Data Communication & Networks
G22.2262-001

Session 9 - Main Theme
The Internet Transport Protocols: TCP, UDP

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

- Internet Transport Protocols
- Transport Layer Addressing
- Standard Services and Port Numbers
- TCP Overview
- Reliability in an Unreliable World
- TCP Flow Control
- Why Startup / Shutdown Difficult?
- TCP Connection Management
- Timing Problem
- Implementation Policy Options
- UDP: User Datagram Protocol
- Conclusion

Part I

Internet Transport Protocols

Internet Transport Protocols

- Two Transport Protocols Available
  - Transmission Control Protocol (**TCP**)
    - *connection oriented*
    - most applications use TCP
    - RFC 793
  - User Datagram Protocol (**UDP**)
    - *Connectionless*
    - RFC 768

Part II

Transport Layer Addressing

Transport Layer Addressing

- Communications endpoint addressed by:
  - IP address (32 bit) **in IP Header**
  - Port number (16 bit) **in TP Header**[1]
  - Transport protocol (TCP or UDP) **in IP Header**

[1] TP => Transport Protocol (UDP or TCP)

Part III

Standard Services and Port Numbers

Standards Services and Port Numbers

service	tcp	udp
echo	7	7
daytime	13	13
netstat	15	
ftp-data	20	
ftp	21	
telnet	23	
smtp	25	
time	37	37
domain	53	53
finger	79	
http	80	
pop-2	109	
pop	110	
sunrpc	111	111
uucp-path	117	
nntp	119	
talk		517

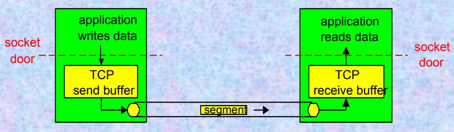
Part IV

TCP: Overview

TCP: Overview

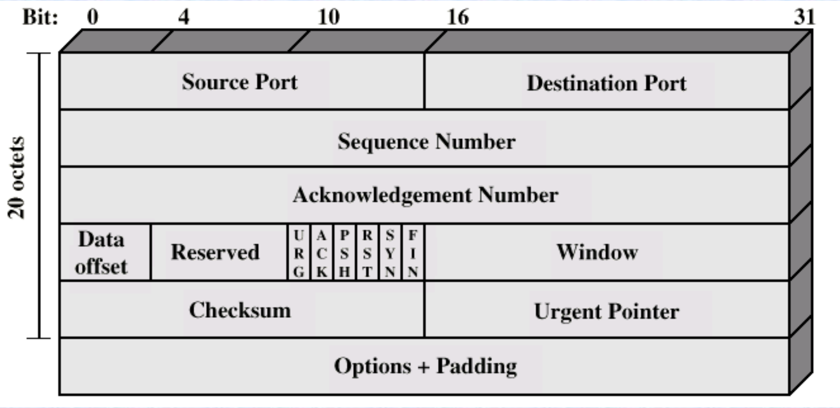
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

