Network Transport Layer:
Transport Reliability:
Sliding Windows; Connection Management; TCP

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

10/31/2017
Admin.: PS4

Part 1
- Discussion checkpoint: Nov. 7; code checkpoint Nov. 10

Part 2
- Discussion checkpoint: Nov. 14; all due Nov. 16

proj-sol:
- 129  400  3045 FishThread.java
- 388 1457 12873 Node.java
- 51  167  1145 PingRequest.java
- 83  250  2106 SimpleTCPSockSpace.java
- 181 605  5248 TCPManager.java
- 889 3088 26381 TCPSock.java
- 60  149  1316 TCPSockID.java
- 123 382  3866 TransferClient.java
- 147 500  5059 TransferServer.java

2051  6998  61039 total

proj:
- 129  400  3045 FishThread.java
- 341 1301 11313 Node.java
- 51  167  1145 PingRequest.java
- 50  128  909 TCPManager.java
- 132 460  3146 TCPSock.java
- 123 382  3866 TransferClient.java
- 147 500  5059 TransferServer.java

973  3338 28483 total
Recap: Reliable Data Transfer Context

**send side**
- **rdt_send()**: called from above, (e.g., by app.)
- **udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver

**unreliable channel**

**receive side**
- **deliver_data()**: called by rdt to deliver data to upper
- **rdt_rcv()**: called from below; when packet arrives on rcv-side of channel

**reliable data transfer protocol (sending side)**
- **data**

**reliable data transfer protocol (receiving side)**
- **data**

**packet**
Recap: rdt2.0: Reliability allowing only Data Msg Corruption

sender

\[
\text{rdt\_send(data)} \\
\text{snkpkt = make\_pkt(data, checksum)} \\
\text{udt\_send(sndpkt)} \\
\text{wait for data} \\
\text{wait for ACK or NAK} \\
\text{rdt\_rcv(rcvpkt) && isNAK(rcvpkt)} \\
\text{udt\_send(sndpkt)} \\
\text{rdt\_rcv(rcvpkt) && isACK(rcvpkt)} \\
\wedge
\]

receiver

\[
\text{rdt\_rcv(rcvpkt) && ncorrupt(rcvpkt)} \\
\text{udt\_send(NAK)} \\
\text{wait for data} \\
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)} \\
\text{extract(rcvpkt, data)} \\
\text{deliver\_data(data)} \\
\text{udt\_send(ACK)}
\]
Recap: Rdt2.0 Analysis

Execution traces of rdt2.0:
{data^ NACK}*  data deliver
ACK
Recap: rdt2.1b: Reliability allowing Data/Control Msg Corruption

sender

Fix wrong guess by checking seq#

receiver

Guess garbled feedback as NAK
Protocol Analysis using (Generic) Execution Traces Technique

- **Issue**: how to systematically enumerate all potential execution traces to understand and verify correctness

- **A systematic approach to enumerating exec. traces is to compute joint sender/receiver/channels state machine**

sender state: waiting for n

receiver state: waiting for n

snd->rcv channel state

rcv->snd channel state

\[ S_n r_{n+1} - \text{ACK} \]

\[ S_n r_n - \text{NAK} \]

\[ w_n r_n - - \quad \text{snd d}_n \quad s_n r_n d_n - \quad \text{rcv d}_n \]

...
Recap: Protocol Analysis using (Generic) Execution Traces Technique

Execution traces of rdt2.1b are all that can be generated by the finite state machine above.
Recap: Protocol Analysis using State Invariants

W1: wait for data with seq. 1
S1: sending data with seq. 1

State invariant:
- When receiver’s state is waiting for seq #n, sender’s state can be sending either seq#n-1 or seq#n, and only either #n or #n-1 packets can arrive
rdt2.1c: Sender, Handles Garbled ACK/NAKs: Using 1 bit (Alternating-Bit Protocol)
**rdt2.1c: Receiver, Handles Garbled ACK/NAKs: Using 1 bit**

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

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK, chksum)`
  - `udt_send(sndpkt)`

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

- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt)`
  - `sndpkt = make_pkt(NAK, chksum)`
  - `udt_send(sndpkt)`
rdt2.1c: Summary

