Performance of RDT3.0 and of stop-and-wait in general

RDT3.0 works, but performance stinks
e.g.: 1 Gb/s link, 15 ms e2e prop. delay, 1KB packet:

\[ T_{\text{transmit}} = \frac{L}{R} = \frac{8\text{kb}}{10^{12}\text{ b/sec}} = 8 \text{ microsec} \]

\[ U_{\text{sender}} = \frac{L}{RTT + \frac{L}{R}} \]

\[ = \frac{0.008}{30.008} = 0.00027 \]

\[ U_{\text{sender}}: \text{utilization} – \text{fraction of time sender busy sending} \]

1KB packet every 30ms -> 33kB/sec throughput over 1 Gbps link

The protocol limits the use of physical resources!

†Assuming negligible time to transmit/receive ack, or this is absorbed into RTT.
RDT3.0: stop-and-wait operation

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

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets
  o range of sequence numbers must be increased
  o buffering at sender and/or receiver

• But if one frame gets errored, what happens to frames that don’t get errored that were transmitted after it, but before the error was detected? 2 approaches: Go-back-N and Selective repeat
• Multiple outstanding frames also introduces the possibility of sending too much for the receiver to buffer ⇒ flow control.
**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 2nd packet arrives, send ACK
- Last bit of 3rd packet arrives, send ACK

**Sender**

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

**Receiver**

*Increase utilization by a factor of 3!*
Outline
Types of acknowledgements

In addition to positive vs negative acknowledgements discussed earlier:

**Cumulative acknowledgements**: Indicate that everything up to, and including, a certain point has been received. If receiver receives something beyond that point, then it may either send nothing or resend a cumulative ack indicating the point to which it has received all information (latter is typical, e.g. TCP sends “dupacks” which when repeated act as a nack)

**Selective acknowledgements**: Indicate that a specific frame/info has been received. Other info transmitted earlier may not have been received.

**Numerical value of ack:**
- Theory: Sequence number matches that of received frame.
- Practice: (e.g. HDLC variants and TCP) Sequence number is that of the *next data unit expected*.

**Unit of ack**: Frames or bytes?
- Frames (acked by link layer protocols) impose less load on sequence number space
- Bytes (acked by TCP) allow different units of transmission and retransmission
Go-Back-N

Sender:
- k-bit seq # in packet header
- “window” of up to N, consecutive unack’ed packets allowed
- ACK(n): ACKs all packets up to, including seq # n - “cumulative ACK”†
- timer for each in-flight packet
- timeout(n): retransmit packet n and all higher seq # packets in window

† In the context of TCP, we’ll see what info can be gleaned from duplicate acks.
GBN: sender extended FSM

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

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

Wait

timeout

start_timer

udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum-1])

```
base=1
nextseqnum=1
```

```
rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
```
GBN: receiver extended FSM

ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **expectedseqnum**

- **out-of-order packet:**
  - discard (don’t buffer) -> no receiver buffering!
  - Re-ACK packet with highest in-order seq #
GBN in action

sender

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

receiver

rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
duplicate ack
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3
Outline
Selective Repeat

Receiver *individually*\(^\dagger\) acknowledges all correctly received packets
- buffers packets, as needed, for eventual in-order delivery to upper layer
Sender only resends packets for which ACK not received
- sender timer for each unACKed packet

Sender window
- \(N\) consecutive seq #’s
- again limits seq #’s of sent, unACKed packets

\(^\dagger\) Strictly, it need only be *able* to ack individual packets. When consecutive packets are received, it may be more efficient to ack these as a group. Send individual acks for packets received before predecessors.
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 packet

timeout(n):
  • resend packet n, restart timer

ACK(n) in [sendbase, sendbase+N]:
  • mark packet n as received
  • if n smallest unACKed packet, advance window base to next unACKed seq #

receiver:

packet n in [rcvbase, rcvbase+N-1]:
  • send ACK(n)
  • out-of-order: buffer
  • in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

otherwise:
  • ignore
Selective repeat in action

Individual ack

May instead send cumulative ack: ack all up to 5.
Sliding windows (Circular)

Sender

Initially

After sending first frame

After receiving first frame

After sender receives ack for first frame

Receiver

Shaded = info transmitted but not yet acked

Shaded = info willing to accept (enough buffer space)

T Fig. 3-13
Selective repeat: dilemma

Example:
- seq #’s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?
A: window <= (MAX_SEQ+1)/2
**Duplex data transfer**

The protocols examined so far provide unidirectional data transfer:

- Sender
  - Data
- Receiver
  - Control

\[= \text{header, e.g. framing & integrity check}\]

To provide duplex data transfer, can use:

- **a. 2 simplex protocols** operating independently
  - Node1
    - Data
    - Control
  - Node2
    - Data
    - Control

- **b. Share channels & extend header** to differentiate frame types
  - Node1
    - Data
    - Control
  - Node2
    - Data
    - Control

- **c. Piggyback** backward control with any forward data
  - Node1
    - Data
    - Control
  - Node2
    - Data
    - Control
Piggybacking

Advantages:
✓ **Save bandwidth:**
  - No framing & integrity check overheads for ACK – just another field added to the payload frame.
  - Saving is more pronounced at higher layers, where packets may have other overheads, e.g. large addresses and control fields
✓ **Fewer frames to process**
  - Fewer interrupts at end-systems
  - Lower packet processing overhead in routers (many routing functions, e.g. looking up address in routing tables, are independent of packet length)

Disadvantages:
✗ **Higher chance of ack being lost** since it’s carried in a longer packet. (May want separate header and payload error checks s.t. can process header when payload is corrupt.)
✗ **Acks may be delayed** while waiting for data to piggyback on
  - Source must factor this into its timeout
  - Receiver must run a timer and send ack alone if data is not forthcoming.

Source of image unknown
Lecture summary

- Stop-and-wait performs poorly when packet length $<<$ bandwidth-delay product of link.
- Sliding window protocols enable pipelining: Multiple packets can be propagating to the receiver (or acks from receiver) at any instant.
- After detecting loss:
  - Go-back-N retransmits from first lost packet onwards. Receiver need not resequence information.
  - Selective repeat retransmits only the lost packet(s). Saves bandwidth.
- Implementation issues:
  - Bounded sequence numbers handled by circular rotation of window.
  - Piggybacking acknowledgements with data saves bandwidth but costs delay.
Chapter 5
Data Link Layer

A note on the use of these ppt slides:
We’re making these slides freely available to all (faculty, students, readers). They’re in powerpoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:
- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we’d like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2002
J.F Kurose and K.W. Ross, All Rights Reserved