

Chapter 3

Transport Layer

A note on the use of these ppt slides:

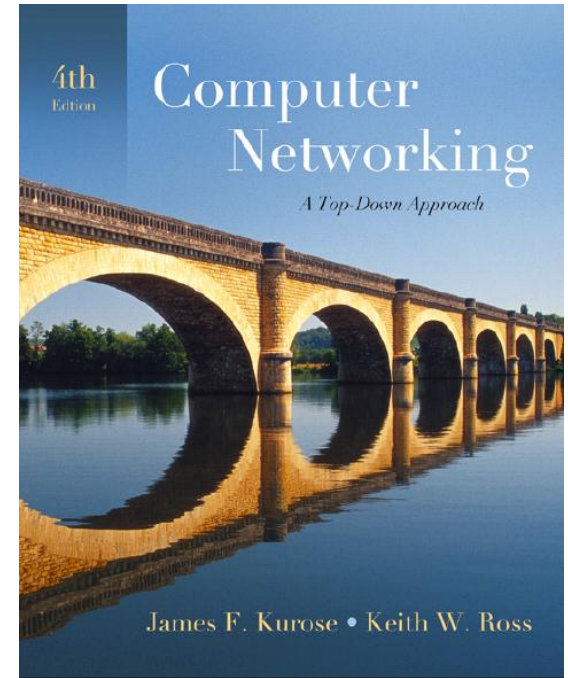
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2007

J.F Kurose and K.W. Ross, All Rights Reserved



*Computer Networking:
A Top Down Approach
4th edition.*

*Jim Kurose, Keith Ross
Addison-Wesley, July
2007.*

Chapter 3 outline

- ❑ 3.1 Transport-layer services
- ❑ 3.2 Multiplexing and demultiplexing
- ❑ 3.3 Connectionless transport: UDP
- ❑ 3.4 Principles of reliable data transfer
- ❑ 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- ❑ 3.6 Principles of congestion control
- ❑ 3.7 TCP congestion control

TCP reliable data transfer

- ❑ TCP creates rdt service on top of IP's unreliable service
- ❑ Pipelined segments
- ❑ Cumulative acks
- ❑ TCP uses single retransmission timer
- ❑ Retransmissions are triggered by:
 - timeout events
 - duplicate acks
- ❑ Initially consider simplified TCP sender:
 - ignore duplicate acks
 - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- ❑ Create segment with seq #
- ❑ seq # is byte-stream number of first data byte in segment
- ❑ start timer if not already running (think of timer as for oldest unacked segment)
- ❑ expiration interval: `TimeoutInterval`

timeout:

- ❑ retransmit segment that caused timeout
- ❑ restart timer

Ack rcvd:

- ❑ If acknowledges previously unacked segments
 - update what is known to be acked
 - start timer if there are outstanding segments

```
NextSeqNum = InitialSeqNum
```

```
SendBase = InitialSeqNum
```

```
loop (forever) {  
  switch(event)
```

```
  event: data received from application above  
    create TCP segment with sequence number NextSeqNum  
    if (timer currently not running)  
      start timer  
    pass segment to IP  
    NextSeqNum = NextSeqNum + length(data)
```

```
  event: timer timeout  
    retransmit not-yet-acknowledged segment with  
      smallest sequence number  
    start timer
```

```
  event: ACK received, with ACK field value of y  
    if (y > SendBase) {  
      SendBase = y  
      if (there are currently not-yet-acknowledged segments)  
        start timer  
    }  
}
```

```
} /* end of loop forever */
```

TCP sender (simplified)

Comment:

- $SendBase-1$: last cumulatively ack'ed byte

Example:

- $SendBase-1 = 71$;
 $y = 73$, so the rcvr wants 73+ ;
 $y > SendBase$, so that new data is acked

