

# CS/ECE 4457

## Computer Networks: Architecture and Protocols

### Lecture 21 Reliable Transport

**Qizhe Cai**



**Where are we?**

# Recall the end-to-end story from our fifth lecture!

- Application opens a **socket** that allows it to connect to the **network stack**
- Maps **name** of the web site to its **address** using **DNS**
- The network stack at the source embeds the address and **port** for both the source and the destination in **packet header**
- Each **router** constructs a **routing table** using a distributed algorithm
- Each router uses destination address in the packet header to look up the **outgoing link** in the routing table
  - And when the link is free, forwards the packet
- When a packet arrives the destination:
  - The network stack at the destination uses the port to forward the packet to the right application

**You now know how the Internet works!!!!**

**All that is remaining:**

**Reliability.**

# Goal of Today's Lecture

- **Understanding reliable transport conceptually**
  - What are the fundamental aspects of reliable transport
- **Back to architectural principles for one lecture**
- The goal is not to understand a particular protocol (e.g., TCP)
  - TCP involves lots of detailed mechanisms, covered later
- Ground rules for discussion
  - No mention of TCP
  - No mention of detailed practical issues
  - Focus only on “ideal” world of packets and links

# You must think for yourself

- **Today's lecture requires you to engage**
  - How would I design a reliable service
- **I will ask a lot of questions today, want you to think about them**
  - Be that kid!

# Decisions and Their Principles

- How to break system into modules?
  - Dictated by layering
- Where are modules implemented?
  - Dictated by End-to-End Principle
- Where state is stored?
  - Dictated by fate-sharing

# Today We Design Reliable Delivery

- The end-to-end principle tells us?
  - Put reliability in the end-host, not the network
- Layering dictates putting reliability in what layer?
  - Above network layer
  - L4 focusses on process-to-process delivery (“flow”)
- Fate sharing tells us?
  - Keep all reliability state in ends, not in network

# Best Effort Service (L3)

- Packets can be lost
- Packets can be corrupted
- Packets can be reordered
- Packets can be delayed
- Packets can be duplicated
- ...

**How can you possibly make anything work  
with such a service model?**

# Making Anything Work on Top of Best Effort Service

- Engineer network so that average case is decent
  - You can't make guarantees, but the operator must try...
- Engineer apps so they can tolerate the worst case
  - They don't have to thrive, they just can't die
- A classical case of **architecting for flexibility**
  - And then **engineering for performance**

# Reliable Transport is Necessary

- Some app semantics involve reliable transport
  - E.g., file transfer
- **Layer 3 and below provide only unreliable packet delivery**
- Today's question:
  - How can we build a reliable transport service on top of arbitrary unreliable packet delivery?
- A central challenge in bridging the gap between
  - **The abstractions application designers want**
  - **The abstractions networks can easily support**

# Important Distinctions

- For functionality implemented in network:
  - Keep minimal (easy to build, broadly applicable)
- For functionality implemented in the application:
  - Keep minimal (easy to write)
  - Restricted to application-specific functionality
- Functionality implemented in “**network stack**”
  - The shared networking code on the host
  - This relieves burden from both application and network
  - **This is where reliability belongs**

# Two Different Statements

- **Some applications need reliable service**
  - This means that application writers should be able to assume this, to make their job easier
- **The network must provide reliable service**
  - This contends that applications cannot implement this functionality, so the network must provide it
- Today we're making the first statement and refuting the second...
  - And this simple observation is what advocates of reliable networks (as in telephony) never understood

# Challenge For Today

- Building a stack that supports reliable transfer
  - So that individual applications don't need to deal with packet losses, etc.
- What mechanisms can we put in the transport layer to provide reliability?
- Reliability is focused on single “flow”
  - Flow: stream of packets between two **processes**
  - **Usually defined using the 5-tuple:**
    - (sourceIP, destIP, sourcePort, destPort, protocol)

# Four Goals for Reliable Transfer

- **Correctness**
  - To be defined
- **“Fairness”**
  - Every flow must get a fair share of network resources
- **Flow Performance**
  - Latency, jitter, etc.
- **Utilization**
  - Would like to maximize bandwidth utilization
  - If network has bandwidth available, flows should be able to use it!

# Start With Transfer of a Single Packet

- We can later worry about larger files, but in the beginning it is cleaner to focus on this simple case

# Correctness Condition

- Routing had a clean correctness condition
- **We want same kind of “if and only if” characterization of “correct” reliable transport designs**
- This condition is **for the design to be correct**, not the best performant
- One obvious requirement:
  - Transport never claims to have delivered data that wasn't delivered...
- But we need more than that. What?

