Mobile Ad Hoc Networking

Transport Layer

Issues

Contents:
- overview principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- TCP Performance analysis
But first, a general overview of networks (and the Internet)

- Telecommunication networks
  - Circuit-switched networks
    - FDM
  - Packet-switched networks
    - Networks with VCs
    - Datagram Networks

What Is the Internet?

- A network of networks, joining many government, university and private computers together and providing an infrastructure for the use of E-mail, bulletin boards, file archives, hypertext documents, databases and other computational resources.
- The vast collection of computer networks which form and act as a single huge network for transport of data and messages across distances which can be anywhere from the same office to anywhere in the world.

Written by William F. Slater, III
1996
President of the Chicago Chapter of the Internet Society

Copyright 2002, William F. Slater, III, Chicago, IL, USA
What is the Internet?

- The largest network of networks in the world.
- Uses TCP/IP protocols and packet switching.
- Runs on any communications substrate.

From Dr. Vinton Cerf, Co-Creator of TCP/IP

Brief History of the Internet

- 1968 - DARPA (Defense Advanced Research Projects Agency) contracts with BBN (Bolt, Beranek & Newman) to create ARPAnet
- 1970 - First five nodes:
  - UCLA
  - Stanford
  - UC Santa Barbara
  - U of Utah, and
  - BBN
- 1974 - TCP specification by Vint Cerf
- 1984 - On January 1, the Internet with its 1000 hosts converts en masse to using TCP/IP for its messaging
A Brief Summary of the Evolution of the Internet

1945
Mathematical Theory of Communication

1948
Memex Conceived

1958
Silicon Chip

1962
First Vast Computer Network Envisioned

1964
Packet Switching Invented

1969
Hypertext Invented

1972
TCP/IP Created

1983
WWW Created

1984
TCP/IP Goes Commercial

1993
Mosaic Created

1995
Age of Commerce Begins

1995
Review: Transport Layer
## From Simple, But Significant Ideas Bigger Ones Grow 1940s to 1969

<table>
<thead>
<tr>
<th>1945</th>
<th>1969</th>
</tr>
</thead>
<tbody>
<tr>
<td>We will prove that packet switching works over a WAN.</td>
<td>Hypertext can be used to allow rapid access to text data</td>
</tr>
<tr>
<td>We can do it cheaply by using Digital circuits etched in silicon.</td>
<td>Packet switching can be used to send digitized data through computer networks</td>
</tr>
<tr>
<td>We can accomplish a lot by having a vast network of computers to use for accessing information and exchanging ideas.</td>
<td>We can do it reliably with “bits”, sending and receiving data</td>
</tr>
<tr>
<td>We can access information using electronic computers.</td>
<td></td>
</tr>
</tbody>
</table>

## From Simple, But Significant Ideas Bigger Ones Grow 1970s to 1995

<table>
<thead>
<tr>
<th>1970</th>
<th>1995</th>
</tr>
</thead>
<tbody>
<tr>
<td>Great efficiencies can be accomplished if we use The Internet and the World Wide Web to conduct business.</td>
<td>The World Wide Web is easier to use if we have a browser that To browser web pages, running in a graphical user interface context.</td>
</tr>
<tr>
<td>Computers connected via the Internet can be used more easily if hypertext links are enabled using HTML and URLs: it’s called World Wide Web</td>
<td>The ARPANET needs to convert to a standard protocol and be renamed to The Internet</td>
</tr>
<tr>
<td>We need a protocol for Efficient and Reliable transmission of Packets over a WAN: TCP/IP</td>
<td>Ideas from 1940s to 1969</td>
</tr>
</tbody>
</table>
The Creation of the Internet

The creation of the Internet solved the following challenges:
- Basically inventing digital networking as we know it
- Survivability of an infrastructure to send / receive high-speed electronic messages
- Reliability of computer messaging

