Topic 5 – Transport

Our goals:
• understand principles behind transport layer services:
  – multiplexing/demultiplexing
  – reliable data transfer
  – flow control
  – congestion control
  – buffers

• learn about transport layer protocols in the Internet:
  – UDP: connectionless transport
  – TCP: connection-oriented transport
  – TCP congestion control
  – TCP flow control
Transport Layer

- Commonly a layer at end-hosts, between the application and network layer
Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application/processes/tasks at hosts
  - Need a way to decide which packets go to which applications (*more multiplexing*)
Why a transport layer?
Why a transport layer?

many application processes

---

Host A

---

Host B

---

Application
Transport
Network
Datalink
Physical
Why a transport layer?

Communication between processes at hosts

Communication between hosts

(128.4.5.6 ←→ 162.99.7.56)

Host A

Host B

many application processes
Why a transport layer?

• IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  – Need a way to decide which packets go to which applications (mux/demux)

• IP provides a weak service model (*best-effort*)
  – Packets can be corrupted, delayed, dropped, reordered, duplicated
  – No guidance on how much traffic to send and when
  – Dealing with this is tedious for application developers
Role of the Transport Layer

- Communication between application processes
  - Multiplexing between application processes
  - Implemented using *ports*
Role of the Transport Layer

• Communication between application processes
• Provide common end-to-end services for app layer [optional]
  – Reliable, in-order data delivery
  – Paced data delivery: flow and congestion-control
    • too fast may overwhelm the network
    • too slow is not efficient

(Just Like Computer Networking Lectures....)
Role of the Transport Layer

• Communication between processes
• Provide common end-to-end services for app layer [optional]
• TCP and UDP are the common transport protocols
  – also SCTP, MTCP, SST, RDP, DCCP, ...
Role of the Transport Layer

• Communication between processes
• Provide common end-to-end services for app layer [optional]
• TCP and UDP are the common transport protocols
• UDP is a minimalist, no-frills transport protocol
  – only provides mux/demux capabilities
Role of the Transport Layer

• Communication between processes
• Provide common end-to-end services for app layer [optional]
• TCP and UDP are the common transport protocols
• UDP is a minimalist, no-frills transport protocol
• TCP is the *totus porcus* protocol
  – offers apps a reliable, in-order, byte-stream abstraction
  – with congestion control
  – but no performance (delay, bandwidth, ...) guarantees
Role of the Transport Layer

- Communication between processes
  - mux/demux from and to application processes
  - implemented using ports
Context: Applications and Sockets

• Socket: software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system
  – socketID = socket(…, socket.TYPE)
  – socketID.sendto(message, …)
  – socketID.recvfrom(…)

• Two important types of sockets
  – UDP socket: TYPE is SOCK_DGRAM
  – TCP socket: TYPE is SOCK_STREAM
Ports

- Problem: deciding which app (socket) gets which packets

  - Solution: *port* as a transport layer identifier
    - 16 bit identifier
      - OS stores mapping between sockets and *ports*
      - a packet carries a source and destination port number in its transport layer header

- For UDP ports (SOCK_DGRAM)
  - OS stores (local port, local IP address) \(\leftrightarrow\) socket

- For TCP ports (SOCK_STREAM)
  - OS stores (local port, local IP, remote port, remote IP) \(\leftrightarrow\) socket
<table>
<thead>
<tr>
<th>4-bit Version</th>
<th>4-bit Header Length</th>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
<td>13-bit Fragment Offset</td>
<td></td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
<td>16-bit Header Checksum</td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP Payload</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identification</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protocol</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IP Payload
<table>
<thead>
<tr>
<th>Field</th>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of Service (TOS)</td>
<td>8</td>
<td>4-bit Type of Service Type (TOS)</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16</td>
<td>16-bit Total Length (Bytes)</td>
</tr>
<tr>
<td>Identification</td>
<td>16</td>
<td>16-bit Identification</td>
</tr>
<tr>
<td>Flags</td>
<td>3</td>
<td>3-bit Flags</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13</td>
<td>13-bit Fragment Offset</td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8</td>
<td>8-bit Time to Live (TTL)</td>
</tr>
<tr>
<td>Protocol</td>
<td>6, 17</td>
<td>6 = TCP, 17 = UDP</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16</td>
<td>16-bit Header Checksum</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32</td>
<td>32-bit Source IP Address</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32</td>
<td>32-bit Destination IP Address</td>
</tr>
</tbody>
</table>

TCP or UDP header and Payload
<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-bit Time to Live (TTL)</td>
<td>6 = TCP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Source IP Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Destination IP Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16-bit Source Port</td>
<td>16-bit Destination Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TCP or UDP</td>
<td>header and Payload</td>
</tr>
</tbody>
</table>
Recap: Multiplexing and Demultiplexing

- Host receives IP packets
  - Each IP header has source and destination IP address
  - Each Transport Layer header has source and destination port number

- Host uses IP addresses and port numbers to direct the message to appropriate socket
More on Ports

• Separate 16-bit port address space for UDP and TCP

• “Well known” ports (0-1023): everyone agrees which services run on these ports
  – e.g., ssh:22, http:80, https:443
  – helps client know server’s port

• Ephemeral ports (most 1024-65535): dynamically selected: as the source port for a client process
UDP: User Datagram Protocol

• Lightweight communication between processes
  – Avoid overhead and delays of ordered, reliable delivery

• UDP described in RFC 768 – (1980!)
  – Destination IP address and port to support demultiplexing
  – Optional error checking on the packet contents
    • (checksum field of 0 means “don’t verify checksum”) not in IPv6!
    • ((this idea of optional checksum is removed in IPv6))

```plaintext
<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
<tr>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>
```
Why a transport layer?

• IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  – Need a way to decide which packets go to which applications (mux/demux)

• IP provides a weak service model (*best-effort*)
  – Packets can be corrupted, delayed, dropped, reordered, duplicated
Principles of Reliable data transfer

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

- In a perfect world, reliable transport is easy

But the Internet default is best-effort

- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost
  - a packet is delayed (*why?*)
  - packets are reordered (*why?*)
  - a packet is duplicated (*why?*)
Principles of Reliable data transfer

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

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

(a) provided service
(b) service implementation
Reliable data transfer: getting started

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

**rdt_rcv()**: called by rdt to deliver data to upper

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

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

- State 1: when in this “state” next state uniquely determined by next event
- State 2

**Event causing state transition**

**Actions taken on state transition**
KR state machines – a note.

Beware
Kurose and Ross has a confusing/confused attitude to state-machines.
I’ve attempted to normalise the representation.

UPSHOT: these slides have differing information to the KR book (from which the RDT example is taken.) in KR “actions taken” appear wide-ranging, my interpretation is more specific/relevant.