# Correctness Condition?

- How about: “Packet is always delivered to receiver”?
- i.e., Transport is reliable if and only if packets are always delivered to the receiver...
- Isn't that simple?

# WRONG!

- **What if network is partitioned?**
  - Partitioned means that the network is broken into two or more disconnected components...
- We can't claim a transport design is incorrect if it doesn't work in a partitioned network!
  - After all, there is no way to reach the destination!

# Correctness Condition?

- Packet is delivered to receiver if and only if its possible to deliver packet

# WRONG!

- If the network is only available at one instant of time, only an Oracle would know when to send
- **We can't claim a transport design is incorrect if it doesn't know the unknowable...**
- So we need to focus on what the transport design is trying to do, not what it actually accomplishes

# Correctness Condition?

- Resend packet if and only if the previous transmission was lost or corrupted
- This is better because it refers to:
  - what the design does (which it can control)
  - not whether it always succeeds (which it can't)

# WRONG!

- Impossible
  - “Coordinated Attack” over an unreliable network
- Consider two cases:
  - Packet delivered; all packets from receiver are dropped
  - Packet dropped; all packets from receiver are dropped
- They are indistinguishable to sender
  - In both cases, packet was sent, and no feedback at all
  - **Does it resend, or not?**

# Correctness Condition?

- Packet is always resent if the previous transmission was lost or corrupted
- Packet **may** be resent at other times
- Note:
  - This invariant gives us a simple criterion for deciding if an implementation is correct
  - Efficiency and simplicity are separate criteria

# Almost Right!

- What's wrong with it?
- An implementation that never sent the packet at all is reliable according to the definition.

# Complete Correctness Condition

- A transport mechanism is “reliable” if and only if
  - (a) It resends all dropped or corrupted packets
  - (b) It attempts to make progress
- Making progress means:
  - **If there is data to send, transport eventually attempts to send data**
    - Very important: “eventually attempts”!
    - It should not be blocked for ever
    - And, it may not succeed, but it must attempt
  - Example: If there are ten packets to send, transport can’t just send the first five and then stop **for ever**

# Note!

- A transport mechanism can “give up”, but must announce this to application
- If the transport mechanism has tried for some period to deliver the data, and has not succeeded:
  - It might decide that it is better to give up
  - And applications can reinitiate data transfer
  - That is allowed...
- **But it can never falsely claim to have delivered a packet**

# We have the correctness condition

- How do we achieve it?
- Focus on single-packet solutions

# Solution v1

- **Send every packet as often and fast as possible...**
- Is it correct
  - No.
  - Why?
  - The “if” condition is not satisfied:
    - (a) Transport must **attempt to make progress**
  - No way to check whether the packet was dropped or corrupted
    - So, must continue sending the same packet

# What's missing?

- Feedback from receiver!
- If receiver does not respond, no way for sender to tell when to stop resending
  - Cannot achieve correctness without feedback

# Forms of Feedback

- ACK: Yes, I got a packet
- NACK: No, I did not get the packet
- When is NACK a natural idea?
  - Packet Corruption (I got packet#5 but it was corrupted)
- Ignore NACKs for rest of the lecture...

# Solution v2

- Resend packet until you get an ACK
  - And receiver sends per-packet ACKs until data finally stops
- Correct?
  - Yes:
    - All dropped/corrupted packets will be retransmitted
    - The transport will attempt to make progress
- Fair?
  - Over long-term, yes:
    - all sources will get an equal chance to use network resources
- Flow performance?
  - Good but not necessarily optimal
    - Some packets may be retransmitted unnecessarily
- Efficiency:
  - suboptimal; packets retransmitted unnecessarily

# Solution v3

- Send packet
  - But now, **set a timer**
- When receiver gets packet, sends ACK
- If sender receives ACK, done
- If no ACK when timer expires, resend
  - Still **correct**, and **fair**
  - **Performance** would argue for **small timeout**
  - **Utilization** would argue for **larger timeout**
    - May want to increase timer each time you try
    - May want to cap the number of retries
    - Problems with this design?

# Have “Solved” the Single Packet Case

- Send packet
  - Set a timer
- If no ACK when timer goes off, resend packet
  - And reset timer
- Tradeoff between performance and utilization in selection of timeout:
  - Too small: unnecessary retransmissions (underutilization)
  - Too large: waiting unnecessarily (poor performance)

# Multiple Packets

- Service model: reliable stream of packets
  - Hand up contiguous block of packets to application
- Why not use single-packet solution?
  - Send the next packet once the first one has been delivered
  - Problem: Only one packet in flight at a time
    - **Low Effective throughput: Packet Size / RTT**
- Use window based approach
  - Allow for a window of  $W$  packets in-flight at any time (unack'ed)
  - Slide the window as packets are ack'ed
  - Sliding window implies  $W$  packets are continuous

# Window-based Algorithms

- Very simple concept
  - Send  $W$  packets
  - When one gets ACK'ed send the next packet in line
  - It really is that simple (until we got to TCP)
- Will consider several variations...
  - But first...

