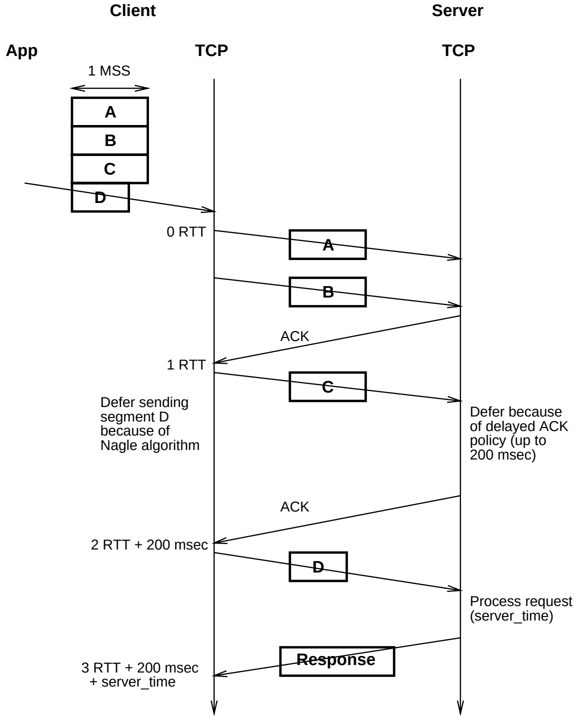


Figure 2: Example of Nagle-delayed transfer

This is a heavily abstracted rendition of the BSD tcp_output() function, showing only the features relevant to the Nagle algorithm, and omitting all error checks.

```
tcp_output(struct tcpcb *tp) /* input is TCP control block */
{    int idle, off, win, len, emptying;

    /* We're idle if everything we have sent has been ACKed */
    idle = (tp->snd_max == tp->snd_una);

again:
    off = tp->snd_nxt - tp->snd_una;  /* offset of first unsent byte in buffer */
    win = min(tp->snd_wnd, tp->snd_cwnd);  /* min of flow control & cong. windows */
    len = min(so->so_snd.sb_cc, win) - off;    /* number of unsent bytes in buffer */

    if (len >= tp->t_maxseg)
       goto send;    /* always send if we have a full segment */

    /* emptying = sending this would empty the output buffer */
    emptying = ((len + off) >= so->so_snd.sb_cc);
    if ((idle || tp->t_flags & TF_NODELAY) && emptying)
        goto send;    /* Nagle's algorithm */
    if (len >= tp->max_sndwnd / 2)
        goto send;    /* send if at least 1/2 of receiver's window */
    if (SEQ_LT(tp->snd_nxt, tp->snd_max))
        goto send;    /* always send if retransmitting lost data */

    ... other tests for whether to send a packet ...

    return(0);

send:
    ... fill in headers and call ip_output() ...
    goto again;
}
```

Figure 3: Pseudocode: relevant aspects of BSD's tcp_output()

tra packets cause just a minor inefficiency (for example, a message length of $5 * 1460 = 7300$ bytes is sent as 6 packets), in many cases buffer tearing can lead to OF+SFS delays.

Buffer tearing can also occur if the application buffer is large enough to hold the entire message, but the socket output buffer is not. Many older systems had very small default TCP output buffer sizes, and practical considerations limit the feasible output buffer sizes for systems such as busy Web servers, because these buffers consume physical memory (whereas application buffers normally consume pageable virtual memory). Semke *et al.*[18] suggest that the kernel could automatically tune a connection's socket output buffer size, with a target value of about twice the $bandwidth * delay$ product for the connection. This target might be much less than the message size, especially for low-bandwidth or low-delay networks, yet it is on low-delay networks that one most notices an additional 200 msec delay.

The problem analyzed by Crowcroft *et al.*[3] is an example of buffer tearing, in which sosend() has N bytes to write, and N is both larger than the amount of space remaining in the socket buffer and smaller than the MSS. Their solution (defer sending in this situation) is helpful, but not a general solution to the buffer-tearing problem.

3.2 Phase-locking effects

In the traditional BSD implementation, the delayed-ACK timeout is not a 200-msec interval starting from the point when an ACK is deferred; rather, it is simply the next “tick” of a free-running 200-msec clock. One might therefore expect the excess delay from the interaction between the Nagle algorithm and the delayed-ACK policy to be uniformly distributed between 0 and 200 msec.

Minshall *et al.*[11] observed, however, that repeated operations between the same pair of systems, when subject to the OF+SFS problem, lead to a sort of phase-locking between the sending application and the receiver's 200 msec clock tick.

Consider an application sending a series of requests, for each of which the response would normally arrive within 5 msec (i.e., where the RTT is small). If a response is delayed by the OF+SFS problem, it will arrive about 1 RTT after a tick of the receiver's 200 msec clock. If the very next response is delayed, it will wait until the next clock tick, and will suffer almost the worst-case 200 msec delay. Even if only 1 out of every 10 responses is delayed, the total execution time for all 10 responses will be 200 msec, and the delay experienced by the unlucky response will be close to 150 msec. This explains why relatively simple experiments can suffer from non-uniform excess delays.

Minshall *et al.* speculated that this kind of phase-locking could be coupled between busy Internet Web servers by busy proxy servers. If so, that could create synchronized bursts of packets flowing through the Internet. We do not know if anyone has observed this mass synchronization in practice.

3.3 Prevalence of OF+SFS messages in Web traffic

Minshall *et al.*[11] used a benchmark based on Usenet news, and reported that 8.7% of the responses in that benchmark required an even number of FDDI packets (although presumably very few were an exact multiple of the FDDI MTU). These are the responses that would probably suffer from OF+SFS delays. However, Minshall *et al.* used a synthetic benchmark; how important is the OF+SFS problem in real traffic flows?

We chose a widely-used HTTP traffic trace, from Digital Equipment Corporation[4], to answer this question. For reasons of time and disk space, we selected only the busiest one-day trace segment from the DEC traces, containing 1,565,207 responses.

Because of the idle-connection quirk in the BSD TCP stack, as described earlier, we adopted a somewhat more complex test for whether a response is OF+SFS-vulnerable. We assumed that both the application buffer and socket output buffer were large enough to hold the entire message. Our model predicts that a message would be delayed if its length is between $(2 * N + 1) * MSS$ and $(2 * N + 2) * MSS$, exclusive, for $N >= 0$. However, it also predicts that messages shorter than $2 * MSS$ are never delayed, and that messages shorter than MCLBYTES are never delayed. We modelled several options for MSS (Ethernet, 1460 bytes, and FDDI, 4312 bytes) and for MCLBYTES (4096 and 8192 bytes, and one case where this test is ignored).

	MCLBYTES ignored	MCLBYTES = 4096	MCLBYTES = 8196
MSS = 1460	18.78%	18.78%	10.05%
MSS = 4312	6.71%	6.71%	6.71%

Entries show the fraction of responses vulnerable to OF+SFS

Table 1: Prevalence of OF+SFS-vulnerable response lengths in an HTTP trace

Table 1 shows the results of this simple study. Depending on our assumptions for MSS and MCLBYTES, our model predicts between 6.7% and 18.8% of the responses in this trace could be delayed by the OF+SFS problem. (For the FDDI MSS of 4312 bytes, $2 * MSS$ is larger than all of the modelled values for MCLBYTES, so the latter test is irrelevant.)

4. WHEN SHOULD ONE USE THE NAGLE ALGORITHM?

In Nagle's original description of the algorithm, he stated that it works for all kinds of TCP connections[13]. However, as mentioned in section 2, people quickly realized that there are applications in which the possibility of 200 msec delays cannot be tolerated. The conventional wisdom is, therefore, that applications should disable the Nagle algorithm if they seem to be experiencing this kind of delay.

Recall that the Nagle algorithm was devised to protect the network from congestion by lots of small packets. We believe that the protection afforded by the algorithm is still valuable, especially in the face of applications with suboptimal buffer management, and therefore it is necessary to have a clearer understanding of when the Nagle algorithm should and should not be used.

