Chapter 1

L14 — Transport Layer and TCP

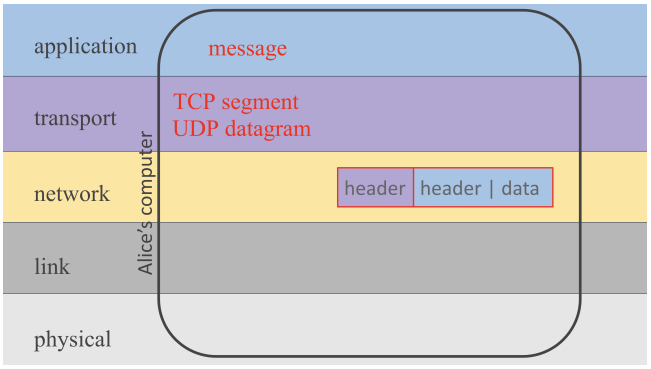
1.1 Introduction to the Transport Layer

Understanding the role and responsibilities of the transport layer in network communication.
Throughout this semester, we have focused on the application layer, examining how processes interact through system calls and how applications like web clients, servers, and DNS operate. Today, we explore what happens beneath the application layer when processes make network system calls.

1.1.1 Protocol Stack and Data Structures

Understanding how data is transformed as it moves through network layers.
The Internet uses a layered architecture where each layer adds header information:

1. **Application Layer** — Creates **messages** for inter-process communication
2. **Transport Layer** — Adds transport headers, creating:
   - **UDP datagrams** (User Datagram Protocol)
   - **TCP segments** (Transmission Control Protocol)
3. **Network Layer** — Adds IP headers, creating **IP packets**



The transport layer provides essential services enabling reliable process-to-process communication over an unreliable network infrastructure.

1.2 User Datagram Protocol (UDP)

A minimal transport protocol providing basic services with low overhead.
UDP adds minimal functionality to the network layer, making it suitable for applications requiring low overhead that can tolerate data loss.

1.2.1 UDP Services and Communication

Core functions and operation of UDP.
UDP provides three fundamental services:

Multiplexing and Demultiplexing

- **Multiplexing** — Handles messages from multiple application processes, encapsulating each in UDP datagrams
- **Demultiplexing** — Uses port numbers to deliver incoming datagrams to the correct application process
- Enables multiple applications to share network resources efficiently

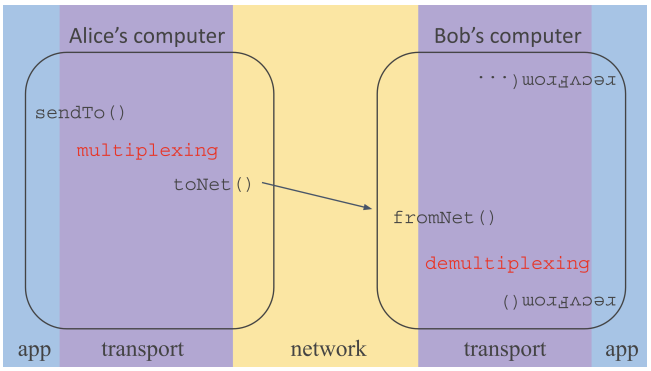
Basic Error Detection

- **Checksum computation** — Sender computes checksum from header and data
- **Integrity verification** — Receiver recomputes and compares checksums
- **Corruption handling** — Corrupted datagrams are discarded

The checksum uses one's complement arithmetic on 16-bit words, providing simple but effective error detection for most transmission errors.

1.2.2 UDP Communication Process

Step-by-step UDP communication between client and server.



1. Both processes create UDP sockets: `socket()`
2. Server binds to specific address: `bind()`
3. Server prepares to receive: `recvfrom()`
4. Client sends message: `sendto()`
5. Transport layers handle UDP datagram transmission
6. Server receives message and `recvfrom()` returns

1.2.3 UDP Header Structure and Limitations

Understanding UDP's simple structure and constraints.