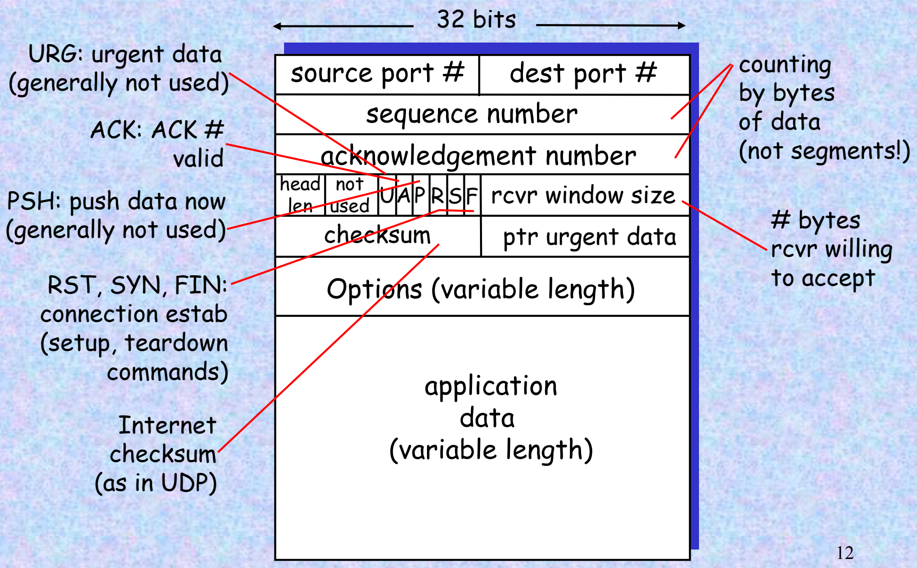


- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP Header



TCP Segment Structure



URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept

Part V

Reliability in an Unreliable World

Reliability in an Unreliable World

- IP offers best-effort (unreliable) delivery
- TCP uses IP
- TCP provides completely reliable transfer
- How is this possible? How can TCP realize:
  - Reliable connection startup?
  - Reliable data transmission?
  - Graceful connection shutdown?

Reliability Data Transmission

- **Positive acknowledgment**
  - Receiver returns short message when data arrives
  - Called *acknowledgment*
- **Retransmission**
  - Sender starts timer whenever message is transmitted
  - If timer expires before acknowledgment arrives, sender *retransmits* message
  - THIS IS NOT A TRIVIAL PROBLEM! – more on this later

Part VI

TCP Flow Control

TCP Flow Control

- **Receiver**
  - Advertises available buffer space
  - Called *window*
  - This is a known as a ***CREDIT*** policy
- **Sender**
  - Can send up to entire window before ACK arrives
- Each acknowledgment carries new window information
  - Called *window advertisement*
  - Can be zero (called *closed window*)
- Interpretation: I have received up through X, and can take *Y* more octets

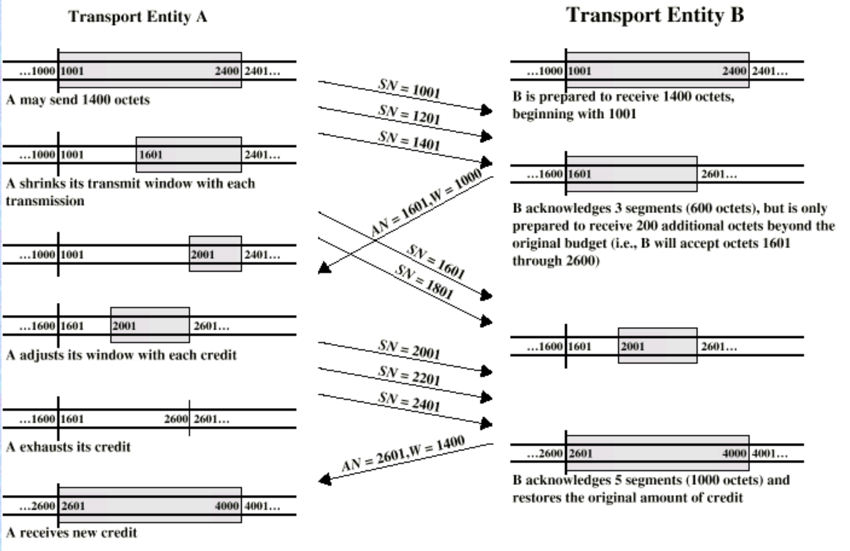
Credit Scheme

- Decouples flow control from ACK
  - May ACK without granting credit and vice versa
- Each octet has sequence number
- Each transport segment has seq number, ack number and window size in header

Use of Header Fields

- When sending, seq number is that of first octet in segment
- ACK includes AN=i, W=j
- All octets through SN=i-1 acknowledged
  - Next expected octet is I
- Permission to send additional window of W=j octets
  - i.e. octets through i+j-1

Credit Allocation



Transport Entity A

Transport Entity B

A may send 1400 octets

A shrinks its transmit window with each transmission

A adjusts its window with each credit

A exhausts its credit

A receives new credit

B is prepared to receive 1400 octets, beginning with 1001

B acknowledges 3 segments (600 octets), but is only prepared to receive 200 additional octets beyond the original budget (i.e., B will accept octets 1601 through 2600)

B acknowledges 5 segments (1000 octets) and restores the original amount of credit

