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

Y. Richard Yang

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

3/9/2016
Admin.: PS4

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

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

**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

`rdt_send()`: called from above, (e.g., by app.)
`udt_send()`: called by `rdt`, to transfer packet over unreliable channel to receiver
`deliver_data()`: called by `rdt` to deliver data to upper
`rdt_rcv()`: called from below; when packet arrives on rcv-side of channel
Recap: Potential Channel Errors

- bit errors
- loss (drop) of packets
- reordering or duplication

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).
Recap: rdt3.0 Sender (Bit Error/Loss)

- **Send (Bit Error/Loss)**
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`

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

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

- **Wait for call 0 from above**
- **Wait for ACK0**
- **Wait for call 1 from above**
- **Wait for ACK1**

- **timeout**
  - `udt_send(sndpkt)`
  - `start_timer`

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

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

- **rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))**
  - `Lambda`
Question to think about: What is a good timeout value?
rdt3.0 in Action

(c) lost ACK
(d) premature timeout
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 sender 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:

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

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

- 1KB pkt every 30 msec → 33kB/sec thruput 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

- Two generic forms of pipelined protocols: go-Back-N, selective repeat
**Pipelining: Increased Utilization**

- First packet bit transmitted, $t = 0$.
- Last bit transmitted, $t = L / R$.
- First packet bit arrives.
- Last packet bit arrives, send ACK.
- ACK arrives, send next packet, $t = \text{RTT} + L / R$.
- Last bit of 2nd packet arrives, send ACK.
- Last bit of 3rd packet arrives, send ACK.

Increase utilization by a factor of 3!

$$U_{\text{sender}} = \frac{3 \times L / R}{\text{RTT} + L / R} = \frac{0.024}{30.008} = 0.0008$$

**Question:** a rule-of-thumb window size?
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

\[
\text{rdt\_send(data)}
\]

\[
\text{if} \ (\text{nextseqnum} < \text{base} + W) \ \{ \\
\quad \text{sndpkt[nextseqnum]} = \text{make\_pkt(nextseqnum, data, chksum)} \\
\quad \text{udt\_send(sndpkt[nextseqnum])} \\
\quad \text{if} \ (\text{base} == \text{nextseqnum}) \text{ start\_timer} \\
\quad \text{nextseqnum}++
\}
\]

\[
\text{else} \\
\text{block sender}
\]

\[
\text{timeout}
\]

\[
\text{start\_timer} \\
\text{udt\_send(sndpkt[base])} \\
\text{udt\_send(sndpkt[base+1])}
\]

\[
\ldots
\]

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

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

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

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

\[
\text{else} \\
\text{start\_timer for the packet at new base}
\]
**GBN: Receiver FSM**

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

```plaintext
udt_send(sndpkt)

default

Wait

expectedseqnum=1
sndpkt = make_pkt(expectedseqnum,ACK,chksum)

ndt_rcv(rcvpkt)
&& notcurrect(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++
```
GBN in Action

window size = 4

sender

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

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
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 $[\text{sendbase}, \text{sendbase}+W-1]$:**
- mark pkt $n$ as received
- update sendbase to the first packet unACKed

**receiver**

- pkt $n$ in $[\text{rcvbase}, \text{rcvbase}+W-1]$
  - 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
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
pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 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?
State Invariant: Window Location

- Go-back-n (GBN)

- Selective repeat (SR)
Window Location

Go-back-n (GBN)

Selective repeat (SR)

Q: what relationship between seq # size and window size?
Selective Repeat: Seq# Ambiguity

Example:
- seq #’s: 0, 1, 2, 3
- window size=3
- incorrectly passes duplicate data as new in (a)
# Sliding Window Protocols: Go-back-n and Selective Repeat

<table>
<thead>
<tr>
<th>Data Bandwidth: Sender to Receiver (avg. number of times a pkt is transmitted)</th>
<th>Go-back-n</th>
<th>Selective Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less efficient</td>
<td>More efficient</td>
<td></td>
</tr>
<tr>
<td>$\frac{1-p+pw}{1-p}$</td>
<td>$\frac{1}{1-p}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACK Bandwidth (receiver to sender)</th>
<th>Go-back-n</th>
<th>Selective Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>More efficient</td>
<td>Less efficient</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationship between $M$ (the number of seq#) and $W$ (window size)</th>
<th>Go-back-n</th>
<th>Selective Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M &gt; W$</td>
<td>$M \geq 2W$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buffer size at receiver</th>
<th>Go-back-n</th>
<th>Selective Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$W$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Go-back-n</th>
<th>Selective Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpler</td>
<td>More complex</td>
<td></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]

- **Host A**: SYN(seq=x)
- **Host B**: ACK(seq=x), SYN(seq=y)
- **Host A**: ACK(seq=y)
- **Host B**: DATA(seq=x+1)

**Explanation**:
- **SYN**: Indicates connection setup
- **ACK**: Acknowledges receipt of SYN
- **DATA**: Transfers data

**Steps**:
1. **Host A** sends a SYN packet with an initial sequence number (x).
2. **Host B** responds with an ACK and a SYN with a new sequence number (y), acting as a challenge.
3. **Host A** sends an ACK for the SYN from **Host B**.
4. **Host B** sends a DATA packet with a sequence number (x+1).

