Transport Layer

Date	@October 4, 2024
tag	chapter 3

▼ 3.1 Introduction and Transport-Layer Services

- **Purpose of the Transport Layer:**
  - Provides **logical communication** between application processes on different hosts.
  - Allows applications to send messages without concerning themselves with the underlying physical infrastructure.
- **Implementation of Transport-Layer Protocols:**
  - Implemented only in **end systems** (not in network routers).
  - Converts application-layer messages into **transport-layer segments**.
  - Segments are passed to the **network layer** and encapsulated in **network-layer packets** (datagrams).
  - **Routers** only process the network-layer fields of the datagram.
- **Transport-Layer Protocols:**
  - Multiple transport protocols available for network applications (e.g., **TCP** and **UDP**).
  - Each protocol provides different services.

3.1.1 Relationship Between Transport and Network Layers

- **Layer Comparison:**
  - **Transport layer:** Provides logical communication between **processes** running on different hosts.

    - **Network layer:** Provides logical communication between **hosts**.
- **Analogy:**
    - **Processes = cousins**, **Hosts = houses**, **Transport-layer protocol = Ann and Bill**, **Network-layer protocol = postal service**.
    - Ann and Bill deliver messages between cousins (processes) but are not involved in the actual transport across the network (handled by the postal service).
- **Role of End Systems:**
    - Transport protocols operate in end systems and handle message delivery to and from the **network layer**.
    - Intermediate routers do not process transport-layer information.
- **Multiple Transport Protocols:**
    - Different transport protocols offer **different service models** (e.g., reliability, frequency of delivery).
    - Services provided by transport protocols are constrained by the underlying network layer capabilities.
- **Additional Transport Layer Services:**
    - Some transport protocols offer services that the network layer doesn't provide:
        - **Reliable data transfer** despite network unreliability (e.g., packet loss, corruption).
        - **Encryption** for data confidentiality, even if the network layer doesn't guarantee it.

3.1.2 Overview of the Transport Layer in the Internet

- **Transport Layer Protocols in the Internet:**
    - **UDP (User Datagram Protocol):**
        - Provides **unreliable, connectionless** service.
        - Minimal services: process-to-process delivery and error checking.

        - Does not guarantee data integrity, order, or delivery.
    - **TCP (Transmission Control Protocol):**
        - Provides **reliable, connection-oriented** service.
        - Ensures **reliable data transfer**, data arrives intact and in order.
        - Includes additional services: **flow control, sequence numbers, acknowledgments**, and **timers**.
        - Offers **congestion control** to prevent excessive traffic and ensure fair bandwidth distribution.
- **Terminology Simplification:**
    - **Transport-layer packets** (both TCP and UDP) referred to as **segments**.
    - **Datagram** reserved for network-layer packets to avoid confusion.
- **Network Layer Overview:**
    - **Internet Protocol (IP):**
        - Provides **best-effort delivery** service between hosts.
        - No guarantees for segment delivery, order, or data integrity.
        - **IP Address:** Each host has at least one, used for logical communication.
- **Transport Layer Functions:**
    - Extends IP's host-to-host service to **process-to-process** delivery.
    - Known as **transport-layer multiplexing and demultiplexing**.
- **Service Summary:**
    - **UDP:** Minimal services, process-to-process delivery and error detection.
    - **TCP:** Reliable data transfer, congestion control, converting IP's unreliable service into a reliable one.
- **Congestion Control:**
    - Ensures that one TCP connection does not overwhelm the network.

    - **TCP:** Regulates traffic rate to ensure fair bandwidth usage.
    - **UDP:** Unregulated, allowing applications to send data at any rate.
- **Complexity of TCP:**
    - Reliable data transfer and congestion control add complexity to TCP.
    - Detailed in Sections 3.4 to 3.8, alternating between basic principles and specific TCP implementation.

▼ 3.2 Multiplexing and Demultiplexing

- **Definition**:
    - Transport-layer services extend host-to-host delivery from the network layer to process-to-process delivery for applications running on hosts.
    - Multiplexing gathers data from different processes and sends it to the network.
    - Demultiplexing delivers received data to the correct process on the destination host.
- **Real-world Example**:
    - When downloading web pages and running FTP and Telnet sessions, the transport layer directs incoming data to the correct application.
- **Socket and Process Interaction**:
    - A process communicates via a socket, which acts as a door for data to pass between the network and the application.
    - Each socket has a unique identifier.
- **Multiplexing and Demultiplexing Process**:
    - Multiplexing: Data chunks are gathered from different sockets and encapsulated into segments.

- Demultiplexing: Segments are directed to the appropriate socket based on segment fields like port numbers.

Port Numbers

- **Types**:
  - **Well-known ports** (0 to 1023): Reserved for specific application protocols (e.g., HTTP uses port 80, FTP uses port 21).
  - **Dynamic ports** (1024 to 65535): Used for temporary communication.
- **Demultiplexing with Port Numbers**:
  - The transport layer examines the destination port number in a segment to direct it to the corresponding socket.
  - Each socket is associated with a unique port number.

Connectionless Multiplexing and Demultiplexing (UDP)

- **UDP Socket Identification**:
  - UDP sockets are identified by a two-tuple (destination IP address, destination port number).
- **Example of UDP Communication**:
  - Host A with port 19157 sends data to Host B with port 46428.
  - Transport layer in Host A adds source and destination port numbers and sends the segment to the network layer.
  - Host B receives the segment, examines the destination port number (46428), and directs the data to the correct socket.
- **Creating a UDP Socket:**
  - `clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)`
- **Binding to a Specific Port:**
  - `clientSocket.bind(('', 19157))`

Connection-Oriented Multiplexing and Demultiplexing (TCP)

- **TCP Socket Identification**:
  - TCP sockets are identified by a four-tuple: (source IP address, source port number, destination IP address, destination port number).
- **TCP Example**:
  - When a TCP client requests a connection on port 12000, the server creates a new socket for the connection.
  - The server uses the four values to identify and manage each connection.
- **Multiple TCP Connections**:
  - Multiple connections can share the same destination port but have different source IP addresses or source port numbers.
  - Each connection is distinguished by its four-tuple.

Connection Establishment:

- **Server Side:**
  - **Welcoming Socket:** Listens on a well-known port (e.g., 12000).
  - **Accepting Connections:**
  - `connectionSocket, addr = serverSocket.accept()`
  - Creates a new socket for each incoming connection.
- **Client Side:**
  - Creates a socket and initiates a connection to the server's port.
  - ```
    clientSocket = socket(AF_INET, SOCK_STREAM)
    clientSocket.connect((serverName, 12000))
    ```

Web Servers and TCP

- **Web Server Example**:
  - A web server (e.g., Apache) listens for connections on port 80.
  - Clients connect using source IP addresses and port numbers.
- **Connection Handling**:
    ```

- High-performance web servers often use threads to handle multiple client connections.

Port Scanning

- **Purpose**:
  - Used to identify open ports and the applications running on them.
  - Useful for both system administrators and attackers.
- **Port Scanner Tools**:
  - **nmap**: A widely used port scanner for identifying open TCP and UDP ports.

▼ 3.3 Connectionless Transport: UDP

Overview of UDP

- UDP (User Datagram Protocol) is a minimalistic transport protocol defined in RFC 768.
- It provides basic services like:
  - **Multiplexing/demultiplexing** of messages using port numbers.
  - **Light error checking**.
- Unlike TCP, UDP does not establish a connection before sending data, hence it is called **connectionless**.

How UDP Works

- On the **sending side**, UDP takes messages from the application, adds source and destination port numbers, and passes them to the network layer.
- On the **receiving side**, UDP receives the message from the network layer, uses the destination port number to deliver data to the appropriate application.

- No **handshaking** is required between sending and receiving entities before transmitting a segment.

Key Features of UDP

1. **Finer Application-Level Control**
    - UDP allows the application to send data immediately without waiting for congestion control mechanisms like TCP.
    - Suitable for real-time applications that can tolerate some data loss but require timely delivery.
2. **No Connection Establishment**
    - UDP does not require a three-way handshake like TCP, reducing delays.
    - Example: DNS uses UDP to avoid the connection establishment delay that would slow down name resolution.
3. **No Connection State**
    - UDP does not maintain any connection state (buffers, sequence numbers, etc.) unlike TCP, making it lightweight.
    - This allows servers to handle more clients with UDP than with TCP.
4. **Small Packet Header Overhead**
    - UDP has only 8 bytes of header, compared to TCP's 20 bytes.

Applications That Use UDP

- **DNS**: Avoids connection establishment delays, critical for quick queries.
- **SNMP**: Used for network management, where communication needs to happen even under stressed network conditions.
- **RIP**: Periodic updates replace lost ones, so reliability isn't critical.
- **Multimedia Applications**: UDP is used for real-time video conferencing, Internet telephony, etc., where some packet loss is tolerable.

Why Choose UDP Over TCP?

- Applications that benefit from UDP include those that:
  - Need control over when data is sent (e.g., real-time applications).
  - Don’t require reliable delivery (e.g., multimedia streaming).
  - Can handle packet loss and prefer faster transmission without the overhead of connection establishment.

Controversies with UDP

- **No Congestion Control**: UDP doesn’t adjust the sending rate during congestion, which can lead to high packet loss and potentially degrade TCP traffic.
  - Researchers have proposed mechanisms to enforce adaptive congestion control for UDP.

UDP with Reliability

- Applications can implement their own reliability mechanisms (e.g., acknowledgments, retransmissions) over UDP to achieve reliability without TCP’s congestion control.

Examples of Applications and Protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	Typically proprietary	UDP or TCP
Internet telephony	Typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

3.3.1 UDP Segment Structure

- The structure of a UDP segment is defined in **RFC 768**.
- The segment consists of:
    1. **Source Port**: Identifies the sending application.
    2. **Destination Port**: Allows the receiving host to pass data to the correct process.
    3. **Length Field**: Specifies the length of the UDP segment, including header and data.
    4. **Checksum**: Used for error detection.
- **Data Field**: Contains the application data, such as:
    - DNS: Contains a query or response message.
    - Streaming Audio: Contains audio samples.

Key Points

- The **port numbers** help with demultiplexing (directing data to the correct application).
- The **length field** ensures UDP can handle variable-length data segments.
- **Checksum** allows for error detection in the segment.

3.3.2 UDP Checksum

Purpose of UDP Checksum

- Provides **error detection** by checking whether bits have been altered during transmission (due to noise or storage issues).
- Calculated by performing the **1's complement** of the sum of all 16-bit words in the segment.

Example of Checksum Calculation

1. Suppose we have three 16-bit words:

- 0110011001100000
- 0101010101010101
- 1000111100001100

2. **Summing** these words:
   - First two: 0110011001100000 + 0101010101010101 = 1011101110110101
   - Adding the third word: 1011101110110101 + 1000111100001100 = 0100101011000010 (with overflow wrapped around, add the msb with lsb).
3. **1's complement** of the sum: 0100101011000010 → 1011010100111101 (the checksum).
4. At the receiver, all 16-bit words, including the checksum, are summed. If no errors are introduced, the sum should be **1111111111111111**.

Why UDP Provides a Checksum

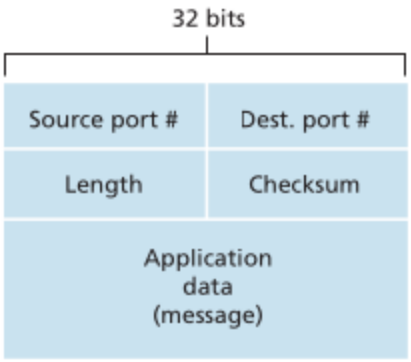
- Some **link-layer protocols** (e.g., Ethernet) provide error checking, but not all links guarantee this.
- Errors can occur when segments are stored in a router's memory.
- UDP ensures **end-to-end error detection** as part of the **end-to-end principle** in system design.

What Happens if Errors are Detected?

- UDP does not correct errors, but:
  - Some implementations **discard** the damaged segment.
  - Others **pass it** to the application with a warning.

Conclusion on UDP

- **UDP** offers basic services like multiplexing, demultiplexing, and error detection but lacks reliability features like TCP.
- Next, the focus shifts to **TCP**, which provides reliable data transfer and other advanced services.



▼ 3.4 Principles of Reliable Data Transfer

Overview

- **Reliable Data Transfer (RDT)** is crucial across different layers of networking: transport, link, and application layers.
- Ensures data is delivered without corruption, loss, and in the correct order.