**Sender:**
- state must “remember” whether “current” pkt has 0 or 1 seq. #

**Receiver:**
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
rdt2.2: a NAK-free protocol

- Same functionality as rdt2.1c, 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

- rdt_send(data)
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
- rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
  - sndpkt = make_pkt(ACK,1, checksum)
  - udt_send(sndpkt)

receiver FSM fragment

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK,0, checksum)
  - udt_send(sndpkt)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
  - udt_send(sndpkt)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
  - \( \Lambda \)
Outline

- Admin and review
  - Reliable data transfer
    - perfect channel
    - channel with bit errors
  - channel with bit errors and losses
**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
- requires countdown timer
- 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

**Q:** Does rdt2.2 work under losses?
### rdt3.0 Sender

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

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

- rdt_rcv(rcvpkt) &&
  - notcorrupt(rcvpkt)
  - isACK(rcvpkt,1)
  - stop_timer

- timeout
  - udt_send(sndpkt)
  - start_timer

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

- timeout
  - udt_send(sndpkt)
  - start_timer

- rdt_send(data)
  - sndpkt = make_pkt(1, data, checksum)
  - udt_send(sndpkt)
  - start_timer

- rdt_rcv(rcvpkt)
  - Lambda
rdt3.0 in Action

(a) operation with no loss

(b) lost packet
rdt3.0 in Action

Question to think about: How to determine a good timeout value?
Home exercise: What are execution traces of rdt3.0? What are some state invariants of rdt3.0?
rdt3.0: Protocol Analysis using State Invariants

State consistency:
When receiver’s state is waiting n, the state of the sender is either sending for n-1 or sending for n.

When sender’s state is sending for n, receiver’s state is waiting for n or n + 1.
What is $U_{\text{sender}}$: utilization – fraction of time link busy sending?

Assume: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[
T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{9} \text{ b/sec}} = 8 \text{ microsec}
\]

\[
U_{\text{sender}} = \frac{L / R}{\text{RTT} + L / R} = \frac{0.008}{30.008} = 0.00027
\]

- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!
A Summary of Questions

- How to improve the performance of rdt3.0?
- What if there are reordering and duplication?
- How to determine the “right” timeout value?
**Sliding Window Protocols: Pipelining**

**Pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

![Diagram showing a stop-and-wait protocol in operation and a pipelined protocol in operation](attachment:image.png)
Pipelining: Increased Utilization

Question: a rule-of-thumb window size?
Realizing Sliding Window: Go-Back-n

Sender:

- **k-bit seq #** in pkt header
- “**window**” of up to \( W \), consecutive unack’ed pkts allowed

- **ACK(n):** ACKs all pkts up to, including seq # n - “cumulative ACK”
  - note: ACK(n) could mean two things: I have received **upto and include** n, or I am waiting for n
- **timer for the packet at base**
- **timeout(n):** retransmit pkt n and all higher seq # pkts in window
GBN: Sender FSM

\[ \text{ramdom}_{\text{send}(\text{data})} \]

\[ \begin{align*}
&\text{if (nextseqnum} < \text{base}+W) \{ \\
&\quad \text{sndpkt[nextseqnum] = make_pkt(nextseqnum, data, checksum)} \\
&\quad \text{udt_send(sndpkt[nextseqnum])} \\
&\quad \text{if (base} == \text{nextseqnum}) \text{start_timer} \\
&\quad \text{nextseqnum}++ \\
&\} \text{ else} \\
&\quad \text{block sender} \\
\end{align*} \]

\[ \text{timeout}_{\text{udt_send(sndpkt[base])}} \]

\[ \text{udt_send(sndpkt[base+1])} \]

\[ \ldots \]

\[ \text{udt_send(sndpkt[nextseqnum-1])} \]

\[ \Lambda \]

\[ \text{base}=1 \]

\[ \text{nextseqnum}=1 \]

\[ \text{rdt}_{\text{recv}(\text{rcvpkt})} \&\& \text{corrupt}(\text{rcvpkt}) \]

\[ \text{rdt}_{\text{recv}(\text{rcvpkt})} \&\& \text{notcorrupt}(\text{rcvpkt}) \]

\[ \text{if (new packets ACKed)} \{ \\
&\quad \text{advance base;} \\
&\quad \text{if (more packets waiting)} \\
&\quad \quad \text{send more packets} \\
&\} \]

\[ \text{if (base} == \text{nextseqnum}) \text{stop_timer} \]

\[ \text{else} \\
&\quad \text{start_timer for the packet at new base} \]
Only state: expectedseqnum

