Application performance pitfalls and TCP's Nagle algorithm

Greg Minshall minshall@siara.com
Siara Systems

Yasushi Saito yasushi@cs.washington.edu
Department of Computer Science and Engineering
University of Washington

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

Ben Verghese verghese@pa.dec.com
Compaq Computer Corp. Western Research Lab.

Abstract

Performance improvements to networked applications can have unintended consequences. In a study of the performance of the Network News Transport Protocol (NNTP), the initial results suggested it would be useful to disable TCP's Nagle algorithm for this application. Doing so significantly improved latencies. However, closer observation revealed that with the Nagle algorithm disabled, the application was transmitting an order of magnitude more packets. We found that proper application buffer management significantly improves performance, but that the Nagle algorithm still slightly increases mean latency. We suggest that modifying the Nagle algorithm would eliminate this cost.

1 Introduction

The performance of a client/server application depends on the appropriate use of the transport protocol over which it runs. In today's Internet, most application protocols run over the Transport Control Protocol (TCP) [12] or the User Datagram Protocol (UDP) [11].

Usenet, one of the main Internet applications, runs over TCP using an application protocol known as the Network News Transport Protocol (NNTP) [7]. Several of us made a prior study [15] assessing the performance of a popular NNTP implementation. We measured the response time for NNTP interactions, and noticed a sharp peak in the response-time distribution at exactly 200ms. We discovered that this 200ms delay was a consequence of interactions between the application and two TCP mechanisms: the *Nagle algorithm* and delayed acknowledgments.

In this paper, we analyze this situation in more detail. We show that, once we resolved several buffering problems, use of the Nagle algorithm still leads to a slight increase in mean latency for NNTP interactions, and we analyze the mechanism behind this. Finally, we describe a simple modification to the Nagle algorithm that should eliminate the problem.

2 TCP and the Nagle Algorithm

TCP controls the transmission of data between the client and server processes. It attempts to balance a number of requirements, including fast responsiveness (low latency), high throughput, protection against network congestion, and observing the receiver's buffer limits. As part of this balancing act, TCP attempts to inject the fewest possible packets into the network, mainly to avoid congesting the network, and to avoid adding load to routers and switches.

In the early 1980s, terminal (telnet) traffic constituted a large amount of the traffic on the Internet. Often a client would send a sequence of short packets, each containing a single keystroke, to the server. At the time, these short packets placed a significant load on Internet, especially given the relatively slow links, routers, and servers.

To alleviate this glut of small packets, in 1984 John Nagle proposed what has become known as the *Nagle algorithm* [9]. The 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.

The effect of the Nagle algorithm is that at most one "small" packet will be transmitted on a given connection per round trip time (RTT). Here, "round trip time" means the time it takes to transmit data and subsequently receive the acknowledgment for that data.

The algorithm imposes no limit on the number of large packets transmitted within one round trip time.

The Nagle algorithm has been very effective in reducing the number of small packets in the Internet. Today, most TCP implementations include this algorithm.

3 Delayed Acknowledgments

TCP data packets are acknowledged by the recipient. In order to reduce the number of packets in the network, the receiver can "delay" sending the acknowledgment packet for a finite interval. This is done in the hope that the delayed acknowledgment can be "piggy-backed" on a data packet in the opposite direction. The delayed acknowledgment can also carry window update information, informing the TCP sender that the application at the receiver has "read" the previously transmitted data, and that the TCP sender can now send that many more bytes of data. Delayed acknowledgments are particularly useful in client/server applications, where data frequently flows in both directions.

Many TCP implementations are derived from the 4.x BSD code base (described in great detail by Wright and Stevens [19]). These TCP stacks mark a connection that has just received new data by setting the TF_DELACK flag, indicating that the connection needs a delayed acknowledgment. Subsequent transmission of an acknowledgment on the connection clears this flag. A separate timer-driven background activity polls the list of active TCP connections once every 200ms, sending an acknowledgment for each connection that has TF_DELACK set. This means that, on average, an acknowledgment will be delayed for 100ms, and never more than about 200ms.

