Outline

- Admin and recap
  - Overview of transport layer
  - UDP
  - Reliable data transfer, the stop-and-wait protocol
Admin

- Namratha will send an email to give more details on Assignment one P3 quad-core solution
- Assignment three questions

Recap: Content Lookup

- Example unstructured p2p systems
  - Napster (central query server)
  - Gnutella (decentralized, flooding)
  - Freenet (search by routing)

- Advantages of unstructured p2p
  - algorithms tend to be simple

- Disadvantages
  - hard to make performance guarantee
  - failure even when files exist
Recap: The Computer Networks
Distributed Search Question

Question: what kind of long distance links to maintain so that distributed network search is effective?

Assume that each node has
- a fixed # (say p distance away) local links
- a small # (say a total of q) long-distance links s.t. the probability of a link between nodes x and y is some (α) inverse-power of the distance d(x, y) of x and y

\[ \frac{d(x, y)^{-\alpha}}{\Delta x} \]

A: α should be the dimension

Recap: Distributed Search

In other words, a guideline on long-distance links: roughly the same number of nodes in each region A1, A2, A4, A8, where A1 is the set of nodes who are one lattice step away, A2 is those two steps away, A4 is four steps away...

probability is proportional to (lattice steps)^d
Recap: Small World

Distributed Hash Tables (DHT)

- In 2000-2001, academic researchers jumped on to the P2P bandwagon
- Motivation:
  - frustrated by popularity of all these “half-baked” P2P apps. We can do better! (so they said)
  - guaranteed lookup success for data in system
  - provable bounds on search time (motivated by the Kleinberg observation)
- Examples
  - CAN, Chord
  - See optional slides at the end of lecture on Tuesday
Summary: P2P

- Many issues remaining, e.g.,
  - Streaming
  - ISP/P2P integration
  - ...

Outline

- Recap
  - Overview of transport layer
Transport Layer vs. Network Layer

- Provide **logical communication between app’ processes**
- Transport protocols run in end systems
  - send side: breaks app messages into **segments**, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- Transport vs. network layer services:
  - **Network layer:** data transfer between end systems
  - **Transport layer:** data transfer between processes
    - relies on, enhances network layer services

Transport Layer Services and Protocols

- **Reliable, in-order delivery (TCP)**
  - multiplexing
  - reliability and connection setup
  - congestion control
  - flow control

- **Unreliable, unordered delivery: UDP**
  - multiplexing

- **Services not available:**
  - delay guarantees
  - bandwidth guarantees
Transport Layer: Road Ahead

- Transport overview (class 1, today):
  - transport layer services
  - connectionless transport: UDP
  - reliable data transfer using stop-and-wait
  - sliding window reliability

- Sliding window and TCP (class 2):
  - Connection management
  - TCP reliability

- Congestion control and AIMD (class 3)

- TCP/Reno, TCP/Vegas, TCP in 3G/4G networks (class 4)

- The Primal/Dual framework, Nash Bargaining solution (class 5)

Outline

- Admin and recap
- Overview of transport layer
  - UDP
- Reliable data transfer, the stop-and-go protocols
**UDP: User Datagram Protocol [RFC 768]**

- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive

- Other UDP uses
  - DNS
  - SNMP

**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>

**UDP Checksum**

**Goal:** end-to-end detection of “errors” (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition of segment contents to be zero
- sender puts checksum value into UDP checksum field

**Receiver:**
- compute checksum of received segment
- compute sum of segment and checksum; check if sum zero
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless?
- compute checksum of received segment
One’s Complement Arithmetic

- UDP checksum is based on one’s complement arithmetic
  - one’s complement was a common representation of signed numbers in early computers
- One’s complement representation
  - bit-wise NOT for negative numbers
  - example: assume 8 bits
    - 00000000: 0
    - 00000001: 1
    - 01111111: 127
    - 10000000: ?
    - 11111111: ?
  - addition: conventional binary addition except adding any resulting carry back into the resulting sum
    - Example: -1 + 2