---

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

**Relevant event causing state transition**

**Relevant action taken on state transition**
Rdt1.0: 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 read data from underlying channel
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 packet received is OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of NAK

- new mechanisms in *rdt2.0* (beyond *rdt1.0*):
  - error detection
  - receiver feedback: control msgs (ACK, NAK) receiver->sender
Dealing with Packet Corruption

Diagram showing the communication process with 'ack' and 'nack' signals between Sender and Receiver over time.
rdt2.0: FSM specification

**Sender**

- `udt_send(data)`
- `udt_send(packet)`
- `udt_rcv(reply) && isACK(reply)`
- `udt_send(packet)`
  - **Waiting for reply**

**Receiver**

- `udt_rcv(packet) && corrupt(packet)`
- `udt_send(NAK)`
- `udt_send(packet)`
  - **IDLE**

Note: the sender holds a copy of the packet being sent until the delivery is acknowledged.
rdt2.0: operation with no errors

- rdt_send(data)
- udt_send(packet)
- udt_rcv(reply) && isNAK(reply) => udt_send(packet)
- udt_rcv(reply) && isACK(reply)
- udt_send(ACK)
- udt_rcv(packet) && notcorrupt(packet) => rdt_rcv(data)
- udt_send(NAK)
- udt_rcv(packet) && corrupt(packet) => udt_send(NAK)
- IDLE
- Waiting for reply

Λ
rdt2.0: error scenario

- `rdt_send(data)`
- `udt_send(packet)`
- `udt_rcv(reply) && isNAK(reply)`
- `udt_send(packet)`
- `udt_rcv(packet) && corrupt(packet)`
- `udt_send(NAK)`
- `udt_rcv(packet) && notcorrupt(packet)`
- `rdt_rcv(data)`
- `udt_send(ACK)`
- `udt_rcv(reply) && isACK(reply)`
- `Lambda`
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 packet if ACK/NAK garbled
• sender adds sequence number to each packet
• receiver discards (doesn’t deliver) duplicate packet

stop and wait
Sender sends one packet, then waits for receiver response
Dealing with Packet Corruption

What if the ACK/NACK is corrupted?

Packet #1 or #2?

Data and ACK packets carry sequence numbers

This is packet #1
rdt2.1: sender, handles garbled ACK/NAKs

```
udt_send(packet)
rdt_send(data)
sequence=0
udt_send(packet)

udt_rcv(reply) &&
notcorrupt(reply)
&& isACK(reply)

udt rcv(reply) &&
corrupt(reply) ||
isNAK(reply)
udt send(packet)

udt send(packet)
udt rcv(reply) &&
notcorrupt(reply)
&& isACK(reply)
```

```
udt rcv(reply) &&
corrupt(reply) ||
isNAK(reply)
udt send(packet)
```

```
 Waiting For reply

 IDLE
```
rdt2.1: receiver, handles garbled ACK/NAKs

receive(packet) && corrupt(packet)
  ---------------
  udt_send(NAK)

receive(packet) && not corrupt(packet) && has_seq1(packet)
  ---------------
  udt_send(ACK)

receive(packet) && corrupt(packet)
  ---------------
  udt_send(NAK)

udt_rcv(packet) && not corrupt(packet)
  ---------------
  udt_send(ACK)
  rdt_rcv(data)

udt_rcv(packet) && corrupt(packet)
  ---------------
  udt_send(NAK)

udt_rcv(packet) && not corrupt(packet)
  ---------------
  udt_send(ACK)
  rdt_rcv(data)

Wait for 0 from below

Wait for 1 from below

udt_rcv(packet) && not corrupt(packet)
  ---------------
  udt_send(ACK)
  rdt_rcv(data)

receive(packet) && not corrupt(packet) && has_seq0(packet)
  ---------------
  udt_send(ACK)

receive(packet) && not corrupt(packet) && has_seq0(packet)
  ---------------
  udt_send(ACK)
**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 “current” pkt has a 0 or 1 sequence number

**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
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

sender FSM fragment

Wait for call 0 from above

sequence=0
udt_send(packet)

rdt_send(data)

Wait for ACK 0

rdt_rcv(reply) &&
  ( corrupt(reply) ||
    isACK1(reply) )
udt_send(packet)

receiver FSM fragment

Wait for 0 from below

receive(packet) && not corrupt(packet)
&& has_seq1(packet)

send(ACK1)
rdt_rcv(data)

udt_send(ACK1)

udt_rcv(packet) &&
  (corrupt(packet) ||
    has_seq1(packet))
rdt3.0: channels with errors and loss

**New assumption:** underlying channel can also lose packets (data or 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 use of seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer
rdt3.0 sender

The rdt3.0 sender is in an 'IDLE' state. It uses the following processes:

- **udt_send(packet)**
- **udt_send(data)**
- **udt_rcv(reply)**

The diagram shows the state transitions and conditions for each state:

- **IDLE state 0**:
  - **udt_rcv(reply)**
  - **timeout**

- **Wait for ACK0**:
  - **udt_rcv(reply)**
  - **timeout**

- **Wait for ACK1**:
  - **udt_rcv(reply)**
  - **udt_send(packet)**

- **IDLE state 1**:
  - **udt_rcv(reply)**
  - **udt_send(packet)**
Dealing with Packet Loss

Timer-driven loss detection
Set timer when packet is sent; retransmit on timeout
Dealing with Packet Loss

Sender

Time

Receiver

Timeout

1

1

P(1)

P(1)

ack(1)

P(2)

duplicate!

P(1)

1

Sender

Time

Receiver
Dealing with Packet Loss

Timer-driven retx. can lead to duplicates
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[
d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9 \text{bps}} = 8\text{microseconds}
\]

\[U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027\]

- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!
rdt3.0: stop-and-wait operation

![Diagram showing sender and receiver timelines for data transmission.]