Our analysis looks at four cases where TCP is used:

1. **One-way bulk data transfer**: This is the paradigmatic use of TCP, in which a large quantity of data is transferred in one direction, with little or no data transferred on the reverse path. For example, an FTP data connection.

2. **Telnet-style two-way data transfer**: In the Telnet protocol, characters typed by the user are sent as quickly as possible to the remote host. In some instances, the host echos the typed characters back to the user; the echo may also take place local to the user's system. Command output is sent from the host to the user, via the same connection.

3. **RPC-style exchanges**: Several protocols use TCP for request-response exchanges, with the client waiting for the server's response before it sends another request. Examples include the client-server mode of NNTP[9] (the USENET News protocol), and traditional SMTP[16].

4. **Pipelined exchanges in soft-realtime applications**: Although TCP clearly is not well suited to real-time applications, because it converts packet losses into delays on the

order of a second or longer, it can still be usable for interactive applications. These applications expect short messages, in either direction, to be delivered as quickly as possible. Examples include the X window system [17], NFS over TCP, and P-HTTP, all of which successfully run over TCP. A pipelined exchange is similar to an RPC-style exchange, except that the client does not wait to receive an entire response before it sends the next request(s). In the case of NFS, at least, the server might not even reply to the requests in order. Pipelining is an especially effective way of hiding long latencies, but it depends on being able to start processing one operation before another is finished.

In a perfect world, the Nagle algorithm would work well in all of these cases. Nagle's original description of his algorithm showed that it works well for cases (1) and (2), and a decade of subsequent practice has born that out (but see section 4.1 for a complication).

Cases (3) and (4), however, have proved problematic. We show in this paper that it is possible to fix the Nagle algorithm for case (3). That is, minor modifications to the Nagle algorithm provide near-optimal performance for case (3) without contradicting the intention behind the algorithm, minimization of the number of packets sent. However, it is not possible to "fix" the algorithm for case (4), because the desired behavior here is antithetical to the original intent of the Nagle algorithm: the application wants a small request or response message to be sent as soon as possible, even if the other end has not replied recently.

4.1 An aside: multiple-byte keystrokes in Telnet

We should point out that it is not entirely clear how to classify the problem of multi-byte function keys. Consider a Telnet client, with keystrokes coming from a serial-line keyboard. The user types a function key that issues a multiple-byte sequence; the Telnet client program sees the resulting bytes separated by enough time (e.g., on a 9600 baud terminal, by about 1 millisecond) that it sends the first byte before seeing the second. However, the receiver's delayed-ACK policy defers sending an ACK (nowhere near two full segments having arrived, and there is nothing to echo until the full sequence has been received by the server Telnet program), and so when the Telnet client program sees and sends the second byte, the Nagle algorithm defers transmission (about to send a small packet, but waiting for an ACK). Stevens[19] used this as an example of why it can be necessary to disable the Nagle algorithm; this would put multiple-byte keystrokes in case (4).

One could argue that a clever Telnet client could buffer its input for a few milliseconds, looking for the first byte of multi-byte sequences, and thus sending such sequences as single packets; i.e., this is actually case (2). Such a small delay would not be perceptible to the user, although it would add complexity to the Telnet client implementation.

5. PROPOSED SOLUTIONS

Given the significant delays induced by the OF+SFS problem, many implementors have been led to disable the Nagle algorithm. This can lead to excessive transmissions of short packets, if the application is poorly designed (or if it has an undetected bug)[11]. We would like to find a solution to the OF+SFS problem that does not give up the protection provided by the Nagle algorithm.

In this section, we describe a number of previously proposed and new solutions. In section 8, we will show how each one performs.

5.1 The Minshall variant

Minshall[10] proposed a simple modification to Nagle's algorithm. Rather than delaying the transmission of a small packet if there is any unacknowledged data, Minshall's variant delays only if "any previously transmitted less than full-sized packet" has not been acknowledged. In theory, this should allow a single "short final segment" to be transmitted without delay. Minshall's suggested implementation adds two (32-bit) sequence number variables to the sender's per-connection TCP state. We refer the reader to [10] for a description of this implementation.

5.2 The MSMV variant

Minshall *et al.*[11] subsequently proposed a slightly different variant, which we refer to as MSMV (after the authors' initials). This variant requires only one bit of additional per-connection state, instead of 64 bits. Also, it allows transmission without delay of two small packets in a row, if the first was caused by tearing a buffer at a multiple of MCLBYTES. The implementation, described in detail in [11], uses a new TF_SMALL_PREV flag to remember whether the most recently transmitted packet was small (less than the MSS). MSMV defers the transmission of a short packet only if TF_SMALL_PREV is set; that is, it bypasses the Nagle algorithm if the previous packet was full-sized. Note that if the socket buffer size is a multiple of MCLBYTES, the TF_SMALL_PREV flag is not set even when a short packet is sent, because MSMV tries not to punish an application for short packets caused by buffer tearing.

5.3 The EOM variant

Both the Minshall and MSMV variants attempt to indirectly prevent the delay of the last part of a large message buffer. For this paper, we experimented with a new variant, called EOM (for "End of Message"). In this variant, the sosend() function explicitly informs tcp_output() whether it has reached the end of a large message, by setting a new TF_EOM flag bit. The tcp_output() function will not defer the transmission of a short segment if EOM is set, because this could lead to a deadlock-like delay. If EOM is not set, meaning that there is more data in the application's buffer, the usual Nagle algorithm applies.

Specifically, the TF_EOM is set if (1) sosend() has exhausted either the application buffer or the socket output buffer, and (2) this call to sosend() has previously called tcp_output() at least once. (Remember that because of the way that tcp_output() checks the connection's idle status, if sosend() can deliver the entire application buffer in one call to tcp_output(), the Nagle algorithm will not be invoked.)

5.4 The MORE variant

If the socket output buffer is too small to hold the entire message, sosend() will call tcp_output() before the application's buffer is exhausted; this can cause buffer tearing. We attempt to resolve this problem by defining a new TF_MORE flag, which is set by sosend() if the socket buffer is to small to hold all of the application's buffer. If the TF_MORE flag is set, tcp_output() will *not* send a short segment, even if the Nagle algorithm allows it, because we know that as soon as an acknowledgment arrives to free up buffer space, sosend() will provide additional data, and that might give us the chance to send a full-sized segment.

The MORE variant in itself is not a solution to the OF+SFS problem, but it can be combined with EOM to avoid tearing. In this variant (EOM+MORE), if the TF_MORE flag is set on entry to sosend(), and sosend() exhausts the application's buffer, it will set the TF_EOM flag even if it is calling tcp_output() only once. The net effect of EOM+MORE is that, no matter how many times sosend() is called during the transmission of a single application buffer, tcp_output() will generate a sequence of full-sized segments, followed if necessary by an undelayed short final segment. Source code for EOM+MORE is in figure 4

5.5 The Borman variant

The combination of EOM+MORE appears similar to an unpublished algorithm apparently used in BSD/OS, partially described by David Borman in a message to a mailing list[1]. Borman's algorithm "does the Nagle decision once on the whole chunk of data written in a single write by the application." That is, it remembers whether the connection was idle when the application issued its write() system call (i.e., on entry to sosend()), and if so, it does not delay the final segment. However, if the connection was waiting for an acknowledgment on entry to sosend(), the normal Nagle algorithm applies.

We implemented this algorithm based on Borman's description, without examining the BSD/OS code. In our implementation, sosend() on entry sets a local `idleOnEntry` flag if the connection is idle. When sosend() is ready to call tcp_output(), if `idleOnEntry` is set, and if it has exhausted the application buffer, then it disables the Nagle algorithm, by setting the connection's TF_NODELAY flag. This flag is cleared on every entry to sosend(), so an application cannot push out more than one successive short segment before an acknowledgment arrives.

We also found it necessary to set the TF_NODELAY flag (at the end of the application buffer) if the TF_MORE flag was set on entry to sosend(). Otherwise, the user's buffer might be chopped into several pieces because the socket buffer is not large enough to hold the entire message. Source code for our implementation of Borman's variant is in figure 5; one advantage of this variant is that it requires no extra per-connection state.