Internet Pioneers

<table>
<thead>
<tr>
<th>Name</th>
<th>Contribution</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vannevar Bush</td>
<td>(APARNet)</td>
<td>Claude Shannon</td>
<td>(Information theory)</td>
</tr>
<tr>
<td>Leonard Kleinrock</td>
<td>(Packet switching)</td>
<td>Ted Nelson</td>
<td>(Hypertext)</td>
</tr>
<tr>
<td>Vinton Cerf</td>
<td>(TCP/IP)</td>
<td>Robert Kahn</td>
<td>(TCP/IP)</td>
</tr>
<tr>
<td>Mark Andreesen</td>
<td>(Mosaic/Netscape)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Internet was not known as "The Internet" until January 1984, at which time there were 1000 hosts that were all converted over to using TCP/IP.

ISO 7-layer reference model
Internet protocol stack

- **application**: supporting network applications
  - FTP, SMTP, HTTP
- **transport**: host-host data transfer
  - TCP, UDP
- **network**: routing of datagrams from source to destination
  - IP, routing protocols e.g. OSPF, BGP
- **link**: data transfer between neighboring network elements
  - PPP, Ethernet
- **physical**: bits “on the wire”

Internet Standardization Process

- All standards of the Internet are published as RFC (Request for Comments)
  - but not all RFCs are Internet Standards!
  - available: http://www.ietf.org
  - Till now: RFC4333
- A typical (but not the only) way of standardization:
  - Internet draft
  - RFC
  - Proposed standard
  - Draft standard (requires 2 working implementations)
  - Internet standard (declared by Internet Architecture Board)
Outline

- 1. Transport-layer services
- 2. Multiplexing and demultiplexing
- 3. Connectionless transport: UDP
- 4. Principles of reliable data transfer
- 5. Connection-oriented transport: TCP
- 6. TCP congestion control
- 7. TCP fairness and delay performance

Transport layer - the other side of the door

- API: (1) choose transport protocol; (2) set parameters
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

Transport vs. network layer

- **network layer**: logical communication between hosts
  - Point-to-point
- **transport layer**: logical communication between processes
  - relies on and enhances, network layer services
  - also called “End-to-End”

---

Outline

- 1. Transport-layer services
- 2. Multiplexing and demultiplexing
- 3. Connectionless transport: UDP
- 4. Principles of reliable data transfer
- 5. Connection-oriented transport: TCP
- 6. TCP congestion control
- 7. TCP fairness and delay performance

How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

<table>
<thead>
<tr>
<th>32 bits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>dest port #</td>
</tr>
<tr>
<td>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data</td>
<td>(message)</td>
</tr>
</tbody>
</table>

(source: IP address, destination IP address, source port number, destination port number)
Connection-oriented demux

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

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

Q:
- Why use 4-tuple?

Examples:
- 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 a different socket for each request
UDP: User Datagram Protocol [RFC 768]

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

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS - why?
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

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>
Outline

1. Transport-layer services
2. Multiplexing and demultiplexing
3. Connectionless transport: UDP
4. Principles of reliable data transfer
5. Connection-oriented transport: TCP
6. TCP congestion control
7. TCP fairness and delay performance

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)
Reliable data transfer: getting started

We’ll:
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- What is unreliability?
  - Bit error
  - Packet loss - congestion
  - Delay - too long
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

```
Wait for
call from
above
```

```
rdt_send(data)
packet = make_pkt(data)
udt_send(packet)
```

```
.Wait for
call from
below
```

```
wait for rcv
```

```
extract (packet, data)
deliver_data(data)
```

sender

receiver

Rdt2.0: channel with bit errors

```
wait for
ACK or NAK
```

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

```
wait for
NAK
```

```
rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
```

```
udt_send(sndpkt)
```

receiver

```
wait for
NAK
```

```
udt_send(NAK)
```

sender

```
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
```

Review: Transport Layer
rt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?
- sender doesn’t know what happened at receiver!

What to do?
- sender NAKs for receiver’s ACK/NAK? What if sender NAK corrupted?
- retransmit, assuming it is NAK …
- but this might cause retransmission of correctly received pkt!
   - packet duplications!