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = \frac{L}{R}$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = \text{RTT} + \frac{L}{R}$

**Inefficient if $t \ll \text{RTT}$**

The throughput equation for the sender is:

$$U_{\text{sender}} = \frac{\frac{L}{R}}{\text{RTT} + \frac{L}{R}} = \frac{0.008}{30.008} = 0.00027$$
Pipelined (Packet-Window) protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
- buffering at sender and/or receiver

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation
A Sliding Packet Window

- **window** = set of adjacent sequence numbers
  - The size of the set is the window size; assume window size is $n$

- General idea: send up to $n$ packets at a time
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets “slides” on successful reception/acknowledgement
A Sliding Packet Window

- Let A be the last ack’d packet of sender without gap; then window of sender = \{A+1, A+2, ..., A+n\}

- Let B be the last received packet without gap by receiver, then window of receiver = \{B+1, ..., B+n\}

Sequence number

- Already ACK’d
- Sent but not ACK’d
- Cannot be sent

- Received and ACK’d
- Acceptable but not yet received
- Cannot be received
Acknowledgements w/ Sliding Window

• Two common options
  – cumulative ACKs: ACK carries next in-order sequence number that the receiver expects
Cumulative Acknowledgements (1)

- At receiver

- After receiving $B+1$, $B+2$

- Receiver sends $\text{ACK}(B_{\text{new}}+1)$
Cumulative Acknowledgements (2)

• At receiver

[Diagram showing received and ACK’d, acceptable but not yet received, cannot be received]

• After receiving B+4, B+5

[Diagram showing received and ACK’d, acceptable but not yet received, cannot be received]

• Receiver sends \text{ACK}(B+1)

How do we recover?
Go-Back-N (GBN)

- Sender transmits up to $n$ unacknowledged packets

- Receiver only accepts packets in order
  - discards out-of-order packets (i.e., packets other than $B+1$)

- Receiver uses cumulative acknowledgements
  - i.e., sequence# in ACK = next expected in-order sequence#

- Sender sets timer for 1st outstanding ack ($A+1$)
- If timeout, retransmit $A+1$, $A+2$, ..., $A+n$
Sliding Window with GBN

- Let $A$ be the last ack’ed packet of sender without gap; then window of sender = \{A+1, A+2, ..., A+n\}

- Let $B$ be the last received packet without gap by receiver, then window of receiver = \{B+1, ..., B+n\}

**Diagram:**

- Green blocks represent packets already ACK’ed.
- Light grey blocks represent sent but not ACK’ed packets.
- White blocks represent packets that cannot be sent.
- Dark grey blocks represent received and ACK’ed packets.
- Purple blocks represent acceptable but not yet received packets.
- Light grey blocks represent packets that cannot be received.
GBN Example w/o Errors

Window size = 3 packets

Sender Window

{1} 1
{1, 2} 2
{1, 2, 3} 3
{2, 3, 4} 4
{3, 4, 5} 5
{4, 5, 6} 6
. . .

Receiver Window

Sender

Time

Receiver
GBN Example with Errors

Window size = 3 packets

Timeout Packet 4
GBN Example with Errors - ALTERNATIVE

Window size = 3 packets

Timeout Packet 2
GBN: sender extended FSM

```c
rdt_send(data)
if (nextseqnum < base+N) {
    udt_send(packet[nextseqnum])
    nextseqnum++
}
else
    refuse_data(data)  // Block?
```

```
\[ \lambda \]
base = 1
nextseqnum = 1
```

```
\[ \lambda \]
udt_rcv(reply) \\
& & corrupt(reply)
```

```
\[ \lambda \]
udt_rcv(reply) \\
& & notcorrupt(reply)
```

```
\[ \lambda \]
base = getacknum(reply)+1
```

```
Wait
```

```
timeout
udt_send(packet[base])
udt_send(packet[base+1])
... 
udt_send(packet[nextseqnum-1])
```
ACK-only: always send an ACK for correctly-received packet with the highest \textit{in-order} seq #
- may generate duplicate ACKs
- need only remember $\text{expectedseqnum}$

\begin{itemize}
\item out-of-order packet:
\begin{itemize}
\item discard (don’t buffer) -> \textit{no receiver buffering}!
\item Re-ACK packet with highest in-order seq #
\end{itemize}
\end{itemize}
Acknowledgements w/ Sliding Window

• Two common options
  – cumulative ACKs: ACK carries next in-order sequence number the receiver expects
  – selective ACKs: ACK individually acknowledges correctly received packets

• Selective ACKs offer more precise information but require more complicated book-keeping

• Many variants that differ in implementation details
Selective Repeat (SR)

• Sender: transmit up to \( n \) unacknowledged packets

• Assume packet \( k \) is lost, \( k+1 \) is not

• Receiver: indicates packet \( k+1 \) correctly received

• Sender: retransmit only packet \( k \) on timeout

• Efficient in retransmissions but complex book-keeping
  – need a timer per packet
SR Example with Errors

Window size = 3 packets

Sender

Receiver
Observations

• With sliding windows, it is possible to fully utilize a link, provided the window size (n) is large enough. Throughput is \( \sim \frac{n}{\text{RTT}} \)
  – Stop & Wait is like \( n = 1 \).

• Sender has to buffer all unacknowledged packets, because they may require retransmission

• Receiver may be able to accept out-of-order packets, but only up to its buffer limits

• Implementation complexity depends on protocol details (GBN vs. SR)
Recap: components of a solution

• Checksums (for error detection)
• Timers (for loss detection)
• Acknowledgments
  – cumulative
  – selective
• Sequence numbers (duplicates, windows)
• Sliding Windows (for efficiency)

• Reliability protocols use the above to decide when and what to retransmit or acknowledge
What does TCP do?

Most of our previous tricks + a few differences

- Sequence numbers are byte offsets
- Sender and receiver maintain a sliding window
- Receiver sends cumulative acknowledgements (like GBN)
- Sender maintains a single retransmit timer
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmit
- Introduces timeout estimation algorithms
TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>Advertised window</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data

Used to mux and demux
What does TCP do?

Many of our previous ideas, but some key differences

• Checksum
## TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>Advertised window</td>
<td>Flags</td>
</tr>
<tr>
<td>Urgent pointer</td>
<td>Options (variable)</td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

Computed over header and data.
What does TCP do?

Many of our previous ideas, but some key differences

• Checksum

• Sequence numbers are byte offsets
TCP: Segments and Sequence Numbers
TCP “Stream of Bytes” Service…

Application @ Host A

Application @ Host B
… Provided Using TCP “Segments”

Segment sent when:
1. Segment full (Max Segment Size),
2. Not full, but times out
TCP Segment

- **IP packet**
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- **TCP packet**
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- **TCP segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU − (IP header) − (TCP header)
Sequence Numbers

Host A

ISN (initial sequence number)

$k$ bytes

Sequence number

$= 1^{st}$ byte in segment

$= \text{ISN} + k$
Sequence Numbers

ISN (initial sequence number)

Host A

Sequence number
= 1st byte in segment =
ISN + k

TCP Data
TCP HDR

Host B

ACK sequence number
= next expected byte
= seqno + length(data)
TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data

Starting byte offset of data carried in this segment
Sequence Numbers

Sequence number = 1st byte in segment = ISN + k

Host A

Sequence number
Acknowledgment

Data

Host B

TCP Data
TCP HDR

Host A - > B
DATA

Host B - > A
ACK

ACK sequence number = next expected byte = seqno + length(data)
TCP Sequences and ACKS