5.6 The DLDET variant

We have already observed that because the OF+SFS problem most directly afflicts request-response interactions, it can be viewed as a form of deadlock. So, one way to frame its solution is as a deadlock-detection problem. We propose a new variant, named DLDET, which follows this approach.

If one squints a little, one could also view this as a bounded priority inversion. The Nagle algorithm has decided to make the transmission of the final request segment into a low-priority task, and the application (the high-priority task) is about to block until that low-priority task (and its consequent response transmission) are complete.

Priority inversion is often solved using priority inheritance, and we can apply the same idea. If a process is trying to read from a TCP socket and is about to block for lack of data, we can first check to see if the sending side of the same socket is currently blocked by the Nagle algorithm. If so, we can force out a remaining short final segment (if any), which should cause the block-for-read period to be as short as possible.

This can be implemented by simply calling tcp_output(), with the TF_NODELAY flag temporarily set, just before blocking in the soreceive() function. Similarly, the sosend() function, which could block if the socket output buffer is full, can do the same thing. (A somewhat more efficient implementation would check to see if a short final segment is pending output, and an implementation that respects the BSD layering model would involve additional complexity.)

Unfortunately, this simple method does not always work. Because the sender's congestion window might not allow data in the output socket buffer to be sent immediately, at the time that soreceive() or sosend() needs to block, the sender might still be holding far more than one pending segment (i.e., not because of the Nagle algorithm). The decision in tcp_output() whether to send a short final segment might be made *after* the application has blocked in soreceive() or sosend(). We resolve this by having tcp_output() test the SB_WAIT flags set when a process is blocked on a socket buffer, and releasing a short final segment if either flag is set.

One subtle issue remains: The Nagle algorithm was originally designed to prevent a Telnet client from sending each keystroke in a separate packet, but the DLDET variant as described so far will violate this requirement for a multi-threaded Telnet client. Such an application uses one thread to write keystrokes to a TCP connection, and another thread (or process) to read from the connection. The problem arises because if the receiving thread is always blocked, the TCP sender will always detect a potential "deadlock," and so will always send single-byte segments.

The solution is to keep an extra one-bit per-connection counter, incremented each time the receiving thread tests for deadlock or blocks. The complete DLDET algorithm then sends a short segment only if the counter is non-zero, and then decrements the counter. With this mechanism, we must also use the TF_MORE flag, to prevent buffer tearing in sosend() from causing the counter to be decremented. This prevents a multi-threaded application from sending a flood of packets, but it avoids the OF+SFS problem for single-threaded applications, or for any application using a large enough application buffer size. See figure 6.

5.6.1 Limitations on DLDET

Unfortunately, our DLDET solution does not work for event-driven applications, which manage many sockets using the select() or poll() system calls. (This is a popular structure for Internet server applications, and many others.) When an application blocks in select(), it might be waiting for events on hundreds or thousands of TCP sockets, and it seems infeasible to prod each one of these in case it is deferring the transmission of a short final segment. Also, when an application blocks in select(), it does not set the SB_WAIT flag for any of these sockets.

The DLDET solution also fails for applications that send requests on one socket but receive the corresponding responses on another. While this is not a common structure for communicating processes, we are not sure if it is unimportant.

Finally, the counter-based DLDET algorithm might not prevent the OF+SFS problem for a multithreaded application when the receiving thread has blocked, and then the sending thread transmits a long message using more than one write() system call. In this case, buffer tearing might lead to the transmission of multiple short segments, but the counter would only allow one to be sent without delay.

One might be able to solve these restrictions on DLDET by providing applications with an explicit system call to mark the ends of their messages, as Minshall suggested[10]. This is roughly what Linux's TCP_CORK option achieves using twice as many system calls. Careful use of such a call might give an application direct control over when segments are sent (while still observing congestion and flow control), although at the cost of additional application complexity. We have not explored this possibility in detail.

6. BENCHMARK METHODOLOGY

To evaluate all of the proposed variations on the Nagle algorithm, we needed a benchmark that could create all of the various conditions that might lead to the OF+SFS problem. We considered using a real application as a benchmark, because many of the details of TCP performance depending on timing. For example, an ap-

In sosend() the `outcount` *variable is initialized to 0, and the following code is inserted before the call to tcp_output():*

```
    tp->t_flags &=  TF_EOM;
    /* Set EOM if no more user data and we previously sent a full MCLBYTES
     * from this user buffer, or if we ran out of socket buffer space on
     * previous send. (outcount appears superfluous, but would be necessary if
     * we were to delete the MORE code from EOM.) */
    if (((resid == 0) || (space <= 0)) && ((outcount > 0) || (tp->t_flags & TF_MORE)))
        tp->t_flags |= TF_EOM;
    outcount++; tp->t_flags &=  TF_MORE;
    /* Set MORE (& clear EOM) if we have more user data to send */
    if (resid) {
        tp->t_flags |= TF_MORE; tp->t_flags &=  TF_EOM;
    }
```

In tcp_output(), the Nagle algorithm test becomes:

```
   if ((idle || tp->t_flags & TF_NODELAY || tp->t_flags & TF_EOM ) &&
        ((tp->t_flags & TF_MORE) == 0) && (len + off >= so->so_snd.sb_cc))
      goto send;
```

Figure 4: Implementation of EOM and MORE

At entry to sosend(), these local variables are set:

```
    idleOnEntry = (tp->snd_max == tp->snd_una);
    moreOnEntry = (tp->t_flags & TF_MORE);
```

At the same point, ensure that special treatment does not last past end of previous buffer:

```
    tp->t_flags &=  TF_NODELAY;
```

The following code is inserted before the call to tcp_output(), to avoid delaying the last segment of this buffer:

```
    if ((idleOnEntry || moreOnEntry) && (resid == 0)) )
        tp->t_flags |= TF_NODELAY;
```

Note: this code excerpt omits conditionals to allow the original Nagle algorithm to be selected instead of Borman's variant.

Figure 5: Implementation of Borman's variant

In soreceive(), before calling sosbwait() to block a receiving thread, and in sosend(), before blockin a sending thread, insert:

```
  if (so->so_proto->pr_usrreq == tcp_usrreq) tcp_dldet(so, "snd");
```

For reasons of space, this code lacks locking and error-handling:

```
void tcp_dldet(struct socket *so) {
    struct inpcb *inp = sotoinpcb(so); struct tcpcb *tp = intotcpcb(inp);
    int off, win, len;

    if ((tp->t_flags & TF_DLDET) == 0 || tp->t_flags & TF_NODELAY) return;

    off = tp->snd_nxt - tp->snd_una; win = min(tp->snd_wnd, tp->snd_cwnd);
    len = min(so->so_snd.sb_cc, win) - off;
    if (len < tp->t_maxseg) { /* output might be delayed by Nagle alg. */
        tp->t_flags |= (TF_NODELAY|TF_DLDETCTR);
        tcp_output(tp);
        tp->t_flags &=  TF_NODELAY;
    }
    tp->t_flags |= TF_DLDETCTR;   /* we are about to block */
}
```

In tcp_output(), after the Nagle test:

```
if ((tp->t_flags & TF_DLDETCTR) && ((len + off) >= so->so_snd.sb_cc) &&
    ((so->so_rcv.sb_flags & SB_WAIT) || (so->so_snd.sb_flags & SB_WAIT))){
    tp->t_flags &=  TF_DELDETCTR; goto send;
}
```

Figure 6: DLDET implementation

plication might exploit the parallelism created by an earlier packet transmission, by using the time to read more data from the disk.

However, we could not find a real application that generates sufficiently repeatable results, and whose operation can be controlled enough to assure us that we were testing as much of the parameter space as possible. Instead, we wrote a relatively simple synthetic client-server benchmark (called *naglemark*), which allows us to vary these parameters: (1) the response message length, either as a series of fixed increments, or as an unordered set of random values; (2) the server application write() buffer size; (3) the server output socket buffer size; and (4) the number of requests per connection. When the program is invoked with multiple requests per connection and a randomized response message length, the length can be chosen either once per request, or once for all the requests. The program reports the elapsed time, measured at the client, from the time at which the request is sent to the time when the last response byte is received.