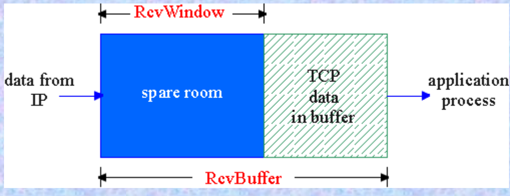
TCP Flow Control

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

`RcvBuffer` = size of TCP Receive Buffer

`RcvWindow` = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- **`RcvWindow` field** in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received **`RcvWindow`**

TCP Seq. #'s and ACKs

Seq. #'s:

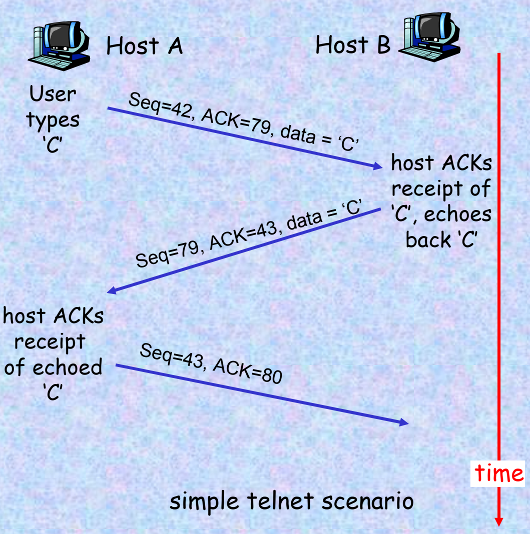
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



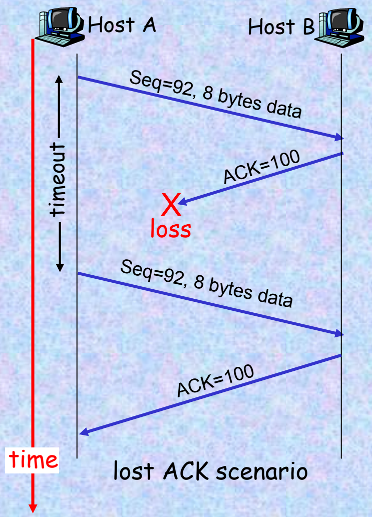
Host A
Host B
User types 'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80
time

simple telnet scenario

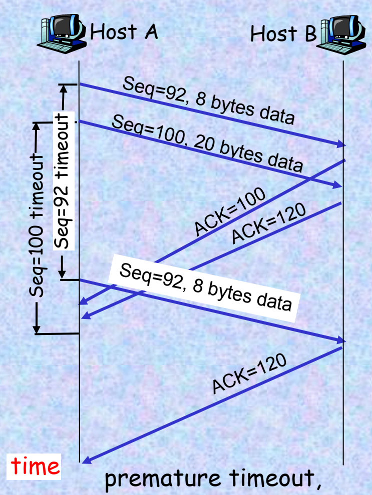
TCP ACK Generation
[RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms 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-expect 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

TCP: Retransmission Scenarios



lost ACK scenario



premature timeout,
cumulative ACKs

Part VII

Why Startup / Shutdown Difficult?

Why Startup / Shutdown Difficult?

- Segments can be
  - Lost
  - Duplicated
  - Delayed
  - Delivered out of order
  - Either side can crash
  - Either side can reboot
- Need to avoid duplicate ‘‘shutdown’’ message from affecting later connection

Part VIII

TCP Connection Management

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

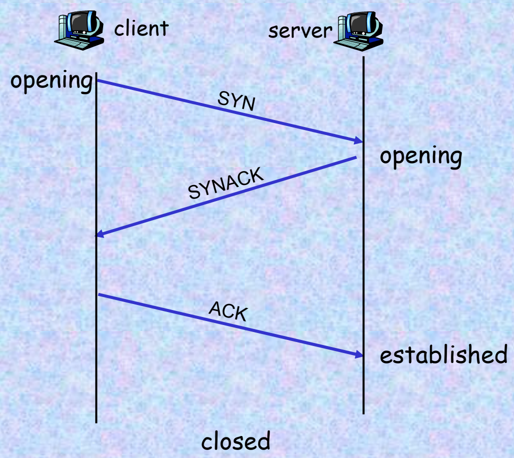
  Socket connectionSocket = welcomeSocket.accept();