TCP is full duplex by default
  • two independently flows of sequence numbers

Sequence acknowledgement is given in terms of BYTES (not packets); the window is in terms of bytes.

number of packets = window size (bytes) / Segment Size

Servers and Clients are not Source and Destination

Piggybacking increases efficiency but many flows may only have data moving in one direction
What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
ACKing and Sequence Numbers

- **Sender sends packet**
  - Data starts with sequence number $X$
  - Packet contains $B$ bytes [$X, X+1, X+2, \ldots X+B-1$]

- **Upon receipt of packet, receiver sends an ACK**
  - If all data prior to $X$ already received:
    - ACK acknowledges $X+B$ (because that is next expected byte)
  - If highest in-order byte received is $Y$ s.t. $(Y+1) < X$
    - ACK acknowledges $Y+1$
    - Even if this has been ACKed before
Normal Pattern

• Sender: seqno=X, length=B
• Receiver: ACK=X+B
• Sender: seqno=X+B, length=B
• Receiver: ACK=X+2B
• Sender: seqno=X+2B, length=B

• Seqno of next packet is same as last ACK field
**TCP Header**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data

Acknowledgment gives seqno just beyond highest seqno received **in order** ("What Byte is Next")
What does TCP do?

Most of our previous tricks, but a few differences

• Checksum
• Sequence numbers are byte offsets
• Receiver sends cumulative acknowledgements (like GBN)
• Receivers can buffer out-of-sequence packets (like SR)
Loss with cumulative ACKs

• Sender sends packets with 100B and seqnos:
  – 100, 200, 300, 400, 500, 600, 700, 800, 900, ...

• Assume the fifth packet (seqno 500) is lost, but no others

• Stream of ACKs will be:
  – 200, 300, 400, 500, 500, 500, 500, ...
What does TCP do?

Most of our previous tricks, but a few differences

• Checksum
• Sequence numbers are byte offsets
• Receiver sends cumulative acknowledgements (like GBN)
• Receivers may not drop out-of-sequence packets (like SR)
• Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
Loss with cumulative ACKs

• “Duplicate ACKs” are a sign of an isolated loss
  – The lack of ACK progress means 500 hasn’t been delivered
  – Stream of ACKs means some packets are being delivered

• Therefore, could trigger resend upon receiving k duplicate ACKs
  • TCP uses k=3

• But response to loss is trickier....
Loss with cumulative ACKs

• Two choices:
  – Send missing packet and increase $W$ by the number of dup ACKs
  – Send missing packet, and wait for ACK to increase $W$

• Which should TCP do?
What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout
Retransmission Timeout

- If the sender hasn’t received an ACK by timeout, retransmit the first packet in the window

- How do we pick a timeout value?
Timing Illustration

Timeout too long $\rightarrow$ inefficient

Timeout too short $\rightarrow$ duplicate packets
Retransmission Timeout

• If haven’t received ack by timeout, retransmit the first packet in the window

• How to set timeout?
  – Too long: connection has low throughput
  – Too short: retransmit packet that was just delayed

• Solution: make timeout proportional to RTT

• But how do we measure RTT?
RTT Estimation

• Use exponential averaging of RTT samples

\[
\text{SampleRTT} = \text{AckRcvdTime} - \text{SendPacketTime}
\]

\[
\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}
\]

\[0 < \alpha \leq 1\]
Exponential Averaging Example

\[ \text{EstimatedRTT} = \alpha \cdot \text{EstimatedRTT} + (1 - \alpha) \cdot \text{SampleRTT} \]

Assume RTT is constant \( \rightarrow \text{SampleRTT} = \text{RTT} \)
Problem: Ambiguous Measurements

• How do we differentiate between the real ACK, and ACK of the retransmitted packet?
Karn/Partridge Algorithm

• Measure $SampleRTT$ only for original transmissions
  – Once a segment has been retransmitted, do not use it for any further measurements
• Computes EstimatedRTT using $\alpha = 0.875$

• Timeout value (RTO) = $2 \times$ EstimatedRTT
• Employs exponential backoff
  – Every time RTO timer expires, set RTO $\leftarrow 2 \cdot$ RTO
  – (Up to maximum $\geq 60$ sec)
  – Every time new measurement comes in (= successful original transmission), collapse RTO back to $2 \times$ EstimatedRTT
Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer

from Jacobson and Karels, SIGCOMM 1988
Jacobson/Karels Algorithm

• Problem: need to better capture variability in RTT
  – Directly measure deviation

• Deviation = | SampleRTT – EstimatedRTT |
• EstimatedDeviation: exponential average of Deviation

• RTO = EstimatedRTT + 4 x EstimatedDeviation
With Jacobson/Karels

Figure 5: Performance of an RFC793 retransmit timer

Figure 6: Performance of a Mean+Variance retransmit timer
What does TCP do?

Most of our previous ideas, but some key differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout
TCP Header: What’s left?

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Options (variable)</td>
</tr>
</tbody>
</table>

Data

“Must Be Zero” 6 bits reserved

Number of 4-byte words in TCP header; 5 = no options
TCP Header: What’s left?

- Source port
- Destination port
- Sequence number
- Acknowledgment
- Advertised window
- HdrLen
- Flags
- Urgent pointer
- Checksum
- Options (variable)

Used with **URG** flag to indicate urgent data (not discussed further)
TCP Header: What’s left?

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data
TCP Connection Establishment and Initial Sequence Numbers
Initial Sequence Number (ISN)

- Sequence number for the very first byte
- Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
- TCP therefore requires changing ISN
- Hosts exchange ISNs when they establish a connection
Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

Each host tells its ISN to the other host.
## TCP Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Source port</td>
</tr>
<tr>
<td>Destination port</td>
<td>Destination port</td>
</tr>
<tr>
<td>Sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>Advertised window</td>
<td>Advertised window</td>
</tr>
<tr>
<td>HdrLen</td>
<td>HdrLen</td>
</tr>
<tr>
<td>Flags</td>
<td>Flags: SYN, ACK, FIN, RST, PSH, URG</td>
</tr>
<tr>
<td>Checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urgent pointer</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td>Options (variable)</td>
</tr>
<tr>
<td>Data</td>
<td>Data</td>
</tr>
</tbody>
</table>

**Flags:**
- SYN
- ACK
- FIN
- RST
- PSH
- URG

**Checksum and Urgent Pointer:**
- Checksum
- Urgent pointer
Step 1: A’s Initial SYN Packet

<table>
<thead>
<tr>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A’s Initial Sequence Number

(Irrelevant since ACK not set)

<table>
<thead>
<tr>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[Advertised window]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum</td>
</tr>
<tr>
<td>Urgent pointer</td>
</tr>
</tbody>
</table>

Options (variable)

A tells B it wants to open a connection…

Flags:
- SYN
- ACK
- FIN
- RST
- PSH
- URG
Step 2: B’s SYN-ACK Packet

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FIN</td>
<td>RST</td>
</tr>
<tr>
<td></td>
<td>PSH</td>
<td>URG</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B’s port</th>
<th>A’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B’s Initial Sequence Number</th>
<th>ACK = A’s ISN plus 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Advertised window</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data
Step 3: A’s ACK of the SYN-ACK

A tells B it’s likewise okay to start sending

\[ \text{... upon receiving this packet, B can start sending data...} \]
Timing Diagram: 3-Way Handshaking

Client (initiator)

Active
Open

connect()

SYN, SeqNum = x

SYN + ACK, SeqNum = y, Ack = x + 1

ACK, Ack = y + 1

Passive
Open

Server

listen()
What if the SYN Packet Gets Lost?

