Reliable Transport Fundamentals

Daniel Zappala

CS 460 Computer Networking
Brigham Young University
How do you reliably send data across an unreliable network?
Components

- positive acknowledgements (ACKs) or negative acknowledgements (NACKs)
- timers
- selective or cumulative ACKs

⇒ in the context of three different protocols
Stop-and-Wait
RDT Roadmap

- **RDT 1.0**: the network is reliable
  - a transport protocol can be viewed as a finite state machine
- **RDT 2.0**: the network can cause bit errors
  - need a checksum to detect errors
  - send an ACK or NACK
  - retransmit data upon receiving a NACK
- **RDT 2.1**: ACKs, NACKs can be corrupted
  - retransmit when ACK or NACK is corrupted
  - need sequence numbers to detect duplicates – if an ACK is corrupted you’re re-sending data that the receiver already has
- **RDT 2.2**: eliminate NACKs
- **RDT 3.0**: network can also lose packets
  - need a timer in case packet or ACK lost
  - retransmit if timer expires before ACK received
RDT 3.0 Sender

- `rdt_send(data)`
- `sndpkt=make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_rcv(rcvpkt)`
- `& & notcorrupt(rcvpkt)`
- `& & isACK(rcvpkt,1)`
- `start_timer`
- `timeout`
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_rcv(rcvpkt)`
- `& & notcorrupt(rcvpkt)`
- `& & isACK(rcvpkt,0)`
- `stop_timer`
- `rdt_send(data)`
- `sndpkt=make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
RDT 3.0 Packet Trace

Sender: Send pkt0, rcv ACK0, send pkt1, rcv ACK1, send pkt0
Receiver: rcv pkt0, send ACK0, rcv pkt1, send ACK1, rcv pkt0, send ACK0

a. Operation with no loss

Sender: Send pkt0, rcv ACK0, send pkt1
Receiver: rcv pkt0, send ACK0, rcv pkt1, send pkt1, timeout, resend pkt1, rcv pkt1, send ACK1

b. Lost packet

Receiver: rcv pkt0, send ACK0
RDT 3.0 Packet Trace

c. Lost ACK

Sender:
- send pkt0
- rcv pkt0
- send pkt1
- timeout
- resend pkt1
- rcv pkt0
- send ACK0

Receiver:
- rcv pkt0
- send ACK0
- rcv pkt1
- send ACK1
- (loss) X
- rcv pkt1
- send ACK1
- rcv pkt1
- (detect duplicate)
- send ACK1
- rcv pkt0
- send ACK0

Sender:
- send pkt0

Receiver:
- rcv pkt0
- send ACK0
- rcv pkt1
- send pkt1
- timeout
- resend pkt1
- rcv pkt1
- send ACK1
- rcv pkt1
- (detect duplicate)
- send ACK1
- rcv pkt0
- send ACK0

d. Premature timeout
RDT 3.0 = Stop-and-Wait
Stop-and-Wait Performance

- example: 1 Gbps link, 15 ms propagation delay, 1000 byte packet
- calculate $U_{sender}$: utilization: fraction of time sender is busy sending
  - $T_{transmit} = \frac{L}{R} = \frac{8kb}{10^9 bps} = 8\mu s$
  - $U_{sender} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30.008} = 0.00027 = 0.027\%$
- 1000 bits every 30 ms = 33 kbps over a 1 Gbps link
- performance is lousy

$\Rightarrow$ transport protocol limits use of physical resources
Go-Back-N
Pipelining

- send multiple packets at a time, wait for ACKs
  - must increase sequence number space
  - need buffering at sender and receiver
Increased Utilization

- send 1000 packets at a time
- \( U_{\text{sender}} = \frac{1000 \times L/R}{RTT + L/R} = \frac{8}{38} = 0.21 = 21\% \)
Go-Back-N Overview