- out-of-order pkt:
  - discard (don’t buffer) -> no receiver buffering!
  - re-ACK pkt with highest in-order seq #
  - may generate duplicate ACKs
**GBN in Action**

Window size = 4

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
Analysis: Efficiency of Go-Back-n

- Assume window size $W$
- Assume each packet is lost with probability $p$
- On average, how many packets do we send for each data packet received?
Selective Repeat

- Sender window
  - Window size $W$: $W$ consecutive unACKed seq #’s
- Receiver *individually* acknowledges correctly received pkts
  - buffers out-of-order pkts, for eventual in-order delivery to upper layer
  - $ACK(n)$ means received packet with seq# n only
  - buffer size at receiver: window size
- Sender only resends pkts for which $ACK$ not received
  - sender timer for each unACKed pkt
Selective Repeat: Sender, Receiver Windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective Repeat

sender
data from above:
- unACKed packets is less than window size $W$, send; otherwise block app.

timeout($n$):
- resend pkt $n$, restart timer

$ACK(n)$ in $[sendbase, sendbase+W-1]$:
- mark pkt $n$ as received
- update sendbase to the first packet unACKed

receiver

pkt $n$ in $[rcvbase, rcvbase+W-1]$
- send $ACK(n)$
- if (out-of-order)
  - mark and buffer pkt $n$
  else /* in-order */
    - deliver any in-order packets

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

pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9

pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9

ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9
Discussion: Efficiency of Selective Repeat

- Assume window size $W$

- Assume each packet is lost with probability $p$

- On average, how many packets do we send for each data packet received?
Selective Repeat: Seq# Ambiguity

Example:

- seq #’s: 0, 1, 2, 3
- window size=3
- Error: incorrectly passes duplicate data as new.
State Invariant: Window Location

- Go-back-n (GBN)

  - Sender window
  - Receiver window

- Selective repeat (SR)

  - Sender window
  - Receiver window
Window Location

- Go-back-n (GBN)
- Selective repeat (SR)

Q: what relationship between seq # size and window size?
Sliding Window Protocols: Go-back-n and Selective Repeat

<table>
<thead>
<tr>
<th></th>
<th>Go-back-n</th>
<th>Selective Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>data bandwidth: sender to receiver (avg. number of times a pkt is transmitted)</td>
<td>Less efficient ( \frac{1-p+pw}{1-p} )</td>
<td>More efficient ( \frac{1}{1-p} )</td>
</tr>
<tr>
<td>ACK bandwidth (receiver to sender)</td>
<td>More efficient</td>
<td>Less efficient</td>
</tr>
<tr>
<td>Relationship between ( M ) (the number of seq#) and ( W ) (window size)</td>
<td>( M &gt; W )</td>
<td>( M \geq 2W )</td>
</tr>
<tr>
<td>Buffer size at receiver</td>
<td>1</td>
<td>( W )</td>
</tr>
<tr>
<td>Complexity</td>
<td>Simpler</td>
<td>More complex</td>
</tr>
</tbody>
</table>

\( p \): the loss rate of a packet; \( M \): number of seq# (e.g., 3 bit \( M = 8 \)); \( W \): window size
Question: What is Initial Seq#?
Question: What is Initial Seq#?
Outline

- Review
  - Reliable data transfer
    - perfect channel
    - channel with bit errors
    - channel with bit errors and losses
    - sliding window: reliability with throughput
  - connection management
Three Way Handshake (TWH) [Tomlinson 1975]

- SYN: indicates connection setup
- notify initial seq#. Accept?
- think of y as a challenge
- accept data only after verified y is bounced back
- x is the init. seq

SYN: indicates connection setup
Scenarios with Duplicate Request/SYN Attack

Host A

- SYN(seq=x)
- ACK(seq=x), SYN(seq=y)
- no such request

Host B

- accept?
- reject
- REJECT(seq=y)
Scenarios with Duplicate Request/SYN Attack

Host A

no such request

host B

accept?