UDP Checksum: Algorithm

- Example checksum:

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
\text{wraparound} & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\text{sum} & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 &
UDP Checksum: Coverage

Calculated over:
- A pseudo-header
  - IP Source Address (4 bytes)
  - IP Destination Address (4 bytes)
  - Protocol (2 bytes)
  - UDP Length (2 bytes)
- UDP header
- UDP data

Outline
- Admin and recap
- Overview of transport layer
- UDP
  - Reliable data transfer
Principles of Reliable Data Transfer

- Important in app., transport, link layers
- Top-10 list of important networking topics!
Reliable Data Transfer: APIs

- **rdt_send()**: called from above, (e.g., by app.)
- **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 from below; when packet arrives on rcv-side of channel

Reliable Data Transfer: Getting Started

**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 Transition Diagram**:
- **State 1**: uniquely determines next state and actions
  - event causing state transition
  - actions taken on state transition
- **State 2**:
Outline

- Admin and recap
- Overview of transport layer
- UDP
  - Reliable data transfer
    - perfect channel

Rdt1.0: reliable transfer over a reliable channel

- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender

receiver
Potential Channel Errors

- bit errors
- loss (drop) of packets
- reordering or duplication

Characteristics of unreliable channel will determine the complexity of a reliable data transfer protocol (rdt).

Outline

- Admin and recap
- Overview of transport layer
- UDP
  - Reliable data transfer
    - perfect channel
    - channel with only bit errors
Rdt2.0: Channel With Bit Errors

- Underlying channel may flip bits in packet

- New mechanisms in rdt2.0 (beyond rdt1.0):
  - receiver error detection: recall: UDP checksum to detect bit errors
  - sender retransmission
  - receiver feedback: control msgs (ACK, NAK) rcvr->sender
    - 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

rdt2.0: FSM Specification

```
sender

wait for data

udt_send(sndpkt)

receiver

wait for ACK or NAK

snkpkt = make_pkt(data, checksum)

rdt_send(data)

udt_send(sndpkt)

rdt_recv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

rdt_recv(rcvpkt) && isACK(rcvpkt)

\Lambda
```
### rdt2.0: Operation with No Errors

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

**Wait for data**

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

- `rdt_rcv(rcvpkt)` & `isNAK(rcvpkt)`
  - `udt_send(NAK)`

- `rdt_rcv(rcvpkt)` & `corrupt(rcvpkt)`
  - `Wait for data`

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

### rdt2.0: Error Scenario

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

**Wait for data**

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

- `rdt_rcv(rcvpkt)` & `isNAK(rcvpkt)`
  - `udt_send(NAK)`

- `rdt_rcv(rcvpkt)` & `corrupt(rcvpkt)`
  - `Wait for data`

- `rdt_rcv(rcvpkt)` & `notcorrupt(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `udt_send(ACK)`
Big Picture of rdt2.0

rdt2.0 is Incomplete!

What happens if ACK/NAK corrupted?
- Although sender receives feedback, but doesn’t know what happened at receiver!

(sender data (n) waiting for N/ACK)

(data (n) NACK)*(data (n) ACK)

(data (n+1))

(sender data (n) waiting for N/ACK)

(receiver)

(ACK)

(data (n))

(NACK)

(data (n))
**Two Possibilities**

It is always harder to deal with control message errors than data message errors.

```plaintext
sender

waiting for N/ACK

data (n)

NACK

data (n)

receiver

deliver

Wait for data

NACK

data (n+1)

deliver

Ack

Wait for data

N/ACK
```

**Handle Control Message Corruption**

- **Sender can’t do anything even if the control pkt is corrupted**
  - The pkt might be an NACK
- **If it just retransmits: possible duplicate**

