Internet Engineering Task Force
INTERNET-DRAFT
draft-minshall-nagle-01.txt

Greg Minshall
Siara Systems

June 17, 1999

A Proposed Modification to Nagle's Algorithm

Status of This Memo

This draft proposes a modification to Nagle's algorithm (as specified in RFC896) to allow TCP, under certain conditions, to send a small sized packet immediately after one or more maximum segment sized packet.

Abstract

The Nagle algorithm is one of the primary mechanisms which protects the internet from poorly designed and/or poorly implemented applications. However, for a certain class of applications (notably, request-response protocols) the Nagle algorithm interacts poorly with delayed acknowledgements to give these applications poorer performance.

This draft is NOT suggesting that these applications should disable the Nagle algorithm.

This draft suggests a fairly small and simple modification to the Nagle algorithm which preserves the Nagle algorithm as a means of protecting the internet while at the same time giving better performance to a wider class of applications.

Introduction to the Nagle algorithm

The Nagle algorithm [RFC896] protects the internet from applications (most notably Telnet [RFC854], at the time the algorithm was developed) which tend to dribble small amounts of data to TCP. Without the Nagle algorithm, TCP would transmit a packet, with a small amount of data, in response to each of the application's writes to TCP. With the Nagle algorithm, a first small packet will be transmitted, then subsequent writes from the application will be buffered at the sending TCP until either i) enough application data has accumulated to enable TCP to transmit a maximum sized packet, or ii) the initial small packet is acknowledged by the receiving TCP. This limits the number of small packets to one per round trip time.

While the current Nagle algorithm does a very good job of protecting the internet from such applications, there are other applications, such as request-response protocols (with HTTP [RFC2068] being a topical example) in which the current Nagle algorithm interacts with TCP’s ‘‘delayed ACK’’ policy [RFC1122] to produce non-optimal results.

Delayed ACKs

A receiving TCP tries to avoid acknowledging every received data packet in the hope of ‘‘piggy-backing’’ the acknowledgement on a data packet flowing in the reverse direction or combining the acknowledgement with a window update flowing in the reverse direction. This process, known as ‘‘delayed ACKing’’ [RFC1122], typically causes an ACK to be generated for every other received (full-sized) data packet. In the case of an ‘‘isolated’’ TCP packet (i.e., where a second TCP packet is not going to arrive anytime soon), the delayed ACK policy causes an acknowledgement for the data in the isolated packet to be delayed up to 200 milliseconds of the receipt of the isolated packet (the actual maximum time the acknowledgement can be delayed is 500ms [RFC1122], but most systems implement a maximum of 200ms, and we shall assume that number in this document). The way delayed ACKs are implemented in some systems causes the delayed ACK to be generated anytime between 0ms and 200ms; in this case, the average amount of time before the delayed ACK is generated is 100ms.

The interaction of delayed ACKs and Nagle

If a TCP has more application data to transmit than will fit in one packet, but less than two full-sized packets’ worth of data, it will transmit the first packet. As a result of Nagle, it will not transmit the second packet until the first packet has been acknowledged. On the other hand, the receiving TCP will delay acknowledging the first packet until either i) a second packet arrives (which, in this case, won’t arrive), or ii) approximately 100ms (and a maximum of 200ms) has elapsed.

When the sending TCP receives the delayed ACK, it can then transmit its second packet.

In a request-response protocol, this second packet will complete either a request or a response, which then enables a succeeding response or request.

Note two (related) bad results of the interaction of delayed ACKs and the Nagle algorithm in this case: the request-response time may be increased by up to 400ms (if both the request and the response are delayed); and, consequently, the number of transactions per second is substantially reduced.

A proposed modification to the Nagle algorithm

In the following discussion we make use of the following variables defined in the TCP RFC [RFC793] and in the host requirements RFC [RFC1122]: ‘‘snd.nxt’’ is a TCP variable which names the next byte of data to be transmitted; ‘‘snd.una’’ is a TCP variable which names the next byte of data to be acknowledged (if snd.nxt equals snd.una, then all previous packets have been acknowledged); Eff.snd.MSS is the largest TCP payload (user data) that can be transmitted in one packet.

The current Nagle algorithm does not require any other state to be kept by TCP on a system.

The proposed modification to the Nagle algorithm does, unfortunately, require one new state variable to be kept by TCP: ‘‘snd.sml’’ is a TCP variable which names the last byte of data in the most recently transmitted small packet.

The current Nagle algorithm can be described as follows:

> "If a TCP has less than a full-sized packet to transmit,
> and if any previous packet has not yet been acknowledged,
> do not transmit a packet."

and in pseudo-code:

```
if ((packet.size < Eff.snd.MSS) && (snd.nxt > snd.una)) {
        do not send the packet;
}
```

The proposed Nagle algorithm modifies this as follows:

> "If a TCP has less than a full-sized packet to transmit,
> and if any previously transmitted less than full-sized
> packet has not yet been acknowledged, do not transmit
> a packet."

and in pseudo-code:

```
if (packet.size < Eff.snd.MSS) {
        if (snd.sml > snd.una)) {
                do not send the packet;
        } else {
                snd.sml = snd.nxt+packet.size;
                send the packet;
        }
}
```

In other words, when running Nagle, only look at the recent transmission (and acknowledgement) of small packets (rather than all packets, as in the current Nagle).