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

rt2.1: sender, handles garbled ACK/NAKs
rtd 2.1: receiver, handles garbled ACK/NAKs

rtd 2.1 in action

a) operation with no corruption

b) packet corrupted
rdt 2.1 in action (cont)

c) ACK corrupted

rdt2.2: a NAK-free protocol

receiver FSM fragment

sender FSM fragment
**rdt 2.2 in action**

sender

- send pkt0
- rcv ACK0
- send pkt1
- rcv ACK1
- send pkt0
- rcv ACK0

receiver

- pkt0
- ACK0
- pkt1
- ACK1
- pkt0
- ACK0

a) operation with no corruption

**rdt 2.2 in action (cont)**

sender

- send pkt0
- rcv ACK0
- resend pkt1
- rcv ACK1
- send pkt0
- rcv ACK0

receiver

- pkt0
- ACK0
- pkt1
- ACK1
- pkt0
- ACK0

( коррумпировано )

b) packet corrupted

c) ACK corrupted
**rdt3.0 channels with errors and loss**

- **Sender** sends one packet, then waits for receiver response.

**rdt3.0: Poor performance**

- "Stop-and-Wait" protocol:
  - Sender sends one packet, then waits for receiver response.
  - Receiver waits for ACK before sending the next packet.

**Equation**:

\[ U_{sender} = \frac{L/R}{RTT + L/R} \]
Performance of rdt3.0

- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[ T_{\text{transmit}} = \frac{L}{R} = \frac{8 \text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec} \]

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

- \( U_{\text{sender}} \): utilization - fraction of time sender busy sending
- 1KB pkt every 30 msec \( \rightarrow \) 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
- microsec = \( 10^{-6} \) sec, millisec=ms=\( 10^{-3} \) s, Gb, Mb, Kb

Pipelined 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
Pipelining: increased utilization

- Increased utilization by a factor of 3

Two generic forms of pipelined protocols: go-Back-N, selective repeat

Go-Back-N

Sender:
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed - sliding window
- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for the packet of send_base
- timeout(n): retransmit pkt n and all higher seq # pkts in window
**GBN: sender extended FSM**

- `rdt_send(data)`
  - `if (nextseqnum < base+N) {
      sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)
      udt_send(sndpkt[nextseqnum])
      if (base == nextseqnum)
        start_timer
      nextseqnum++
  }
  else
    refuse_data(data)`

- `udt_send(sndpkt[base+1])`
- `...`
- `udt_send(sndpkt[nextseqnum-1])`

**GBN: receiver extended FSM**

- `default` `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && hasseqnum(rcvpkt, expectedseqnum)`

**ACK-only:** always send ACK for correctly-received pkt
- with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember expectedseqnum

**out-of-order pkt:**
- discard (don’t buffer) -> no receiver buffering!
- Re-ACK pkt with highest in-order seq #
**GBN in action**

- **sender**
  - send pkt0
  - send pkt1
  - send pkt2
  - send pkt3 (wait)
  - rcv pkt2, timeout
  - send pkt2
  - send pkt3
  - send pkt4
  - send pkt5

- **receiver**
  - rcv pkt0
  - send ACK0
  - rcv pkt1
  - send ACK1
  - rcv pkt3, discard
  - send ACK1
  - rcv pkt4, discard
  - send ACK1
  - rcv pkt5, discard
  - send ACK1
  - rcv pkt2, deliver
  - send ACK2
  - rcv pkt3, deliver
  - send ACK3

