

Computer Networking

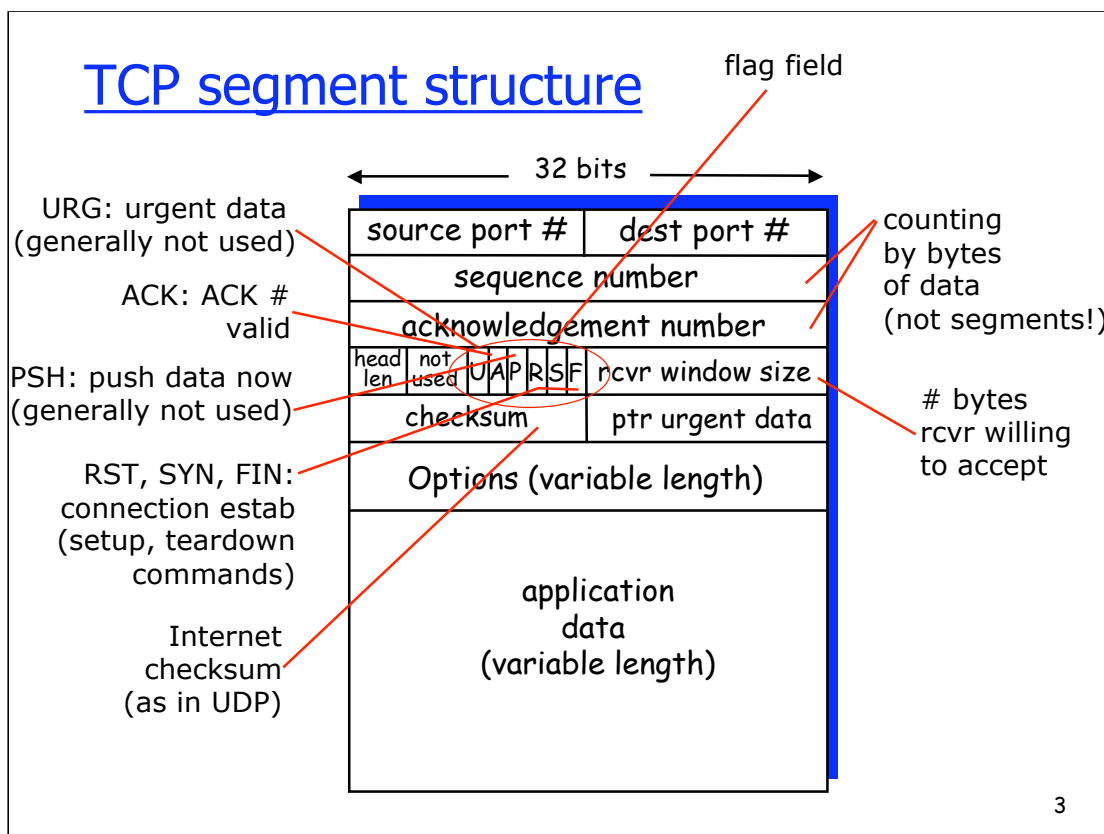
TCP

Prof. Andrzej Duda
duda@imag.fr

`http://duda.imag.fr`

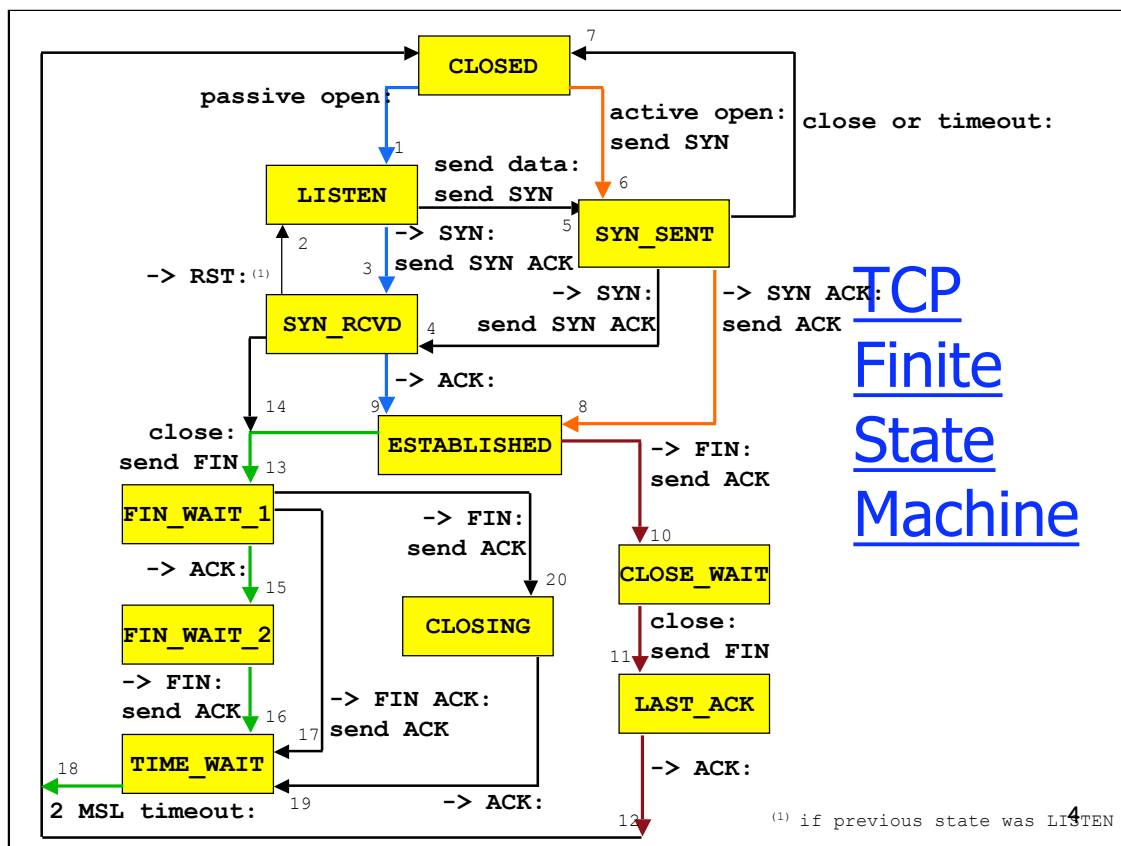
Details of the TCP protocol

- More details on TCP
 - connection management
 - reliable transfer
 - interactive traffic
 - Nagle algorithm
 - silly window syndrome
 - RTT estimation and Karn's rule
 - fast retransmit
 - congestion control



The TCP segment consists of header fields and a data field. The MSS limits the maximum size of a segment's data field. When TCP sends a large file, it typically breaks the file into chunks of size MSS. As with UDP, the header includes **source and destination port numbers**, that are used for multiplexing/ demultiplexing data from/to upper layer applications. Also, as with UDP, the header includes a **checksum field**. A TCP segment header also contains the following fields:

- The 32-bit **sequence number field** and the 32-bit **acknowledgment number field** are used by the TCP sender and receiver in implementing a reliable data-transfer service, as discussed below.
- The 16-bit **window-size** field is used for flow control. We will see shortly that it is used to indicate the number of bytes that a receiver is willing to accept.
- The 4-bit **length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field, discussed below. (Typically, the options field is empty, so that the length of the typical TCP header is 20 bytes.)
- The optional and variable length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A timestamping option is also defined. See RFC 854 and RFC 1323 for additional details.
- The **flag field** contains 6 bits. The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid. The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown, as we will discuss at the end of this section. When the **PSH** bit is set, this is an indication that the receiver should pass the data to the upper layer immediately. Finally, the **URG** bit is used to indicate that there is data in this segment that the sending-side upper layer entity has marked as "urgent." The location of the last byte of this urgent data



TCP Finite State Machine

If the application issues a half-close (eg. shutdown(1)) then data can be received in states **FIN_WAIT_1** and **FIN_WAIT_2**.

“TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request” (RFC 793). The connection stays in that state for a time of $2 \times \text{MSL}$, where MSL = maximum segment lifetime (typically 2×2 mn). This also has the effect that the connection cannot be reused during that time.

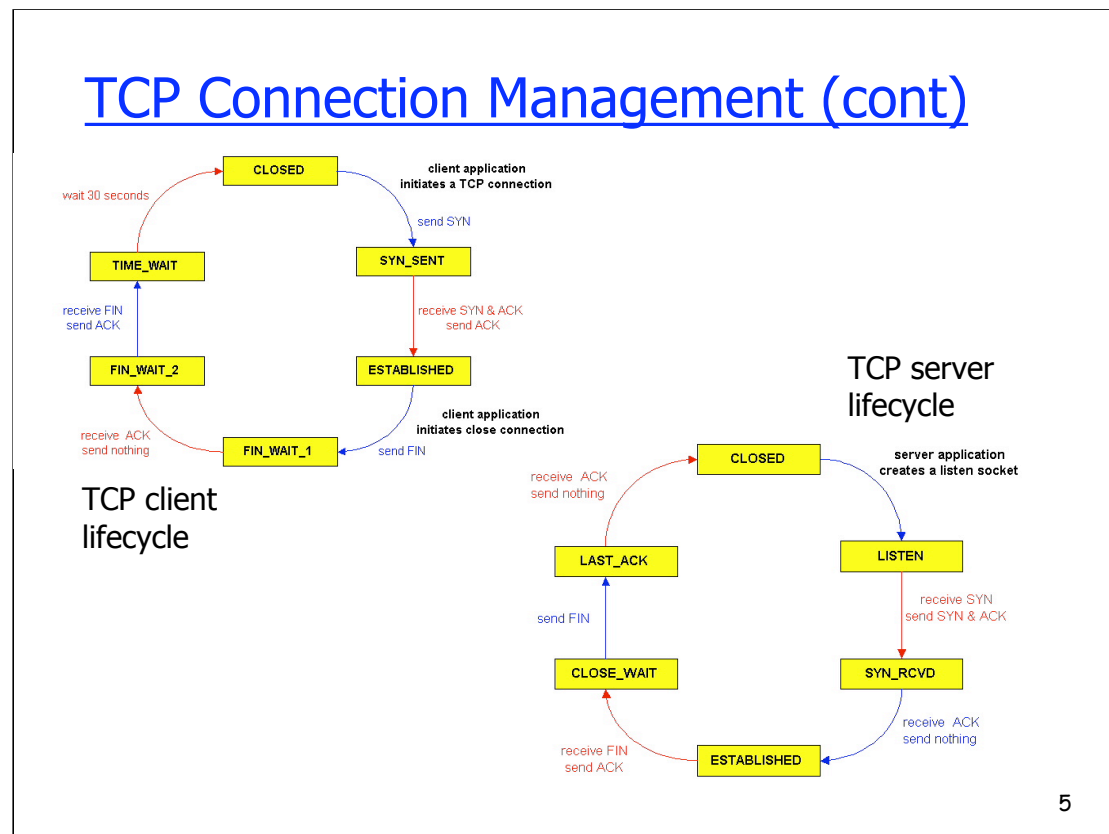
Entering the **FIN_WAIT_2** state on a full close (not on a half-close) causes the **FIN_WAIT_2** timer to be set (eg. to 10 mn). If it expires, then it is set again (eg. 75 sec) and if it expires again, then the connection is closed. This is to avoid connections staying in the half-close state for ever if the remote end disconnected.

