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

Y. Richard Yang

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

11/6/2018
Admin.: PS4

- Part 1
  - Discussion checkpoint: Nov. 11; code checkpoint Nov. 13
- Part 2
  - Discussion checkpoint: Nov. 16; all due Nov. 27

proj-sol:

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

proj:

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

**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
Recap: Potential Channel Errors

Factors to pay attention when designing rdt

- Types of channel errors: Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).
  - bit errors
  - loss (drop) of packets
  - reordering or duplication

- Not only protocol but also analysis techniques
Recap: rdt2.0: Reliability allowing only Data Msg Corruption

sender

wait for data

\[\text{rdt\_send(data)}\]
\[\text{snkpkt = make\_pkt(data, checksum)}\]
\[\text{udt\_send(sndpkt)}\]

receiver

wait for data

\[\text{wait for ACK or NAK}\]

\[\text{rdt\_rcv(rcvpkt) && isNAK(rcvpkt)}\]
\[\text{udt\_send(sndpkt)}\]

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

\[\text{extract(rcvpkt, data)}\]
\[\text{deliver\_data(data)}\]
\[\text{udt\_send(ACK)}\]

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

\[\text{wait for data}\]
Recap: Rdt2.0 Analysis

Analyzing set of all possible execution traces is a common technique to understand and analyze many types of distributed protocols.

Execution traces of rdt2.0:

\{\text{data}^\wedge \text{NACK}\}^* \text{data deliver ACK}
Recap: rdt2.1b: Reliability allowing Data/Control Msg Corruption

sender

Fix wrong guess by checking seq#
Recap: Protocol Analysis using (Generic) Execution Traces Technique

A systematic approach to enumerating execution traces is to compute joint sender/receiver/channels state machine, and then convert the state machine to traces (or analyze properties on the machine directly).
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

Implication: One bit (the last bit) is enough to distinguish the two states
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, checksum)
  - udt_send(sndpkt)

- **rdt_rcv(rcvpkt) && (corrupt(rcvpkt)**
  - sndpkt = make_pkt(NAK, checksum)
  - udt_send(sndpkt)

- **rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)**
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- **rdt_rcv(rcvpkt) && (corrupt(rcvpkt)**
  - sndpkt = make_pkt(NAK, checksum)
  - udt_send(sndpkt)

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

**Receiver FSM Fragment**
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))`
- `sndpkt = make_pkt(ACK,1, checksum)`
- `udt_send(sndpkt)`

**Sender FSM Logic**
- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`

**Receiver FSM Logic**
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))`
- `sndpkt = make_pkt(ACK,1, checksum)`
- `udt_send(sndpkt)`

**Send Packet to Receiver**
- `udt_send(sndpkt)`

**Receive Packet from Sender**
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))`
- `udt_send(sndpkt)`

**Wait for ACK 0**
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)`

**Wait for Call 0 from Above**
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)`

**Extract Data**
- `extract(rcvpkt, data)`
- `deliver_data(data)`

**Send ACK 0**
- `sndpkt = make_pkt(ACK,0, checksum)`
- `udt_send(sndpkt)`

**End State**
- `Lambda`
Outline

- Admin and review
  - Reliable data transfer
    - perfect channel
    - channel with bit errors
    - channel with bit errors and losses
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: What can rdt2.2 go wrong under losses?
**rdt3.0 Sender**

```c
sendpkt = make_pkt(0, data, checksum)
udt_send(sendpkt)
start_timer

sendpkt = make_pkt(1, data, checksum)
udt_send(sendpkt)
start_timer

timeout
udt_send(sendpkt)
start_timer

timeout
udt_send(sendpkt)
start_timer
```

Diagram:

- **Wait for call 0 from above**
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)`
  - `stop_timer`

- **Wait for call 0 from above**
  - `rdt_send(data) && (corrupt(rcvpkt) || isACK(rcvpkt,0))`  
  - `Lambda`

- **Wait for ACK0**
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))`
  - `Lambda`

- **Wait for call 1 from above**
  - `timeout`
  - `udt_send(sendpkt)`
  - `start_timer`

- **Wait for call 1 from above**
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)`
  - `stop_timer`

- **Wait for ACK1**
  - `rdt_send(data)`
  - `Lambda`

- **Wait for call 1 from above**
  - `timeout`
  - `udt_send(sendpkt)`
  - `start_timer`