We modified the server's kernel (Compaq's Tru64 UNIX V4.0F, formerly known as Digital UNIX) to support all of the Nagle algorithm variants described in section 5, as well as several combinations of these. The kernel allows an application to choose the variant (or combination) on a per-socket basis, so we can test each variant on an otherwise identical kernel. This avoids the possibility that relinking the kernel will cause different instruction cache behavior; on the other hand, it does result in longer code paths, since the modified functions must test several flags to choose the appropriate variant.

For each set of parameters, we ran a large number of trials, either using randomly chosen response lengths or a sequential series. One would hope the response time to be a smooth line, whose slope is the underlying network bandwidth, as shown in figure 7. On a system using the traditional Nagle algorithm, however, the elapsed time varies erratically with length, as shown in figure 8[2]. (Each of these two trials was done using 1000 requests, in order of increasing length, on a single connection, with application buffer and socket buffer sizes both larger than the largest message size.)

Since we are comparing a large set of algorithm variants, we would like to have a simple characterization of the difference between figures 7 and 8. We do this by computing a score that counts the fraction of samples with elapsed time much larger than necessary. Our simplistic estimate of the necessary transfer time for given message length is the length divided by the LAN's raw transfer rate, plus an arbitrary 10 msec for startup costs. We then set, also somewhat arbitrarily, a limit of three times this estimate, and count any sample above this cutoff as being delayed.

For example, every point in the data set in figure 7 is above the estimated elapsed time (the lower straight line), but below the cutoff of three times the estimate (the upper line). This data set, therefore, has a score of 0.0%. However, many of the points in the data set in figure 8 are above the cutoff line, so this data set has a score of 30.5%. The much higher score for the traditional Nagle algorithm is indicative of its unstable performance.

This simplistic scoring method clearly has its limits. Figure 8 shows that the scores may be too low for larger message lengths (clearly many of the samples below the cutoff line are actually delayed). For short message sizes, experimental noise (such as application scheduling delays) can push some samples, especially for shorter message lengths, over the cutoff. However, the scores do seem to correlate with subjective analysis of the full data sets.

To determine the sensitivity of our scoring system to our arbitrary cutoff of three times the estimated necessary latency, we recalculated the scores (in the following sections) using multipliers of two and four. When the cutoff is set at twice the estimated latency, scores are generally worse (as expected, since this sets a tougher standard), but the relative rankings of algorithm variants are essentially unchanged. (In some cases where the rankings were near-ties, they remained near-ties but in a different order.) When the cutoff is set at four times the estimated latency, scores for Ethernet trials are generally better, but again without significantly changing the relative rankings of the variants. Changing the multiplier from three to four makes almost no difference for the scores in many of the FDDI trials, and reduces them in others; again, the relative rankings of the variants remains unchanged.

7. EXPERIMENTAL SETUP

We ran our benchmark program under a wide variety of parameters. For each trial, we measured 1000 transfers of varying length. We tried varying the length both sequentially, from 100 bytes to 100,000 bytes in steps of 100 bytes, or with a uniform random distribution between 1 and 100,000 bytes. We also tried running each of the 1000 transfers in a separate TCP connection, or all 1000 serially over a single connection. Table 2 shows how we refer to these trial types using single-letter abbreviations.

We varied the Nagle algorithm variant (the ten choices shown in table 3). We used three choices for application buffer size (app_buf_size): 4KBytes, 32KBytes, and 128KBytes. The first corresponds to Apache's buffer size, while the last is large enough to hold any of our response lengths. We also used three choices for the server's socket output buffer: 16KBytes, 64KBytes, and 128Kbytes. The first is a typical default for many systems; the second is the largest available without the TCP Window Scale option; the third is large enough to hold any of our response lengths.

We ran our experiments between a pair of hosts directly connected either via an Ethernet (10Mbit/sec, $MSS = 1460$) or via an FDDI LAN (100Mbit/sec, $MSS = 4312$). The FDDI LAN was used only by these two hosts; the Ethernet was shared with a modest level of traffic from other sources (normally averaging under 8 packets/sec during a typical daytime hour, but a few of our trials apparently coincided with significantly higher loads.)

By using all possible combinations of these choices, for each network configuration we ran $2*2*10*3*3 = 360$ different 1000-transfer trials. We then repeated the entire experiment N times. Rather than presenting the scores for all $324 * N$ trials, for each combination of algorithm variant and buffer sizes we show only the worst-case scores (averaged over all N repetitions) from the parameter set (s, S, r, R). We then rank each algorithm according to its worst-case score across all parameter choices. We believe that this is an objective ranking of how reliably the algorithm avoids delays caused by the Nagle algorithm.

In these experiments, the end hosts were running Tru64 UNIX V4.0F. The server host was modified to support each of the variants in table 3, at the application's choice. The client host ran the same kernel, although the client always used the traditional Nagle algorithm (since it never had to send data while waiting for an acknowledgment). The client was modified to defeat a special variant of the delayed acknowledgment policy peculiar to Tru64 UNIX (see section 8.1), so as to better represent a traditional BSD system. On this system, $MCLBYTES = 8192$. The receiver buffer size was 32KBytes, for all trials. During the tests, no other application programs were running on either host.

8. RESULTS

Tables 4 and 5 summarize the scores for Ethernet and FDDI, respectively. In each table, the first column names the algorithm variant used. The next nine columns include all possible com-

[2] We're not sure why figure 8 shows a lot of samples at 100 msec; this might be a consequence of the 500 msec timer that BSD-based TCP implementations use for retransmissions and several other purposes.

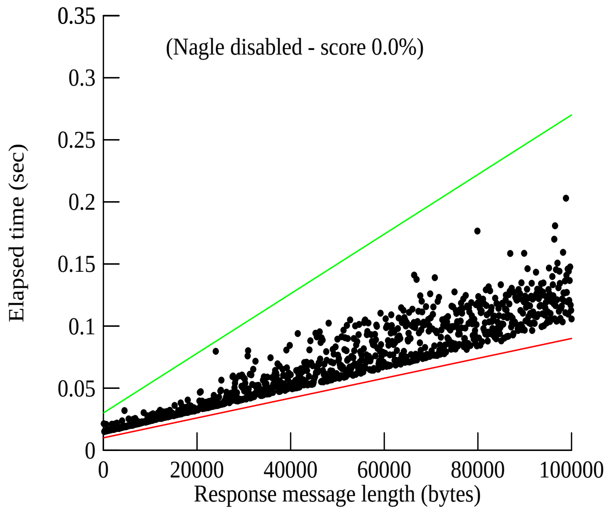


Figure 7: Good naglemark performance

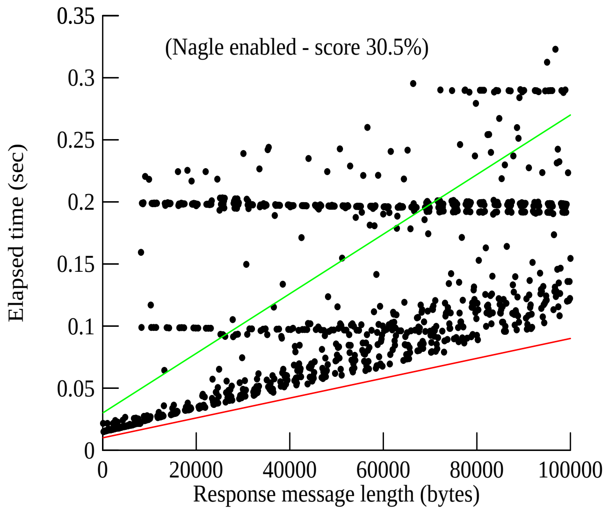


Figure 8: Poor naglemark performance

Connections	Sequential lengths	Random lengths
One per request	s	r
One for all requests	S	R

Table 2: Nomenclature for trial types

Nagle algorithm variant	Nomenclature
Traditional Nagle algorithm	NAGLE
Nagle algorithm disabled	OFF
Minshall variant[10]	MINSHALL
Minshall *et al.*[11] variant	MSMV
EOM variant	EOM
EOM and MORE together	E+M
MINSHALL and MORE together	MIN+MORE
DLDET	DLDET
EOM, MORE, and MSMV	E+M+M
EOM, MORE, and DLDET	E+M+D
Borman variant[1]	BORMAN

Table 3: Nomenclature for algorithms

binations of three application buffer sizes and three socket buffer sizes. In each table entry for these nine columns, we give the worst (highest) score, expressed as percentage, for the given combination of algorithm and buffer sizes, over all four trial types. (Remember that a “score,” as defined in section 6, is the fraction of message lengths for which the measured elapsed time exceeds the threshold of three times the estimated elapsed time.) In each entry, the worst-case trial type is indicated by the letter (as defined in table 2); this information may be of interest to those wishing to replicate our results.