TCP senders make use of returning acknowledgment packets to "clock" the transmission of packets [5]. This conflicts with the desire to reduce the number of packets in the network. For this reason, TCP receivers must transmit at least one acknowledgment for every two full-sized packets received. Thus, a typical transmission sequence (as seen from node B) might look like: A–>B (data), A–>B (data), B–>A (ack), A–>B (data), ...

4 The Network News Transfer Protocol

The Network News Transfer Protocol (NNTP) [7] is used to transfer Usenet articles between news servers, and between news servers and news readers (clients). NNTP is a request-response protocol. Similar to protocols such as SMTP [13] and FTP [14], requests and responses in NNTP are expressed in ASCII, rather than in a binary format. When used between a client and a server, the client waits for one response before sending the next request. (Server-to-server transfers are often pipelined.)

5 The Original Experiment

In a prior study [15], several of us (Saito, Mogul, and Verghese) measured the response time for NNTP transactions (from the time the client sends its request to the time the client receives the last byte of the response) from a server running the popular INN [16] software. We found a sharp peak at 200ms; figure 1 shows the cumulative distribution function for article retrieval latencies. Note that approximately 25% of the transactions completed in almost exactly 200ms.

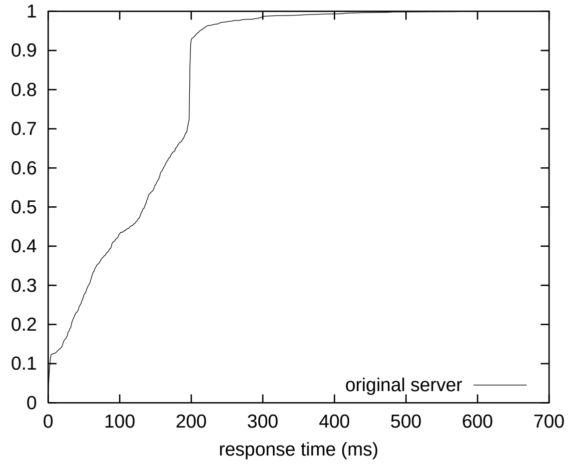

Figure 1: CDF for retrieval latencies: original code

We realized that this 200ms peak was an artifact of an interaction between the Nagle algorithm and the delayed-acknowledgment mechanism. When the server application sends its response, part of the data is presented to the TCP stack in an amount smaller than the MSS. Because the previous response packets have not yet been acknowledged, the Nagle algorithm causes the sender to refrain from sending the small packet until an acknowledgment is received. However, the client application has not read enough data to generate a window update (because the server has not sent enough data since the last acknowledgment from the client), so the acknowledgment for the previous response packet(s) is deferred until the timer expires for the delayed acknowledgment mechanism.

Humans notice, and dislike, a 200ms delay in article retrieval. This delay also increased the mean delay measured by a benchmark we were developing. In order to reduce this delay, we modified the INN server to disable the Nagle algorithm when communicating with clients. (The TCP specification requires that an implementation allow an application to disable this algorithm [1].) We were pleased to note, as shown in figure 2, that the mode at 200ms had disappeared. The mean article retrieval latency dropped, in our benchmark, from 122ms to 71ms.

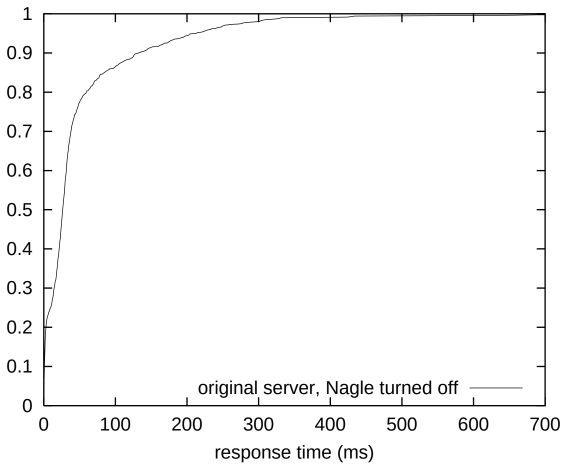

Figure 2: CDF for latencies: Nagle algorithm disabled

Therefore, we recommended that NNTP servers disable the Nagle algorithm for client connections [15].

