Primitives for Achieving Reliability

3035/GZ01 Networked Systems
Kyle Jamieson

Department of Computer Science
University College London
Two fundamental problems in networking:

1. *How can two entities communicate reliably over a medium that may lose or corrupt data?*

2. *How can we increase the performance of reliable communication?*
Achieving reliability: the story so far

1. Apply **forward error correction (FEC)** at the physical layer
   - Corrects *some* errors from the unreliable channel

2. Apply **error detection** algorithms at higher layers
   - *e.g.* Link layer CRC, Internet checksum (IC)
   - Detects errored frames remaining after FEC with high reliability

- The story so far: discard the errored frames, and then...?
Achieving reliability: A first try with coding

• Let’s first try to make reliable links using the tools we have so far (FEC and error detection)

• Checksums: 32-bit CRC and IP checksums together detect most errors—we’ll discard those frames

• How much FEC should we add to correct all errors?
  – Really care about how often frames are lost (frame loss rate)
  – Relationship between BER and frame loss rate (FLR)?
The relationship between FLR and BER

• Suppose after FEC, probability of bit error is $BER$

• $Pr[\text{Frame of length } n \text{ bits delivered}] = (1 - BER)^n$
  – Assumption: Independent bit errors (worst case)
  – Frame loss rate ($FLR$): $1 - Pr[\text{Frame delivered}]

If we added enough parity bits to lower BER such that FLR became vanishingly small, we’d waste a lot of capacity!
Some errors are too severe to be corrected

- No matter how many parity bits we add, the network could flip them and data bits, causing errors!
  - Error detection (CRC) then discards these frames

How to ensure that links are reliable?

Basic idea: Sender applies some FEC; receiver uses error detection, asks for re-sends
New strategy for FEC

- Add enough FEC to keep FLR below the knee, **but no more** (wastes bits on the channel): typically, pick FLR < $10^{-3}$

- Where is the knee, for a given packet size?
  - For small $x$, expansion of $(1+x)^n = 1 + nx + O(x^2) \approx 1 + nx$
  - $FLR = 1 - (1 - BER)^n \approx n \times (BER)$, so keep $n \times (BER) < 10^{-3}$

- Therefore, for data packet of 1250 bytes, add enough FEC to keep BER < $10^{-7}$

```plaintext
\text{Bit error rate (BER)}
```

```plaintext
\text{FLR}
```

```plaintext
n = 10^4 \text{ (1250 bytes)}
```

```plaintext
n = 320 \text{ (40 bytes)}
```
Two fundamental problems in computer networking:

1. How can two entities communicate reliably over a medium that may lose or corrupt data?
   - The stop-and-wait protocol

2. How can we increase the performance of reliable communication?
So far we’ve been casually using the term “reliable.” *So what exactly is reliable delivery?*

An *abstraction* where:

1. No packets submitted to the protocol are **corrupted**
2. All packets submitted are **delivered**
3. All packets are delivered in the **order** they were submitted

```
rdt_send(data)                         deliver_data(data)
```

Reliable channel
Reliable transfer protocol

- Design sender and receiver sides of a *reliable data transfer* (rtt) protocol, using *unreliable data transfer* (udt) protocol

- Recurs at many different layers:
  - Reliable transport layer over an unreliable network layer
  - Reliable link layer over an unreliable physical layer

```
rdt_send(data)
udt_send(pkt)
rdt_recv(pkt)
deliver_data(data)
```
Approach

• Let’s derive a protocol that achieves reliable transfer from first principles

• **Goal:** exactly once, in-order, correct delivery of every packet

• Unidirectional at first; same principles can generalize to bidirectional data transfer

• **Starting assumptions:**
  1. Channel can **not** introduce bit errors into packets
  2. Channel can **not** fail to deliver packets
  3. Channel can **not** delay packets
  4. Channel can **not** reorder packets

• Gradually relax these assumptions, one by one
Reliable data transfer (rdt) v1.0

- Independent state machines at sender, receiver
- No need for feedback (underlying channel is reliable)

Sender state machine:

- IDLE
  - `rdt_send(data)`
  - `packet = make_pkt(data)`
  - `udt_send(packet)`

Receiver state machine:

- Data wait
  - `rdt_recv(pkt)`
  - `data = extract_data(pkt)`
  - `deliver_data(data)`
Assumptions

1. Channel can introduce bit errors into packets
   • Channel (sender to receiver) can introduce bit errors
     – Channel (receiver to sender) can not introduce bit errors

