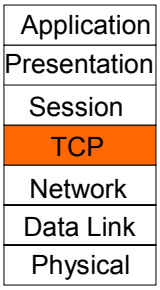
# CSE/EE 461 – Module 12

# TCP End-to-End

## This Time

- End-to-end considerations for TCP
  - How is *connect()* different from *send(SYN)?*
  - What does receiver do?
  - What does sender do?
    - When should data be sent?
    - When should it be resent?
    - When should it conclude connectivity has been lost?



## *connect() vs. send(SYN)*

- Q: Is *connect()* the same thing as *send(syn)* (if the interface allowed the latter)?


A: No. (How are they different?)



## Concurrency and blocking

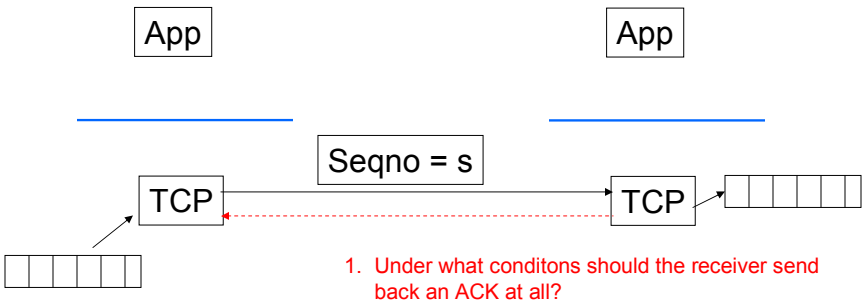
- Protocol implementation involves a lot of concurrency
  - E.g., (S1) sending app thread adds to send buffer; (S2) sending TCP thread removes from buffer and sends; (R1) receving TCP thread puts in buffer; (R2) receiving app reads from buffer
- Whether or not the app thread is blocked is an important part of the semantics
  - Why should app thread block on *connect()*?
  - Why shouldn't it block on *send()?*
    - Why should it block on *send()?*
  - Must *receive()* be blocking?
  - Must *close()* be blocking?

## Socket Semantics vs. Application Architecture

- The application knows best what semantics it needs
  - Suppose your application establishes a data connection and a control connection to some peer
  - Can't do a *read()* from either one without ignoring the other
- One way to get around blocking semantics at lower level: spawn more threads, and synchronize as necessary at the user level
- Problem: performance
- "Solution": Most interfaces provide some form of non-blocking mechanism
  - Usually you can:
    - Ask if some operation would block or not (*poll*)
    - Wait for any of a number of distinct events to happen (*select*)
- A half-way measure: often you can specify a timeout for how long the thread should block (e.g., *receive(250)*)



## What does the receiver do?



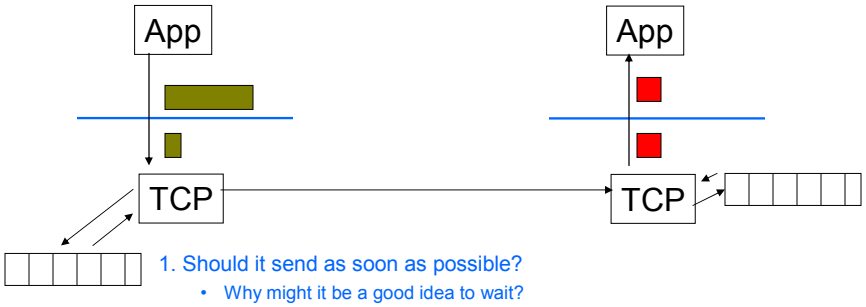

1. Under what conditons should the receiver send back an ACK at all?
2. When it does, what should the ACK seqno be?

## What should the receiver do?

- General philosophy:
  - keep receiver as simple as possible
  - ACKs are the primary feedback the sender has to work with
- With that in mind:
  - Don't ACK ACK's. (What happens if you do?)
  - Do ACK everything else.
    - Receiver **must** ACK already seen data…
- Many possible choices for what ACK should send back
  - TCP: seqno of first byte not yet received
- Can TCP send a segment with no data bytes?
  - What should happen?



## What does the sender do?




1. Should it send as soon as possible?
   - Why might it be a good idea to wait?
2. When it sends, how long should the retry timeout be?
   - Problem with too short? too long?
3. When should it give up?