6 A Closer Look

One of us (Minshall) reacted to this recommendation by asking whether this 200ms latency was a fundamental problem with the Nagle algorithm, or whether it was in fact an aspect of the NNTP server software design that could be corrected in the server software itself. We therefore collected *tcpdump* [6] traces to investigate the detailed behavior of the system.

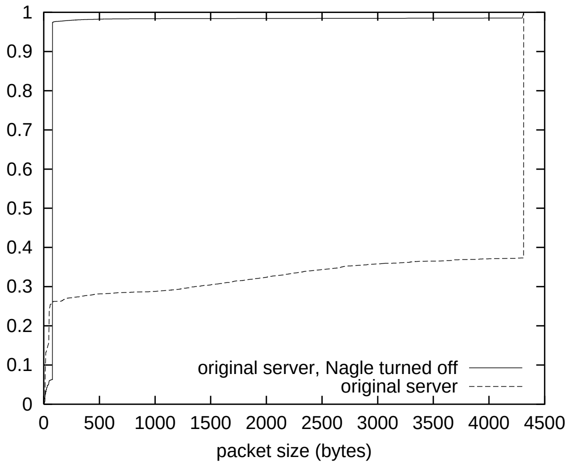

Figure 3: CDFs for packet sizes: original server

These traces revealed a new anomaly. The median packet size (TCP payload) with the Nagle algorithm enabled was 4312 bytes (equal to the MSS of the FDDI LAN used in these tests), but was only 79 bytes with the algorithm disabled (see figure 3). In the former case, a trial transferred 26.7 Mbytes in 9,600 packets; in the latter case, a trial transferred 20.1 Mbytes in 147,339 packets.

This discrepancy led us to discover that the INN server (*nnrpd*), due to an unexpected consequence of the way these experiments were run, was doing line-buffered I/O on its NNTP sockets (rather than fully buffering each response). Therefore, the server was sending each response using many *write()* system calls. With the Nagle algorithm enabled, the TCP stack buffered all but the first line of data until an acknowledgment was received from the news reader, or until at least a packet's worth of response data accumulated at the server. The Nagle algorithm, as intended, prevented the network from being flooded with tiny response packets. However, with the Nagle algorithm disabled, there was nothing to prevent the server TCP from sending a tiny packet for each line written by *nnrpd*, and so it sent lots of them.

7 Ensuring Buffering in the Server

We were surprised to learn that *nnrpd* was using line-buffered I/O. Investigation revealed a subtle cause, specific to the particular experimental setup. When it accepts a new connection, *nnrpd* arranges for that socket's file descriptor to be associated with the *stdout* stream, closing the existing file descriptor initially associated with that stream. This allows *nnrpd* to write its output using the *printf()* library routine, which implicitly names the *stdout* stream. It means, however, that the network connection inherits its buffering mode from *stdout*.

In normal use, *nnrpd* is automatically invoked at system start-time from a script, and *stdout* is fully buffered. However, in the original experiments, we had inserted a debugging *printf()* to *stdout*, and started *nnrpd* from from the command line of a terminal. The *stdio* library causes terminal-output streams to be line-buffered, by default. Thus, the *nnrpd* network connections inherited this line-buffered mode, resulting in multiple writes per news article.

We modified *nnrpd* to ensure that it always used fully-buffered *stdio* streams, regardless of the circumstances of its invocation. Even so, we found responses split into too many packets. We then discovered that *nnrpd* uses a special buffer for some of its output, intermingled with writes via *stdout*. This mingling required the use of extraneous *fflush()* calls. We therefore modified *nnrpd* again, to use only *stdio* buffering. This eliminated all of the excess packet transmissions.

We refer to the original (accidentally line-buffered) *nnrpd* as "poorly-buffered"; the first modified *nnrpd* as "mostly-buffered" (this is the model that *nnrpd* would normally use in practice), and our final modified *nnrpd* as "fully-buffered."

