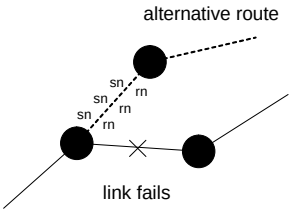
Data Communication Networks

Lecture 4

Saad Mneimneh
Computer Science
Hunter College of CUNY
New York

Link initialization

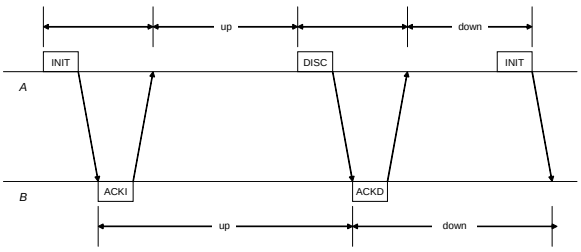
- Stop and Wait or Sliding Window work correctly assuming correct initialization (e.g. sequence numbers)
  - if link fails for a long period of time, transport and/or network layers must take over using alternative paths
  - when failed link eventually returns to operation, it is presented with a completley new stream of packets
  - nodes on both ends of the link need to synchronize their sequence numbers



- More importantly, nodes on both ends of the link must agree at any given instant about whether the link is up or down
- Come up with a protocol for initializing and disconnecting the link

Master-slave protocol

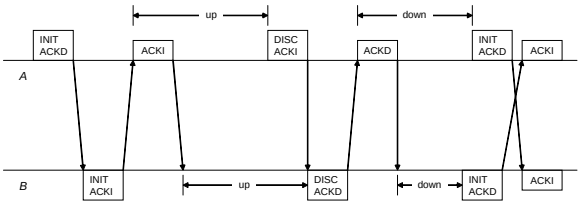
- One node, say A is in charge of determining when the link should be up or down
- Node A then informs node B about the state of the link
- The messages (for this protocol) from A to B form an alternating sequence of INIT and DISC messages
- B responds by ACKI and ACKD



- Conceptually, this can be interpreted as Stop and Wait modulo 2: INIT ($SN = 1$), DISC ($SN = 0$), ACKI ($RN = 0$), ACKD ($RN = 1$)

Balanced protocol

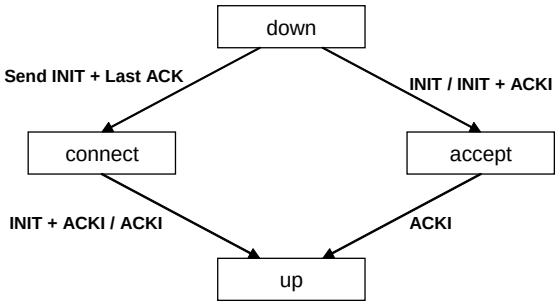
- Need both A and B to be able to initialize/disconnect the link (and possibly send state information)
- Use two master-slave protocols, with node A playing the master for one, and node B playing the master for the other
- ACKI and ACKD are piggybacked on INIT and DISC appropriately
- A node regards the link as up if it is up according to both the $A \rightarrow B$ protocol and the $B \rightarrow A$ protocol
- Therefore, A sends INIT (or DISC) upon receiving INIT (or DISC), same for B



- The initialization part of the protocol
  - sending INIT,
  - waiting for both ACKI and INIT from the other side, and
  - responding to INIT with ACKI

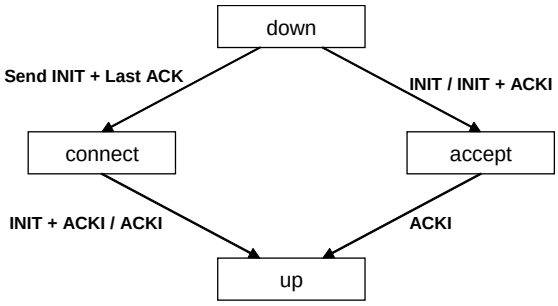
  is called the *three-way handshake*

State diagram for up
(event/action)



