CMPE 150/L : Introduction to Computer Networks

Chen Qian

Computer Engineering
UCSC Baskin Engineering
Lecture 8
A lot of students have been having difficulty seeing the HTTP packets generated when navigating to "http://www.soe.ucsc.edu" in Chromium. I believe this is due to caching in the web browser.

As a fix, use "wget" to navigate to http://www.soe.ucsc.edu" instead of Chromium, and you should see the intended results.
Clarify

- How to call a “packet” in different layers
  - Application layer: message
  - Transport layer: segment
  - Network layer: datagram
  - Data link layer: frame

- They are used for precise presentation.
  - You are not required to be that precise in exams.
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:

How do humans recover from “errors” during conversation?
**rdt2.0: channel with bit errors**

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the question: how to recover from errors:*
  - **acknowledgements (ACKs):** receiver explicitly tells sender that pkt received OK
  - **negative acknowledgements (NAKs):** receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **rdt2.0** *(beyond rdt1.0):*
  - error detection
  - feedback: control msgs (ACK, NAK) from receiver to sender
### rdt2.0: FSM specification

**Sender**

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

**Receiver**

- `rdt_send(data)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`

**States and Transitions**

- **Wait for call from above**
- **Wait for ACK or NAK**
- **Wait for call from below**

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `Lambda`
rdt2.0: operation with no errors

- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(ACK)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(NAK)`
rdt2.0: error scenario

\[
\text{rdt}\_\text{send(data)} \\
\text{snkpkt} = \text{make}\_\text{pkt(data, checksum)} \\
\text{udt}\_\text{send(sndpkt)} \\
\]

Wait for call from above

Wait for ACK or NAK

\[
\text{rdt}\_\text{rcv(rcvpkt)} \&\& \text{isNAK(rcvpkt)} \\
\text{udt}\_\text{send(sndpkt)} \\
\]

Wait for call from below

\[
\text{rdt}\_\text{rcv(rcvpkt)} \&\& \text{notcorrupt(rcvpkt)} \\
\text{extract(rcvpkt, data)} \\
\text{deliver}_\text{data(data)} \\
\text{udt}_\text{send(ACK)} \\
\]

\[
\text{rdt}\_\text{rcv(rcvpkt)} \&\& \text{isACK(rcvpkt)} \\
\]

\[
\Lambda \\
\]
**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 pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn’t deliver up) duplicate pkt

---

**stop and wait**
sender sends one packet, then waits for receiver response
rdt2.1: idea

- Sender puts a seq num 0 or 1 to each segment.
- It sends a segment with 0 and then wait for an ACK.
- If receives ACK
  - Sends a segment with 1
- If receives NAK or corrupted ACK
  - Resends the segment with 0.

- Receiver receives a segment with 0.
  - Replies an ACK.
- Then if it receives a segment with 1.
  - The sender must received the ACK.
- If receives a segment with 0.
  - The sender did not receive the ACK.
**rdt2.1: sender, handles garbled ACK/NAKs**

```
rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  ( notcorrupt(rcvpkt) && isACK(rcvpkt) )

Lambda

Lambda

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
```

- **Wait for call 0 from above**
- **Wait for ACK or NAK 0**
- **Wait for call 1 from above**
- **Wait for ACK or NAK 1**
**rdt2.1: receiver, handles garbled ACK/NAKs**

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

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

- Wait for 0 from below
- Wait for 1 from below

- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK, chksum)`
- `udt_send(sndpkt)}`

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)}`
rdt2.1: discussion

**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 “expected” pkt should have seq # of 0 or 1

**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**

- \( \text{rdt\_send(data)} \)
- \( \text{sndpkt} = \text{make\_pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt\_send(sndpkt)} \)
- Wait for call 0 from above

**receiver FSM fragment**

- \( \text{udt\_send(sndpkt)} \)
- \( \text{rdt\_rcv(rcvpkt)} && \text{notcorrupt(rcvpkt)} && \text{has\_seq1(rcvpkt)} \)

- Wait for 0 from below

- \( \text{rdt\_rcv(rcvpkt)} && \text{notcorrupt(rcvpkt)} && \text{isACK(rcvpkt,1)} \) 
  \( \text{udt\_send(sndpkt)} \)

- \( \text{rdt\_rcv(rcvpkt)} && \text{notcorrupt(rcvpkt)} && \text{isACK(rcvpkt,0)} \)

- \( \Lambda \)

- \( \text{rdt\_rcv(rcvpkt)} && \text{corrupt(rcvpkt)} || \text{has\_seq1(rcvpkt)} \)

- Wait for ACK 0

- \( \text{extract(rcvpkt, data)} \)
- \( \text{deliver\_data(data)} \)

- \( \text{sndpkt} = \text{make\_pkt}(\text{ACK1}, \text{chksum}) \)
- \( \text{udt\_send(sndpkt)} \)
new assumption: underlying channel can also lose packets (data, 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 seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer
rdt3.0 sender