SYN(seq=x)

ACK(seq=x), SYN(seq=y)

reject

ACK(seq=z)

REJECT(seq=y)
To avoid that “SYNC ACK y” comes from reordering and duplication

- for each connection (sender-receiver pair), ensuring that two identically numbered packets are never outstanding at the same time
  - network bounds the life time of each packet
  - a sender will not reuse a seq# before it is sure that all packets with the seq# are purged from the network
  - seq. number space should be large enough to not limit transmission rate
**Connection Close**

- **Why connection close?**
  - so that each side can release resource and remove state about the connection (do not want dangling socket)
General Case: The Two-Army Problem

The gray (blue) armies need to agree on whether or not they will attack the white army. They achieve agreement by sending messengers to the other side. If they both agree, attack; otherwise, no. Note that a messenger can be captured!
Four Way Teardown

- can retransmit the ACK if its ACK is lost

- all states removed
Outline

- Recap
- Reliable data transfer
  - TCP reliability
TCP: Overview

- Point-to-point reliability: one sender, one receiver

- Flow controlled and congestion controlled
Evolution of TCP

- 1983: BSD Unix 4.2 supports TCP/IP.
- 1984: Nagel’s algorithm to reduce overhead of small packets; predicts congestion collapse.
- 1985: Congestion collapse 1st observed.
- 1986: Karn’s algorithm to better estimate round-trip time.
- 1988: Van Jacobson’s algorithms - slow start, congestion avoidance, fast retransmit (all implemented in 4.3BSD Tahoe). SIGCOMM 88.

Evolution of TCP

- 1993: TCP Vegas (not implemented)
  - TCP Vegas: real congestion avoidance (Brakmo et al)
- 1994: ECN
  - ECN: Explicit Congestion Notification (Floyd)
- 1996: Improving TCP startup (Hoe)
- 1996: NewReno
  - NewReno: modified fast recovery
  - SACK TCP: Selective Ack (Floyd et al)
- Multiple versions

TCP Reliable Data Transfer

- **Connection-oriented:**
  - Connection management
    - setup (exchange of control msgs) init’s sender, receiver state before data exchange
    - close

- **Full duplex data:**
  - bi-directional data flow in same connection

- **A sliding window protocol**
  - a combination of go-back-n and selective repeat:
    - send & receive buffers
    - cumulative acks
    - TCP uses a single retransmission timer
    - do not retransmit all packets upon timeout

Diagram:
- Application writes data
- TCP send buffer
- Segment
- TCP receive buffer
- Application reads data
- Socket door
TCP Segment Structure

URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection management (reset, setup teardown commands)
Also in UDP

Options (variable length)

application data (variable length)

32 bits

counting by bytes of data (not segments!)

flow control
Flow Control

- receive side of a connection has a receive buffer:

- app process may be slow at reading from buffer

flow control

sender won’t overflow receiver’s buffer by transmitting too much, too fast

speed-matching service: matching the send rate to the receiving app’s drain rate
TCP Flow Control: How it Works

- spare room in buffer = $\text{RcvWindow}$

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>dest port #</td>
</tr>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>rcvr window size</td>
<td></td>
</tr>
<tr>
<td>checksum</td>
<td>ptr urgent data</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td>application data (variable length)</td>
</tr>
</tbody>
</table>
TCP Seq. #’s and ACKs

**Seq. #’s:**
- byte stream “number” of first byte in segment’s data

**ACKs:**
- seq # of next byte expected from other side
- cumulative ACK

---

**Simple Telnet Scenario**

User types ‘C’

<table>
<thead>
<tr>
<th></th>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=42, ACK=79, data = ‘C’</td>
<td>host ACKs receipt of ‘C’, echoes back ‘C’</td>
<td></td>
</tr>
<tr>
<td>Seq=79, ACK=43, data = ‘C’</td>
<td>host ACKs receipt of echoed ‘C’</td>
<td></td>
</tr>
<tr>
<td>Seq=42, ACK=80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

