We have finished with reliability, now we consider how well reliable protocols perform.

—

Performance is not an issue with unreliable protocols as they can just blast away.

—

Reliable protocols have to wait until they receive ACKs before they can send new data.

—

Outline

- Fundamental transport layer services
  - Multiplexing/Demultiplexing
  - Error detection
  - Reliable data delivery
  - Pipelining
  - Flow control
  - Congestion control

- Service implementation in Internet transport protocols
  - UDP
  - TCP
RDT 3.0 works (ensures reliability) but it turns out that its performance stinks.

Let's first compute the numerator of the utilization equation.

Consider one end-system transmitting data under "RDT 3.0."

- Can one end-system fully use the network under RDT 3.0?
- How busy is the network under RDT 3.0?

\[
\text{utilization} = \frac{\text{time network busy}}{\text{observation interval}} = \frac{\text{time to transmit a packet}}{\text{packet generation time}}
\]

- How long does it take to transmit a 1,000 byte packet?

\[
\text{transmission time} = \frac{1 \text{ KB packet } \times 8 \text{ b/byte}}{10^6 \text{ bps}} = 8 \mu s
\]

- How fast can an end-system generate packets?

What's the best case?

- No data/ACK packets are lost or corrupted.
- All packets experience minimal transmission delay.

Assume for simplicity that ACK packets are the same size as data packets.

(In the end the time to transmit any size of ACK packet is going to be in the noise.)

- How fast can an end-system generate packets?
  - Packet transmission time = 8 \(\mu s\)
  - Propagation delay to receiver = 15 ms
  - ACK generation/transmission time = 8 \(\mu s\)
  - Propagation time for ACK to return to sender = 15 ms
- 1 packet every 30.016 ms
Ignoring for the moment the capabilities of the end-system, in a distributed system one would ideally like the transmission medium to be the performance bottleneck. If this is not the case then in some sense we are over paying for our network (we can’t use it all).

If performance is insufficient one just employs a faster link.

In particular, we don’t want maximal performance to be limited by the transport protocol.

What’s the problem?

- RDT is a stop-and-wait (an alternating-bit) protocol.

How can we do better?

- Somehow overlap the sending of packet \(k\) with the waiting for an ACK of packet \(k-1\) (i.e., pipeline transmissions).

Issues:

- More packets must be buffered because…?

At the sender:
The sender has to buffer all in-flight packets until they are ACKed.

At the receiver:
The receiver can only deliver packets in order to the application hence it has to potentially wait until all missing out-of-order packets are received.

What causes reordering in networks?

- Parallel processing in routers.
- Deep packet inspection of some packets by intrusion-detection devices.

There are two generic forms of pipelined protocols:

- Go-Back-N: Protocols that retransmit groups of packets whenever a packet is lost,
- Selective Repeat: Protocols that only retransmit the packet that is deemed to

Performance can be improved by allowing the sender to have multiple unacknowledged packets “in flight”.

Network protocols limit the use of physical resources!
Explain the figure!
This (and selective repeat) are known as sliding window protocols.

- The window slides forward (advances) as the ACK for the first packet in the window is received.

- For example, TCP is a sliding window protocol that uses 32-bit sequence numbers.

- No NAKs are used—cumulative ACKs are used instead.

- Why limit ourselves to only $N$ in-flight packets? Why not just blast away?
  - What if the receiver can receive more than $N$ packets at a time? (If everyone did this surely the network would become congested.)
  - How about a protocol that just blasts away until it gets a stop signal from the receiver?

How to signal loss?
- In RDT 3.0, at the receiver we generated duplicate ACKs when a packet was received out of order.
- In a sliding window protocol you only ever generate an ACK for the highest numbered in-order packet received.

- (But you always generate an ACK when a packet is received.)

How to detect loss?
- Exactly the same as before: either a timeout or a duplicate ACK.

