What is the class about?

- **Course description and syllabus:**
  - [http://www.nyu.edu/classes/jcf/csci-ga.2262-001/](http://www.nyu.edu/classes/jcf/csci-ga.2262-001/)
  - [http://cs.nyu.edu/courses/fall15/CSCI-GA.2262-001/index.html](http://cs.nyu.edu/courses/fall15/CSCI-GA.2262-001/index.html)

- **Textbooks:**
    - James F. Kurose, Keith W. Ross
    - Addison Wesley
Course Overview

- Computer Networks and the Internet
- Application Layer
- Fundamental Data Structures: queues, ring buffers, finite state machines
- Data Encoding and Transmission
- Local Area Networks and Data Link Control
- Wireless Communications
- Packet Switching
- OSI and Internet Protocol Architecture
- Congestion Control and Flow Control Methods
- Internet Protocols (IP, ARP, UDP, TCP)
- Network (packet) Routing Algorithms (OSPF, Distance Vector)
- IP Multicast
- Sockets
Course Approach

- Introduction to Basic Networking Concepts (Network Stack)
- Origins of Naming, Addressing, and Routing (TCP, IP, DNS)
- Physical Communication Layer
- MAC Layer (Ethernet, Bridging)
- Routing Protocols (Link State, Distance Vector)
- Internet Routing (BGP, OSPF, Programmable Routers)
- TCP Basics (Reliable/Unreliable)
- Congestion Control
- QoS, Fair Queuing, and Queuing Theory
- Network Services – Multicast and Unicast
- Extensions to Internet Architecture (NATs, IPv6, Proxies)
- Network Hardware and Software (How to Build Networks, Routers)
- Overlay Networks and Services (How to Implement Network Services)
- Network Firewalls, Network Security, and Enterprise Networks
Reliable Data Transfer Session in Brief

- Principles of Reliable Data Transfer
- Reliable Data Transfer: Getting Started
- Reliable Data Transfer: Operational Details
- Other Reliable Data Transfer Protocols
- Conclusion
Agenda

1. Session Overview
2. Reliable Data Transfer
3. Summary and Conclusion
Reliable Data Transfer Session in Brief

- Principles of Reliable Data Transfer
- Reliable Data Transfer: Getting Started
- Reliable Data Transfer: Operational Details
- Other Reliable Data Transfer Protocols
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 Session in Brief

- Principles of Reliable Data Transfer
- Reliable Data Transfer: Getting Started
- Reliable Data Transfer: Operational Details
- Other Reliable Data Transfer Protocols
**Reliable Data Transfer: Getting Started**

- **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.

- **rdt_rcv()**: called when packet arrives on rcv-side of channel.

- **deliver_data()**: called by rdt to deliver data to upper.

- **send side**: rdt_send() → data → reliable data transfer protocol (sending side) → udt_send() → packet → unreliable channel → rdt_rcv() → deliver_data() → reliable data transfer protocol (receiving side) → packet.

- **receive side**: data → deliver_data() → reliable data transfer protocol (receiving side) → packet → rdt_rcv() → unreliable channel → udt_send() → packet → reliable data transfer protocol (sending side) → data.
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
Reliable Data Transfer Session in Brief

- Principles of Reliable Data Transfer
- Reliable Data Transfer: Getting Started
- Reliable Data Transfer: Operational Details
- Other Reliable Data Transfer Protocols
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
Rdt2.0: Channel with Bit Errors

- 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) rcvr->sender
Rdt2.0: FSM Specification

```
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)

\Lambda

sender
```

```
receiver

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

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```
Rdt2.0: Operation with No Error

```
rdt_send(data)

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

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
isACK(rcvpkt)

udt_send(ACK)
```

```
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)
```

```
Wait for call from above
```

```
Wait for ACK or NAK
```

```
Wait for call from below
```

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