# How Big Should the Window be?

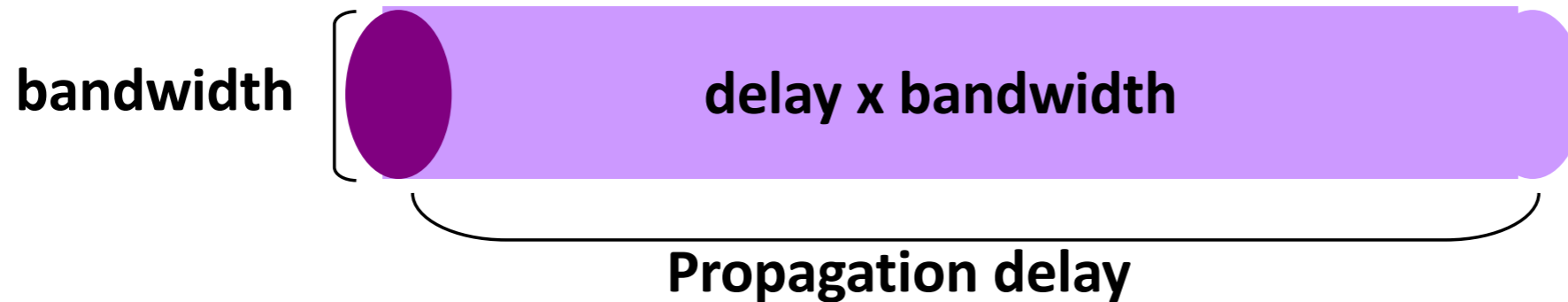
- **Windows serve three purposes**
  - Taking advantage of the bandwidth of the links
  - Limiting bandwidth used by a flow (congestion control)
  - Limiting the amount of buffering needed at the receiver
    - Why do receivers need to buffer packets?
      - Answer: packet re-ordering (discussed later)
- If we ignore all but the first goal, then we want to keep the sender always sending (in the ideal case)
  - RTT: from sending first packet until received first ACK
- **Condition:**
  - $RTT \times B \sim W \times \text{Packet Size}$

# What does this mean?

- B is the minimum link bandwidth along the path
  - Obviously shouldn't send faster than that
  - Don't want to send slower than that (for first goal)
- **We want to set  $W$  such that:**
  - if I am sending at rate B, then
  - the ACK of the first packet arrives
  - exactly when I just finish sending the last of my  $W$  packets
- **Lets me send as fast as the path can deliver...**

# $RTT \times B \sim W \times \text{Packet Size}$

- Recall that **Bandwidth Delay Product**
  - $BDP = \text{bandwidth} \times \text{propagation delay}$



- **$B \times RTT$  is merely  $2x$  BDP**
- Window sizing rule:
  - Total bits in flight is roughly the amount of data that fits into forward and reverse “pipes”
    - Here pipe is complete path, not single link...
    - **This is not “detail”, this is a fundamental concept...**

# Where Are We?

- **Figured out correctness condition:**
  - Always resend lost/corrupted packets
  - Always try to make progress (but can give up entirely)
- **Figured out single packet case:**
  - Send packet, set timer, resend if no ACK when timer expires
- **Some progress towards multiple packet case:**
  - Allow many packets ( $W$ ) in flight at once
  - And know what the ideal window size is
    - $RTT \times B / \text{Packet size}$
- What's left to design?

# Three Design Considerations

- Nature of feedback
  - What should ACKs tell us when we have many packets in flight
- Detection of loss
- Response to loss

# Possible Feedback From Receiver

- Ideas?

# ACK Individual Packets

- Strengths
  - Know fate of each packet
  - Reordering not a problem
  - Simple window algorithm
    - $W$  independent single packet algorithms
    - When one finishes grab next packet
- Weaknesses?
  - Loss of ACK packet requires a retransmission

# Full Information Feedback

- **List all packets that have been received**
  - Give highest cumulative ACK plus any additional packets
  - If packets 1, 2, 3, 5, 6 received: send ACK(3, 5, 6)
- Strengths?
  - As much information as you could hope for
  - Resilient form of individual ACKs
- Weaknesses?
  - Could require sizable overhead in bad cases
  - Feasible if only small holes
    - If packets 1, 5, 6, ....., 100 received: ACK(1, 5, 6, ...,100)

# Cumulative ACK

- ACK the highest sequence number for which all previous packets have been received
  - Implementations often send back “next expected packet”, but that’s just a detail
- Strengths?
  - Resilient to lost ACKs
- Weaknesses?
  - Confused by reordering
  - Incomplete information about which packets have arrived

# Detecting Loss

- If packet times out, assume it is lost...
- How else can you detect loss?

