Chapter_3_v8.2 (1).pdf

Textbook Analysis

A.I. Companion

July 5, 2025

Of course. As an expert technical writer, I will rewrite the provided content slide by slide, ensuring each explanation is clear, concise, and follows a sequential order.

Slide 1: Chapter 3: Transport Layer Introduction

This slide introduces Chapter 3, which focuses on the Transport Layer of the computer networking model. It includes a usage note from the authors, James Kurose and Keith Ross, stating that the slides are freely available but request attribution if used.

Slide 2: Transport Layer: Overview

This slide outlines the main learning objectives for the chapter. The goal is to understand the core principles of transport layer services and to learn about the specific protocols the Internet uses.

The key principles to be covered are:

- **Multiplexing and Demultipiplexing:** How data from multiple applications is combined for sending and separated upon arrival.
- **Reliable Data Transfer:** How to ensure data arrives correctly and completely.
- **Flow Control:** How to prevent a sender from overwhelming a receiver.
- **Congestion Control:** How to manage network traffic to prevent slowdowns and data loss.

The specific Internet protocols to be examined are:

- **UDP (User Datagram Protocol):** A basic, connectionless service.
- **TCP (Transmission Control Protocol):** A reliable, connection-oriented service, including its method for managing network congestion.

Slide 3: Transport Layer: Roadmap

This slide provides a high-level outline of the topics that will be covered in this chapter, serving as a table of contents for the presentation. The topics will be presented in the following order:

1. Transport-layer services
2. Multiplexing and demultiplexing
3. Connectionless transport: UDP
4. Principles of reliable data transfer
5. Connection-oriented transport: TCP
6. Principles of congestion control
7. TCP congestion control
8. Evolution of transport-layer functionality

Slide 4: Transport Services and Protocols

This slide explains the fundamental role of the transport layer.

- **Logical Communication:** The transport layer provides a "logical" communication channel between application processes running on different computers (hosts). This means that from the applications' perspective, they are communicating directly with each other, even though the data is physically passing through many network devices.
- **Transport Protocol Actions:**
  - **On the sending device:** It takes messages from an application, breaks them into smaller pieces called **segments**, adds a transport header, and passes them to the network layer for delivery.
  - **On the receiving device:** It reassembles the received segments back into the original message and passes it to the correct application.
- **Internet Protocols:** The two main transport protocols available on the Internet are **TCP** and **UDP**.

Slide 5: Transport vs. Network Layer Services and Protocols

This slide uses an analogy to clarify the difference between the transport layer and the network layer.

Imagine two houses, Ann's and Bill's, each with 12 children. The children are sending letters to each other.

- **Hosts (Computers) = Houses:** The two physical locations.
- **Processes (Applications) = Kids:** The individual entities communicating.
- **Application Messages = Letters:** The actual data being sent.
- **Transport Protocol = Ann and Bill:** They collect letters from their kids (multiplexing) and deliver incoming mail to the correct child (demultiplexing). This is a process-to-process service within the house.
- **Network-Layer Protocol = Postal Service:** It handles the delivery of mail from one house to the other. This is a host-to-host service.

Slide 6: Transport vs. Network Layer Services and Protocols

This slide directly contrasts the roles of the transport and network layers based on the previous analogy.

- **Network Layer:** Provides communication between **hosts** (e.g., delivering a letter from Ann's house to Bill's house).
- **Transport Layer:** Provides communication between **processes** (e.g., ensuring a letter from a specific child in Ann's house gets to a specific child in Bill's house). It relies on the network layer's host-to-host delivery service and adds features to it.

Slide 7: Transport Layer Actions (Sender)

This slide illustrates the actions taken by the transport layer on the sending device.

When an application needs to send data, the transport layer protocol performs the following steps:

1. It receives a message from the application layer.
2. It creates a **segment** by adding a transport header (`T h`) to the application message. This header contains information needed for delivery, like port numbers.
3. It passes this newly created segment down to the network layer (IP layer) to be sent across the network.

Slide 8: Transport Layer Actions (Receiver)

This slide illustrates the actions taken by the transport layer on the receiving device, which is the reverse of the sender's process.

When a segment arrives from the network layer, the transport layer protocol does the following:

1. It receives the segment from the network (IP) layer.
2. It examines the header fields to check for errors and identify the recipient application.
3. It extracts the original application-layer message from the segment.
4. It delivers the message to the correct application process through a **socket**, a process known as demultiplexing.

Slide 9: Two Principal Internet Transport Protocols

This slide introduces the two main transport protocols used on the Internet: TCP and UDP.

- **TCP (Transmission Control Protocol):**
  - Provides **reliable, in-order delivery**, meaning data arrives without errors and in the correct sequence.
  - Includes **congestion control** and **flow control** to manage traffic and prevent overwhelming the receiver.
  - Requires a **connection setup** phase before data can be sent.
- **UDP (User Datagram Protocol):**
  - Provides **unreliable, unordered delivery**. There are no guarantees that data will arrive or that it will be in the correct order.
  - It is a simple, "no-frills" extension of the network layer's "best-effort" delivery service.
- **Services Not Available:** Neither protocol inherently offers guarantees on delivery time (delay) or network speed (bandwidth).

Slide 10: Chapter 3: Roadmap

This slide serves as a reminder of the chapter's structure, indicating that the next topic to be discussed is Multiplexing and Demultiplexing.

Slide 11: Multiplexing/Demultiplexing

This slide explains the concepts of multiplexing and demultiplexing at the transport layer.

- **Multiplexing (at the sender):** The process of gathering data chunks from different application processes (via sockets), adding a transport header to each chunk to identify it, and passing the resulting segments to the network layer. It's like a mailroom collecting letters from different departments and putting them into one mailbag.
- **Demultiplexing (at the receiver):** The process of receiving segments from the network layer and using the information in the transport header (like port numbers) to deliver each segment to the correct application process (socket). It's like the receiving mailroom sorting the mailbag and delivering each letter to the right department.

Slide 12: Visualizing Multiplexing

This slide, along with the next, uses an animation to show multiplexing in action. An HTTP message from a client application is encapsulated with a transport header (H_t) and then a network header (H_n) before being sent across the network. This visually demonstrates the sender's process of preparing data for transmission.

Slide 13: The Demultiplexing Question

This slide continues the animation from the previous slide. After a message arrives at the server, it poses a key question: How does the server's transport layer know which specific application process (e.g., the Firefox browser process vs. a Netflix or Skype process) should receive the incoming message?

Slide 14: The Answer: De-multiplexing

This slide begins to reveal the answer to the question posed on the previous slide. The process that allows the server to direct the message to the correct application is **de-multiplexing**.

Slide 15: De-multiplexing at the Transport Layer

This slide clarifies that de-multiplexing is a function that occurs at the transport and application layers, enabling the delivery of data to the correct process.

Slide 16: Demultiplexing

This slide simply titles the upcoming section, focusing on the process of demultiplexing.

Slide 17: (No readable text found on this slide)

This slide contains no content.

Slide 18: (No readable text found on this slide)

This slide contains no content.

Slide 19: Multiplexing

This slide titles the corresponding process that occurs on the sender's side: **multiplexing**.

Slide 20: Multiplexing at the Transport Layer

This slide clarifies that multiplexing is a function of the transport and application layers on the sending host, which gathers data from multiple applications to be sent over the network.

Slide 21: Multiplexing

This slide simply titles the upcoming section, focusing on the process of multiplexing.

Slide 22: How Demultiplexing Works

This slide explains the mechanics of demultiplexing. When a host receives an IP datagram (a network-layer packet), it uses two key pieces of information to deliver the data inside to the correct application socket:

- **IP Addresses (Source & Destination):** Found in the IP datagram header.
- **Port Numbers (Source & Destination):** Found in the transport-layer segment header.

Each transport-layer segment is carried within an IP datagram, and the host uses this combination of addresses and port numbers to identify the exact application process the data is intended for.

Slide 23: Connectionless Demultiplexing (UDP)

This slide describes how demultiplexing works for UDP, which is a connectionless protocol.

- When a UDP socket is created, it's bound to a specific port number on the host.
- When the host receives a UDP segment, it only looks at the **destination port number** in the segment's header.
- It then directs the segment to the socket that is bound to that specific port number.
- A key characteristic is that UDP datagrams from **different senders** (with different source IP addresses or source port numbers) will all be directed to the **same socket** as long as they are addressed to the same destination port number.

Slide 24: Connectionless Demultiplexing: An Example

This slide provides a visual example of connectionless (UDP) demultiplexing.

- A server process (P3) on machine B creates a UDP socket bound to port 6428.
- A client process (P1) on machine A sends a datagram to machine B on port 6428.
- The server's operating system receives this datagram and, by looking only at the destination port (6428), delivers it to process P3.
- If another client (say, from machine C) also sent a datagram to machine B on port 6428, it would also be delivered to the same process P3.

Slide 25: Connection-Oriented Demultiplexing (TCP)

This slide explains how demultiplexing works for TCP, a connection-oriented protocol.

