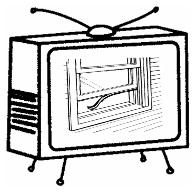
Computer Networks
Flow control with TCP

Saad Mneimneh
Computer Science
Hunter College of CUNY
New York



advertised window

1 Introduction

We have seen that at the conceptual level, TCP uses the same sliding window algorithm used by the DLC layer. However, while DLCs can be matched for speed, the sender and receiver on each side of a TCP connection may have different speeds. This is determined by the application programs communicating over TCP. If the receiver is slower than the sender, bytes will have to be dropped from the receiver's sliding window buffer. TCP deals with this issue using what is known as flow control.

2 Flow control with advertised window

With TCP, a slow receiver can limit how much data a sender can transmit. This is integrated within the sliding window algorithm, so the receiver can control how large the sender's window can be. The receiver advertises a window size in bytes to the sender through a 16 bit number in the TCP header (the advertised window field). Upon receipt of a TCP segment from the receiver, the sender adjusts its window size accordingly.

Let LR be the last byte read by the receiver (delivered to the application program). Ideally, $LR = RN - 1$ if the receiver is fast enough. In general, however, $LR \leq RN - 1$ (with $LR < RN - 1$ when the receiver is slow).

If the current window size at the receiver is m bytes, the receiver will advertise a window size of $advw = m - [(RN-1) - LR]$ bytes.

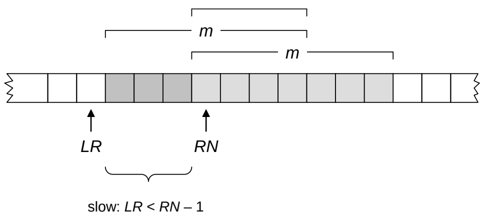


Figure 1: Receiver's window

Upon the receipt of a segment from the receiver, the sender sets its window size n to the advertised window field $advw$. If the sender wants to be more conservative, it will set its window size to $n = \max(0, advw - (LS - SN + 1))$ bytes, where LS is the last byte transmitted by the sender over the network (but may not have reached the receiver yet).

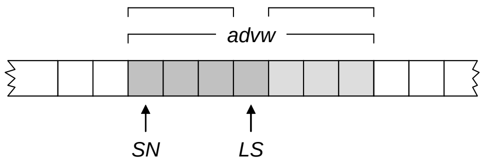


Figure 2: Sender's window

If the sender's window size becomes zero, the sender stops sending immediately (and TCP blocks the corresponding application program).

How does the sender know that the advertised window is no longer zero? Because the sender is not permitted to send any more data, and the receiver sends acks only in response to received data, there is no way for the sender to discover that the receiver has opened up the window. One can imagine the receiver sending some control information telling the sender that the window has opened. This, however, will complicate the receiver. The solution to this problem follows the following paradigm:

smart sender/dump receiver

The receiver simply responds to segments from the sender and never initiates activity on its own. In fact, the replacement of acks and naks with RN follows the same paradigm.

Whenever the receiver advertises a window size of zero, the sender persists in sending a segment with one byte of data every so often. These segments will probably not be accepted at the receiver (window size is zero), but eventually one of them will trigger a response that reports a non-zero advertised window. This technique is called *probing.*

3 Segmentation and Nagling

Recall that TCP is a byte oriented protocol, in the sense that every byte has its own SN. However, TCP sends bytes in segments. A related issue to flow control is how TCP decides when to send. Ideally, TCP uses a Maximum Segment Size (MSS) for its segments imposed by the Maximum Transmission Unit (MTU) allowed by the underlying network. Therefore, MSS $<$ MTU, which prevents additional segmentation at the network level. But should TCP wait to accumulate enough bytes to fill an MSS-sized segment? Or should it send data immediately in a smaller segment of few bytes? The latter case implies inefficient use of bandwidth by sending high overhead "tinygrams" with the TCP (and IP) headers consuming most of the bytes. However, this issue is not relevant unless TCP actually has few bytes to send. But why would that be the case? The answer to this lies in three similar yet different problems:

- Small ack problem: A receiver's application program with no response to the sender causes the receiver to send empty TCP segments (acks) to ack.
- Silly window syndrome (SWS): A receiver's slow application program causes the receiver to advertise a small window size.
- Small segment problem: A sender's slow application program provides the sender with only few bytes per second.

To solve the small ack problem, the TCP specification says that TCP **should** implement a delayed ack strategy. This means is that TCP **should** delay the ack until the receiver has a response to the sender, and piggybacks the ack on the response segment. Such a response may not be available, so the delay **must** be less than 0.5 seconds, typically 200 ms. Moreover, in a stream of MSS-sized segments there **should** be an ack for at least every second segment.

The problem of SWS is easily avoided by making the receiver resist advertising a window size greater than zero unless it is at least $\min(MSS, L/2)$, where L is the receiver's available buffer space (suggested by David Clark).