Three way handshake:

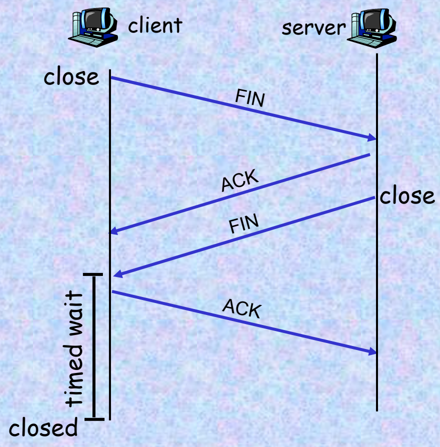
Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

TCP Connection Management (OPEN)



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.

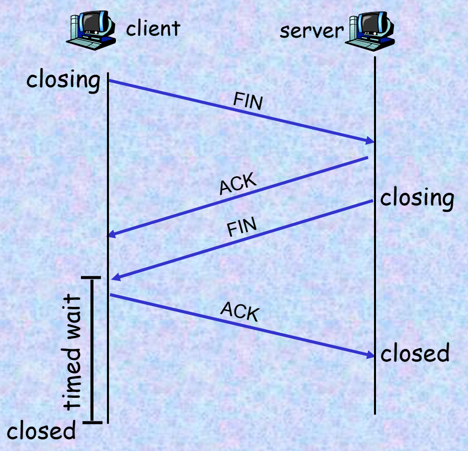
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

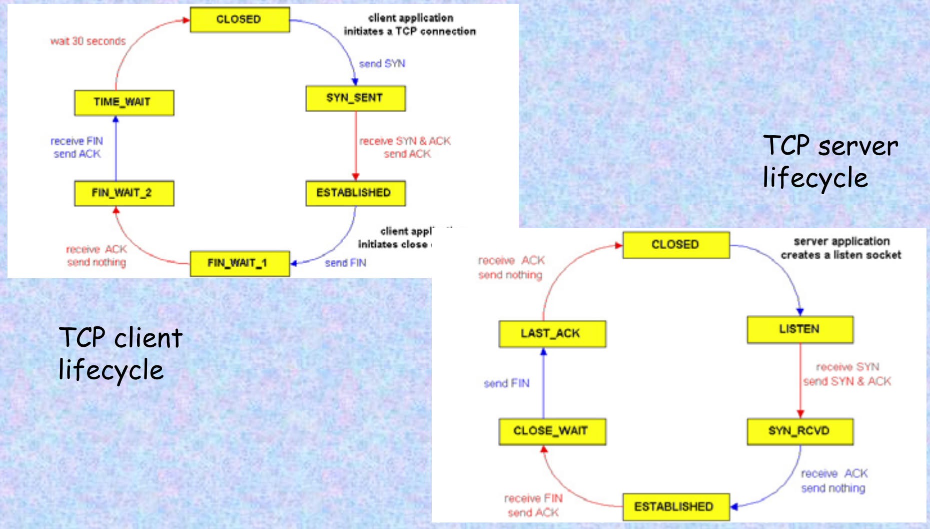
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