2. Channel can not fail to deliver packets

3. Channel can not delay packets

4. Channel can not reorder packets
Simple idea: Receiver asks for re-sends

- Three fundamental mechanisms:
  1. **Error detection**: typically a packet checksum (*e.g.* CRC)
  2. **Acknowledgements**: small control frame (**ACK**) transmitted from the receiver back to the sender
     - Positive ACKs **acknowledge correct receipt**
     - Negative ACKs **inform sender of incorrect receipt**
  3. **Timeouts**: sender waits a reasonable amount of time, then retransmits the frame

- Protocols using these techniques are called **automatic repeat request (ARQ)** protocols
  - Surprisingly challenging to apply correctly!
Reliable data transfer under bit errors

**rdt v2.0 sender state machine:**

<table>
<thead>
<tr>
<th>rdt_send(data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sndpkt = make_pkt(data, checksum)</td>
</tr>
<tr>
<td>udt_send(sndpkt)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rdt_recv(rcvpkt) &amp;&amp; isACK(rcvpkt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(do nothing)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rdt_recv(rcvpkt) &amp;&amp; isNACK(rcvpkt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>udt_send(sndpkt)</td>
</tr>
</tbody>
</table>

**rdt v2.0 receiver state machine:**

<table>
<thead>
<tr>
<th>rdt_recv(pkt) &amp;&amp; (verify_checksum(rcvpkt) == false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sndpkt = make_pkt(NACK)</td>
</tr>
<tr>
<td>udt_send(sndpkt)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rdt_recv(pkt) &amp;&amp; (verify_checksum(rcvpkt) == true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data = extract_data(pkt)</td>
</tr>
<tr>
<td>deliver_data(data)</td>
</tr>
<tr>
<td>sndpkt = make_pkt(ACK)</td>
</tr>
<tr>
<td>udt_send(sndpkt)</td>
</tr>
</tbody>
</table>
Reliable data transfer v2.0 analysis

- **Stop-and-wait protocol**: Sender doesn’t send more data until sure original data received
  - Performance depends on sender-receiver delay, and throughput of link

- **Correctness**:
  1. Data arrives okay
     - ACK returns immediately
     - **Sender sends next packet**
  2. Data arrives corrupted
     - NACK comes back immediately
     - **Sender retransmits corrupted packet**

- **Exactly once, in-order delivery**
Assumptions

1. Channel can introduce bit errors into packets
   - Channel (sender to receiver) can introduce bit errors
   • Channel (receiver to sender) can introduce bit errors

2. Channel can **not** fail to deliver packets

3. Channel can **not** delay packets arbitrarily

4. Channel can **not** reorder packets
Human approach to feedback errors

- One possibility: Apply ARQ in reverse
  - **Sender** requests retransmissions of corrupted feedback (ACK/NACK) from receiver

- If the sender’s packets are corrupted, receiver won’t know if they are data or feedback retransmit requests

- Clearly heading down a difficult path!
Idea: Add checksums to feedback

**rdt v2.0 (with checksums) sender state machine:**

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt_send(data)</td>
<td>sndpkt = make_pkt(data, checksum)</td>
</tr>
<tr>
<td></td>
<td>udt_send(sndpkt)</td>
</tr>
</tbody>
</table>

**rdt v2.0 (with checksums) receiver state machine:**

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt_recv(pkt) &amp; (checksum_ok(rcvpkt) == false)</td>
<td>sndpkt = make_pkt(NACK, checksum)</td>
</tr>
<tr>
<td></td>
<td>udt_send(sndpkt)</td>
</tr>
<tr>
<td>rdt_recv(pkt) &amp; (checksum_ok(rcvpkt) == true)</td>
<td>data = extract_data(pkt)</td>
</tr>
<tr>
<td></td>
<td>deliver_data(data)</td>
</tr>
<tr>
<td></td>
<td>sndpkt = make_pkt(ACK, checksum)</td>
</tr>
<tr>
<td></td>
<td>udt_send(sndpkt)</td>
</tr>
</tbody>
</table>
Problem for rdt v2.0: duplicates

• Three cases:

1. Data okay, ACK okay

2. Data corrupted
   – NACK comes back immediately
   – Sender resends previously-corrupted data packet

3. Data okay, receiver’s ACK is corrupted
   – ACK comes back immediately
   – Sender retransmits an identical data packet

