Module 8
Transport Layer Protocols
Please note: Most of these slides come from this book. Note their copyright notice below…

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 1996-2010
J.F Kurose and K.W. Ross, All Rights Reserved

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.
Transport services & protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP
Transport Layer is End-to-End
# Internet Protocols

<table>
<thead>
<tr>
<th>Physical</th>
<th>Data Link</th>
<th>Network</th>
<th>Transport</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X.25</td>
<td>IP</td>
<td>TCP</td>
<td>FTP, Telnet, NFS, SMTP, HTTP, ...</td>
</tr>
<tr>
<td></td>
<td>Ethernet</td>
<td></td>
<td>UDP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Packet Radio</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ATM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FDDI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Internet Apps: Their Protocols & Transport Protocols

<table>
<thead>
<tr>
<th>Applications</th>
<th>Data Loss</th>
<th>Throughput</th>
<th>Time Sensitive</th>
<th>Application Layer Protocol</th>
<th>Transport Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email</td>
<td>No loss</td>
<td>Elastic</td>
<td>No</td>
<td>smtp</td>
<td>TCP</td>
</tr>
<tr>
<td>remote terminal access</td>
<td>No loss</td>
<td>Elastic</td>
<td>Yes</td>
<td>telnet</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>No loss</td>
<td>Elastic</td>
<td>No</td>
<td>http</td>
<td>TCP</td>
</tr>
<tr>
<td>File transfer</td>
<td>No loss</td>
<td>Elastic</td>
<td>No</td>
<td>ftp</td>
<td>TCP</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>Loss tolerant</td>
<td>audio: 5kbps-1Mbps video: 10kbps-5Mbps</td>
<td>Yes, 100’s msec</td>
<td>Proprietary</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Remote file server</td>
<td>No loss</td>
<td>Elastic</td>
<td>No</td>
<td>NFS</td>
<td>TCP or UDP (typically UDP)</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>Loss tolerant</td>
<td>Depends on encoding, 32kbps typically</td>
<td>Yes, few secs</td>
<td>SIP, RIP, Proprietary</td>
<td>TCP or UDP (typically UDP)</td>
</tr>
</tbody>
</table>
Internet transport-layer protocols

- **UDP**: unreliable, unordered delivery
  - Process-to-process data delivery
    - Multiplexing/demultiplexing
    - End-to-end error checking
- **TCP**: reliable, in-order delivery
  - Congestion control
  - Flow control
  - Connection setup
- Services not available:
  - Delay guarantees
  - Bandwidth guarantees
The programmer's conceptual view of a TCP/IP Internet

© Addison-Wesley Publishers 2000
How Apps Access Transport Services

- Through sockets

Socket: a *host-local, application-created, OS-controlled* interface (a “door”) into which application process can both send and receive messages to/from another application process

Socket API
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented
UDP: User Datagram Protocol

- “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
**Client/server socket interaction: UDP**

**Server (running on hostid)**

create socket, port= x.
serverSocket = DatagramSocket()

read datagram from serverSocket

write reply to serverSocket specifying client address, port number

**Client**

create socket, clientSocket = DatagramSocket()

Create datagram with server IP and port=x; send datagram via clientSocket

read datagram from clientSocket

close clientSocket
UDP Use and Format

- 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 recovery!
UDP checksum

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.
TCP: Transport Control Protocol

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order byte stream:**
  - 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) inits sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver
Socket programming with TCP

Client must contact server
• server process must first be running
• server must have created socket (door) that welcomes client’s contact

Client contacts server by:
• creating client-local TCP socket
• specifying IP address, port number of server process
• when client creates socket: client TCP establishes connection to server TCP

• when contacted by client, server TCP creates new socket for server process to communicate with client
  → allows server to talk with multiple clients
  → source port numbers used to distinguish clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server
TCP Client/server socket interaction

Server (running on **hostid**)  
create socket, port=x, for incoming request:  
```java
welcomeSocket = ServerSocket()
```
wait for incoming connection request  
```java
connectionSocket = welcomeSocket.accept()
```
read request from `connectionSocket`
write reply to `connectionSocket`
close `connectionSocket`

Client  
create socket, connect to **hostid**, port=x  
```java
clientSocket = Socket()
```
send request using `clientSocket`
read reply from `clientSocket`
close `clientSocket`
Stream Jargon

- **stream** is a sequence of characters that flow into or out of a process.
- **input stream** is attached to some input source for the process, e.g., keyboard or socket.
- **output stream** is attached to an output source, e.g., monitor or socket.
Socket programming with TCP

Example client-server app:

1) client reads line from standard input (\texttt{inFromUser} stream), sends to server via socket (\texttt{outToServer} stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (\texttt{inFromServer} stream)
Example: TCP Java client

```java
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        create input stream
        create clientSocket object of type Socket, connect to server
        create output stream attached to socket

        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```
Example: TCP Java client (cont’d)

```java
BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + '\n');
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();
```

- create input stream attached to socket
- send line to server
- read line from server
- close socket (clean up behind yourself!)
Example: TCP Java server

```java
import java.io.*;
import java.net.*;

class TCPServer {

    public static void main(String argv[]) throws Exception {

        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {

            Socket connectionSocket = welcomeSocket.accept();

            BufferedReader inFromClient =
                new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));