## 1. Send as soon as possible?

- "Silly window" problem
  - Reminder: `Effective Window = Receiver advertised window - (LastByteSent - LastByteAcked)`
- Suppose the sender transmits a small frame for some reason.
  - The ACK for that frame opens the effective window by its size
  - The sender sends an equally small segment
  - Etc…
- Want to avoid this!
  - Either don't send small segments, or
  - Don't open window by a small amount



## 1. Send as soon as possible?

- Possible receiver side approaches:
  - Could use a timeout at receiver
    - Send an ACK at most once per timeout?
  - Simpler: if window goes to zero, don't advertise an open window until you have an MSS (maximum segment size) available
- Possible sender side approaches:
  - Could use a sender timeout
  - Could use a Nagle's Algorithm (self-clocking)

## Nagle's Algorithm

```
send() {
    if both available data and eff window ≥ MSS {
       send MSS bytes
    } else if lastByteSent – lastByteAcked > 0 {
       // don't send
    } else {
       send min(available data, eff window) now
    }
}
```



## 2. Deciding When to Retransmit

- How do you know when a packet has been lost?

```
do {
   send(p);
   wait(t);
} while (!p.acked)
```

- How long should the timer t be?
  - Too big: inefficient (large delays ⇒ poor use of bandwidth)
  - Too small: may retransmit unnecessarily (causing extra traffic)
  - A good retransmission timer is important for good performance
- Right timer is based on the round trip time (RTT)
  - Which varies greatly in the wide area (path length and queuing)

## 2. Setting the Retransmission Timeout

- Boils down to estimating RTT
- The straightforward approach:
  - for each packet, note time sent and time ACK received (RTT sample)
  - compute RTT samples and average recent samples for timeout

```
EstimatedRTT = (1-g)(EstimatedRTT) + g(SampleRTT)
0   g    1
```

  - this is an **exponentially-weighted moving average** (low pass filter) that smoothes the samples with a gain of *g*
    - big *g* can be jerky, but adapts quickly to change
    - small g can be smooth, but slow to respond
    - typically, g = .1 or .2 ⇒ stability is more important than precision
      - (lousy estimate right now does more damage than so-so estimate right now, followed by better one a little later)
  - (Why not `EstimatedRTT = (Sum of SampleRTT's) / N?`)



## Original TCP (RFC793) retransmission timeout algorithm

- Use EWMA to estimate RTT:

```
EstimatedRTT = (1-g)(EstimatedRTT) + g(SampleRTT)
0   g    1,  usually g = .1 or .2
```

- Conservatively set timeout to small multiple (2x) of the estimate

```
Retransmission Timeout = EstimatedRTT + EstimatedRTT
```

- Why the '`+ EstimatedRTT`'?
  - Better to wait "too long" than not long enough.

# Jacobson/Karels Algorithm

- Replace "`+ EstimatedRTT`" with measured variation in RTT

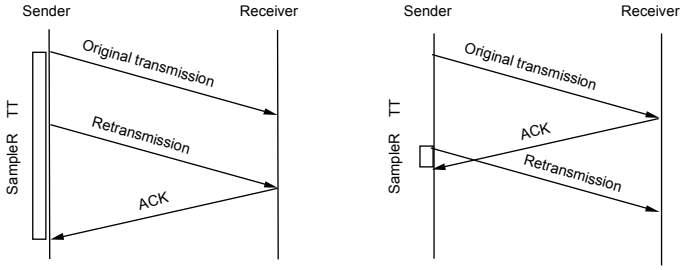
1. Compute a sample deviation statistic
   - `DevRTT = (1-b) * DevRTT + b * |SampledRTT - EstimatedRTT|`
     - typically, b = .25
2. Set timeout interval as:
   - `retransmission timeout = EstimatedRTT + k * DevRTT`
     - k is generally set to 4

- timeout =~ EstimatedRTT when variance is low (estimate is good)
  - timeout quickly moves away from EstimatedRTT (4x!) when the variance is high (estimate is bad)



# Karn/Partridge Algorithm

- Problem: RTT for retransmitted packets ambiguous




- Solution: Don't measure RTT for retransmitted packets
  - Problem: RTT not updated when timeouts occurring
  - Approach: use backoff on timeout until an xmit succeeds with retrransmission

# 3. When do we give up?

RFC 1122 (Requirements for Internet Hosts)

The following procedure MUST be used to handle excessive retransmissions of data segments:

- There are two thresholds R1 and R2 measuring the amount of retransmission that has occurred for the same segment.
- When the number of transmissions of the same segment reaches or exceeds threshold R1, pass negative advice to the IP layer, to trigger dead-gateway diagnosis.
- When the number of transmissions of the same segment reaches a threshold R2 greater than R1, close the connection.
- An application MUST be able to set the value for R2 for a particular connection. TCP SHOULD inform the application of the delivery problem (unless such information has been disabled by the application; see Section 4.2.4.1), when R1 is reached and before R2.
- The value of R1 SHOULD correspond to at least 3 retransmissions, at the current RTO. The value of R2 SHOULD correspond to at least 100 seconds.