• Now we have at least once, in-order delivery of data
Assumptions

1. Channel can introduce bit errors into packets
   - Channel (sender to receiver) can introduce bit errors
   - Channel (receiver to sender) can introduce bit errors

2. Channel can **not** fail to deliver packets

3. Channel can **not** delay packets arbitrarily

4. Channel can **not** reorder packets

5. **Sender or channel can duplicate packets**
From at least once to exactly once

- **Idea**: add sequence numbers to data packets

- Sequence number allows receiver to suppress duplicates

rdt v2.1 sender state machine:

- **Idle 0**: Send data packet with seqno=0
- **ACK wait 0**: Resend seqno=0 data packet
- **Receive corrupt packet or NACK**: Resend seqno=0 data packet
- **Receive ACK with checksum okay**: (do nothing)

- **Idle 1**: Send data packet with seqno=1
- **ACK wait 1**: Resend seqno=1 data packet
- **Receive corrupt packet or NACK**: Resend seqno=1 data packet
- **Receive ACK with checksum okay**: (do nothing)
From at least once to exactly once

- **Two states at receiver:** one for expecting sequence number one; the other for expecting sequence number zero

### rdt v2.1 receiver state machine:

- **Data wait 0**
  - Receive data
  - Receive corrupt packet
  - Send NACK

- **Data wait 1**
  - Receive data
  - Receive corrupt packet
  - Send NACK

- **Deliver data to upper layer**
  - Send ACK
**rdt v2.1 error-free operation**

- **Sender**: send data, wait for reply
- **Receiver**: deliver data, send ACK
- **Sender**: send next data, wait
- **Receiver**: deliver data, send ACK
- **Sender**: transition to Idle State 0

---

**Sender state machine**:

- **I0**: Data (seqno=0), ACK
- **AW0**: Data (seqno=0), DW 0
- **I1**: ACK, DW 1
- **AW1**: Data (seqno=1), DW 0
- **I0**: ACK, DW 0

**Receiver state machine**:

- **DW 0**: A\(\uparrow\), N
- **DW 1**: A\(\uparrow\), A
rdt v2.1: bit errors on the forward link

- Sender: send data, wait for reply
- Receiver: checksum fails; NACK
- Sender: **resend** seqno=0 data
- Receiver: deliver data; send ACK
- Sender: transition to Idle State 1
**rdt v2.1: bit errors on the reverse link**

- **Sender:** send data, wait for reply
- **Receiver:** deliver data; send ACK
- **Sender:** **resend** seqno=0 data
- **Receiver:** **dup. data**; resend ACK
- **Sender:** transition to Idle State 1

---

**Sender state machine:**

- **I0** -> **AW0**
- **AW0** -> **I1**
- **I1** -> **AW0**

**Receiver state machine:**

- **DW 0** -> **DW 1**
- **DW 1** -> **DW 0**

---

**Sender Diagram:**

```
I0  Data, seqno=0  DW 0
AW0  ACK  DW 1
AW0  Data, seqno=0
I1  ACK
```

**Receiver Diagram:**

```
DW 0
A↑
DW 1
```

---

**State Diagrams:**

- **Sender State Machine:**
  - I0: send data
  - AW0: wait for reply
  - AW1: resend seqno=0 data
  - I1: send ACK
- **Receiver State Machine:**
  - DW 0: deliver data
  - DW 1: send ACK
  - DW 0: resend ACK
  - DW 1: transition to Idle State 1
rdt v2.2: Getting rid of NACKs

- rdt v2.1 used different packets in feedback direction (ACK, NACK)
- Instead, we can add sequence numbers to ACKs
  - Sequence number in ACK equals sequence number of last correctly-received data

Sender

<table>
<thead>
<tr>
<th>I0</th>
<th>AW0</th>
<th>Data, seqno=0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ACK 1</td>
</tr>
</tbody>
</table>

Receiver

<table>
<thead>
<tr>
<th>DW 0</th>
<th>DW 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK 0</td>
<td>DW 1</td>
</tr>
</tbody>
</table>

Sender state machine:

```
I0 → AW0 ❌ 0
AW0 → DW 0 ❌ 0
```

Receiver state machine:

```
DW 0 → A0 1
A0 → DW 1 A1
```

Sender

```
I0 → AW0 ❌ 0
AW0 → DW 0 ❌ 0
```

Receiver

```
DW 0 → A0 1
A0 → DW 1 A1
```
**Problem:** These protocols can’t handle dropped packets