**Handling duplicates:**

- **sender adds sequence number to each pkt**
- **sender retransmits current pkt if ACK/NAK garbled**
- **receiver discards (doesn’t deliver up) duplicate pkt**

stop and wait
sender sends one packet, then waits for receiver response
rdt2.1b: Sender, Handles Garbled ACK/NAKs

```plaintext
rdt_send(data)
   sndpkt = make_pkt(n, data, checksum)
   udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
   (corrupt(rcvpkt) ||
    isNAK(rcvpkt))
   udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt)
   && notcorrupt(rcvpkt)
   && isACK(rcvpkt)
   \Lambda
```

```
rdt_send(data)
   sndpkt = make_pkt(n+1, data, checksum)
   udt_send(sndpkt)
```

rdt2.1b: Receiver, Handles Garbled ACK/NAKs

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
   && has_seq(n, rcvpkt)
   extract(rcvpkt, data)
   deliver_data(data)
   sndpkt = make_pkt(ACK, checksum)
   udt_send(sndpkt)
```

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

```
rdt_rcv(rcvpkt) &&
   not corrupt(rcvpkt) &&
   ! has_seq(n,rcvpkt)
   sndpkt = make_pkt(ACK, checksum)
   udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
   sndpkt = make_pkt(NAK, checksum)
   udt_send(sndpkt)
```
**rdt2.1b: Summary**

**Sender:**
- seq # added to pkt
- must check if received ACK/NAK corrupted

**Receiver:**
- must check if received packet is duplicate
  - by checking if the packet has the expected pkt seq #

---

**Another Look at rdt2.1b**

[Diagram showing data flow and packet transmission between sender and receiver]
**State Relationship of rdt2.1b**

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>w0</td>
<td>First time receiving ack 0</td>
</tr>
<tr>
<td>s0</td>
<td>First time sending data for seq. 1</td>
</tr>
<tr>
<td>W1</td>
<td>Wait for data for seq. 1</td>
</tr>
<tr>
<td>S1</td>
<td>Sending data (waiting ACK) for seq. 1</td>
</tr>
<tr>
<td>s2</td>
<td>Sender state can be sending either seq#n-1 or seq#n</td>
</tr>
<tr>
<td>w2</td>
<td>Receiver state leads: if it is waiting for seq #n, sender’s state can be sending either seq#n-1 or seq#n</td>
</tr>
</tbody>
</table>

**State invariant:**
- Receiver’s state leads: if it is waiting for seq #n, sender’s state can be sending either seq#n-1 or seq#n

**rdt2.1c: Sender, Handles Garbled ACK/NAKs: Using 1 bit**

```
rdt_send(data)
    sndpkt = make_pkt(0, data, checksum)
    udt_send(sndpkt)
    rdt_send(data)
    rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
        rdt_send(data)
        sndpkt = make_pkt(1, data, checksum)
        udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
    udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
    udt_send(sndpkt)
```

- Wait for call 0 from above
- Wait for ACK or NAK
- Wait for call 1 from above

Diagram showing the state transitions and conditions for handling garbled ACKs and NAKs with 1 bit.
**rdt2.1c: Receiver, Handles Garbled ACK/NAKs: Using 1 bit**

- rdt$_{rcv}$(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
  - extract(rcvpkt.data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)
- rdt$_{rcv}$(rcvpkt) && (corrupt(rcvpkt) && has_seq1(rcvpkt)
  - extract(rcvpkt.data)
  - deliver_data(data)
  - sndpkt = make_pkt(NAK, chksum)
  - udt_send(sndpkt)

**rdt2.1c: Discussion**

**Sender:**
- state must “remember” whether “current” pkt has 0 or 1 seq. #

**Receiver:**
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
A Flavor of Protocol Correctness Analysis: rdt2.1c

- Combined states of sender, receiver and channel
- Assume the sender always has data to send

sender state: sending 0 and waiting for ACK
receiver state: waiting for 0
channel state: packet seq# corrupt ok corrupt ok corrupt ok corrupt

0 0 N 0 0 0 0 1 A 1 1 N

1 0 0 1 0 A 1 1 1

1 0 N corrupt

rdt2.2: a NAK-free protocol

- Same functionality as rdt2.1c, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must explicitly include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: retransmit current pkt
rdt2.2: Sender, Receiver Fragments

