User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- **connectionless:**
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired
**UDP: more**

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

**UDP segment format**

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Length, in bytes of UDP segment, including header

Application data (message)

UDP segment format
**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum

**Receiver:**
compute checksum of received segment
check if computed checksum equals checksum field value:
- NO - error detected
- YES - no error detected. *But maybe errors nonetheless?* More later....
Internet Checksum Example

• Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result

• Example: add two 16-bit integers

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

wraparound \[\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}\]

sum \[\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{array}\]

checksum
Principles of Reliable data transfer

• important in app., transport, link layers
• top-10 list of important networking topics!

characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable data transfer: getting started

**send side**

- `rdt_send()` called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

- `udt_send()` called by `rdt` to transfer packet over unreliable channel to receiver

**receive side**

- `rdt_rcv()` called when packet arrives on rcv-side of channel

- `deliver_data()` called by `rdt` to deliver data to upper

- `udt_send()` called by `rdt`, to transfer packet over unreliable channel to receiver

- `rdt_rcv()` called when packet arrives on rcv-side of channel

- `rdt_send()` called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
We’ll:

• incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
• consider only unidirectional data transfer
  – but control info will flow on both directions!
• use finite state machines (FSM) to specify sender, receiver

**state:** when in this “state” next state uniquely determined by next event

---

**state diagram:**
- **state 1**
- **state 2**
- Event causing state transition
- Actions taken on state transition
- Event actions
Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel

sender

receiver
underlying channel may flip bits in packet
  - checksum to detect bit errors

the question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
    - sender retransmits pkt on receipt of NAK

new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK, NAK)
**rdt2.0: FSM specification**

**sender**

- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`

**receiver**

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_send(data)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`

- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `Lambda`
rdt2.0: operation with no errors

- **rdt_send(data)**
  - `snkpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`

- **Wait for call from above**

- **Wait for ACK or NAK**
  - `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
  - `udt_send(sndpkt)`

- **rdt_rcv(rcvpkt) && isACK(rcvpkt)**

- **Wait for call from below**

- **rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)**
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `udt_send(ACK)`

- **corrupt(rcvpkt)**
  - `udt_send(NAK)`

- **notcorrupt(rcvpkt)**
rdt2.0: error scenario

```
rdt_send(data)

rkpkt = make_pkt(data, checksum)
udt_send(rkpkt)

rdt_rcv(rcvpkt) &&
  isNAK(rcvpkt)
  udt_send(rkpkt)

rdt_rcv(rcvpkt) &&
  isACK(rcvpkt)
  udt_send(ACK)

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
  extract(rcvpkt, data)
  deliver_data(data)
  udt_send(ACK)

Wait for call from above
```

Wait for ACK or NAK

Wait for call from below
rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

• sender doesn’t know what happened at receiver!

• can’t just retransmit: possible duplicate

Handling duplicates:
sender retransmits current pkt if ACK/NAK garbled
sender adds sequence number to each pkt
receiver discards (doesn’t deliver up)

stop and wait pkt
Sender sends one packet, then waits for receiver response
rdt2.1: sender, handles garbled ACK/NAKs

```
rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

rdt_send(data)

rdt_rcv(rcvpkt) &&
( notcorrupt(rcvpkt) ||
  isACK(rcvpkt) )
udt_send(sndpkt)

Wait for call 0 from above

Wait for ACK or NAK 0

rdt_rcv(rcvpkt) &&
( notcorrupt(rcvpkt) &&
  isACK(rcvpkt) )
udt_send(sndpkt)

Wait for call 1 from above

Wait for ACK or NAK 1

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

Wait for call 1 from above

rdt_rcv(rcvpkt) &&
( notcorrupt(rcvpkt) &&
  isACK(rcvpkt) )
udt_send(sndpkt)
```

Λ

Λ
**rdt2.1: receiver, handles garbled ACK/NAKs**

```
rdt_rcv(rcvpkt) && notcorrutcrcvpkt)
  && has_seq0(rcvpkt)
    extract(rcvpkt, data)
    deliver_data(data)
    sndpkt = make_pkt(ACK, checksum)
    udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrutcrcvpkt)
    sndpkt = make_pkt(NAK, checksum)
    udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
    not corrutcrcvpkt) &&
  has_seq1(rcvpkt)
  sndpkt = make_pkt(ACK, checksum)
  udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
(corrutcrcvpkt)
    sndpkt = make_pkt(NAK, checksum)
    udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
    not corrutcrcvpkt) &&
  has_seq0(rcvpkt)
  sndpkt = make_pkt(ACK, checksum)
  udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
(corrutcrcvpkt)
    sndpkt = make_pkt(NAK, checksum)
    udt_send(sndpkt)
```