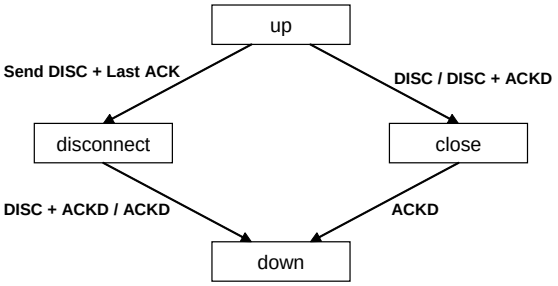
- What if last ack was lost? A thinks connection is up but B thinks it is not up yet
- What if A closes connection before B knows it went up?

State diagram for up
(event/action)



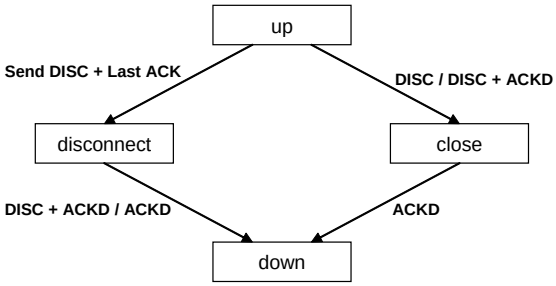
- What if last ack was lost? A thinks connection is up but B thinks it is not up yet
  - B eventually resends INIT
- What if A closes connection before B knows it went up?
  - The last ACK with DISC will tell

State diagram for down
(event/action)



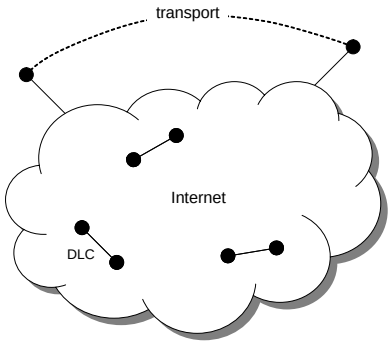
- What if last ack was lost? A thinks the connection is down but B thinks it is not down yet
- What if A reopens connection before B knows it went down?

State diagram for down
(event/action)



- What if last ack was lost? A thinks the connection is down but B thinks it is not down yet
  - B eventually resents DISC
- What if A reopens connection before B knows it went down?
  - the last ACK with INIT will tell

Transport protocols



Transport protocols

- UDP (User Datagram Protocol)
- TCP (Transmission Control Protocol)

Internet (best effort)

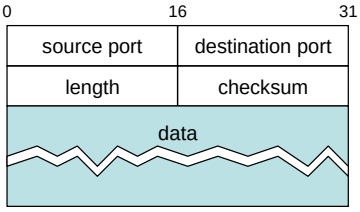
- drops messages
- reorders messages
- delivers duplicate messages
- limits message size (MTU, Maximum Transmission Unit)
- arbitrary delays

We want

- guaranteed message delivery, once, only once, and in order
- large messages
- flow control
- multiple application processes on each node

UDP

- UDP adds a level of demultiplexing to allow multiple application processes to share the network
- Aside from being a demultiplexer, UDP adds no other functionality to the Internet
- Therefore, the only issue is the form of address used to identify processes
  - it is possible to *directly* identify processes with the OS-assigned process id (pid), but such approach is only practical in a closed distributed system with one OS
  - instead, processes are *indirectly* identified using an abstract locator, called **port**
  - UDP header contains a 16 bit port number for both the sender and receiver



- But a 16 bit port number means at most 64K possible port numbers
  - not enough to identify all the processes on all the hosts in the Internet
  - UDP uses <port, host> pair as demultiplexing key, i.e. a process is identified by a port on a particular host

Ports

How does a process learn the port for the process to which it wants to send a message?