- A TCP socket is identified not just by a port number, but by a **4-tuple** of values:
    1. Source IP address
    2. Source port number
    3. Destination IP address
    4. Destination port number
- When a host receives a TCP segment, it uses **all four** of these values to determine which socket should receive it.
- This allows a server to handle many simultaneous TCP connections from different clients, even if they are all connecting to the same destination port (e.g., port 80 for a web server). Each connection will have a unique 4-tuple because the client's source IP or source port will be different.

Slide 26: Connection-Oriented Demultiplexing: Example

This slide shows an example of connection-oriented (TCP) demultiplexing.

- A server is running on IP address B, listening for connections on port 80.
- Three different segments arrive at server B, all destined for port 80.
    1. From client A, port 9157
    2. From client C, port 5775

    3. From client C, port 9157

- Even though they all target the same destination port (80), the server uses the complete 4-tuple (source IP, source port, dest IP, dest port) to demultiplex them to three **different** sockets, one for each unique connection.

Slide 27: Summary of Multiplexing/Demultiplexing

This slide summarizes the key concepts of multiplexing and demultiplexing.

- The process relies on header fields in segments (transport layer) and datagrams (network layer).
- **UDP** uses only the **destination port number** to demultiplex data.
- **TCP** uses a **4-tuple** (source/destination IP addresses and source/destination port numbers) to demultiplex data.
- Multiplexing and demultiplexing are fundamental processes that happen at all layers of the network stack, not just the transport layer.

Slide 28: Chapter 3: Roadmap

This slide indicates the presentation is moving to the next topic in the chapter: Connectionless Transport with UDP.

Slide 29: UDP: User Datagram Protocol

This slide details the characteristics of the User Datagram Protocol (UDP).

- **Core Features:**
    - It is a "no-frills" or "bare bones" protocol.
    - It provides a "best-effort" service, meaning segments can be lost or delivered out of order.
    - It is **connectionless**, meaning there is no initial "handshake" to set up a connection, which avoids delay.
    - It is **simple**, with no connection state to maintain at the sender or receiver.
    - It has a **small header**, adding minimal overhead.
    - It has **no congestion control**, meaning it will send data as fast as the application wants, regardless of network congestion.

Slide 30: UDP: User Datagram Protocol (Use Cases)

This slide explains why and where UDP is used.

- **Common Applications:**
    - **Streaming multimedia:** Applications that are sensitive to delay but can tolerate some data loss.

    - **DNS (Domain Name System):** A simple, fast query-response protocol.
    - **SNMP (Simple Network Management Protocol):** Used for network monitoring.
    - **HTTP/3:** The newest version of the web protocol, which builds its own reliability on top of UDP.
- **Adding Reliability:** If an application needs reliable data transfer but wants to use UDP (perhaps for more control), it must implement reliability and congestion control features itself at the application layer.

Slide 31: UDP: User Datagram Protocol [RFC 768]

This slide shows a diagram illustrating that a UDP segment consists of a short UDP header followed by the application data. It also references RFC 768, the official standards document that defines UDP.

Slide 32: UDP: Transport Layer Actions

This slide introduces a scenario with an SNMP client and server to illustrate the actions of UDP. SNMP (Simple Network Management Protocol) is an application-layer protocol that commonly uses UDP for transport. The diagram shows the protocol stack on both the client and server.

Slide 33: UDP: Transport Layer Actions (Sender)

This slide details the actions of the UDP sender.

1. It is passed a message (e.g., an SNMP message) from the application layer.
2. It determines the values for the UDP header fields (like source and destination ports).
3. It creates a UDP segment by attaching the UDP header (`UDP h`) to the application message.
4. It passes the complete segment to the IP layer for transmission.

Slide 34: UDP: Transport Layer Actions (Receiver)

This slide details the actions of the UDP receiver.

1. It receives a segment from the IP layer.
2. It checks the UDP checksum field in the header to verify data integrity (i.e., check for errors).
3. It extracts the application-layer message (e.g., the SNMP message) from the segment.
4. It demultiplexes the message, delivering it to the correct application via its socket.

Slide 35: UDP Segment Header

This slide breaks down the fields in the UDP header.

- **Source Port # & Dest Port #:** Identify the sending and receiving application processes.
- **Length:** Specifies the total length of the UDP segment (header plus data) in bytes.
- **Checksum:** An optional field used to check for bit errors in the segment.
- **Application Data (payload):** The data passed down from the application layer.

The entire header is 32 bits (4 bytes) wide, making it very lightweight.

Slide 36: UDP Checksum

This slide explains the purpose of the checksum.

- **Goal:** To detect errors, such as flipped bits, that may have occurred during transmission.
- **Process:** The sender calculates a checksum value based on the segment's content. The receiver performs the same calculation on the received segment. If the receiver's calculated checksum doesn't match the value in the checksum field, it knows the data has been corrupted.

Slide 37: Internet Checksum

This slide describes how the Internet checksum is calculated for UDP.

- **Sender:**
    1. Treats the contents of the UDP segment (including parts of the IP header for context) as a sequence of 16-bit integers.
    2. Calculates the "one's complement sum" of all these 16-bit integers. This result is the checksum.
    3. Places this checksum value into the UDP header's checksum field.
- **Receiver:**
    1. Performs the same one's complement sum calculation on the received segment.
    2. Compares its result to the value in the checksum field. If they don't match, an error is detected. If they do match, it's assumed there are no errors, although it's possible for errors to go undetected.

Slide 38: Internet Checksum: An Example

This slide shows an example of the checksum calculation using 16-bit binary numbers. It highlights a key feature of one's complement addition: if the addition of two numbers results in a carry-out from the most significant bit, that carry bit must be "wrapped around" and added to the least significant bit of the result.

Slide 39: Internet Checksum: Weak Protection!

This slide demonstrates a weakness of the Internet checksum. It shows an example where two different bit flips occur in the data, but they cancel each other out in the checksum calculation. The final checksum remains the same, and the error goes undetected. This illustrates that while the checksum is helpful, it is not a foolproof error detection mechanism.

Slide 40: Summary: UDP

This slide summarizes the key aspects of UDP.

- It's a "no-frills" protocol providing a "best-effort" service, meaning segments can be lost or arrive out of order.
- It has several advantages:
    - No connection setup is required, avoiding a round-trip time (RTT) delay.
    - It can still function even when network service is degraded.
    - The checksum provides a basic level of error detection.
- If more advanced features like reliability are needed, they must be built into the application layer on top of UDP (e.g., as done in HTTP/3).

Slide 41: Chapter 3: Roadmap

This slide indicates the presentation is moving to the next major topic: the Principles of Reliable Data Transfer.

Slide 42: Principles of Reliable Data Transfer (Abstraction)

This slide introduces the concept of reliable data transfer (RDT) from an abstract viewpoint. The goal is to provide a reliable service to the application layer, making it seem as though data is being sent over a perfect, reliable channel, even though the underlying network is inherently unreliable.

Slide 43: Principles of Reliable Data Transfer (Implementation)

This slide shows the implementation view of achieving reliable data transfer. A reliable data transfer protocol is created on both the sender and receiver sides. These two protocol components work together over the underlying unreliable channel to create the illusion of the reliable service that the application layer sees.

Slide 44: Principles of Reliable Data Transfer (Complexity)

This slide adds an important point: the complexity of the reliable data transfer (RDT) protocol depends heavily on the characteristics of the underlying unreliable channel. For example, a protocol designed for a channel that only corrupts bits will be simpler than one for a channel that can also lose and reorder packets.

Slide 45: Principles of Reliable Data Transfer (State)

This slide highlights a fundamental challenge in creating a reliable protocol. The sender and receiver operate independently and do not inherently know what is happening on the other side. For instance, the sender doesn't know if its message was successfully received unless the receiver communicates that information back to it. This lack of shared state necessitates a protocol with explicit messaging.

Slide 46: Reliable Data Transfer Protocol (RDT): Interfaces

This slide defines the common function calls (interfaces) used to describe a reliable data transfer protocol.

- `rdt_send()`: Called by the application to send data. The protocol takes this data to prepare it for transmission.
- `udt_send()`: Called by the RDT protocol itself to send a packet over the **u**nreliable **d**ata **t**ransfer channel.
- `rdt_rcv()`: Called when a packet arrives from the unreliable channel on the receiver's side.
- `deliver_data()`: Called by the receiver's RDT protocol to deliver the correct data to the application layer.

Slide 47: Reliable Data Transfer: Getting Started

This slide outlines the approach for the following section.

- The sender and receiver protocols will be developed **incrementally**, starting with the simplest case and gradually adding complexity.
- The focus will be on **unidirectional** data transfer (from sender to receiver), although control information (like acknowledgements) will flow in both directions.
- **Finite State Machines (FSMs)** will be used to formally specify the behavior of the sender and receiver, showing their states, the events that trigger transitions, and the actions taken.

Slide 48: rdt1.0: Reliable Transfer over a Reliable Channel

This slide describes the first and simplest version of the protocol, `rdt1.0`.

- **Assumption:** The underlying channel is perfectly reliable (no bit errors, no packet loss).
- **Sender FSM:** Has one state. When the application calls `rdt_send(data)`, the sender simply creates a packet and sends it using `udt_send(packet)`.
- **Receiver FSM:** Has one state. When a packet arrives (`rdt_rcv(packet)`), the receiver extracts the data and delivers it to the application using `deliver_data(data)`.