- sender keeps a *window* of packets
  - window represents a series of consecutive sequence numbers
  - window size $N$: number of un-ACKed packets allowed
- cumulative ACKs
  - $ACK(n)$: acks packets up to and including $n$
  - sender may receive duplicate acks
- go back $n$
  - sender keeps a timer for each packet
  - $timeout(n)$: retransmit packet $n$ and all higher packets
  - *no receiver buffering!*
Go-Back-N Sender FSM

```
rdt_send(data)
if(nextseqnum<base+N)
    ndpkt[nextseqnum]=make_pkt(nextseqnum,data,checksum)
    udt_send(ndpkt[nextseqnum])
if(base==nextseqnum)
    start_timer
    nextseqnum++
else
    refuse_data(data)
```

```
rdt_rcv(rcvpkt)& notcorrupt(rcvpkt)
```

```
rdt_rcv(rcvpkt)& corrupt(rcvpkt)
```

```
base=getacknum(rcvpkt)+1
if(base==nextseqnum)
    stop_timer
else
    start_timer
```
Go-Back-N Receiver

- cumulative ACK
  - always send ACK for in-order packet with highest sequence number
  - may generate duplicate ACKs
  - only state is expected sequence number

- out-of-order packets are discarded: no buffering
Go-Back-N Packet Trace

- warning: this will lead to congestion collapse
- bad reaction to congestion: send more packets!
- good reaction to congestion: slow down
- more on this when we visit TCP
Selective Repeat
Selective Repeat Overview

- sender keeps a *window* of packets
  - window represents a series of consecutive sequence numbers
  - window size $N$: number of un-ACKed packets allowed
  - $\Rightarrow$ same as Go-Back-N
- selective ACKs
  - $ACK(n)$: ACKS only sequence number $n$
- selective repeat
  - sender keeps a timer for each packet
  - $timeout(n)$: retransmit packet $n$ only
  - receiver must buffer out-of-order packets!
Selective Repeat Sender and Receiver Windows

**a. Sender view of sequence numbers**

- `send_base`
- `nextseqnum`
- Window size $N$

**b. Receiver view of sequence numbers**

- `rcv_base`
- Window size $N$

**Key:**
- Out of order (buffered) but already ACK’d
- Already ACK’d
- Sent, not yet ACK’d
- Usable, not yet sent
- Not usable
- Acceptable (within window)
- Expected, not yet received
- Not usable
Selective Repeat Sender

```python
1  def data():
2      if next available sequence number in window:
3          send packet
4
5  def timeout(n):
6      resend packet n
7      restart timer
8
9  def ACK(n):
10     if n not in [sendbase, sendbase + N]
11        return
12     mark packet n as received
13     if n smallest un-ACKed packet:
14        advance sendbase to next un-ACKed sequence number
15     if buffered packets can be sent:
16        send packets
```
**Selective Repeat Receiver**

```python
def packet():
    if n in [rcvbase, rcvbase + N - 1]:
        send ACK(n)
        if packet is out of order:
            buffer packet
        else:
            deliver all in-order packets
            advance rcvbase to next not-yet-received packet
    else if n in [rcvbase - N, rcvbase - 1]:
        send ACK(n)
    else:
        return
```
Selective Repeat Packet Trace

Sender

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 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

Receiver

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

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
Selective Repeat Window and Sequence Number Sizes

- example
  - 2-bit sequence number space
  - window size = 3
- receiver can’t tell difference between old and new packet 0
- how large should sequence space be?

Sender window (after receipt)

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
	timeout
retransmit pkt0
0 1 2 3 0 1 2

Receiver window (after receipt)

pkt0
pkt1
pkt2
ACK0 0 1 2 3 0 1 2
ACK1 0 1 2 3 0 1 2
ACK2 0 1 2 3 0 1 2

Sender window (after receipt)

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

Receiver window (after receipt)

pkt0
pkt1
pkt2
pkt3
pkt0
pkt

receive packet with seq number 0
receive packet with seq number 0

RFC 1323: TCP sequence space large enough for handling duplicates 3 minutes later
Selective Repeat Window and Sequence Number Sizes

- example
  - 2-bit sequence number space
  - window size = 3
- receiver can’t tell difference between old and new packet 0
- how large should sequence space be?
- sequence number size $\geq 2 \times$ window size
- RFC 1323: TCP sequence space large enough for handling duplicates 3 minutes later
TCP does not use Stop and Wait, Go-Back-N, or Selective Repeat
TCP

- this lecture shows you various design options
- the following lecture will explain how TCP implements reliability