Transitions due to RESET segments except the 2nd case are not shown on the diagram

There is a maximum number of retransmissions allowed for any segment. After R_1 retransmissions, reachability tests should be performed by the IP layer. After unsuccessful transmission lasting for at least R_2 seconds, the connection is aborted. Typically, $R_1 = 3$ and R_2 is a few minutes. R_2 can be set by the application and is typically a few minutes. Transitions due to those timeouts are not shown.

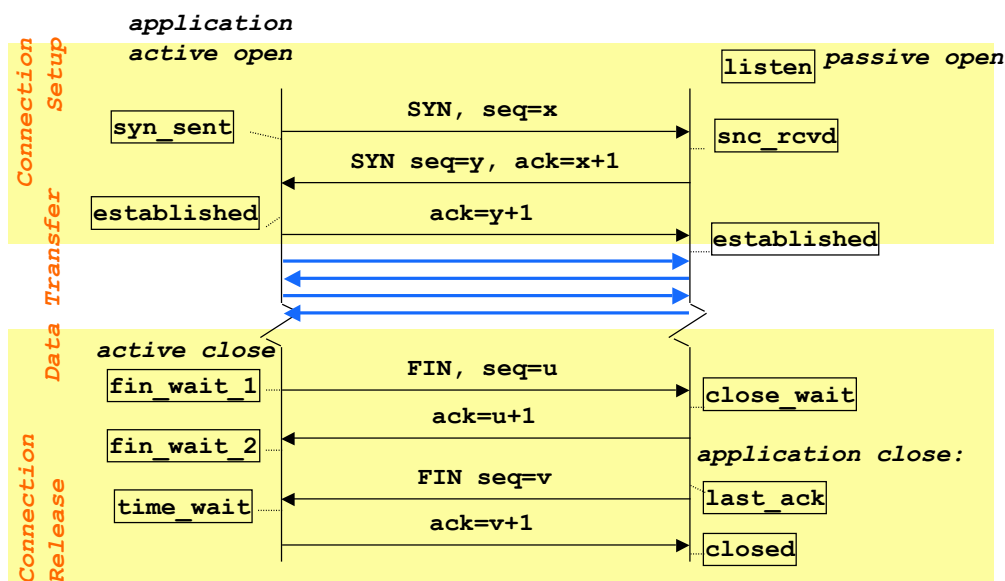
The values are usually set differently for a SYN packet. With BSD TCP, if the connection setup does not succeed after 75 sec (= connectionEstablishmentTimer), then the connection is aborted.

The diagram does not show looping transitions; for example, from **TIME-WAIT** state, reception of a FIN packet causes an ACK to be sent and a loop into the **TIME-WAIT** state itself.



During the life of a TCP connection, the TCP protocol running in each host makes transitions through various **TCP states**. The client TCP begins in the closed state. The application on the client side initiates a new TCP connection (by creating a Socket object in our Java examples from Chapter 2). This causes TCP in the client to send a SYN segment to TCP in the server. After having sent the SYN segment, the client TCP enters the SYN_SENT state. While in the SYN_SENT state, the client TCP waits for a segment from the server TCP that includes an acknowledgment for the client's previous segment as well as the SYN bit set to 1. Once having received such a segment, the client TCP enters the ESTABLISHED state. While in the ESTABLISHED state, the TCP client can send and receive TCP segments containing payload (that is, application-generated) data. Suppose that the client application decides it wants to close the connection. (Note that the server could also choose to close the connection.) This causes the client TCP to send a TCP segment with the FIN bit set to 1 and to enter the FIN_WAIT_1 state. While in the FIN_WAIT_1 state, the client TCP waits for a TCP segment from the server with an acknowledgment. When it receives this segment, the client TCP enters the FIN_WAIT_2 state. While in the FIN_WAIT_2 state, the client waits for another segment from the server with the FIN bit set to 1; after receiving this segment, the client TCP acknowledges the server's segment and enters the TIME_WAIT state. The TIME_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost. The time spent in the TIME_WAIT state is implementation-dependent, but typical values are 30 seconds, 1 minute, and 2 minutes. After the wait, the connection formally closes and all resources on the client side (including port numbers) are released.

TCP Connection Phases



6

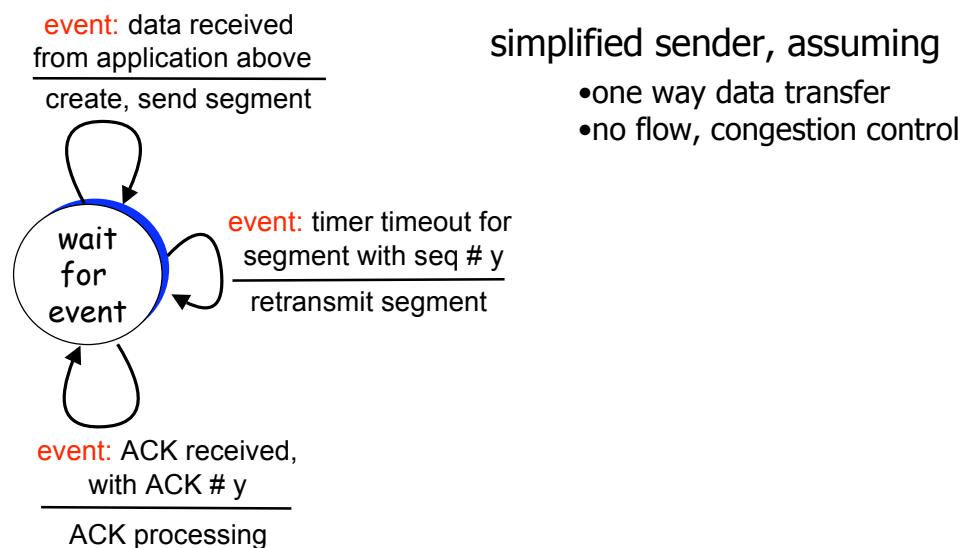
Before data transfer takes place, the TCP connection is opened using SYN packets. The effect is to synchronize the counters on both sides.

The initial sequence number is a random number.

The connection can be closed in a number of ways. The picture shows a graceful release where both sides of the connection are closed in turn.

Remember that TCP connections involve only two hosts; routers in between are not involved.

TCP: reliable data transfer



7

TCP creates a **reliable data-transfer service** on top of IP's unreliable best-effort service. TCP's reliable data-transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence, that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection. There are three major events related to data transmission/retransmission at a simplified TCP sender. Let's consider a TCP connection between host A and B and focus on the data stream being sent from host A to host B. At the sending host (A), TCP is passed application-layer data, which it frames into segments and then passes on to IP. The passing of data from the application to TCP and the subsequent framing and transmission of a segment is the first important event that the TCP sender must handle. Each time TCP releases a segment to IP, it starts a timer for that segment. If this timer expires, an interrupt event is generated at host A. TCP responds to the timeout event, the second major type of event that the TCP sender must handle, by retransmitting the segment that caused the timeout. The third major event that must be handled by the TCP sender is the arrival of an acknowledgment segment (ACK) from the receiver (more specifically, a segment containing a valid ACK field value). Here, the sender's TCP must determine whether the ACK is a **first-time ACK** for a segment for which the sender has yet to receive an acknowledgment, or a so-called **duplicate ACK** that re-acknowledges a segment for which the sender has already received an earlier acknowledgment. In the case of the arrival of a first-time ACK, the sender now knows that *all* data up to the byte being acknowledged has been received correctly at the receiver. The sender can thus update its TCP state variable that tracks the sequence number of the last byte that is known to have been received correctly and in order at the receiver.

TCP: reliable data transfer

Simplified
TCP
sender

```

00 sendbase = initial_sequence number
01 nextseqnum = initial_sequence number
02
03 loop (forever) {
04   switch(event)
05     event: data received from application above
06       create TCP segment with sequence number nextseqnum
07       start timer for segment nextseqnum
08       pass segment to IP
09       nextseqnum = nextseqnum + length(data)
10     event: timer timeout for segment with sequence number y
11       retransmit segment with sequence number y
12       compute new timeout interval for segment y
13       restart timer for sequence number y
14     event: ACK received, with ACK field value of y
15       if (y > sendbase) { /* cumulative ACK of all data up to y */
16         cancel all timers for segments with sequence numbers < y
17         sendbase = y
18       }
19       else { /* a duplicate ACK for already ACKed segment */
20         increment number of duplicate ACKs received for y
21         if (number of duplicate ACKs received for y == 3) {
22           /* TCP fast retransmit */
23           resend segment with sequence number y
24           restart timer for segment y
25         }
26       } /* end of loop forever */

```

8

TCP ACK generation [RFC 1122, RFC 2581]