In this ideal scenario, no error handling is needed.

Slide 49: rdt2.0: Channel with Bit Errors

This slide introduces the first element of unreliability: the channel can now corrupt packets (flip bits). It poses the question of how to recover from such errors, drawing an analogy to how humans handle misunderstandings in a conversation (e.g., "What did you say?").

Slide 50: rdt2.0: Channel with Bit Errors (Solution)

This slide presents the solution for handling bit errors in `rdt2.0`.

- **Error Detection:** A checksum is used to detect if a packet has been corrupted.
- **Feedback Mechanism:** The receiver sends feedback to the sender.
  - **Acknowledgements (ACKs):** A message indicating the packet was received correctly.
  - **Negative Acknowledgements (NAKs):** A message indicating the packet was received but was corrupted.
- **Retransmission:** If the sender receives a NAK, it **retransmits** the last packet.
- This protocol is known as **stop-and-wait**, where the sender sends one packet and then waits for a response (ACK or NAK) before sending the next one.

Slide 51: rdt2.0: FSM Specifications

This slide shows the Finite State Machine (FSM) diagrams for the `rdt2.0` sender and receiver.

- **Sender:** Starts in a "Wait for call from above" state. When it sends a packet, it transitions to a "Wait for ACK or NAK" state.
  - If it receives an ACK, it returns to the initial state, ready to send the next piece of data.
  - If it receives a NAK, it retransmits the same packet and stays in the "Wait for ACK or NAK" state.
- **Receiver:** Has one state, "Wait for call from below."
  - If a non-corrupt packet arrives, it delivers the data and sends an ACK.
  - If a corrupt packet arrives, it sends a NAK.

Slide 52: rdt2.0: FSM Specification (State Knowledge)

This slide reiterates a key concept using the `rdt2.0` FSMs. The sender has no direct knowledge of the receiver's state (e.g., whether the last message was received correctly). It can only infer this state based on the feedback messages (ACKs and NAKs) it receives. This is the fundamental reason a protocol is necessary to coordinate their actions.

Slide 53: rdt2.0: Operation with No Errors

This slide illustrates the ideal operation of `rdt2.0` when no errors occur.

1. The sender sends a packet.
2. The receiver receives the packet, finds it's not corrupt, delivers the data, and sends an ACK.
3. The sender receives the ACK and is ready to send the next packet.

Slide 54: rdt2.0: Corrupted Packet Scenario

This slide shows how `rdt2.0` handles a corrupted data packet.

1. The sender sends a packet.
2. The packet gets corrupted during transmission.
3. The receiver receives the packet, detects the corruption using the checksum, and sends a NAK.
4. The sender receives the NAK, retransmits the original packet, and waits again for an ACK or NAK.

Slide 55: rdt2.0 Has a Fatal Flaw!

This slide points out a critical flaw in `rdt2.0`: what happens if the ACK or NAK message itself gets corrupted? The sender won't know what happened at the receiver.

- **Problem:** If an ACK is corrupted, the sender might retransmit the packet unnecessarily. This could lead to the receiver getting a **duplicate** packet.
- **Solution:** To handle duplicates, the sender must add a **sequence number** to each packet. This allows the receiver to identify whether an incoming packet is a new one or a duplicate of one it has already processed. If it's a duplicate, the receiver can simply discard it.

Slide 56: rdt2.1: Sender, Handling Garbled ACK/NAKs

This slide shows the sender's FSM for `rdt2.1`, which incorporates sequence numbers (0 and 1) to handle corrupted ACKs/NAKs.

- The sender now has two main sending states: "Wait for call 0 from above" and "Wait for call 1 from above".
- After sending packet 0, it waits for ACK or NAK 0. If it receives a corrupted or NAK response, it retransmits packet 0. If it receives a clean ACK 0, it moves to the state where it can send packet 1.
- The logic is mirrored for sending packet 1. This alternating sequence (0, 1, 0, 1...) allows the sender and receiver to stay synchronized.

Slide 57: rdt2.1: Receiver, Handling Garbled ACK/NAKs

This slide shows the receiver's FSM for `rdt2.1`.

- The receiver also has two states: "Wait for 0 from below" and "Wait for 1 from below".

- In the "Wait for 0" state, if it receives a non-corrupt packet with sequence number 0, it delivers the data, sends an ACK 0, and transitions to the "Wait for 1" state.
- If it receives a packet with the wrong sequence number (e.g., receives packet 1 when expecting 0), it knows this is a duplicate transmission, so it just re-sends the ACK for the last correctly received packet (ACK 0 in this case) and stays in its current state.
- If a corrupt packet arrives, it sends a NAK.

Slide 58: rdt2.1: Discussion

This slide summarizes the changes and logic in `rdt2.1`.

- **Sender:**
  - Adds a sequence number (0 or 1 is sufficient for stop-and-wait) to each packet.
  - Must check if an incoming ACK/NAK is corrupted.
  - The FSM has twice as many states because it must remember the expected sequence number of the next packet.
- **Receiver:**
  - Must check if a received packet is a duplicate by looking at its sequence number.
  - Its state indicates which sequence number it is currently expecting (0 or 1).
  - Crucially, the receiver can't know if its last ACK/NAK was successfully received by the sender, which is why it must be prepared to handle retransmissions.

Slide 59: rdt2.2: A NAK-Free Protocol

This slide introduces an improved protocol, `rdt2.2`, that eliminates the need for NAK messages.

- **Functionality:** It performs the same role as `rdt2.1` but only uses ACKs.
- **Mechanism:**
  - Instead of sending a NAK when it receives a corrupted packet, the receiver simply sends an **ACK for the last packet it received correctly**.
  - The receiver must include the sequence number of the packet being ACKed (e.g., "ACK 0" or "ACK 1").
- **Sender's Action:** When the sender receives a **duplicate ACK** (e.g., it sent packet 1 but gets another "ACK 0"), it knows something went wrong with packet 1 and retransmits it. This has the same effect as receiving a NAK. TCP uses this NAK-free approach.

Slide 60: rdt2.2: Sender, Receiver Fragments

This slide shows parts of the FSMs for `rdt2.2` to illustrate the NAK-free logic.

- **Sender FSM Fragment:** When waiting for ACK 0, if it receives a valid ACK 0, it moves on. If it receives a corrupted response or an ACK for packet 1 (which shouldn't happen yet), it knows

something is wrong. (The diagram seems to have a typo, `isACK(rcvpkt,1)` should likely be `isACK(rcvpkt,0)` for the duplicate ACK case, causing a retransmission). The key idea is that receiving an ACK for the *wrong* packet triggers a retransmission.
- **Receiver FSM Fragment:** When waiting for packet 0, if a corrupted packet or a packet with sequence number 1 arrives, the receiver sends back an ACK for the last good packet it saw (in this case, an ACK for a previous packet, labeled `ACK1` here which implies the previous good packet was #1). It essentially says "I'm still waiting for packet 0".

Slide 61: rdt3.0: Channels with Errors and Loss

This slide introduces a new, more realistic problem: the underlying channel can now **lose packets entirely**, in addition to corrupting them. This applies to both data packets (sender to receiver) and ACK packets (receiver to sender). The protocol now needs a way to handle the complete absence of a response.

Slide 62: rdt3.0: Channels with Errors and Loss (Solution)

This slide presents the solution for handling lost packets: a **timeout mechanism**.

- **Approach:** The sender starts a timer after sending a packet. It waits a "reasonable" amount of time for an ACK.
- If the timer expires before an ACK is received, the sender assumes the packet (or its corresponding ACK) was lost and **retransmits** the packet.
- This approach also handles cases where a packet is just severely delayed. The retransmission will be a duplicate, but the existing sequence number mechanism can handle that.
- The use of a countdown timer is essential for implementing this timeout.

Slide 63: rdt3.0 Sender FSM

This slide shows the sender's Finite State Machine for `rdt3.0`, now including the timer logic.

- When the sender sends a packet (e.g., packet 0), it also calls `start_timer`.
- If it receives the correct ACK (e.g., ACK 0) before the timer expires, it calls `stop_timer` and transitions to the next state (e.g., ready to send packet 1).
- If the timer expires (`timeout` event), it retransmits the packet and restarts the timer.

Slide 64: rdt3.0 Sender FSM (Complete)

This slide shows the complete sender FSM for `rdt3.0`, including all transitions for timeouts and corrupted ACKs.

- **New Event:** The `timeout` event is now a possible trigger in each "Wait for ACK" state.
- **Action on Timeout:** If a timeout occurs, the sender retransmits the current packet (`udt_send(sndpkt)`) and restarts the timer (`start_timer`).
- **Other Events:** The logic for handling corrupted or out-of-order ACKs remains, which also triggers a retransmission. Essentially, any event other than receiving the correct ACK results in a retransmission or no action.

Slide 65: rdt3.0 in Action (Packet Loss)

This slide visually demonstrates `rdt3.0` in two scenarios.