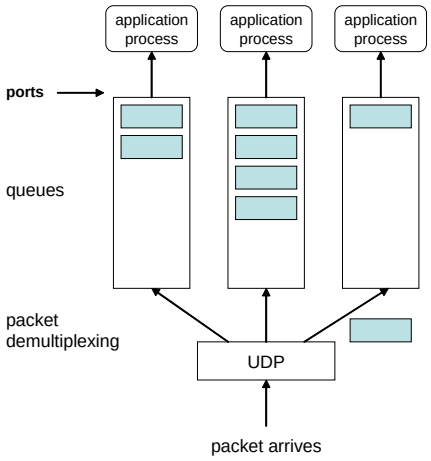
- Typically a clients initiates a messages exchange with a server process (recall from sockets, server listens, client connects)
- Once a client has contacted the server, the server knows the clients port (recall this is usually assigned automatically by socket API)

How does the client learn the server's port to start with?

- Usually a well known port number
  - DNS port 53
  - mail port 25
  - telnet port 23

Needless to say, the socket API provides an implementation of ports. A port is typically implemented as a message queue.

Port implementation



- If queue is full, packet is dropped
- If queue is empty, application process blocks until a message becomes available

Last line of defense

- UDP also ensures correctness of the message by the use of a checksum
  - data is checksummed as a sequence of 16 bit integers
  - the integers are added together using 16 bit ones complement arithmetic
  - checksum is the ones complement of the result
- Relatively weak protection against errors, but easily implemented in software, and adequate since majority of errors are detected previously

```
checksum(int * buf, int count) {
  int sum = 0;
  while (count--) {
    sum+= *buf++;
    if (sum & 0xFFFF0000) {
      sum &= 0xFFFF;
      sum++;
    }
  }
  return ~(sum & 0xFFFF);
}
```

- The checksum is over —UDP header—the content of message —protocol number, source IP address, destination IP address (pseudoheader from IP)—UDPlength field

TCP

- Reliable, in order delivery of a stream of bytes (each byte has a sequence number)
- Full duplex protocol, each TCP connection supports a pair of byte streams, one flowing in each direction
- Flow control mechanism that allows the receiver (on both ends) to limit how much data the sender can transmit at a given time
- Of course, demultiplexing mechanism, like UDP
- Congestion control mechanism, not for the purpose of keeping the sender from overwhelming the receiver, but rather to keep the sender from overloading the network

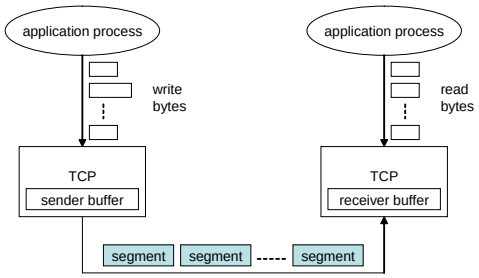
Sliding window

At the heart of TCP is the sliding window algorithm. However, TCP runs over the Internet, not on a single link. Major differences are:

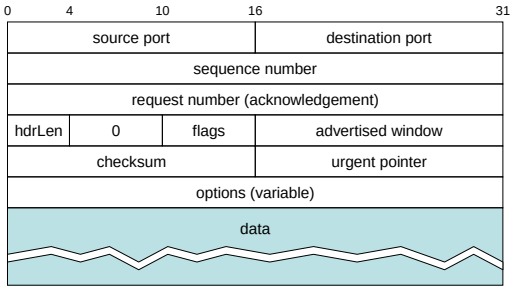
- TCP requires a connection establishment/termination similar to link initialization but slightly different
- With a single link, RTT is fixed. TCP must deal with variable RTT
  - different hosts
  - different times of the day
  - during a connection
- Unlike the case of a single link, packets may be reordered as they cross the Internet. TCP has to be prepared for a very old packet to suddenly show up at receiver, potentially confusing the sliding window algorithm.
- Propagation delay and bandwidth are known on a single link; therefore window size is set. TCP must include a mechanism that each side uses to "learn" what resources (e.g. buffer space) the other side is able to apply for a connection.
- TCP has no idea what links will be traversed in the Internet. Congestion control mechanism needed.