- **TCP** is a practical example of a protocol that implements reliable data transfer over an unreliable layer (IP).

Framework for Reliable Data Transfer

- **Service Abstraction**: Provides a reliable channel, ensuring:
  - No data corruption.
  - No data loss.
  - Data is delivered in the order it was sent.
- **Challenges**:
  - The layer beneath the reliable protocol can be unreliable (e.g., IP network layer).
  - The lower layer may corrupt or lose data, but must not reorder packets.

Reliable Data Transfer Protocol

- A reliable data transfer protocol must:
  - Handle **packet corruption** and **packet loss**.
  - Implement mechanisms to ensure correct packet delivery.

Functions in Reliable Data Transfer

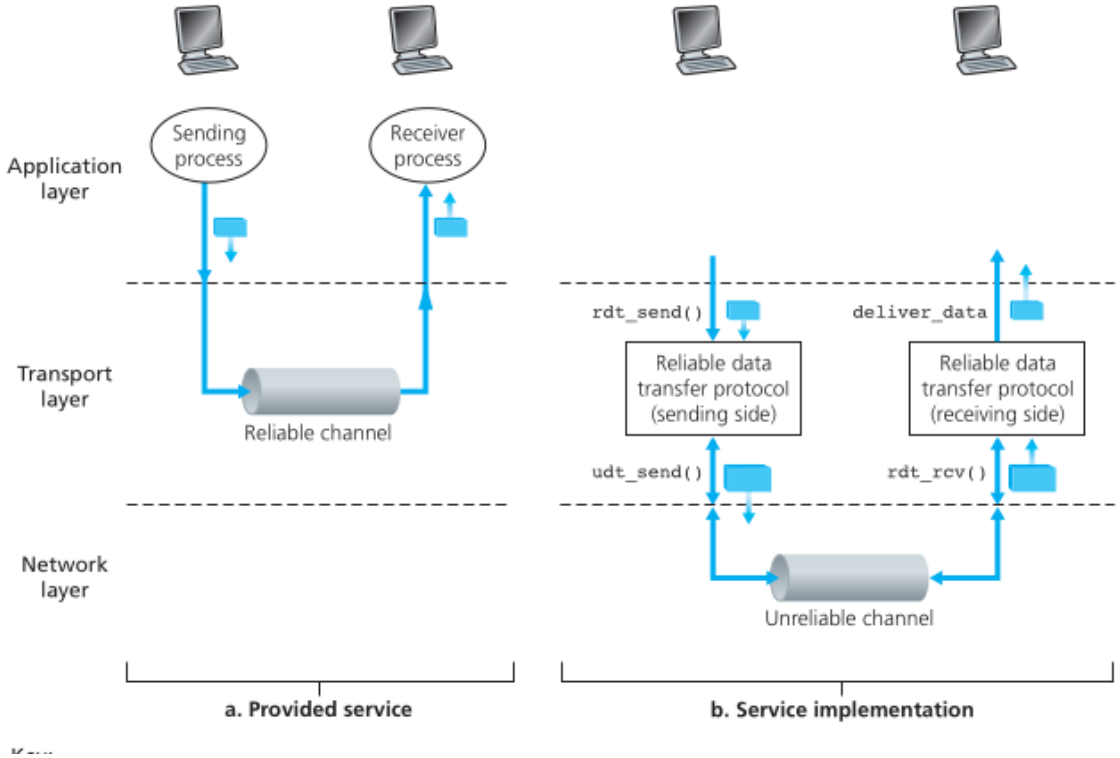
- **rdt_send()**: Called by the sender to send data to the receiver.
- **rdt_rcv()**: Called when a packet arrives at the receiving side.
- **deliver_data()**: Passes data from the receiver to the upper layer (application).

Key Concepts

- **Unidirectional Data Transfer**: Focuses on data transfer from sender to receiver.
  - Even in unidirectional transfer, both sender and receiver exchange **control packets**.
  - Uses **udt_send()** for unreliable data transfer, ensuring data flows even over unreliable channels.

Conclusion

- The reliable data transfer protocol operates over an unreliable layer, ensuring reliable, ordered, and error-free delivery of data.
- The sender and receiver must exchange both data and control packets to achieve reliability.



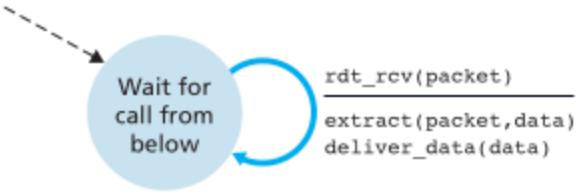
3.4.1 Building a Reliable Data Transfer Protocol

Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0

- **Key Features:**
  - Underlying channel is completely reliable.
  - No packet corruption or loss.
  - Finite-State Machine (FSM) for sender and receiver both have one state.
  - Actions of sender:
    - Accepts data from the upper layer.
    - Creates a packet and sends it.

a. rdt1.0: sending side

- Actions of receiver:
  - Receives packet, extracts data, and delivers it to the upper layer.



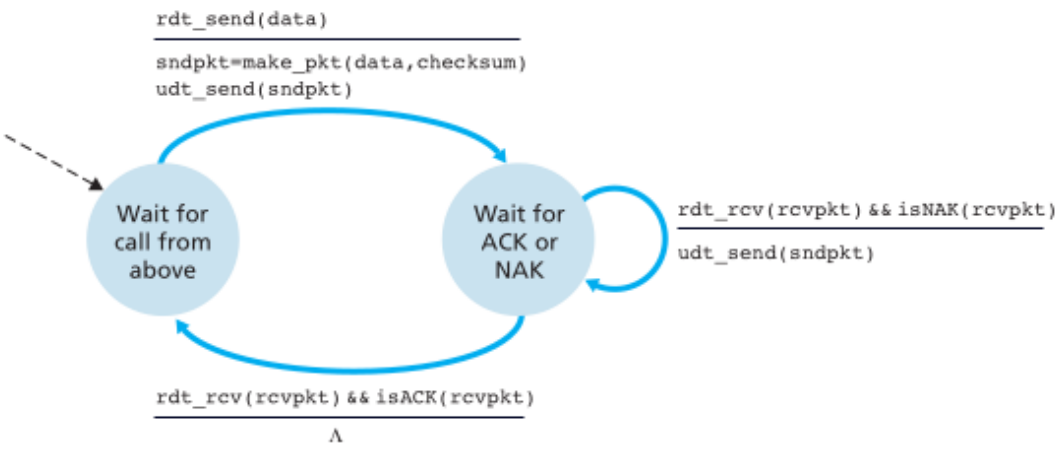
b. rdt1.0: receiving side

- No feedback from receiver to sender is required as the channel is perfect.

Reliable Data Transfer over a Channel with Bit Errors: rdt2.0

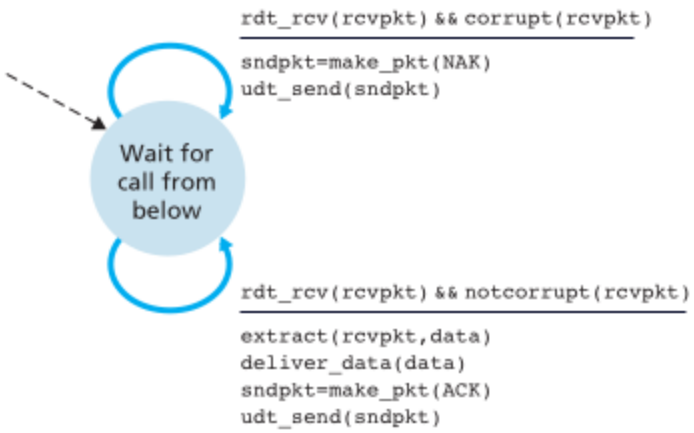
- **Problem**: Channel introduces bit errors, although packets are delivered in order.
- **Solution**: Introduce **ARQ (Automatic Repeat reQuest)** protocols.
  - **Required Capabilities:**
    - **Error detection**: Detect bit errors using checksum.
    - **Receiver feedback**: Use ACK (positive acknowledgment) and NAK (negative acknowledgment) to inform the sender of correct or incorrect packet reception.
    - **Retransmission**: Retransmit corrupted packets.
- **Protocol Mechanism**:
  - **Sender FSM**:
    - Two states: waiting for data and waiting for ACK/NAK.

- After sending a packet, the sender waits for receiver feedback (ACK/NAK).
- No new data is sent until ACK is received for the current packet.
- This is a **stop-and-wait protocol**.



a. rdt2.0: sending side

- **Receiver FSM**:
  - Single state: responds with ACK if packet is correct, NAK if corrupted.

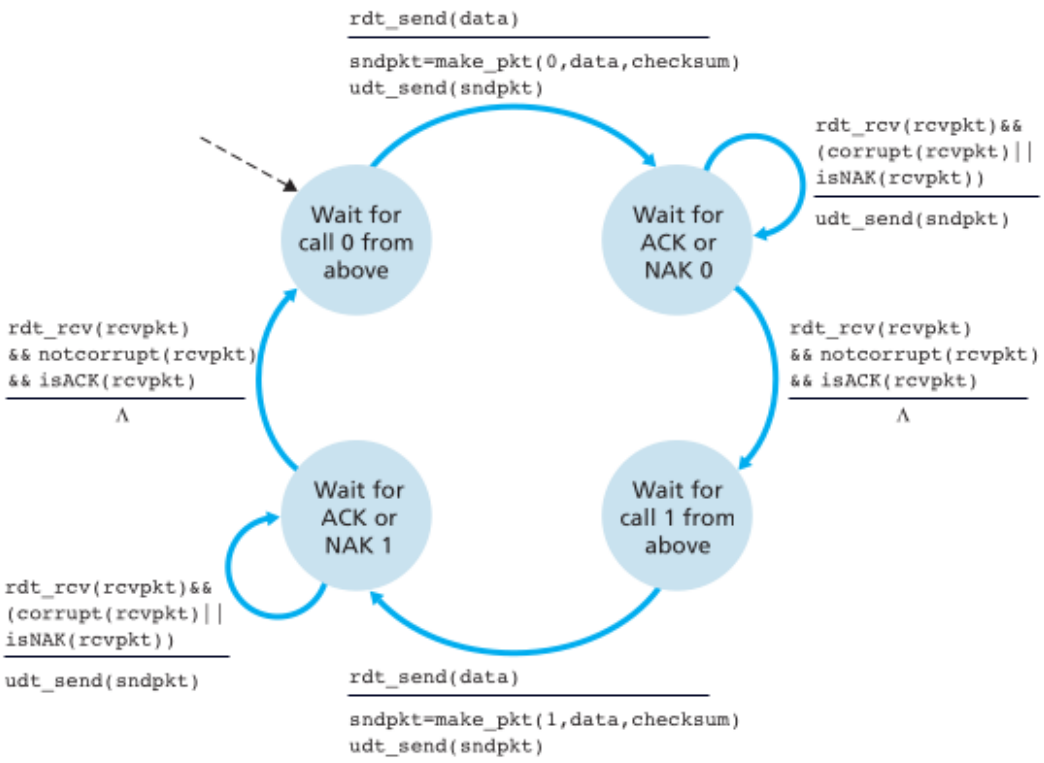


b. rdt2.0: receiving side

- **Flaw**: ACK/NAK packets themselves could be corrupted, and the sender has no way of knowing whether the receiver received the last packet correctly.

Reliable Data Transfer with Error Handling: rdt2.1

- **New Features**:
  - Adds sequence numbers to packets to handle retransmissions of corrupted ACK/NAK.
  - Sender can detect duplicate packets using sequence numbers.
  - Sender retransmits packet when receiving a corrupted ACK(0)/NAK(1).
  - Protocol uses a 1-bit sequence number (0 or 1).
- **FSM Changes**:
  - Sender and receiver FSMs now have multiple states to handle sequence numbers.
  - Receiver sends ACK for out-of-order packets and duplicate ACKs for packets received twice.



ure 3.11 ♦ rdt2.1 sender

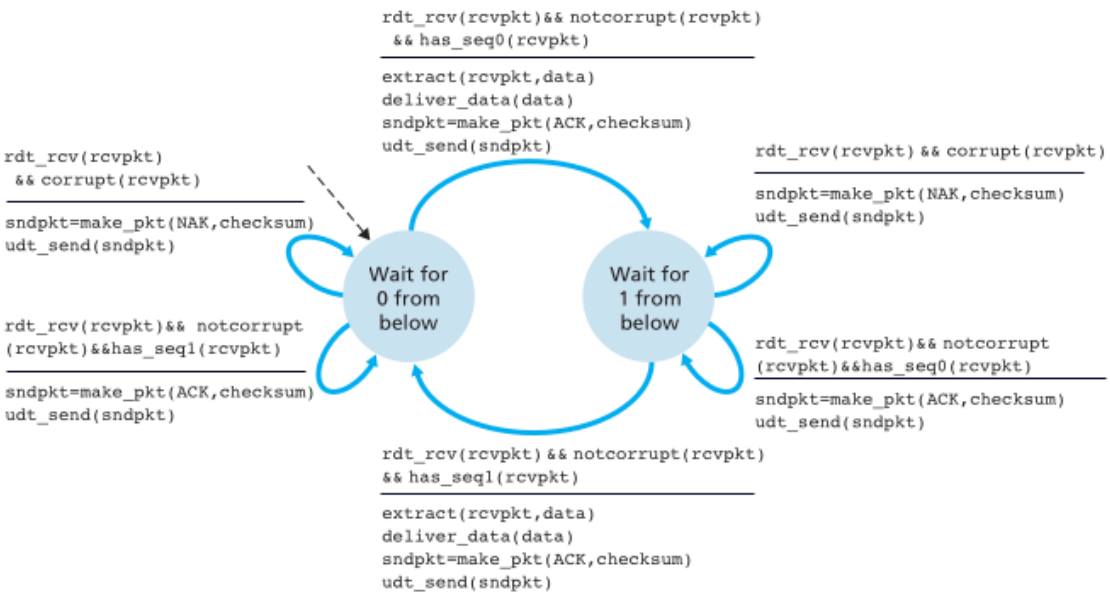


Figure 3.12 ♦ rdt2.1 receiver

▼ Sender FSM explanation

1. Wait for call 0 from above:

- The sender is ready to send a packet with **sequence number 0**.
- When it gets data from the application layer (`rdt_send(data)`), it creates a packet with sequence number 0: `sndpkt = make_pkt(0, data, checksum)`.
- This sequence number 0 is used to label the packet and differentiate it from future packets that will have sequence number 1.
- The packet is then sent using `udt_send(sndpkt)` (Unreliable Data Transfer, where the network might corrupt the packet).

2. Wait for ACK or NAK 0:

- After sending the packet with sequence number 0, the sender moves to the state **"Wait for ACK or NAK 0"**.
- The sender waits for feedback from the receiver, which can be either:

- **ACK 0**: The receiver successfully received the packet with sequence number 0, so the sender can proceed to the next packet with sequence number 1.
- **NAK 0** or corrupted packet: The receiver either got a corrupted packet or the packet didn't arrive correctly. In this case, the sender **retransmits the same packet** with sequence number 0.

If a valid ACK 0 is received (`rdt_recv(rcvpkt) && isACK(rcvpkt) && notcorrupt(rcvpkt)`), the sender knows the packet has been correctly received and moves to the next state, **"Wait for call 1 from above"**.

3. Wait for call 1 from above:

- Now, the sender prepares to send the next packet with **sequence number 1**.
- The process is similar to the first state, but this time, the packet created will have sequence number 1: `sndpkt = make_pkt(1, data, checksum)`.
- This sequence number 1 tells the receiver that this is the second packet, different from the first one that had sequence number 0.

4. Wait for ACK or NAK 1:

- The sender waits for feedback for packet 1.
- If an **ACK 1** is received, the sender knows that packet 1 was received correctly, and it returns to **"Wait for call 0 from above"** to send the next packet with sequence number 0.
- If a **NAK 1** or corrupted packet is received, the sender retransmits packet 1.

The Importance of the Sequence Number:

- **Alternating between 0 and 1**:
  - The sender alternates between sequence number 0 and sequence number 1 for each packet it sends. This allows the receiver to distinguish between new packets and

retransmissions. If a packet is corrupted and the receiver receives it again, the sequence number helps detect whether it's a retransmission or a new packet.

- **Handling duplicates**:
  - The sequence number ensures that the receiver can recognize when it gets the same packet twice. For example, if a packet with sequence number 0 is retransmitted due to an issue, the receiver will notice it's the same packet and handle it accordingly (by sending another ACK without delivering the same data twice).