- **(a) No Loss:** The normal sequence of send-ack, send-ack proceeds smoothly.
- **(b) Packet Loss:**
    1. Sender sends `pkt1`.
    2. `pkt1` is lost in the network.
    3. Receiver never gets `pkt1`, so it sends no ACK.
    4. Sender's timer for `pkt1` expires.
    5. Sender retransmits `pkt1`.
    6. The retransmitted `pkt1` is received, and the process continues.

Slide 66: rdt3.0 in Action (ACK Loss & Premature Timeout)

This slide demonstrates two more complex `rdt3.0` scenarios.

- **(c) ACK Loss:**
    1. Sender sends `pkt1`, and receiver sends `ack1`.
    2. `ack1` is lost in the network.
    3. Sender's timer for `pkt1` expires.
    4. Sender retransmits `pkt1`.
    5. Receiver gets the duplicate `pkt1`, detects it's a duplicate (via sequence number), discards it, and re-sends `ack1`.
- **(d) Premature Timeout:**
    1. Sender sends `pkt1`, and receiver sends `ack1`.
    2. The `ack1` is delayed, but not lost.
    3. Sender's timer expires too early, and it retransmits `pkt1`.
    4. The receiver gets the duplicate `pkt1` and sends another `ack1`.
    5. The sender eventually receives both the original delayed `ack1` and the new `ack1`. It processes the first and ignores the second as a duplicate.

Slide 67: Performance of rdt3.0 (Stop-and-Wait)

This slide analyzes the performance of the stop-and-wait protocol (`rdt3.0`) and shows that it is very inefficient.

- **Scenario:** A 1 Gbps link with a 15 ms propagation delay, sending 8000-bit packets.
- **Transmission Time (`D_trans`):** The time it takes to put the packet on the wire is calculated as `L/R` (packet length / link rate), which is only 8 microseconds.
- **Sender Utilization (`U_sender`):** This is the fraction of time the sender is actually busy sending data. The rest of the time it is idle, waiting for an ACK. The slide sets up the calculation, showing that the tiny transmission time is a small fraction of the total round-trip time.

Slide 68: rdt3.0: Stop-and-Wait Operation (Visual)

This slide provides a timeline visualization of the stop-and-wait process.

1. The sender begins transmitting the packet at t=0.
2. The last bit of the packet is sent a short time later (`L/R`).
3. The packet travels across the network (propagation delay).
4. The receiver gets the packet, processes it, and sends an ACK.
5. The ACK travels back across the network.
6. The sender finally receives the ACK at `t = RTT + L/R` and can send the next packet. The sender is only active for the small `L/R` portion of this entire cycle.

Slide 69: rdt3.0: Stop-and-Wait Operation (Calculation)

This slide completes the performance calculation from slide 67.

- **Total Cycle Time:** The time from sending one packet to being able to send the next is the Round Trip Time (RTT) plus the transmission time (`RTT + L/R`). In this example, RTT is 30ms and L/R is 8μs (0.008ms).
- **Utilization Formula:** `U_sender = (L/R) / (RTT + L/R)`
- **Result:** The utilization is `0.008ms / 30.008ms ≈ 0.00027`, or about **0.027%**.
- **Conclusion:** The stop-and-wait protocol performs terribly on modern high-speed, long-delay networks because it fails to utilize the available channel capacity.

Slide 70: Pipelined Protocols Operation

This slide introduces **pipelining** as the solution to the poor performance of stop-and-wait protocols.

- **Pipelining:** The sender is allowed to send multiple packets ("in-flight") without waiting for each one to be acknowledged first.
- **Requirements:**
  - The range of sequence numbers must be increased to track all the in-flight packets.

    - The sender and/or receiver must have buffers to store these multiple packets.

Slide 71: Pipelining: Increased Utilization

This slide shows how pipelining improves performance.

- By allowing 3 packets to be "in-flight" at once, the sender can continuously send data while the first packet's ACK is traveling back.
- The new utilization is `(3 * L/R) / (RTT + L/R)`, which is 3 times higher than with stop-and-wait.
- In the example, this increases utilization from 0.027% to about 0.081%. While still low, it demonstrates that sending more packets in-flight directly improves performance.

Slide 72: Go-Back-N: Sender

This slide describes the sender's behavior in the **Go-Back-N (GBN)** pipelining protocol.

- **Sending Window:** The sender maintains a "window" of up to `N` consecutive packets that it is allowed to send without yet having received an acknowledgement.
- **Cumulative ACK:** GBN uses cumulative acknowledgements. An `ACK(n)` confirms that the receiver has correctly received all packets up to and including sequence number `n`. When the sender gets `ACK(n)`, it can slide its window forward to start at `n+1`.
- **Timeout:** The sender uses a single timer for the oldest unacknowledged packet in the window. If this timer expires, it assumes that packet and all subsequent packets in the window were lost. It then **retransmits all packets** in the current window.

Slide 73: Go-Back-N: Receiver

This slide describes the receiver's behavior in the GBN protocol.

- **Simplicity:** The GBN receiver is very simple. It only keeps track of one variable: `rcv_base`, the sequence number of the next in-order packet it is expecting.
- **In-Order Packets:** If a packet with sequence number `rcv_base` arrives and is not corrupt, the receiver accepts it, delivers it to the application, and increments `rcv_base`. It then sends a cumulative ACK for this new `rcv_base - 1`.
- **Out-of-Order Packets:** If any other packet arrives (out-of-order or corrupt), the receiver **discards it**. It then re-sends the ACK for the last correctly received in-order packet. This tells the sender which packet it is still waiting for. The receiver does not buffer out-of-order packets.

Slide 74: Go-Back-N in Action

This slide shows a visual example of the GBN protocol with a window size of N=4.

1. Sender sends packets 0, 1, 2, and 3.
2. Packet 2 is lost in the network.
3. Receiver gets packets 0 and 1, sending back `ack0` and `ack1`.
4. Receiver gets packet 3, but since it's expecting packet 2, it discards packet 3 and re-sends `ack1`. It does the same for packets 4 and 5.
5. The sender's timer for packet 2 expires.
6. The sender goes back and retransmits everything in its current window: packets 2, 3, 4, and 5.
7. These packets now arrive in order at the receiver, and the process continues.

Slide 75: Selective Repeat: The Approach

This slide introduces a more efficient pipelining protocol called **Selective Repeat (SR)**.

- **Improvement over GBN:** SR avoids the unnecessary retransmissions of GBN. The sender only retransmits the single packet that was actually lost.
- **Receiver Behavior:** The receiver individually acknowledges every correctly received packet. It also **buffers** correctly received but out-of-order packets until the missing packets arrive.
- **Sender Behavior:** The sender maintains a conceptual timer for each unacknowledged packet. If a timer expires, it retransmits only that specific packet.

Slide 76: Selective Repeat: Sender, Receiver Windows

This slide visualizes the sequence number space from the perspective of the SR sender and receiver.

- **Sender Window:** Shows packets that have been sent and ACKed, sent but not yet ACKed, are usable for new data, or are not usable yet.
- **Receiver Window:** Shows packets that have been received and ACKed, are expected, have been received out-of-order and are now buffered, or are not yet acceptable.

Both sides maintain a window of `N` consecutive sequence numbers.

Slide 77: Selective Repeat: Sender and Receiver Logic

This slide details the logic for the SR sender and receiver.

- **Sender:**
    - Sends a packet if its sequence number is within the sending window.
    - On timeout for packet `n`, retransmits only packet `n`.
    - When `ACK(n)` is received, it marks packet `n` as received. If `n` is the oldest un-ACKed packet, the window slides forward.
- **Receiver:**

- If packet `n` is within the receiving window, it sends `ACK(n)`.
- If the packet is in-order, it's delivered to the application (along with any buffered packets that now follow it), and the window slides forward.
- If the packet is out-of-order, it is buffered.
- If the packet is for a previously ACKed slot (below the window), it still sends an ACK to be safe.

Slide 78: Selective Repeat in Action

This slide provides a visual example of Selective Repeat with N=4.

1. Sender sends packets 0, 1, 2, and 3.
2. Packet 2 is lost.
3. Receiver gets 0 and 1 (sends `ack0`, `ack1`). It gets 3, 4, and 5, buffers them, and sends individual ACKs (`ack3`, `ack4`, `ack5`).
4. Sender's timer for packet 2 expires.
5. Sender retransmits **only packet 2**.
6. Receiver gets packet 2. It can now deliver packets 2, 3, 4, and 5 to the application in the correct order.

This is much more efficient than GBN, which would have re-sent packets 2, 3, 4, and 5.

Slide 79: Selective Repeat: A Dilemma!

This slide presents a critical problem with Selective Repeat if the window size (`N`) is not chosen carefully in relation to the sequence number space.

It shows two scenarios that are indistinguishable from the receiver's perspective:

- **(a) No Loss:** Packets 0, 1, 2 are sent and ACKed. The sender's window slides forward.
- **(b) All ACKs Lost:** The sender sends packets 0, 1, 2. All ACKs are lost. The sender times out, retransmits packet 0. From the receiver's view, its window has already advanced. The retransmitted packet 0, with sequence number 0, falls into the *new* window and is mistaken for a brand new packet, not a duplicate.

Slide 80: Selective Repeat: A Dilemma! (Continued)

This slide emphasizes the problem from the previous slide.