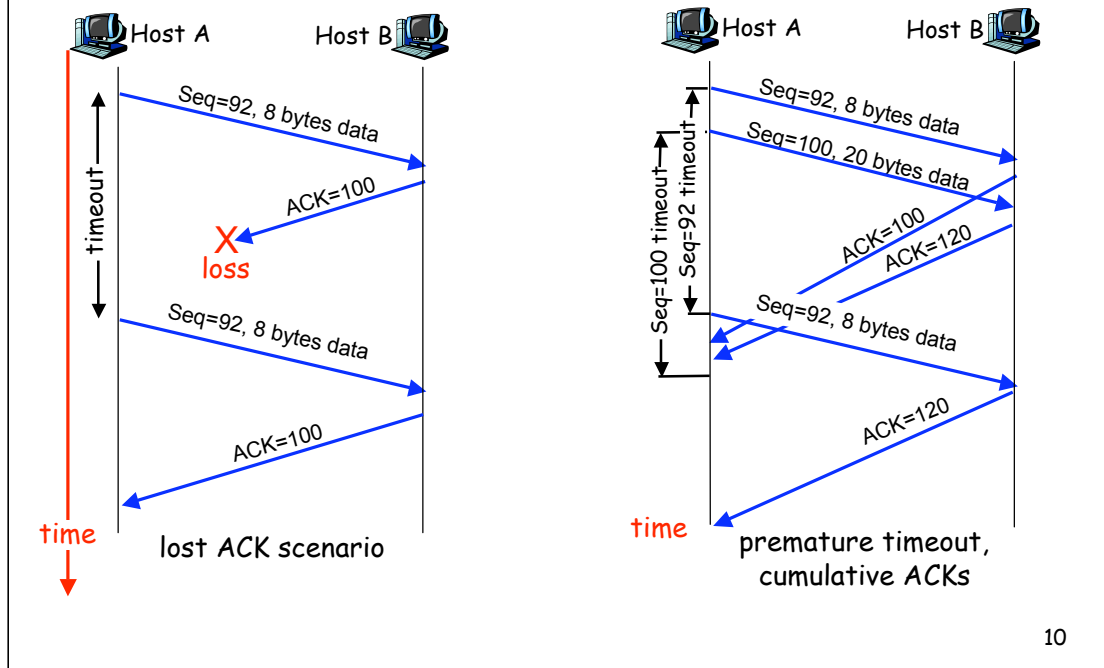
Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 200ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expected seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

Duplicated ACKs can be used for **Fast Retransmission**

9

To understand the sender's response to a duplicate ACK, we must look at why the receiver sends a duplicate ACK in the first place. The table summarizes the TCP receiver's ACK generation policy. When a TCP receiver receives a segment with a sequence number that is larger than the next, expected, in-order sequence number, it detects a gap in the data stream--that is, a missing segment. Since TCP does not use negative acknowledgments, the receiver cannot send an explicit negative acknowledgment back to the sender. Instead, it simply re-acknowledges (that is, generates a duplicate ACK for) the last in-order byte of data it has received. If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost. In this case, TCP performs a **fast retransmit**, retransmitting the missing segment *before* that segment's timer expires.

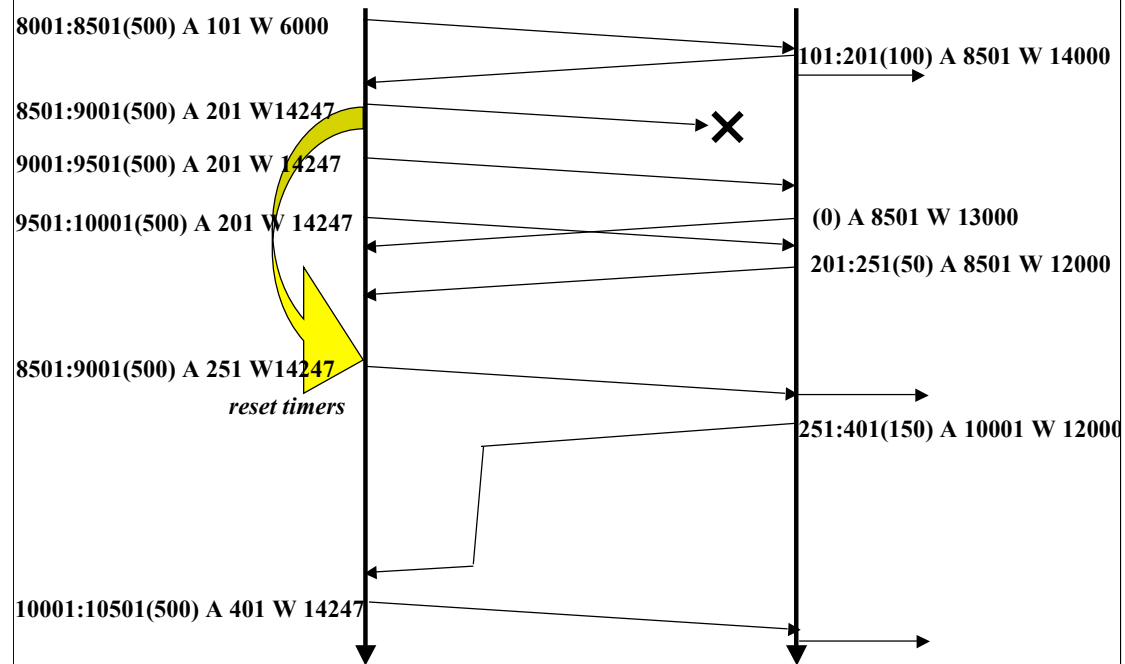
TCP: retransmission scenarios: GBN + SR



10

In the scenario the host A sends one segment to host B. Suppose that this segment has sequence number 92 and contains 8 bytes of data. After sending this segment, host A waits for a segment from B with acknowledgment number 100. Although the segment from A is received at B, the acknowledgment from B to A gets lost. In this case, the timer expires, and host A retransmits the same segment. Of course, when host B receives the retransmission, it will observe from the sequence number that the segment contains data that has already been received. Thus, TCP in host B will discard the bytes in the retransmitted segment. In the second scenario, host A sends two segments back to back. The first segment has sequence number 92 and 8 bytes of data, and the second segment has sequence number 100 and 20 bytes of data. Suppose that both segments arrive intact at B, and B sends two separate acknowledgments for each of these segments. The first of these acknowledgments has acknowledgment number 100; the second has acknowledgment number 120. Suppose now that neither of the acknowledgments arrive at host A before the timeout of the first segment. When the timer expires, host A resends the first segment with sequence number 92. Now, you may ask, does A also resend the second segment? According to the rules described above, host A resends the segment only if the timer expires before the arrival of an acknowledgment with an acknowledgment number of 120 or greater. Thus, if the second acknowledgment does not get lost and arrives before the timeout of the second segment, A does not resend the second segment. Note that, even if the acknowledgment with acknowledgment number 100 was lost, the cumulative acknowledgment coming with the acknowledgment number 120 would avoid the retransmission.

Example of data transfer - Reno

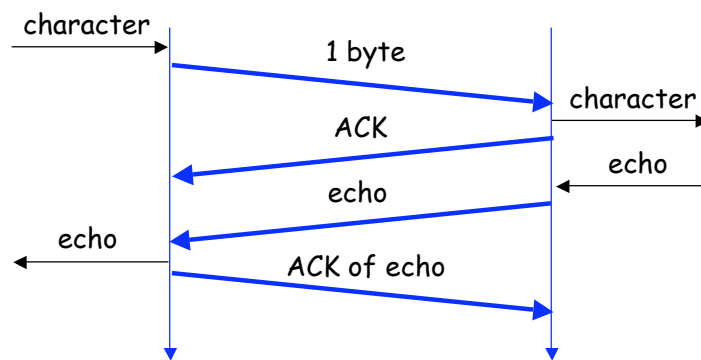


11

Retransmission starting from 8501

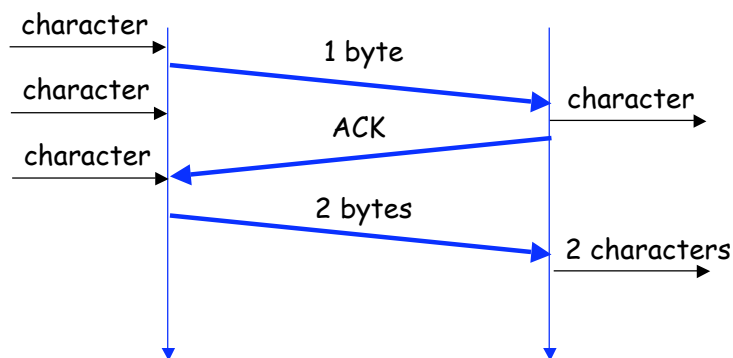
- Go-back-N, but sender retransmits only the first segments
- receiver accepts and stores segments out of order (9001 et 9501)
- on reception of 8501, data 8501:10001 passed to application
- after the reception of 10001, transmission continues

Interactive traffic



- Delayed ACK
 - ACK et echo in the same segment
 - 200 ms delay: ACK sent with echo character

Nagle algorithm



- Sender may only send one small no acknowledged segment - tinygram (small = smaller than MSS)
 - avoid sending small segments on the network - large overhead
 - Nagle algorithm can be disabled by application (TCP_NODELAY socket option):
 - X Window

Silly Window syndrome

- Small advertised window

```

                                ←   Ack 0 W 2000
0:1000                        →   buf = 2000, freebuf = 1000
1000:2000                    →   freebuf = 0
                                ←   Ack 2000 W 0
                                appl lit 1 octet : freebuf = 1
                                ←   Ack 2000 W 1
2000:2001                    →   freebuf = 0
                                appl lit 1 octet : freebuf = 1
                                ←   Ack 2001 W 1
2001:2002                    →   freebuf = 0

```

14

SWS occurs when a slow receiver cannot read data fast enough, and reads them in small increments. The window advertisement method of TCP has the effect of avoiding buffer overflow at the receiver (flow control), however, if no additional means are taken, it results in a large number of small packets to be sent, with no benefit to the receiver since anyhow it cannot read them fast enough. The (new) TCP specification mandates that sender and receiver should implement SWS avoidance.

SWS avoidance at the receiver simply forces the window to move by large increments. As data is read from the receive buffer, the upper window edge could be moved to the right. However, the SWS avoidance algorithm specifies that this should be done only if the upper window edge can be moved by at least the value of one full segment, or, if the buffer is small, by $F \times \text{receiveBuffer}$. As a result, there may be a fraction of the buffer ("reserve" on the picture) which is not advertised.

SWS avoidance at receiver is sufficient, however TCP senders must also implement SWS avoidance, in order to cope with receivers that do not comply with the latest standard. SWS avoidance at sender is in addition to Nagle's algorithm (whose objective is to avoid small packets, which otherwise may occur even if the offered window is large).

The picture "SWS Avoidance Example" also shows a window probe, which aims at avoiding deadlocks if some acks are lost.