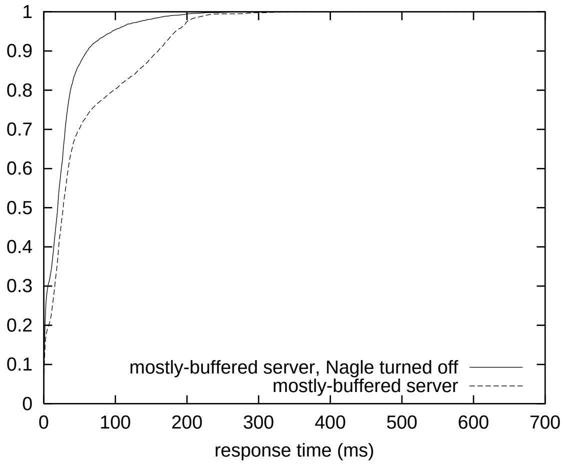

Figure 4: CDFs for latencies: mostly-buffered I/O

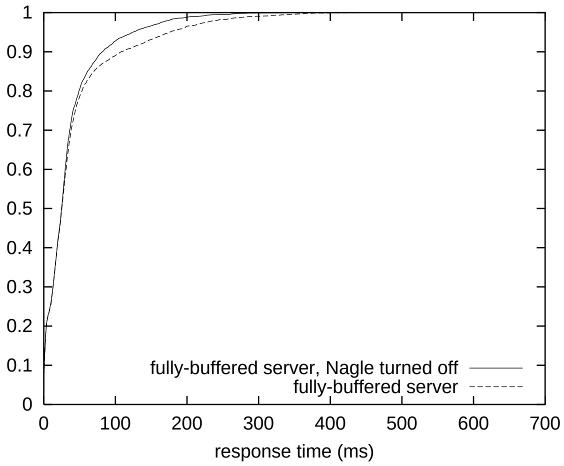

Figure 5: CDFs for latencies: fully-buffered I/O

Figure 4 shows the results for mostly-buffered I/O, with and without the Nagle algorithm; figure 5 shows the results for fully-buffered I/O. Figure 6 combines all of the latency results into one figure, and table 1 shows statistics for each of the six configurations.

From table 1, we see that the mostly-buffered server with the Nagle algorithm disabled has the lowest latency. However, the two fully-buffered server configurations (with and without the Nagle algorithm disabled) show better performance, and fewer packets, than the original poorly-buffered server, even when the poorly-buffered server was running with Nagle disabled.

We are still trying to understand why the mostly-buffered server beats the fully-buffered server; this might be a measurement artifact.

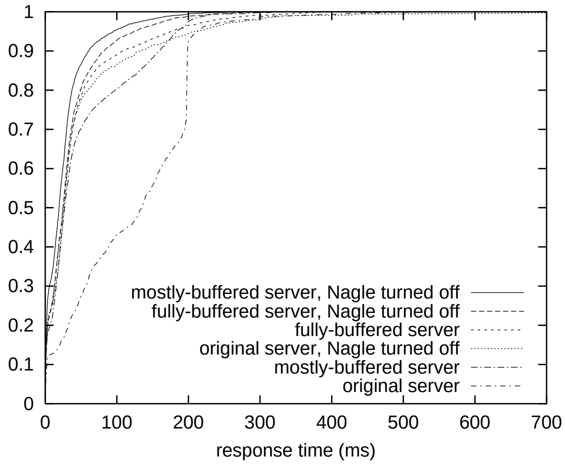

Figure 6: CDFs for latencies: all six configurations

8 Analysis of algorithm interactions

As table 1 shows, even after we ensured the use of fully-buffered I/O in *nnrpd*, our benchmark still yielded better results with the Nagle algorithm disabled. The Nagle algorithm increases mean latency by 19%, and median latency by 4%. What causes this effect? We present a somewhat simplified explanation (see also [4]).

Consider the number of full-sized TCP segments required to convey an NNTP response (such as a news article). If the response exactly fits an integral number of segments, then the Nagle algorithm has no effect, and the TCP stack sends each packet immediately. However, response sizes vary considerably, so most do not exactly fit an integral number of segments.

Now consider an article that requires an odd number of segments; i.e., that requires $2N + 1$ segments. TCP will quickly dispose of the first $2N$ segments, since each pair will result in an immediate acknowledgment; the delayed-acknowledgment mechanism will not be invoked. When the time comes to send the last segment, all previous segments will have been (or will quickly be) acknowledged, so the Nagle algorithm also will not be invoked; there will be no delay.