# Loss With Individual ACKs

- Assume that packet 5 is lost, but no others
- Stream of ACKs will be
  - 1
  - 2
  - 3
  - 4
  - 6
  - 7
  - 8
  - ...

# Loss With Individual ACKs

- Could resend packet when  $k$  “subsequent packets” are received
- Response to loss
  - Resend missing packet
  - Continue window based protocol

# Loss With Full Information

- Same story, except that the “hole” is explicit in each ACK
- Stream of ACKs will be
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4, plus 6
  - Up to 4, plus 6,7
  - Up to 4, plus 6,7,8
  - ...

# Loss With Full Information

- Could resend packet when  $k$  “subsequent packets” are received
- Response to loss
  - Resend missing packet
  - Continue window-based protocol

# Loss With Cumulative ACKs

- Assume packet 5 is lost, but no others
- Stream of ACKs will be
  - 1
  - 2
  - 3
  - 4
  - 4 (Sent when packet 6 arrives)
  - 4 (Sent when packet 7 arrives)
  - 4 (Sent when packet 8 arrives)
  - ...

## Loss With Cumulative ACKs (cont'd)

- Duplicate ACKs are a sign of an isolated loss
  - The lack of ACK progress means 5 hasn't been delivered
  - Stream of duplicate ACKs means some packets are being delivered (one for each subsequent packet)
- Therefore could trigger resend upon receiving k duplicate ACKs
- But response to loss is trickier...

# Loss With Cumulative ACKs (cont'd 2)

- Two choices
  - Send missing packet and optimistically assume that subsequent packets have arrived
    - i.e., increase  $W$  by the number of duplicate ACKs
  - Send missing packet, wait for ACK
- Timeout-detected losses also problematic
  - If packet 5 times out, packet 6 is about to timeout also
  - Do you resend both?
  - Do you resend 5 and wait?
  - ...

# Cumulative ACKs

- They make no sense, except as a cheap alternative to full information
  - Less state than full information
  - More resilient than individual ACKs
- But ambiguity in feedback leads to many problems
  - Have other packets arrived?
- Makes retransmission and congestion window management hard
- Will deal with these issues when we come to TCP

# All The Bad Things Best Effort Can Do

- Packets can be lost
- Packets can be corrupted
- Packets can be **reordered**
- Packets can be **delayed**
- Packets can be **duplicated**

# Effect of Reordering?

- For all designs this looks like “subsequent ACKs”
- This can be mistaken for packet loss
- Hard to realize the difference between these packet arrival patterns:
  - 1, 2, 3, 4, 6, 7, 8, 9,...
  - 1, 2, 3, 4, 6, 7, 8, 9, 5, 10,...

# Effect of Long Delays?

- Possible timeouts (for all designs)

# Effect of Duplication

- Produce duplicate ACKs
  - Could be confused for loss with cumulative ACKs
  - But duplication is rare...

# Possible Design For Reliable Transport

- Full information ACKs
- Window based, with retransmissions after
  - Timeout
  - $K$  subsequent ACKs
- This is correct, high-performant and high-utilization
- How about fairness?

# Fairness? (Come back to later)

- Adjust  $W$  based on losses...
- In a way that flows receive same shares
- Short version:
  - Loss: cut  $W$  by 2
  - Successful receipt of window:  $W$  increased by 1

# Overview of Reliable Transport

- Window based self control separate concerns
  - Size of  $W$
  - Nature of feedback
  - Response to loss
- Can design each aspect relatively independently
- Can be correct, fair, high-performant and high-utilization

# Many Implementation Choices

- Feedback from receiver: ACKs vs NACKs
  - Can NACKs alone achieve correctness
  - Can ACKs alone achieve correctness
- Variations on ACKs
  - Full information
  - Individual packets
  - Cumulative
- When to resend
  - Timeout
  - Duplicate ACKs
  - NACKs

# Implementation Choices

- These implementation choices affect:
  - Performance
  - Utilization
  - Fairness
  - ..
- These are important concerns
  - **But correctness is more fundamental**
- Design **must** start with correctness
  - Can then “engineer” its performance with various hacks
  - These hacks can be “fun”, but don’t let them distract you

# What Have We Done Today?

- Gone from first principles
  - Correctness condition for reliable transport
- ... to design for single packets
- ... to design for multiple packets
  - Very close to modern TCP
- ... to radically different designs
  - Which could replace TCP
- All done by **you**, in 75 minutes