**Sender:** transmit data, wait for a reply from the receiver

**Result:** Deadlock
Assumptions

1. Channel can introduce bit errors into packets
   - Channel (sender to receiver) can introduce bit errors
   - Channel (receiver to sender) can introduce bit errors

2. Channel can fail to deliver packets

3. Channel can delay packets arbitrarily

4. Channel can not reorder packets

5. Sender or channel can duplicate packets
Sender retransmits to break deadlock

**rdt v3.0 sender state machine:**

- **Idle 0**
  - Upper layer send call
  - Send data seqno=0, start timer
  - (timeout)
  - Resend data, start timer

- **ACK wait 0**
  - Receive corrupt packet or ACK 1
    - (do nothing)
  - Receive ACK 0
    - stop timer
  - receive anything
    - (do nothing)

- **ACK wait 1**
  - Receive corrupt packet or ACK 0
    - (do nothing)
  - (timeout)
  - Resend data, start timer

- **Idle 1**
  - Upper layer send call
  - Send data seqno=1, start timer
  - receive anything
    - (do nothing)

Timer start

- 0,T+
- 0,T+
- T–
- T–

Timer stop

- 0,T+
- 1,T+
- 1,T+
- 1,T+
rdt v3.0 receiver

rdt v3.0 receiver state machine:

- **Data wait 0**
  - Receive corrupt data or seqno=1
    - Resend ACK 1
  - Receive data with seqno=0, checksum okay
    - Deliver data to upper layer; Send ACK 0

- **Data wait 1**
  - Receive corrupt data or seqno=0
    - Resend ACK 0
  - Receive data with seqno=1, checksum okay
    - Deliver data to upper layer; Send ACK 1
rdt v3.0: recovering lost packets

- Sender: send, start timer, wait/reply
- Receiver: deliver; send ACK 0 (lost)
- Sender: timeout, **resend**, start timer
- Receiver: **dup. data**; resend ACK 0
- Sender: stop timer, go to Idle State 1
rdt v3.0: delays in the network

- Sender: send, start timer, wait/reply
- Receiver: deliver; send ACK (delayed)
- Sender: timeout, resend, start timer
- Sender: stop timer, go to Idle State 1
- Receiver: dup. data, resend ACK 0
- Sender: recv. dup. ACK 0, do nothing
Two fundamental problems in computer networking:

1. *How can two entities communicate reliably over a medium that may lose or corrupt data?*

2. *How can we increase the performance of reliable communication?*
Stop-and-and-wait performance

Data packet size $L$ bits, link bitrate $R$ bits/second

First bit transmitted, $t = 0$
Last bit transmitted, $t = L/R$

Round trip time (RTT)

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

First bit arrives
Last bit arrives, send ACK

Link utilization: $U_{sender} = \frac{L/R}{RTT + L/R}$
Performance of stop-and-wait protocols

- Packet size $L$, link bitrate $R$; utilization $U_{\text{sender}} = \frac{L/R}{\text{RTT} + L/R}$

- Internet *e.g.*: 8000 bit packet; 1 Mbit/s link, 30 ms RTT:
  
  $U_{\text{sender}} = \frac{L/R}{\text{RTT} + L/R} = \frac{8 \text{ ms}}{30 \text{ ms} + 8 \text{ ms}} = 21\%$

- WiFi *e.g.*: 8000 bit packet; 54 Mbit/s link, 100 ns RTT
  
  $U_{\text{sender}} = \frac{L/R}{\text{RTT} + L/R} = \frac{148 \mu\text{s}}{100 \text{ ns} + 148 \mu\text{s}} = 99.93\%$

When packet time $\ll$ RTT, stop-and-wait underperforms.
Abandon stop-and-wait for small link rate, large RTT

**Pipelining**: sender allows multiple unacknowledged packets “in-flight” from sender to receiver
- Need to increase range of sequence numbers
- Need to buffer packets at sender and/or receiver

**Idea: Pipelined protocols**

- A stop-and-wait protocol in operation
- A pipelined protocol in operation
Increasing utilization with pipelining

Data packet size $L$ bits, link bitrate $R$ bits/second

First bit sent, $t = 0$
Last bit sent, $t = L / R$

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

last bit of 1st packet, send ACK
last bit of 2nd packet, send ACK
last bit of 3rd packet, send ACK
The bandwidth-delay product