The final column in each table is the worst score for an algorithm variant over all of the possible combinations; the table is sorted in order of increasing score, so the best algorithm is listed first.

These experiments showed relatively little variation between trials. We computed the standard deviations of the scores for each table entry. For the Ethernet experiments, only a few entries show standard deviations above 4%; several of these appear to be the result of periods of heavy use of the shared Ethernet. For the FDDI network, which is private, none of the standard deviations exceeded 3%, and only six entries had standard deviations above 2%.

We used all four trial types because we hypothesized that the order in which the individual transfers occurred would affect the results. Indeed, for any given combination of buffer sizes and algorithm variant, there was often a large variation in scores among the four trial types. (The largest variation, for Ethernet, was for the MSMV algorithm with a 4K application buffer and a 64K socket buffer: this scored 1.2% with random message lengths and one request per connection, but scored 12.6% with sequential message lengths using a single connection for all requests.) However, if the results in tables 4 and 5 had reported the best-scoring trial type, instead of the worst-scoring trial type, the algorithm rankings would not have changed significantly. In other words, our hypothesis (that the measurements would depend on transfer order and number of connections) was quantitatively correct, but it has no significant qualitative effect on the choice between algorithm variants.

The results in tables 4 and 5 show significant differences between the algorithm variants. However, it requires some additional analysis to understand the strengths and weaknesses of each variant.

- **Minshall variant**: The Minshall variant does well in almost all of our trials, except when the application buffer size (app_buf_size) is smaller than the MSS and the message length is larger than app_buf_size (see figure 9). The application thus does two or more write() calls. The first one generates an immediate transmission of a small segment (because the connection is idle). The data from second write() must then wait, because it is $< MSS$, the sender is waiting for an ACK, and the receiver is waiting for a second full segment. (We believe that the high score for one entry in this row is the result of Ethernet cross-traffic.)

  However, once the message length reaches $app\_buf\_size + 2 * MSS$, the sender stops waiting (because after sending the first packet with app_buf_size, it can now send at least two full-sized segments, and so the receiver acknowledges

	4K Application buffer			**32K Application buffer**			**128K Application buffer**			
Sockbuf size:	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	
Variant										**Worst**
MIN+MORE	S 0.0%	R 0.5%	R 0.4%	S 0.0%	R 0.7%	R 0.8%	S 0.0%	R 1.4%	R 1.4%	1.4%
E+M+D	S 0.0%	R 0.5%	R 0.4%	S 0.0%	R 0.6%	R 0.5%	S 0.0%	R 1.4%	R 1.7%	1.7%
OFF	S 0.0%	R 0.5%	R 0.4%	R 0.3%	R 0.6%	R 0.6%	S 0.0%	R 1.5%	R 1.7%	1.7%
DLDET	S 0.0%	R 0.4%	R 0.5%	S 0.0%	R 0.5%	s 1.1%	S 0.3%	R 1.9%	R 1.6%	1.9%
MINSHALL	s 0.3%	r 1.1%	R 0.5%	S 0.0%	R 1.5%	r 2.6%	s 1.1%	R 1.6%	R 1.6%	2.6%
MSMV	S 17.9%	S 12.6%	S 10.1%	S 2.5%	S 1.4%	R 0.6%	S 2.1%	s 1.6%	R 1.5%	17.9%
E+M+M	S 18.2%	S 12.7%	S 10.0%	S 2.9%	R 1.4%	R 0.7%	S 0.0%	R 1.4%	R 1.7%	18.2%
EOM	S 30.9%	S 31.6%	S 31.9%	s 7.6%	S 7.2%	S 7.2%	s 2.3%	s 2.3%	R 2.7%	31.9%
BORMAN	S 31.5%	S 32.3%	S 32.1%	S 7.8%	S 17.6%	S 18.4%	S 0.0%	R 1.6%	R 1.4%	32.3%
NAGLE	S 31.1%	S 32.4%	S 32.4%	S 28.3%	S 29.6%	S 28.8%	S 30.0%	S 30.8%	S 30.4%	32.4%
E+M	S 31.2%	S 32.7%	S 33.4%	s 7.6%	S 7.3%	S 6.9%	s 2.3%	s 3.6%	s 2.3%	33.4%

Each entry is mean of 10 repetitions of 1000 message lengths ($N = 10$)

Table 4: Scores for Ethernet ($MSS = 1460$)

	4K Application buffer			**32K Application buffer**			**128K Application buffer**			
Sockbuf size:	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	
Variant										**Worst**
DLDET	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	0.0%
E+M+D	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	0.0%
OFF	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	0.0%
MIN+MORE	S 8.7%	S 8.7%	S 8.7%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	8.7%
MINSHALL	S 8.8%	S 8.7%	S 8.7%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	8.8%
E+M+M	S 34.5%	S 34.5%	R 21.7%	S 10.2%	S 6.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	34.5%
MSMV	R 34.6%	R 34.3%	S 21.6%	S 10.3%	S 5.9%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	34.6%
BORMAN	S 66.4%	R 56.9%	S 51.5%	S 22.2%	S 38.9%	S 34.6%	S 0.0%	S 0.0%	S 0.0%	66.4%
E+M	S 66.4%	S 56.0%	R 51.9%	S 14.1%	S 14.1%	S 14.2%	S 0.0%	S 0.0%	S 0.0%	66.4%
NAGLE	S 66.4%	S 56.0%	S 51.6%	S 43.8%	S 44.5%	S 44.6%	S 44.1%	S 44.5%	S 44.5%	66.4%
EOM	R 66.6%	R 56.0%	S 51.6%	S 14.1%	S 14.1%	S 14.2%	S 0.0%	S 0.0%	S 0.0%	66.6%

Each entry is mean of 10 repetitions of 1000 message lengths ($N = 10$)

Table 5: Scores for FDDI ($MSS = 4312$)

immediately).

- **MSMV variant**:

  The MSMV variant does much worse than the Minshall variant, although significantly better than the original Nagle algorithm. MSMV is quite prone to buffer tearing, and only performs well when both the application buffer and socket buffer are larger than the message length. The algorithm does include a simple test to ignore tearing if the socket buffer is larger than MCLBYTES, but apparently this only works for messages smaller than twice the socket buffer size (see figure 10).

  While MSMV does not perform as well as the Minshall variant, it delays messages far less often than does the original Nagle algorithm, and requires far less per-connection state than the Minshall variant.

- **EOM variant**:

  The EOM variant, in which sosend() tells tcp_output() that it has delivered the end of a large message, performs identically to the original Nagle algorithm if either app_buf_size or the message length is less than MCLBYTES. (This is a somewhat arbitrary design decision; we could have chosen a lower threshold.)

  For larger messages, it generally works well, except for some buffer tearing when the message is just slightly longer than the socket buffer (see figure 11). In this case, sosend() delivers a socket buffer's worth of data to tcp_output(), marked as "end of message," and tcp_output() sends the entire buffer without delay. This generates a short segment at the end of the buffer (because the buffer size is not a multiple of MSS). If the remainder of the message is shorter than the MSS, the Nagle algorithm will defer sending it, because the second invocation of sosend() will not see enough data to mark the remainder as "end of message."