Unlike the previous two problems, a solution to the small segment problem must be placed at the sender, but the solution is more complicated as illustrated by the following example.

A user on a telnet application types 25 *characters, one character every* 200 *ms*

If TCP sends data immediately, then each byte of data (character) will generate 20 bytes of TCP header and 20 bytes of IP header (we will see this when we study the IP packet format), resulting in a 40 to 1 overhead (a total of 1000 bytes). While this is acceptable on a LAN like Ethernet, it is not acceptable on highly congested WANs. If on the other hand, TCP waits to fill an MSS-sized segment, the question is then how long should the wait be? This will affect the responsiveness of the application. For instance, imagine the user hits return and nothing happens because TCP is waiting for more characters!

An engineering solution would be to wait for a predetermined amount of time before sending a small segment (less than MSS bytes). Therefore, TCP can use a timer that fires, say, every 500 ms. When the timer fires, TCP fills the largest possible segment and sends it. Unfortunately, this approach is not satisfactory on most networks.

3.1 Example 1: LAN with 50 ms RTT

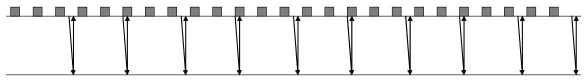


Figure 3: Timer solution on LAN

As illustrated in the above figure, the user gets a response every 2 or 3 characters. The overhead is 16 to 1 (a total of 400 bytes). This is not a good compromise.

3.2 Example 2: WAN with 5 sec RTT

In this case, the responsiveness is not that bad since the user has to wait 5 sec for the first response anyway. But the overhead is still high; we still have too many segments injected into the WAN (10 segments). What this means is that TCP should have waited longer before sending each segment. But this will only make it worse for the LAN situation of Example 1.

3.3 Nagle's algorithm

What we conclude from the above two examples is that we need a timer that is adaptive to the network, e.g. fast for LAN and slow for WAN. John Nagle suggested a self-clocking algorithm for the sender that uses the ack as the firing of a timer. The sender delays a segment until all outstanding data has been acknowledged (stop and wait) or until TCP has an MSS-sized segment.

```
while more data
    if both available data and the window $\geq MSS$
        send a full segment
    else
        // stop and wait
        if there is unACKed data in transit
            buffer the new data
        else
            send data in a small segment
```

Therefore, on fast LANs, Nagle's algorithm behaves like stop and wait for small segments, but has high overhead. On slow WANs, Nagle's algorithm implies more wait but is more efficient. Let's revisit the two examples above. For the LAN example with a 50 ms RTT, every character will see no data in transit (50 ms < 200 ms) and, therefore, is sent immediately. The response is fast, but the overhead is 40 to 1 again (a total of 1000 bytes for 25 characters). This is appropriate for a LAN. For the WAN example with a 5 sec RTT, the first character will see no data in transit and, therefore, will be sent immediately. The rest of the characters will wait for the ack (5 sec = 25×200 ms). When the ack arrives, the 24 remaining characters are sent in one segment. The overhead is 3.2 to 1 (a total of 80 bytes). Only two segments are injected into the WAN. This is appropriate for a WAN.

3.4 To Nagle or not to Nagle

It turns out that Nagle's algorithm performs badly when combined with delayed acks. This is best exemplified when we have a repeated pattern of communication involving small messages in a sequence of write, write, and read operations. The receiver must wait for the second write to issue a response. Therefore, every read operation is delayed. The following figure illustrates the idea:

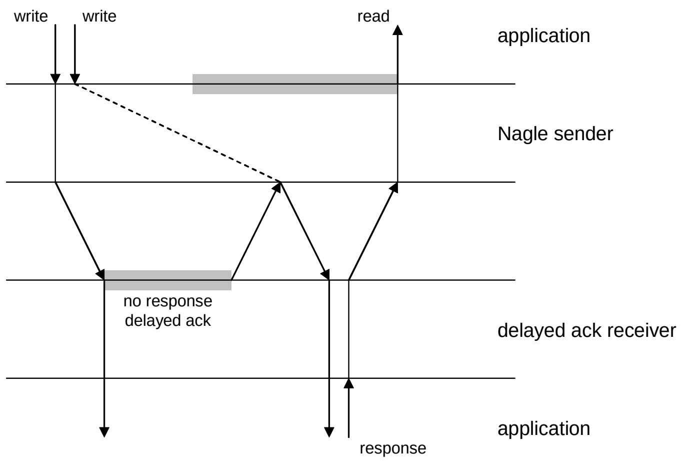


Figure 4: Nalge combined with delayed ack

For this reason, the TCP specification says that there **must** be a way for an application to disable the Nagle algorithm on an individual connection. The socket API provides a way to do this by setting the socket option TCP_NODELAY.

```
socklen_t i=1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &i, sizeof(i));
```

References

Dimitri Bertsekas and Robert Gallager, Data Networks
Larry Peterson and Bruce Davie, Computer Networks: A Systems Approach