• Suppose the SYN packet gets lost
  – Packet is lost inside the network, or:
  – Server discards the packet (e.g., it’s too busy)

• Eventually, no SYN-ACK arrives
  – Sender sets a timer and waits for the SYN-ACK
  – … and retransmits the SYN if needed

• How should the TCP sender set the timer?
  – Sender has no idea how far away the receiver is
  – Hard to guess a reasonable length of time to wait
  – SHOULD (RFCs 1122 & 2988) use default of 3 seconds
    • Some implementations instead use 6 seconds
Tearing Down the Connection
Normal Termination, One Side At A Time

- Finish (FIN) to close and receive remaining bytes
  - FIN occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A’s side of the connection, but not B’s
  - Until B likewise sends a FIN
  - Which A then acks

Connection now **closed**

Connection now **half-closed**

**TIME_WAIT**: Avoid reincarnation
B will retransmit FIN if ACK is lost
Normal Termination, Both Together

• Same as before, but B sets **FIN** with their ack of A’s **FIN**

**TIME_WAIT:**
Avoid reincarnation
Can retransmit FIN ACK if ACK lost

Connection now closed
Abrupt Termination

• A sends a RESET (RST) to B
  – E.g., because application process on A crashed
• That’s it
  – B does not ack the RST
  – Thus, RST is not delivered reliably
  – And: any data in flight is lost
  – But: if B sends anything more, will elicit another RST
TCP State Transitions

Data, ACK exchanges are in here
An Simpler View of the Client Side

- **CLOSED**
  - SYN (Send)
- **SYN_SENT**
  - Rcv. SYN+ACK, Send ACK
- **ESTABLISHED**
  - Send FIN
- **FIN_WAIT1**
  - Rcv. ACK, Send Nothing
- **FIN_WAIT2**
  - Rcv. FIN, Send ACK
- **TIME_WAIT**
  - Send Nothing
## TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
</table>

| Data |
• What does TCP do?
  – ARQ windowing, set-up, tear-down
• Flow Control in TCP
Recap: Sliding Window (so far)

• Both sender & receiver maintain a **window**

• **Left edge** of window:
  – Sender: beginning of *unacknowledged* data
  – Receiver: beginning of *undelivered* data

• **Right edge**: Left edge + *constant*
  – constant only limited by buffer size in the transport layer
Sliding Window at Sender (so far)

Sending process

TCP

Previously ACKed bytes

First unACKed byte

Last byte written

Buffer size (B)

Last byte can send

First unACKed byte

Previously ACKed bytes

Last byte written
Sliding Window at Receiver (so far)

Receiving process

Last byte read

Buffer size (B)

Received and ACKed

Next byte needed (1\textsuperscript{st} byte not received)

Last byte received

Sender might overrun the receiver’s buffer
Solution: Advertised Window (Flow Control)

- Receiver uses an “Advertised Window” (W) to prevent sender from overflowing its window
  - Receiver indicates value of W in ACKs
  - Sender limits number of bytes it can have in flight <= W
Sliding Window at Receiver

Next byte needed
(1st byte not received)

Last byte received

W = B - (LastByteReceived - LastByteRead)
Sliding Window at Sender (so far)

Sending process

TCP

W

First unACKed byte

Last byte written

Last byte can send
Sliding Window w/ Flow Control

- Sender: window *advances* when new data ack’d
- Receiver: window advances as receiving process *consumes* data
- Receiver *advertises* to the sender where the receiver window currently ends (“righthand edge”)
  - Sender agrees not to exceed this amount
Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec

- Receiver only advertises more space when it has consumed old arriving data

- In original TCP design, that was the sole protocol mechanism controlling sender’s rate

- What’s missing?
TCP

• The concepts underlying TCP are simple
  – acknowledgments (feedback)
  – timers
  – sliding windows
  – buffer management
  – sequence numbers
• What does TCP do?
  – ARQ windowing, set-up, tear-down
• Flow Control in TCP
• Congestion Control in TCP
We have seen:

– **Flow control**: adjusting the sending rate to keep from overwhelming a slow *receiver*

Now let's attend…

– **Congestion control**: adjusting the sending rate to keep from overloading the *network*
Statistical Multiplexing → Congestion

- If two packets arrive at the same time
  - A router can only transmit one
  - … and either buffers or drops the other
- If many packets arrive in a short period of time
  - The router cannot keep up with the arriving traffic
  - … delays traffic, and the buffer may eventually overflow
- Internet traffic is **bursty**
Congestion is undesirable

Typical queuing system with bursty arrivals

Must balance utilization versus delay and loss
Who Takes Care of Congestion?

• Network? End hosts? Both?

• TCP’s approach:
  – **End hosts** adjust sending rate
  – Based on *implicit feedback* from network

• Not the only approach
  – A consequence of history rather than planning
Some History: TCP in the 1980s

• Sending rate only limited by flow control
  – Packet drops $\rightarrow$ senders (repeatedly!) retransmit a full window’s worth of packets

• Led to “congestion collapse” starting Oct. 1986
  – Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec

• “Fixed” by Van Jacobson’s development of TCP’s congestion control (CC) algorithms
Jacobson’s Approach

• Extend TCP’s existing window-based protocol but adapt the window size in response to congestion
  – required no upgrades to routers or applications!
  – patch of a few lines of code to TCP implementations

• A pragmatic and effective solution
  – but many other approaches exist

• Extensively improved on since
  – topic now sees less activity in ISP contexts
  – but is making a comeback in datacenter environments
Three Issues to Consider

- Discovering the available (bottleneck) bandwidth
- Adjusting to variations in bandwidth
- Sharing bandwidth between flows
Abstract View

- Ignore internal structure of router and model it as having a single queue for a particular input-output pair
Discovering available bandwidth