        }

    }

}
```
Example: TCP Java server (cont’d)

create output stream, attached to socket → DataOutputStream outToClient =
               new DataOutputStream(connectionSocket.getOutputStream());

read in line from socket → clientSentence = inFromClient.readLine();

          capitalizedSentence = clientSentence.toUpperCase() + '\n';

write out line to socket → outToClient.writeBytes(capitalizedSentence);

} } }}

end of while loop, loop back and wait for another client connection
TCP segment structure

- **Source port #**
- **Destination port #**
- **Sequence number**
- **Acknowledgement number**
- **Receive window**
- **Checksum**
- **Urg data ptr**
- **Options (variable length)**
- **Application data (variable length)**

- **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**
Principles of Reliable Data Transfer

- important in app., transport, link layers

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol

- Note: slides use the term “packet” but at transport layer, these are segments
Principles of Reliable Data Transfer

**send side**

- **rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer.

- **udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver.

**receive side**

- **deliver_data()**: called by rdt to deliver data to upper.

- **rdt_rcv()**: called when packet arrives on rcv-side of channel.

- **rdt_rcv()**: called when packet arrives on rcv-side of channel.
Reliable Transfer over a Reliable Channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel
What can go wrong (1)?

- Underlying channel may flip bits in segment
- Error detection:
  - Checksum to detect bit errors
- Recovering from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that segment received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that segment had errors
  - sender retransmits segment on receipt of NAK
- *Stop-and-wait*
  - Sender sends one segment, then waits for the receiver to respond
  - We will come back to this later
Handling duplicates

• What happens if ACK/NAK corrupted?
  ➡ sender doesn’t know what happened at receiver!
  ➡ can’t just retransmit: possible duplicate

• Sender retransmits current segment if ACK/NAK garbled

• Sender adds sequence number to each segment
  ➡ For stop-and-go protocol a 1-bit sequence number with modulo-2 arithmetic is sufficient

• Receiver discards (does not deliver up to the application) duplicate segments
What can go wrong (2)?

- Segments (data or ACK) may be lost
- Sender waits a “reasonable” amount of time for ACK
  - Retransmits if no ACK received in this time
  - If segment (data or ACK) simply delayed (not lost)
    - Retransmission will be duplicate, but use of sequence numbers already handles this
    - Receiver must specify the sequence number of segment being ACKed
  - Requires countdown timer

- Sequence numbers
  - For data: byte stream “number” of first byte in segment’s data
  - For ACKs: if pipelined, segment number of next byte expected from other side
    - Cumulative ACK
What can go wrong (3)?

- Data segments may come out of order
- TCP specification does not say what to do
- Two alternatives
  - Receiver discards out of order segments
    - Simplifies receiver design
    - Wastes network bandwidth
  - Receiver keeps out of order segments and fills in the missing ones when they arrive
    - Usually this is what is implemented
TCP Sender Events

data rcvd from app:
- Create segment with sequence number
- 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
TCP sender
(simplified)

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 Round Trip Time and Timeout

• How to set TCP timeout value?
  ➡ longer than RTT
    ✦ but RTT varies
  ➡ too short: premature timeout
    ✦ unnecessary retransmissions
  ➡ too long: slow reaction to segment loss

• How to estimate RTT?
  ➡ SampleRTT: measured time from segment transmission until ACK receipt
    ✦ ignore retransmissions
  ➡ SampleRTT will vary, want estimated RTT “smoother”
    ✦ average several recent measurements, not just current SampleRTT
TCP Flow Control

- Receive side of TCP connection has a receive buffer:

  - Speed-matching service: matching the send rate to the receiving app’s drain rate

- App process may be slow at reading from buffer

- Flow control: sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP Flow control: how it works

- spare room in buffer (ignoring out-of-order segments)
  \[ \text{RcvWindow} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}) \]
- Receiver advertises spare room by including value of \textbf{RcvWindow} in segments
- Sender limits unACKed data to \textbf{RcvWindow}
  - guarantees receive buffer doesn’t overflow
**TCP Connection Management**

**Recall:** TCP is a connection-oriented protocol

- sender, receiver establish “connection” before exchanging data segments
- initialize TCP variables:
  - Sequence numbers
  - buffers, flow control info (e.g. RcvWindow)
- **client**: connection initiator

  ```java
  Socket clientSocket = new Socket("hostname","port number");
  ```
- **server**: contacted by client

  ```java
  Socket connectionSocket = welcomeSocket.accept();
  ```
Three-Way Handshake

**Step 1:** client host sends TCP SYN segment to server
- specifies initial sequence number
- no data

**Step 2:** server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial sequence number

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data
## Closing a TCP Connection

client closes socket:
```java
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.

**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.
Principles of Congestion Control

• **Congestion**: informally, “too many sources sending too much data too fast for the network to handle”

• different from flow control!

• manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
Causes/costs of congestion: Simple Scenario

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput
Causes and Effects of Congestion

- With finite buffers at routers, packets may be dropped as $\lambda_{in}$ increases
  - Retransmission needed
  - Offered load $\lambda'_{in} > \lambda_{in}$
    - More work (retransmissions) for given $\lambda_{out}$
    - Unneeded retransmissions: link carries multiple copies of segment
- With multi-hop connections, upstream routers receive two types of traffic:
  - Forwarded traffic from downstream routers
  - Traffic they may receive directly from hosts
  - As $\lambda'_{in}$ increases, more dropped packages and more transmissions
    - $\lambda_{out}$ will approach 0
    - When packet dropped, any upstream transmission capacity used for that packet was wasted!
Approaches to Congestion Control

Two broad approaches towards congestion control:

1. **end-end congestion control:**
   - no explicit feedback from network
   - congestion inferred from end-system observed loss, delay
   - approach taken by TCP

2. **network-assisted congestion control:**
   - routers provide feedback to end systems
     - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
     - explicit rate sender should send at