Fast Retransmit

- ❑ Time-out period often relatively long:
 - long delay before resending lost packet
- ❑ Detect lost segments via duplicate ACKs.
 - Sender often sends many segments back-to-back
 - If segment is lost, there will likely be many duplicate ACKs.
- ❑ If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
 - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
    if (y > SendBase) {
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
            start timer
    }
    else {
        increment count of dup ACKs received for y
        if (count of dup ACKs received for y = 3) {
            resend segment with sequence number y
        }
    }
```

a duplicate ACK for
already ACKed segment

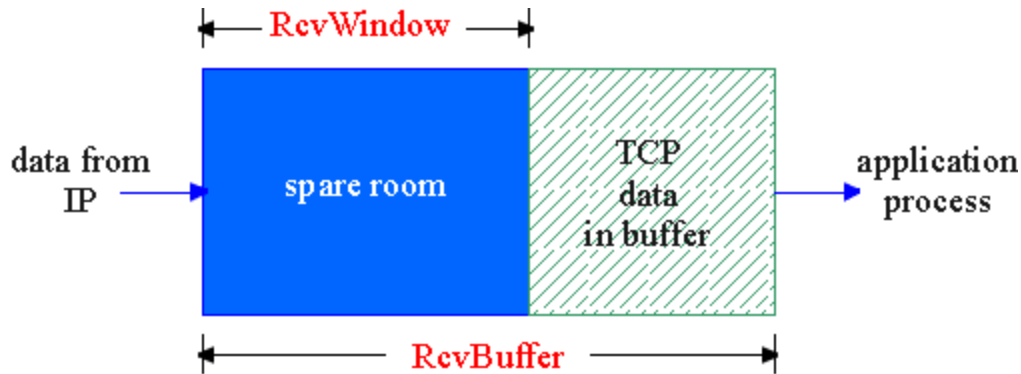
fast retransmit

Chapter 3 outline

- ❑ 3.1 Transport-layer services
- ❑ 3.2 Multiplexing and demultiplexing
- ❑ 3.3 Connectionless transport: UDP
- ❑ 3.4 Principles of reliable data transfer
- ❑ 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- ❑ 3.6 Principles of congestion control
- ❑ 3.7 TCP congestion control

TCP Flow Control

- receive side of TCP connection has a receive buffer:



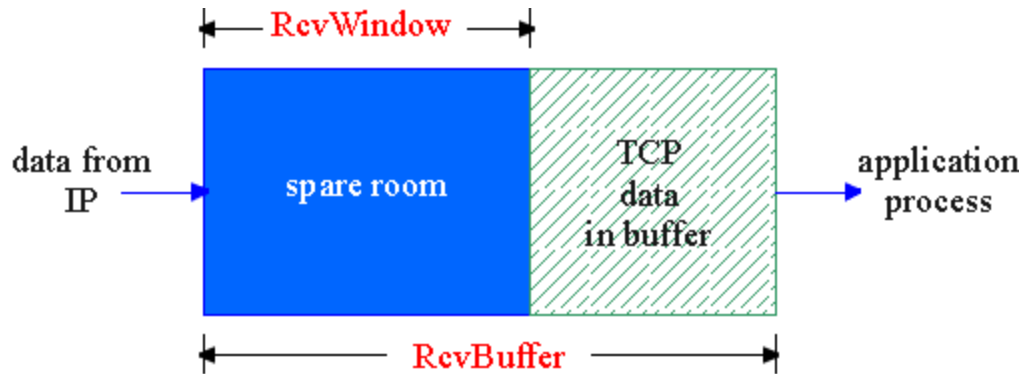
- app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
- = RcvWindow
- = $\text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
 - guarantees receive buffer doesn't overflow

Chapter 3 outline

- ❑ 3.1 Transport-layer services
- ❑ 3.2 Multiplexing and demultiplexing
- ❑ 3.3 Connectionless transport: UDP
- ❑ 3.4 Principles of reliable data transfer
- ❑ 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- ❑ 3.6 Principles of congestion control
- ❑ 3.7 TCP congestion control

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

□ initialize TCP variables:

- seq. #s
- buffers, flow control info (e.g. RcvWindow)

□ *client*: connection initiator

```
Socket clientSocket = new  
Socket("hostname", "port  
number");
```

□ *server*: contacted by client