```
Lambda
```
Rdt2.0: Error Scenario

Rdt_send(data)

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

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

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

Wait for
call from
above

Wait for
ACK or
NAK

Wait for
call from
below

rdt_rcv(rcvpkt) &&
isACK(rcvpkt)

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

Lambda
Rdt 2.0 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) duplicate pkt

*stop and wait*
Sender sends one packet, then waits for receiver response
Rdt2.1: Sender Handles Garbled ACK/NAKs

\[ \text{rdt\_send(data)} \]
\[ \text{sndpkt = make\_pkt(0, data, checksum)} \]
\[ \text{udt\_send(sndpkt)} \]

Wait for call 0 from above

Wait for ACK or NAK 0

\[ \text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt) \&\& isACK(rcvpkt)} \]

Wait for call 1 from above

Wait for ACK or NAK 1

\[ \text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt) \&\& isACK(rcvpkt)} \]

\[ \text{rdt\_send(data)} \]
\[ \text{sndpkt = make\_pkt(1, data, checksum)} \]
\[ \text{udt\_send(sndpkt)} \]
Rdt2.1: Receiver Handles Garbled ACK/NAKs

\[
\begin{align*}
&\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) && has\_seq0(rcvpkt) } \\
&\text{extract(rcvpkt, data)} \\
&\text{deliver\_data(data)} \\
&\text{sndpkt = make\_pkt(ACK, checksum)} \\
&\text{udt\_send(sndpkt)} \\
\end{align*}
\]

Wait for 0 from below

\[
\begin{align*}
&\text{rdt\_rcv(rcvpkt) && not corrup\_t(rcvpkt) && has\_seq1(rcvpkt) } \\
&\text{sndpkt = make\_pkt(NAK, checksum)} \\
&\text{udt\_send(sndpkt)} \\
\end{align*}
\]

Wait for 1 from below

\[
\begin{align*}
&\text{rdt\_rcv(rcvpkt) && (corrupt(rcvpkt) } \\
&\text{sndpkt = make\_pkt(NAK, checksum)} \\
&\text{udt\_send(sndpkt)} \\
\end{align*}
\]
Sender:

- Seq # added to pkt
- Two seq. #’s (0,1) will suffice. Why?
- Must check if received ACK/NAK corrupted
- Twice as many states
  » 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 #
- Note: receiver can *not* know if its last ACK/NAK received OK at sender
Rdt2.2: A NAK-Free Protocol

- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - Receiver must \textit{explicitly} include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: \textit{retransmit current pkt}
Rdt2.2: Sender, Receiver Fragments

Sender FSM fragment:
- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `rdt_send(data)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK1, checksum)`
- `udt_send(sndpkt)`

Receiver FSM fragment:
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK1, checksum)`
- `udt_send(sndpkt)`

Wait for call 0 from above:
- ` sndpkt = make_pkt(0, data, checksum)`
- ` udt_send(sndpkt)`

Wait for ACK 0:
- ` rdt_rcv(rcvpkt) && ( corrupt(rcvpkt) || isACK(rcvpkt, 1) )
- ` udt_send(sndpkt)`

Wait for 0 from below:
- ` rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)`

Corruption checks:
- `corrupt(rcvpkt)`
- `has_seq1(rcvpkt)`
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

- 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

- Requires countdown timer
Rdt3.0: Sender

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,1) )

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

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,1) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,0) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,1) )

rupt_send(sndpkt)
 start_timer
```

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,0) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,1) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,1) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,0) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,1) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,0) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,1) )

rupt_send(sndpkt)
 start_timer
```

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

```
rdt_rcv(rcvpkt) &&
 ( corrupt(rcvpkt) ||
   isACK(rcvpkt,0) )