(In writing the above, I am aware that TCP acknowledges bytes, not packets. However, expressing the algorithm in terms of packets seems to make the explanation a bit clearer.)

Implementing Nagle at Send

The above description of the current Nagle algorithm and of the proposed modification assumes that the Nagle algorithm is being implemented just as TCP is about to hand a packet to IP to be transmitted, i.e., the algorithm is looking at the sizes of the packets it transmits.

In reality, many TCPs essentially implement Nagle at the interface where applications present data to TCP to be transmitted (i.e., in the call to ‘‘SEND’’, as defined in section 3.8 of the TCP specification [RFC793]). The motivation for this is to not penalize applications that provide data to TCP in large chunks (ideally a multiple of Eff.snd.MSS).

This allows a single application send to be broken into zero or more full-sized packets, possibly followed by one small packet, without forcing any delay on the trailing small packet. For example, one implementation with which the author is familiar first captures the boolean ‘‘snd.nxt > snd.una’’ in a temporary variable (‘‘busy’’):

```
     busy = (snd.nxt > snd.una);
```

then goes into a loop transmitting packets out of the data which has been presented to TCP by the application; the loop contains the following code to implement the current Nagle algorithm:

```
     if ((packet.size < Eff.snd.MSS) && busy) {
             do not send the packet;
     }
```

Since ‘‘busy’’ is a constant in the loop transmitting packets, a trailing small packet will be transmitted (after zero or more large packets transmitted by the same call to send) if the connection had no outstanding data at the time the application presented data to TCP for transmission (assuming the TCP window allows this).

To implement the modified Nagle algorithm in such a system, we replace snd.sml with two variables: ‘‘snd.sml.add’’ is a TCP variable which names the last byte presented to TCP by the application with a ‘‘small’’ send (i.e., the application called SEND with fewer than Eff.snd.MSS bytes of data); and ‘‘snd.sml.snt’’ is a TCP variable which names the highest value of snd.sml.add which has, in fact, been transmitted. The send routine contains the following code:

```
     if (byte.count < Eff.snd.MSS) {
             snd.sml.add = snd.una + snd.bytes.queued;
     }
```

(where ‘‘snd.bytes.queued’’ is the number of bytes queued for transmission, and has already been updated with ‘‘byte.count’’, the number of bytes being presented to TCP in this call to SEND).

The loop that transmits packets contains the following code:

```
     if (packet.size < Eff.snd.MSS) {
             if (snd.sm.snt > snd.una) {
                     do not send the packet;
             } else {
                     if ((snd.nxt + packet.size) <= snd.sm.add) {
                             snd.sm.snt = snd.sm.add;
                     }
                     send the packet;
             }
     }
```

(In most implementations, the most deeply nested ‘‘if’’ statement above is unnecessary, as a small-sized packet will contain all the data available to be transmitted, and so will include, or be beyond, snd.sm.add. In this case, the modified Nagle algorithm adds one test, one addition, and one assignment in the send routine, and one assignment in the output routine.)

A Failure Mode

If an application sends a large amount of data, followed by a small amount of data, followed by a large amount of data, the current Nagle algorithm would perform better than the proposed modification. The current Nagle algorithm would send at most one small packet (possibly the last packet), delaying the middle (small) amount of data which would allow the application to send the following large amount of data; the modified Nagle algorithm would send as many as two small packets (the middle packet, plus possibly a last packet).

A separate, but desirable, system facility

In addition to the Nagle algorithm (or the modification proposed by this draft), it would be desirable for a system providing TCP service to applications to allow the application to set TCP into a mode in which the TCP would only transmit small packets at the explicit direction of the application. For example, a system based on BSD might implement a socket option (using setsockopt(2)) SO_EXPLICITPUSH, as well as a flag to sendto(2) (possibly overloading the semantics of an existing flag, such as MSG_EOF).

In this scenario, an application would set a socket into SO_EXPLICITPUSH mode, then enter a mode of writing data to the socket and, at the last write, using send(2) with the MSG_EOF flag. The underlying TCP would recognize the MSG_EOF flag as an indicator to transmit the (possibly) small packet.

Like the proposed modification to the Nagle algorithm, this is fairly simple to implement.

If a system were to implement this interface, it would be important to NOT disable Nagle when using this interface. In other words, when using this interface, the default mode for TCP would be to NOT transmit a small packet (even in the presence of MSG_EOF) if a previously transmitted small packet was as yet unacknowledged.

Note, also, that implementing this interface does not eliminate the desirability of using the modification of the Nagle as the default for applications. More sophisticated networking applications might well use the new interface, but naive applications will often be adequately served by the modified Nagle algorithm.

Application scenarios that will not be helped by this modification

The proposed modification helps applications which do not need to transmit more than one small packet in a single round-trip time. This characterizes one way file transfer applications (such as FTP [RFC959]) and request/response protocols (such as NNTP [RFC977] and HTTP [RFC2068] without pipelining).