Data packet size $L$ bits, link bitrate $R$ bits/second

• **How many packets $N$ do we need to be “in flight” in order to get maximum link utilization?**

$$U_{\text{sender}} = \frac{N \cdot L / R}{\text{RTT} + L / R} = 1$$

$$\left( N - 1 \right) L = \text{RTT} \cdot R$$

Number of bits “in flight”  Delay $\times$ Bandwidth product
Today

Two fundamental problems in computer networking:

1. *How can two entities communicate reliably over a medium that may lose or corrupt data?*

2. *How can we increase the performance of reliable communication?*
   - Exploiting pipelining: The Go-Back-N Protocol
   - The Selective Retransmit Protocol
Assumptions

1. Channel can introduce bit errors into packets
   – Channel (sender to receiver) can introduce bit errors
   – Channel (receiver to sender) can introduce bit errors

2. Channel can fail to deliver packets

3. Channel can delay packets arbitrarily

4. Channel can reorder packets in forward direction

5. Sender or channel can duplicate packets
The Go-Back-\(N\) (GBN) protocol

- **Exploits pipelining** available in the network
  - Up to \(N\) unacknowledged packets “in flight”

- **Sender**: send without waiting for an acknowledgement
  - Timer for **oldest unacknowledged packet**
  - On timer expire: retransmit all unacknowledged packets (sender can “go back” up to \(N\) packets)

- **Receiver**: sends **cumulative acknowledgement**: acknowledge receipt of all packets up to and including packet \(n\): **ACK(n)**
**GBN: At the sender**

### rdt_send(data):
- if nextseqnum < send_base + N:
  - sndpkt[nextseqnum] = data
  - send(sndpkt[nextseqnum])
  - nextseqnum++
- else:
  - (refuse data)

### timeout:
- send(sndpkt[send_base])
- send(sndpkt[send_base+1])
  - send(sndpkt[nextseqnum−1])

### rdt_recv(ackpkt):
- send_base = ackpkt.seqno + 1

(Timer code not shown)
GBN: At the receiver

• Maintain **expectedseqnum** variable: sequence number of the next in-order packet
  – $\text{send}_\text{base} \leq \text{expectedseqnum} < \text{nextseqnum}$

• Incoming data packet with $\text{seqno} = n$
  – $n = \text{expectedseqnum}$: send ACK($n$), increment $\text{expectedseqnum}$
    – $n \not= \text{expectedseqnum}$: discard packet, send ACK($\text{expectedseqnum} - 1$)

• Nothing more, because in the event of loss, the sender will go back to $\text{expectedseqnum}$!

• Receiver could buffer out-of-order packets, but the sender will transmit them later anyway, so don’t buffer
GBN ($N = 4$) in operation

Sender

| Send 0 | Send 1 | Send 2 | Send 3 (wait) | Receive ACK 0; send 4 | Receive ACK 1; send 5 (wait) | Timeout; resend 2 | Resend 3 | Resend 4 | Resend 5 |

Receiver

| Receive 0; send ACK 0 | Receive 1; send ACK 1 | Receive 3; discard; send ACK 1 | Receive 4; discard; send ACK 1 | Receive 5; discard; send ACK 1 | Receive 2; send ACK 2 | Receive 3; send ACK 3 | Receive 4; send ACK 4 | Receive 5; send ACK 5 |
The role of the retransmission timer

• Keeps track of time at sender since the **oldest unacknowledged packet** was sent
  – This is the packet at the left edge of the sender’s window

• Issue: Choosing a reasonable timer value
  – Compared to the RTT:
    – Too small causes unnecessary retransmissions
    – Too big wastes time

• Later: we will see how TCP solves this problem robustly
Two fundamental problems in computer networking:

1. How can two entities communicate reliably over a medium that may lose or corrupt data?

2. How can we increase the performance of reliable communication?
   - Exploiting pipelining: The Go-Back-N Protocol
   - Better error recovery: The Selective Retransmit Protocol
Selective Repeat: Introduction

• Go-Back-N
  – Allows pipelining, for high channel utilization
  – Receiver sends *cumulative* acknowledgements ACK(n) acknowledging packet n and all those before
  – Large pipeline $\Rightarrow$ a single packet error results in many duplicate packet transmissions

• Selective Repeat (SR)
  – Main idea: sender retransmits only packets that don’t reach the receiver correctly
  – Receiver *individually* acknowledges each packet