- Pick sending rate to match bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be gigabit link, could be a modem
Adjusting to variations in bandwidth

- Adjust rate to match *instantaneous* bandwidth
  - Assuming you have rough idea of bandwidth
Multiple flows and sharing bandwidth

Two Issues:
• Adjust total sending rate to match bandwidth
• Allocation of bandwidth between flows
Reality

Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics.
View from a single flow

• Knee – point after which
  – Throughput increases slowly
  – Delay increases fast

• Cliff – point after which
  – Throughput starts to drop to zero (congestion collapse)
  – Delay approaches infinity
General Approaches

(0) Send without care
   – Many packet drops
General Approaches

(0) Send without care

(1) Reservations
   – Pre-arrange bandwidth allocations
   – Requires negotiation before sending packets
   – Low utilization
General Approaches

(0) Send without care
(1) Reservations
(2) Pricing
   – Don’t drop packets for the high-bidders
   – Requires payment model
General Approaches

(0) Send without care
(1) Reservations
(2) Pricing
(3) Dynamic Adjustment
  – Hosts probe network; infer level of congestion; adjust
  – Network reports congestion level to hosts; hosts adjust
  – Combinations of the above
  – Simple to implement but suboptimal, messy dynamics
General Approaches

(0) Send without care
(1) Reservations
(2) Pricing
(3) Dynamic Adjustment

All three techniques have their place
- Generality of dynamic adjustment has proven powerful
- Doesn’t presume business model, traffic characteristics, application requirements; does assume good citizenship
TCP’s Approach in a Nutshell

• TCP connection has window
  – Controls number of packets in flight

• Sending rate: \(\sim\)Window/RTT

• Vary window size to control sending rate
Windows, Buffers, and TCP
Windows, Buffers, and TCP

• TCP connection has a window
  – Controls number of packets in flight;
    filling a channel to improve throughput, and
    vary window size to control sending rate

• Buffers adapt mis-matched channels
  – Buffers smooth bursts
  – Adapt (re-time) arrivals for multiplexing
Windows, Buffers, and TCP

Buffers & TCP can make link utilization 100%

but

Buffers add delay, \textbf{variable} delay
Sizing Buffers in Routers

– Packet loss
  • Queue overload, and subsequent packet loss

– End-to-end delay
  • Transmission, propagation, and queueing delay
  • The only variable part is queueing delay

– Router architecture
  • Board space, power consumption, and cost
  • On chip buffers: higher density, higher capacity
Buffer Sizing Story

# of packets
Intuition
Assume
Evidence

Rule-of-thumb

$2T \times C$

1,000,000
TCP Sawtooth
Single TCP Flow, 100% Utilization
Simulation, Emulation
Continuous ARQ (TCP) adapting to congestion

Only $W$ packets may be outstanding

Rule for adjusting $W$
- If an ACK is received: $W \leftarrow W + 1/W$
- If a packet is lost: $W \leftarrow W/2$

$W = 1$

Util = 0%
Continuous ARQ (TCP) adapting to congestion

Only $W$ packets may be outstanding

- If an ACK is received: $W \leftarrow W + 1/W$
- If a packet is lost: $W \leftarrow W/2$

Rule for adjusting $W$
Rule-of-thumb – Intuition

Only $W$ packets may be outstanding

Rule for adjusting $W$
- If an ACK is received: $W \leftarrow W + 1/W$
- If a packet is lost: $W \leftarrow W/2$

Source \[\rightarrow\] Window size \[\rightarrow\] Dest

$W_{\text{max}}$ $\frac{W_{\text{max}}}{2}$ $2T \times C$

t

$2T \times C$
Buffers in Routers
So how large should the buffers be?

Buffer size matters

– Packet loss
  • Queue overload, and subsequent packet loss

– End-to-end delay
  • Transmission, propagation, and queueing delay
  • The only variable part is queueing delay
## Buffer Sizing Story

<table>
<thead>
<tr>
<th># of packets</th>
<th>Rule-of-thumb</th>
<th>Intuition</th>
<th>Assume</th>
<th>Evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>$2T \times C$</td>
<td>TCP Sawtooth</td>
<td>Single TCP Flow, 100% Utilization</td>
<td>Simulation, Emulation</td>
</tr>
<tr>
<td>TCP Sawtooth</td>
<td>10,000</td>
<td>Sawtooth Smoothing</td>
<td>Many Flows, 100% Utilization</td>
<td>Simulations, Test-bed and Real Network Experiments</td>
</tr>
</tbody>
</table>

$2T \times C \div \sqrt{n}$
Buffers in Routers
So how large should the buffers be?

Buffer size matters

– Packet loss
  • Queue overload, and subsequent packet loss

– End-to-end delay
  • Transmission, propagation, and queueing delay
  • The only variable part is queueing delay

– Router architecture
  • Board space, power consumption, and cost
  • On chip buffers: higher density, higher capacity
Small Buffers – Intuition

Synchronized Flows
• Aggregate window has same dynamics
• Therefore buffer occupancy has same dynamics
• Rule-of-thumb still holds.

Many TCP Flows
• Independent, desynchronized
• Central limit theorem says the aggregate becomes Gaussian
• Variance (buffer size) decreases as N increases
What size do we make the buffer? Well it depends…


Rule-of-thumb: $2T \times C$

Simulations, Test-bed and Real Network Experiments

Evidence

Intuition

Assume

# of packets

Buffer Sizing Story

$2T \times C \frac{10,000}{\sqrt{n}}$

$O(\log W)$
TCP’s Approach in a Nutshell

• TCP connection has window
  – Controls number of packets in flight

• Sending rate: ~Window/RTT

• Vary window size to control sending rate
All These Windows...

• Congestion Window: \( \text{CWND} \)
  – How many bytes can be sent without overflowing routers
  – Computed by the sender using congestion control algorithm

• Flow control window: \( \text{AdvertisedWindow (RWND)} \)
  – How many bytes can be sent without overflowing receiver’s buffers
  – Determined by the receiver and reported to the sender

• Sender-side window = \( \text{minimum}\{\text{CWND, RWND}\} \)
  • Assume for this material that \( \text{RWND} \gg \text{CWND} \)
Note

• This lecture will talk about CWND in units of MSS
  – (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
  – This is only for pedagogical purposes

• In reality this is a LIE: Real implementations maintain CWND in bytes
Two Basic Questions

• How does the sender detect congestion?