Here, we have a simplistic loss adaptation:
- The sender retransmits all packets in the window.
- (Note that the packet that times out is always the first packet in the window.)
**Go-Back-n Protocol**

**Sender extended FSM**

```
rdt_send(data)
    if (nextseqnum < base+N) {
        compute chksum
        make_pkt(sndpkt[nextseqnum],nextseqnum,data,chksum)
        udt_send(sndpkt[nextseqnum])
        if (base == nextseqnum) start_timer
        nextseqnum += 1
    }
    else
        refuse_data(data)
```

- **wait for data/ACK/timer**

```
rdt_rcv(rcvpkt) & notcorrupt(rcvpkt)
    base := getacknum(rcvpkt) + 1
    if (base == nextseqnum) stop_timer
    else
        start_timer
```

- **wait for packet/timer**

**Receiver extended FSM**

```
default
    udt_send(sndpkt)
```

- **wait for packet/timer**

```
rdt_rcv(rcvpkt) & notcorrupt(rcvpkt) & has_seqnum(rcvpkt,expectedSeqNum)
    extract(rcvpkt,data)
    deliver_data(data)
    make_pkt(sndpkt,ACK,expectedSeqNum)
    expectedSeqNum += 1
    udt_send(sndpkt)
```

- **In-order packets processed, out-of-order packets discarded**
  - Sender will eventually timeout and retransmit out-of-order packets
  - Thus the receiver need not buffer any packets

- **Always send ACK for correctly-received packet with highest in-order sequence number**
  - May generate duplicate ACKs
  - But minimal state — need only remember expectedSeqNum

---

These are extended finite-state-machines in the sense that now we explicitly reference and operate on variables. The variables just represent more state that the protocol must maintain.

The sender waits for one of three events:

- **The application has data to send**
  - Start the timer only if this packet is the first one in the window (otherwise the timer must have been previously been set).
  - Note that there is only one timer and it always timing the ACK for the oldest in-flight packet.
  - Note also that if a window's worth of data is in-flight then the application's send call will either be blocked or the OS will buffer the data and wait to send it.

- **The timer goes off**
  - A packet is assumed to be lost, retransmit the current window contents.

The receiver is significantly more simple.

- No buffering and 1 state variable (the next expected packet number).
  - (What are the state requirements at the sender? Just the starting sequence number of the window and the current sequence number.)

- The default transition is taken for all other event occurrences (e.g., an out-of-order arrival or a duplicate packet).

There's no need to buffer as we discard all out-of-order packets.

- If the receiver discards out-of-order packets without ACKing them (but generating ACKs), the sender will eventually timeout waiting for an ACK.
  - At this point the sender will retransmit all packets in the window.

Note that ACKing out-of-order packets (as is done here with cumulative ACKs) delays the timeout and hence the detection of loss.

- The simplest thing to do would be to not even generate cumulative ACKs in this case (i.e., do nothing in the default case).

(Are we concerned about lost ACKs? — No! They'll just timeout!)
Here the window would advance after the receipt of the first two ACKs.

—

And after each of these ACKs, the timer would be restarted to time the receipt of the next expected ACK.

At the receiver, all packets that arrive after packet 1 are out-of-order and hence are discarded.

—

But ACKs for the highest numbered packet received in-order (packet 1 in this case) are generated.

Eventually packet 2 (the first packet in the current window) times out.

—

The timer is set when the ACK for packet 0 is received.

—

And reset when the ACKs for packets 3 and higher are received.

All packets in the window are retransmitted.

Before, the key issue was that an end-system couldn't transmit packets very fast under a stop-and-wait protocol.

We had to wait a round-trip-time before sending a next packet.

Now we can have a window size of packets before we have to receive an ACK (within a round-trip-time).

Assume a window size of 4 packets

Receiver ignores out-of-order packets

Sender retransmits only on timeout

(Duplicate ACKs now have no effect)

Transport Protocol Performance

Performance of Go-Back-n protocols

Can an end-system make more efficient use of a network under a Go-Back-n protocol?

Consider again transmitting 1,000 byte packets on a 1 Gbps link with 15 ms end-to-end propagation delay

\[
\text{utilization} = \frac{\text{time to transmit a packet}}{\text{packet generation time}}
\]

\[
\text{transmission time} = \frac{1 \text{ kB packet} \times 8 \text{ b/byte}}{10^6 \text{ bps}} = 8 \mu s
\]

How fast can an end-system transmit packets?

» Depends on the window size!
This example is the same as the one from the beginning of the lecture hence the numbers for packet generation and transmission time are the same.

The first bit of the first packet is received at time 15 ms, however the packet itself is not fully received until time 15.008 ms. Hence the first ACK is not generated until time 15.008 ms.

This example assumes the transmission time for the ACK is 8 µs (which is bogus).

