Chapter 3
Transport Layer

Reti degli Elaboratori
Canale AL
Prof.ssa Chiara Petrioli
a.a. 2013/2014

We thank for the support material Prof. Kurose-Ross
All material copyright 1996-2012
© J.F Kurose and K.W. Ross, All Rights Reserved
Chapter 3: Transport Layer

our goals:
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control
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
Transport services and 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 vs. network layer

- **network layer**: logical communication between hosts
  - relies on, enhances, network layer services

- **transport layer**: logical communication between processes

  household analogy:
  12 kids in Ann’s house sending letters to 12 kids in Bill’s house:
  - hosts = houses
  - processes = kids
  - app messages = letters in envelopes
  - transport protocol = Ann and Bill who demux to in-house siblings
  - network-layer protocol = postal service
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
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
**Multiplexing/demultiplexing**

**Multiplexing at sender:**
handle data from multiple sockets, add transport header (later used for demultiplexing)

**Demultiplexing at receiver:**
use header info to deliver received segments to correct socket
How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses **IP addresses & port numbers** to direct segment to appropriate socket

TCP/UDP segment format:

- source port #
- dest port #
- other header fields
- application data (payload)
- 32 bits

Transport Layer 3-9
Connectionless demultiplexing

- **recall**: created socket has host-local port #: 
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  ```

- **recall**: when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

  IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest
Connectionless demux: example

```
DatagramSocket serverSocket = new DatagramSocket (6428);
DatagramSocket mySocket1 = new DatagramSocket (5775);
DatagramSocket mySocket2 = new DatagramSocket (9157);
```

Transport Layer 3-11
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request
Connection-oriented demux: example

three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets
Connection-oriented demux: example

Transport Layer 3-14
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
**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

- **UDP use:**
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP

- **reliable transfer over UDP:**
  - add reliability at application layer
  - application-specific error recovery!
**UDP: segment header**

<table>
<thead>
<tr>
<th></th>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

**UDP segment format**

- 32 bits
- Length, in bytes of UDP segment, including header
- Application data (payload)

**why is there a UDP?**

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

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

**sender:**
- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one’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. *But maybe errors nonetheless? More later* ....
Internet checksum: example

example: add two 16-bit integers

\[
\begin{array}{cccccccccccccccccccc}
& 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
\text{wraparound} & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
\text{sum} & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
\text{checksum} & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
\]

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
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
Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable data transfer: getting started

**Reliable data transfer protocol (sending 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

**Reliable data transfer protocol (receiving side)**
- `delivered_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
- `deliver_data()`: called by `rdt` to deliver data to upper
- `udt_send()`: called by `rdt`, to transfer packet over unreliable channel to receiver
Reliable data transfer: getting started

we’ll:

- incrementally develop sender, receiver sides of **reliable data transfer protocol (rdt)**
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

![Diagram with states and transitions]

**state:** when in this “state” next state uniquely determined by next event

**event causing state transition**

**actions taken on state transition**

**state 1**

**state 2**
rdt 1.0: 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 reads data from underlying channel

sender

receiver

```plaintext
<table>
<thead>
<tr>
<th>rdt_send(data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>packet = make_pkt(data)</td>
</tr>
<tr>
<td>udt_send(packet)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rdt_rcv(packet)</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract (packet, data)</td>
</tr>
<tr>
<td>deliver_data(data)</td>
</tr>
</tbody>
</table>
```
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:

How do humans recover from “errors” during conversation?
**rdt2.0: channel with bit errors**

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in *rdt2.0 (beyond rdt1.0)*:
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender
rdt2.0: FSM specification

**Sender**

- `rdt_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `Λ`

**Receiver**

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`
rdt2.0: operation with no errors

```
rdt_send(data)

snkpkt = make_pkt(data, checksum)

udt_send(snkpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

udt_send(ACK)

extract(rcvpkt, data)

deliver_data(data)

udt_send(NAK)

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

Wait for call from below

Lambda
```
rdt2.0: error scenario
**rdt2.0 has a fatal flaw!**

