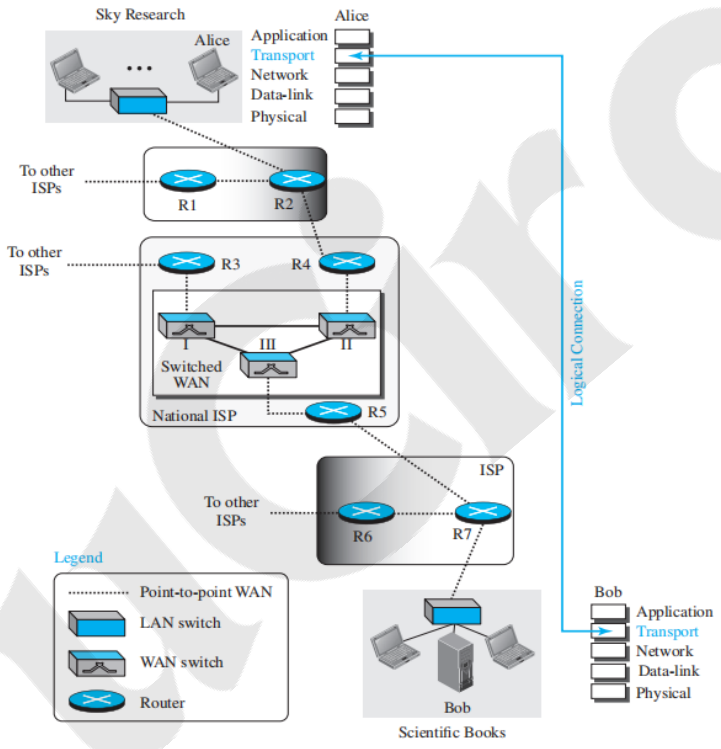
Module 4

Introduction to Transport layer

INTRODUCTION

The transport layer is located between the application layer and the network layer. It provides a process-to-process communication between two application layers, one at the local host and the other at the remote host. Communication is provided using a logical connection, which means that the two application layers, which can be located in different parts of the globe, assume that there is an imaginary direct connection through which they can send and receive messages.

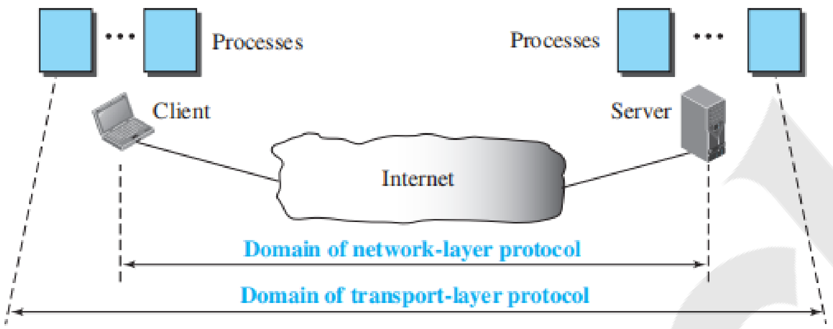


Transport-Layer Services

The transport layer is responsible for providing services to the application layer; it receives services from the network layer.

The first duty of a transport-layer protocol is to provide process-to-process communication. A process is an application-layer entity (running program) that uses the services of the transport layer.

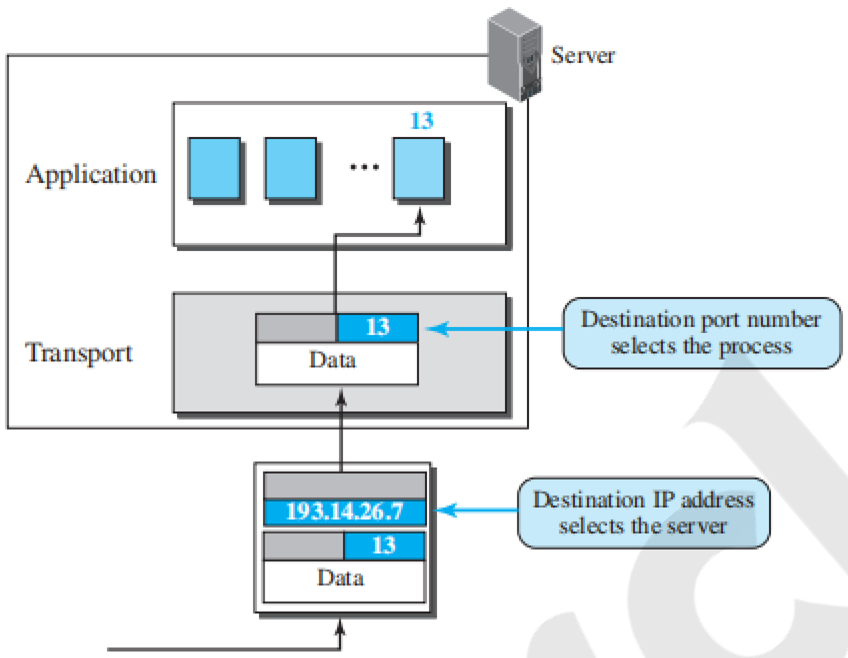
The network layer is responsible for communication at the computer level (host-to-host communication). A network-layer protocol can deliver the message only to the destination computer. However, this is an incomplete delivery. The message still needs to be handed to the correct process. This is where a transport-layer protocol takes over. A transport-layer protocol is responsible for delivery of the message to the appropriate process.