UDP Header (8 bytes total)

- **Source Port (16 bits)** — Sending process identifier
- **Destination Port (16 bits)** — Receiving process identifier
- **Length (16 bits)** — Total datagram length
- **Checksum (16 bits)** — Error detection value

UDP Capabilities and Limitations

- **Provides:** Process identification, basic error detection, low overhead
- **Lacks:** Reliability guarantees, ordering, flow control, congestion control, error correction

The network layer operates on a **best-effort** basis, potentially dropping, corrupting, or reordering packets. UDP provides a minimal abstraction over this unreliable foundation.

1.3 Transmission Control Protocol (TCP)

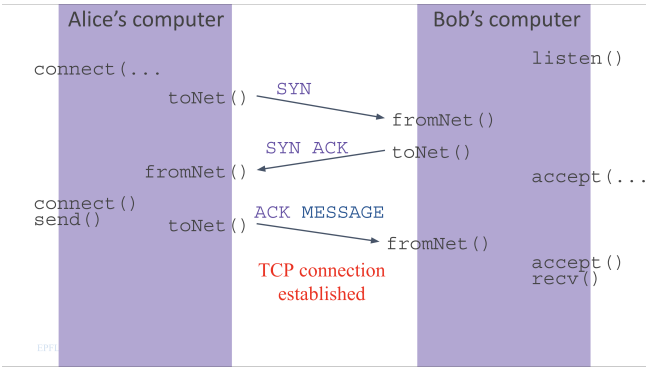
A comprehensive transport protocol providing reliable, ordered data delivery.
TCP builds upon UDP's basic services while adding connection management, reliability, flow control, and congestion control.

1.3.1 TCP Connection Management

How TCP establishes and maintains reliable communication channels.

Connection Establishment: Three-Way Handshake

Unlike UDP's connectionless approach, TCP requires establishing a logical connection before data exchange. This ensures both parties are ready to communicate and agree on communication parameters.



Handshake Concept. The connection establishment works like a conversation:

1. **Client Request:** "I want to establish a connection with you"
2. **Server Response:** "I accept your request, and I also want to establish a connection with you"
3. **Client Confirmation:** "I acknowledge your acceptance, connection established"

Technical Implementation. TCP implements this using control flags in segment headers:

1. **Server Setup:** `socket()`, `bind()`, `listen()`
2. **Client Initiation:** `socket()`, `connect()` → sends **SYN** (synchronize) segment
3. **Server Response:** Sends **SYN-ACK** (synchronize-acknowledge) segment
4. **Client Confirmation:** Sends **ACK** (acknowledge) segment (may include data)
5. **Server Acceptance:** `accept()` creates connection socket

Resource Allocation. During handshake, both sides allocate send/receive buffers and establish connection state (sequence numbers, window sizes).

1.3.2 TCP Socket Types and Multiplexing

Understanding TCP's sophisticated connection management.

Socket Type Distinction

- **Listening Socket** — Server uses for accepting new connections (`socket()`, `bind()`, `listen()`)
- **Connection Socket** — Dedicated to specific client-server communication, created by `accept()`
- **Connection Identification** — Four-tuple: (source IP, source port, destination IP, destination port)

Key Differences from UDP

- Each TCP connection socket communicates with exactly one remote process
- Servers can handle multiple simultaneous connections using separate connection sockets
- Client uses `connect()` to establish connection to specific server process
- Server uses `accept()` to create dedicated socket for each client connection

1.3.3 TCP Reliability Mechanisms

How TCP ensures reliable data delivery over unreliable networks.
TCP implements comprehensive reliability mechanisms to overcome network limitations including corruption, loss, and reordering.

Error Detection and Acknowledgment

Basic Operation:

1. Sender transmits TCP segment with data and sequence number
2. Receiver verifies integrity using checksum
3. If uncorrupted: receiver sends ACK and delivers data to application
4. If corrupted or lost: sender detects via timeout and retransmits

Sequence Numbers and Acknowledgments

Sequence Numbers:

- Identify data bytes: sequence number indicates the first data byte in segment
- Enable duplicate detection: retransmitted segments have same sequence number
- Support ordered delivery: receiver can reorder segments if needed

Cumulative Acknowledgments:

- ACK n means "I have received all bytes up to and including byte $n-1$, expecting byte n"
- No explicit negative ACKs: repeated ACKs serve as implicit NACKs
- Simplifies protocol: single ACK field confirms multiple segments

Example. If sender transmits SEQ 1 then SEQ 2, and receiver responds with ACK 2 twice, the second ACK 2 implicitly signals that SEQ 2 was not received correctly.

