EEC-484/584
Computer Networks

Lecture 17

Wenbing Zhao

wenbing@ieee.org

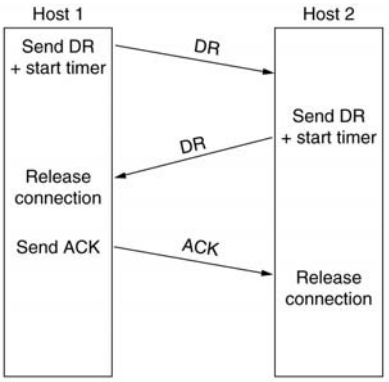
(Lecture nodes are based on materials supplied by
Dr. Louise Moser at UCSB and Prentice-Hall)

Outline

- Review of last lecture
- The Internet Transport Protocol: TCP
  - TCP transmission policy
  - TCP congestion control
  - TCP timer management
  - Wireless TCP and UDP
  - Transactional TCP
- Reminder: Midterm #2, Nov 9 Wednesday
  - Chapters 5-6
  - Closed book, closed notes

Connection Release

- In practice, three-way handshake works most of the time
  - Host 1 sends DR (Disconnect Request), starts timer in case DR lost
  - Host 2 sends DC (Disconnect Confirm), starts timer in case DC lost
  - When DC arrives, host 1 sends ACK, deletes connection
  - When ACK arrives, host 2 deletes connection

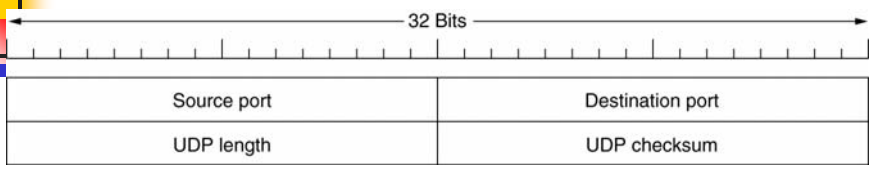


Flow Control and Buffering

- Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (…) indicates a lost TPDU

	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	•••	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	•••	<ack = 6, buf = 4>	←	Potential deadlock

UDP Header



- The ***source port*** and ***destination port*** serve to identify the end points within the source and destination machines.
  - When a UDP packet arrives, its payload is handed to the process attached to the destination port.
  - This attachment occurs when BIND primitive is used
- The ***UDP length*** field includes the 8-byte header and the data
- The ***UDP checksum*** is optional and stored as 0 if not computed (a true computed 0 is stored as all 1s)

The Internet Transport Protocols: TCP

- Introduction to TCP
- The TCP Service Model
- The TCP Protocol
- The TCP Segment Header
- TCP Connection Establishment
- TCP Connection Release
- TCP Connection Management Modeling

The TCP Service Model

- Requires both sender and receiver to create socket
- Each socket has socket number (address) consisting of IP address of host and 16-bit port number
- Port is TCP name for TSAP
- Connections are identified by the socket identifiers at both ends, i.e., (socket1, socket2)
- Port numbers below 1024 are called **well-known ports** and are reserved for standard services
  - E.g., 21 for FTP, 23 for Telnet
- Full-duplex, point-to-point