Finally, consider an article that requires, but does not fill, an even number of segments, i.e., that requires $2N$ segments but contains less than $2N * MSS$ bytes of data. TCP will quickly send the $2N - 1$ full-sized segments, because the Nagle algorithm is not invoked. However, the receiver will delay its acknowledgment of the final full-sized segment, since it is waiting for a pair of segments. Simultaneously, the sender will delay its transmission of the final, partial segment, since the Nagle algorithm is triggered by the outstanding unacknowledged data and the lack of a full segment to send.

With a uniform distribution of response sizes, this

Server configuration	Mean response time (ms)	Median response time (ms)	Mean packet size	Median packet size
poorly-buffered, Nagle enabled	122	135	2913	4312
poorly-buffered, Nagle disabled	71	27	143	79
mostly-buffered, Nagle enabled	53	28	3114	4312
mostly-buffered, Nagle disabled	28	20	3040	3864
fully-buffered, Nagle enabled	43	26	3192	4312
fully-buffered, Nagle disabled	36	25	3131	3864

Table 1: Response time and packet size statistics, one run of the benchmark for each configuration

analysis suggests that just under half of the responses would be delayed by the interaction between the Nagle algorithm and the delayed-acknowledgment mechanism. In reality, the article size distribution is not at all uniform [15]. At least 50% of Usenet articles would fit into a single Ethernet packet, and a substantially larger fraction would fit into a single FDDI packet. Thus, most Usenet articles require an odd number of segments and do not encounter a delay. We found, for our benchmark runs, that 8.7% of the responses do require an even number of FDDI segments, which explains why disabling the Nagle algorithm leads to improved latency statistics.

8.1 Results with Ethernet MTU

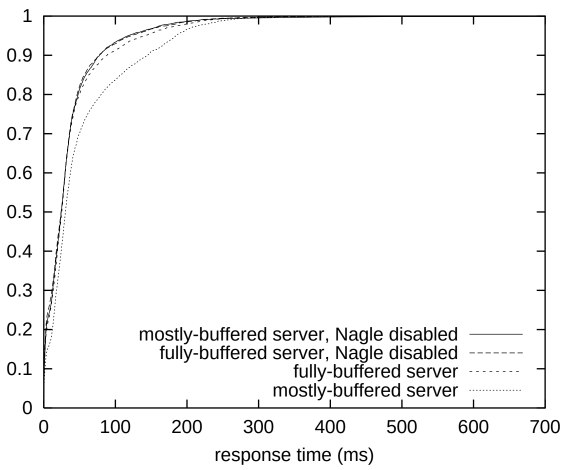

Figure 7: CDFs for latencies with Ethernet MTU

Encouraged by our success in predicting the frequency of packet delays using the FDDI segment size, we decided to verify our model using a different packet size. We ran trials over an FDDI network that had been reconfigured to use the same MSS as would be used on an Ethernet (1460 bytes). By using a reconfigured FDDI network, rather than an actual Ethernet network, we avoided changing several irrelevant variables (primarily, the hardware implementation and the associated kernel driver software) that might have affected the results.

We ran trials using both the mostly-buffered and fully-buffered servers, with and without the Nagle algorithm enabled. Figure 7 shows the CDFs for these four trials. Table 2 shows the mean and median latencies and packet sizes.

Again, we found that the Nagle algorithm does slightly increase latency. However, the increase was much smaller than we expected, since the simple model in Section 8, applied to the distribution of response sizes, suggests that 45% of the responses should encounter delays. This clearly does not happen.

Indeed, our model is too simple. The BSD TCP implementation of the Nagle algorithm never delays a packet transmission if the connection was idle when the *tcp_output()* function is called, so the first "cluster" of data can be sent without delay. On our systems, the cluster size is 8192 bytes, large enough to cover a significant fraction of the NNTP responses.

8.2 Other contexts where this effect occurs

We are not the first to point out the performance implications of the the interaction between the Nagle algorithm and the delayed-acknowledgment mechanism. The effect occurs in contexts besides NNTP.

Early examples including the use of multi-byte "function keys" in interactive terminal sessions [18], and the interactive X window system [17]. In both of these cases, implementors quickly realized the need to disable the Nagle algorithm [3].

More recently, the introduction in HTTP of "persistent connections" (the use of a single TCP connections for several HTTP requests) has led several researchers [4, 10] to similar conclusions.