N packets are sent every 30.016 ms. This value does not depend on N.

This example assumes that:

- propagation delay > N \times (transmission delay + time for the application to generate a packet)

(a valid assumption for high-speed links).

---

**Transport Protocol Performance**

**Performance of Go-Back-n protocols**

```c
rtt_send(data)
if (nextseqnum < base+N) {
    compute checksum
    make_pkt(sndpkt[nextseqnum],nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum) start_timer
    nextseqnum ++
}
```

- How fast can an end-system transmit packets?
  - N packets can be sent before the sender must wait for an ACK
- N packets sent every 30.016 ms
  - Packet generation/transmission time = 8 µs
  - Round-trip-time to receiver = 30 ms
  - ACK generation/transmission time ≈ 8 µs
**Transport Protocol Performance**

**Performance of Go-Back-n protocols**

- Performance with a window size of $N = 64$ packets:

  \[
  \text{utilization} = \frac{\text{time to transmit } N \text{ packets}}{\text{time to receipt of first ACK}} = \frac{512 \text{ ms}}{30.016 \text{ ms}} = 1.7% 
  \]

  A 64x improvement!

- Is this good?
  - 64 1,000 byte packets every 30 ms results in (maximum) throughput of 17 Mbps over a 1 Gbps link!

---

**Pipelined Protocols**

**“Selective Repeat” protocols**

- Receiver *individually* acknowledges all correctly received packets
  - Buffers packets as needed for eventual in-order delivery to upper layer
- Sender only resends packets for which an ACK has not been received
  - Sender maintains a timer for each unACK’ed packet
- Sender window is the same as before
  - $N$ consecutive sequence numbers
    (Limits the sequence numbers of sent, unACK’ed packets)
Now there is a fifth packet type: Sent & ACKed but still in window.

In selective repeat protocols, sender’s and receiver’s windows are not guaranteed to be the same.

Was this the case in Go-Back-N protocols? (Check!)

How can they be out of sync?

Lost ACKs!

The state machine for the sender again has only a single state with three state transitions.

Again, each packet now has its own logical timer.

This is because only one packet will be retransmitted on a timeout.

The timer is only a logical timer because it is possible to simulate the effect of multiple timers with only a single one.

Call from above:

- If next available sequence number is within window, send the packet and start a timer for it
- Timeout for packet $n$:
  - Resend packet $n$, restart timer for packet $n$
  - ACK received for packet with sequence number $n$:
    - If $n$ in $[sendBase, sendBase+N-1]$ then mark packet $n$ as received
    - If $n == sendBase$, advance sendBase to next highest unACKed sequence number and move the window forward by that amount
The state machine for the receiver also has only a single state with three state transitions.

- Note that $rcv_{base}$ need not equal $send_{base}$.

- How can we receive a packet that $s$ already outside our window?
  - This is a duplicate packet whose original ACK was lost or delayed to the point that the sender timed out and retransmitted.
  - Note that even though this packet has been previously received and delivered to the application (it's outside our window), it still must be ACKed or else the sender will continually resend.
  - The limit as to how old a duplicate packet can be is the limit on the skew possible between sender and receiver windows (which is $N$ packets).

- A packet arriving with a sequence number greater than $rcv_{base} + N$ is assumed to be an error.

This is the same example as before.

- Sender has a window size of 4 packets and the second packet is lost.
- When the retransmitted packet 2 arrives, the sender's window jumps forward by 4 sequence numbers.
  - (Since four packets are delivered to the application in one operation.)
- Selective Repeat is efficient.
  - 1 lost packet, 1 retransmission.

Compare the effect of a lost ACK in our SR protocol and our GBN protocol.
The potential for a lack of synchronization between a sender's and receiver's windows can lead to problems if the sequence number space is too small.

Consider a protocol instance with 4 sequence numbers and a window size of 3.

Assume two stylized (contrived) patterns of loss.

- Case 1: A window's worth of ACKs are lost.
- Case 2: The first data packet in a new window is lost.

The receiver can't tell the difference between these two cases!
- Is the packet with sequence number 0 a retransmission or an out-of-order arrival?

How many sequence numbers do we need?
- As many as the largest number of packets that can be in flight?

If the sequence number space is close to the window size then the receiver can get confused.

Theorem: At least twice as many as the largest window size.