▼ Receiver FSM

1. Wait for 0 from below:

- The receiver is expecting a packet with **sequence number 0**.
- When a packet arrives, the receiver performs two checks:
  - **Is the packet corrupted?** (`rdt_recv(rcvpkt) && notcorrupt(rcvpkt)`): If the packet is **corrupted**, the receiver sends a **NAK** (Negative Acknowledgment) for sequence 0, indicating that the packet needs to be resent (`sndpkt = make_pkt(NAK, checksum)`).
  - If a packet arrives out of order (e.g., the receiver is expecting sequence 0, but sequence 1 arrives again), the receiver can identify that this is a duplicate and **ignore** it or send another acknowledgment for that packet.
  - **Does the packet have sequence number 0?** (`rdt_recv(rcvpkt) && has_seq0(rcvpkt)`): If the packet is correct and has the expected sequence number 0, the receiver:
    - Extracts the data from the packet (`extract(rcvpkt, data)`).
    - Delivers the data to the upper layer (`deliver_data(data)`).
    - Sends an **ACK 0** to the sender (`sndpkt = make_pkt(ACK, checksum)`), confirming that packet 0 was received successfully.

- After sending the ACK 0, the receiver moves to the next state, **"Wait for 1 from below"**, expecting the next packet in the sequence with sequence number 1.

2. Wait for 1 from below:

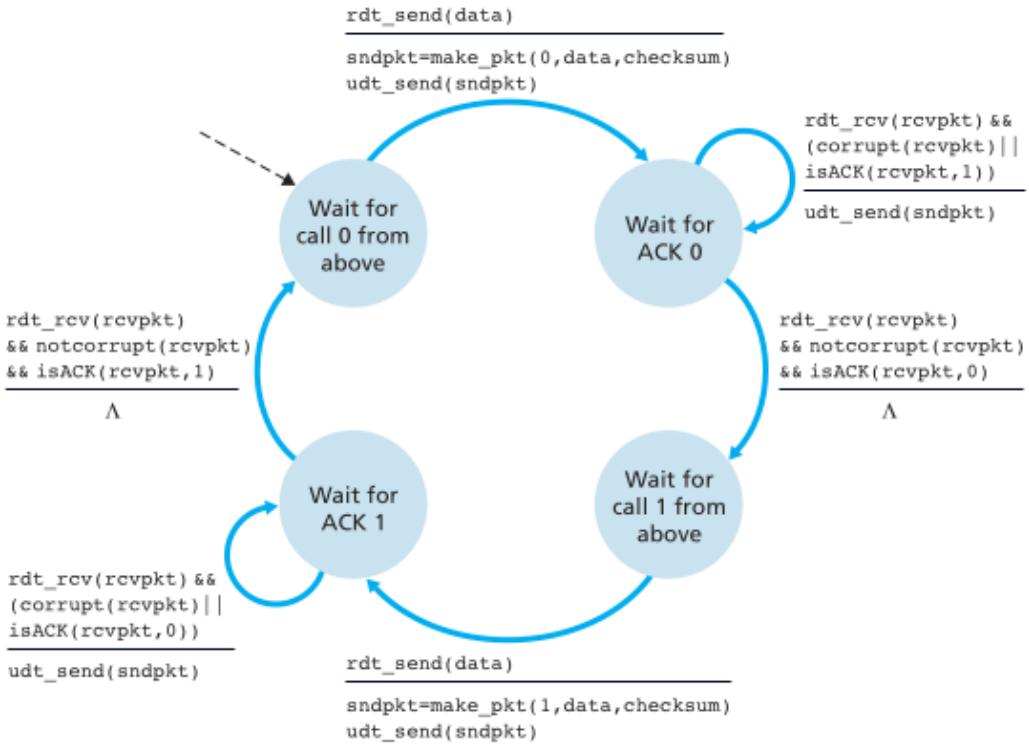
- The receiver is now expecting a packet with **sequence number 1**.
- Similar to the previous state, the receiver checks:
  - **Is the packet corrupted?** (`rdt_recv(rcvpkt) && notcorrupt(rcvpkt)`): If the packet is corrupted, the receiver sends a **NAK 1** (Negative Acknowledgment), asking for a retransmission.
  - If a packet arrives out of order (e.g., the receiver is expecting sequence 1, but sequence 0 arrives again), the receiver can identify that this is a duplicate and **ignore** it or send another acknowledgment for that packet.
  - **Does the packet have sequence number 1?** (`rdt_recv(rcvpkt) && has_seq1(rcvpkt)`): If the packet is correct and has sequence number 1, the receiver:
    - Extracts the data (`extract(rcvpkt, data)`).
    - Delivers the data to the upper layer (`deliver_data(data)`).
    - Sends an **ACK 1** to the sender, confirming successful receipt of packet 1.
- After sending the ACK 1, the receiver moves back to the **"Wait for 0 from below"** state, ready to receive the next packet in the sequence with sequence number 0.

How Sequence Numbers Work in the Receiver:

- The sequence numbers (0 or 1) tell the receiver **which packet** it should be expecting at any given time.
- The receiver alternates between waiting for packets with sequence number 0 and packets with sequence number 1.

NAK-Free Reliable Data Transfer: rdt2.2

- **Improvement**: Replace NAK with duplicate ACKs.
  - ACKs carry the sequence number of the last correctly received packet.
  - Sender knows a packet was lost if it receives two ACKs for the same sequence number.
  - Problem: if first ack fails then no ack is sent then infinite wait occurs



igure 3.13 ♦ rdt2.2 sender

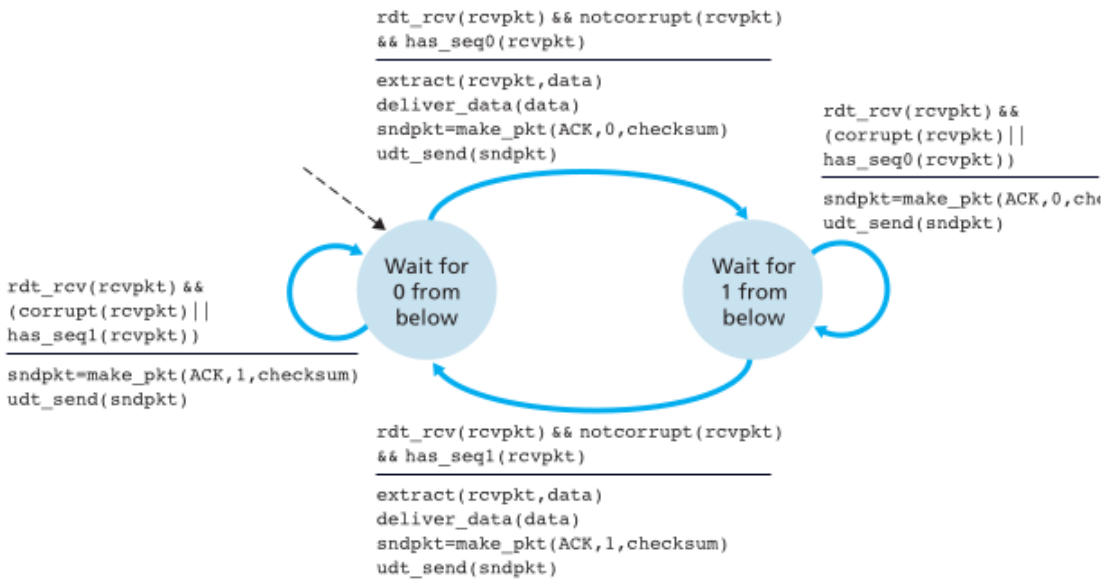


Figure 3.14 ♦ rdt2.2 receiver

Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0

- **Problem**: Channel can lose packets in addition to corrupting them. (Both data and ack)
- **Solution**: Introduce a timer for detecting packet loss.
  - **Timer-based retransmission**:
    - If no ACK is received within a specified time, the sender retransmits the packet.
    - Sender assumes packet loss after a round-trip delay and retransmits.
  - **Handling duplicates**: The existing sequence number mechanism handles duplicate packets.
- **FSM for rdt3.0**:
  - Sender starts a timer when sending a packet.
  - If the timer expires without receiving an ACK, the sender retransmits the packet.

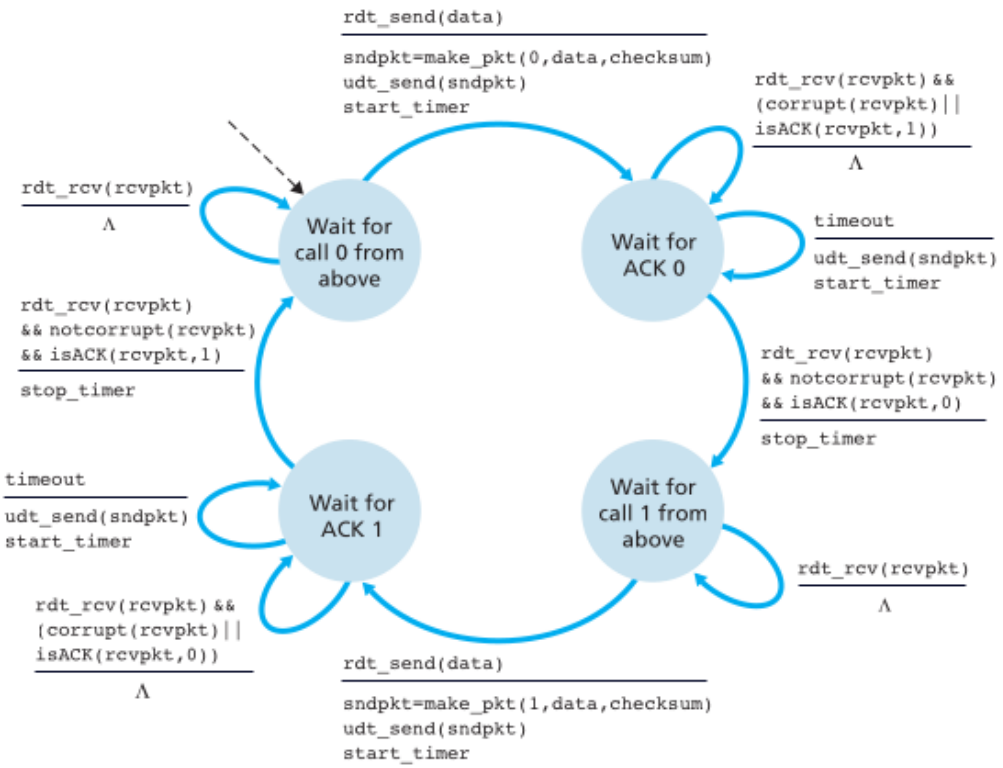


Figure 3.15 ♦ rdt3.0 sender

3.4.2 Pipelined Reliable Data Transfer Protocols

Limitations of Protocol rdt3.0:

- **Stop-and-wait behavior**: rdt3.0 is a stop-and-wait protocol, which leads to performance issues, especially in high-speed networks.
- **Example**: Two hosts on opposite coasts of the U.S. prop delay 15 so Rtt (30ms round-trip time) using a 1 Gbps link.
  - Packet size: 1,000 bytes (8,000 bits)
  - **Transmission time (d_trans)**:
    - $d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8\mu sec$
  - **Round-trip time (RTT)**:
    - With stop-and-wait protocol, the sender sends a packet and waits for the acknowledgment (ACK), leading to very low efficiency.

- **Sender Utilization (Usender)**:
  - $U_{sender} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30.008} = 0.00027$
  - The sender is busy only 0.027% of the time, leading to poor throughput (267 kbps) on a 1 Gbps link.

Solution: Pipelining

- **Pipelining**: Instead of waiting for an ACK after each packet, the sender can transmit multiple packets before waiting for acknowledgments.
  - **Impact**: This significantly increases sender utilization by allowing multiple packets to be in transit simultaneously.
  - **Visualization**: Multiple in-transit packets can be visualized as filling a "pipeline."

Consequences of Pipelining on Reliable Data Transfer Protocols:

1. **Increased Range of Sequence Numbers**:
   - Each in-transit packet must have a unique sequence number.
   - More sequence numbers are needed due to multiple unacknowledged packets.
2. **Buffering Requirements**:
   - **Sender**: Needs to buffer packets that are transmitted but not yet acknowledged.
   - **Receiver**: May need to buffer correctly received packets to handle out-of-order arrivals.
3. **Error Recovery Approaches**:
   - **Go-Back-N**: Involves retransmitting all packets after a lost or corrupted packet.
   - **Selective Repeat**: Only the specific lost or corrupted packets are retransmitted, reducing unnecessary transmissions.

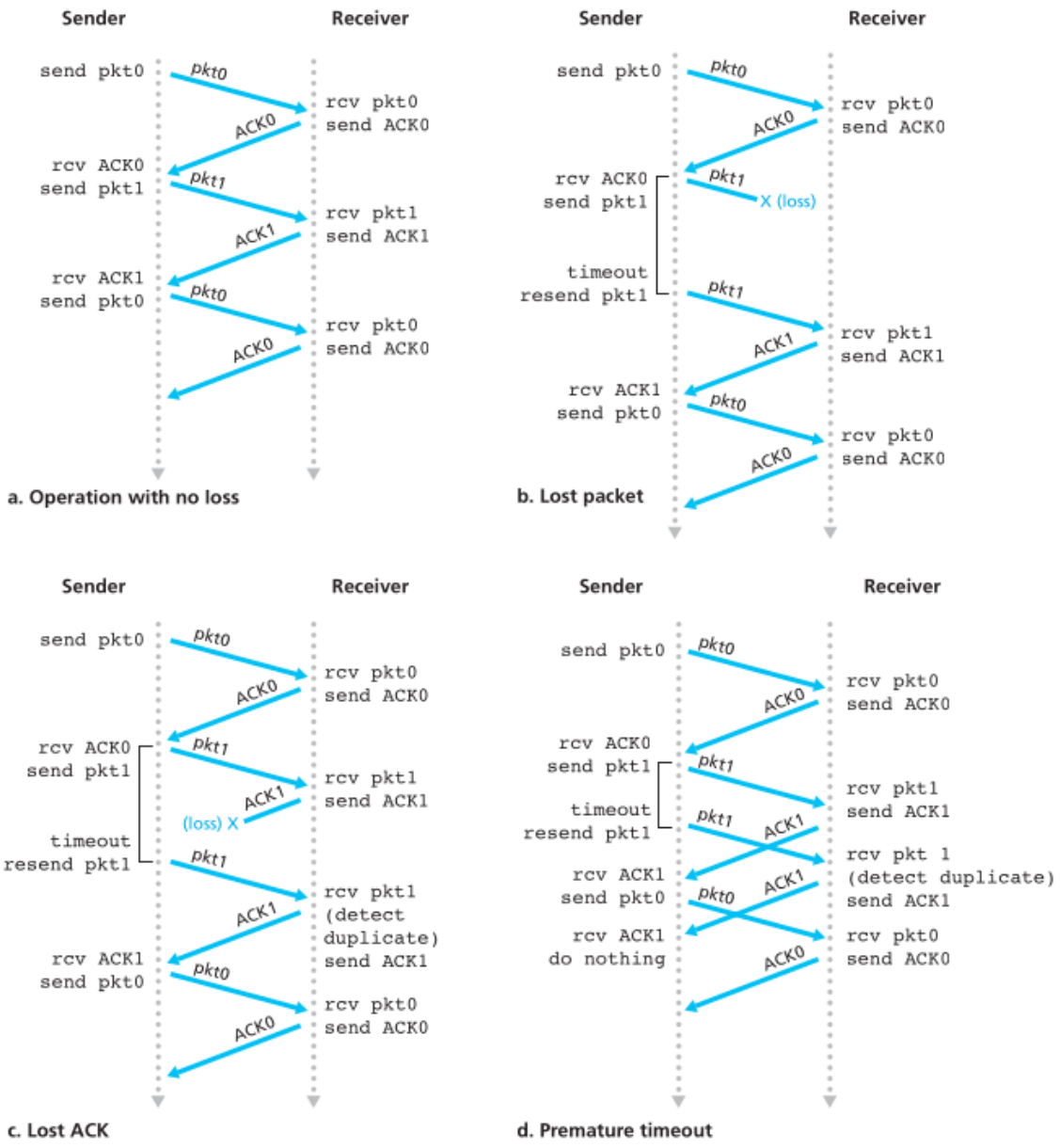


Figure 3.16 ♦ Operation of `rdt3.0`, the alternating-bit protocol

3.4.3 Go-Back-N (GBN) Protocol

Overview of Go-Back-N (GBN)

- Allows sender to transmit multiple packets without waiting for an acknowledgment.
- Sender can have a maximum of **N** unacknowledged packets in the pipeline.

Key Concepts in GBN:

1. **Base and Next Sequence Number**:
   - **Base**: Sequence number of the oldest unacknowledged packet.
   - **Next Sequence Number (nextseqnum)**: The smallest unused sequence number.
   - Four intervals of sequence numbers:
     - `[0, base-1]`: Packets sent and acknowledged.
     - `[base, nextseqnum-1]`: Packets sent but not acknowledged.
     - `[nextseqnum, base+N-1]`: Sequence numbers for new packets.
     - `[base+N, ∞]`: Sequence numbers not yet usable until earlier packets are acknowledged.