- **The Problem:** The receiver cannot tell the difference between the retransmitted packet 0 in scenario (b) and a brand new packet 0 that would be sent much later in a normal sequence (after the sequence numbers wrap around).
- **The Cause:** The sequence number space is too small relative to the window size.

- **The Question:** What relationship must hold between the sequence number space and the window size to prevent this ambiguity? The unstated rule is that the **sequence number space must be at least twice the window size.**

Slide 81: Chapter 3: Roadmap

This slide indicates that the presentation is moving on to discuss the **Transmission Control Protocol (TCP)**, which implements many of the reliable data transfer principles just covered.

Slide 82: TCP: Overview

This slide provides a high-level overview of the key features of TCP.

- **Point-to-point:** A connection involves one sender and one receiver.
- **Reliable, in-order byte stream:** Data is treated as a continuous stream of bytes, and TCP ensures it arrives completely and in the correct order. There are no "message boundaries" like in UDP.
- **Pipelining:** TCP uses a sliding window to allow multiple segments in-flight, with the window size managed by flow and congestion control.
- **Full duplex:** Data can flow in both directions simultaneously within the same connection.
- **Connection-oriented:** It requires a three-way handshake to establish a connection before data is exchanged.
- **Flow controlled:** The sender will not send data faster than the receiver can process it.

Slide 83: TCP Segment Structure

This slide shows the structure of a TCP header and explains its important fields.

- **Source Port # / Dest Port #:** Identify the sending and receiving applications.
- **Sequence Number:** The byte-stream number of the *first byte* of data in this segment.
- **Acknowledgement Number:** If the ACK bit is set, this is the sequence number of the *next byte* the sender is expecting from the other side.
- **Head Len:** The length of the TCP header (as it can have variable-length options).
- **Flags (RST, SYN, FIN, ACK, C, E, U, P):** Control bits used for connection management (e.g., `SYN` to start, `FIN` to close), acknowledgements (`ACK`), and congestion notification (`C` , `E`).
- **Receive Window:** Used for flow control; it specifies how many bytes the receiver is currently able to accept.
- **Checksum:** Used for error detection.

Slide 84: TCP Sequence Numbers, ACKs

This slide explains how TCP uses sequence numbers and acknowledgements.

- **Sequence Numbers:** They don't count segments; they count **bytes** in the data stream. If a segment contains 100 bytes of data and its sequence number is 500, the bytes are numbered 500 through 599.
- **Acknowledgements (ACKs):** They are **cumulative**. When a receiver sends an ACK with acknowledgement number `Y`, it is confirming that it has correctly received all bytes up to `Y-1` and is now expecting byte `Y`.
- **Out-of-Order Segments:** The official TCP specification does not dictate how a receiver should handle out-of-order segments; this decision is left to the implementor of the TCP stack.

Slide 85: TCP Sequence Numbers, ACKs (Example)

This slide illustrates the use of sequence and acknowledgement numbers in a simple Telnet (remote terminal) scenario.

1. **Host A to Host B:** User types 'C'. Host A sends a segment with `Seq=42`, `ACK=79`, and one byte of data ('C'). The sequence number is 42, and it is acknowledging receipt of all data up to byte 78 from Host B.
2. **Host B to Host A:** Host B receives the 'C' and acknowledges it. It also echoes the 'C' back. It sends a segment with `Seq=79`, `ACK=43`, and data='C'. The sequence number is 79 (its own data stream), and the ACK number is 43, confirming receipt of the byte at sequence number 42.
3. **Host A to Host B:** Host A receives the echoed 'C'. It sends an empty ACK segment with `Seq=43`, `ACK=80`. This acknowledges receipt of the byte at sequence number 79.

Slide 86: TCP Round Trip Time, Timeout

This slide addresses the critical question of how to set the retransmission timeout value for TCP.

- **The Challenge:** The timeout must be longer than the current Round Trip Time (RTT), but the RTT can vary significantly.
  - **Too Short:** Leads to premature timeouts and unnecessary retransmissions, wasting bandwidth.
  - **Too Long:** Leads to a slow reaction to actual segment loss, hurting performance.
- **Estimating RTT:** TCP measures the time from when a segment is sent until its ACK is received. This is called `SampleRTT`. To avoid reacting to every minor fluctuation, TCP calculates a smoothed, average RTT based on recent measurements.

Slide 87: TCP Round Trip Time, Timeout (Calculation)

This slide shows how TCP calculates a smoothed RTT.

- **`EstimatedRTT`:** It is calculated using an **Exponential Weighted Moving Average (EWMA)**.
  - `EstimatedRTT = (1 - α) * EstimatedRTT + α * SampleRTT`

- This formula gives more weight to recent `SampleRTT` values while still considering past history. The influence of older samples decreases exponentially. A typical value for `α` (alpha) is 0.125.
- The graph shows how the `EstimatedRTT` (red line) is a much smoother, more stable value than the highly variable individual `SampleRTT` measurements (blue dots).

Slide 88: TCP Round Trip Time, Timeout (Final Calculation)

This slide explains how the final `TimeoutInterval` is determined.

- **Safety Margin:** To be safe, the timeout isn't set to just `EstimatedRTT`. A safety margin is added to account for variations in RTT.
- **`DevRTT`:** TCP also calculates the deviation of the RTT (`DevRTT`), which is an EWMA of how much `SampleRTT` differs from `EstimatedRTT`.
    - `DevRTT = (1 - β) * DevRTT + β * |SampleRTT - EstimatedRTT|`
- **`TimeoutInterval` Formula:**
    - `TimeoutInterval = EstimatedRTT + 4 * DevRTT`
- This ensures that when RTT is highly variable, the safety margin is larger, making premature timeouts less likely.

Slide 89: TCP Sender (Simplified)

This slide provides a simplified, event-based overview of the TCP sender's logic for reliable data transfer.

- **Event: Data received from application:**
    - Create a segment with the correct sequence number.
    - If no timer is currently running, start one. The timer is for the oldest unacknowledged segment.
- **Event: Timeout occurs:**
    - Retransmit the segment that caused the timeout (the oldest unacknowledged one).
    - Restart the timer.
- **Event: ACK received:**
    - If the ACK confirms receipt of new data, update the record of which segments have been acknowledged.
    - If there are still unacknowledged segments in flight, restart the timer.

Slide 90: TCP Receiver: ACK Generation

This slide describes the TCP receiver's strategy for sending ACKs, as recommended by RFC 5681.

- **In-order segment arrives, no ACKs pending:** Use a **delayed ACK**. Wait a short period (e.g., up to 500ms) to see if another segment arrives. If so, you can acknowledge both with a single cumulative ACK. If not, send the ACK.

- **In-order segment arrives, another ACK is pending:** Send a single cumulative ACK immediately to acknowledge both segments.
- **Out-of-order segment arrives (gap detected):** Immediately send a **duplicate ACK** for the last in-order byte received. This signals to the sender which segment is missing.
- **Segment arrives that fills a gap:** Immediately send an ACK for the newly contiguous block of data.

Slide 91: TCP: Retransmission Scenarios

This slide illustrates two common TCP retransmission scenarios.

- **Lost ACK Scenario:**
    1. Host A sends data (Seq=92). Host B receives it and sends ACK=100.
    2. The ACK=100 is lost.
    3. Host A's timer expires, so it retransmits the data (Seq=92).
    4. Host B receives the duplicate, discards it, and re-sends ACK=100.
- **Premature Timeout:**
    1. Host A sends data (Seq=92). Host B receives it and sends ACK=100.
    2. Host A also sends more data (Seq=100). Host B receives it and sends ACK=120.
    3. The first ACK (100) is delayed. Host A's timer for Seq=92 expires too early.
    4. Host A retransmits Seq=92.
    5. Shortly after, the delayed ACK=120 arrives at Host A. This single cumulative ACK confirms receipt of everything up to byte 119, so Host A knows the retransmission was unnecessary and can move on.

Slide 92: TCP: Retransmission Scenarios (Cumulative ACK)

This slide shows how cumulative ACKs can make the protocol more efficient.

1. Host A sends data at Seq=92. Host B receives it and sends ACK=100.
2. ACK=100 is lost.
3. In the meantime, Host A sends more data at Seq=100. Host B receives this and sends a new cumulative ACK=120.
4. This ACK=120 arrives at Host A. Since 120 is greater than 100, this single ACK confirms receipt of *both* the segment at Seq=92 and the one at Seq=100. The lost ACK becomes irrelevant.

Slide 93: TCP Fast Retransmit

This slide explains an important optimization called **Fast Retransmit**.

- **Problem:** Waiting for a timeout to detect a lost packet can be slow.

- **Mechanism:** If a sender receives **three duplicate ACKs** for the same sequence number, it is a strong indicator that the subsequent packet has been lost, and the receiver is successfully receiving later packets.
- **Action:** Instead of waiting for the timer to expire, the sender immediately retransmits the unacknowledged segment with the smallest sequence number. This significantly speeds up recovery from single packet losses.

Slide 94: Chapter 3: Roadmap

This slide indicates the presentation is now moving to the topic of **TCP Flow Control**, a distinct concept from reliable data transfer.

