Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - Segment structure
   - Reliable data transfer
   - Flow control
   - Connection management
3.6 Principles of congestion control
3.7 TCP congestion control
UDP: User Datagram Protocol [RFC 768]

- Standardized in 1980
  - Hasn’t changed since
- Best-effort service
- UDP segments may be:
  - Lost or corrupted
  - Delivered out of order to the application
- Connectionless:
  - No handshaking between UDP sender and receiver
  - Each UDP segment handled independently of others

Why is there a UDP?
- Low overhead: no connection establishment or retransmission
- Simplicity: no connection state at sender/receiver
- Small segment header
- No congestion control
  - For short transfers, this is completely unnecessary
  - In other cases, desirable to control rate directly from application
UDP: More

- Often used for streaming multimedia or online gaming
  - Loss tolerant
  - Rate/delay sensitive
- Other UDP uses
  - DNS
  - SNMP
  - NFSv2 (1989)
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recovery

UDP segment format:

- Source port #
- Destination port #
- Length
- Checksum

Length (in bytes) of UDP segment, including header

32 bits
UDP Checksum

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

**Sender (simplified):**
- Set checksum = 0 in hdr
- Treat packet contents as a sequence of 16-bit integers (padded with 0s to 2-byte boundary)
- **Checksum:** add all integers, then XOR with 0xffff
- Sender puts checksum value into UDP checksum field

**Receiver:**
- Sum all 16-bit words in entire received segment (including the checksum field in the header)
- Check if result = 0xffff
  - NO - error detected
  - YES - no error detected
- Idea: $(x \text{ XOR } 0xffff) + x = 0xffff$
- *Are undetected errors possible nonetheless?*
UDP Checksum Example

• Note on 1’s complement addition:
  - When adding numbers, a carryout from the most significant bit needs to be added to the result

• Example: add two 16-bit integers

```
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
```

```
1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
```

wraparound

```
1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
```

sum

```
1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 0
```

checksum

```
0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```
UDP Checksum (Cont)

• How many corrupted bits does UDP detect?
• Example of undetected single-bit corruption?
  – Not possible
• Example of undetected 2-bit corruption?
  – Two words (0, 5) result in sum = 5
  – Suppose 0 is corrupted to become 1 and 5 is corrupted to become 4, then the checksum is the same
• Example of undetected 3-bit corruption w/two words?
  – Two words (1, 1) \(\rightarrow\) (0, 2)
• What if the transmitted words are 0 and 12?
  – Can two-bit corruption produce the same checksum?
  – If yes, how many ways can (0,12) be affected by 2-bit corruption so as to avoid detection?
UDP Checksum (Cont)

• Is there a pair of integers (x,y) that allow the UDP checksum to detect any 2-bit corruption?
• Data-link and physical layers are often assumed to have their own checksums and error correction
  – Why is transport-level checksum important then?
• Reasons:
  1) Lower layers do not always run error checking
     – Even then, implementation bugs may affect the result
  2) Corruption may occur in router RAM or faulty hardware, outside the control of data-link protocols
Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - Segment structure
   - Reliable data transfer
   - Flow control
   - Connection management
3.6 Principles of congestion control
3.7 TCP congestion control
Principles of Reliable Data Transfer

- Important in application, transport, link layers

- Characteristics of unreliable channel will determine complexity of reliable data transfer (rdt) protocol
Reliable Data Transfer: Getting Started

**send side**

- **udt_send()**: called by rdt to pass packets to lower layer
- **udt_send()**: called by layer above to pass data to rdt

**receive side**

- **deliver_data()**: called by rdt to deliver data to upper layer
- **rdt_rcv()**: called by lower layer when it has a packet to deliver to rdt
- **rdt send()**: called by layer above to pass data to rdt
- **rdt_rcv()**: called by lower layer when it has a packet to deliver to rdt

Diagram:

[Diagram showing the interaction between udt_send(), rdt_send(), deliver_data(), and rdt_recv().]
Reliable Data Transfer: Getting Started

We will:

- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - With receiver feedback, packets travel in both directions!
- Use finite state machines (FSM) to specify both sender and receiver

- From any state, the next state is uniquely determined by next event

![Finite State Machine Diagram]

- event causing state transition
- actions taken on state transition

- state 1
- state 2
Rdt1.0: Transfer Over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
  - No reordering

- Separate FSMs for sender and receiver:
  - Sender transmits app data into underlying channel
  - Receiver passes data from underlying channel to app

sender

receiver
Rdt2.0: Channel With Bit Errors

• Underlying channel may flip bits in packet (no loss)
  – Checksum to detect bit errors (assume perfect detection)

• Question: how to recover from errors?

• One possible approach is to use two feedback msgs:
  – *Positive acknowledgments (ACKs)*: receiver explicitly tells sender that packet was received OK
  – *Negative acknowledgments (NAKs)*: receiver explicitly tells sender that packet had errors
  – Sender retransmits packet on receipt of NAK

• New mechanisms in rdt 2.0 (beyond rdt 1.0):
  – Error detection
  – Receiver feedback (control msgs ACK/NAK)
  – Retransmission
**Rdt2.0: FSM Specification**

*Rdt_send(data)*

\[ \text{sndpkt} = \text{make_pkt(data, checksum)} \]
\[ \text{udt_send(sndpkt)} \]

**sender**

\[ \text{rdt_rcv(rcvpkt) AND isACK(rcvpkt)} \]
\[ \Lambda \]

\[ \Lambda = \text{empty action, i.e., do nothing} \]

**receiver**

\[ \text{extract(rcvpkt, data)} \]
\[ \text{deliver_data(data)} \]
\[ \text{udt_send(ACK)} \]

\[ \text{udt_send(NAK)} \]

\[ \text{rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt)} \]

\[ \text{Wait for call from below} \]

\[ \text{rdt_rcv(rcvpkt) AND isNAK(rcvpkt)} \]
\[ \text{udt_send(sndpkt)} \]

\[ \text{Wait for call from above} \]
Rdt2.0: Operation With No Errors

- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(ACK)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(NAK)`
- `wait for call from above`
- `wait for ACK or NAK`
- `wait for call from below`
Rdt2.0: Error Scenario

```
rdt_send(data)

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

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
udt_send(sndpkt)

rdt_send(sndpkt)

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
udt_send(NAK)

Wait for
ACK or
NAK

Wait for call from above

rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ

extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)

Wait for call from below
```

Any problems with this protocol?