rupt_send(sndpkt)
 start_timer
```
Rdt3.0 in Action

(a) operation with no loss

(b) lost packet
Rdt3.0 in Action

(c) lost ACK

(d) premature timeout
Rdt3.0 works, but performance stinks

Example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**9} \text{ b/sec}} \approx 8 \text{ microsec}$$

\[ U_{sender}: \textit{utilization} – \text{fraction of time sender busy sending} \]

\[ U_{sender} = \frac{L / R}{\text{RTT} + L / R} = \frac{0.008}{30.008} = 0.00027 \text{ microsec} \]

\[ \text{1KB pkt every 30 msec} \rightarrow 33\text{kB/sec thruput over 1 Gbps link} \]

\[ \text{network protocol limits use of physical resources!} \]
Rdt 3.0: Stop-and-Wait Operation

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = L / R$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = RTT + L / R$

Sender

Receiver

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027 \text{ microsec}$$
Reliable Data Transfer Session in Brief

- Principles of Reliable Data Transfer
- Reliable Data Transfer: Getting Started
- Reliable Data Transfer: Operational Details
- Other Reliable Data Transfer Protocols
- Conclusion
Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

- Two generic forms of pipelined protocols: go-Back-N, selective repeat
Pipelining: Increased Utilization

\[ U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008 \text{ microsecond} \]

Increase utilization by a factor of 3!
Go-Back-N

**Sender:**
- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

![Diagram showing the concepts of Go-Back-N](image)

- **ACK(n):** ACKs all pkts up to, including seq # n - “cumulative ACK”
  - May receive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- **timeout(n):** retransmit pkt n and all higher seq # pkts in window
GBN: Sender Extended FSM

```
runtime

```
```rout
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])

...udt_send(sndpkt[nextseqnum-1])
```rout

```rout
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        stop_timer
    nextseqnum++
}
#else
    refuse_data(data)
```

```rout
base = getacknum(rcvpkt)+1
if (base == nextseqnum)
    stop_timer
nextseqnum++
else
    start_timer
```

```rout
rdt_send(data)
```rout

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

```rout
base=1
nextseqnum=1
```

```rout
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
```

```rout
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
```

```rout
base = getacknum(rcvpkt)+1
if (base == nextseqnum)
    stop_timer
else
    start_timer
```

```
L
```
GBN: Receiver Extended FSM

- **ACK-only**: always send ACK for correctly-received pkt with highest *in-order* seq 
  - May generate duplicate ACKs
  - Need only remember *expectedseqnum*

- **Out-of-order pkt**: 
  - Discard (don’t buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq 

\[
\begin{align*}
&\text{default} \\
&\text{udt\_send(sndpkt)} \\
&\text{Wait} \\
&\text{rdt\_rcv(rcvpkt)} \\
&\quad \text{\&\& notcurrupt(rcvpkt)} \\
&\quad \text{\&\& hasseqnum(rcvpkt,expectedseqnum)} \\
&\text{extract(rcvpkt,data)} \\
&\text{deliver\_data(data)} \\
&\text{sndpkt = make\_pkt(expectedseqnum,ACK,chksum)} \\
&\text{udt\_send(sndpkt)} \\
&\text{expectedseqnum++}
\end{align*}
\]
GBN In Action

sender

send pkt0
send pkt1
send pkt2
send pkt3 (wait)

receiver

rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1

pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5

rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3
Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - Buffers pkts, as needed, for eventual in-order delivery to upper layer
- Sender only resends pkts for which ACK not received
  - Sender timer for each unACKed pkt
- Sender window
  - N consecutive seq #’s
  - Again limits seq #s of sent, unACKed pkts
Selective Repeat: Sender, Receiver Windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective Repeat

**sender**

Data from above:
- If next available seq # in window, send pkt

Timeout(n):
- Resend pkt n, restart timer

ACK(n) in [sendbase, sendbase+N]:
- Mark pkt n as received
- If n smallest unACKed pkt, advance window base to next unACKed seq #

**receiver**

Pkt n in [rcvbase, rcvbase+N-1]
- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

Pkt n in [rcvbase-N, rcvbase-1]
- ACK(n)

Otherwise:
- Ignore
Selective Repeat in Action

pkt0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 sent
0 1 2 3 4 5 6 7 8 9

pkt2 sent
0 1 2 3 4 5 6 7 8 9

pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9

pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2 resent
0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9

pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2, pkt3, pkt4, pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9
Example:
- Seq #'s: 0, 1, 2, 3
- Window size=3
- Receiver sees no difference in two scenarios!
- Incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?
Summary

- Principles of Reliable Data Transfer
- Reliable Data Transfer: Getting Started
- Reliable Data Transfer: Operational Details
- Other Reliable Data Transfer Protocols
Assignments & Readings

- Readings
  - Chapter 3 (sections 3.1-3.4)
- Assignment #6