In general, any application that uses TCP as a transport for a series of reciprocal exchanges can suffer from this interaction. The effect would not appear when an entire request or response always fits in one TCP segment, but may be seen when the request or response is so large as to need to be transmitted in more than one packet.

Server configuration	Mean response time (ms)	Median response time (ms)	Mean packet size	Median packet size
mostly-buffered, Nagle enabled	52	31	1283	1460
mostly-buffered, Nagle disabled	37	25	1236	1460
fully-buffered, Nagle enabled	38	26	1294	1460
fully-buffered, Nagle disabled	35	25	1247	1460

Table 2: Response time and packet size statistics, one run of the benchmark for each configuration with Ethernet MTU

9 Phase Effects in the Original Experiment

Figure 1 shows something odd. As we explained in section 3, the BSD delayed acknowledgments mechanism should produce delays uniformly distributed in the interval [0...200] ms. A graph of the cumulative distribution of response times should include a nearly linear section starting at the undelayed response time, and extending 200ms beyond this point. Instead, figure 1 shows a sharp jump at 200ms, as if the delayed acknowledgment mechanism were waiting exactly 200ms before sending the delayed acknowledgment.

While human users of NNTP servers often pause for several seconds between articles, the benchmark we used in these experiments sends client requests as rapidly as possible; this allows us to measure worst-case server behavior. When the delayed acknowledgment first delays the completion of an interaction in a series, we would indeed expect the delay to be uniformly distributed. However, the subsequent interaction now starts shortly after a 200ms timer "clock tick," and if it is delayed, then it will complete upon the subsequent timer expiration – almost exactly 200ms after it starts.

Floyd and Jacobson [2] discuss synchronization of nodes in the Internet, in the context of periodic routing messages. As they point out, such synchronization can produce significant burst loads. In the case we analyzed in this paper, the synchronization is based on the clock of the news reader (client) host. In practice, many news clients with independent clocks access one server, which should reduce the overall synchronization effect. Our benchmark, which uses multiple news reader processes on a single client machine, leads to a worst-case synchronization effect. We note, however, that if the bulk of the data had been flowing from many senders to one recipient, this might have synchronized the delays to the recipient's clock, thereby inducing large synchronized bursts of data arriving at the recipient. For example, an HTTP proxy retrieving responses from many Internet servers, using persistent connections, could create synchronized bursts of incoming data through an interaction between Nagle's algorithm and delayed acknowledgments.

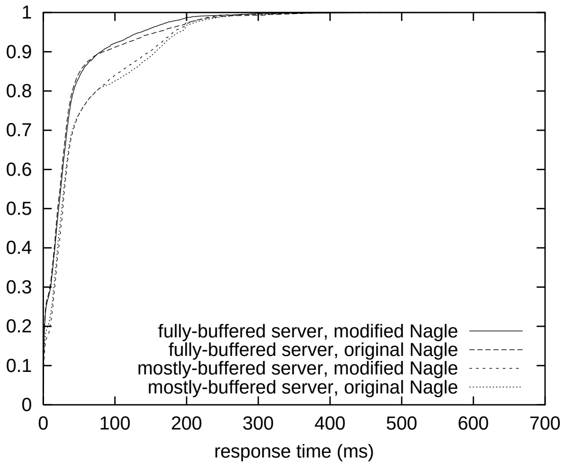

Figure 8: Effect of modified Nagle algorithm

10 A Proposed Solution

The Nagle algorithm, as we saw when running the "accidentally unbuffered" version of the NNTP software, does as intended prevent the transmission of excessive numbers of small packets. Can we preserve this feature without adversely affecting the performance of properly buffered applications? And can we avoid the need for application programmers to determine whether or not to disable the Nagle algorithm?

One of us (Minshall) has recently proposed a modification to the Nagle algorithm [8]. Instead of delaying the transmission of a short packet when there is any unacknowledged data, this modified version would delay only if the unacknowledged data was itself sent in a short packet. This approach would preserve the algorithm's original intention, preventing a flood of small packets, but would not add delay to the transmission of responses requiring $2N$ segments.