- **EOM+MORE**: We tried combining the EOM variant with the MORE variant, to see if this would avoid some delays, by reducing the number of times that tearing in sosend() on MCLBYTES boundaries causes tcp_output() to send a small segment. The results suggest that EOM+MORE is no improvement over EOM, in terms of the likelihood that segments will be delayed.

- **EOM+MORE+MSMV**: We also tried combining all three of EOM, MORE, and MSMV. This seems to be no improvement over MSMV.

- **MINSHALL+MORE**: This seems to provide no significant improvement over MINSHALL in the FDDI experiments. In the Ethernet experiments, it appears to outperform MINSHALL, but we suspect that the reason for MINSHALL's poorer ranking may simply be the result of an outlier trial (i.e., cross traffic on the shared Ethernet). Otherwise, the differences between these variants are smaller than the standard deviations for the MINSHALL experiments.

- **Borman variant**: The Borman variant was designed to avoid delays when the application buffer is large enough to hold the entire message; in our tests, it succeeds at this goal. However, if the application buffer is too small, requiring the message to be sent using more than one write(), this variant does not seem to improve on the traditional Nagle algorithm. (Nor was it intended to).

  The results do show some apparent improvements for 32K-byte application buffers and 16K socket buffers. This is a consequence of our decision to include the MORE variant in our implementation of the Borman algorithm, which prevents the application from suffering delay if its write() buffer is larger than the socket buffer. So, for application buffer sizes between the socket buffer size and the total message size, our implementation of the Borman variant does avoid delays that the traditional Nagle algorithm would impose.

- **DLDET variant**: The DLDET variant, on our benchmark, performs nearly perfectly. (The very small fraction of samples exceeding the target elapsed time, for larger values of the socket buffer size, might be related to Ethernet congestion.) Combining DLDET with EOM+MORE seems to be no improvement over DLDET by itself, although this is hardly surprising.

We conclude by noting that if we ignore the cases where the application buffer size is smaller than the message length, all of the algorithm variants have near-perfect scores, except for the traditional Nagle algorithm and for MSMV (which is still prone to buffer tearing if the socket buffer is smaller than the message length). This reinforces the recommendation in [11] that applications should use sufficiently large buffers whenever possible. Space does not permit us to show the actual scores for this version of the results.

8.1 Effect of algorithms on throughput

So far, we have evaluated the various algorithm variants based on the frequency with which they excessively delay packets. When the algorithm sends packets without delay, however, it still might not be making optimal use of TCP. Therefore, we also evaluated each algorithm variant's throughput under no-delay conditions, if possible.

For each trial, we computed a linear regression of the elapsed time versus the message length. If the correlation coefficient was above a threshold, we report the algorithm variant's mean cost in terms of nanoseconds/byte; this is the slope of the regression line. The slope gives the transfer efficiency for the algorithm variant, and can be used as a way to detect variants that send a excessive number of packets. However, we have not yet tried to directly measure the number of packets transmitted in each trial.

The results are inconclusive, because we have no foolproof way of eliminating from the linear regression every sample that experienced a Nagle-related delay. There seems to be no clear difference in slopes, but the choice of algorithm could have small throughput effects that are not distinguishable using this analysis. In the interest of space we omit detailed results. We are currently looking for another approach to measuring the efficiency of these algorithm variants.

9. MODIFYING THE DELAYED-ACK POLICY

The OF+SFS problem is the result of an interaction between the sender's Nagle algorithm and the receiver's delayed-ACK policy. Although we have concentrated on fixing the Nagle algorithm, to dissuade application implementors from disabling it entirely, we briefly consider the possibility of modifying the delayed-ACK policy as well. Because one cannot always control the operating system on both the sender and receiver, it might be necessary to attack the OF+SFS problem from both sides.

The delayed-ACK policy can create 200 msec delays even without help from the Nagle algorithm, when either the sender or receiver has a socket buffer smaller than $2 * MSS$. This is not an uncommon situation; many older systems default to socket buffer sizes of 4096, 8192, or 16384 bytes, and an application might explicitly request such a small buffer size. However, FDDI LANs allow an MSS of 4312 bytes (so $2 * MSS = 8624$, larger than an 8192-byte buffer), and some Gigabit Ethernet hardware supports non-standard “jumbo frames” with an MSS of 8960 bytes (so $2 * MSS = 17920$, larger than a 16384-byte buffer).

If either the sender or receiver buffer is smaller than $2 * MSS$, the receiver will never be able to acknowledge two full-sized segments. Therefore, the traditional implementation of the delayed-ACK policy will always wait for a 200 msec timeout, unless it has data to send in the reverse direction. On an FDDI network, using 8192-byte buffers, this scenario can result in a TCP throughput of 21,560 bytes/sec, over a link with a raw capacity of 12.5 Mbytes/sec!

Because of these problems, several proposals have been made to modify the delayed-ACK policy:

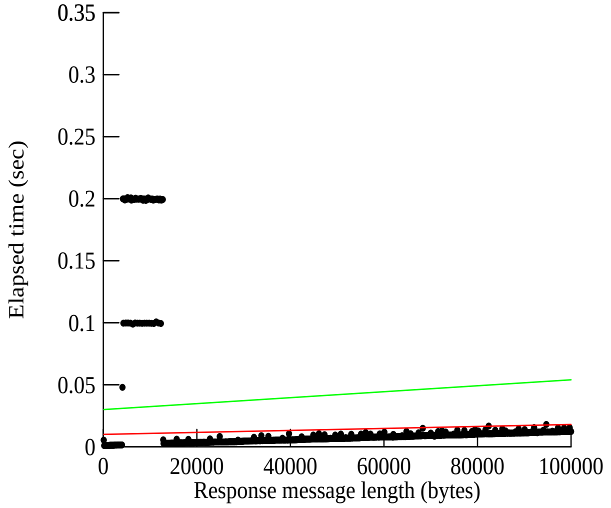
- **Decrease the timeout**:

  The 200 msec timer is an arbitrary choice, based originally on an estimate of the inter-arrival time for segments on the ARPAnet; however, the original proposal for delayed acknowledgments suggests using an adaptive policy[2]. Heidemann[7] repeats this suggestion, although there is no evidence that anyone has actually implemented it yet. This approach would require the receiver to maintain a timer with a much finer resolution than those used in existing BSD implementations, however, and the overhead of doing this for a large number of connections on a fast LAN might be prohibitive.

- **Delay only in the presence of reverse-path data**: Nagle has suggested [14] that acknowledgments should normally be given for every full-sized segment, rather than for every second segment, unless data is also flowing in the opposite direction (i.e., the host R that is deciding whether to delay an acknowledgment is also sending data on the connection back to host S). In this case, if host R is about to send data and the last packet it sent was an ACK-only packet, it should increment a counter; when this counter reaches a threshold, the receiver enables the traditional delayed-ACK policy, but zeros the counter if the delayed-ACK timer goes off.

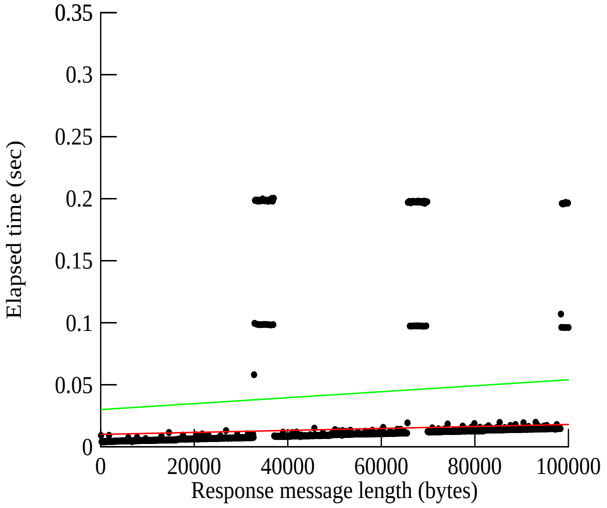
  The underlying insight here is that the 200 msec delayed-ACK timeout is an indication to the receiver that it has made a mistake in delaying the ACK, while sending an ACK-only packet just before sending a data packet is an indication that the receiver mistakenly failed to delay. The solution is intended to adaptively avoid making either of these mistakes. It remains untested, as far as we know.