Timeout and Retransmission

Timeout Mechanism:

- Sender starts timer for each transmitted segment
- If ACK not received before timeout, assume segment lost and retransmit
- Handles both segment loss and ACK loss scenarios
- May cause unnecessary retransmissions due to delayed ACKs

Timeout Calculation:

$$\text{EstimatedRTT} = 0.875 \times \text{EstimatedRTT} + 0.125 \times \text{SampleRTT} \tag{1.1}$$

$$\text{Timeout} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \tag{1.2}$$

Where DevRTT measures RTT variance. Conservative estimation prevents premature timeouts while adapting to network conditions.

1.3.4 TCP Header Structure

Understanding the complexity required for reliable communication.

Key TCP Header Fields (minimum 20 bytes)

- **Source/Destination Ports** — Process identification
- **Sequence Number** — First data byte number in this segment
- **Acknowledgment Number** — Next expected byte number
- **Control Flags** — SYN, ACK, FIN, RST for connection management
- **Window Size** — Flow control information
- **Checksum** — Error detection

Control Flags

- **SYN** — Connection establishment (handshake), 1-bit field
- **ACK** — Acknowledgment field is valid
- **FIN** — Connection termination
- **RST** — Immediate connection reset

1.4 Reliability Mechanisms Summary

Comprehensive overview of transport layer reliability techniques.

1.4.1 Basic Reliability Components

TCP achieves reliable data delivery through the combination of three fundamental mechanisms:

1. **Checksums** — Detect corruption in transmitted data
2. **Sequence Numbers + Acknowledgments + Retransmissions** — Overcome corruption through confirmed delivery
3. **Timeouts + Sequence Numbers + Acknowledgments + Retransmissions** — Detect and overcome loss

1.4.2 Protocol Comparison

Understanding when to use UDP vs TCP.

Protocol Trade-offs

- **UDP:** Low overhead (8-byte header), fast, simple; no reliability guarantees
- **TCP:** Higher overhead ($\geq$20-byte header), complex; provides reliability, ordering, flow control

Application Suitability

- **UDP Applications:** Real-time communication (VoIP, gaming), DNS queries, streaming media
- **TCP Applications:** File transfer (HTTP, FTP), email, remote login (SSH), e-commerce

Design Principle. The transport layer demonstrates a fundamental trade-off in protocol design: simplicity and efficiency versus feature richness and guarantees. Both approaches serve different application requirements in the Internet ecosystem.

1.5 TCP Bidirectional Communication

Understanding sequence numbers and acknowledgments in realistic scenarios.
In practice, TCP connections carry bidirectional communication where both client and server processes exchange data simultaneously. This requires careful coordination of sequence numbers and acknowledgments.

1.5.1 Bidirectional Data Exchange

How TCP handles simultaneous data transmission in both directions.

Basic Bidirectional Example

Consider a simple exchange where both client and server send single-byte messages:

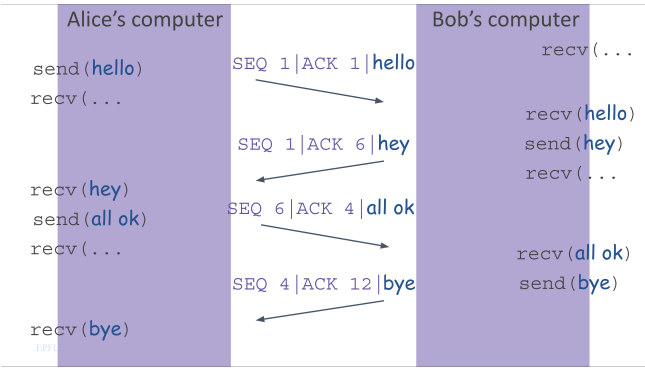
- **Client sends "A":** SEQ 1, ACK 1 (first byte to server, expecting server's first byte)
- **Server sends "B":** SEQ 1, ACK 2 (first byte to client, received client's byte 1, expecting byte 2)

Key Insight. Each TCP segment contains both data (if any) and acknowledgment information, enabling efficient bidirectional communication without separate ACK-only segments.