- **Wait for call 0 from above**
  - `rdt_send(sendpkt)`
  - `start_timer`

- **Wait for call 1 from above**
  - `rdt_send(sendpkt)`
  - `start_timer`
**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 exercises: (1) What are execution traces of rdt3.0? What are some state invariants of rdt3.0? (2) What are some state invariants?
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 thruput over 1 Gbps link
- network protocol limits use of physical resources!
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

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation
Pipelining: Increased Utilization

- First packet bit transmitted, \( t = 0 \)
- Last bit transmitted, \( t = \frac{L}{R} \)
- First packet bit arrives
- Last packet bit arrives, send ACK
- Last bit of 2\(^{nd}\) packet arrives, send ACK
- Last bit of 3\(^{rd}\) packet arrives, send ACK
- ACK arrives, send next packet, \( t = \text{RTT} + \frac{L}{R} \)

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

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

\[
\text{send\_base} \quad \text{nextseqnum} \quad \text{already} \quad \text{ack’ed} \quad \text{usable, not} \quad \text{sent} \\
\text{window size} \quad \text{sent, not} \quad \text{yet ack’ed} \quad \text{not usable}
\]

- 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
- \text{timeout(n)}: retransmit pkt n and all higher seq # pkts in window
GBN: Sender FSM

\[
\begin{align*}
&\text{rdt\_send(data)} \\
&\text{if (nextseqnum < base+W)} \\
&\quad \text{sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)} \\
&\quad \text{udt\_send(sndpkt[nextseqnum])} \\
&\quad \text{if (base == nextseqnum) start\_timer} \\
&\quad \text{nextseqnum++} \\
&\} \text{ else} \\
&\quad \text{block sender} \\
&\text{timeout} \\
&\text{start\_timer} \\
&\text{udt\_send(sndpkt[base])} \\
&\text{udt\_send(sndpkt[base+1])} \\
&\text{...} \\
&\text{udt\_send(sndpkt[nextseqnum-1])} \\
&\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)} \\
&\text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} \\
&\text{if (new packets ACKed)} \\
&\quad \text{advance base;} \\
&\quad \text{if (more packets waiting)} \\
&\quad \quad \text{send more packets} \\
&\} \\
&\text{if (base == nextseqnum)} \\
&\quad \text{stop\_timer} \\
&\text{else} \\
&\quad \text{start\_timer for the packet at new base}
\end{align*}
\]
**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
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
  - question: buffer size at receiver?

- 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

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

ACK3 rcvd, nothing 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# Size and Window Size

Example:
- seq #’s (2 bits): 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

Q: what relationship between seq # size and window size?

- **Go-back-n (GBN)**
  - Sender window
  - Receiver window

- **Selective repeat (SR)**
  - Sender window
  - Receiver window
## 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><strong>data bandwidth: sender to receiver</strong> (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><strong>ACK bandwidth</strong> (receiver to sender)</td>
<td>More efficient</td>
<td>Less efficient</td>
</tr>
<tr>
<td><strong>Relationship between M (the number of seq#) and W (window size)</strong></td>
<td>(M &gt; W)</td>
<td>(M \geq 2W)</td>
</tr>
<tr>
<td><strong>Buffer size at receiver</strong></td>
<td>1</td>
<td>(W)</td>
</tr>
<tr>
<td><strong>Complexity</strong></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# and When to Accept First Packet?
Question: What is Initial Seq# and When to Accept First Packet?

Discussion: Condition for receiver to deliver first data?
Outline

- Admin and recap
  - 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

**Host A**

- SYN(seq=x)

**Host B**

- notify initial seq#. Accept?

  - think of y as a challenge for *freshness*

  - accept data only after verified y is bounced back

  - x is the init. seq
Scenarios with Duplicate Request/SYN Attack

Host A

no such request

Host B

accept?

SYN(seq=x)

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

REJECT(seq=y)

reject
Scenarios with Duplicate Request/SYN Attack

Host A

no such request

SYN(seq=x)

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

ACK(seq=z)

REJECT(seq=y)

Host B

accept?

reject
Make “Challenge $y$” Robust

- 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!

Discussion: Potential approaches to close state?
Four Way Teardown

- can retransmit the ACK if its ACK is lost

- propose close A->B

A->B closed

- can retransmit the ACK if its ACK is lost

A->B closed

all states removed

Host A

Host B

A->B closed

all states removed

propose close B->A

closed