Silly Window syndrome

- Sender has a lot of data to send
- Small advertised window forces to send small segments
- Solution at receiver
 - advertise window by large chunks: $\min(\text{MSS}, 1/2 \text{ RcvBuffer size})$
- Solution at sender
 - delay sending small segments: send at least $\min(\text{MSS}, 1/2 \text{ maximum RcvWindow})$

15

Sender side

Data come from the sending application 1 byte at a time

- o send first segment, no matter of which size
- o don't send until:
 - + ACK for the outstanding segment arrives
 - + there is a half rwnd of data to send
 - + full-sized segment (MSS) can be sent
- o Nagle's algorithm may be disabled by application
 (`setsockopt(sd, IPPROTO_TCP, TCP_NO_DELAY, ... , ...)`) for better interactive data exchange.

Receiver side (Clark's algorithm, RFC 0813)

Application on the receiving side reads data 1 byte at a time

- o don't advertise a larger window until the window can be increased by $\min(\text{MSS}, 1/2 \text{ the receiver's buffer space})$

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value - RTO?

- RTO: Retransmission Timeout
- longer than RTT
 - note: RTT will vary
- too short: premature timeout
 - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
 - ignore retransmissions, cumulatively ACKed segments
- **SampleRTT** will vary, want estimated RTT "smoother"
 - use several recent measurements, not just current **SampleRTT**

16

When a host sends a segment into a TCP connection, it starts a timer. If the timer expires before the host receives an acknowledgment for the data in the segment, the host retransmits the segment. The time from when the timer is started until when it expires is called the **timeout** of the timer. Clearly, the timeout should be larger than the connection's round-trip time, that is, the time from when a segment is sent until it is acknowledged. Otherwise, unnecessary retransmissions would be sent. But the timeout should not be much larger than the round-trip time; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, and it would thereby introduce significant data transfer delays into the application. The sample RTT, denoted **SampleRTT**, for a segment is the amount of time from when the segment is sent (that is, passed to IP) until an acknowledgment for the segment is received. Obviously, the **SampleRTT** values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems. Because of this fluctuation, any given **SampleRTT** value may be atypical. In order to estimate a typical RTT, it is therefore natural to take some sort of average of the **SampleRTT** values.

TCP Round Trip Time and Timeout

$$\text{EstimatedRTT} = (1-x) * \text{EstimatedRTT} + x * \text{SampleRTT}$$

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.125

Setting the timeout

- **EstimatedRTT** plus "safety margin"
- large variation in **EstimatedRTT** -> larger safety margin

$$\text{RTO} = \text{EstimatedRTT} + 4 * \text{Deviation}$$

$$\text{Deviation} = (1-x) * \text{Deviation} + x * |\text{SampleRTT} - \text{EstimatedRTT}|$$

17

TCP maintains an average, called **EstimatedRTT**, of the **SampleRTT** values. Upon receiving an acknowledgment and obtaining a new **SampleRTT**, TCP updates **EstimatedRTT** according to the following formula:

$$\text{EstimatedRTT} = (1-x) \cdot \text{EstimatedRTT} + x \cdot \text{SampleRTT}.$$

The above formula is written in the form of a programming language statement--the new value of **EstimatedRTT** is a weighted combination of the previous value of **EstimatedRTT** and the new value for **SampleRTT**. A typical value of x is x = 0.125 (i.e., 1/8). Note that **EstimatedRTT** is a weighted average of the **SampleRTT** values. This weighted average puts more weight on recent samples than on old samples. The timeout should be set so that a timer expires early (that is, before the delayed arrival of a segment's ACK) only on rare occasions. It is therefore natural to set the timeout equal to the **EstimatedRTT** plus some margin. The margin should be large when there is a lot of fluctuation in the **SampleRTT** values; it should be small when there is little fluctuation. TCP uses the following formula:

$$\text{RTO} = \text{EstimatedRTT} + 4 \cdot \text{Deviation},$$

where **Deviation** is an estimate of how much **SampleRTT** typically deviates from **EstimatedRTT**

$$\text{Deviation} = (1-x) \cdot \text{Deviation} + x \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

Dynamic RTO calculation (Jacobson)

Keeps track of RTT variance in addition to smoothed RTT itself.

version A (Comer, Tannenbaum)

$$\text{RTT} = a * \text{RTT} + (1 - a) * \text{measured_RTT} \quad (a = 7/8)$$

$$D = b * D + (1 - b) * |\text{RTT} - \text{measured_RTT}| \quad (b = 3/4)$$

$$\text{RTO} = \text{RTT} + 4 * D$$

Version B (Stevens, Jacobson)

initialize

$$\text{RTT} = 0$$

$$D = 3$$

$$\text{RTO} = \text{RTT} + 2 * D = 0 + 2 * 3 = 6 \quad (\text{for initial SYN})$$

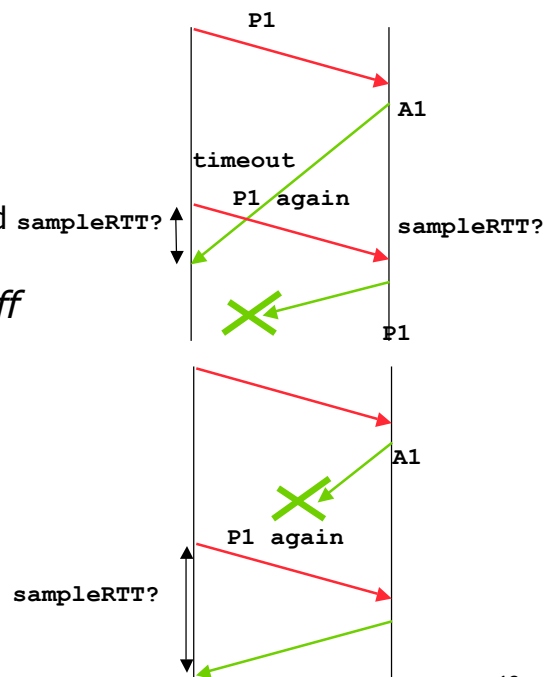
$$\text{RTT} = \text{RTT} + g * (\text{measured_RTT} - \text{RTT}) \quad (g = 1/8)$$

$$D = D + h * (|\text{measured_RTT} - \text{RTT}| - D) \quad (h = 1/4)$$

$$\text{RTO} = \text{RTT} + 4 * D$$

Karn and Partridge rules

- Do not measure RTT if retransmission
 - is it the ACK for the first transmission or the second one?
- *Timer exponential backoff*
 - double RTO at each retransmission



18

(Karn's Algorithm)

Solution to the retransmission ambiguity problem.

If (a segment has been retransmitted after a timeout)

$RTO = \min(2 \cdot RTO, 64 \text{ sec})$

RTT and D are not updated

Don't update RTO until

- another timeout

$RTO = \min(2 \cdot RTO, 64 \text{ sec})$ (exponential backoff)

- ACK for a segment that was not retransmitted

resume the Jacobson's algorithm

Beginning of the connection

- SYN segment timeout
- 1st
 - $D = 3, RTT = 0$
 - $RTO = RTT + 2 * D = 0 + 2 * 3 = 6 \text{ s}$
- 2nd
 - $RTO = RTT + 4 * D = 12 \text{ s}$
 - apply exp. backoff -> 24 s
- 3rd
 - apply exp. backoff -> 48 s
- 6, 24, 48, then drop
 - max. 75 s
- Implementation dependent

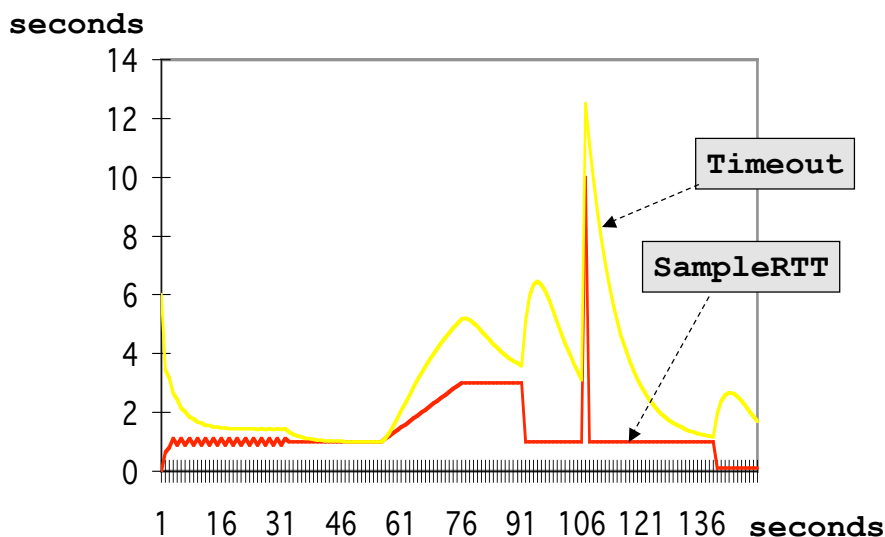
19

Data packets

- 1st
 - RTO = 1.5 s (3 ticks)
- 2nd
 - apply exp. backoff -> 3 s
- 7th
 - apply exp. backoff -> 64 s
- nth
 - max (64, 2xRTO)
- 13th
 - drop
- Total time
 - 542,5s = 9 minutes

20

A Simulation of RTO

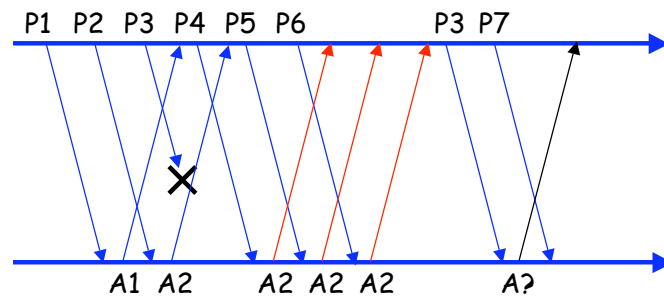


21

Round trip estimation is based on a low pass filter. Originally, the first TCP specification used a formula similar to estimatedRTT . However, it became apparent that RTT estimates fluctuate a lot, with fluctuations sometimes meaning a change of path. The formula is based on estimation of both average and deviation (which is an estimator of the absolute value of the error). The coefficients 0.125 and 0.25 (the estimation *gains*) are chosen to be simple negative powers of 2, which makes implementation of the multiplication simple (a bit shift). The specific values were tuned based on measured data.

