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

*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
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

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data (payload)</td>
<td></td>
</tr>
</tbody>
</table>

32 bits
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);
```

P3

transport

network

link

physical

P4

transport

network

link

physical

source port: 6428
dest port: 9157

source port: 9157
dest port: 6428

source port: ?
dest port: ?

source port: ?
dest port: 9157

source port: 9157
dest port: 6428

source port: ?
dest port: ?
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**: protocol that provides connection services with flow control and error recovery.
- **Application**: protocol that performs high-level functions like file transfer or interactive sessions.
- **Physical** and **Link**: layers that handle the raw transmission of data over a physical medium.

**Example**: Source IP, port: A, 9157 to destination IP, port: B, 80. Threaded server at IP address B.

- **Host**: IP address A
- **Destination**: IP address B
- **Source**: IP address C
- **Network Location**: server at IP address B

**Diagram**: Connections between the application, transport, network, link, and physical layers are illustrated with arrows indicating the flow of data.
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

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 segment format

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

application data (payload)

32 bits

Length, in bytes of UDP segment, including header
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

\[
\begin{array}{ccccccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
\]

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
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

**send side**

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

**receive side**

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

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

- **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

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

<table>
<thead>
<tr>
<th>state</th>
<th>event causing state transition</th>
<th>actions taken on state transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>event</td>
<td>actions</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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

```
Wait for call from above
rdt_send(data)
  packet = make_pkt(data)
  udt_send(packet)

sender
```

```
Wait for call from below
rdt_rcv(packet)
  extract (packet, data)
  deliver_data(data)

receiver
```
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)

wait for call from above

wait for ACK or NAK

rdt_rcv(rcvpkt) &&

isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&

isACK(rcvpkt)

receiver

wait for call from below

rdt_rcv(rcvpkt) &&

corrupt(rcvpkt)

udt_send(NAK)

extract(rcvpkt, data)

deliver_data(data)

udt_send(ACK)
rdt2.0: operation with no errors

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

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Lambda

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)

Wait for call from below

corrupt(rcvpkt)
udt_send(NAK)
rdt2.0: error scenario

\[
\begin{align*}
\text{rdt}_\text{send}(\text{data}) \\
\text{snkpkt} &= \text{make}_\text{pkt}(\text{data}, \text{checksum}) \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\text{rdt}_\text{rcv}(\text{rcvpkt}) && \text{isNAK}(\text{rcvpkt})
\end{align*}
\]

\[
\begin{align*}
\text{udt}_\text{send}(\text{sndpkt}) \\
r\text{dt}_\text{rcv}(\text{rcvpkt}) \\
\text{notcorrupt}(\text{rcvpkt})
\end{align*}
\]

\[
\begin{align*}
\text{udt}_\text{send}(\text{NAK}) \\
r\text{dt}_\text{rcv}(\text{rcvpkt}) && \text{isACK}(\text{rcvpkt})
\end{align*}
\]

\[
\begin{align*}
\Lambda \\
\text{wait for \text{call from \text{above}}}
\end{align*}
\]

\[
\begin{align*}
\text{rdt}_\text{rcv}(\text{rcvpkt}) \\
\text{extract}(\text{rcvpkt}, \text{data}) \\
\text{deliver}_\text{data}(\text{data}) \\
\text{udt}_\text{send}(\text{ACK})
\end{align*}
\]

\[
\begin{align*}
\text{wait for \text{call from \text{below}}}
\end{align*}
\]