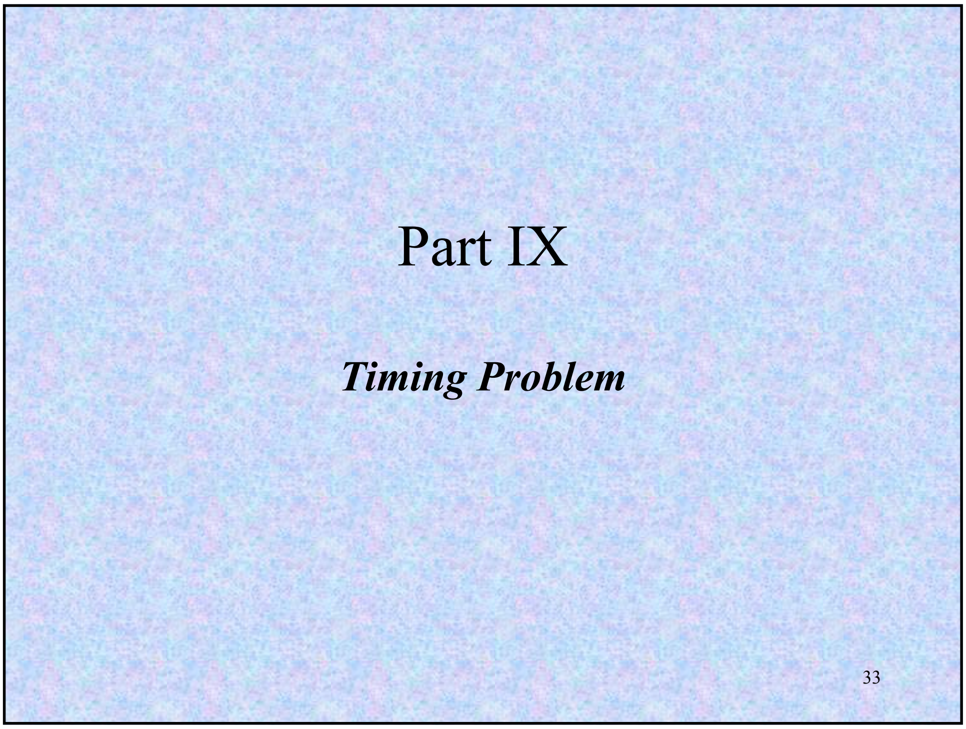


TCP Connection Management (cont.)



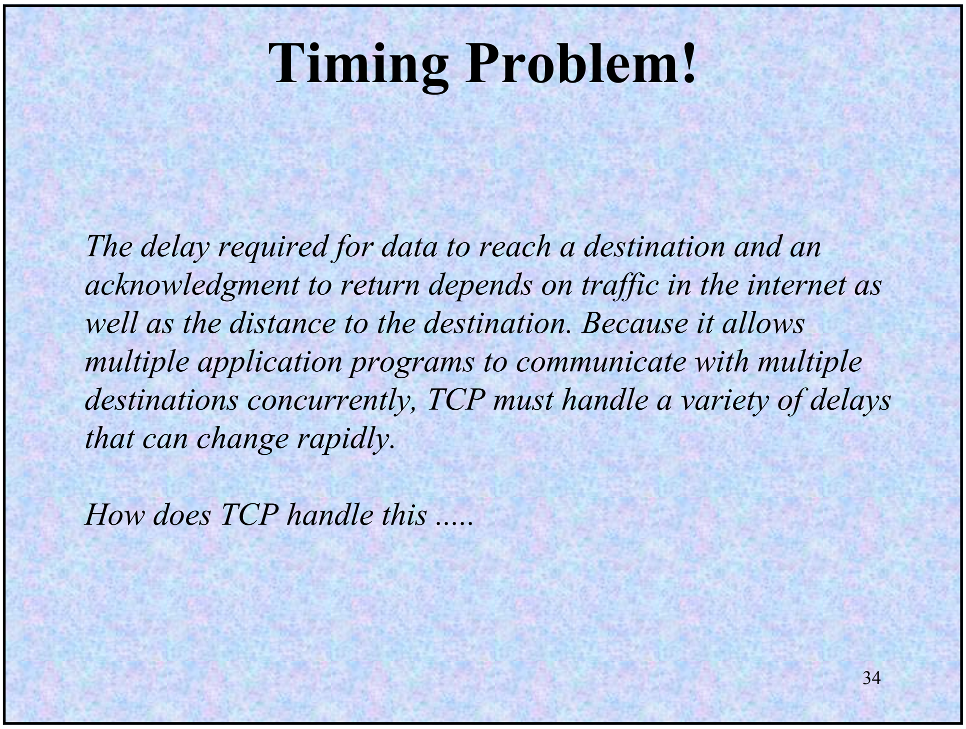
TCP server lifecycle

TCP client lifecycle



Part IX

Timing Problem



Timing Problem!

The delay required for data to reach a destination and an acknowledgment to return depends on traffic in the internet as well as the distance to the destination. Because it allows multiple application programs to communicate with multiple destinations concurrently, TCP must handle a variety of delays that can change rapidly.

How does TCP handle this .....