Processes
Processes
Client
Server
Internet
Domain of network-layer protocol
Domain of transport-layer protocol

Addressing: Port Numbers

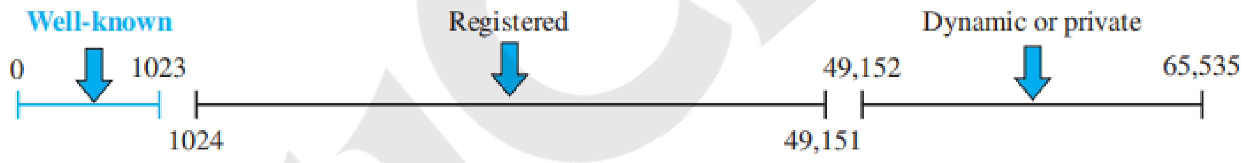
Although there are a few ways to achieve process-to-process communication, the most common is through the **client-server paradigm**.

- A process on the local host, called a *client*, needs services from a process usually on the remote host, called a *server*.
- The operating systems today support both multiuser and multiprogramming environments. A remote computer can run multiple server programs at once, just like different local computers can each run one or more client programs at the same time.
- For communication, we must define the local host, local process, remote host, and remote process. The local host and the remote host are defined using IP addresses. To define the processes, we need **port numbers** (0-65,535 (16 bits)).
- The client program defines itself with a port number, called the **ephemeral port number**. The word ephemeral means "short-lived" and is used because the life of a client is normally short.
- The server process must also define itself with a port number. TCP/IP has decided to use universal port numbers for servers; these are called well-known port numbers.


Server
Application
13
Transport
13
Data
Destination port number selects the process
193.14.26.7
13
Data
Destination IP address selects the server

ICANN Ranges

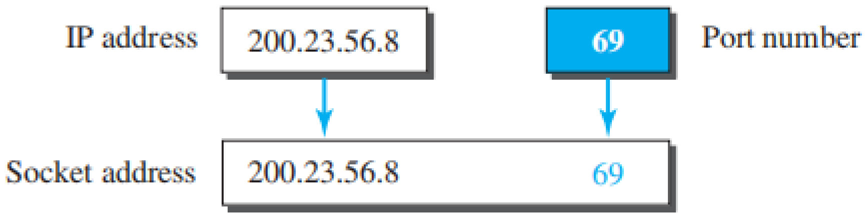
ICANN has divided the port numbers into three ranges: well-known, registered, and dynamic (or private).



1. Well-known ports. The ports ranging from 0 to 1023 are assigned and controlled by ICANN. These are the well-known ports.
2. Registered ports. The ports ranging from 1024 to 49,151 are not assigned or controlled by ICANN. They can only be registered with ICANN to prevent duplication.
3. Dynamic ports. The ports ranging from 49,152 to 65,535 are neither controlled nor registered. They can be used as temporary or private port numbers.

Socket Addresses

A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a **socket address**. The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely.