Segments

- TCP is a byte oriented protocol
  - sender writes bytes into a TCP connection
  - receiver reads bytes out of the TCP connection
- TCP does not, itself, transmit individual bytes
  - sender generates bytes into buffer (sliding window)
  - TCP collects enough bytes from the sending process buffer to fill a reasonably sized packet
  - TCP at the receiving end empties the content of the packet into the receiving process buffer
  - receiver reads bytes from buffer (sliding window)
- The packets exchanged between TCP peers are called segments



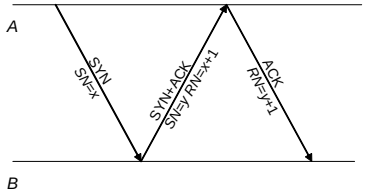
Segment format



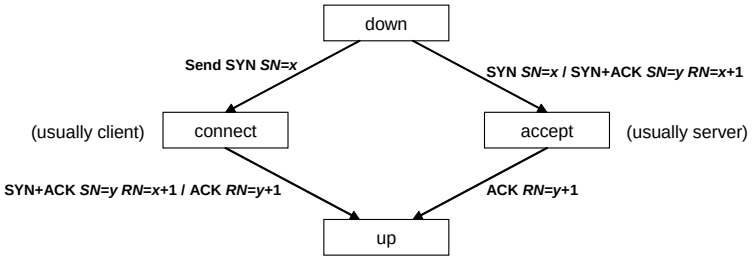
- TCP's demux key is the 4-tuple <source port, source IP, destination port, destination IP>
- Sequence number (SN), request number (RN), and advertised window are used for the sliding window algorithm, but SN is the sequence number of the first byte in the segment
- HdrLen is the length of the header in 32 bit words
- Flags: SYN, FIN, RESET, PUSH, URG, ACK
  - SYN and FIN are used to establish and terminate a TCP connection
  - ACK is set whenever an acknowledgement must be read
  - RESET is used by receiver to abort connection, e.g. receiver confused because it received an unexpected segment
- checksum used same way as for UDP, computer over the TCP header, the TCP data, and the pseudoheader.

Connection establishment

- Similar in concept to link initialization, i.e. the 3-way handshake
- But TCP does not start at $SN = 0$
- Need to protect against two incarnations of the same connection reusing the same sequence numbers too soon (this is not an issue on a single link).
  - connection established
  - few segments exchanged
  - connection aborted by receiver
  - segments are still in the network
  - connection re-established
- TCP requires each side to select an initial starting SN at random
- Sequence numbers need to be exchanged when connection is established
- SYN flag is used and the sequence number field is set
- The result is a 3-way handshake similar to link initialization

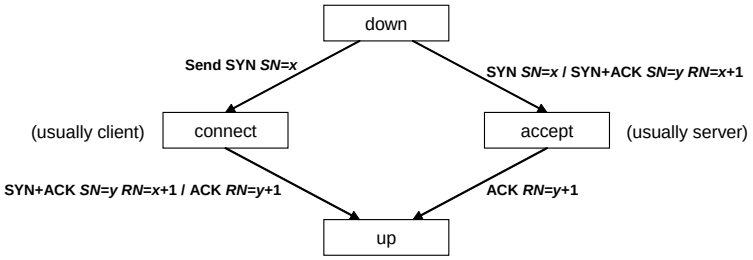


State diagram for up
(event/action)



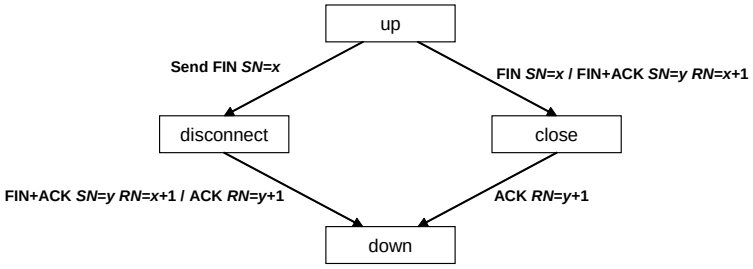
- What if last ack was lost? The client thinks connection is up but the server thinks it is not up yet
- What if the client closes connection before the server knows it went up?