In practice, most OSs do not check timeouts individually, but rather implement a timeout routine that wakes up periodically. On BSD Unix, such a routine wakes up every 0.5 seconds, which means that timers are computed with a granularity of 500 ms. This results in retransmission timeouts that may occur almost 500 ms after the due time. The same granularity applies to the RTT estimation.

Fast Retransmit

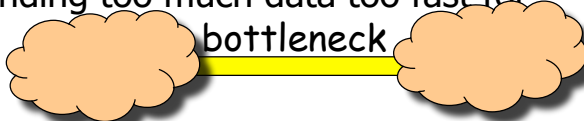


- Fast retransmit
 - timeout may be large
 - add the Selective Repeat behavior
 - if the sender receives 3 **deduplicated ACKs**, retransmit the missing segment

Congestion Control

Congestion:

- “too many sources sending too much data too fast for *network* to handle”
- manifestations:
 - lost packets (buffer overflow at routers)
 - long delays (queueing in router buffers)



Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
 - single bit indicating congestion
 - explicit rate sender should send at

23

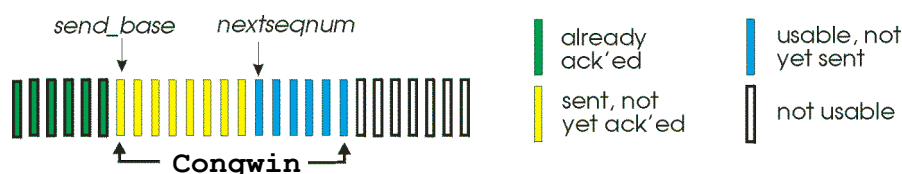
Packet retransmission treats a symptom of network congestion (the loss of a specific transport-layer segment, so long delay) but does not treat the cause of network congestion--too many sources attempting to send data at too high a rate. There exist two broad approaches that are taken in practice toward congestion control. We can distinguish among congestion-control approaches based on whether or not the network layer provides any explicit assistance to the transport layer for congestion-control purposes:

•*End-end congestion control.* In an end-end approach toward congestion control, the network layer provides *no explicit support* to the transport layer for congestion-control purposes. Even the presence of congestion in the network must be inferred by the end systems based only on observed network behaviour (for example, packet loss and delay).

•*Network-assisted congestion control.* With network-assisted congestion control, network-layer components (that is, routers) provide explicit feedback to the sender regarding the congestion state in the network. This feedback may be as simple as a single bit indicating congestion at a link. This approach was taken in the early IBM SNA and DEC DECnet architectures, was recently proposed for TCP/IP networks. Congestion information is typically fed back from the network to the sender in one of two ways, either as direct feedback from a network router to the sender (**choke packet**), or as marked packet flowing from sender to receiver. Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication. Note that this latter form of notification takes at least a full round-trip time.

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, **Congwin**, over segments:



- w segments, each with MSS bytes sent in one RTT:

$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

24

TCP must use end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion.

A TCP connection controls its transmission rate by limiting its number of transmitted-but-yet-to-be-acknowledged segments. Let us denote this number of permissible unacknowledged segments as w , often referred to as the TCP **window size**. Ideally, TCP connections should be allowed to transmit as fast as possible (that is, to have as large a number of outstanding unacknowledged segments as possible) as long as segments are not lost (dropped at routers) due to congestion. In very broad terms, a TCP connection starts with a small value of w and then "probes" for the existence of additional unused link bandwidth at the links on its end-to-end path by increasing w . A TCP connection continues to increase w until a segment loss occurs (as detected by a timeout or duplicate acknowledgments). When such a loss occurs, the TCP connection reduces w to a "safe level" and then begins probing again for unused bandwidth by slowly increasing w .

An important measure of the performance of a TCP connection is its throughput--the rate at which it transmits data from the sender to the receiver. Clearly, throughput will depend on the value of w . If a TCP sender transmits all w segments back to back, it must then wait for one round-trip time (RTT) until it receives acknowledgments for these segments, at which point it can send w additional segments. If a connection transmits w segments of size MSS bytes every RTT seconds, then the connection's throughput, or transmission rate, is $(w \cdot MSS)/RTT$ bytes per second.

Suppose now that K TCP connections are traversing a link of capacity R . Suppose also that there are no UDP packets flowing over this link, that each TCP connection is transferring a very large amount of data and that none of these TCP connections traverse any other congested link. Ideally, the window sizes in the TCP connections traversing this link should be such that each connection achieves a throughput of R/K . More generally, if a connection passes through N links, with link n having transmission rate R_n and supporting a total of K_n TCP connections, then ideally this connection should achieve a rate of R_n/K_n on the n th link. However, this connection's end-to-end average rate cannot exceed the minimum rate achieved at all of the links along the end-to-end path. That is, the end-to-end transmission rate for this connection is $r = \min\{R_1/K_1, \dots, R_N/K_N\}$. We could think of the goal of TCP as providing this connection with this end-to-end rate, r .

TCP congestion control:

- “**probing**” for usable bandwidth:
 - **ideally**: transmit as fast as possible (**Congwin** as large as possible) without loss
 - *increase Congwin* until loss (congestion)
 - loss: *decrease Congwin*, then begin probing (increasing) again
- two “phases”
 - **slow start**
 - **congestion avoidance**
- important variables:
 - **Congwin**
 - **threshold**: defines threshold between slow start phase and congestion avoidance phase

25

The TCP congestion-control mechanism has each side of the connection keep track of two additional variables: the **congestion window** and the **threshold**. The congestion window, denoted **CongWin**, imposes an additional constraint on how much traffic a host can send into a connection. Specifically, the amount of unacknowledged data that a host can have within a TCP connection may not exceed the minimum of **CongWin** and **RcvWin**. TCP congestion is composed of two phases:

- **Slow start**: When the congestion window is below the threshold, the congestion window grows exponentially.
- **Congestion avoidance**: When the congestion window is above the threshold, the congestion window grows linearly.
- Whenever there is a timeout, the threshold is set to one-half of the current congestion window and the congestion window is then set to 1.

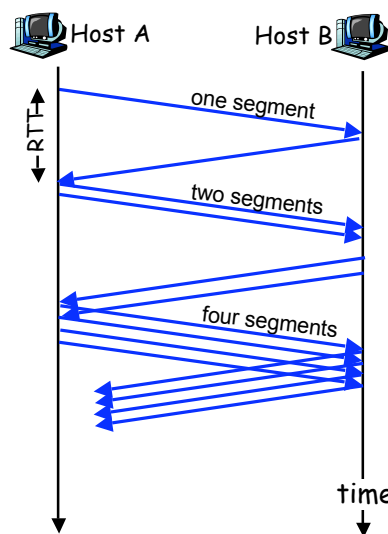
TCP Slowstart

Slowstart algorithm

```

initialize: Congwin = 1
for (each ACK)
    Congwin++
until (loss event OR
      CongWin > threshold)
  
```

- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)



26

Slow start: The congestion window is initialized to one MSS; after one RTT, the window is increased to two segments; after two round-trip times, the window is increased to four segments; after three round-trip times, the window is increased to eight segments, and so forth. (When the congestion window is below the threshold, the congestion window grows exponentially). This phase ends when either the congestion window becomes larger than the current value of `threshold` or when a segment is lost (Timeout for **TCP Tahoe**: the sender side of the application may have to wait a long period of time for the timeout. Fast retransmit mechanism in **TCP Reno**: it triggers the transmission of a dropped segment if three duplicate ACKs for a segment are received before the occurrence of the segment's timeout. Reno also employs a **fast-recovery** mechanism that essentially cancels the slow-start phase after a fast retransmission.)

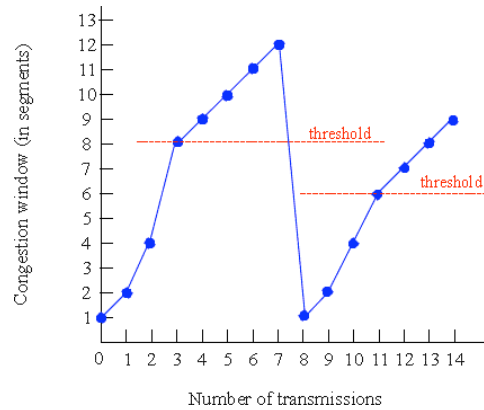
TCP Congestion Avoidance

Congestion avoidance

```

/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
    every w segments ACKed:
        Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart1

```



1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

27

Congestion avoidance: Once the congestion window is larger than the current value of `threshold`, the congestion window grows linearly rather than exponentially. Specifically, if w is the current value of the congestion window, and w is larger than `threshold`, then after w acknowledgments have arrived, TCP replaces w with $w + 1$. This has the effect of increasing the congestion window by 1 in each RTT for which an entire window's worth of acknowledgments arrives. The congestion-avoidance phase continues as long as the acknowledgments arrive before their corresponding timeouts. But the window size, and hence the rate at which the TCP sender can send, cannot increase forever. Eventually, the TCP rate will be such that one of the links along the path becomes saturated, at which point loss (and a resulting timeout at the sender) will occur.

When a timeout occurs, the value of `threshold` is set to half the value of the current congestion window, and the congestion window is reset to one MSS. The sender then again grows the congestion window exponentially fast using the slow-start procedure until the congestion window hits the threshold.

TCP Fairness

Fairness goal: if N TCP sessions share same bottleneck link, each should get $1/N$ of link capacity

WHY?

- Additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally

TCP congestion avoidance:

- **AIMD:** *additive increase, multiplicative decrease*
 - increase window by 1 per RTT
 - decrease window by factor of 2 on loss event

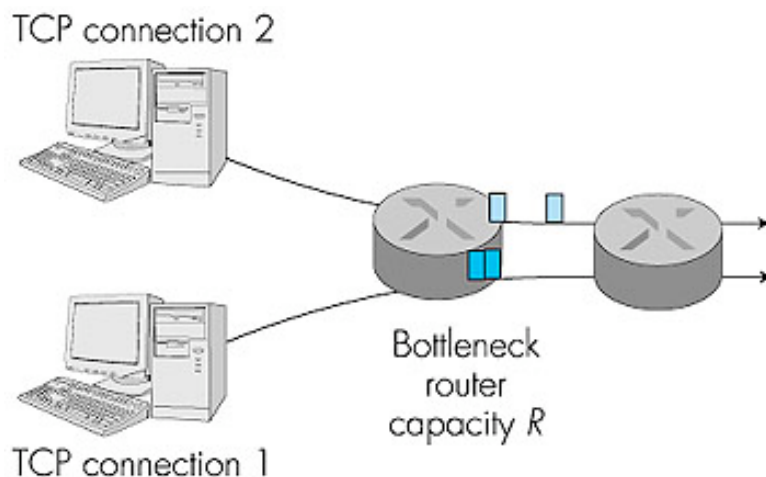
28

TCP congestion control converges to provide an equal share of a bottleneck link's bandwidth among competing TCP connections. If we ignore the slow-start phase, we see that TCP essentially increases its window size by 1 each RTT (and thus increases its transmission rate by an additive factor) when its network path is not congested, and decreases its window size by a factor of 2 each RTT when the path is congested. For this reason, TCP is often referred to as an **additive-increase, multiplicative-decrease (AIMD)** algorithm.

Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally



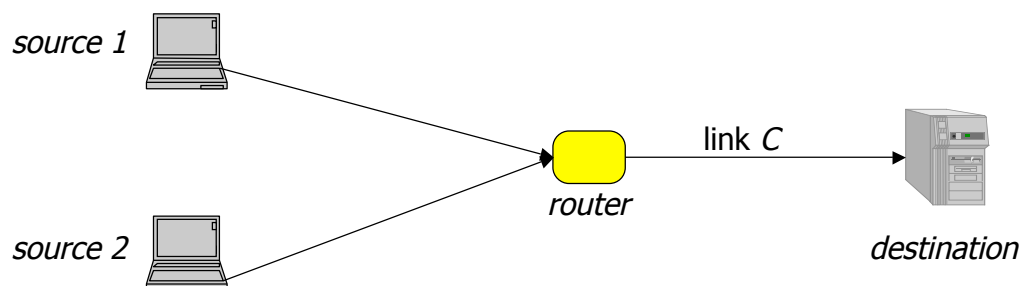
29

A goal of TCP's congestion-control mechanism is to share a bottleneck link's bandwidth evenly among the TCP connections that are bottlenecked at that link. But why should TCP's additive-increase, multiplicative-decrease algorithm achieve that goal, particularly given that different TCP connections may start at different times and thus may have different window sizes at a given point in time? [Chiu 1989] provides an elegant and intuitive explanation of why TCP congestion control converges to provide an equal share of a bottleneck link's bandwidth among competing TCP connections.

Let's consider the simple case of two TCP connections sharing a single link with transmission rate R , as shown in the figure. We'll assume that the two connections have the same MSS and RTT (so that if they have the same congestion window size, then they have the same throughput), that they have a large amount of data to send, and that no other TCP connections or UDP datagrams traverse this shared link. Also, we'll ignore the slow-start phase of TCP and assume the TCP connections are operating in congestion-avoidance mode (additive-increase, multiplicative-decrease) at all times.

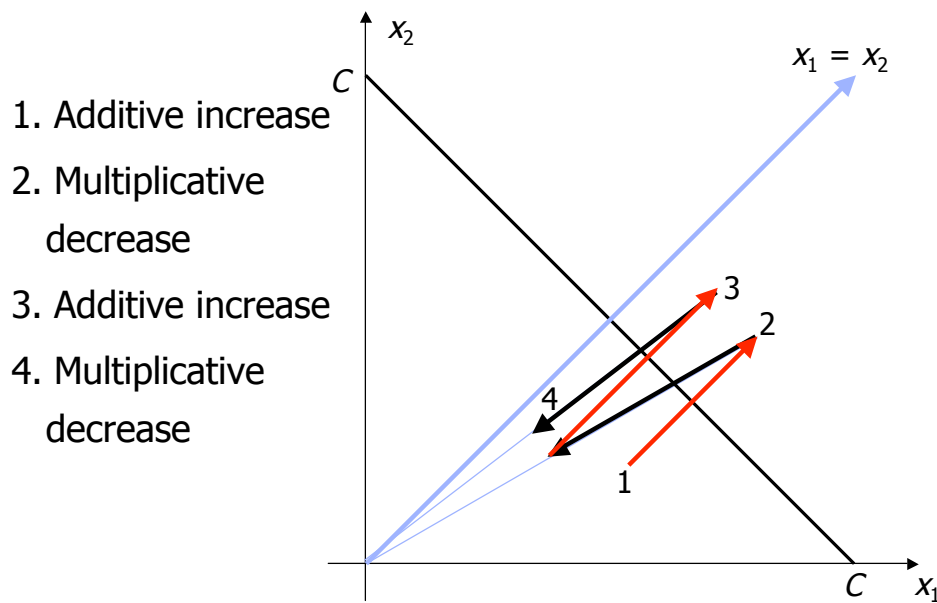
Suppose that the TCP window sizes are such that at a given point in time, that the amount of link bandwidth jointly consumed by the two connections is less than R . No loss will occur, and both connections will increase their window by 1 per RTT as a result of TCP's congestion-avoidance algorithm. Eventually, the link bandwidth jointly consumed by the two connections will be greater than R and eventually packet loss will occur. Connections 1 and 2 then decrease their windows by a factor of two. Suppose the joint bandwidth use is now less than R , the two connections again increase their throughputs. Eventually, loss will again occur, and the two connections again decrease their window sizes by a factor of two, and so on. You should convince yourself that the bandwidth realized by the two connections eventually fluctuates along sharing an equal bandwidth. You should also convince yourself that the two connections will converge to this behaviour regardless their starting point. Although a number of idealized assumptions lay behind this scenario, it still provides an intuitive feel for why TCP results in an equal sharing of bandwidth among connections.

Why AI-MD works?



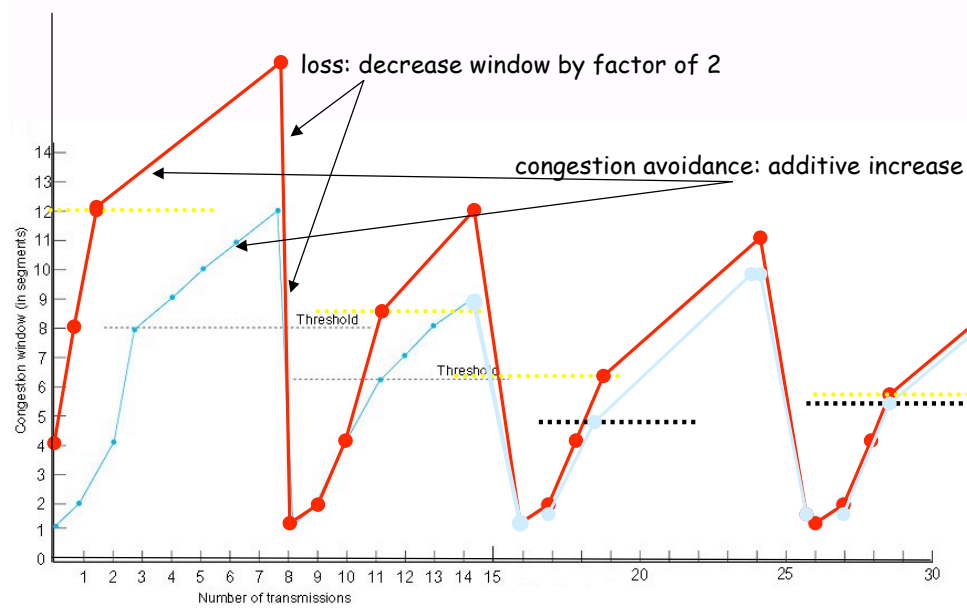
- Simple scenario with two sources sharing a bottleneck link of capacity C

Throughput of sources



Starting from any point, the throughput converges to the equal share vector

TCP Fairness

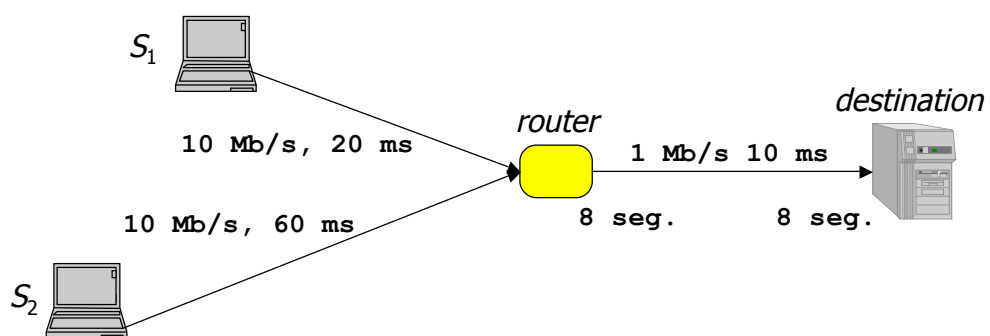


32

Fairness of the TCP

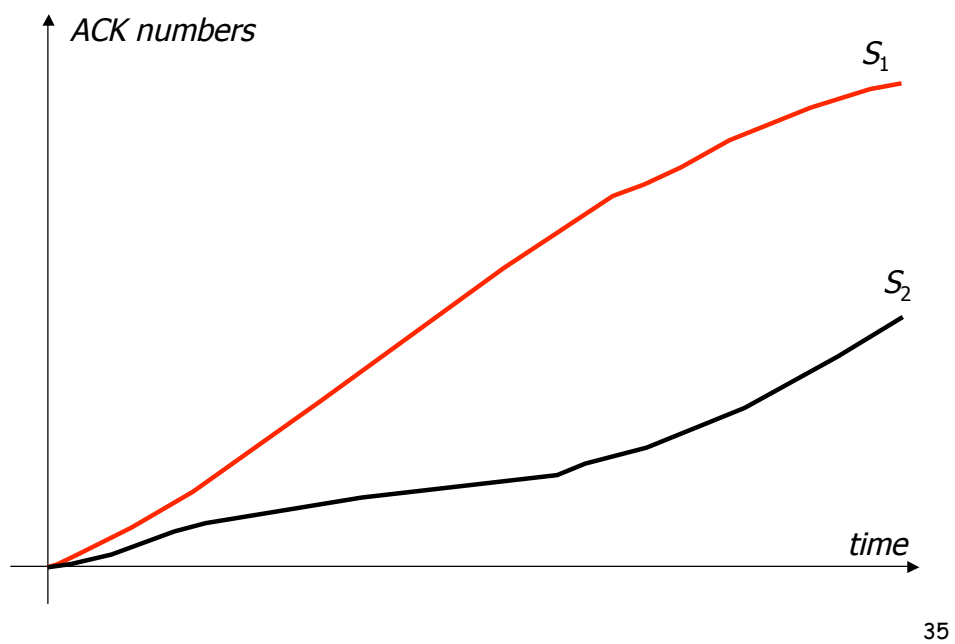
- TCP differs from the pure AI-MD principle
 - window based control, not rate based
 - increase in rate is not strictly additive - window is increased by $1/W$ for each ACK
- Adaptation algorithm of TCP results in a negative bias against long round trip times
 - adaptation algorithm gives less throughput to sources having larger RTT

Fairness of TCP



- Example network with two TCP sources
 - link capacity, delay
 - limited queues on the link (8 segments)
- NS simulation

Throughput in time



UDP: User Datagram Protocol [RFC 768]

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
 - lost
 - delivered out of order to app
- *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

36

UDP is a vacuous transport protocol, that does just about as little as a transport protocol can do. In particular, on the sending side, it does not provide much more than taking the messages from the application process and passing them directly to the network layer; and on the receiving side, more than taking the messages arriving from the network layer and passing them directly to the application process. More than this, UDP provides a multiplexing/demultiplexing service in order to pass data between the network layer and the correct process. However, aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP. In fact, if the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP. UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the port numbers and the IP destination address to deliver the segment's data to the correct application process. Note that with UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be *connectionless*.