Slide 95: TCP Flow Control (The Problem)

This slide poses a question to introduce the concept of flow control. It shows data arriving from the network into a TCP socket buffer, and an application process reading data out of that buffer. The question is: What happens if the network delivers data faster than the application reads it?

Slide 96: TCP Flow Control (Buffer Overflow)

This slide illustrates the answer to the previous question. If the sender sends data too quickly, the receiver's socket buffer will fill up and eventually overflow, leading to data loss.

Slide 97: TCP Flow Control (The Solution in the Header)

This slide points to the solution within the TCP header. The **receive window** (`rwnd`) field is used for flow control. It's the mechanism by which the receiver tells the sender how much buffer space it has available.

Slide 98: TCP Flow Control (The Mechanism)

This slide defines flow control. It is a mechanism that allows the receiver to control the sender's transmission rate. This ensures the sender doesn't transmit too much data too fast and overflow the receiver's buffer.

Slide 99: TCP Flow Control (Implementation)

This slide explains how TCP implements flow control.

- The TCP receiver "advertises" its available buffer space in the **`rwnd`** field of every TCP segment it sends back to the sender.

- The available space (`rwnd`) is the total buffer size (`RcvBuffer`) minus the amount of data currently buffered.
- The sender limits the amount of unacknowledged, "in-flight" data to the last `rwnd` value it received from the receiver.
- This mechanism guarantees that the receiver's buffer will not overflow.

Slide 100: TCP Flow Control (Header Field)

This slide re-emphasizes that the **receive window** field in the TCP segment header is the specific mechanism used to implement flow control. It's the way the receiver communicates its available buffer space to the sender.

Slide 101: TCP Connection Management

This slide introduces the concept of connection management in TCP. Before any data can be exchanged, the sender and receiver must perform a "handshake" to:

- Agree to establish a connection, with each side confirming the other is willing and ready.
- Agree on connection parameters, most importantly the initial sequence numbers for both directions of data flow.
- Exchange other information, like buffer sizes (`rcvBuffer`).

This process establishes the necessary state on both the client and server.

Slide 102: Agreeing to Establish a Connection (2-Way Handshake)

This slide questions whether a simple two-way handshake ("Let's talk" -> "OK") is sufficient for establishing a reliable connection in a real network. It suggests that network issues like variable delays, retransmissions, and message reordering make a two-way handshake unreliable.

Slide 103: 2-Way Handshake Scenarios (No Problem)

This slide shows the ideal case where a two-way handshake works perfectly. The client sends a connection request, the server accepts, and data exchange proceeds without issue.

Slide 104: 2-Way Handshake Scenarios (Half-Open Connection)

This slide illustrates a failure case for the two-way handshake.

1. A client sends a connection request (`req_conn(x)`).

2. The server receives it and allocates resources for the connection. It sends back an acceptance (`acc_conn(x)`).
3. However, a delayed, duplicate connection request from the client arrives at the server.
4. The server, not knowing it's a duplicate, thinks it's a new request and sends another acceptance.
5. Meanwhile, the original client may have terminated or moved on. The server is now left with a "half-open" connection, wasting resources on a client that no longer exists.

Slide 105: 2-Way Handshake Scenarios (Duplicate Data)

This slide illustrates another failure case for the two-way handshake.

1. A client establishes a connection and sends data (`data(x+1)`).
2. The connection is closed, and the server forgets about it.
3. Later, a delayed, duplicate `req_conn(x)` and a delayed, duplicate `data(x+1)` arrive at the server.
4. The server, thinking this is a new connection, accepts it and incorrectly accepts the old, duplicated data as new data.

Slide 106: TCP 3-Way Handshake

This slide explains the robust three-way handshake process that TCP uses to reliably establish a connection.

1. **SYN:** The client sends a TCP segment with the `SYN` (synchronize) flag set and a randomly chosen initial sequence number (`Seq=x`). The client enters the `SYNSENT` state.
2. **SYN-ACK:** The server receives the SYN, allocates resources, and responds with a segment that has both the `SYN` and `ACK` flags set. It includes its own random initial sequence number (`Seq=y`) and acknowledges the client's SYN by setting the acknowledgement number to `ACKnum=x+1` . The server enters the `SYN RCVD` state.
3. **ACK:** The client receives the SYN-ACK. This confirms the server is live and ready. The client sends a final segment with the `ACK` flag set, acknowledging the server's SYN by setting `ACKnum=y+1` . Both client and server then enter the `ESTABLISHED` state, and data transfer can begin.

Slide 107: A Human 3-Way Handshake Protocol

This slide provides a simple, memorable analogy for the three-way handshake from the world of rock climbing.

1. **Climber (Client):** "On belay?" (Are you ready to support me?)
2. **Belayer (Server):** "Belay on." (Yes, I'm ready and acknowledging your request.)
3. **Climber (Client):** "Climbing." (Great, I acknowledge your readiness, and I'm starting.)

Slide 108: Closing a TCP Connection

This slide briefly explains how a TCP connection is closed.

- It is a graceful process where both the client and server must independently close their side of the connection.
- To initiate closing, a host sends a TCP segment with the `FIN` (finish) flag set.
- The other host responds to the received `FIN` with an `ACK`. It can then send its own `FIN` segment to close its direction of the connection.
- The protocol is designed to handle cases where both sides try to close the connection simultaneously.

Slide 109: Chapter 3: Roadmap

This slide indicates the presentation is moving to the next topic: the Principles of Congestion Control.

Slide 110: Principles of Congestion Control

This slide defines network congestion and distinguishes it from flow control.

- **Congestion:** Informally, it means "too many sources sending too much data too fast for the network to handle."
- **Symptoms:**
  - **Long delays**, caused by packets waiting in long queues inside routers.
  - **Packet loss**, which occurs when router buffers overflow.
- **Congestion Control vs. Flow Control:**
  - **Congestion Control:** Manages the problem of too many senders overwhelming the *network*.
  - **Flow Control:** Manages the problem of one sender overwhelming one *receiver*.

Slide 111: Causes/Costs of Congestion: Scenario 1

This slide presents the simplest congestion scenario.

- **Setup:** Two senders, one router with infinite buffers, and a shared output link with capacity `R`.
- **Observation:** As the total arrival rate of data (`λ_in`) approaches the link's capacity (`R`), the queuing delay for packets at the router approaches infinity. The maximum possible throughput per connection is `R/2`.
- **Cost:** Unbounded delay is the primary cost of congestion in this ideal scenario.

Slide 112: Causes/Costs of Congestion: Scenario 2

This slide introduces a more realistic scenario where the router has **finite** buffers.

- **New Factor:** Because buffers are finite, packets can be dropped (lost) when the buffer is full. This requires the sender to retransmit lost packets.
- **Two Input Rates:**
  - `λ_in`: The original data rate from the application.
  - `λ'_in`: The total data rate offered to the network, which includes both original data and retransmissions.

Slide 113: Causes/Costs of Congestion: Scenario 2 (Perfect Knowledge)

This slide considers an idealized version where the sender has perfect knowledge and only sends a packet when there is free buffer space at the router. In this unrealistic case, no packets are lost, no retransmissions are needed, and the throughput (`λ_out`) can perfectly match the offered load (`λ_in`) up to the maximum capacity of R/2 per flow.

Slide 114: Causes/Costs of Congestion: Scenario 2 (Realistic Loss)

This slide moves to a more realistic case where the sender only retransmits a packet after it knows for sure it was lost. Even in this case, a cost emerges. As the offered load increases, some of the link's capacity is used to carry retransmitted packets, which is work the network has to do a second time.

Slide 115: Causes/Costs of Congestion: Scenario 2 (Wasted Capacity)

This slide quantifies the cost from the previous slide. To achieve a certain useful throughput (`λ_out`), the total traffic on the link (`λ'_in`) must be higher because of the necessary retransmissions. This means some of the link's capacity is "wasted" on retransmitting lost data, reducing the maximum achievable *useful* throughput.

Slide 116: Causes/Costs of Congestion: Scenario 2 (Unneeded Duplicates)

This slide introduces another layer of realism and another cost. Senders don't have perfect knowledge of loss; they rely on timeouts. A sender might time out prematurely and retransmit a packet that was not actually lost, just delayed. Now, the network must carry both the original and the unneeded duplicate packet.

Slide 117: Causes/Costs of Congestion: Scenario 2 (Summary of Costs)

This slide summarizes the costs of congestion identified in this scenario:

1. **More Work:** The network has to perform more work (retransmissions) to achieve a given level of useful throughput.
2. **Unneeded Retransmissions:** Link capacity is wasted carrying duplicate copies of packets, which further decreases the maximum achievable throughput.

Slide 118: Causes/Costs of Congestion: Scenario 3

This slide presents a more complex, multi-hop scenario that reveals another major cost of congestion.

- **Setup:** Four senders, two routers. A "red" flow passes through both routers, while a "blue" flow only passes through the first router before exiting.
- **Observation:** As the sending rate of the red flow increases, it can completely fill up the buffer at the first router. When this happens, arriving blue packets find no buffer space and are dropped.
- **Result:** As the red flow's rate (`λ'_in`) increases, the blue flow's throughput can drop to zero, even though the blue flow doesn't even use the second, congested link.