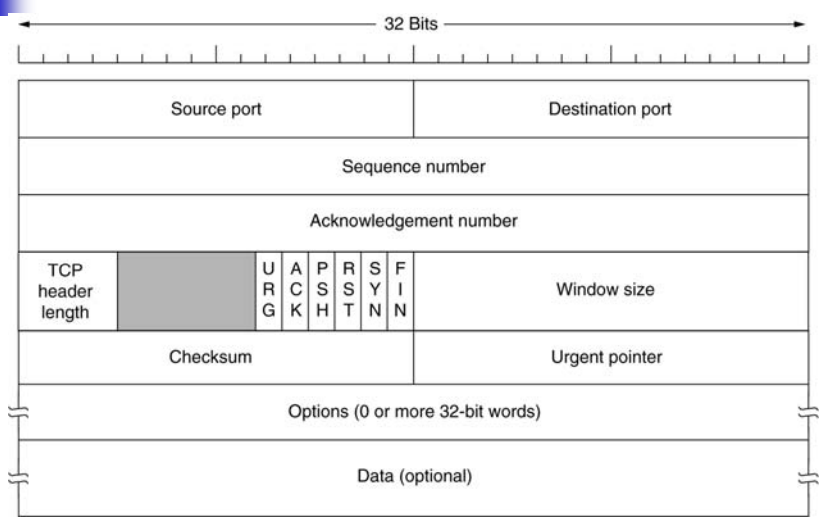
TCP Protocol

- Sequence numbers used for ACKs and also for window mechanism
- Sender and receiver TCP entities exchange data in the form of segments
- **Segment** – fixed 20 byte header + optional part followed by data
- TCP software decides size of segments, accumulate data from several writes into 1 segment, or split data from one write over multiple segments

TCP Protocol

- Uses sliding window for flow control
  - Resembles go-back-n protocol
  - No selective ack, or nack, e.g., if 1-1024 and 2049-3072 are received, can only ack 1025
  - RFC 1106 propose a solution, using TCP options

The TCP Segment Header

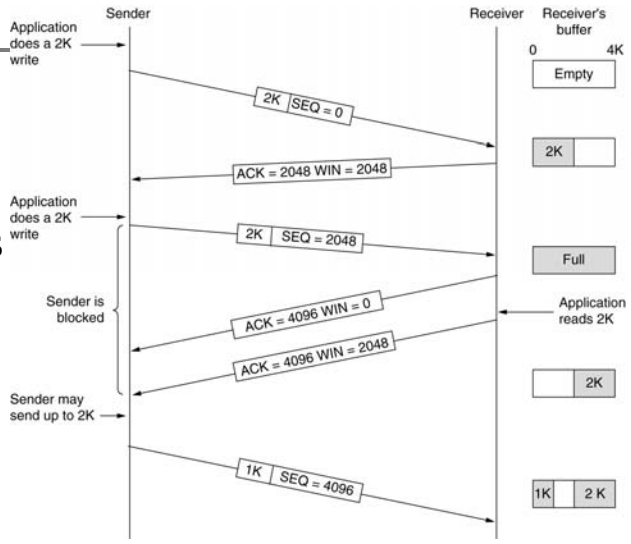


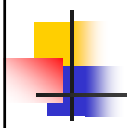
TCP Transmission Policy

- Window management not directly tied to ACKs
  - Even if the receiver has acked all the segments sent by the sender, the sender can send new segments only if the receiver has room to receive them
- What if the receiver's window drops to 0 ?
  - Sender may not normally send segments with two exceptions
  - Exception 1: urgent data may be sent, e.g., to allow user to kill process running on the remote machine
  - Exception 2: sender may send a 1-byte segment to make the receiver re-announce the next byte expected and window size

TCP Transmission Policy

- Window management not directly tied to ACKs





TCP Transmission Policy

- To improve performance
  - Sender not required to transmit data as soon as gets data from application
  - Receiver not required to transmit ACKs immediately

TCP Transmission Policy

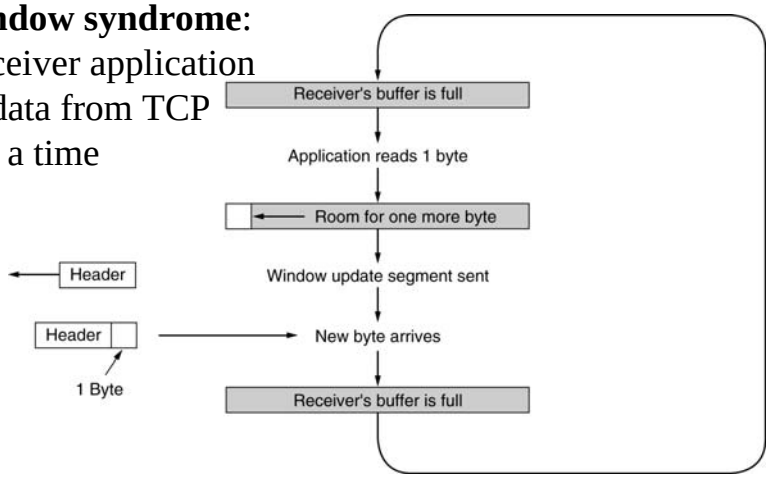
- **Problems when the sender sends 1 byte at a time,** e.g., telnet/rsh/ssh program
  - Sender sends 1 byte (e.g., typed one character in a editor)
  - A segment of 1 byte is sent to the remote machine (41-byte IP packet)
  - Remote machine acks immediately (40-byte IP packet)
  - Editor (in remote machine) program reads the received 1 byte, a windows update segment is sent to user (40-byte IP packet)
  - Editor program echoes the 1 byte received to the user terminal (41-byte IP packet)
  - In all, 162 bytes of bandwidth used, 4 segments are sent for each character typed