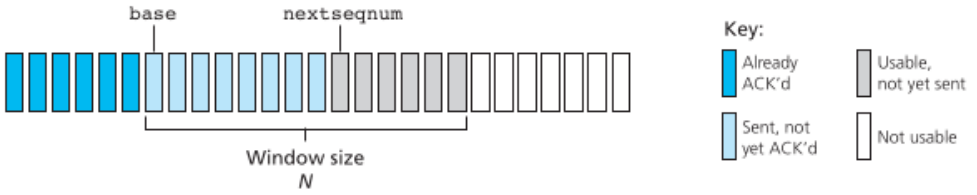


Figure 3.19 ♦ Sender's view of sequence numbers in Go-Back-N

2. **Sliding Window**:
   - GBN is a **sliding-window protocol**, where the window slides over the sequence numbers as packets are acknowledged.
   - **Window size (N)**: Limits the number of outstanding unacknowledged packets.
3. **Sequence Number Arithmetic**:
   - Sequence numbers are finite and use **modulo arithmetic**.
   - If the sequence number field has **k bits**, the sequence number range is `[0, 2^k - 1]`.

Events in GBN Protocol:

1. **Invocation from Upper Layer**:
    - Sender checks if the window is full (i.e., N unacknowledged packets).
    - If the window is not full, the packet is created, sent, and variables are updated.
    - If the window is full, the sender returns the data to the upper layer (or buffers it for later transmission).
2. **Receipt of ACK**:
    - Acknowledgment is **cumulative**: ACK for packet n means all packets up to n have been received.
    - When an ACK is received, the sender slides the window forward.
3. **Timeout and Retransmission**:
    - If a timeout occurs, the sender retransmits all unacknowledged packets starting from the oldest unacknowledged packet.
    - A **single timer** is used for the oldest transmitted but unacknowledged packet.

Receiver's Role in GBN:

1. **Correct and In-Order Packet**:
    - If the received packet is in order, the receiver sends an ACK and delivers the data to the upper layer.
2. **Out-of-Order Packet**:
    - The receiver discards out-of-order packets and resends the last ACK.
    - Simplifies buffering requirements: the receiver only needs to track the sequence number of the next in-order packet.

Advantages and Disadvantages:

- **Advantage**: Simplicity in receiver buffering—no need to buffer out-of-order packets.

- **Disadvantage**: Discarding out-of-order packets may lead to unnecessary retransmissions.

GBN Example:

- For a window size of 4 packets:
  - Sender sends packets 0-3, then waits for acknowledgments.
  - If packet 2 is lost, packets 3, 4, and 5 are discarded by the receiver as out-of-order.

GBN Protocol Implementation:

- The protocol would be implemented using event-based programming, with events triggered by:
  - **rdt_send()**: Called from the upper layer.
  - **Timer interrupt**: Triggers retransmission on timeout.
  - **rdt_rcv()**: Called when a packet is received.

Techniques Used in GBN:

- Sequence numbers.
- Cumulative acknowledgments.
- Checksum verification.
- Timeout and retransmission mechanism.

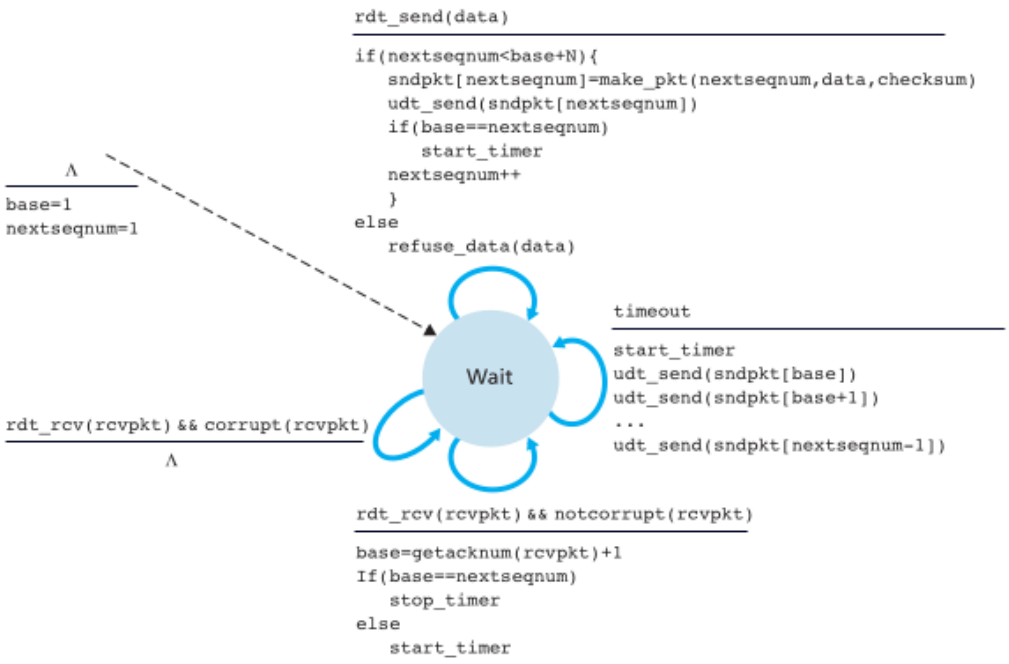


Figure 3.20 ♦ Extended FSM description of GBN sender

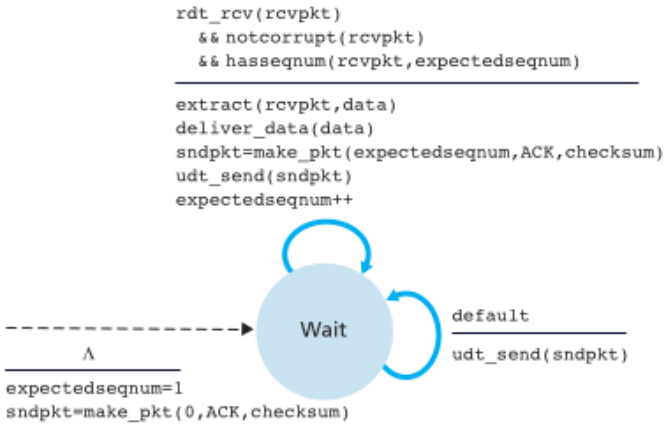


Figure 3.21 ♦ Extended FSM description of GBN receiver

3.4.4 Selective Repeat (SR) Protocol

Overview

- **Purpose**: Avoids unnecessary retransmissions in contrast to Go-Back-N (GBN) by only retransmitting packets suspected to be lost or corrupted.

- **Window Size**: Limits the number of outstanding, unacknowledged packets, allowing for individual acknowledgment of packets.
- **Operation**:
  - Sender retransmits only lost/corrupted packets.
  - Receiver buffers out-of-order packets until all missing packets are received.

Sender and Receiver Views

- **Sender View**:
  - Transmits packets within the window size.
  - Receives individual acknowledgments for packets, unlike GBN.
  - Uses sequence number space to track packet transmission status.
- **Receiver View**:
  - Acknowledges every correctly received packet, whether in order or not.
  - Buffers out-of-order packets until all missing packets are received, then delivers them to the upper layer.

Key Scenarios and Examples

1. **Out-of-order Reception**:
   - Receiver buffers packets 3, 4, 5, and delivers them with packet 2 once it’s received.
2. **Reacknowledgment of Received Packets**:
   - Important for ensuring the sender's window progresses even if some ACKs are lost.
   - Example: If the sender does not receive an ACK for `send_base`, it will retransmit even if the packet was already received.
3. **Window Misalignment**:

- Sender and receiver windows might not always coincide, leading to retransmissions of old packets (sequence number overlaps).
- **Example**:
  - Sequence number space of 0, 1, 2, 3 with a window size of 3 can lead to retransmission confusion between old and new packets.

Consequences of Finite Sequence Number Space

- **Reordering Problem**:
  - In a network, packets may be reordered or delayed, leading to retransmission confusion.
  - To avoid duplicate packet issues, sequence numbers are not reused until it's ensured that the packet has been fully transmitted and acknowledged.

Important Aspects of SR Protocol

- **Timers**: Each packet has its own timer to handle individual timeouts.
- **Acknowledgments**: If the window moves forward, untransmitted packets are sent.

Sender Actions in SR Protocol

1. **Data Received from Above**:
   - Checks sequence number availability.
   - Sends packet if within window size, otherwise buffers it.
2. **Timeout Occurs**:
   - Each packet has its own timer, and only the specific timed-out packet is retransmitted.
3. **ACK Received**:
   - Marks the packet as received.
   - Moves the window forward if necessary and sends untransmitted packets.

Receiver Actions in SR Protocol

1. **Packet Received Within Window**:
    - Sends ACK.
    - Buffers packet if out-of-order.
    - Delivers consecutively numbered buffered packets to upper layer.
2. **Packet Received Outside Window**:
    - Generates ACK even if it was previously acknowledged.
3. **Invalid Packet**:
    - The packet is ignored.

Summary of Reliable Data Transfer Mechanisms

Mechanism	Use
Checksum	Detects bit errors in transmitted packets.
Timer	Retransmits packets if lost or delayed; handles ACK loss.
Sequence Number	Ensures proper packet ordering and helps detect lost or duplicate packets.
Acknowledgment (ACK)	Confirms correct reception of packets by the receiver.
Negative Acknowledgment (NACK)	Indicates incorrect reception or loss of a packet.
Window, Pipelining	Controls the number of unacknowledged packets to improve efficiency.

▼ 3.5 Connection-Oriented Transport: TCP

Overview

- **TCP (Transmission Control Protocol)**:
  - Connection-oriented transport-layer protocol.
  - Provides reliable data transfer.

3.5.1 The TCP Connection

- **Connection Establishment**:
  - TCP requires a handshake before data transfer.
  - Both sides initialize TCP state variables.
- **Characteristics of TCP Connection**:
  - **Not a physical circuit**: Connection state resides in the end systems (hosts).
  - **Full-duplex service**: Data can flow simultaneously in both directions.