**Notes**:
- **x** is the initial sequence number.
- **y** is a challenge sequence number.
- **Accept data only after verified y is bounced back**.
- **Think of y as a challenge**.
Scenarios with Duplicate Request/SYN Attack

Host A

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

REJECT(seq=y)

no such request

Host B

SYN(seq=x)

accept?

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

REJECT(seq=y)

reject
Scenarios with Duplicate Request/SYN Attack

Host A

SYN(seq=x)

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

Host B

accept?

no such request

ACK(seq=z)

reject

REJECT(seq=y)
Make Challenge “y” Robust

- To avoid that “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)

I am done. Are you done too?
I am done too. Goodbye!
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

propose close A->B

A->B closed

close

timed wait

- can retransmit the ACK if its ACK is lost

closed

all states removed

Host A

FIN

 ACK

 FIN

 ACK

A->B closed

propose close B->A

closed

all states removed

Host B

A->B closed
A Summary of Questions

- How to improve the performance of rdt3.0?
  - sliding window protocols
- What if there are duplication and reordering?
  - network guarantee: max packet life time
  - transport guarantee: not reuse a seq# before life time
  - seq# management and connection management
- How to determine the “right” parameters?
Outline

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

- Point-to-point reliability: one sender, one receiver
- Flow controlled and congestion controlled

RFCs: 793, 1122, 1323, 2018, 2581
Evolution of TCP

1974
TCP described by Vint Cerf, Bob Kahn
In IEEE Trans Comm

1975
Three-way handshake
Ray Tomlinson
In SIGCOMM 75

1981
TCP & IP
RFC 793 & 791

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

1987
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)

1990
4.3BSD Reno
Fast recovery delayed ACK’s

Evolution of TCP

- 1993: TCP Vegas (*not implemented*), real congestion avoidance (Brakmo et al)
- 1994: ECN Explicit Congestion Notification (Floyd)
- 1996: Improving TCP startup (Hoe)
- 1996: 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
- Socket door
- TCP send buffer
- TCP receive buffer
- Socket door
- Application reads data
- Segment
## TCP Segment Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>rcvr window size</td>
<td>Receiver window size</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>ptr urgent data</td>
<td>Pointer to urgent data</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td>Options field (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 management (reset, setup teardown commands)
- Also in UDP

**Flow control**: counting by bytes of data (not segments!)

**Application data**: (variable length)
Flow Control

- receive side of a connection has a receive 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

- app process may be slow at reading from buffer
TCP Flow Control: How it Works

- spare room in buffer = RcvWindow

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

Options (variable length)

application data (variable length)
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

---

**User types ‘C’**

**Host A**

- Seq=42, ACK=79, data = ‘C’

**Host B**

- Seq=79, ACK=43, data = ‘C’

- Seq=42, ACK=80

**host ACKs receipt of echoed ‘C’**

**host ACKs receipt of ‘C’, echoes back ‘C’**

**simple telnet scenario**
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 ($y > \text{SendBase}$) {

...  

$\text{SendBase} = y$

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

```plaintext
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

- CLOSED
- LISTEN
- SYN_RCVD
- SYN_SENT
- SYN_RCVD
- SYN_SENT
- ESTABLISHED
- OPEN
- CLOSE
- FIN_WAIT_1
- FIN_WAIT_2
- CLOSED

CONNECT/SYN
CLOSE/-
SYN/SYN + ACK
LISTEN/-
CLOSE/-
SYN/SYN + ACK
(estimul英镑ous open)
SEND/SYN
ACK/-
SYN_RCVD
FIN/ACK
(Step 3 of the three-way handshake)
CLOSE/FIN
(Active close)
FIN/ACK
(CLOSE/FIN)
(ESTABLISHED)
(closed)
FIN/ACK
(FIN_WAIT_1)
FIN/ACK
(FIN_WAIT_2)
TIME_WAIT
FIN/ACK
(TIME_WAIT)
CLOSED
(Timeout/)
ACK/-
(CLOSED)
(Go back to start)
TCP Connection Management

TCP lifecycle: init SYN/FIN

- CLOSED
  - SYN_SENT
    - ESTABLISHED
      - FIN_WAIT_2
        - FIN_WAIT_1
          - TIME_WAIT
            - CLOSED
              - SYN_SENT
                - ESTABLISHED
                  - FIN_WAIT_1
                    - FIN_WAIT_2
                      - TIME_WAIT

Client application initiates a TCP connection
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
  - send FIN
  - receive FIN: send ACK

- LAST_ACK
  - receive ACK: send nothing

- CLOSE_WAIT
  - receive FIN: send ACK

- ESTABLISHED
  - receive ACK: send nothing

- SYN_RCVD
  - receive SYN: send SYN & ACK

- LISTEN
  - receive SYN/ACK: ESTABLISHED

- server application creates a listen socket

- CLOSED
  - SYN_SENT
  - SYN_RCVD
  - ESTABLISHED

- SYN/ACK
  - ACK

- ESTABLISHED
  - FIN WAIT 1
  - FIN

- TIME_WAIT
  - FIN WAIT 2
  - FIN
  - LAST ACK
  - ACK
A Summary of Questions

- How to improve the performance of rdt3.0?
  ➤ sliding window protocols

- What if there are duplication and reordering?
  ➤ network guarantee: max packet life time
  ➤ transport guarantee: not reuse a seq# before life time
  ➤ seq# management and connection management

- How to determine the “right” parameters?