```
Socket connectionSocket =  
welcomeSocket.accept();
```

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (cont.)

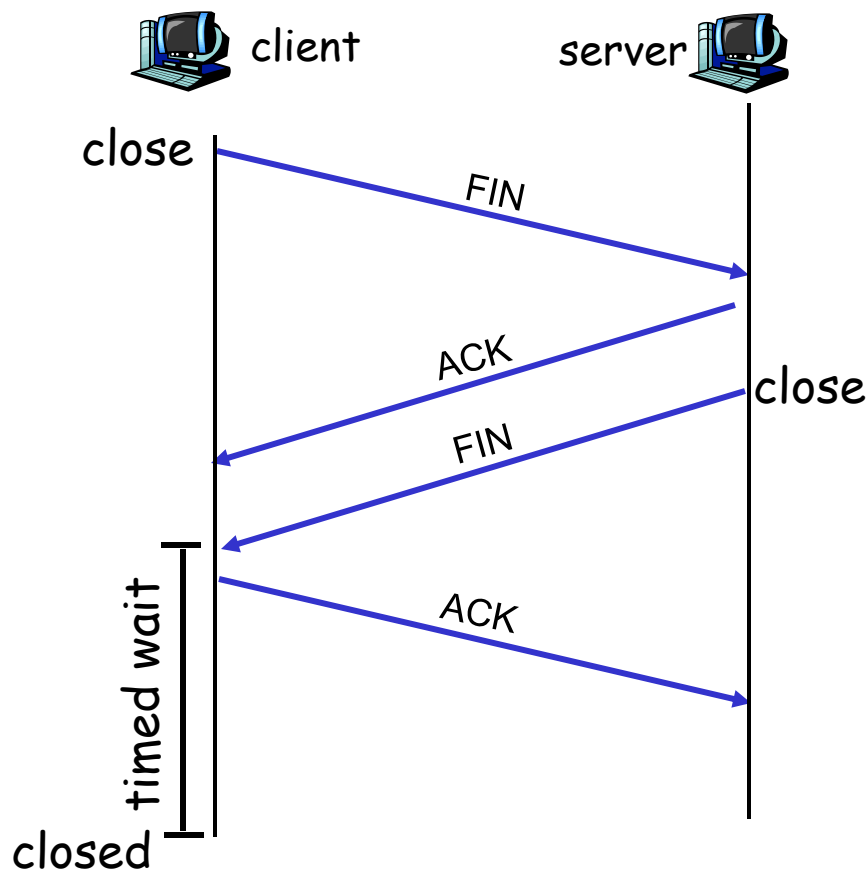
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



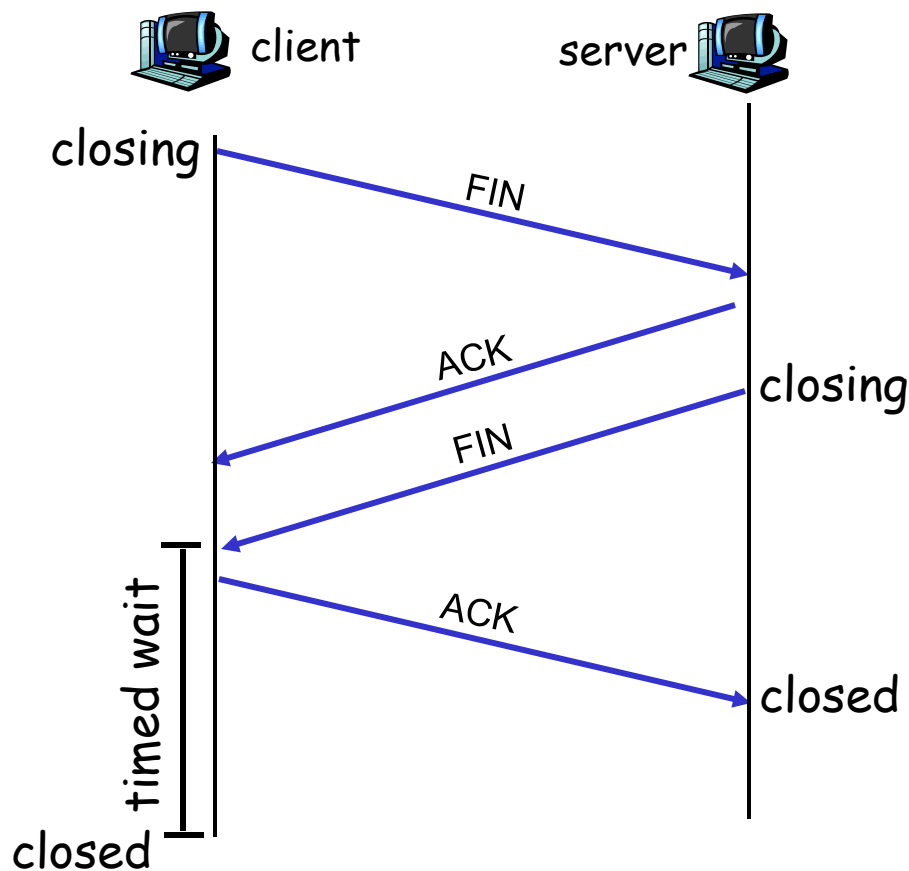
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

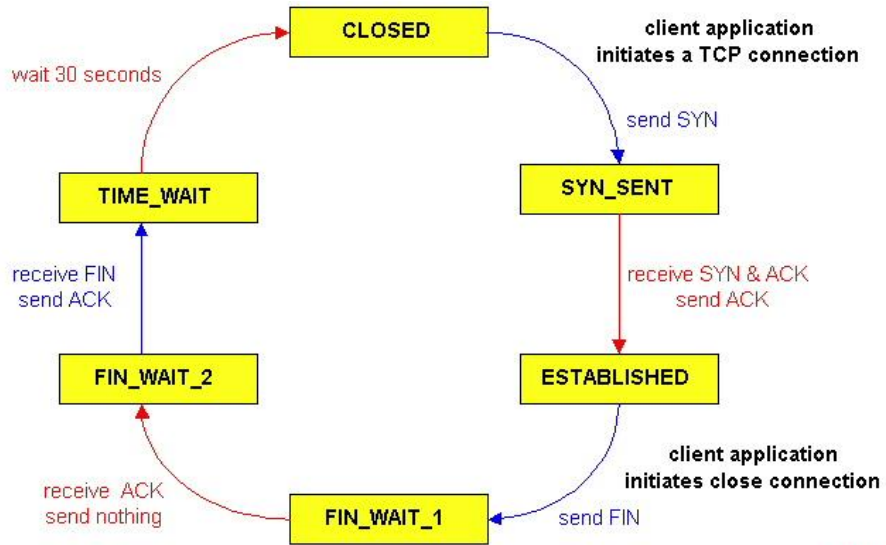
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP Connection Management (cont)



TCP client lifecycle