58
Fast Retransmit

- Problem: Timeout period often relatively long:
  - 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:
  - resend segment before timer expires
Triple Duplicate Ack

Packets

1 2 3 4 5 6 7

Acknowledgements (waiting seq#)

2 3 4 4 4 4 4
Fast Retransmit:

**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: reliable data transfer

Simplified TCP sender

```c
sendbase = initial_sequence number agreed by TWH
nextseqnum = initial_sequence number by TWH
loop (forever) {
    switch(event)
    event: data received from application above
        if (window allows send)
            create TCP segment with sequence number nextseqnum
            if (no timer) start timer
            pass segment to IP
            nextseqnum = nextseqnum + length(data)
        else put packet in buffer
    event: timer timeout for sendbase
        retransmit segment
        compute new timeout interval
        restart timer
    event: ACK received, with ACK field value of y
        if (y > sendbase) { /* cumulative ACK of all data up to y */
            cancel the timer for sendbase
            sendbase = y
            if (no timer and packet pending) start timer for new sendbase
            while (there are segments and window allow)
                sent a segment;
        }
        else { /* y==sendbase, duplicate ACK for already ACKed segment */
            increment number of duplicate ACKs received for y
            if (number of duplicate ACKS received for y == 3) {
                /* TCP fast retransmit */
                resend segment with sequence number y
                restart timer for segment y
            }
        }
    } /* end of loop forever */
```
## TCP Receiver 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 out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq. # of next expected byte</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>
</tbody>
</table>
%netstat -t -a

Diagram of TCP connection states and transitions:

1. **CLOSED**
   - Transition to **LISTEN**
   - Transition to **SYN_RCVD**
   - Transition to **FIN_WAIT_1**
   - Transition to **CLOSING**
   - Transition to **CLOSE_WAIT**
   - Transition to **LAST_ACK**
   - Transition to **TIME_WAIT**
   - Transition to **FIN_WAIT_2**
   - Transition to **TIME_WAIT**
   - Transition to **CLOSED**

2. **LISTEN**
   - Transition to **SYN_SENT**
   - Transition to **SYN_RCVD**

3. **SYN_RCVD**
   - Transition to **FIN_WAIT_1**
   - Transition to **CLOSING**
   - Transition to **CLOSE_WAIT**
   - Transition to **LAST_ACK**
   - Transition to **TIME_WAIT**

4. **FIN_WAIT_1**
   - Transition to **FIN_WAIT_2**

5. **CLOSING**
   - Transition to **CLOSE_WAIT**
   - Transition to **LAST_ACK**

6. **CLOSE_WAIT**
   - Transition to **LAST_ACK**

7. **LAST_ACK**
   - Transition to **TIME_WAIT**

8. **TIME_WAIT**
   - Transition to **CLOSED**

9. **FIN_WAIT_2**
   - Transition to **TIME_WAIT**

Diagrams show transitions between states with labels indicating the type of message being exchanged (e.g., SYN, ACK, FIN, etc.).
TCP Connection Management

TCP lifecycle: init SYN/FIN

- CLOSED
  - wait 30 seconds
  - receive FIN send ACK
  - send SYN
  - client application initiates a TCP connection

- TIME_WAIT
  - receive FIN send ACK

- SYN_SENT
  - receive SYN & ACK send ACK
  - client application initiates close connection

- FIN_WAIT_2
  - receive ACK send nothing

- FIN_WAIT_1
  - send FIN

- ESTABLISHED
  - client application initiates close connection

http://dsd.lbl.gov/TCP-tuning/ip-sysctl-2.6.txt
TCP Connection Management

TCP lifecycle: wait for SYN/FIN

CLOSED
- receive ACK, send nothing

LISTEN
- receive SYN, send SYN & ACK

SYN_RCVD
- receive ACK, send nothing

ESTABLISHED
- receive FIN, send ACK

CLOSE_WAIT
- send FIN

LAST_ACK
- send ACK

server application creates a listen socket