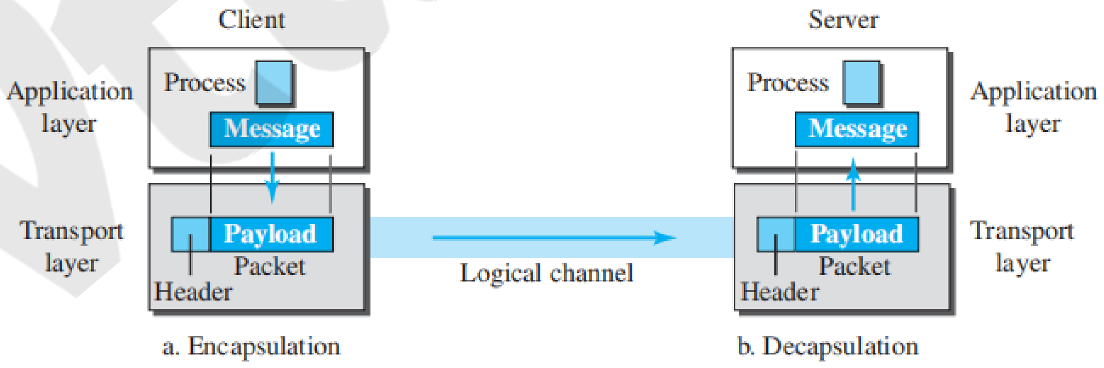


To use the services of the transport layer in the Internet, we need a pair of socket addresses: the client socket address and the server socket address. These four pieces of information are part of the network-layer packet header and the transport-layer packet header. The first header contains the IP addresses; the second header contains the port numbers.

Encapsulation and Decapsulation

Encapsulation happens on the sender's side: When a program wants to send a message, it gives the message to the transport layer, along with the socket addresses and other necessary info. The transport layer then adds a header to the message, creating a packet. Depending on the transport protocol, this packet might be called a user datagram, segment, or just a packet.

Decapsulation happens on the receiver's side: When the packet reaches its destination, the transport layer removes the header and delivers the message to the application program. The sender's socket address is also passed along so the program can respond if needed.



Multiplexing and Demultiplexing

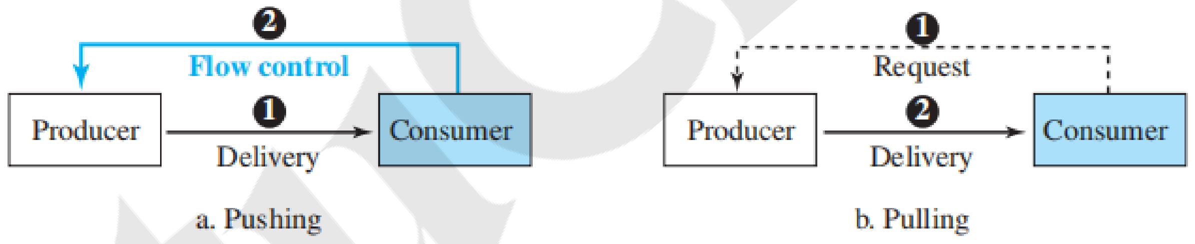
Whenever an entity accepts items from more than one source, this is referred to as ***multiplexing*** (many to one); whenever an entity delivers items to more than one source, this is referred to as ***demultiplexing*** (one to many). The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing.

Flow Control

Whenever an entity produces items and another entity consumes them, there should be a balance between production and consumption rates. If the items are produced faster than they can be consumed, the consumer can be overwhelmed and may need to discard some items. If the items are produced more slowly than they can be consumed, the consumer must wait, and the system becomes less efficient. Flow control is related to the first issue. We need to prevent losing the data items at the consumer site.

Pushing or Pulling

Delivery of items from a producer to a consumer can occur in one of two ways: *pushing* or *pulling*. If the sender delivers items whenever they are produced without a prior request from the consumer the delivery is referred to as *pushing*. If the producer delivers the items after the consumer has requested them, the delivery is referred to as *pulling*.