Slide 119: Causes/Costs of Congestion: Scenario 3 (Wasted Upstream Work)

This slide explains the cost revealed in the previous scenario. When a packet is dropped at a downstream router, all the transmission capacity and buffering that was used on the upstream links to get the packet that far has been completely wasted. This is a significant source of inefficiency in a congested network.

Slide 120: Causes/Costs of Congestion: Insights

This slide provides a comprehensive summary of the negative effects (costs) of network congestion:

- Throughput can never exceed the network's capacity.
- Packet loss and the resulting retransmissions decrease effective throughput.
- Unneeded duplicate packets from premature timeouts further decrease effective throughput.
- As network capacity is approached, queuing delays increase dramatically.
- When packets are lost downstream, the upstream bandwidth used to transmit them is wasted.

Slide 121: Approaches Towards Congestion Control (End-to-End)

This slide describes the first major approach to congestion control.

- **End-to-End Congestion Control:** The network itself provides no explicit feedback to the sender about congestion. Instead, the sender **infers** congestion by observing network behavior, such as packet loss (indicated by timeouts or duplicate ACKs) and increasing delays.
- This is the approach taken by the standard versions of **TCP**.

Slide 122: Approaches Towards Congestion Control (Network-Assisted)

This slide describes the second major approach to congestion control.

- **Network-Assisted Congestion Control:** Routers in the network actively participate by providing **direct feedback** to the sender or receiver.
- This feedback can be an explicit signal indicating the level of congestion or even a command telling the sender to reduce its rate to a specific value.
- Examples include DECbit, ATM ABR, and the **Explicit Congestion Notification (ECN)** mechanism used with TCP.

Slide 123: Chapter 3: Roadmap

This slide indicates that the presentation will now focus on the specifics of **TCP Congestion Control**.

Slide 124: TCP Congestion Control: AIMD

This slide introduces the core algorithm for TCP congestion control: **AIMD (Additive Increase, Multiplicative Decrease)**.

- **Approach:** Senders cautiously increase their sending rate until they detect packet loss (which signals congestion), at which point they drastically decrease their rate.
- **Additive Increase (AI):** For every Round Trip Time (RTT) that passes without a loss event, the sender increases its sending rate by 1 Maximum Segment Size (MSS). This is a slow, linear increase.
- **Multiplicative Decrease (MD):** When a loss event is detected, the sender cuts its sending rate in half.
- This behavior creates a characteristic "sawtooth" pattern as TCP probes for available bandwidth.

Slide 125: TCP AIMD: More Details

This slide provides more specifics on the AIMD algorithm and its rationale.

- **Multiplicative Decrease in Practice:**
  - For a loss detected by **triple duplicate ACKs** (implying mild congestion), the rate is cut in half (this is TCP Reno behavior).
  - For a loss detected by a **timeout** (implying more severe congestion), the rate is cut drastically to just 1 MSS (this is TCP Tahoe behavior).
- **Why AIMD?** This distributed algorithm, where each sender acts independently, has been proven to be stable and to converge towards a fair and efficient allocation of network resources among competing flows.

Slide 126: TCP Congestion Control: Details

This slide explains how TCP's sending rate is controlled by the **congestion window (`cwnd`)**.

- **Sending Limit:** The TCP sender ensures that the amount of unacknowledged data in flight (`LastByteSent - LastByteAcked`) is always less than the `cwnd`.
- **Dynamic Adjustment:** The `cwnd` is not fixed; it is dynamically adjusted based on perceived network congestion, implementing the AIMD algorithm.
- **Effective Rate:** The sending rate of a TCP connection is roughly `cwnd / RTT` (bytes/sec). The sender sends `cwnd` bytes of data and then waits about one RTT to get acknowledgements before it can send more.

Slide 127: TCP Slow Start

This slide explains the **Slow Start** phase of a TCP connection.

- **Goal:** To quickly ramp up the sending rate at the beginning of a connection to find the available bandwidth.
- **Mechanism:**
  - The `cwnd` starts at a small value, typically 1 MSS.
  - For every ACK received, the `cwnd` is incremented by 1 MSS. This results in the `cwnd` **doubling every RTT**, leading to exponential growth.
- This "slow start" is only slow initially; the rate increases very rapidly. The exponential growth continues until the first loss event occurs.

Slide 128: TCP: From Slow Start to Congestion Avoidance

This slide explains how TCP transitions from the aggressive exponential growth of Slow Start to the more cautious linear growth of Congestion Avoidance (AIMD).

- **The Trigger:** The switch occurs when the `cwnd` reaches a value called the **slow start threshold (`ssthresh`)**.
- **Setting `ssthresh`:** When a loss event occurs, `ssthresh` is set to half of the `cwnd` value at the time of the loss.
- **The Logic:**
  - If `cwnd` < `ssthresh`, TCP is in Slow Start (exponential growth).
  - If `cwnd` > `ssthresh`, TCP is in Congestion Avoidance (linear growth). This prevents TCP from repeatedly and aggressively overshooting the network's capacity.

Slide 129: Summary: TCP Congestion Control (FSM)

This is a complex Finite State Machine diagram summarizing the different states of TCP congestion control (specifically, TCP Reno).

- **Initial State:** Begins in **Slow Start** (`cwnd` =1 MSS, `ssthresh` =64 KB). `cwnd` grows exponentially (doubles each RTT).
- **Transition to Congestion Avoidance:** When `cwnd` exceeds `ssthresh`, the state changes to **Congestion Avoidance**. `cwnd` now grows linearly (by 1 MSS per RTT).
- **Loss Event (Timeout):**
  - `ssthresh` is set to `cwnd/2`.
  - `cwnd` is reset to 1 MSS.
  - The state returns to **Slow Start**.
- **Loss Event (3 Duplicate ACKs):**
  - `ssthresh` is set to `cwnd/2`.
  - `cwnd` is set to `ssthresh + 3` (to account for the packets that triggered the duplicate ACKs).
  - This is known as **Fast Recovery**. The sender retransmits the missing segment and stays in a state of congestion avoidance, rather than dropping back to slow start.

Slide 130: TCP CUBIC

This slide introduces **TCP CUBIC**, a more modern congestion control algorithm that is now the default in many operating systems like Linux.

- **Insight:** After a packet loss, the network's state probably hasn't changed dramatically. Instead of slowly probing back up to the previous maximum sending rate (`W_max`) like AIMD, we can be more strategic.
- **CUBIC's Approach:**
  - After a loss, the sending rate is cut (similar to AIMD).
  - It then rapidly increases its rate to get close to the old `W_max`.
  - As it gets very near `W_max`, it slows down its growth, approaching the limit more cautiously to avoid causing another loss.
- The graph shows that CUBIC can achieve higher average throughput than classic TCP by spending more time at or near the bottleneck link's capacity.

Slide 131: TCP CUBIC (Details)

This slide provides more detail on how CUBIC works.

- **Cubic Function:** CUBIC's window growth is governed by a cubic function of the time elapsed since the last congestion event. The function is centered on the point in time (`K`) when the window size would reach the last known maximum (`W_max`).

- **Behavior:** The shape of the cubic function means that the window grows slowly when it's very close to `W_max` (the "plateau" of the curve) but grows very quickly when it's far away from it. This combines rapid recovery with stability near the congestion point.

Slide 132: TCP and the Congested "Bottleneck Link"

This slide explains that in any end-to-end path, there is typically one link that is the slowest or most congested, known as the **bottleneck link**. TCP's congestion control mechanism effectively discovers and adapts to the capacity of this bottleneck. When the bottleneck link's queue is full, packets are dropped, signaling congestion to the TCP sender.

Slide 133: TCP and the Congested "Bottleneck Link" (Insights)

This slide presents key insights about the bottleneck link.

- **Throughput Insight:** Simply increasing the TCP sending rate beyond the bottleneck's capacity will **not** increase the end-to-end throughput. The data will just get stuck in the queue.
- **Delay Insight:** Increasing the TCP sending rate **will** increase the measured RTT, because packets spend more time waiting in the bottleneck's queue.
- **Goal:** The ideal state is to "keep the end-to-end pipe just full, but not fuller." This means keeping the bottleneck link busy but avoiding long queues and high delays.

Slide 134: Delay-based TCP Congestion Control

This slide introduces an alternative to loss-based congestion control algorithms like AIMD and CUBIC.

- **Approach:** Instead of using packet loss as the primary signal for congestion, **delay-based TCP** uses increasing RTT as the signal.
- **Mechanism:**
    1. It measures the minimum RTT ever observed (`RTT_min`), which represents the uncongested path delay.
    2. It calculates the current measured throughput.
    3. It compares this measured throughput to the theoretical uncongested throughput (`cwnd / RTT_min`).
    4. If the measured throughput is close to the theoretical max, the path is likely not congested, so `cwnd` is increased.
    5. If the measured throughput is far below the theoretical max, the path is likely congested (due to queuing delay), so `cwnd` is decreased.

Slide 135: Delay-based TCP Congestion Control (Benefits)

This slide summarizes the advantages of delay-based TCP.

- It can control congestion **without needing to cause packet loss**, leading to lower latency and smoother performance.
- It aims to maximize throughput while simultaneously keeping delay low, directly addressing the goal of "keeping the pipe just full, but not fuller."
- **BBR (Bottleneck Bandwidth and Round-trip propagation time)** is a prominent example of a delay-based TCP, deployed on Google's internal network.