**GBN: sender extended FSM**

```c
void rdt_send(data)
{
    if (nextseqnum < base+N) {
        sndpkt[nextseqnum] = make_pkt(nextseqnum, data, checksum)
        udt_send(sndpkt[nextseqnum])
        if (base == nextseqnum)
            start_timer
        nextseqnum++
    } else
    refuse_data(data)
}

void rdt_rcv(rcvpkt)
{
    if (notcorrupt(rcvpkt)
        if (corrupt(rcvpkt)
            base = getacknum(rcvpkt)+1
            if (base == nextseqnum)
                stop_timer
            else
                start_timer
            }
    ```
GBN in action

Cumulative ACK

Sender

send pkt0
send pkt1
send pkt2
send pkt3
rcv ACK0
send pkt4
rcv ACK1
send pkt5
rcv ACK5
send pkt6
send pkt7
send pkt8
send pkt9

Receiver

rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt2
send ACK2
rcv pkt3
send ACK3
rcv pkt4
send ACK4
rcv pkt5
send ACK5

(loss)X

(loss)X

(loss)X
GBN in action

Premature timeout

Sender
send pkt0
send pkt1
send pkt2
send pkt3
rcv ACK0
send pkt4
rcv ACK1
send pkt5

Receiver
rcv pkt0 send ACK0
rcv pkt1 send ACK1
rcv pkt3 send ACK1
rcv pkt4 send ACK2
rcv pkt5 send ACK2

pkt2 timeout
send pkt2,3,4,5

rcv pkt2 send ACK2
send ACK2
rcv pkt4,discard
send ACK2
rcv pkt5,discard

GBN in action

Premature timeout

Sender
send pkt0
send pkt1
send pkt2
send pkt3
rcv ACK0
send pkt4
rcv ACK1
send pkt5

Receiver
rcv pkt0 send ACK0
rcv pkt1 send ACK1
rcv pkt3,discard
send ACK1
rcv pkt2 send ACK2
rcv pkt4,discard
send ACK2
rcv pkt5,discard

pkt2 timeout
send pkt2,3,4,5
Selective Repeat

- receiver individually acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #'s of sent, unACKed pkts

Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective repeat

sender

data from above:
- if next available seq # in window, send pkt
- timeout(n):
  - resend pkt n, restart timer
- ACK(n) in [sendbase, sendbase+N]:
  - mark pkt n as received
  - if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N, rcvbase-1]
- ACK(n)
- otherwise:
  - ignore

Selective repeat in action

pkt0 sent
0 [1 2 3 4 5 6 7 8 9]

pkt1 sent
0 [1 2 3 4 5 6 7 8 9]

pkt2 sent
0 [1 2 3 4 5 6 7 8 9]

pkt3 sent, window full
0 [1 2 3 4 5 6 7 8 9]

ACK0 recv, pkt4 sent
0 [1 2 3 4 5 6 7 8 9]

ACK1 recv, pkt5 sent
0 [1 2 3 4 5 6 7 8 9]

pkt2 TIMEOUT, pkt2 resend
0 [1 2 3 4 5 6 7 8 9]

ACK2 recv, nothing sent
0 [1 2 3 4 5 6 7 8 9]

pkt6 recv, delivered, ACK0 sent
0 [1 2 3 4 5 6 7 8 9]

pkt3 recv, buffered, ACK3 sent
0 [1 2 3 4 5 6 7 8 9]

pkt4 recv, buffered, ACK4 sent
0 [1 2 3 4 5 6 7 8 9]

pkt5 recv, buffered, ACK5 sent
0 [1 2 3 4 5 6 7 8 9]

pkt6 recv, pkt2, pkt3, pkt4, pkt5 delivered, ACK2 sent
0 [1 2 3 4 5 6 7 8 9]
Selective repeat: dilemma

Example:
- seq #’s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size? Will this happen in GBN?

Go Back N vs. Selective Repeat

- Efficiency
  - No loss
  - Loss
    - Bursty loss
    - Sporadic loss

- Resource consumption
  - Buffer space
  - Timer
    - How to implement multi-timers?
Outline

- 1. Transport-layer services
- 2. Multiplexing and demultiplexing
- 3. Connectionless transport: UDP
- 4. Principles of reliable data transfer
- 5. Connection-oriented transport: TCP
- 6. TCP congestion control
- 7. TCP fairness and delay performance

TCP: Overview