TCP Transmission Policy

- **Nagle's algorithm**: solution for the 1-byte-at-a-time sender problem
  - When sender application passes data to TCP one byte at a time
    - Send first byte
    - Buffer the rest until first byte ACKed
    - Then send all buffered bytes in one TCP segment
    - Start buffering again until all ACKed
  - Implemented widely in TCP, can be disabled/enabled by using socket options
  - For some application, it is necessary to disable the Nagle's algorithm, *e.g.*, X Windows program over Internet, to avoid erratic mouse movement, *etc*.

TCP Transmission Policy

- **Silly window syndrome**: when receiver application accepts data from TCP 1 byte at a time



TCP Transmission Policy

- **Clark's algorithm** (to avoid the silly window syndrome)
  - Receiver should not send window update until it can handle max segment size it advertised when connection established or its buffer is half empty, whichever is smaller
  - Sender should wait until it has accumulated enough space in window to send full segment or one containing at least half of receiver's buffer size
- Nagle's algorithm and Clark's algorithm are complementary

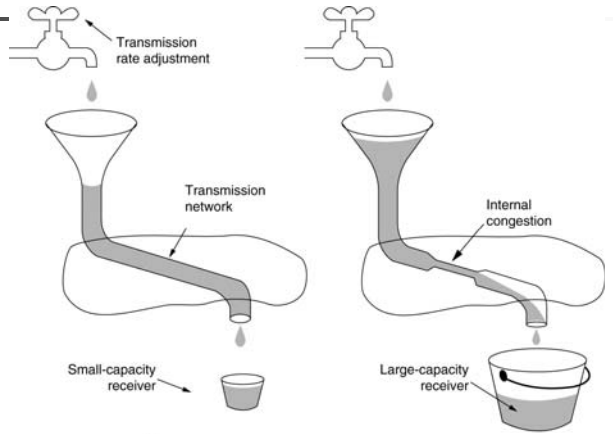
TCP Congestion Control

- Two limiting factors on sending rate
  - Limited network capacity: problem of congestion control
  - Limited receiver capacity: problem of flow control
- Congestion control – slow down data rate by dynamically changing window size
- To detect congestion, use time outs
  - Assumes the loss of a packet is caused by congestion rather than corruption during transmission – applicable for wired network only
  - Repeated acks for same sequence number is also used to detect congestion

TCP Congestion Control

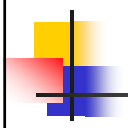
- To control the sending rate, each sender maintains two windows
  - Window that receiver granted (flow control)
  - Sender congestion window
  - Number of bytes sender can transmit = minimum of two window sizes

TCP Congestion Control



A fast network feeding a low capacity receiver

A slow network feeding a high-capacity receiver



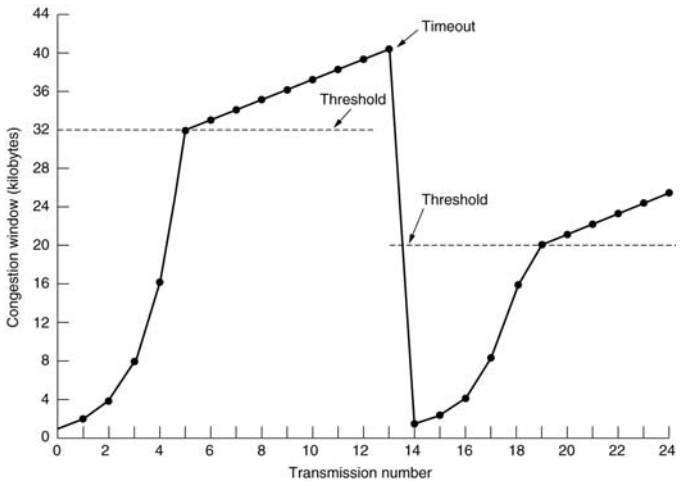
Slow Start Algorithm

- When connection is established, sender initializes congestion window size to size of max segment in use on connection, it then sends one max segment
- If segment is acked before timer goes off, it increases congestion window size by max segment size, it then sends two segments
- Doubling each time (congestion window size is increased exponentially) until timeout occurs or receiver's window size is reached