  - **Point-to-point**: A TCP connection exists between a single sender and a single receiver; multicasting is not supported.
- **Connection Establishment Procedure**:
  - Initiating process (client) requests connection to another process (server).
  - Example of establishing connection in Python:

    ```
    clientSocket.connect((serverName, serverPort))
    ```

  - Three-way handshake process:
    1. Client sends a special TCP segment.
    2. Server responds with a second segment.
    3. Client sends a third segment (may include payload).

Data Transmission

- **Sending Data**:

    - Data is passed through a socket and directed to the TCP send buffer.
    - TCP sends buffered data to the network layer at its convenience.
- **Maximum Segment Size (MSS)**:
    - Limits the amount of data sent in a segment.
    - Determined by the largest link-layer frame (MTU).
    - Common MSS value: 1,500 bytes for Ethernet and PPP.
- **Encapsulation**:
    - TCP segments are encapsulated in IP datagrams.
    - Received segments are placed in the TCP receive buffer for the application to read.

Connection Structure

- **Components of TCP Connection**:
    - Each side has:
        - Send buffer
        - Receive buffer
        - Variables associated with the connection
- **No connection state in network elements** (routers and switches).

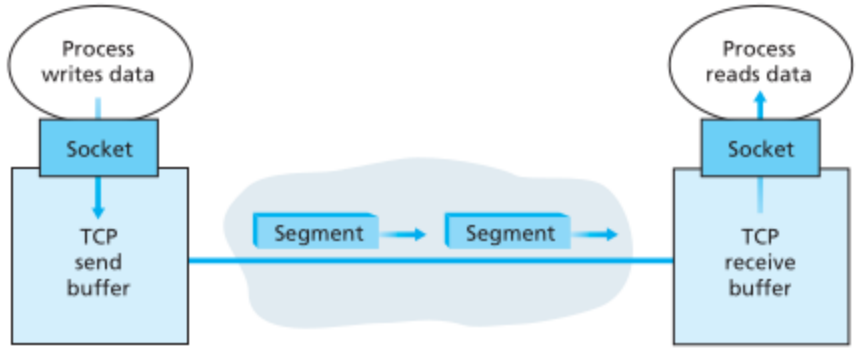
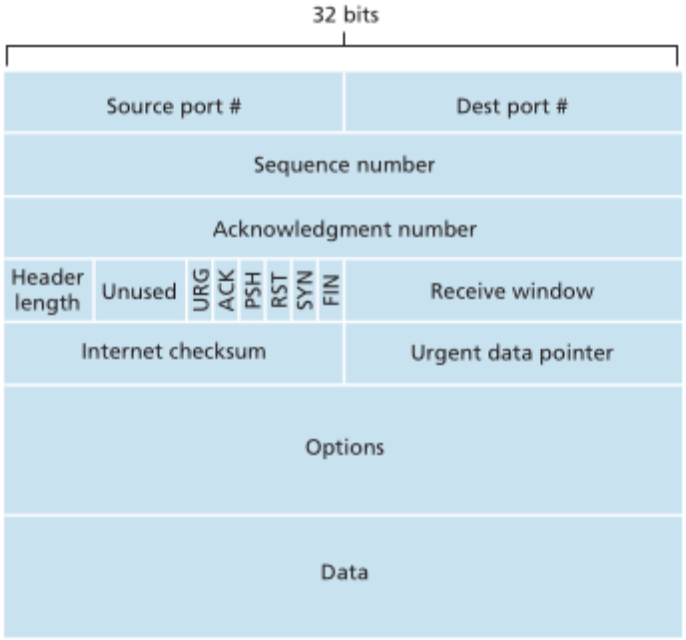


Figure 3.28 ♦ TCP send and receive buffers

3.5.2 TCP Segment Structure



Overview of TCP Segment

- **Components**:
  - **Header Fields**: Control the transfer of data.
  - **Data Field**: Contains application data.
  - **MSS (Maximum Segment Size)**: Limits the maximum size of the data field.

Characteristics of TCP Segments

- **Data Chunk Sizes**:
  - Large files (e.g., images) are broken into MSS-sized chunks.
  - Interactive applications (e.g., Telnet) may send smaller chunks, sometimes as small as one byte.
- **Header Size**: Typically 20 bytes, larger than the UDP header.

TCP Segment Header Fields

- **Port Numbers**: For multiplexing/demultiplexing data.
- **Checksum Field**: For error-checking.
- **Sequence Number**: 32-bit field indicating the byte-stream number of the first byte in the segment.
- **Acknowledgment Number**: 32-bit field indicating the next byte expected from the other host.
- **Receive Window**: 16-bit field used for flow control; indicates bytes receiver is willing to accept.
- **Header Length**: 4-bit field indicating the length of the TCP header in 32-bit words.
- **Options Field**: Optional and variable-length; used for negotiating MSS and window scaling.
- **Flag Field**: 6 bits indicating control flags:
  - **ACK**: Valid acknowledgment number.
  - **RST, SYN, FIN**: Connection management.
  - **PSH**: Immediate data processing by receiver.
  - **URG**: Urgent data indication (with an urgent data pointer).

Sequence Numbers and Acknowledgment Numbers

- **Sequence Numbers**:
  - Reflect the unstructured, ordered stream of bytes.
  - Each byte in the data stream is numbered.
- **Acknowledgment Numbers**:
  - Indicates the next expected byte from the sender.
  - Example:
    - If Host A has received bytes 0-535, it expects byte 536; thus, it sends 536 in the acknowledgment number.

Handling Out-of-Order Segments

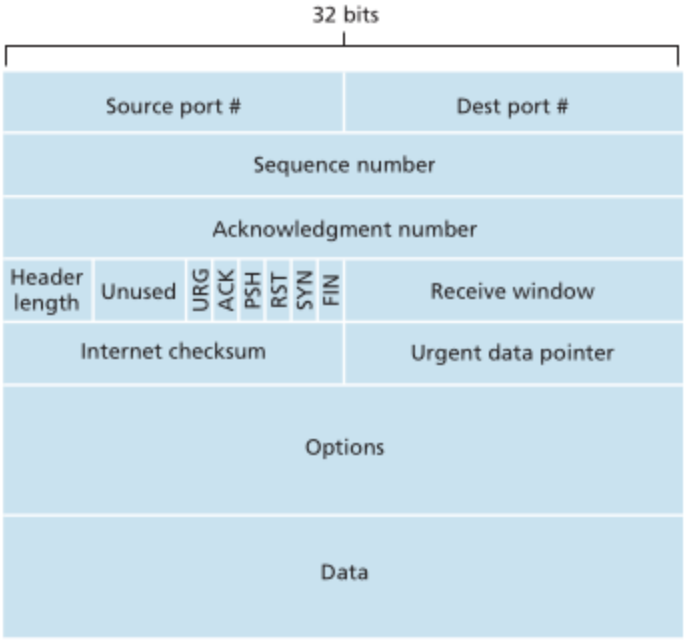
- TCP can receive segments out of order:
  - **Choices**:
    1. Discard out-of-order segments.
    2. Keep out-of-order bytes until missing bytes arrive (commonly used).

Initial Sequence Number

- Randomly chosen for both sides to avoid confusion with previously terminated connections.

Telnet: Case Study for TCP

- **Definition**: Application-layer protocol for remote login, runs over TCP.
- **Character Transmission**:
  - Each character is sent and echoed back for confirmation.
- **Example**:
  - Host A sends ‘C’ with sequence number 42; Host B acknowledges with 43 and sends ‘C’ back.
  - Acknowledgment is piggybacked on the segment carrying data from the server to the client.



3.5.3 Round-Trip Time Estimation and Timeout

Overview of TCP Timeout Mechanism

- TCP uses a **timeout/retransmit mechanism** to recover lost segments.
- Key considerations for implementing this mechanism:
  - Timeout interval must be longer than the **Round-Trip Time (RTT)** to avoid unnecessary retransmissions.
  - Estimation of RTT is crucial.
  - Timer association with unacknowledged segments.

Estimating Round-Trip Time

- **SampleRTT**:
  - Time taken between segment sent and acknowledgment received.
  - Only one SampleRTT is measured at a time.
  - No SampleRTT for retransmitted segments.

- **EstimatedRTT**:
  - Average of SampleRTT values.
  - Updated using the formula:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

  - Recommended value: $\alpha = 0.125$
- **Exponential Weighted Moving Average (EWMA)**:
  - Recent samples are weighted more heavily than older ones.
- **RTT Variation (DevRTT)**:
  - Measures variability of SampleRTT around EstimatedRTT.
  - Updated using the formula:

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}$$

  - Recommended value: $\beta = 0.25$

Retransmission Timeout Interval

- **Setting the Timeout Interval**:
  - Should be greater than or equal to EstimatedRTT.
  - Set using the formula:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

  - Initial recommended value: **1 second**.
- **Timeout Adjustment**:
  - If a timeout occurs, double the TimeoutInterval to avoid premature timeouts for subsequent segments.
  - After receiving an acknowledgment, recompute TimeoutInterval using the formula above.

3.5.4 Reliable Data Transfer

Overview of IP and TCP

- **IP Service**:
  - Unreliable; does not guarantee:
    - Datagram delivery.
    - In-order delivery of datagrams.
    - Integrity of data in datagrams.
  - Issues:
    - Datagrams may overflow router buffers.
    - Out-of-order arrival of datagrams.
    - Corruption of bits in datagrams.
- **TCP's Role**:
  - Creates a reliable data transfer service on top of IP’s unreliable best-effort service.
  - Ensures:
    - Uncorrupted data stream.
    - No gaps, duplication, or out-of-order delivery.
    - Exact byte stream delivery as sent by the end system.

TCP Timer Management

- **Single Retransmission Timer**:
  - Uses one timer for multiple segments instead of individual timers.
  - Reduces overhead in timer management.

TCP Sender Process

1. **Events**:
   - **Data Received from Application**:

    - Create TCP segment with sequence number.
    - Start timer if not currently running.
- **Timer Timeout**:
    - Retransmit segment with the smallest sequence number.
    - Restart timer.
- **ACK Received**:
    - Update SendBase if ACK value is greater.
    - Restart timer if there are unacknowledged segments.

Pseudocode for TCP Sender