```
<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt_send(data)</td>
<td>sndpkt = make_pkt(0, data, checksum)</td>
</tr>
<tr>
<td></td>
<td>udt_send(sndpkt)</td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt)</td>
<td>rdt_rcv(rcvpkt) &amp;&amp; (corrupt(rcvpkt)</td>
</tr>
<tr>
<td></td>
<td>sndpkt = make_pkt(ACK,0, checksum)</td>
</tr>
<tr>
<td></td>
<td>udt_send(sndpkt)</td>
</tr>
<tr>
<td></td>
<td>rdt_rcv(rcvpkt) &amp;&amp; notcorrupt(rcvpkt) &amp;&amp; isACK(rcvpkt,0)</td>
</tr>
<tr>
<td></td>
<td>sndpkt = make_pkt(ACK,1, checksum)</td>
</tr>
<tr>
<td></td>
<td>udt_send(sndpkt)</td>
</tr>
</tbody>
</table>

**Outline**

- **Review**
- **Overview of transport layer**
- **UDP**
  - **Reliable data transfer**
    - perfect channel
    - channel with bit errors
    - channel with bit errors and losses
**rdt3.0: Channels with Errors and Loss**

**New assumption:**
underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

**Approach:** sender waits “reasonable” amount of time for ACK
- requires countdown timer
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed

**Q:** how to deal with loss?

---

**rdt3.0 Sender**

```
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

wait for ACK0

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

rrt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt, 1)
stop_timer

wait for call 1 from above

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

rrt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt, 0)
stop_timer

wait for call 0 from above

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

rrt_rcv(rcvpkt)
&& ( corrupt(rcvpkt) || isACK(rcvpkt, 1) )
stop_timer

wait for ACK1

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

rrt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt, 0)
stop_timer

wait for call 1 from above

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

rrt_rcv(rcvpkt)
&& ( corrupt(rcvpkt) || isACK(rcvpkt, 0) )
stop_timer

```

---

49

50

25
**Question to think about:** How to determine a good timeout value?
rdt3.0: Stop-and-Wait Operation

What is $U_{\text{sender}}$: utilization – fraction of time sender busy sending?

Assume: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[ T_{\text{transmit}} = \frac{L}{R} = \frac{8kb/\text{pkt}}{10^{9} \text{ b/sec}} = 8 \text{ microsec} \]

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

- 1KB pkt every 30 msec → 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

A Summary of Questions

- How to improve the performance of rdt3.0?
- What if there are reordering and duplication?
- How to determine the “right” timeout value?
Outline

- Review
- Overview of transport layer
- UDP
  - Reliable data transfer
    - perfect channel
    - channel with bit errors
    - channel with bit errors and losses
  - sliding window: reliability with throughput

---

Sliding Window Protocols: Pipelining

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

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation
**Pipelining: Increased Utilization**

- First packet bit transmitted, $t = 0$.
- Last bit transmitted, $t = L / R$.
- First packet bit arrives, sender.
- Last packet bit arrives, send ACK.
- Last bit of 2nd packet arrives, send ACK.
- Last bit of 3rd packet arrives, send ACK.

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

Question: a rule-of-thumb window size?

**Forms of Pipelining Protocols**

- Two generic forms of pipelined protocols:
  - *go-Back-N*
  - *selective repeat*
A Summary of Questions

- How to improve the performance of rdt3.0?
  - sliding window protocols

- What if there are duplication and reordering?
  - network guarantee: max packet life time
  - transport guarantee: not reuse a seq# before life time
  - seq# management and connection management

- How to determine the “right” parameters?