- End-to-end, unicast:
  - one sender, one receiver
- reliable, in-order byte steam:
  - no "message boundaries"
- Pipelined (not stop-wait):
  - TCP congestion and flow control set window size
  - send & receive buffers
- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver
TCP segment structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port #</td>
<td>16</td>
<td>Source port number</td>
</tr>
<tr>
<td>Dest port #</td>
<td>16</td>
<td>Destination port number</td>
</tr>
<tr>
<td>Sequence number</td>
<td>32</td>
<td>Sequence number</td>
</tr>
<tr>
<td>Acknowledgement number</td>
<td>32</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>Receive window</td>
<td>16</td>
<td>Receive window</td>
</tr>
<tr>
<td>Urgent data pointer</td>
<td>8</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>Checksum</td>
<td>16</td>
<td>Checksum for TCP</td>
</tr>
<tr>
<td>Options</td>
<td>Varies</td>
<td>Additional options has to do with the transport layer (variable length)</td>
</tr>
</tbody>
</table>

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)

TCP Connection Setup

Three way handshake:

- **Step 1:** Client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data
- **Step 2:** Server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #
- **Step 3:** Client receives SYNACK, replies with ACK segment, which may contain data - piggyback

Q: Is 3-way handshake perfect?
TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer

- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

**Data rcvd from app:**

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- 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 - cumulative ack
  - start timer if there are outstanding segments
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

**Comment:**
- **SendBase-1:** last cumulatively acknowledged byte

**Example:**
- **SendBase-1 = 71;**
- **$y = 73,$** so the rcvr wants 73+;
- $y > SendBase,$ so that new data is acked.

---

**TCP: retransmission scenarios**

**Host A**
- Seq=92, 8 bytes data
- SendBase = 100
- lost ACK scenario

**Host B**
- ACK=100

**Host A**
- Seq=92, 8 bytes data
- SendBase = 100
- ACK=100

**Host B**
- Seq=100, 20 bytes data
- ACK=120

**Host A**
- Seq=92, 8 bytes data
- SendBase = 120
- ACK=120

**Host B**
- Seq=100, 20 bytes data
- ACK=100

**Premature timeout**
**TCP retransmission scenarios (more)**

TCP ACK generation [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
</tbody>
</table>
Fast Retransmit

- Time-out period may be relatively long:
  - $eRTT + 4DevRTT$
  - long delay before resending lost packet
- Solution: Fast Retransmit
  - Hint: GBN

GBN in action

sender
- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)
- rcv ACK0
  - send pkt4
  - rcv ACK1
    - send pkt5

receiver
- rcv pkt0
  - send ACK0
- rcv pkt1
  - send ACK1
- rcv pkt3
  - discard
  - send ACK1
- rcv pkt4
  - discard
  - send ACK1
- rcv pkt5
  - discard
  - send ACK1
- rcv pkt2
  - deliver
  - send ACK2
- rcv pkt3
  - deliver
  - send ACK3
- pkt2 timeout
Fast Retransmit

- Time-out period may be relatively long:
  - eRTT + 4DevRTT
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - **fast retransmit**: resend segment before timer expires

Fast retransmit algorithm:

```c
event: ACK received, with ACK field value of y
if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
        start timer
} else {
    increment count of dup ACKs received for y
    if (count of dup ACKs received for y = 3) {
        resend segment with sequence number y
    }
```

- A duplicate ACK for already ACKed segment
- Fast retransmit
TCP Round Trip Time and Timeout

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt

One RTT sample

**Problem 2:**
- SampleRTT will vary -> atypical
  - Need the trend of RTT: history -> future
  - average several recent measurements, not just current

SampleRTT

**RTT: gaia.cs.umass.edu to fantasia.eurecom.fr**

<table>
<thead>
<tr>
<th>time (seconds)</th>
<th>RTT (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td>29</td>
<td>36</td>
</tr>
<tr>
<td>36</td>
<td>43</td>
</tr>
<tr>
<td>43</td>
<td>50</td>
</tr>
<tr>
<td>50</td>
<td>57</td>
</tr>
<tr>
<td>57</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>71</td>
</tr>
<tr>
<td>71</td>
<td>78</td>
</tr>
<tr>
<td>78</td>
<td>85</td>
</tr>
<tr>
<td>85</td>
<td>92</td>
</tr>
<tr>
<td>92</td>
<td>99</td>
</tr>
<tr>
<td>99</td>
<td>106</td>
</tr>
</tbody>
</table>

**Estimated RTT**
TCP Round Trip Time and Timeout

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

- typical value: \( \alpha = 0.125 \)
- influence of past sample decreases exponentially fast
  - Exponential weighted moving average

Outline

- 1. Transport-layer services
- 2. Multiplexing and demultiplexing
- 3. Connectionless transport: UDP
- 4. Principles of reliable data transfer
- 5. Connection-oriented transport: TCP
- 6. TCP congestion control
- 7. TCP fairness and delay performance
**Principles of Congestion Control**

**Congestion:**
- informally: “too many sources sending too many data too fast for network to handle”

**Solution**
- Sender controls sending rate
- different from flow control!
  - Flow control: not overwhelm receiver
  - Congestion control: not overwhelm network
- another top-10 problem!

**Approaches towards congestion control**

Two broad approaches towards congestion control:

**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

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

Fast, accurate, but expensive
TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:
  \[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]

\[ RcvWindow? \leq \min \{ \text{rcwWindow}, \text{CongWin} \} \]

- \text{CongWin} is dynamic, function of perceived network congestion
  - Too high a rate \( \rightarrow \) congestion
  - Too low a rate \( \rightarrow \) low network utilization

TCP Congestion Control

How does sender perceive congestion?

- loss event
- TCP sender reduces rate (\text{CongWin}) after loss event

  \[ \text{Loss event} = \text{timeout} \text{ or } 3 \text{ duplicate acks} \]

three mechanisms:

- AIMD (additive increase multiplicative decrease)
- slow start
- conservative after timeout events
1. TCP AIMD

- **additive increase:**
  - increase CongWin by 1 MSS every RTT in the absence of loss events: probing

- **multiplicative decrease:**
  - cut CongWin in half after loss event

![Sawtooth diagram]

Long-lived TCP connection

2. TCP Slow Start

- When connection begins, CongWin = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast until first loss event
2. TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double CongWin every RTT
  - done by incrementing CongWin for every ACK received
- **Summary:** initial rate is slow but ramps up exponentially fast

3. Refinement (TCP Reno)

- After 3 dup ACKs:
  - CongWin is cut in half
  - window then grows linearly
- But after timeout event:
  - CongWin instead set to 1 MSS;
  - window then grows exponentially
  - to a **Threshold**, then grows linearly

**Philosophy:**
- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is "more alarming"

**TCP versions:**
- Tahoe -> Reno -> Sack
- Vegas, Westwood ... (Nevada)
**Refinement (more)**

**Q:** Threshold: When will exponential increase switch to linear?

**A:** When CongWin gets to 1/2 of its value before timeout.

**Implementation:**
- Variable Threshold
- At a loss event, Threshold is set to 1/2 of CongWin just before loss event

---

**TCP congestion behavior (1)**

---
TCP congestion behavior (2)

TCP congestion behavior (3)
Summary: TCP Congestion Control (Reno)

- When CongWin is below Threshold, sender in slow-start phase, window grows exponentially.