State diagram for up
(event/action)



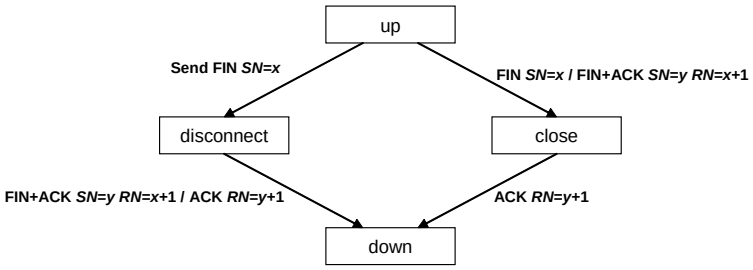
- What if last ack was lost? The client thinks connection is up but the server thinks it is not up yet
  - client starts to send data, so ACK will eventually get to server
- What if the client closes connection before the server knows it went up?
  - client can resend last ACK with FIN (connection is still up)

State diagram for down
(event/action)



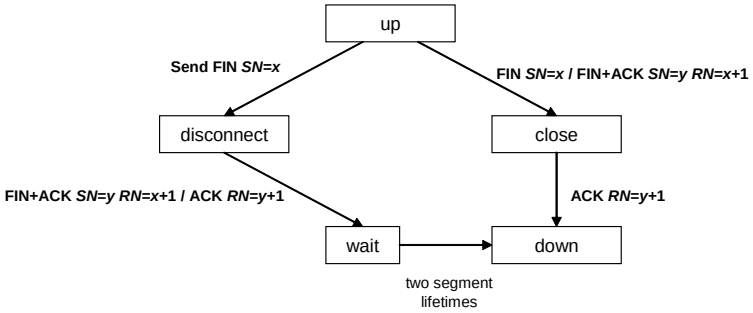
- What if last ack was lost? The client thinks connection is down but the server thinks it is not down yet
- What if the client opens connection before the server knows it went down?

State diagram for down
(event/action)



- What if last ack was lost? The client thinks connection is down but the server thinks it is not down yet
  - client is sending no more data!
- What if the client opens connection before the server knows it went down?
  - no last ACK, connection is gone
  - maybe it's another client with the same port (since connection was closed) number and host
  - if server resends FIN (and delayed), it might be wrongly interpreted as termination

State diagram for down
(event/action)



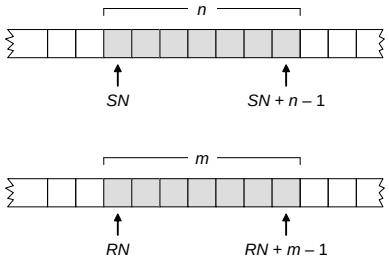
Sliding window revisited

The sliding window algorithm is essentially the same with mainly two differences:

- Flow control: the receiver advertises a window size to the sender (through the TCP header)
- Protection against wraparound: TCP uses 32 bit long sequence numbers

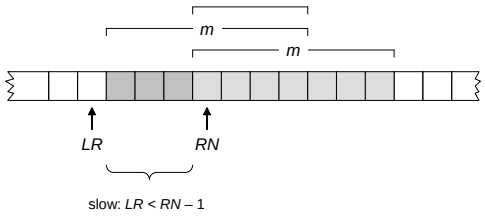
We will discuss the advertised window first, but for simplicity:

- Ignore that both buffers and sequence numbers are of some finite size
- Do not distinguish between a byte's sequence number and its position in the buffer