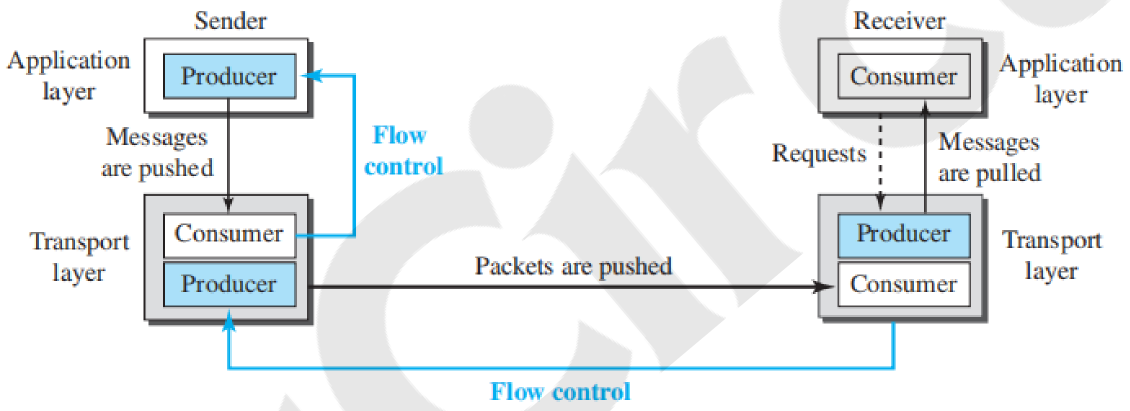


When the producer *pushes* the items, the consumer may be overwhelmed and there is a need for flow control, in the opposite direction, to prevent discarding of the items.

Flow Control at Transport Layer

- In communication at the transport layer, we are dealing with four entities: sender process, sender transport layer, receiver transport layer, and receiver process.
- The sending process at the application layer is only a producer. It produces message chunks and pushes them to the transport layer.

- ➔ The sending transport layer has a double role: it is both a consumer and a producer. It consumes the messages pushed by the producer. It encapsulates the messages in packets and pushes them to the receiving transport layer.
- ➔ The receiving transport layer also has a double role: it is the consumer for the packets received from the sender and the producer that decapsulates the messages and delivers them to the application layer.
- ➔ The last delivery, however, is normally a pulling delivery; the transport layer waits until the application-layer process asks for messages.
- ➔ Figure shows that we need at least two cases of flow control: from the sending transport layer to the sending application layer and from the receiving transport layer to the sending transport layer.



Buffers

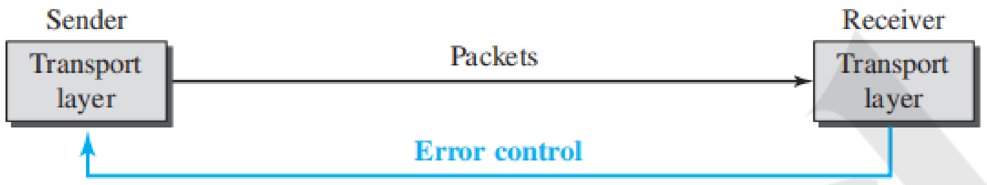
A buffer is a set of memory locations that can hold packets at the sender and receiver. The flow control communication can occur by sending signals from the consumer to the producer. When the buffer of the sending transport layer is full, it informs the application layer to stop passing chunks of messages; when there are some vacancies, it informs the application layer that it can pass message chunks again.

When the buffer of the receiving transport layer is full, it informs the sending transport layer to stop sending packets. When there are some vacancies, it informs the sending transport layer that it can send packets again.

Error Control

In the Internet, since the underlying network layer (IP) is unreliable, we need to make the transport layer reliable if the application requires reliability. Reliability can be achieved to add error control services to the transport layer. Error control at the transport layer is responsible for

1. Detecting and discarding corrupted packets.
2. Keeping track of lost and discarded packets and resending them.
3. Recognizing duplicate packets and discarding them.
4. Buffering out-of-order packets until the missing packets arrive.



Sequence Numbers

- ➔ Error control requires that the sending transport layer knows which packet is to be resent and the receiving transport layer knows which packet is a duplicate, or which packet has arrived out of order. This can be done if the packets are numbered.
- ➔ We can add a field to the transport-layer packet to hold the **sequence number** of the packet. When a packet is corrupted or lost, the receiving transport layer can somehow inform the sending transport layer to resend that packet using the sequence number.
- ➔ The receiving transport layer can also detect duplicate packets if two received packets have the same sequence number.
- ➔ The out-of-order packets can be recognized by observing gaps in the sequence numbers. Packets are numbered sequentially.
- ➔ Packets are numbered sequentially. However, because we need to include the sequence number of each packet in the header, we need to set a limit.
- ➔ If the header of the packet allows *m* bits for the sequence number, the sequence numbers range from 0 to $2^m - 1$.
- ➔ For example, if *m* is 4, the only sequence numbers are 0 through 15, inclusive. However, we can wrap around the sequence. So the sequence numbers in this case are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

Acknowledgment

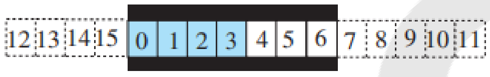
The receiver side can send an acknowledgment (ACK) for each of a collection of packets that have arrived safe and sound. The receiver can simply discard the corrupted packets. The sender can detect lost packets if it uses a timer. When a packet is sent, the sender starts a timer. If an ACK does not arrive before the timer expires, the sender resends the packet. Duplicate packets can be silently discarded by the receiver. Out-of-order packets can be either discarded (to be treated as lost packets by the sender), or stored until the missing one arrives.