Multi-Byte Message Example

Realistic example with variable-length messages.
Consider an application-layer exchange:

- Client sends: "hello" (5 bytes)
- Server responds: "hey" (3 bytes)
- Client responds: "all ok" (6 bytes)
- Server responds: "bye" (3 bytes)



Transport Layer Implementation:

1. **Client → Server:** "hello" with SEQ 1, ACK 1
2. **Server → Client:** "hey" with SEQ 1, ACK 6 (received bytes 1-5, expecting byte 6)
3. **Client → Server:** "all ok" with SEQ 6, ACK 4 (received bytes 1-3, expecting byte 4)
4. **Server → Client:** "bye" with SEQ 4, ACK 12 (received bytes 6-11, expecting byte 12)

1.5.2 Real-World HTTP Example

Analyzing TCP behavior in web communication.

HTTP Request-Response Pattern

Web communication demonstrates practical TCP usage with asymmetric data flows:

- **Client:** HTTP GET request (200 bytes)
- **Server:** HTTP response with data (3,100 bytes)

Segmentation and Acknowledgment Pattern

Client Request:

- Single segment: SEQ 1, ACK 1 (200-byte GET request)

Server Response (segmented):

- Segment 1: SEQ 1, ACK 201 (bytes 1-1500)
- Segment 2: SEQ 1501, ACK 201 (bytes 1501-3000)
- Segment 3: SEQ 3001, ACK 201 (bytes 3001-3100)

Client Acknowledgments:

- ACK 1501: Confirming receipt of first 1500 server bytes
- ACK 3001: Confirming receipt of bytes 1-3000
- ACK 3101: Confirming receipt of entire response

Important Observations

- **Persistent SEQ:** Client ACKs use SEQ 201 (no new data to send)
- **Cumulative ACKs:** Each ACK confirms all bytes received so far
- **Data Segmentation:** Large messages split across multiple segments for efficient transmission
- **Piggybacked ACKs:** Acknowledgments combined with data when possible

Segmentation Rationale. Transport layers segment large messages to optimize network utilization and enable efficient error recovery at the segment level rather than requiring retransmission of entire large messages.

1.6 TCP Flow Control and Congestion Control

Managing sender transmission rates to prevent receiver overload and network congestion.
TCP must control the rate at which data is transmitted to prevent overwhelming both the receiver and the network infrastructure. This requires two complementary mechanisms operating simultaneously.

1.6.1 Maximum Segment Size and Segmentation

Understanding how TCP determines segment boundaries.

Maximum Segment Size (MSS)

The Maximum Segment Size represents the maximum amount of application-layer data that a single TCP segment may carry:

- **Determination:** MSS is dictated by network properties, particularly the bit error rate of links between sender and receiver
- **Discovery:** Transport layer discovers MSS through network interface properties and path MTU discovery
- **Typical Value:** 1500 bytes in current Internet infrastructure
- **Impact:** Large messages require segmentation across multiple TCP segments

Practical Segmentation Example

Consider an HTTP response scenario:

- Client sends 200-byte GET request → Single segment
- Server responds with 3100-byte data → Three segments (1500 + 1500 + 100 bytes)
- Each segment acknowledged independently for reliable delivery

1.6.2 Sender Window Management

Controlling the maximum number of unacknowledged bytes in transmission.