- When CongWin is above Threshold, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.


Outline

1. Transport-layer services
2. Multiplexing and demultiplexing
3. Connectionless transport: UDP
4. Principles of reliable data transfer
5. Connection-oriented transport: TCP
6. TCP congestion control
7. TCP fairness and delay performance
**TCP Fairness**

**Fair:**
1. Equal share
2. Full utilization

**Goal:** If K TCP sessions share the same bottleneck link of bandwidth R, each should have an average rate of $R/K$.

---

**TCP AIMD**

**additive increase:**
- Increase $\text{CongWin}$ by 1 MSS every RTT in the absence of loss events: probing

**multiplicative decrease:**
- Cut $\text{CongWin}$ in half after loss event

---

*Long-lived TCP connection*
Why is TCP fair?

Two competing sessions:
- Additive increase gives slope of 1, as throughput increases.
- Multiplicative decrease decreases throughput proportionally.
**Why is TCP fair?**


---

**Fairness (more)**

**Fairness and UDP**
- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly, more on later

**Fairness and parallel TCP connections**
- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers/FTP client do this
  - NetAnts, GetRight
- Example: link of rate R with 9 ongoing TCP connections
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets > R/2!
Delay performance

Q: How long does it take to receive an object from a Web server after sending a request?

Methods
- Measurement
  - Ping, traceroute
- Simulation
  - Ns-2
- Analytical modeling
  - Math

Delay modeling - No Congestion

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:
- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:
- Assume one link between client and server of rate R
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Window size:
- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start
Fixed congestion window (1)

First case:
WS/R > RTT + S/R: ACK for first segment in window returns before window’s worth of data sent

delay = 2RTT + O/R
Second case:
- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

\[ \text{delay} = 2\text{RTT} + \frac{O/R}{K} + (K-1)[S/R + \text{RTT} - \frac{W/O}{S/R}] \]
Fixed congestion window (2)

Second case:
- \( WS/R < RTT + S/R \): wait for ACK after sending window's worth of data sent

\[
\text{delay} = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]
\]

\( K = O/(WS) \)

TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

But no congestion

Will show that the delay for one object is:

\[
\text{Latency} = 2RTT + \frac{O}{R} + P \left( RTT + \frac{S}{R} \right) - (2^P - 1) \frac{S}{R}
\]

where \( P \) is the number of times TCP idles at server:

\[
P = \min\{Q, K-1\}
\]

- \( Q \) is the number of times the server idles if the object were of infinite size.
- \( K \) is the number of windows that cover the object.
**Case 1: P = Q**

**Delay components:**
- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

Server idles:
\[ P = \min(K-1,Q) \] times

**Example:**
- O/S = 15 segments
- K = 4 windows
- Q = 2
- \( P = \min(K-1,Q) = 2 \)

Server idles \( P = 2 \) times

**Case 2: P = K-1**

**Delay components:**
- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

Server idles:
\[ P = \min(K-1,Q) \] times

**Example:**
- O/S = 3 segments
- K = 2 windows
- Q = 2
- \( P = \min(K-1,Q) = 1 \)

Server idles \( P = 1 \) times
TCP Delay Modeling (contd)

\[
\frac{S}{R} + RTT = \text{time from when server starts to send segment until server receives acknowledgement}
\]

\[
2^{k-1} \frac{S}{R} = \text{time to transmit the } k\text{th window}
\]

\[
\left[ \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]^+ = \text{idle time after the } k\text{th window}
\]

\[
\text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^{K} \text{idleTime}_p
\]
\[
= \frac{O}{R} + 2RTT + \sum_{k=1}^{K} \left[ \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]
\]
\[
= \frac{O}{R} + 2RTT + P[RTT + \frac{S}{R}] - (2^p - 1) \frac{S}{R}
\]

Recall \( K = \text{number of windows that cover object} \)

How do we calculate \( K \)?

\[
K = \min \{ k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O \}
\]
\[
= \min \{ k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O/S \}
\]
\[
= \min \{ k : 2^k - 1 \geq O/S \}
\]
\[
= \min \{ k : k \geq \log_2 \left( \frac{O}{S} + 1 \right) \}
\]
\[
= \left\lceil \log_2 \left( \frac{O}{S} + 1 \right) \right\rceil
\]

Calculation of \( Q, \text{number of idles for infinite-size object} \), is similar

\[
\max \{ q : 2^{q-1} S/R \leq RTT + S/R \}
\]