```
NextSeqNum = InitialSeqNumber
SendBase = InitialSeqNumber

loop (forever) {
    switch(event) {
        event: data received from application above
            create TCP segment with sequence number NextSeq
Num
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum = NextSeqNum + length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with sm
allest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
```

```
                SendBase = y
                if (there are currently any not-yet-acknowl
edged segments)
                    start timer
            }
            break;
    }
}
```

Scenarios

Scenario 1: Segment Loss

- Host A sends segment with sequence number 92 (8 bytes).
- ACK from Host B is lost.
- Host A retransmits the segment after timeout; Host B discards the duplicate.

Scenario 2: Multiple Segments

- Host A sends two segments (92 and 100).
- Both segments arrive intact at Host B, but ACKs are lost.
- Host A retransmits the first segment after timeout; second segment remains intact if its ACK arrives before the next timeout.

Scenario 3: Out-of-Order ACK

- Host A sends two segments; ACK for the first is lost, but the second ACK arrives before timeout.
- Host A does not resend segments since ACK indicates all data up to byte 119 was received.

Modifications to TCP

Timeout Interval Doubling

- On timeout, TCP doubles the timeout interval for retransmissions.
- Helps manage congestion by increasing intervals between retransmissions.

Fast Retransmit

- Detects packet loss before timeout via duplicate ACKs.
- Receives three duplicate ACKs indicates a missing segment; fast retransmits the segment without waiting for the timer to expire.

TCP Receiver ACK Generation Policy

Event	TCP Receiver Action
In-order segment with expected seq. number	Delayed ACK; wait up to 500 msec.
In-order segment with one waiting ACK	Send cumulative ACK immediately.
Out-of-order segment with higher seq. number	Send duplicate ACK indicating next expected byte.
Segment fills in gap in received data	Send ACK immediately.

Error Recovery Mechanism

- **TCP vs. GBN**:
  - TCP acknowledges segments cumulatively; correctly received but out-of-order segments are buffered.
  - GBN would retransmit multiple packets upon loss; TCP retransmits only the lost segment.
- **Selective Acknowledgment (SACK)**:
  - Allows TCP to acknowledge out-of-order segments selectively.
  - Enhances TCP's error recovery mechanism, making it resemble a hybrid of GBN and selective repeat protocols.

3.5.5 Flow Control

Overview

- TCP connections have a receive buffer for storing incoming bytes.
- Data is placed in the receive buffer, which may not be read immediately by the application.
- **Flow Control**: A mechanism to prevent the sender from overwhelming the receiver's buffer.

Key Concepts

- **Receive Buffer**: Memory allocated for incoming data.
- **Flow Control**: Matches the sender's sending rate with the receiver's reading rate.
- **Congestion Control**: Manages sender throttling due to network congestion.

TCP Flow Control Mechanism

- The sender maintains a variable called the **receive window (rwnd)**, which indicates the free buffer space available at the receiver.
- TCP is **full-duplex**, meaning both sender and receiver maintain distinct receive windows.

Definitions

- **LastByteRead**: The last byte number read by the receiving application.
- **LastByteRcvd**: The last byte number received and placed in the buffer.
- **Receive Window (rwnd)**:

$$rwnd = RcvBuffer - (LastByteRcvd - LastByteRead)$$

Functionality

- Host B informs Host A of the spare room (rwnd) by including it in the segment sent.
- Initially, $rwnd = RcvBuffer$

Sender Variables

- **LastByteSent**: The last byte number sent by Host A.
- **LastByteAcked**: The last byte number acknowledged by Host B.
- Condition to prevent overflow:

$$LastByteSent - LastByteAcked \leq rwnd$$

Technical Considerations

- If Host B's buffer is full `rwnd = 0`, Host A may not be informed about available space.
- To address this, Host A continues sending segments with one data byte to ensure that acknowledgments are sent back.

UDP Flow Control

- **UDP does not provide flow control**:
  - Segments may overflow the buffer if the application does not read them quickly enough, leading to dropped segments.

Summary

- TCP ensures reliable flow control to match sending and reading rates, preventing buffer overflow. In contrast, UDP lacks this mechanism, risking data loss due to overflow.

3.5.6 TCP Connection Management

Overview

- **Importance**: Understanding TCP connection management is crucial for perceived delays and security vulnerabilities, such as SYN flood attacks.

Connection Establishment

- **Client Initiation**:

    - The client process requests a connection to the server.
- **Three-Step Process**:
    1. **Step 1** (SYN Segment):
        - Client sends a TCP segment with the SYN flag set to 1.
        - A random initial sequence number (`client_isn`) is included.
    2. **Step 2** (SYNACK Segment):
        - Server receives the SYN, allocates resources, and responds with a SYNACK segment:
            - SYN flag set to 1.
            - Acknowledgment set to `client_isn + 1`.
            - Server's initial sequence number (`server_isn`) is included.
    3. **Step 3** (ACK Segment):
        - Client acknowledges the SYNACK by sending a segment with the acknowledgment set to `server_isn + 1`.
        - SYN flag set to 0, indicating the connection is established.
- **Result**: After these three steps, data segments can be exchanged, and the SYN flag is set to 0.

Connection Teardown

- **Closing the Connection**:
    - Either process can initiate the connection closure:
        - Client sends a segment with the FIN flag set to 1.
        - Server acknowledges, then sends its own FIN segment.
        - Client acknowledges the server’s FIN.
- **Resource Deallocation**: Buffers and variables in both hosts are released after the connection is closed.

TCP States

- **State Transitions**:
  - **CLOSED**: Initial state before connection.
  - **SYN_SENT**: After sending SYN, waiting for SYNACK.
  - **ESTABLISHED**: Connection is active; can send and receive data.
  - **FIN_WAIT_1**: Waiting for acknowledgment of FIN.
  - **FIN_WAIT_2**: Waiting for server's FIN.
  - **TIME_WAIT**: Allows for resending of final acknowledgment; lasts a few minutes.

Handling Invalid Connections

- **Reset Segment**:
  - If a host receives a TCP segment with unmatched port numbers, it sends a reset (RST) segment to the source, indicating no socket exists for that segment.

Port Scanning with Nmap

- **Functionality**:
  - Sends a TCP SYN segment to a target port.
  - Possible outcomes:
    - **Open**: Receives SYNACK, indicating an application is running.
    - **Closed**: Receives RST, indicating no application at that port.
    - **Blocked**: Receives nothing, indicating potential firewall blockage.

SYN Flood Attack

- **Description**:
  - Attackers send numerous SYN segments, exhausting server resources.
- **Defense Mechanism (SYN Cookies)**:
  - The server creates a hashed initial TCP sequence number (cookie) without allocating resources for the SYN.

- Upon receiving an ACK, the server verifies it using the hash function.
- If valid, it establishes a full connection; otherwise, no harm is done.

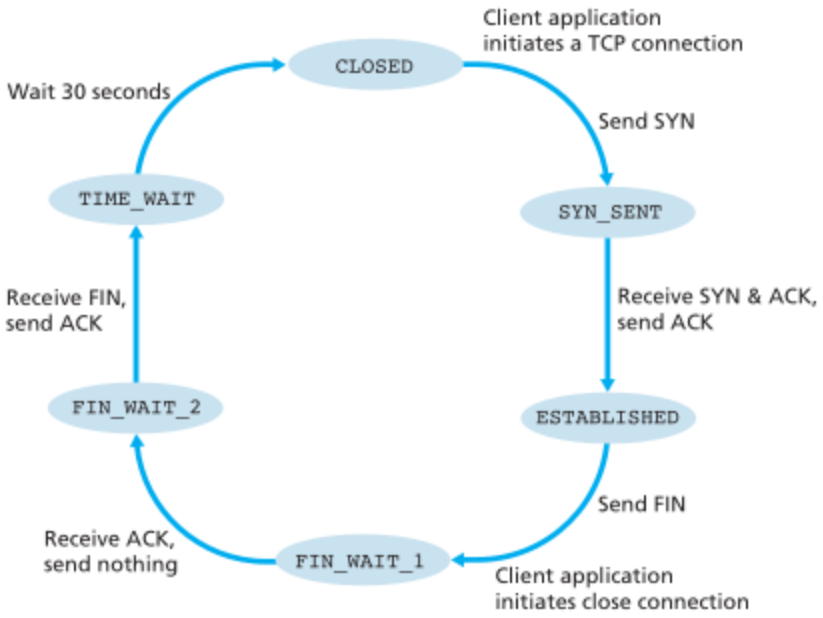


Figure 3.41 ♦ A typical sequence of TCP states visited by a client TCP

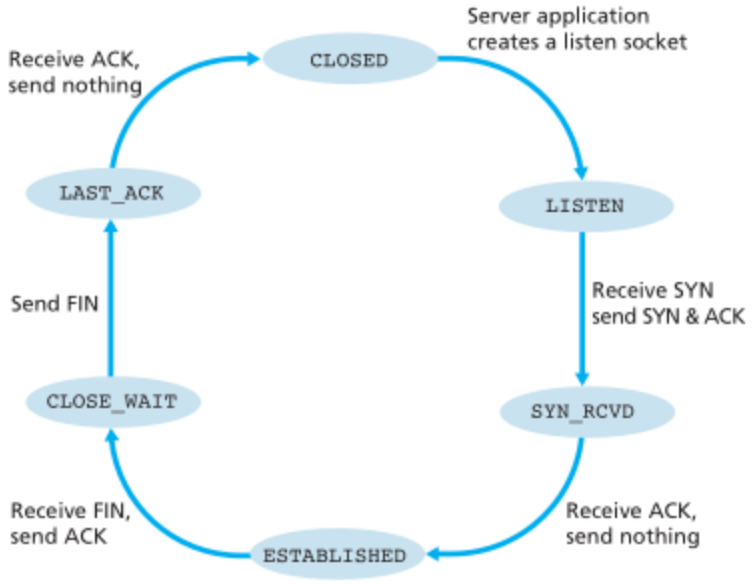


Figure 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

▼ 3.6 Principles of Congestion Control

- **Overview**
  - Focus on congestion control as a critical networking issue.
  - Congestion results from too many sources attempting to send data at high rates.
  - Congestion control mechanisms are necessary to throttle senders during congestion.

3.6.1 Causes and Costs of Congestion

General Study of Congestion Control

- Importance of understanding:
  - Why congestion is detrimental.
  - Its impact on upper-layer application performance.
  - Various approaches to avoid or react to congestion.

Scenario 1: Two Senders, Infinite Buffers

- **Setup**
  - Two hosts (A and B) sharing a single hop.
  - Both send data at a rate of λ_in bytes/sec.
  - Router with infinite buffering.
- **Performance Insights**
  - Throughput behavior:
    - Sending rate (0 to R/2): Throughput = Sending Rate.
    - Sending rate (> R/2): Throughput = R/2.

- **Cost of Congestion**
  - Large queuing delays as sending rate approaches link capacity.

Scenario 2: Two Senders, Finite Buffers

- **Setup**
  - Modified from Scenario 1 with finite buffering.
  - Packets dropped when the buffer is full.
  - Reliable connections with retransmissions.
- **Performance Insights**
  - Impact of retransmission strategies:
    - Ideal case: Throughput equals λ_in with no packet loss.
    - More realistic case: Retransmission only for confirmed losses.
    - Possible premature retransmissions leading to wasted bandwidth.
  - **Cost of Congestion**
    - Increased retransmissions due to buffer overflow.
    - Wasted bandwidth forwarding unnecessary packets.

Scenario 3: Four Senders, Finite Buffers, Multihop Paths

- **Setup**
  - Four hosts transmitting over multihop paths with overlapping connections.
  - All hosts share the same λ_in and router link capacity R.
- **Performance Insights**
  - Low traffic: Throughput approximately equals offered load.
  - High traffic: Arrival rates exceed link capacities, causing competition for buffer space.
  - **Cost of Congestion**

    - Wasted work when packets are dropped.
    - First-hop router capacity wasted on packets that are dropped at the second-hop router.

3.6.2 Approaches to Congestion Control

Overview

- Two broad approaches to congestion control:
    - **End-to-end Congestion Control**
    - **Network-assisted Congestion Control**

1. End-to-End Congestion Control

- **Definition**:
    - The network layer provides no explicit support for congestion control.
    - End systems infer congestion from observed behaviors (e.g., packet loss, delay).
- **Example**:
    - TCP uses this approach; segment loss indicates network congestion.

2. Network-Assisted Congestion Control

- **Definition**:
    - Network-layer components (routers) provide explicit feedback to the sender about network congestion.
- **Feedback Mechanisms**:
    - **Direct Feedback**: Sent from a router to the sender (e.g., choke packets).
    - **Indirect Feedback**: A router marks/updates a field in a packet to indicate congestion, and the receiver notifies the sender (requires one round-trip time).

3.6.3 Network-Assisted Congestion-Control Example: ATM ABR Congestion Control

Overview

- **Purpose**: Illustrates a protocol that differs from TCP's congestion control.
- **Key Concept**: ATM uses a virtual-circuit (VC) oriented approach to packet switching.

ATM ABR Characteristics

- **Elastic Data Transfer**:
  - Adapts to available bandwidth; throttles during congestion.

Congestion Control Mechanism

- **Resource Management (RM) Cells**:
  - Used to convey congestion-related information between hosts and switches.
  - Can be generated by either the destination or switches.

Rate-Based Approach

- **Mechanisms for Signaling Congestion**:
  - **EFCI Bit**:
    - Indicates congestion in data cells.
    - Destination checks the EFCI bit and notifies the sender via RM cells.
  - **CI and NI Bits**:
    - CI bit indicates severe congestion, NI bit indicates mild congestion.
    - RM cells sent back to the sender retain these bits.
  - **Explicit Rate (ER) Setting**:
    - Contains a field that reflects the minimum supportable rate across the path.

Rate Adjustment

- The source adjusts its sending rate based on the CI, NI, and ER values in returned RM cells.

▼ 3.7 TCP Congestion Control

Overview

- TCP provides a reliable transport service and includes a congestion-control mechanism.
- TCP uses **end-to-end** congestion control since the IP layer does not provide explicit feedback on network congestion.

Key Questions

1. How does a TCP sender limit the rate of traffic sent?
2. How does a TCP sender perceive congestion?
3. What algorithm adjusts the send rate based on perceived congestion?

Rate Limiting

- Each TCP sender tracks a variable called **congestion window (cwnd)**.
- The amount of unacknowledged data must not exceed the minimum of `cwnd` and `rwnd`:
  - **Equation**: LastByteSent − LastByteAcked ≤ min{cwnd, rwnd}
- **Assumptions**:
  - Ignore receive-window constraint (large receive buffer).
  - Sender always has data to send.

Send Rate Calculation

- In a congestion-free network, the send rate is approximately:

- **Rate**: cwnd / RTT bytes/sec

Perceiving Congestion

- A **loss event** occurs due to:
  - Timeout
  - Receipt of three duplicate ACKs
- Loss indicates congestion, prompting the sender to react.

Acknowledgments and Self-Clocking

- Acknowledgments for unacknowledged segments lead to:
  - Increase in `cwnd` (congestion window size).
- TCP is **self-clocking**: ACKs trigger increases in the congestion window.

Principles of TCP Congestion Control

- **Lost Segment**: Implies congestion; sender decreases its send rate.
- **Acknowledged Segment**: Indicates successful delivery; sender increases its send rate.
- **Bandwidth Probing**: Increase rate upon receiving ACKs until a loss event occurs; then back off and probe again.

TCP Congestion-Control Algorithm

- Components:
  1. **Slow Start**
  2. **Congestion Avoidance**
  3. **Fast Recovery** (recommended but not mandatory)

Slow Start

- **Initialization**: `cwnd` starts at 1 MSS (Maximum Segment Size).

- **Exponential Growth**:
  - Increases by 1 MSS for each acknowledged segment.
  - Results in doubling the sending rate every RTT.

Conditions to End Slow Start

1. **Loss Event**: Set `cwnd` to 1 and restart slow start.
2. **Slow Start Threshold (ssthresh)**: If `cwnd` equals or exceeds `ssthresh`, transition to congestion avoidance mode.
3. **Three Duplicate ACKs**: Trigger fast retransmit and enter fast recovery state.

Congestion Avoidance

- **Entry to Congestion-Avoidance State**
  - `cwnd` (congestion window) is set to approximately half its last value when congestion was last encountered.
  - Increases `cwnd` by **1 MSS** (Maximum Segment Size) every RTT (Round-Trip Time).
- **Mechanism of Increase**
  - Commonly, the TCP sender increases `cwnd` by **MSS/cwnd** for each new acknowledgment received.
  - Example:
    - If `MSS = 1,460 bytes` and `cwnd = 14,600 bytes`:
      - Sends 10 segments within an RTT.
      - Each ACK increases `cwnd` by **1/10 MSS**.
      - Total increase becomes **1 MSS** after all 10 ACKs are received.
- **End of Linear Increase**
  - Linear increase of `cwnd` ends upon a timeout or triple duplicate ACK event.
  - In case of a loss:

- **Timeout**:
  - Set `cwnd` to **1 MSS**.
  - Update `ssthresh` (slow-start threshold) to half the value of `cwnd`.
- **Triple Duplicate ACKs**:
  - Halve `cwnd` and add **3 MSS**.
  - Update `ssthresh` to half the value of `cwnd`.

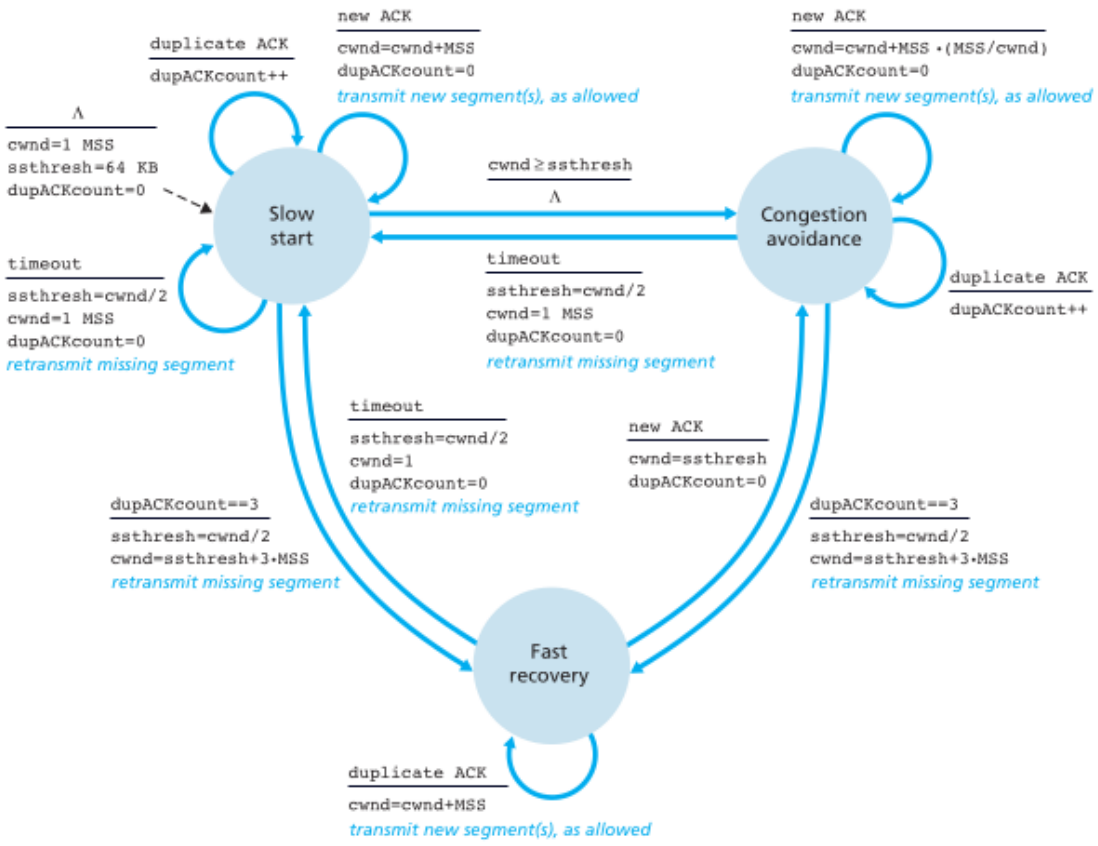
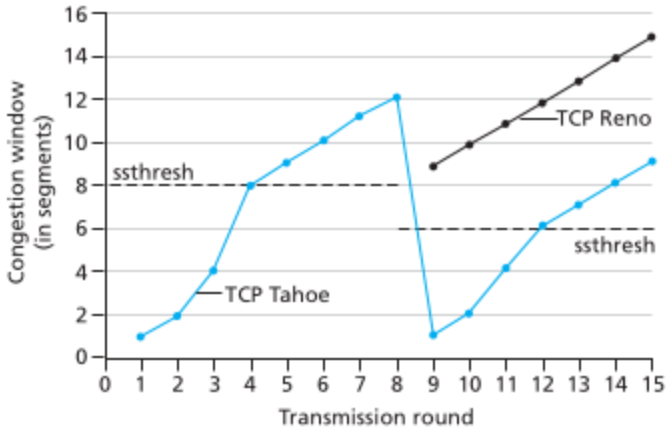


Figure 3.52 ♦ FSM description of TCP congestion control

Fast Recovery

- **Mechanism of Fast Recovery**
  - Increase `cwnd` by **1 MSS** for each duplicate ACK received.

    - Transition to congestion-avoidance state when an ACK for the missing segment arrives.
- **Transition on Timeout**
    - If a timeout occurs:
        - Transition to slow-start state.
        - Set `cwnd` to **1 MSS**.
        - Update `ssthresh` to half the value of `cwnd` at the loss event.
- **TCP Variants**
    - **TCP Tahoe**:
        - Cuts `cwnd` to **1 MSS** after any loss event (timeout or triple duplicate ACK).
    - **TCP Reno**:
        - Incorporates fast recovery mechanism.

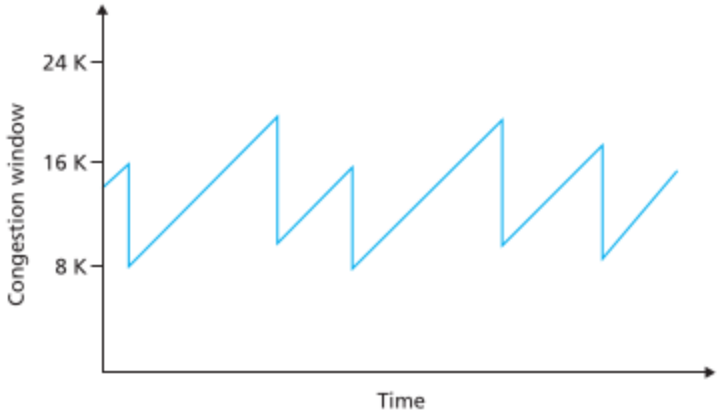


3.53 ♦ Evolution of TCP's congestion window (Tahoe and Reno)

TCP Congestion Control: Overview

- **Congestion Control Behavior**
    - Linear increase of `cwnd` by **1 MSS** per RTT.
    - Halving of `cwnd` on triple duplicate ACK event.

    - This approach is termed **Additive Increase, Multiplicative Decrease (AIMD)**.
- **Performance Characteristics**
    - Results in a "saw-tooth" behavior.
    - TCP probes for bandwidth by increasing `cwnd` until loss occurs, then reduces `cwnd`.
- **TCP Variants and Developments**
    - Many implementations use **Reno** algorithm.
    - Variations exist, including **TCP Vegas**, which aims to detect congestion before packet loss occurs.



e 3.54 ♦ Additive-increase, multiplicative-decrease congestion control

Macroscopic Description of TCP Throughput

- **Average Throughput Analysis**
    - Average throughput during steady-state (ignoring slow-start):
        - **Average Throughput = 0.75 × W / RTT**
            - `W`: Congestion window size at the time of loss.

TCP Over High-Bandwidth Paths

- **Need for Evolution**
  - TCP must evolve to accommodate high-speed applications (e.g., cloud computing).
  - Example scenario:
    - For 10 Gbps throughput with a **100 ms RTT** and **1,500-byte segments**:
      - Requires a congestion window size of **83,333 segments**.
      - Investigates new TCP versions tailored for high-speed environments.

Relationship to Loss Rate

- Theoretical model for TCP throughput:
  - **Average Throughput = 1.22 × MSS / RTT × (L)^1/2**
    - `L`: Loss rate.
    - Achieving high throughput requires a very low loss rate (e.g., for 10 Gbps, `L` should be less than **2 × 10^-10**).

3.7.1 Fairness in TCP Connections

Overview

- Fairness in TCP connections refers to the equal distribution of bandwidth among multiple connections sharing a bottleneck link.

Key Points

- **Definition of Fairness**:
  - A congestion-control mechanism is fair if each connection receives approximately $\frac{R}{K}$ bandwidth, where `R` is the link rate and `K` is the number of connections.

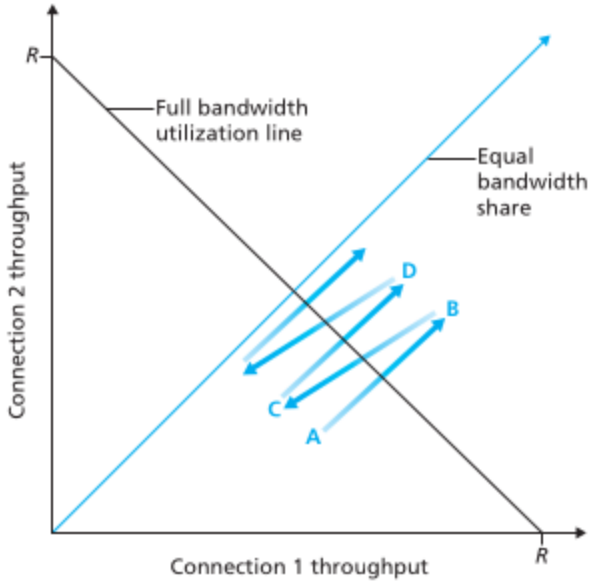
TCP's AIMD Algorithm

- **Behavior in Bottleneck Links**:

- If TCP connections share a bottleneck link with rate `R`, the aim is for both connections to achieve throughput close to $\frac{R}{2}$.
- **Throughput Dynamics**:
  - As window sizes increase without loss, throughputs increase equally.
  - Upon packet loss, TCP halves the window size, leading to a fluctuating throughput along the equal bandwidth share line.

Factors Affecting Fairness

- **RTT Variations**:
  - Connections with smaller Round-Trip Times (RTTs) can occupy bandwidth more quickly than those with larger RTTs.
  - In practice, multiple TCP connections can result in unequal bandwidth allocation due to RTT differences.



3.56 ♦ Throughput realized by TCP connections 1 and 2

Fairness and UDP

- **UDP Characteristics**:

- Multimedia applications often use UDP instead of TCP to avoid transmission rate throttling during congestion.
- UDP allows for consistent data transmission even if packet loss occurs, potentially leading to unfair bandwidth distribution compared to TCP connections.

Research Areas

- **Development of Congestion-Control Mechanisms**:
  - Research is ongoing to create mechanisms that ensure UDP traffic does not negatively impact overall network throughput.

Fairness and Parallel TCP Connections

- **Multiple Connections Issue**:
  - Applications using multiple parallel TCP connections can gain a larger share of the bandwidth.
  - Example:
    - In a congested link of rate `R` :
      - Each of 10 single connections shares $\frac{R}{10}$.
      - A new application with 11 parallel connections can secure more than $\frac{R}{2}$.

Conclusion

- Fairness in TCP connections remains a challenge due to variations in RTT, the behavior of UDP traffic, and the common practice of using multiple parallel connections. Addressing these issues is crucial for optimizing network performance.