Combination of Flow and Error Control

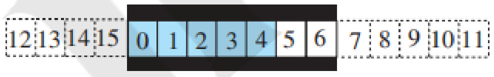
The flow control requires the use of two buffers, one at the sender site and the other at the receiver site. The error control requires the use of sequence and acknowledgment numbers by both sides. These two requirements can be combined if we use two numbered buffers, one at the sender, one at the receiver. At the sender, when a packet is prepared to be sent, we use the number of the next free location, *x*, in the buffer as the sequence number of the packet. When the packet is sent, a copy is stored at memory location *x*, awaiting the acknowledgment from the other end. When an acknowledgment related to a sent packet arrives, the packet is purged and the memory location becomes free. At the receiver, when a packet with sequence number *y* arrives, it is stored at the memory location *y* until the application layer is ready to receive it. An acknowledgment can be sent to announce the arrival of packet *y*.

Sliding Window

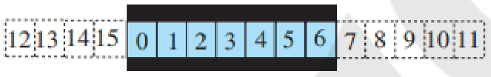
The sliding window is a technique for sending multiple frames at a time. It controls the data packets between the two devices where reliable and gradual delivery of data frames is needed. It is also used in TCP (Transmission Control Protocol).
In this technique, each frame has sent from the sequence number. The sequence numbers are used to find the missing data in the receiver end. The purpose of the sliding window technique is to avoid duplicate data, so it uses the sequence number.



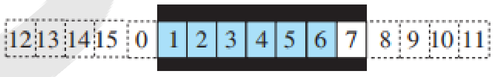
a. Four packets have been sent.



b. Five packets have been sent.



c. Seven packets have been sent;
window is full.



d. Packet 0 has been acknowledged;
window slides.

Congestion Control

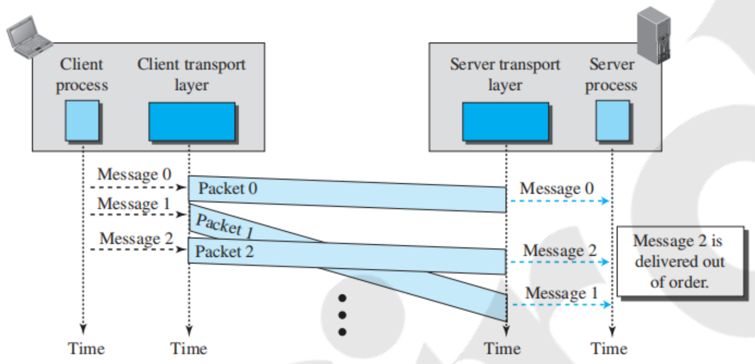
Congestion in a packet-switched network, like the Internet, happens when the load—number of packets sent—exceeds the network's capacity to handle them. Congestion control refers to methods used to prevent this by keeping the load below capacity. Similar to traffic jams on a freeway, congestion occurs in networks when there are delays, such as when routers and switches can't process packets as fast as they receive them. Routers have input and output queues, and when these queues get overloaded, congestion happens. This problem starts at the network layer but affects the transport layer, where protocols like TCP implement their own congestion control if the network layer doesn't handle it.

Connectionless and Connection-Oriented Protocols

A transport-layer protocol, like a network-layer protocol, can provide two types of services: connectionless and connection-oriented

Connectionless Service

In a **connectionless service**, the source application divides its message into chunks of data and sends them to the **transport layer**, which treats each chunk independently. There is no relationship between the chunks, so they may arrive out of order at the destination. For example, a client might send three chunks (0, 1, and 2), but due to delays, the server could receive them out of order (0, 2, 1), as shown in **Figure** . This could result in a garbled message. If one packet is lost, since there is no numbering or coordination between the transport layers, the receiving side won't know and will deliver incomplete data. This lack of **flow control**, **error control**, and **congestion control** makes the system inefficient.