Receiver side

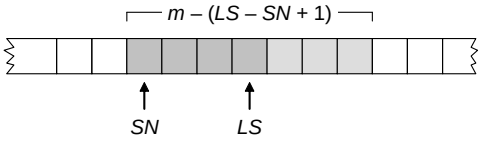
- Let LR be the last byte read by the receiver (delivered to application)
- Ideally, $LR = RN - 1$ (receiver is fast enough)
- Generally, $LR \leq RN - 1$



- The receiver will advertise a window size of $m' \leftarrow m - [(RN - 1) - LR]$

Sender side

- The sender sets its window size to the advertised window $n' \leftarrow m$
- Sender cannot send bytes unless window size is $n' > 0$
- If $n' = 0$, eventually sender's buffer fills up, and TCP blocks the sending process.
- If sender wants to be more cautious, it can set its window size to $n \leftarrow m - (LS - SN + 1)$, where LS is the last byte sent by the sender (over the network)



- In this case, if receiver advertises a window size less or equal to $LS - SN + 1$ (receiver has not seen these yet), then sender stops sending immediately

Probing

- How does the sender know that the advertised window is no longer 0?
  - the sender is not permitted to send any more data
  - the receiver sends ACKs in response to received data
  - no way to discover that advertised window is no longer 0
- Whenever the receiver advertises a window size of 0, the sender persists in sending a segment with 1 byte of data every so often
  - these will probably be not accepted at the receiver
  - but eventually one of these 1 byte segments will trigger a response that reports a non-zero advertised widow
- This is called *probing*, and it is done in this way to simplify the receiver
  - the receiver simply responds to segments from the sender and never initiates activity on its own
  - smart sender/dumb receiver
  - a similar simplification is the replacement of ACKS and NAKS with RN

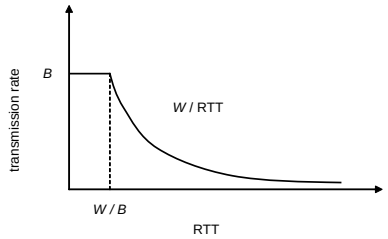
Wrap around

- TCP uses a 32 bit sequence number, we have 2^{32} distinct numbers
- The window size is a 16 bit number, so the maximum window size is $2^{16} - 1$ bytes ($\approx$ 64 KB)
- $2^{32} >> 2(2^{16} - 1)$, so theoretically, wrapping around should not be a problem
- But the ordered delivery condition true on a single link is not satisfied in the Internet!
- TCP must be prepared for a very old segment to suddenly appear at the receiving side
- Solution: Packets cannot survive in the Internet for longer than the Maximum Segment Lifetime MSL, which is 120 seconds.
- Therefore, we need to make sure that the sequence number does not wrap around within a 120 second period of time
- How long does it take the sequence number to wrap around?
  - it depends on the network
  - e.g. Ethernet: bandwidth = 10Mbps, we need $\frac{2^{32} \times 8}{10 \times 10^6}$ seconds to transmit 2^{32} bytes (to wrap around). This is 57 min.
- On faster networks (e.g. 622Mbps and 1.2 Gbps), the time to wrap around is much smaller than 120 seconds.

Window size

How big should the window size W be in bits?

- The window size should be big enough to use the full capacity of the network
- If the bandwidth is B, then the sender should be able to send B bps
- But the sender can only send W bits every $RTT + MSS/B$ seconds, where MSS is the Maximum Segment Size
- sender is limited to $\min(B, W/RTT)$ ($RTT >> MSS/B$)



- We need RTT$\leq W/B$; therefore, $W \geq RTT \times B$
- TCP can support 64 KB windows only

How does TCP send data?

Why is that an issue?