Internet Congestion Control Algorithm

- Three parameters
  - Receiver's window size
  - Sender's congestion window size
  - Threshold, initially 64KB
- When timeout occurs,
  - Threshold is set to half **current** congestion window size
  - Congestion window size is set to one max segment size
- Use slow start algorithm but stop when threshold is reached, then increase congestion window size linearly

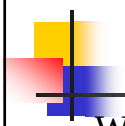
TCP Congestion Control



An example of the Internet congestion algorithm

TCP Congestion Control

- Fast retransmission algorithm
  - An ACK is generated and sent immediately when an out-of-order segment is received to notify the sender
  - When 3 or more duplicated ACKs are received in a row, it is taken as strong evidence that a segment has been lost
  - The segment that appears to have been lost is retransmitted before the retransmission timer expires
- Fast recovery algorithm
  - On fast retransmission, slow start algorithm is not used
  - Why fast recovery ?
    - Data are still flowing between two ends



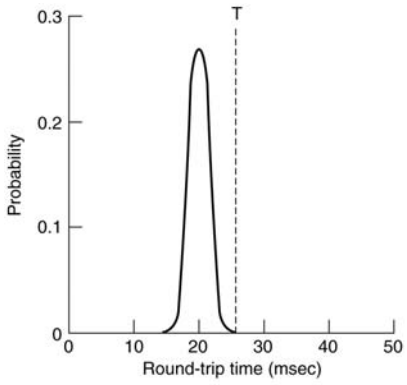
Fast Recovery Algorithm

- When third duplicate ACK is received
  - Set threshold to half the current congestion window
  - Set congestion window to (threshold + 3 × segment size)
- Each time another duplicate ACK arrives
  - Increment congestion window by the segment size
  - Transmit a packet (if allowed by new congestion window)
- When the next ACK arrives that acknowledges new data, set congestion window to threshold value
  - This should be the ACK of the retransmission from step 1
  - Additionally, this ACK should acknowledge all the intermediate segments sent between the lost packet and the receipt of the first duplicate ACK

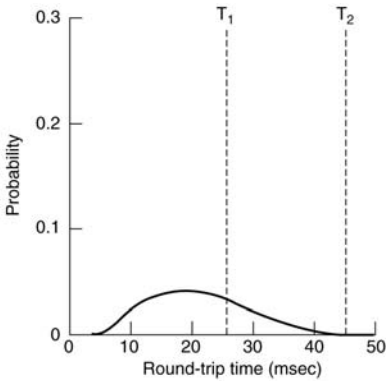
TCP Timer Management

- Retransmission timer (most important)
  - When TCP segment is sent, retransmission timer is started
  - If segment is acked before timer expires, timer is stopped
  - Else, segment is retransmitted, timer is restarted
- Question: how long should timeout be?
  - If too short, unnecessary retransmissions
  - If too long, long retransmission delay when packet is lost
  - As congestion occurs, mean and variance can change rapidly
- Solution: use dynamic algorithm that adjusts timeout based on continuous measurement of network performance

TCP Timer Management



Probability density of ACK arrival times in the data link layer



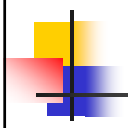
Probability density of ACK arrival times for TCP

TCP Timer Management

- The algorithm generally used by TCP is due to Jacobson (1988)
- For each connection, TCP maintains a variable, *RTT*, that is the best current estimate of the round-trip time to the destination in question
- *RTT* is updated according to formula
  $RTT = \alpha RTT + (1-\alpha)M$
  - M is the newly measured roundtrip time
  - α is a smoothing factor that determines how much weight is given to the old value. Typically $\alpha = 7/8$