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

**handling duplicates:**
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn’t deliver up) duplicate pkt

**stop and wait**
sender sends one packet, then waits for receiver response
rdt2.1: sender, handles garbled ACK/NAKs

\[
\text{rdt\_send}(\text{data}) \\
\text{sndpkt} = \text{make\_pkt}(0, \text{data}, \text{checksum}) \\
\text{udt\_send(sndpkt)} \\
\]

Wait for call 0 from above

\[
\text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt) \&\& isACK(rcvpkt)} \\
\Lambda \\
\text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt) \&\& isACK(rcvpkt)} \\
\Lambda \\
\]

Wait for ACK or NAK 0

Wait for ACK or NAK 1

\[
\text{rdt\_send(data)} \\
\text{sndpkt} = \text{make\_pkt}(1, \text{data}, \text{checksum}) \\
\text{udt\_send(sndpkt)} \\
\]

Wait for call 1 from above

\[
\text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt) \&\& isACK(rcvpkt)} \\
\Lambda \\
\text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt) \&\& isACK(rcvpkt)} \\
\Lambda \\
\]

\[
\text{udt\_send(sndpkt)} \\
\text{udt\_send(sndpkt)} \\
\]

\[
\text{udt\_send(sndpkt)} \\
\text{udt\_send(sndpkt)} \\
\]
**rdt2.1: receiver, handles garbled ACK/NAKs**

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) 
&& has_seq0(rcvpkt)
  extract(rcvpkt, data)
  deliver_data(data)
  sndpkt = make_pkt(ACK, chksum)
  udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
  sndpkt = make_pkt(ACK, chksum)
  udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) &&
  has_seq1(rcvpkt)
  sndpkt = make_pkt(ACK, chksum)
  udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) 
&& has_seq1(rcvpkt)
  extract(rcvpkt, data)
  deliver_data(data)
  sndpkt = make_pkt(ACK, chksum)
  udt_send(sndpkt)
```
**rdt2.1: discussion**

**sender:**
- seq # added to pkt
- two seq. #’s (0, 1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

**receiver:**
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

Transport Layer 3-35
rdt2.2: a **NAK-free protocol**

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*
rdt2.2: sender, receiver fragments

```plaintext
rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

Wait for call 0 from above

sender FSM fragment

Wait for ACK 0

rdt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
  isACK(rcvpkt, 1) )
udt_send(sndpkt)

receiver FSM fragment

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
&& isACK(rcvpkt, 0)

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))
udt_send(sndpkt)

Wait for 0 from below

extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
```

Transport Layer 3-37
**rdt3.0: channels with errors and loss**

**new assumption:** underlying channel can also lose packets (data, ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

**approach:** sender waits “reasonable” amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer
rdt3.0 sender

```
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

ravel for call 0 from above
```

```
ravt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
```

```
timeout
udt_send(sndpkt)
start_timer
```

```
ravt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
```

```
ravt_rcv(rcvpkt)
```

```
rdt_send(data)
```

```
sndpkt = make_pkt(1, data, checksum)
```

```
udt_send(sndpkt)
start_timer
```

```
ravt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
isACK(rcvpkt,1) )
```

```
Lambda
```

Transport Layer 3-39
rdt3.0 in action

(a) no loss

(b) packet loss

Transport Layer 3-40
rdt3.0 in action

sender

send pkt0

rcv pkt0
send ack0

rcv pkt1
send ack1

rcv pkt1
send ack1

timeout
resend pkt1

rcv pkt1
(send pkt0)

rcv ack0
send pkt1

rcv ack0
send pkt1

rcv ack1
(send pkt0)

rcv ack1
(send pkt0)

rcv ack1
(send pkt0)

rcv ack0
send ack1

rcv ack0
send ack1

rcv ack0
send ack1

(c) ACK loss

(d) premature timeout/ delayed ACK

receiver

send pkt0

rcv pkt0
rcv pkt0
rcv pkt1

rcv pkt1
rcv pkt1
rcv pkt1

rcv(pkt0)
send ack0

rcv(pkt0)
send ack0

rcv(pkt0)
send ack0

rcv(pkt0)
send ack0

rcv(pkt0)
send ack0

rcv(pkt0)
send ack0

pecia