• How does the sender adjust its sending rate?
  – To address three issues
    • Finding available bottleneck bandwidth
    • Adjusting to bandwidth variations
    • Sharing bandwidth
Detecting Congestion

• Packet delays
  – Tricky: noisy signal (delay often varies considerably)

• Router tell end-hosts they’re congested

• Packet loss
  – Fail-safe signal that TCP already has to detect
  – Complication: non-congestive loss (checksum errors)

• Two indicators of packet loss
  – No ACK after certain time interval: timeout
  – Multiple duplicate ACKs
Not All Losses the Same

• Duplicate ACKs: isolated loss
  – Still getting ACKs

• Timeout: much more serious
  – Not enough packets in progress to trigger duplicate-acks, OR
  – Suffered several losses

• We will adjust rate differently for each case
Rate Adjustment

• Basic structure:
  – Upon receipt of ACK (of new data): increase rate
  – Upon detection of loss: decrease rate

• How we increase/decrease the rate depends on the phase of congestion control we’re in:
  – Discovering available bottleneck bandwidth vs.
  – Adjusting to bandwidth variations
Bandwidth Discovery with Slow Start

• Goal: estimate available bandwidth
  – start slow (for safety)
  – but ramp up quickly (for efficiency)

• Consider
  – RTT = 100ms, MSS=1000bytes
  – Window size to fill 1Mbps of BW = 12.5 packets
  – Window size to fill 1Gbps = 12,500 packets
  – Either is possible!
“Slow Start” Phase

• Sender starts at a slow rate but increases **exponentially** until first loss

• Start with a small congestion window
  – Initially, CWND = 1
  – So, initial sending rate is MSS/RTT

• Double the CWND for each RTT with no loss
Slow Start in Action

- For each RTT: double CWND
- Simpler implementation: for each ACK, CWND += 1
Adjusting to Varying Bandwidth

- Slow start gave an estimate of available bandwidth

- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (rate decrease)

- TCP uses: “Additive Increase Multiplicative Decrease” (AIMD)
  - We’ll see why shortly…
AIMD

• Additive increase
  – Window grows by one MSS for every RTT with no loss
  – For each successful RTT, $\text{CWND} = \text{CWND} + 1$
  – Simple implementation:
    • for each ACK, $\text{CWND} = \text{CWND} + 1/\text{CWND}$

• Multiplicative decrease
  – On loss of packet, divide congestion window in half
  – On loss, $\text{CWND} = \text{CWND}/2$
Leads to the TCP “Sawtooth”

Window

Loss

Exponential “slow start”
Slow-Start vs. AIMD

• When does a sender stop Slow-Start and start Additive Increase?

• Introduce a “slow start threshold” (ssthresh)
  – Initialized to a large value
  – On timeout, ssthresh = CWND/2

• When CWND = ssthresh, sender switches from slow-start to AIMD-style increase
• What does TCP do?
  – ARQ windowing, set-up, tear-down
• Flow Control in TCP
• Congestion Control in TCP
  – AIMD
• What does TCP do?
  – ARQ windowing, set-up, tear-down
• Flow Control in TCP
• Congestion Control in TCP
  – AIMD, Fast-Recovery
One Final Phase: Fast Recovery

• The problem: congestion avoidance too slow in recovering from an isolated loss
Example (in units of MSS, not bytes)

• Consider a TCP connection with:
  – CWND=10 packets
  – Last ACK was for packet # 101
    • i.e., receiver expecting next packet to have seq. no. 101

• 10 packets [101, 102, 103,..., 110] are in flight
  – Packet 101 is dropped
  – What ACKs do they generate?
  – And how does the sender respond?
The problem – A timeline

• ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
• ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
• ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
• RETRANSMIT 101 ssthresh=5 cwnd= 5
• ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
• ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
• ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
• ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
• ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
• ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
• ACK 111 (due to 101) ➡️ only now can we transmit new packets
• Plus no packets in flight so ACK “clocking” (to increase CWND) stalls for another RTT
Solution: Fast Recovery

Idea: Grant the sender temporary “credit” for each dupACK so as to keep packets in flight

• If dupACKcount = 3
  – ssthresh = cwnd/2
  – cwnd = ssthresh + 3

• While in fast recovery
  – cwnd = cwnd + 1 for each additional duplicate ACK

• Exit fast recovery after receiving new ACK
  – set cwnd = ssthresh
Example

• Consider a TCP connection with:
  – CWND=10 packets
  – Last ACK was for packet # 101
    • i.e., receiver expecting next packet to have seq. no. 101

• 10 packets [101, 102, 103,..., 110] are in flight
  – Packet 101 is dropped
Timeline

- ACK 101 (due to 102) cwnd=10  dup#1
- ACK 101 (due to 103) cwnd=10  dup#2
- ACK 101 (due to 104) cwnd=10  dup#3
- REXMIT 101 ssthresh=5  cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ➔ exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5 ➔ back in congestion avoidance
Putting it all together:
The TCP State Machine (partial)

- How are ssthresh, CWND and dupACKcount updated for each event that causes a state transition?
TCP Flavors

- TCP-Tahoe
  - $\text{cwnd} = 1$ on triple dupACK
- TCP-Reno
  - $\text{cwnd} = 1$ on timeout
  - $\text{cwnd} = \text{cwnd}/2$ on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements
• What does TCP do?
  – ARQ windowing, set-up, tear-down
• Flow Control in TCP
• Congestion Control in TCP
  – AIMD, Fast-Recovery, Throughput
TCP Flavors

• TCP-Tahoe
  – CWND =1 on triple dupACK
• TCP-Reno
  – CWND =1 on timeout
  – CWND = CWND/2 on triple dupack
• TCP-newReno
  – TCP-Reno + improved fast recovery
• TCP-SACK
  – incorporates selective acknowledgements

Our default assumption
Interoperability

• How can all these algorithms coexist? Don’t we need a single, uniform standard?

• What happens if I’m using Reno and you are using Tahoe, and we try to communicate?
TCP Throughput Equation
A Simple Model for TCP Throughput

\[ cwnd \]

Loss

\[ \frac{1}{2} W_{\text{max}} \text{ RTTs between drops} \]

\[ W_{\text{max}} \]

\[ \frac{W_{\text{max}}}{2} \]

\[ \text{Avg. } \frac{3}{4} W_{\text{max}} \text{ packets per RTTs} \]

\[ RTT \]
A Simple Model for TCP Throughput

Packet drop rate, \( p = 1 / A \), where \( A = \frac{3}{8} W_{\text{max}}^2 \)

Throughput, \( B = \frac{A}{\left(\frac{W_{\text{max}}}{2}\right) \text{RTT}} = \sqrt{\frac{3}{2}} \frac{1}{\text{RTT} \sqrt{p}} \)
Implications (1): Different RTTs

\[
\text{Throughput} = \sqrt[2]{\frac{3}{2}} \cdot \frac{1}{\text{RTT} \sqrt{p}}
\]

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!