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

wait for ACK0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Lambda
udt_send(sndpkt)
start_timer

wait for call 0 from above
rdt_rcv(rcvpkt)
Lambda

wait for ACK1
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer

wait for call 1 from above
timeout
udt_send(sndpkt)
start_timer

Lambda
```

```
Lambda
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
```
**rdt3.0 in action**

**sender**
- send pkt0
  - rcv pkt0
  - send ack0

**receiver**
- rcv pkt0
  - send ack0

- send pkt1
  - rcv pkt1
  - send ack1

- rcv ack0
- send pkt0
  - rcv pkt0
  - send ack0

- rcv ack1
- send pkt0
  - rcv pkt0
  - send ack0

(a) no loss

**sender**
- send pkt0
  - rcv pkt0
  - send ack0

**receiver**
- rcv pkt0
  - send ack0

- send pkt1
  - rcv pkt1
  - send ack1

- rcv ack0
- send pkt0
  - rcv pkt0
  - send ack0

- rcv ack1
- send pkt0
  - rcv pkt0
  - send ack0

- send ack0

(b) packet loss
**rdt3.0 in action**

**sender**
- send pkt0
  - pkt0
  - rcv pkt0
    - ack0
      - send ack0

**receiver**
- rcv pkt0
  - send ack0
  - ack0
    - send pkt1
      - pkt1
        - rcv pkt1
          - ack1
            - send ack1
              - ack1
                - send ack0
                  - ack0

- rcv ack0
  - send ack0
  - ack0
    - send pkt0
      - pkt0
        - rcv pkt0
          - send ack0
            - ack0
              - send ack0

- rcv ack1
  - (detect duplicate)

- timeout
  - resend pkt1
    - pkt1
      - rcv pkt1
        - ack1
          - send ack1

(c) ACK loss

(d) premature timeout/ delayed ACK
Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[ D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs} \]

- **U_{\text{sender}}**: utilization – fraction of time sender busy sending

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

- if RTT=30 msec, 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**

- First packet bit transmitted, \( t = 0 \)
- Last packet bit transmitted, \( t = L / R \)
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, \( t = RTT + L / R \)

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

- **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

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

ACK arrives, send next packet, $t = RTT + L / R$

3-packet pipelining increases utilization by a factor of 3!

$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$
Pipelined protocols: overview

Go-back-N:
- sender can have up to N unacked packets in pipeline
- receiver only sends cumulative ack
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit all unacked packets

Selective Repeat:
- sender can have up to N unack’ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet
Go-Back-N: sender

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack'ed pkts allowed

\[ \text{send\_base} \quad \text{nextseqnum} \quad \text{window size} \quad N \]

- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window
Animation

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/
r dt _send(data)

if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)
    u dt _send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
    nextseqnum++
}
else
    refuse_data(data)

if (base == nextseqnum)
    stop_timer
else
    start_timer
ACK-only: always send ACK for correctly-received pkt with highest \textit{in-order} seq #

- may generate duplicate ACKs
- need only remember \texttt{expectedseqnum}

\textbf{out-of-order pkt:}

- discard (don’t buffer): \textit{no receiver buffering!}
- re-ACK pkt with highest in-order seq #
GBN in action

**sender window (N=4)**

<table>
<thead>
<tr>
<th>sender window (N=4)</th>
<th>sender</th>
<th>receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt0, send ack0</td>
<td>receive pkt0, send ack0</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt1</td>
<td>receive pkt1, send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt2</td>
<td>receive pkt3, discard, (re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt3, (wait)</td>
<td>receive pkt3, discard, (re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>rcv ack0, send pkt4</td>
<td>receive pkt4, discard, (re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>rcv ack1, send pkt5</td>
<td>receive pkt5, discard, (re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>ignore duplicate ACK</td>
<td>(re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>pkt 2 timeout</td>
<td>(re)send ack1</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt2</td>
<td>rcv pkt2, deliver, send ack2</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt3</td>
<td>rcv pkt3, deliver, send ack3</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt4</td>
<td>rcv pkt4, deliver, send ack4</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>send pkt5</td>
<td>rcv pkt5, deliver, send ack5</td>
</tr>
</tbody>
</table>

- GBN: Go Back N protocol
- N: Window size
- pkt: Packet
- ack: Acknowledgment
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
  - 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
Animation

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/
Selective repeat in action

sender window \((N=4)\)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

sender

send pkt0
send pkt1
send pkt2
send pkt3 (wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,

receive pkt4, buffer,

rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?
Selective repeat: dilemma

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

Q: what relationship between seq # size and window size to avoid problem in (b)?

sender window
(after receipt)

receiver window
(after receipt)

(a) no problem

receiver can’t see sender side.
receiver behavior identical in both cases!
something’s (very) wrong!

(b) oops!

will accept packet with seq number 0

timeout retransmit pkt0

will accept packet with seq number 0
Next class

- Please read Chapter 3.5-3.6 of your textbook BEFORE Class