•*No connection establishment.* As we'll discuss later, TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection.

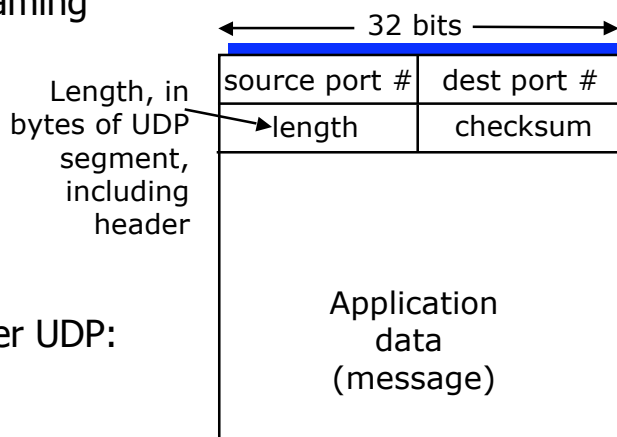
•*No connection state.* UDP does not maintain connection state (receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters) and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support several active clients when the application runs over UDP.

•*Small packet header overhead.* The UDP segment has only 8 bytes of header overhead in every segment.

•*Unregulated send rate.* The speed at which UDP sends data is only constrained by the rate at which the application generates data, the capabilities of the source (CPU, clock rate, and so on) and the access bandwidth to the Internet. We should keep in mind, however, that the receiving host does not necessarily receive all the data. When the network is congested, some of the data could be lost due to router buffer overflow. Thus, the receive rate can be limited by network congestion even if the sending rate is not constrained.

UDP: more

- often used for streaming multimedia apps
 - loss tolerant
 - rate sensitive
- other UDP uses
 - DNS
 - SNMP
- reliable transfer over UDP: add reliability at application layer
 - application-specific error recover!



UDP segment format

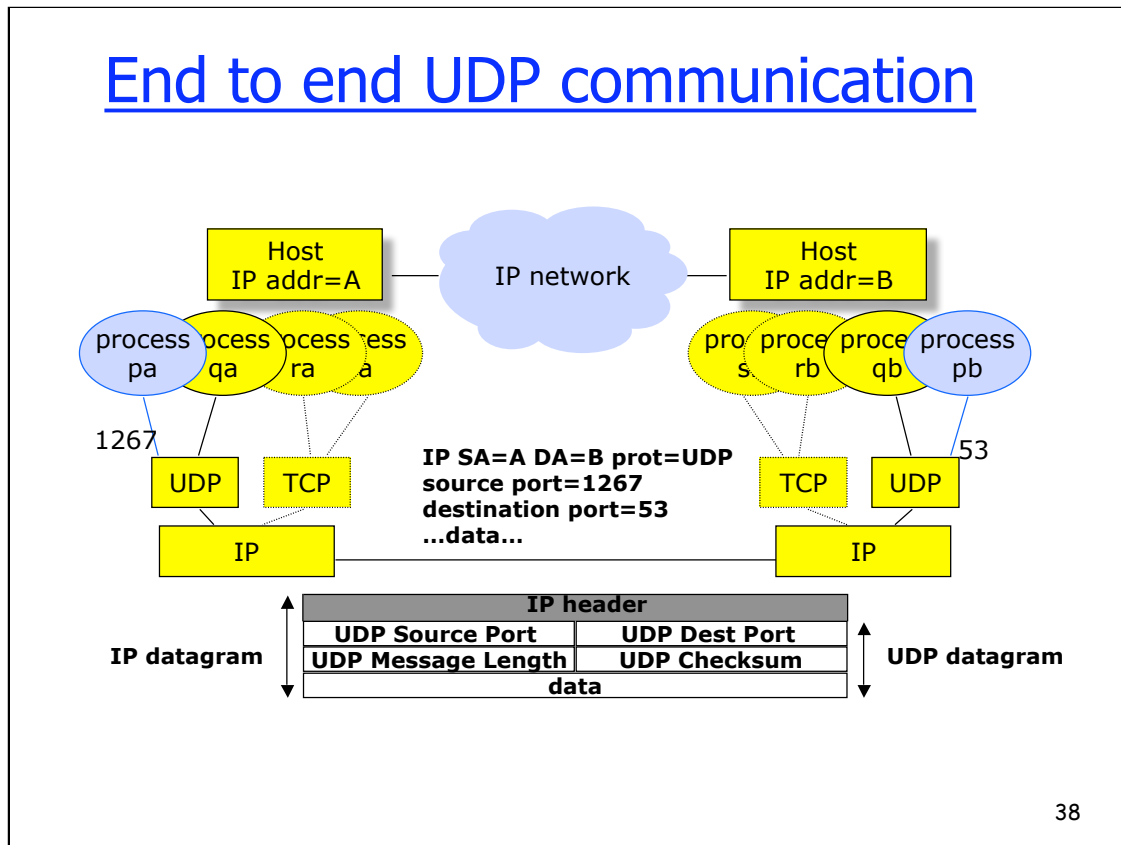
37

Many important applications run over UDP. UDP is used for RIP routing table updates, because the updates are sent periodically (typically every five minutes), so that lost updates are replaced by more recent updates. UDP is used to carry network management (SNMP) data. UDP is preferred to the more reliable TCP in this case, since network management applications must often run when the network is in a stressed state--precisely when reliable, congestion-controlled data transfer is difficult to achieve. Also DNS runs over UDP, thereby having no connection-establishment delays. UDP is also commonly used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video. All of these applications can tolerate a small fraction of packet loss, so that reliable data transfer is not absolutely critical for the success of the application. Furthermore, real-time applications, like Internet phone and video conferencing, react very poorly to (TCP's) congestion control. For these reasons, developers of multimedia applications often choose to run their applications over UDP instead of TCP. Finally, because TCP cannot be employed with multicast, multicast applications run over UDP. Although commonly done today, running multimedia applications over UDP is controversial to say the least. In fact, UDP has no congestion control. But congestion control is needed to prevent the network from entering a state in which very little useful work is done.

It is possible for an application to have reliable data transfer when using UDP. This can be done if reliability is built into the application itself (for example, by adding acknowledgment and retransmission mechanisms, such as those we shall study in the next section).

In the UDP segment structure the application data occupies the data field of the UDP datagram. The UDP header has only four fields, each consisting of two bytes. The port numbers allow the destination host to pass the application data to the correct process running on the destination (that is, the demultiplexing function). The checksum is used by the receiving host to check if errors have been introduced into the segment. The length field specifies the length of the UDP segment, including the header, in bytes.

End to end UDP communication



Let us have a closer look at UDP communication.

Two processes (= application programs) pa, and pb, are communicating. Each of them is associated locally with a port, as shown in the figure.

In addition, every machine (in reality: every communication adapter) has an IP address.

The example shows a packet sent by the name resolver process at host A, to the DNS name server process at host B. The UDP header contains the source and destination ports. The destination port number is used to contact the name server process at B; the source port is not used directly; it will be used in the response from B to A.

The UDP header also contains a checksum to protect the UDP data plus the IP addresses and packet length. Checksum computation is not performed by all systems.

Sockets

- Interface between applications and the transport layer protocols
 - socket - communication end-point
 - network communication viewed as a file descriptor (socket descriptor)
- Two main types of sockets
 - connectionless mode (or datagram, UDP protocol)
 - connection mode (or stream, TCP protocol)

39

The socket library is an application programming interface (API) that provides access to the transport layer and with some restrictions (raw sockets) to the network layer.

A socket means a communication end-point. Unix considers a socket as a file. A socket corresponds to a receive and a send buffer in the operating system kernel.