![Diagram of network with nodes A1, A2, B1, B2 and bottleneck link 100ms, 200ms]
Implications (2): High Speed TCP

\[
\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{\text{RTT} \sqrt{p}}
\]

• Assume RTT = 100ms, MSS=1500bytes

• What value of \( p \) is required to reach 100Gbps throughput
  – \( \sim 2 \times 10^{-12} \)

• How long between drops?
  – \( \sim 16.6 \) hours

• How much data has been sent in this time?
  – \( \sim 6 \) petabits

• These are not practical numbers!
Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
  - A proposed standard [Floyd’03]: once speed is past some threshold, change equation to $p^{-0.8}$ rather than $p^{-0.5}$
  - Let the additive constant in AIMD depend on CWND

• Other approaches?
  - Multiple simultaneous connections (*hacky* but works today)
  - Router-assisted approaches (will see shortly)
Implications (3): *Rate*-based CC

TCP throughput is “choppy”
- repeated swings between $W/2$ to $W$

Some apps would prefer sending at a steady rate
- e.g., streaming apps

A solution: “Equation-Based Congestion Control”
- ditch TCP’s increase/decrease rules and just follow the equation
- measure drop percentage $p$, and set rate accordingly

Following the TCP equation ensures we’re “TCP friendly”
- i.e., use no more than TCP does in similar setting

\[
\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{RTT \sqrt{p}}
\]
TCP Cubic V TCP Reno

CUBIC Probing

Faster Increase
Sharp decrease

RENO Probing

Slow Increase
Sharp decrease
New world of fairness....

CUBIC window curve

Time (second)

CHNIC (packets)
None of the 12 congestion algorithms is perfect. Our BBR is finally integrated in Linux 4.9 but it is not the best in all the cases. Let’s code the ultimate one to rule them all. Now we have 13 different algorithms!
Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Could fix many of these with some help from routers!
Router-Assisted Congestion Control

• Three tasks for CC:
  – Isolation/fairness
  – Adjustment*
  – Detecting congestion

* This may be *automatic* eg loss-response of TCP
How can routers ensure each flow gets its “fair share”? 
Fairness: General Approach

• Routers classify packets into “flows”
  – (For now) flows are packets between same source/destination

• Each flow has its own FIFO queue in router

• Router services flows in a fair fashion
  – When line becomes free, take packet from next flow in a fair order

• What does “fair” mean exactly?
Max-Min Fairness

• Given set of bandwidth demands \( r_i \) and total bandwidth \( C \), max-min bandwidth allocations are:

\[
a_i = \min(f, r_i)
\]

where \( f \) is the unique value such that \( \sum(a_i) = C \).
Example

- $C = 10; \ r_1 = 8, \ r_2 = 6, \ r_3 = 2; \ N = 3$
- $C/3 = 3.33 \rightarrow$
  - Can service all of $r_3$
  - Remove $r_3$ from the accounting: $C = C - r_3 = 8; \ N = 2$
- $C/2 = 4 \rightarrow$
  - Can’t service all of $r_1$ or $r_2$
  - So hold them to the remaining fair share: $f = 4$

\[
\begin{align*}
f = 4: \\
\min(8, 4) &= 4 \\
\min(6, 4) &= 4 \\
\min(2, 4) &= 2
\end{align*}
\]
Max-Min Fairness

• Given set of bandwidth demands $r_i$ and total bandwidth $C$, max-min bandwidth allocations are:
  
  $$a_i = \min(f, r_i)$$

• where $f$ is the unique value such that $\sum(a_i) = C$

• Property:
  – If you don’t get full demand, no one gets more than you

• This is what round-robin service gives if all packets are the same size
How do we deal with packets of different sizes?

• Mental model: Bit-by-bit round robin ("fluid flow")

• Can you do this in practice?

• No, packets cannot be preempted

• But we can approximate it
  – This is what "fair queuing" routers do
Fair Queuing (FQ)

• For each packet, compute the time at which the last bit of a packet would have left the router if flows are served bit-by-bit

• Then serve packets in the increasing order of their deadlines
Fair Queuing (FQ)

• Think of it as an implementation of round-robin generalized to the case where not all packets are equal sized

• **Weighted** fair queuing (WFQ): assign different flows different shares

• Today, some form of WFQ implemented in almost all routers
  – Not the case in the 1980-90s, when CC was being developed
  – Mostly used to isolate traffic at larger granularities (e.g., per-prefix)
FQ vs. FIFO

• FQ advantages:
  – Isolation: cheating flows don’t benefit
  – Bandwidth share does not depend on RTT
  – Flows can pick any rate adjustment scheme they want

• Disadvantages:
  – More complex than FIFO: per flow queue/state, additional per-packet book-keeping
FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion

Blue and Green get 0.5Gbps; any excess will be dropped.

Will drop an additional 400Mbps from the green flow.

If the green flow doesn’t drop its sending rate to 100Mbps, we’re wasting 400Mbps that could be usefully given to the blue flow.
FQ in the big picture

• FQ does not eliminate congestion → it just manages the congestion
  – robust to cheating, variations in RTT, details of delay, reordering, retransmission, etc.

• But congestion (and packet drops) still occurs

• And we still want end-hosts to discover/adapt to their fair share!

• What would the end-to-end argument say w.r.t. congestion control?
Fairness is a controversial goal

• What if you have 8 flows, and I have 4?
  – Why should you get twice the bandwidth

• What if your flow goes over 4 congested hops, and mine only goes over 1?
  – Why shouldn’t you be penalized for using more scarce bandwidth?

• And what is a flow anyway?
  – TCP connection
  – Source-Destination pair?
  – Source?
Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for when routers set the bit
  - tradeoff between (link) utilization and (packet) delay
- Congestion semantics can be exactly like that of drop
  - i.e., endhost reacts as though it saw a drop

- Advantages:
  - Don’t confuse corruption with congestion; recovery w/ rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - defined as extension to TCP/IP in RFC 3168 (uses diffserv bits in the IP header)
TCP in detail

• What does TCP do?
  – ARQ windowing, set-up, tear-down

• Flow Control in TCP

• Congestion Control in TCP
  – AIMD, Fast-Recovery, Throughput

• Limitations of TCP Congestion Control

• Router-assisted Congestion Control (eg ECN)
Transport Recap

A “big bag”:
- Multiplexing, reliability, error-detection, error-recovery, flow and congestion control, ….

- **UDP:**
  - Minimalist - multiplexing and error detection

- **TCP:**
  - somewhat hacky
  - but practical/deployable
  - good enough to have raised the bar for the deployment of new, more optimal, approaches
  - though the needs of datacenters might change the status quos

- **Beyond TCP (discussed in Topic 6):**
  - QUIC / application-aware transport layers