10.1 Validation of proposed solution

To validate the proposed modifications to Nagle's algorithm, we implemented a somewhat simplified version of Minshall's approach. This version requires only one bit of additional state per TCP connection, and allows

Server configuration	Mean response time (ms)	Median response time (ms)	Mean packet size	Median packet size
mostly-buffered, original Nagle	50	26	3019	4312
mostly-buffered, modified Nagle	48	28	2975	3864
fully-buffered, original Nagle	36	21	3092	4312
fully-buffered, modified Nagle	34	21	3043	3864

Table 3: Response time and packet size statistics, with original and modified Nagle algorithms

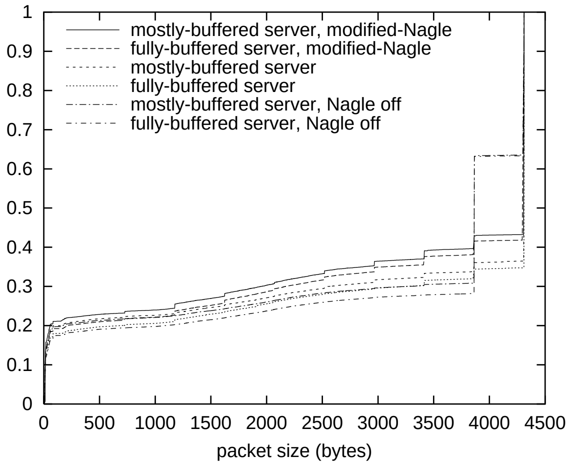

Figure 9: CDFs for packet sizes

two small packets in a row (without delay) if the first one results from the kernel's use of a fixed cluster size when filling the TCP output buffer. The modifications are shown in Appendix A.

We ran trials using both the mostly-buffered and fully-buffered servers, with either the standard or modified Nagle algorithms. Figure 8 shows the CDFs for these four trials. Table 3 shows the mean and median latencies and packet sizes.

Figure 8 shows that the fully-buffered server generally outperforms the mostly-buffered server, in that the fully-buffered distributions are shifted towards lower latencies. This figure also shows only a slight improvement from the modified Nagle algorithm, for either buffering model. However, we ran additional trials and found that, on the whole, the modified Nagle algorithm generally provides a modest improvement for both server buffering models. We intend to do a further study using a careful statistical analysis of a large set of trials (and apologize for the use of single trials in this paper).

Figure 9 shows the packet size distributions for various configurations of server buffering model and the Nagle algorithm. It is difficult to see this from the black and white version of the figure, but generally the modified Nagle algorithm causes slightly more packets at sizes under 3880 bytes, while entirely disabling the Nagle algorithm leads to the fewest such packets. However, disabling the Nagle algorithm creates a large number of 3880-byte packets. This threshold corresponds to the difference between MCLBYTES (8192) and the FDDI MSS (4312); we are not entirely sure why the Nagle algorithm specifically suppresses packets of that size.

11 Summary and conclusions

We identified a performance problem that appeared to arise from an interaction between TCP's Nagle algorithm and delayed acknowledgment policies. We found that disabling the Nagle algorithm appeared to solve this problem, but unleashed a torrent of small packets. By fixing the buffering problem that caused this torrent, we showed that the Nagle algorithm still causes a slight increase in mean latency, and we analyzed the causes behind this increase. This suggests that a simple modification to the Nagle algorithm would improve latency without removing the protection against packet floods.

We conclude that:

1. Disabling the Nagle algorithm can greatly increase network congestion, if application buffering is imperfect.

2. Even carefully-written applications can, on occasion, exhibit sub-optimal buffering behavior.

3. Even with optimal buffering, the interaction between the Nagle algorithm and delayed acknowledgments can increase application latency, if only slightly.

4. The best solution would be the use of careful buffering, combined with an improved Nagle algorithm that defends the network against buffering errors without adding much latency.

5. Implementors who are tempted to disable the Nagle algorithm should take care not only to use careful buffering, but also to measure the result to ensure that the buffering works as expected (perhaps by inspecting the resulting packet stream using a tool such as tcpdump [6]).

12 Acknowledgements

We would like to thank John Nagle for his insightful comments. We would also like to thank the WISP reviewers for their suggestions.