Solving Timing Problem

- Keep estimate of round trip time on each connection
- Use current estimate to set retransmission timer
- Known as ***adaptive retransmission***
- Key to TCP's success

TCP Round Trip Time & Timeout

- Q: how to set TCP timeout value?
- 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`**

TCP Round Trip Time & Timeout

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
```

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

<u>Setting the timeout</u>

- **EstimatedRTT** plus “safety margin”
- large variation in **EstimatedRTT ->** larger safety margin

```
    Timeout = EstimatedRTT + 4*Deviation

  Deviation = (1-x)*Deviation +
                x*|SampleRTT-EstimatedRTT|
```

Part X

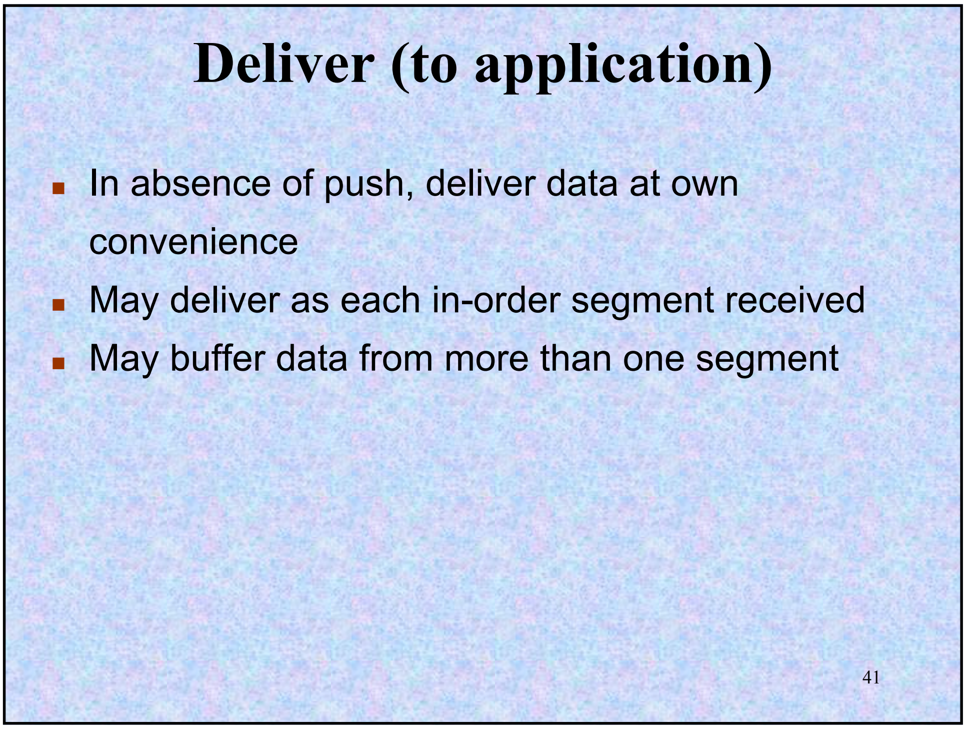
Implementation Policy Options

Implementation Policy Options

- Send
- Deliver
- Accept
- Retransmit
- Acknowledge

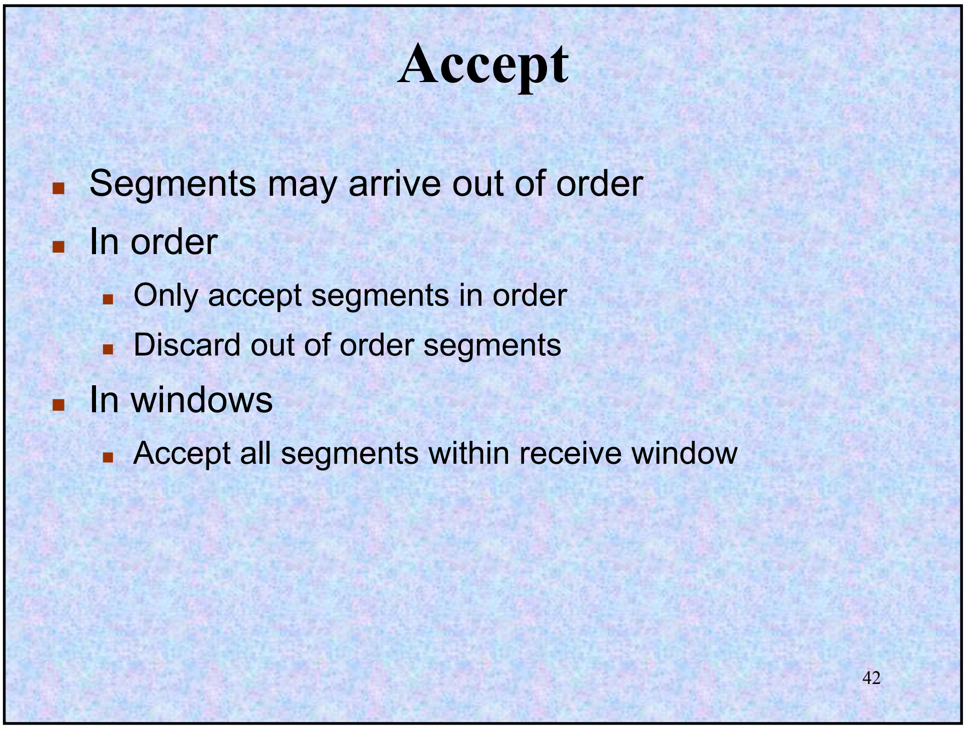