• GBN, SR, and later TCP are all *sliding window* protocols
Finite sequence number space

- Now we’ll use just a **k-bit** sequence number per packet
- Sequence number range: \([0, 2^k - 1]\)
- All arithmetic is modulo \(2^k\): wrap-around the end
- Used in practice for both GBN and SR
Selective Repeat: Sequence numbers

- **Window size** $N$ limits number of unacked packets in pipeline
- **send_base**: lowest unacked packet
- **nextseqnum**: last sent packet sequence number, plus 1
- **rcv_base**: last frame delivered, plus 1
SR sender: Data received from above

Sender’s view:
- `send_base`:
  - Already ACKed
  - Sent, not yet ACKed
  - Usable, not yet sent
  - Not usable

Receiver’s view:
- `rcv_base`:
  - Out of order but ACKed
  - Acceptable
  - Expected, not yet received
  - Not usable

Window size $N$

- Sender checks `nextseqnum`:
  - If in sender window, transmit, increment `nextseqnum`
  - If not in sender window, refuse data from above
SR sender: Timeout event

- Each packet has its own retransmission timer
- Only packets’ timers in send window will be active
- Sender retransmits packet then restarts that packet’s timer
SR receiver: packet reception

Sender’s view:

Window size $N$

Receiver’s view:

• Correct packet reception with received seqno $r$. Three cases:
  1. seqno $r$ in receiver window: deliver data, send ACK, advance window
  2. seqno $r$ in $[\text{rcv_base} - N, \text{rcv_base} - 1]$: resend ACK
  3. Otherwise: ignore the packet
1. Sender marks packet as already ACKed, stops retransmit timer
2. Sender compares send_base and ACK sequence number:
   – If ACK is not for send_base, keep window where it is
   – If ACK is for send_base, advance send_base and send window by one
SR \((N = 4)\) in operation

**Sender**
- Send 0
  - [0 1 2 3] 4 5 6 7 8 9
- Send 1
  - [0 1 2 3] 4 5 6 7 8 9
- Send 2
  - [0 1 2 3] 4 5 6 7 8 9
- Send 3
  - [0 1 2 3] 4 5 6 7 8 9
  - Recv ACK 0, send 4
  - 0 [1 2 3 4] 5 6 7 8 9
  - Recv ACK 1, send 5
  - 0 [1 2 3 4 5] 6 7 8 9
  - Timeout 2, resend 2
  - 0 [1 2 3 4 5] 6 7 8 9
  - Recv ACK 3, send –
  - 0 [1 2 3 4 5] 6 7 8 9

**Receiver**
- Recv 0, deliver, ACK 0
  - 0 [1 2 3 4] 5 6 7 8 9
- Recv 1, deliver, ACK 1
  - 0 1 [2 3 4 5] 6 7 8 9
- Recv 3, buffer, ACK 3
  - 0 1 [2 3 4 5] 6 7 8 9
- Recv 4, buffer, ACK 4
  - 0 1 [2 3 4 5] 6 7 8 9
- Recv 5, buffer, ACK 5
  - 0 1 [2 3 4 5] 6 7 8 9
- Recv 2, deliver 2–5, ACK 2
  - 0 1 2 3 4 5 [6 7 8 9]
• How to choose $k$ ($2^k$: the sequence number space size), and $N$ (the window size)?
  – A larger window size ($N$) allows us to **utilize links with larger bandwidth-delay products**
  – A smaller $k$ allows us to allocate less space in each header for the sequence number, **reduces protocol overhead**

• **Therefore:** In general, want to increase $N$, reduce $k$

• **Are there any adverse effects in doing so?**
Sequence number ambiguity

$N = 3, k = 2$ (four possible sequence numbers)
Preventing the ambiguity

- $N > 2^{k-1}$: window too large
  - Right edge of receiver’s window can wrap past left edge of sender’s window
- $N \leq 2^{k-1}$: no overlap
Two fundamental problems in computer networking:

1. *How can two entities communicate reliably over a medium that may lose or corrupt data?*

2. *How can we increase the performance of reliable communication?*
   - Exploiting pipelining: The Go-Back-N Protocol
   - Better error recovery: The Selective Retransmit Protocol

- These concepts will culminate in our later discussion of TCP, the Internet’s Transmission Control Protocol
Introduction to Internetworking (KJ)

Pre-Reading: P & D, §§3.2, 4.1 (5/e); §§4.1, 4.3 (4/e)

NEXT TIME