Slide 136: Explicit Congestion Notification (ECN)

This slide describes **ECN**, a network-assisted congestion control mechanism.

- **Mechanism:** It uses two bits in the IP header. A router experiencing the onset of congestion can **mark** a packet with an ECN signal instead of dropping it.
- **Feedback Loop:**
  1. A router marks an IP datagram (e.g., ECN=11).
  2. The destination host receives the marked packet.
  3. The destination sets the **ECE (ECN-Echo)** bit in its next TCP ACK segment back to the sender.
  4. The sender receives the ACK with the ECE bit set and reacts as if it had detected a mild congestion event (e.g., by reducing its `cwnd`), but without any packet being lost.

Slide 137: TCP Fairness

This slide introduces the concept of fairness in the context of TCP.

- **Fairness Goal:** If `K` different TCP sessions are sharing a single bottleneck link with a total bandwidth of `R`, each session should ideally receive an average data rate of `R/K`.
- The diagram shows two TCP connections competing for the capacity of a single bottleneck router.

Slide 138: Is TCP Fair?

This slide explores whether TCP's AIMD algorithm achieves fairness.

- **Idealized Behavior:** The graph shows the throughput of two competing connections. When both are increasing their rates additively, they move towards a state of equal bandwidth share. When loss occurs, both decrease their rates multiplicatively (by the same factor), which also pushes them towards an equal share.
- **Conclusion:** Yes, under idealized assumptions (all connections have the same RTT and are always in congestion avoidance mode), TCP's AIMD algorithm is proven to converge to a fair allocation of bandwidth.

Slide 139: Fairness: Must All Network Apps Be "Fair"?

This slide raises important questions about the practical limits of TCP fairness.

- **Fairness and UDP:** Multimedia applications often use UDP to avoid TCP's congestion control, which can throttle their sending rate. A high-rate UDP stream does not "play fair" and can starve competing TCP flows of bandwidth. There is no "Internet police" to enforce fairness.
- **Fairness and Parallel TCP Connections:** An application can gain an unfair advantage by opening multiple parallel TCP connections between two hosts. For example, if 9 applications are each using one connection, and a new application opens 11 connections, the new application will get 11/20ths (more than half) of the bandwidth, not a fair 1/10th share. Web browsers commonly use this technique to speed up page loads.

Slide 140: Transport Layer: Roadmap

This slide indicates the presentation is moving to its final topic: the Evolution of Transport-Layer Functionality.

Slide 141: Evolving Transport-Layer Functionality

This slide discusses how the transport layer has evolved.

- For decades, TCP and UDP have been the primary transport protocols.
- Different "flavors" or versions of TCP have been developed to address specific challenges, such as:
  - **Wireless networks:** Where loss is not always due to congestion.
  - **Long, fat pipes:** High-speed, long-delay links that need very large windows.
  - **Data centers:** Where low latency is critical.
- A major trend is moving transport-layer functions (like reliability and congestion control) into the **application layer**, running them on top of the simple UDP protocol. HTTP/3 is a prime example.

Slide 142: QUIC: Quick UDP Internet Connections

This slide introduces **QUIC**, the protocol that powers HTTP/3.

- **What it is:** QUIC is an application-layer protocol that runs on top of UDP. It was developed by Google to increase the performance of HTTP.
- **Protocol Stack:**
  - **Old:** Application (HTTP/2) -> Transport (TCP) -> Security (TLS) -> Network (IP)
  - **New:** Application (HTTP/2 concepts) -> Transport/Security (QUIC) -> Transport (UDP) -> Network (IP) QUIC essentially redesigns and combines the roles of TCP and TLS and runs them over UDP.

Slide 143: QUIC: Quick UDP Internet Connections (Features)

This slide details the features of QUIC.

- It implements its own versions of connection establishment, error control, and congestion control, drawing heavily on the principles proven in TCP.
- **Streams:** It allows multiple application-level "streams" to be multiplexed over a single QUIC connection. Each stream has its own independent reliable data transfer, but they share a common congestion controller.
- **Fast Connection Establishment:** It combines the transport handshake (like TCP's) and the security handshake (like TLS's) into a single round trip, significantly reducing connection setup time.

Slide 144: QUIC: Connection Establishment

This slide visually compares the connection establishment process for TCP+TLS versus QUIC.

- **TCP + TLS:** This requires two separate, sequential handshakes. First, the TCP 3-way handshake establishes the transport connection. Then, the TLS handshake establishes the secure session. This can take 2 or more round trips before application data can be sent.
- **QUIC:** The QUIC handshake is designed to establish all necessary state—for reliability, congestion control, authentication, and encryption—in a single handshake, typically requiring just **one round trip**.

Slide 145: QUIC: Streams: Parallelism, No HOL Blocking

This slide illustrates one of the most significant advantages of QUIC over TCP for web traffic: the elimination of **Head-of-Line (HOL) blocking**.

- **(a) HTTP/2 over TCP:** When multiple HTTP requests are multiplexed over a single TCP connection, if one TCP segment is lost, the TCP protocol must wait for it to be retransmitted before it can deliver *any* of the subsequent data to the application, even if that data belongs to a different, unrelated HTTP request. This stalls all streams.
- **(b) HTTP/3 over QUIC:** QUIC streams are independent at the transport level. If a packet carrying data for one stream is lost, it only stalls that specific stream. Other streams can continue to be processed and delivered to the application without waiting.

Slide 146: Chapter 3: Summary

This slide concludes the chapter by summarizing the key takeaways.

- The presentation covered the core **principles** of transport layer services: multiplexing/demultiplexing, reliable data transfer, flow control, and congestion control.
- It also covered the **implementation** of these principles in the Internet's main protocols, UDP and TCP.
- **Up Next:** The course will move from the "edge" of the network (application and transport layers) into the "core," focusing on the network layer, which is split into two parts: the data plane and the control plane.

Slide 147: Additional Chapter 3 Slides

This slide indicates that the following slides contain supplementary material not included in the main presentation flow.

Slide 148: Go-Back-N: Sender Extended FSM

This slide provides a more detailed, code-like Finite State Machine (FSM) for the Go-Back-N (GBN) sender. It specifies the variables used (`base` , `nextseqnum`) and the logic for handling data from the application, timeouts, and received ACKs in a more formal way than the earlier, simplified diagrams.

Slide 149: Go-Back-N: Receiver Extended FSM

This slide presents the detailed FSM for the GBN receiver. It shows that the receiver maintains only one state variable, `expectedseqnum` . If a packet with the correct sequence number arrives, the data is delivered, an ACK is sent, and `expectedseqnum` is incremented. For any other event (corrupt packet, out-of-order packet), it discards the packet and re-sends an ACK for the last successfully received in-order packet. This highlights the "no receiver buffering" characteristic of GBN.

Slide 150: TCP Sender (Simplified)

This slide presents a more formal, event-driven pseudocode for the TCP sender's logic, building on the simplified description from slide 89. It details the management of the `NextSeqNum` and `SendBase` state variables in response to the three main events: data received from the application, a timer timeout, and the receipt of an ACK.

Slide 151: TCP 3-Way Handshake FSM

This slide shows a more detailed Finite State Machine diagram for the TCP 3-way handshake. It illustrates the states on both the client and server sides (`LISTEN` , `SYN_SENT` , `SYN_RCVD` , `ESTAB`) and the specific SYN and ACK packets that trigger transitions between these states. It also links these transitions to the socket API calls like `connect()` and `accept()` .

Slide 152: Closing a TCP Connection (FSM)

This slide presents the FSM for closing a TCP connection, which is more complex than the opening handshake. It shows the various states a connection can pass through during teardown, such as `FIN_WAIT_1`, `FIN_WAIT_2`, `CLOSE_WAIT`, `LAST_ACK`, and `TIMED_WAIT`. The `TIMED_WAIT` state is particularly important, as it ensures any lingering duplicate packets from the connection have expired before the connection state is fully removed.

Slide 153: TCP Throughput (vs. W and RTT)

This slide provides a simplified formula for calculating the average throughput of a TCP connection as a function of its window size and RTT, assuming it's in congestion avoidance mode.

- When a loss occurs, the window size `W` is cut in half to `W/2`. It then additively increases back up to `W` over the course of one "cycle".
- The average window size during this cycle is approximately `¾ * W`.
- Therefore, the average throughput is `(¾ * W) / RTT` bytes/sec.

Slide 154: TCP over "Long, Fat Pipes"

This slide discusses the challenges of using TCP over networks with both high bandwidth and high latency (so-called "long, fat pipes").

- **Example:** To achieve 10 Gbps throughput with a 100ms RTT, the required congestion window (`W`) would need to hold 83,333 in-flight segments.
- **Throughput vs. Loss:** It presents a formula from a 1997 paper by Mathis et al. that relates TCP throughput to the packet loss rate (`L`).
- **Implication:** To achieve 10 Gbps on this link, the loss rate would have to be incredibly small (2 x 10^{-10}). This shows that standard TCP is not well-suited for these environments and that specialized versions of TCP are needed.