Send

- If no push or close TCP entity transmits at its own convenience (IFF send window allows!)
- Data buffered at transmit buffer
- May construct segment per data batch
- May wait for certain amount of data



Deliver (to application)

- In absence of push, deliver data at own convenience
- May deliver as each in-order segment received
- May buffer data from more than one segment



Accept

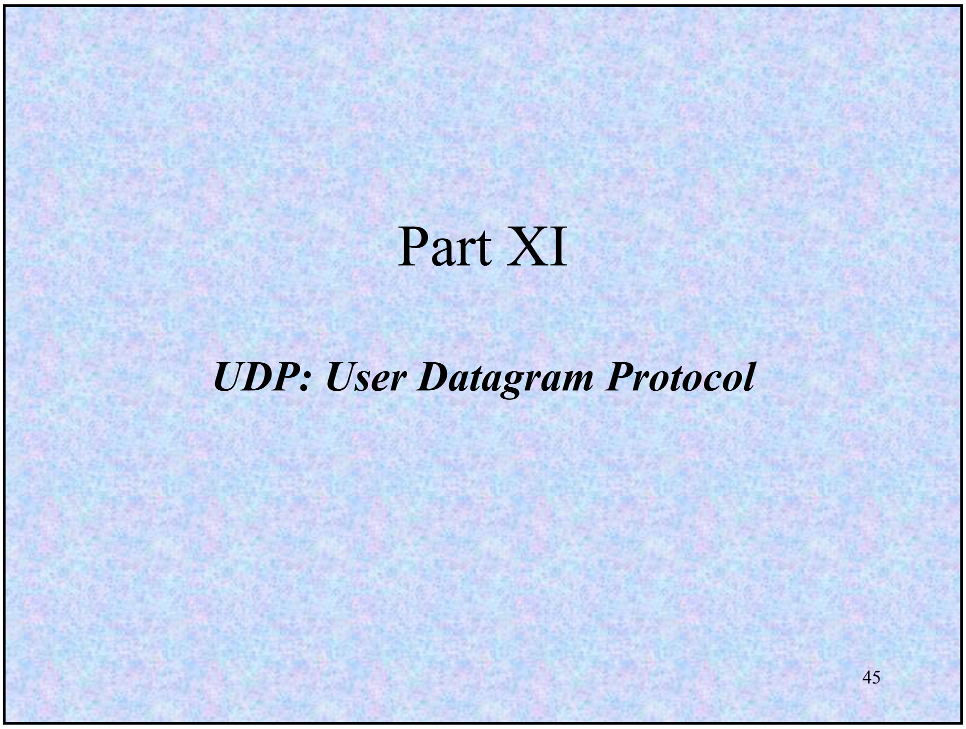
- Segments may arrive out of order
- In order
  - Only accept segments in order
  - Discard out of order segments
- In windows
  - Accept all segments within receive window

Retransmit

- TCP maintains queue of segments transmitted but not acknowledged
- TCP will retransmit if not ACKed in given time
  - First only
  - Batch
  - Individual

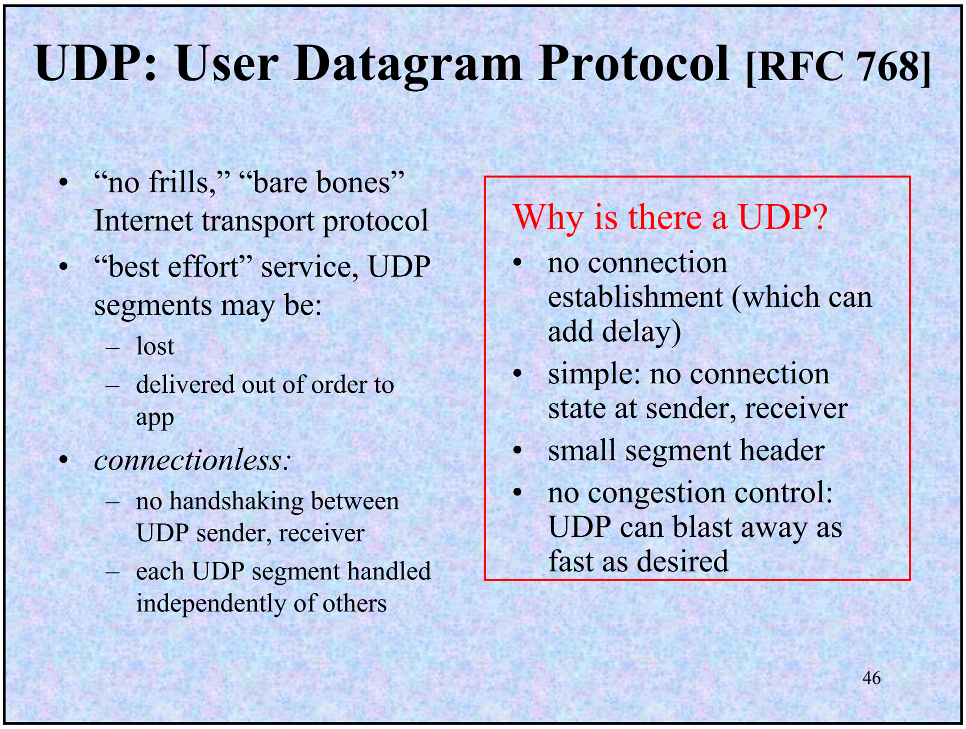
Acknowledgement

- Immediate
  - as soon as segment arrives.
  - will introduce extra network traffic