- **Infer the use of a small buffer**: If the receiver knows that the sender is using a buffer smaller than $2 * MSS$, it could avoid delaying acknowledgments in this case. Since TCP does not explicitly communicate the sender's buffer size, the receiver would have to infer this value. (The receiver's own



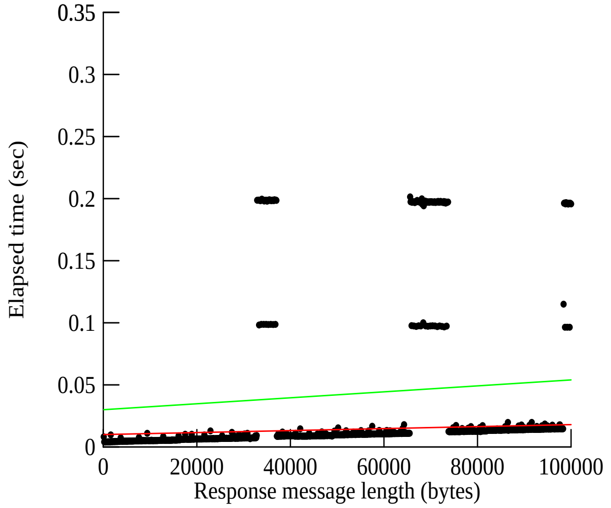
application buffer = 4KB, socket buffer = 16KB, MSS = 4312

Figure 9: Minshall performance



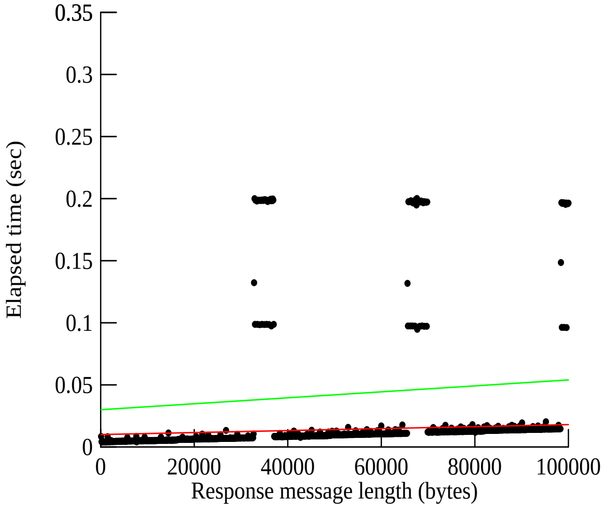
application buffer = 32KB, socket buffer = 16KB, MSS = 4312

Figure 10: MSMV performance



application buffer = 32KB, socket buffer = 16KB, MSS = 4312

Figure 11: EOM performance



application buffer = 32KB, socket buffer = 16KB, MSS = 4312

Figure 12: EOM+MORE+MSMV performance

buffer size might also be smaller than $2*MSS$, but obviously this value is known to the receiver.)

In fact, the Tru64 UNIX TCP implementation we used for our experiments does infer the use of a small sender buffer. (We are not aware of other systems that do this.) We discovered this after finding it surprisingly difficult to trigger the delays characteristic of the OF+SFS problem. The implementation of this inference is quite simple: in the TCP receiving path, the code tracks the largest difference between the sender's sequence number and the receiver's own acknowledgment number. If this value is at least one MSS, and an acknowledgment would advance the advertised window by at least half of the value, then tcp_output() sends an acknowledgment that it would otherwise delay.

If the sender's buffer really is less than $2 * MSS$, this mechanism never delays the acknowledgment of a full-sized segment. However, it can underestimate the size of a larger buffer, if the sender is not aggressively using a large congestion window. Therefore, it can generate non-delayed acknowledgments even with a large sender buffer, especially at the beginning of a connection when `cwnd` is small.

This hides the OF+SFS problem, albeit non-deterministically. Therefore, to conduct the experiments reported in section 8, we modified the client's Tru64 UNIX kernel to defeat this mechanism.

We also ran a set of trials with the inference mechanism enabled. The results for Ethernet and FDDI are shown in tables 6 and 7, respectively. In these tests, the results do vary a lot between repetitions, especially for the variants that scored poorly when the inference mechanism is disabled. This appears to be caused by the inherent non-determinism in the inference algorithm. For the Ethernet experiments, many of the entries have standard deviations above 5%, and a few are worse than 8%; For the FDDI experiments, many entries have standard deviations ranging from 10% to almost 20% (although only three entries have standard deviations between 3.9% and 10.3%). These large variations, especially in the FDDI experiments on a private network, are clearly due to the algorithms and not to cross-traffic.

Even with the large variances, a comparison of these tables against tables 4 and 5 reveals that, in general, the inference algorithm avoids much of the delay associated with the Nagle algorithm. However, the inference algorithm actually worsens the

	4K Application buffer			**32K Application buffer**			**128K Application buffer**			
Sockbuf size:	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	
Variant										**Worst**
E+M+D	S 0.0%	R 0.6%	R 0.6%	S 0.0%	R 0.7%	R 0.7%	S 0.0%	R 1.4%	R 1.4%	1.4%
DLDET	S 0.0%	R 0.5%	R 0.5%	S 0.0%	R 0.6%	R 0.6%	S 0.0%	R 1.5%	R 1.6%	1.6%
MINSHALL	S 0.0%	R 0.4%	R 0.5%	S 0.0%	R 0.7%	R 0.6%	S 0.0%	R 1.6%	R 1.5%	1.6%
OFF	S 0.0%	R 0.5%	R 0.5%	S 0.0%	R 0.7%	R 1.0%	s 0.7%	R 1.6%	R 1.6%	1.6%
MIN+MORE	R 0.5%	R 0.6%	R 0.3%	S 0.0%	R 0.6%	R 0.6%	S 0.0%	R 1.4%	R 2.0%	2.0%
E+M+M	R 6.6%	R 7.0%	R 5.7%	S 2.0%	R 1.3%	R 0.7%	S 0.0%	R 1.4%	R 1.5%	7.0%
MSMV	R 8.0%	R 7.4%	S 6.6%	S 1.8%	R 1.1%	R 0.5%	S 0.0%	R 1.3%	R 1.6%	8.0%
E+M	R 19.1%	R 17.3%	R 18.6%	R 3.7%	R 5.1%	S 4.2%	S 0.0%	R 1.7%	R 1.4%	19.1%
BORMAN	R 17.7%	R 16.3%	R 19.5%	S 4.8%	S 9.1%	R 9.9%	S 0.0%	R 1.3%	R 1.5%	19.5%
EOM	R 17.1%	R 20.6%	R 17.7%	S 3.9%	S 4.5%	R 4.3%	S 0.0%	R 2.3%	R 1.6%	20.6%
NAGLE	R 20.8%	R 18.7%	R 16.2%	R 18.5%	R 16.5%	R 17.7%	R 15.1%	R 18.2%	S 19.1%	20.8%

Each entry is mean of 10 repetitions of 1000 message lengths ($N = 10$)

Table 6: Scores for Ethernet ($MSS = 1460$), receiver inferring sender buffer size

	4K Application buffer			**32K Application buffer**			**128K Application buffer**			
Sockbuf size:	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	**16K**	**64K**	**128K**	
Variant										**Worst**
DLDET	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	0.0%
OFF	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	0.0%
E+M+D	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	s 0.1%	S 0.0%	S 0.0%	0.1%
MINSHALL	s 4.6%	R 4.8%	S 4.4%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	4.8%
MIN+MORE	S 4.6%	R 5.3%	S 4.4%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	5.3%
EOM	R 20.9%	S 18.1%	R 18.4%	S 10.0%	S 10.3%	S 9.8%	S 0.0%	S 0.0%	S 0.0%	20.9%
BORMAN	R 21.6%	R 18.1%	S 15.2%	S 13.0%	S 14.0%	S 16.5%	S 0.0%	S 0.0%	S 0.0%	21.6%
NAGLE	R 17.6%	R 21.8%	S 16.6%	R 14.9%	S 15.5%	R 20.4%	R 21.6%	R 14.8%	R 19.3%	21.8%
E+M	R 14.5%	R 15.7%	R 24.3%	S 11.2%	S 11.6%	S 10.0%	S 0.0%	S 0.0%	S 0.0%	24.3%
MSMV	R 51.9%	R 50.8%	S 33.3%	S 4.7%	S 1.3%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	51.9%
E+M+M	R 52.5%	S 51.1%	S 34.0%	S 4.5%	S 4.6%	S 0.0%	S 0.0%	S 0.0%	S 0.0%	52.5%

Each entry is mean of 10 repetitions of 1000 message lengths ($N = 10$)

Table 7: Scores for FDDI ($MSS = 4312$), receiver inferring sender buffer size

performance of the MSMV variant when $app\_buf\_size < MSS$ (and deterministically; these entries have small standard deviations).