However, applications that need to transmit more than one small packet in a single round-trip time are not served by this modification. An example of such an application is HTTP [RFC2068] using ‘‘pipelining’’, in which multiple requests (responses) are transmitted asynchronously.

Applications needing to transmit more than one small packet in a single round-trip time will need other mechanisms to satisfy their requirements. (One possible such mechanism would be to use more than one TCP connection.)

If an application developer is considering disabling the Nagle algorithm, they should be very careful to ensure that their application will generally provide data to TCP in chunks larger than two full-sized segments (> 2*Eff.snd.MSS), and they should verify after their development that this is, in fact, true. With Nagle disabled, many writes of small blocks of data can add significant load to the network, reducing the network’s performance.

Acknowledgements

Jim Gettys, Henrik Frystyk Nielsen, Jeff Mogul, and Yasushi Saito, as well as a message forwarded to the end2end-interest list by Sean Doran, have motivated my current interest in the Nagle algorithm. John Heidemann’s work related to the Nagle algorithm has informed some of the thinking in this draft; discussions with John have also been helpful. Members of the End-to-End Research Group (under the direction of Bob Braden) patiently listened to my discussion of the current state of the Nagle algorithm and to the modifications proposed in this document.

Members of the TCP implementors mailing list <tcp-impl@lerc.nasa.gov> have been very helpful in refining this proposal. In particular, Rick Jones, Neal Cardwell, Vernon Schryver, Bernie Volz, Sam Manthorpe, Art Shelest, David Borman, Kacheong Poon, Jon Snader, Eric Hall, Joe Touch, and Alan Cox.

Security Considerations

The Nagle algorithm does not have major security consequences.

Implementation of this algorithm should not negatively impact the performance of the internet. The negative impact of implementation of this algorithm should be significantly less than disabling the Nagle algorithm.

Appendix -- Sample application code

The following code is provided to give application developers a model for buffering. We assume a BSD-style sockets API.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

#define SNDBUF_MULT 3     /* * 2 * TCP_MAXSEG -> SO_SNDBUF */

/*
 * Given a connected socket (s), configure the socket
 * with good buffer size defaults, and return the
 * the size the application should use for issuing
 * writes to the socket.
 *
 * Returns size to use for application buffering, or
 * zero (0) on error.
 */
int
getbufsize(int s)
{
        unsigned long bufsize, parm;
        int buflen;

        buflen = sizeof bufsize;
        if (getsockopt(s, IPPROTO_TCP, TCP_MAXSEG,
                                &bufsize, &buflen) == -1) {
                perror("getsockopt(...TCP_MAXSEG...)");
                return 0;
        }

        /* Set socket transmit buffer */
        parm = 2*SNDBUF_MULT*bufsize;
        if (setsockopt(s, SOL_SOCKET, SO_SNDBUF,
                                &parm, sizeof parm) == -1) {
                perror("setsockopt(SO_SNDBUF)");
                return 0;
        }

        /* Now, set socket low water threshhold */
        parm = 2*bufsize;
        if (setsockopt(s, SOL_SOCKET, SO_SNDLOWAT,
                                &parm, sizeof parm) == -1) {
                perror("setsockopt(...SO_SNDLOWAT...)");
                return 0;
        }

        return 2*bufsize;
}

int
main(int argc, char *argv[])
{
        char *buffer = 0;
        int buflen;
        int sock;

        /*
         * ... allocate a socket (sock) and get it connected
         * via either connect(2) or listen(2)/accept(2).
         */

        buflen = getbufsize(sock);
        if (buflen == 0) {
                fprintf(stderr, "aborting\n");
                exit(1);
        }

        buffer = malloc(buflen);
        if (buffer == 0) {
                fprintf(stderr,
                        "no room for buffer of size %d\n",
                                                    buflen);
                exit(1);
        }
```

```
                /*
                 * ... loop generating ‘‘buflen’’ data in buffer
                 * and using send(2) to hand it to TCP.
                 * When there is no more data to send, call
                 * send(2) one last time with <= ‘‘buflen’’
                 * bytes.
                 */

                return 0;
        }
```

References

[RFC793]   Postel, J. (ed), "Transmission Control Protocol",
                Sep-1981.

[RFC854]   Postel, J., J. Reynolds, "Telnet Protocol
                Specification", May-1983.

[RFC959]   Postel, J., J. Reynolds, "File Transfer Protocol
                (FTP)", Oct-1985.

[RFC977]   Kantor, B., P. Lapsley, "Network News Transfer
                Protocol", Feb-1986.

[RFC896]   Nagle, J., "Congestion control in IP/TCP internetworks",
                Jan-06-1984.

[RFC1122]   Braden, R. T., "Requirements for Internet hosts -
                communication layers", Oct-01-1989.

[RFC2068]   Fielding, R., J. Gettys, J. Mogul, H. Frystyk,
                T. Berners-Lee, "Hypertext Transfer Protocol
                -- HTTP/1.1".

Author’s Address

Greg Minshall
Siara Systems
300 Ferguson Drive, 2nd floor
Mountain View, CA 94043
USA

<minshall@siara.com>