TCP Timer Management

- Timeout = βRTT
  - Constant value was inflexible because it failed to respond when the variance went up
  - Need another smoothed variable, *D*, the deviation.
  - Whenever an acknowledgement comes in, the difference between the expected and observed values, $|RTT - M|$, is computed.
  - A smoothed value of this is maintained in *D* by the formula $D = \alpha D + (1-\alpha)|RTT - M|$
  - Time out is set to $RTT + 4 \times D$

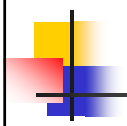


TCP Timer Management

- How to update RTT if there is a timeout?
  - **Karn's algorithm**: The timeout is doubled on each failure until the segments get through the first time
    - Do not update *RTT* on any segments that have been retransmitted

Persistence Timer

- **Persistence timer** – used on the sending side. Designed to prevent the following deadlock
  - The receiver sends an acknowledgement with a window size of 0, telling the sender to wait.
  - Later, the receiver updates the window, but the packet with the update is lost.
  - Now both the sender and the receiver are waiting for each other to do something.
- When the persistence timer goes off, the sender transmits a probe to the receiver.
- The response to the probe gives the window size. If it is still zero, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent



TCP Timer Management

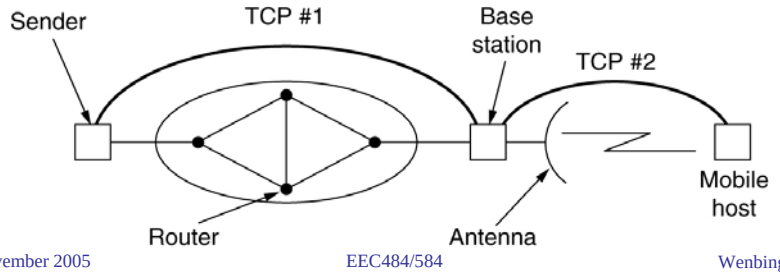
- **Keepalive timer** - when a connection has been idled for a long time, the keepalive timer may go off to cause one side to check whether the other side is still there. If it fails to respond, the connection is terminated
- The last timer used on each TCP connection is the one used in the *TIMED WAIT* state while closing
  - It runs for **twice the maximum segment lifetime** to make sure that when a connection is closed, all packets created by it have died off

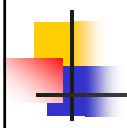
Wireless TCP and UDP

- The correctness of TCP protocol is independent of the technology of the underlying network layer. However, it matters a lot on the performance of TCP
- The principal problem is the congestion control algorithm
  - If there is a timeout (or repeated ACKs), a network congestion is assumed by TCP, consequently, the sending rate is reduced sharply
  - In wireless network, the probability of packet loss is significant due to unreliable transmission link
  - In effect, when a packet is lost on a wired network, the sender should slow down. When one is lost on a wireless network, the sender should retransmit

Wireless TCP

- **Indirect TCP** - split the TCP connection into two separate connections.
  - The first connection goes from the sender to the base station.
  - The second one goes from the base station to the receiver.
  - The base station simply copies packets between the connections in both directions





Problems with Indirect TCP

- Violate TCP semantics:
  - Since each part of the connection is a full TCP connection, the base station acknowledges each TCP segment in the usual way
  - Receipt of an acknowledgement by the sender does not mean that the receiver got the segment, only that the base station got it

Wireless TCP – Using Snooping Agent

- A snooping agent runs in the base station. It observes and caches TCP segments going out to the mobile host and ACKs coming back from mobile
  - It uses a timer with short timeout for retransmission. The timer is started when it sees a segment sent to the mobile
  - It suppresses duplicated ACKs from mobile
  - It sends NACK for specific segments if a gap in sequence number is seen for incoming segments
- Drawback
  - Cannot completely avoid the sender from invoking the congestion control algorithm if wireless link is too lossy

Wireless UDP

- Main trouble is that programs use UDP expecting it to be highly reliable
  - They know that no guarantees are given, but they still expect it to be near perfect
  - In a wireless environment, UDP will be far from perfect
  - For programs that can recover from lost UDP messages but only at considerable cost, going from an environment where messages theoretically can be lost but rarely are, to one in which they are constantly being lost can result in a performance disaster

Transactional TCP

- RPC using TCP might not be efficient
- T/TCP - an experimental TCP variant
  - Modify the standard connection setup sequence slightly to allow the transfer of data during setup