A Modifications to the Nagle algorithm

Our modifications to the Nagle algorithm, a somewhat simplified version of those proposed by Minshall [8], are confined to the *tcp_output()* function from the BSD networking code. We show them as insertions to the code given by Wright and Stevens [19], in Figure 10. Only certain lines of the code are shown; the source line numbers are taken from Wright and Stevens, and our insertions are shown with "***" instead of line numbers.

References

[1] R. Braden. Requirements for internet hosts - communication layers. RFC 1122, Internet Engineering Task Force, ftp://ftp.isi.edu/in-notes/rfc1122.txt. October 1989.

[2] Sally Floyd and Van Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, April 1994.

[3] Jim Gettys. Personal communication. 1999.

[4] J. Heidemann. Performance interactions between P-HTTP and TCP implementations. *ACM Computer Communication Review*, 27(2):65–73, http://www.acm.org/sigcomm/ccr/archive/. April 1997.

[5] Van Jacobson. Congestion avoidance and control. In *Proc. SIGCOMM '88*, pages 314–32, Stanford, CA, ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z. August 1988.

[6] Van Jacobson et al. *tcpdump(1), BPF*, ftp://ftp.ee.lbl.gov/tcpdump.tar.Z. 1990.

[7] 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, ftp://ftp.isi.edu/in-notes/rfc977.txt. February 1986.

[8] Greg Minshall. A suggested modification to nagle's algorithm. Internet-Draft draft-minshall-nagle-00, Internet Engineering Task Force, http://www.ietf.org/internet-drafts/draft-minshall-nagle-00.txt. December 1998. This is a work in progress.

[9] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Internet Engineering Task Force, ftp://ftp.isi.edu/in-notes/rfc896.txt. January 1984.

[10] 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, September 1997.

[11] J. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, ftp://ftp.isi.edu/in-notes/rfc768.txt. August 1980.

[12] J. Postel. Transmission control protocol. Request for Comments (Standard) STD 7, RFC 793, Internet Engineering Task Force, ftp://ds.internic.net/rfc/rfc793.txt. September 1981.

[13] J. Postel. Simple mail transfer protocol. RFC 821, Internet Engineering Task Force, ftp://ftp.isi.edu/in-notes/rfc821.txt. August 1982.

[14] J. Postel and J. Reynolds. File transfer protocol. RFC 959, Internet Engineering Task Force, ftp://ftp.isi.edu/in-notes/rfc959.txt. October 1985.

[15] Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. A Usenet Performance Study, http://www.research.digital.com/wrl/projects/newsbench/usenet.ps. November 1998.

[16] Rich Salz. InterNetNews: Usenet Transport for Internet Sites. In *USENIX Conference Proceedings*, pages 93 – 98, San Antonio, TX, Summer 1992. USENIX.

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

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

[19] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated Volume 2.* Addison-Wesley, Reading, MA, 1995.

Define a new flag value for the tcp_flags field, in tcp_var.h:

```
***  #define TF_SMALL_PREV 0x800
```

On entry to tcp_output() (in particular, before the `again` *label), record the number of bytes in the socket buffer:*

```
47      struct socket *so = inp->inp_socket;
***     int start_cc = so->so_snd.sb_cc;
```

After the check made by the traditional Nagle algorithm, which sends a small packet at the end of the socket buffer in certain situations, allow sending in one more situation: if the previous packet was not "small" (note that these two conditional statements could be merged):

```
144         if ((idle || tp->t_flags & TF_NODELAY) &&
145             len + off >= so->so_snd.sb_cc)
146                 goto send;
***         if ( ((tp->t_flags & TF_SMALL_PREV) == 0) &&
***             (len + off >= so->so_snd.sb_cc)) {
***                 /* end-of-buffer & previous packet was not "small" */
***                 goto send;
***         }
```

Just before sending a packet, remember that it was "small" if the length is less than the MSS, and if the original buffer size was not a multiple of the cluster size:

```
222  send:
***      /* Remember end of most recent "small" output packet */
***      if ((len < tp->t_maxseg) && ((start_cc % MCLBYTES) != 0)) {
***          tp->t_flags |= TF_SMALL_PREV;
***      }
***      else
***          tp->t_flags &= ~TF_SMALL_PREV;
```

Figure 10: Modifications to *tcp_output()*