For example, when $app\_buf\_size = 4096$ and $msglength = 12289 = app\_buf\_size * 3 + 1$, on FDDI MSMV immediately sends the first 4096-byte packet, then once sosend() has copied two more application buffers to the socket buffer, tcp_output() sends a 4312-byte packet (a full segment) and then a 3880-byte packet (since the previous packet was full-sized). The standard delayed-ACK policy waits until the third packet and then immediately acknowledges it. However, with buffer-length inference, the receiver acknowledges the second (full-length segment), and then must delay its acknowledgment of the third (short) segment. This prevents the sender from transmitting the final (1-byte) segment of the message, because it is waiting for an acknowledgment of a short segment.

Also, while combining EOM and MORE showed no particular advantage over simple EOM when the inference algorithm was disabled, with the inference algorithm enabled, EOM+MORE does significantly better than EOM in some cases (e.g., FDDI, and $app\_buf\_size = 4096$). In other cases (e.g., Ethernet, and $app\_buf\_size = 4096$) EOM+MORE yields mean scores similar to EOM, but with a much larger variance. Both effects seem to be related to the somewhat non-deterministic point at which the receiver realizes that the sender's socket buffer size is indeed larger than $2 * MSS$.

10. FUTURE WORK

The results in section 8 imply that a combination of the DLDET and Minshall variants, or possibly of the DLDET and MSMV variants, might work well in almost every case. However, there are a few hard cases that require more study, especially multi-threaded applications using relatively small application buffers.

Our work so far has concentrated entirely on TCP implementations derived from BSD. We showed that the details of the OF+SFS problem depend on the way that tcp_output() tests for an idle connection, and the buffer-management policy imposed by sosend(). Several widely used operating systems have TCP implementations with different pedigrees and buffer-management frameworks, and we have not yet characterized their vulnerability to the OF+SFS problem. A brief examination of the Linux source code, for example, suggests that it will behave differently from BSD in this respect.

As we indicated in section 8.1, it might also be useful to modify the TCP delayed acknowledgment policy. Based on our simple experiments with the buffer-size inference mechanism, we suspect this could be an important area for further work.

The EOM variant, as described in this paper, only applies when the application writes more than MCLBYTES at once. Since the value of MCLBYTES is system-dependent, this is an arbitrary threshold. It might make sense to set a lower threshold, which should reduce the frequency of delays. Picking the appropriate threshold will require additional study. All of the algorithms, in fact, should be tested on systems with different values of MCL-

BYTES.

All of the experiments reported in this paper used a LAN, and so the RTTs were negligible. In many Internet applications, RTTs are on the order of 100 msec or more; this almost certainly would have an effect both on the dynamics of the algorithm variants we tested, and on the relative significance of the 200 msec delays associated with the original Nagle algorithm. We would like to conduct experiments in a WAN environment, but the logistics of doing so (and especially of controlling for other causes of delay variation) will be challenging.

11. SUMMARY AND CONCLUSIONS

We have explored in detail the interaction between TCP's Nagle algorithm and delayed acknowledgment policy, in the context of the popular BSD-based implementation. This interaction can lead to lengthy delays, especially in request-response interactions using certain message lengths; this is the OF+SFS problem.

We experimentally evaluated a variety of solutions for the OF+SFS problem, including several that have not been proposed before. We showed that DLDET, an approach based on explicit deadlock detection, works well in contexts where an application explicitly sends and receives on the same TCP connection (although it may not always work as well for some multi-threaded applications.) We also showed that the algorithm variant proposed by Minshall works well in most cases, but not for small application buffer sizes. We showed that a number of other variants, while they use less per-connection state than the Minshall variant and they do improve upon the original Nagle algorithm, still lead to delays in many situations; some of these variants may be amenable to further improvement.

We also showed that some minor modifications to the receiver's delayed acknowledgment mechanism can reduce the likelihood of excessive delays.

We confirmed that the use of an application buffer smaller than both the message size and MCLBYTES can lead to the OF+SFS problem; applications should use large buffers for write() system calls whenever possible.

None of our modifications go against the original intention of the Nagle algorithm, which was to avoid a flood of many short TCP packets. If the appropriate modifications were to be made to widely deployed TCP implementations, we could then recommend to implementors that there are almost no circumstances in which it is to their benefit to disable the Nagle algorithm, and the protection that it affords to the network.

12. ACKNOWLEDGMENTS

We would like thank several members of the Western Research Lab for their help: Glenn Cooper, Drew Kramer, and Jason Wold for network and system support, and Marc Viredaz for LaTeXwizardry. We would also like to thank the anonymous reviewers for *CCR* and from several conferences, who while rejecting earlier versions of this paper gave us plenty of good suggestions.

13. REFERENCES

[1] D. Borman. Re: internet draft on suggested mod to the Nagle algorithm. Message to TCP-IMPL mailing list, http://tcp-impl.grc.nasa.gov/tcp-impl/list/archive/1498.html, Feb. 1999.

[2] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, Internet Engineering Task Force, July 1982.

[3] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is Layering Harmful? *IEEE Network*, 6(1):20–24, January 1992.

[4] Digital Equipment Corporation. Digital's Web Proxy Traces. ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html, 1996.

[5] J. Gettys. Personal communication. 1999.

[6] G. Hardin. The Tragedy of the Commons. *Science*, 162:1243–1248, 1968.

[7] J. Heidemann. Performance interactions between P-HTTP and TCP implementations. *ACM Computer Communication Review*, 27(2):65–73, Apr. 1997.

[8] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314–32, Aug. 1988.

[9] B. Kantor and P. Lapsley. Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News. RFC 977, Internet Engineering Task Force, Feb. 1986.

[10] G. Minshall. A Suggested Modification to Nagle's Algorithm. Internet-Draft draft-minshall-nagle-01, Internet Engineering Task Force, June 1999. This is a work in progress.

[11] G. Minshall, Y. Saito, J. C. Mogul, and B. Verghese. Application Performance Pitfalls and TCP's Nagle Algorithm. In *Proc. 2nd Workshop on Internet Server Performance*, Atlanta, GA, May 1999.

[12] D. Mosberger. Re: question about nagle algorithm. Message to TCP-IMPL mailing list, http://tcp-impl.grc.nasa.gov/tcp-impl/list/archive/0961.html, Feb. 1998.

[13] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Internet Engineering Task Force, Jan. 1984.

[14] J. Nagle. Personal communication., Feb. 1999.

[15] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proc. SIGCOMM '97*, pages 155–166, Cannes, France, Sept. 1997.

[16] J. B. Postel. Simple mail transfer protocol. RFC 821, Internet Engineering Task Force, Aug. 1982.

[17] R. Scheifler and J. Gettys. The X Window System. *ACM Trans. on Graphics*, 5(2):79–109, April 1986.

[18] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. In *Proc. SIGCOMM 1998*, pages 315–323, Vancouver, BC, September 1998.

[19] W. Stevens. *TCP/IP Illustrated Volume 1*. Addison-Wesley, Reading, MA, 1994.