  - Keeps sender’s pipe open
- Cumulative
  - Wait a bit before sending ACK (called “delayed ACK”)
  - Must use timer to insure ACK is sent
  - Less network traffic
  - May let sender’s pipe fill if not timely!



Part XI

UDP: User Datagram Protocol



UDP: User Datagram Protocol [RFC 768]

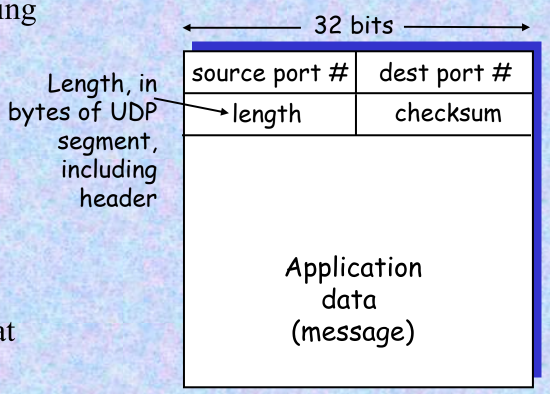
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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

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!



Length, in bytes of UDP segment, including header

UDP segment format

UDP Uses

- Inward data collection
- Outward data dissemination
- Request-Response
- Real time application

Part XII

Conclusion

Assignment & Readings

- Assignment #5 (due 04/15/10)
  - Assigned at the completion of Session 9
- Readings
  - Chapter 3 (3.5)
  - RFC 793 (introduction, sections 1 and 2)

Next Session:

Network Congestion