Sender Window Concept

The **sender window** indicates the maximum number of unacknowledged bytes that the sender may transmit:

- **Purpose:** Prevents overwhelming receiver and network
- **Dynamic Adjustment:** Changes based on current network and receiver conditions
- **Computation:** Minimum of flow control window and congestion control window

Dual Control Mechanisms

TCP sender window management combines two independent mechanisms:

1. **Flow Control** — Prevents overwhelming the receiver
   - Receiver explicitly signals maximum acceptable unacknowledged bytes
   - Communicated via receiver window field in TCP header
   - Direct feedback mechanism
2. **Congestion Control** — Prevents overwhelming the network
   - Sender estimates network capacity independently
   - No explicit network feedback available
   - Inferred through packet loss and acknowledgment patterns

Window Computation.
At each moment: Sender Window = $\min(\text{Flow Control Window}, \text{Congestion Window})$

1.7 TCP Congestion Control Algorithms

Adaptive algorithms for inferring and responding to network congestion.
TCP congestion control operates on the principle of **self-clocking**, where the sender adjusts its transmission rate based on acknowledgment patterns without explicit network feedback.

1.7.1 Key Congestion Control Concepts

Essential terminology and variables for understanding TCP algorithms.

Slow Start Threshold (ssthresh)

The **slow start threshold** is a critical variable that acts as TCP's "memory" of previous network congestion:

- **Purpose:** Remembers the congestion window size when congestion was last detected
- **Initial Value:** Set to a large value (effectively infinite) when connection starts
- **Updated When:** Congestion occurs (timeout or duplicate acknowledgments)
- **Calculation:** Always set to half the current congestion window when congestion detected
- **Usage:** Determines when to switch from exponential growth to linear growth

Why Half the Window? When congestion occurs, TCP assumes the network can handle approximately half of what was being sent, providing a conservative estimate for future transmissions.

Congestion Window (cwnd)

The **congestion window** represents the sender's estimate of how much data the network can handle:

- **Purpose:** Controls the maximum unacknowledged data the sender may transmit
- **Initial Value:** 1 Maximum Segment Size (conservative start)
- **Dynamic Adjustment:** Increases when network performs well, decreases when congestion detected
- **Units:** Measured in bytes

Maximum Segment Size

The **Maximum Segment Size** is the largest amount of application data that fits in one TCP segment:

- **Typical Value:** 1500 bytes in most modern networks
- **Determined By:** Network path properties and interface capabilities
- **Usage:** Unit of measurement for window adjustments

1.7.2 Self-Clocking Principle

How TCP infers network conditions from acknowledgment behavior.

Basic Self-Clocking Logic

The sender makes decisions based on acknowledgment patterns, following this simple logic:

- **New acknowledgment received** → Network conditions good → Increase congestion window
- **No new acknowledgment (timeout)** → Network conditions bad → Decrease congestion window
- **Adaptive Behavior** → Continuously adjusts to changing network conditions

Step-by-Step Decision Process:

1. Send data segments up to the current congestion window limit
2. Wait for acknowledgments from the receiver
3. **If acknowledgments arrive promptly:** Network can handle current load → increase window size
4. **If acknowledgments are missing or delayed:** Network may be congested → decrease window size
5. Repeat this process for every round of transmission

Algorithm Variations

Multiple congestion control algorithms implement this principle with different strategies:

- **Historical Algorithms:** Tahoe and Reno (foundational concepts we will study)
- **Modern Algorithms:** Cubic, New Reno (currently deployed in practice)
- **Key Design Questions:** How aggressively to react to new acknowledgments vs. missing acknowledgments

1.7.3 TCP Tahoe Algorithm

Foundational congestion control algorithm demonstrating core principles.
The Tahoe algorithm operates in two distinct states, each with different strategies for growing the congestion window based on network feedback.

Tahoe Algorithm States

Slow Start State. Aggressive window growth for initial connection:

- **Initial Window:** 1 Maximum Segment Size (conservative start)
- **Growth Rule:** Increase window by 1 Maximum Segment Size for each new acknowledgment received
- **Growth Pattern:** Window doubles every round-trip time (exponential growth)
- **Transition Condition:** Switch to congestion avoidance when window reaches slow start threshold

Slow Start Step-by-Step Process:

1. Start with congestion window = 1 Maximum Segment Size
2. Send data segments up to the current window limit
3. For each acknowledgment received, increase window by 1 Maximum Segment Size
4. Send more data with the larger window
5. Continue until window size equals slow start threshold

Congestion Avoidance State. Cautious window growth near suspected limits:

- **Growth Rule:** Increase window by $\frac{\text{Maximum Segment Size}^2}{\text{current window size}}$ for each new acknowledgment
- **Growth Pattern:** Window increases by approximately 1 Maximum Segment Size per round-trip time (linear growth)
- **Purpose:** Probe for additional network capacity carefully without causing congestion

Congestion Avoidance Step-by-Step Process:

1. Current window size has reached slow start threshold
2. Send data segments up to the current window limit
3. For each acknowledgment received, increase window by $\frac{\text{Maximum Segment Size}^2}{\text{current window size}}$ bytes
4. This small increase results in approximately 1 Maximum Segment Size growth per round-trip time
5. Continue until congestion is detected (timeout occurs)

Tahoe Transition Events

Timeout Event. Response to suspected severe congestion:
When the sender waits too long for an acknowledgment (timeout occurs), it assumes severe network congestion and responds conservatively:

1. **Update slow start threshold:** Set slow start threshold = $\frac{\text{current window size}}{2}$ (remember where congestion occurred)
2. **Reset congestion window:** Set congestion window back to 1 Maximum Segment Size (start over conservatively)
3. **Retransmit lost data:** Send the oldest unacknowledged segment again
4. **Return to slow start state:** Begin exponential growth again, but more cautiously

Why These Steps?

- **Halving the threshold:** Assumes network can handle about half of what caused congestion
- **Resetting to 1:** Conservative restart ensures we don't immediately cause more congestion
- **Retransmission:** Ensures data reliability despite network problems
- **Slow start restart:** Allows gradual ramp-up to test current network conditions

Window Reaches Slow Start Threshold. Transition to careful growth:
When the congestion window grows to equal the slow start threshold during slow start:

1. **State change:** Switch from slow start to congestion avoidance
2. **Window maintenance:** Keep current window size unchanged
3. **Growth pattern change:** Begin linear growth instead of exponential growth
4. **Rationale:** We're approaching the size that previously caused congestion, so be more careful

1.7.4 Detailed Tahoe Example

Step-by-step illustration of Tahoe algorithm behavior.

Scenario Setup

Consider Alice establishing a TCP connection to Bob with Maximum Segment Size = 100 bytes:

- **Connection Established:** Three-way handshake completed successfully
- **Initial State:** Alice starts in slow start state
- **Initial Congestion Window:** 1 Maximum Segment Size = 100 bytes
- **Initial Slow Start Threshold:** Undefined (will be set after first congestion event)

What Alice Will Do: Alice will start sending data conservatively, then gradually increase her sending rate based on network feedback. Let's trace through exactly what happens step by step.

Phase 1: Slow Start Growth

Round 1 — Conservative Beginning:

- **Alice sends:** Sequence 1 (bytes 1-100), current window = 100 bytes
- **Bob responds:** Acknowledgment 101 (confirming receipt of bytes 1-100)
- **Window update:** Alice increases window: 100 + 100 = 200 bytes (doubled)
- **Explanation:** Alice received 1 acknowledgment, so she adds 1 Maximum Segment Size to her window

Round 2 — Exponential Growth:

- **Alice sends:** Sequence 101 (bytes 101-200), Sequence 201 (bytes 201-300)
- **Bob responds:** Acknowledgment 201, Acknowledgment 301
- **Window update:** Alice increases window: 200 + 100 + 100 = 400 bytes (doubled again)
- **Explanation:** Alice received 2 acknowledgments, so she adds 2 Maximum Segment Sizes to her window

Round 3 — Network Limit Reached:

- **Alice sends:** Sequence 301, Sequence 401, Sequence 501, Sequence 601 (400 bytes total)
- **Network congestion:** All 4 segments lost due to network overload
- **Timeout occurs:** Alice detects no acknowledgments received within timeout period
- **Alice's conclusion:** The network cannot handle 400 bytes sent simultaneously

Phase 2: Timeout Response and Recovery

Congestion Detection and Response: Alice now realizes the network is congested and responds with the Tahoe algorithm's timeout procedure:

1. **Set slow start threshold:** $\frac{400}{2} = 200$ bytes (remember where congestion occurred)
2. **Reset congestion window:** 400 $\rightarrow$ 100 bytes (back to 1 Maximum Segment Size)
3. **Return to slow start state:** Begin conservative exponential growth again
4. **Retransmit lost data:** Send Sequence 301 (oldest unacknowledged segment)

Why Alice Does This:

- **Threshold = 200 bytes:** Alice remembers that 400 bytes caused problems, so 200 bytes is probably safe
- **Window = 100 bytes:** Start conservatively to avoid immediately causing more congestion
- **Slow start state:** Use exponential growth to quickly find the right sending rate
- **Retransmit:** Ensure data reliability by resending what was lost

Recovery Transmission:

- **Alice sends:** Sequence 301, current window = 100 bytes
- **Bob responds:** Acknowledgment 401 (confirming receipt of bytes 301-400)
- **Window update:** Alice increases window: 100 + 100 = 200 bytes
- **Important:** Window now equals slow start threshold (200 bytes)

Phase 3: Transition to Congestion Avoidance

Reaching Slow Start Threshold:

- **Current window:** 200 bytes = slow start threshold
- **State change:** Slow start $\rightarrow$ Congestion avoidance
- **Alice sends:** Sequence 401, Sequence 501 (200 bytes total)
- **Reason for change:** Alice is approaching the window size that previously caused congestion

Linear Growth Phase: Now Alice switches to much more conservative growth to avoid causing congestion again:

- **Acknowledgment 501 received:** Window = $200 + \frac{100^2}{200} = 200 + 50 = 250$ bytes
- **Acknowledgment 601 received:** Window = $250 + \frac{100^2}{250} = 250 + 40 = 290$ bytes
- **Acknowledgment 701 received:** Window = $290 + \frac{100^2}{290} = 290 + 34 = 324$ bytes

Understanding the Linear Growth Formula:

- **Formula:** New window = Current window + $\frac{(\text{Maximum Segment Size})^2}{\text{Current window}}$
- **Effect:** As window gets larger, the increase gets smaller per acknowledgment
- **Result:** Window grows by approximately 1 Maximum Segment Size per round-trip time
- **Contrast:** Much slower than slow start's exponential doubling

1.7.5 TCP Reno Algorithm Enhancement

Improved congestion control with fast retransmit and fast recovery.
TCP Reno enhances Tahoe by reacting more intelligently to packet loss, distinguishing between single segment loss and severe network congestion.

Fast Retransmit Mechanism

Duplicate ACK Detection Scenario: Consider Alice with 500-byte congestion window sending 5 segments where only the first is lost:

- **Alice sends:** SEQ 301, SEQ 401, SEQ 501, SEQ 601, SEQ 701
- **Network behavior:** First segment (SEQ 301) lost, others received successfully
- **Bob's response:** ACK 301, ACK 301, ACK 301, ACK 301 (duplicate ACKs)

Fast Retransmit Logic: Upon receiving 3 duplicate ACKs, Alice infers single segment loss rather than network collapse:

1. **Immediate retransmit:** Send SEQ 301 without waiting for timeout
2. **Set ssthresh:** $\frac{500}{2} = 250$ bytes
3. **Enter fast recovery:** Specialized state for handling isolated loss

Fast Recovery State

Window Inflation Strategy: Alice must handle segments that are officially unacknowledged but likely received:

1. **Base window:** Set to ssthresh = 250 bytes
2. **Inflation:** Add 3 MSS (300 bytes) for segments indicated by duplicate ACKs
3. **Working window:** 250 + 300 = 550 bytes
4. **New transmission:** Can send 50 additional bytes beyond already transmitted 500 bytes

Continued Fast Recovery: For each additional duplicate ACK received:

- **Interpretation:** One more segment confirmed received by Bob
- **Window adjustment:** Inflate by 1 MSS
- **Transmission:** Send additional data if window permits

Recovery Completion: When Alice receives new ACK (e.g., ACK 801):

1. **Conclusion:** Lost segment successfully retransmitted and received
2. **State transition:** Exit fast recovery
3. **Window reset:** Set congestion window to ssthresh (250 bytes)
4. **Mode switch:** Enter congestion avoidance state