- TCP is a byte stream, so every byte has a sequence number
- But TCP sends bytes in segments
- How does TCP decide when to send?
  - TCP uses a Maximum Segment Size (MSS) imposed by the Maximum Transmission Unit (MTU) allowed by the immediate underlying network (prevents additional segmentation at the network level)
  - but should TCP wait to have enough bytes to fill an MSS?
  - or should it send data immediately?
- Before we answer this question, why would TCP have few bytes to send
  - application process is providing few bytes / seconds, e.g. Telnet application sending one character at a time (small packet problem)
  - window size is small (silly window syndrome)
- A solution to the small packet problem is also a solution to the silly window syndrome

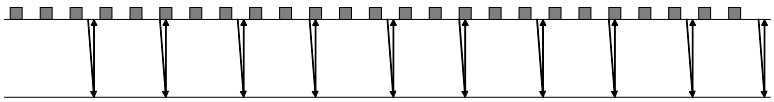
Send vs. Wait

- If TCP sends immediately, then we have too much overhead
  - e.g. 1 byte of data with 20 bytes of TCP header and 20 bytes of IP header, overhead is 40/1
  - while this is acceptable on LANs like Ethernet, it is not acceptable on highly congested WANs
- An alternative is for TCP to wait until it has enough data to fill an MSS, but how much time?
  - is user on Telnet application generating more characters?
  - when will the receiver open up the window?
  - if sender waits too long, it hurts interactive applications like Telnet (imagine you hit return and nothing happens)
  - if sender does not wait long enough, silly window syndrome persists
- Use a timer, say 500 ms. Every time the timer fires, TCP is allowed to send a small segment (less than MSS)
  - Problem: Cannot make everyone happy!

Why is a timer bad?

Imagine a user on a Telnet application generating 25 characters, one character every 200 ms

- LAN with 50 ms RTT



  - response every 2 or 3 characters
  - overhead 16/1 (400 bytes overhead)
  - not a good compromise
- WAN with 5 sec RTT
  - responsiveness is not that bad since we have to wait 5 sec for first response
  - but we still have too many packets congesting the WAN
  - we should have waited more (but then LAN will be worse)

Nagle's algorithm

- We somehow need an adaptive timer
- Use a self-clocking algorithm at sender
  - as long as TCP has any data in transit, the sender will receive an ACK
  - treat this ACK as a timer firing

```
while more data
    if both available data and the window ≥ 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
```

- When window size becomes 0, receiver does not advertise window size > 0 until window size $\geq MSS$

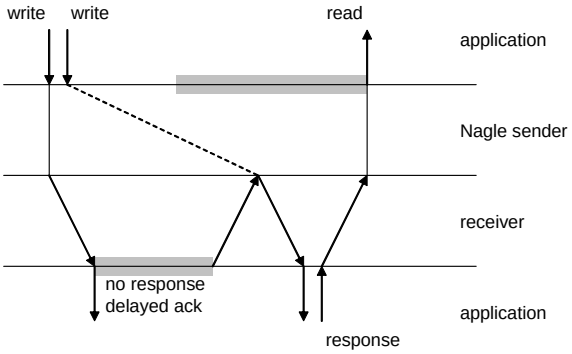
Revisit the example

Imagine a user on a Telnet application generating 25 characters, one character every 200 ms

- LAN with 50 ms RTT
  - since characters are generated every 200 ms, every character will see no data in transit
  - every character is sent immediately in a small segment
  - response is fast
  - overhead is 40/1 again
  - appropriate for LAN
- WAN with 5 sec RTT
  - the first character will see no data in transit
  - the rest of the characters will wait the ACK
  - all 24 characters are sent in one segment after the ACK
  - overhead is 3.2/1 (only 80 bytes overhead)
  - only 2 packets injected in network
  - appropriate for WAN

To Nagle or not to Nagle

- Nagle's algorithm performs badly when combined with delayed ACKs
  - The receiver piggybacks the ACK on the response
  - If there is no response, the receiver waits for up to, say 200ms, and then sends the (now delayed) ACK



- TCP provides a way to disable Nagle's algorithm (socket TCP option NO_DELAY)

```
socklen_t i=1;
setsocketopt(sock, IPPROTO_TCP, TCP_NODLAY, &i, sizeof(i));
```