A TCP/IP socket is identified by the IP address of a communication interface and a port number. There are three types of sockets:

- UDP sockets: type = datagram; protocol = UDP
- TCP sockets: type = stream; protocol = TCP
- raw sockets: type = raw; protocol = IP or ICMP

Connection mode

- System calls in connection mode (TCP)
 - **socket** - create a socket descriptor
 - **bind** - associate with a local address
 - **listen** - signal willingness to wait for incoming connections (S)
 - **accept** - accept a new incoming connection (S)
 - **connect** - ask to establish a new connection (C)
 - **send** - send a buffer of data
 - **recv** - receive data
 - **close** - close socket descriptor

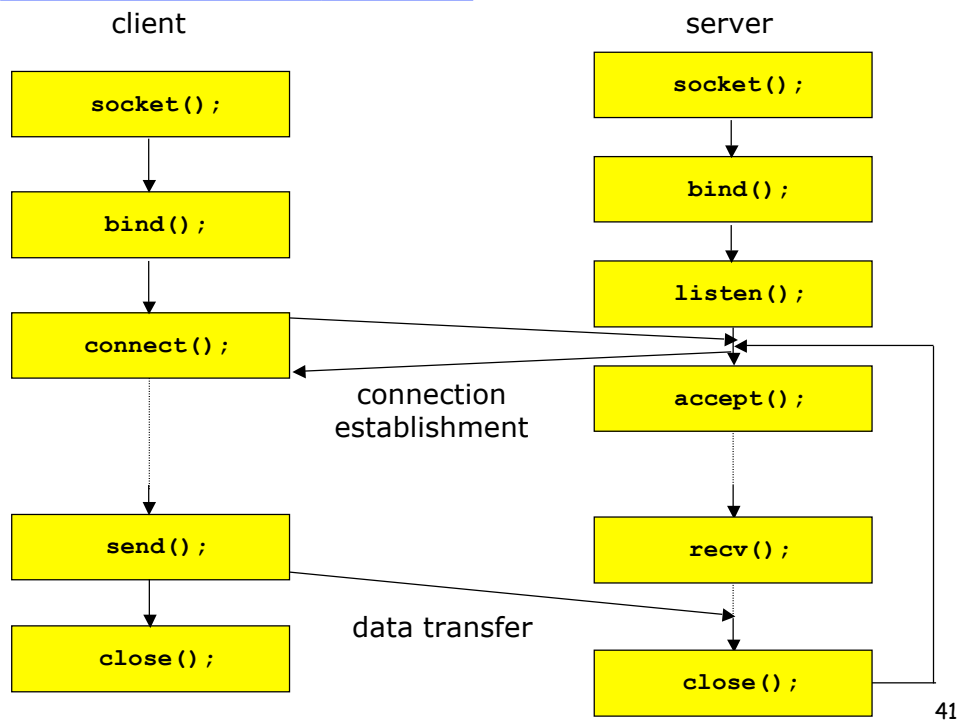
40

TCP sockets differ from UDP sockets in a number of ways. Since TCP is connection oriented, a TCP socket can be used only after a connection establishment phase. This uses the **connect** (at client), and **listen**, and **accept** (at server) calls.

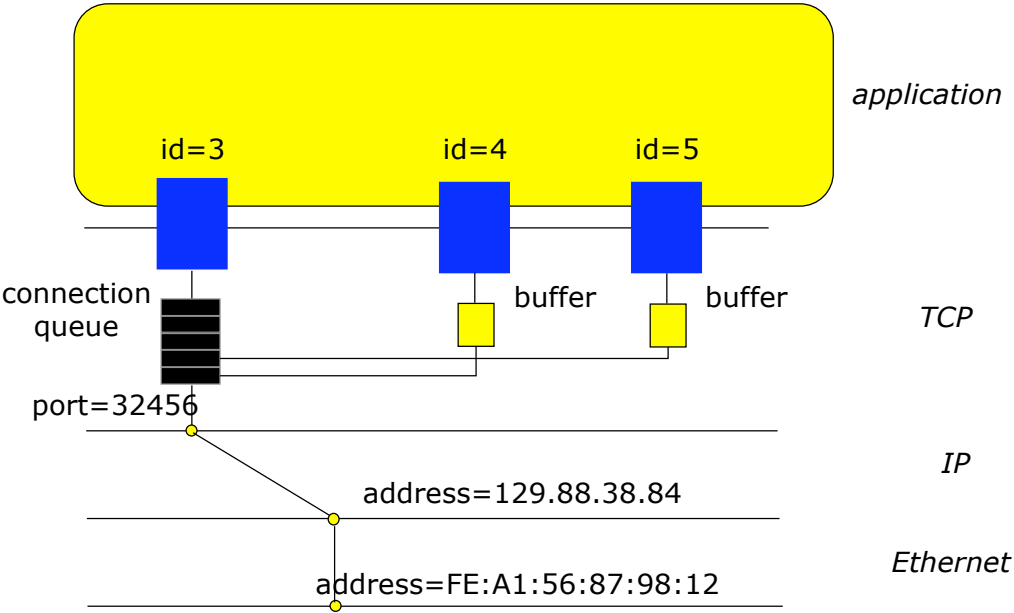
On the client side, a connection establishment is requested with **connect** call.

A TCP server uses at least two sockets. One socket is non-connected and is used to receive connection requests ("SYN" segments). Once a connection request is accepted, a new socket is created; this new socket is connected to the remote end. **listen** is used to tell the operating system to wait for incoming connection requests on a socket. **accept** consumes one connection request (if any has arrived; otherwise it blocks). It creates a new socket and returns the socket descriptor.

Connection mode



Connection mode



Connectionless mode

- System calls in connectionless mode (UDP)
 - **socket** - create a socket descriptor
 - **bind** - associate with a local address
 - **sendto** - send a buffer of data
 - **recvfrom** - receive data
 - **close** - close socket descriptor

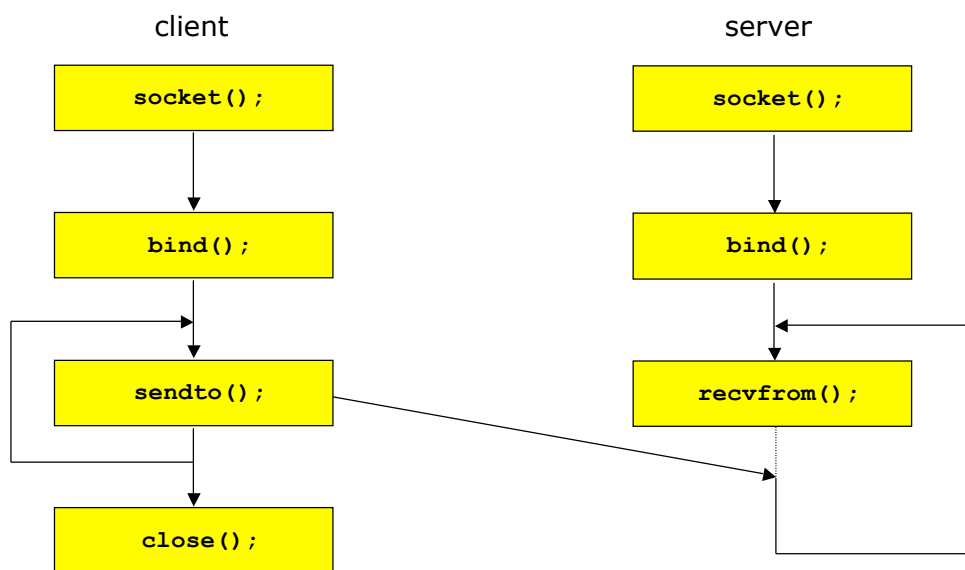
43

A program creates a socket with the **socket** system call. The call returns a (small) integer (the "socket descriptor") which can be used later to reference the socket. The socket descriptor is in the same space as a file descriptor. A Unix process has at least three open files: 0 (input); 1 (output); 2 (error). A process that accesses no disk file and opens one socket is likely to receive a socket descriptor 3. File access functions such as **read** and **write** can be performed on sockets too.

When a socket is created with the **socket** call, it is not yet usable. It must be associated with a network interface and a port number. This is called "binding" and is performed with the **bind** system call.

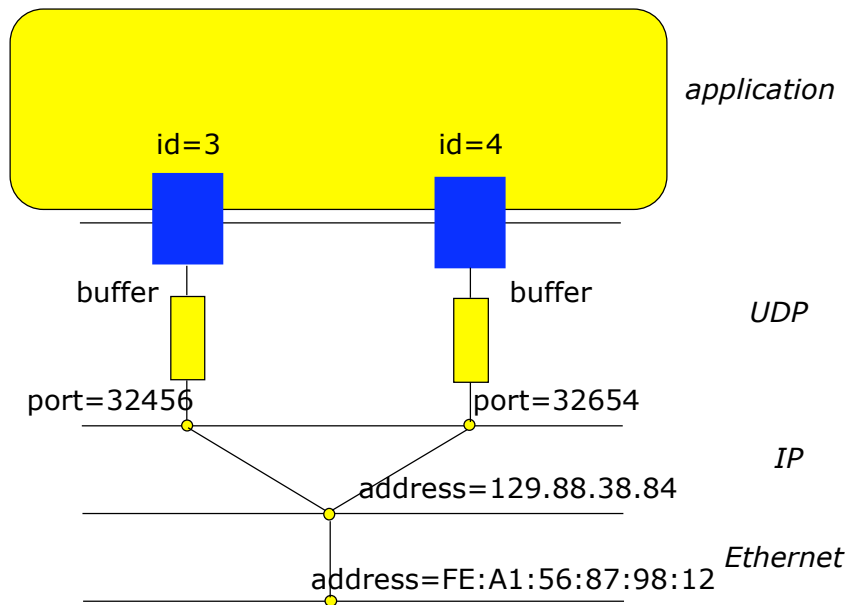
The function pair **sendto()**, **recvfrom()** are used to send data to any destination, or receive from any source, over one single socket. The remote system is part of the system call arguments.

Connectionless mode



44

Connectionless mode



45

Summary

- TCP protocol is complex!
 - connection management
 - reliable transfer
 - interactive traffic, Nagle algorithm
 - silly window syndrome
 - RTT estimation and Karn's rule
 - fast retransmit
 - congestion control
- UDP is simple
 - adds multiplexing to IP datagrams
 - used by RTP/RTCP for multimedia streaming
- Sockets - application interface to network communication
 - connection sockets (TCP)
 - connectionless sockets (UDP)

46

Question

- We have observed the following traces
 - relay1 -> in1sun1 TCP D=38662 S=9 Ack=399593749
Seq=2756727981 Len=0 Win=24820
 - in1sun1 -> relay1 TCP D=9 S=38662 Ack=2756727981
Seq=399613205 Len=1024 Win=24820
 - in1sun1 -> relay1 TCP D=9 S=38662 Ack=2756727981
Seq=399614229 Len=1024 Win=24820
 - in1sun1 -> relay1 TCP D=9 S=38662 Ack=2756727981
Seq=399615253 Len=1024 Win=24820
 - in1sun1 -> relay1 TCP D=9 S=38662 Ack=2756727981
Seq=399616277 Len=1024 Win=24820
 - relay1 -> in1sun1 TCP D=38662 S=9 Ack=399595797
Seq=2756727981 Len=0 Win=24820
- We have measured the RTT with **relay1** of 158 ms
- What is the throughput shown by netperf?

47

Réponse

- Voici la solution :
- a la reception les donnees sont retirees le plus vite possible (port discard), donc l'emetteur n'est limite que par la fenetre annonce - 24820 octets. Il peut envoyer alors 24820 octets par RTT. Ca donne le debit de
- $24820 \times 8 / 158\text{ms} = 1,25 \text{ Mb/s}$ qui est proche de 1,139 Mb/s mesure par netperf dans cette experience.

48