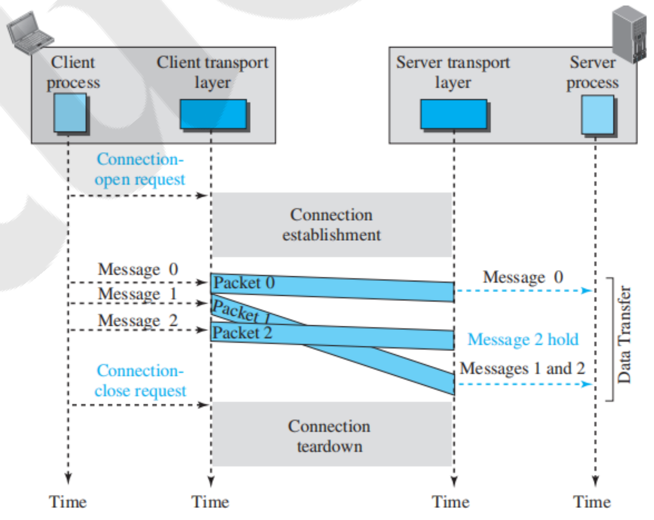


Connection-Oriented Service

In a connection-oriented service, the client and the server first need to establish a logical connection between themselves. The data exchange can only happen after the connection establishment. After data exchange, the connection needs to be torn down.

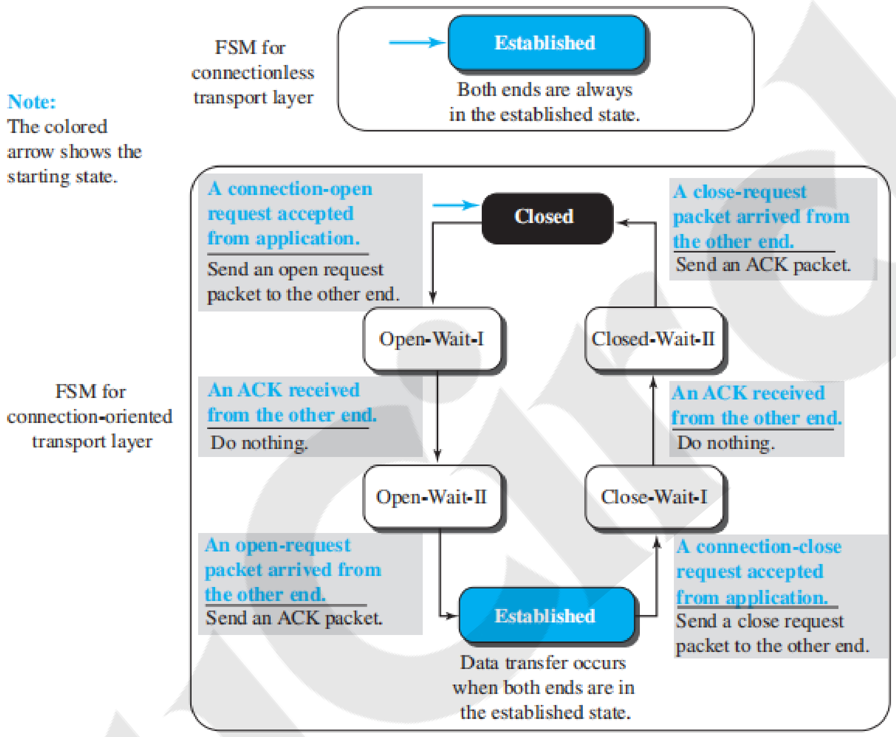
Figure shows the connection establishment, data-transfer, and tear-down phases in a connection-oriented service at the transport layer.

We can implement flow control, error control, and congestion control in a connection oriented protocol.



The behavior of a transport-layer protocol, both when it provides a connectionless and when it provides a connection-oriented protocol, can be better shown as a **finite state machine (FSM).** Using this tool, each transport layer (sender or receiver) is taught as a machine with a finite number of states.

Every FSM must have an initial state, which is where the machine starts when it turns on. In diagrams, rounded rectangles are used to represent states, colored text indicates events, and black text shows actions. A horizontal line or a slash separates the event from the action, and arrows depict the transition to the next state.


FSM for connectionless transport layer
Established
Both ends are always in the established state.
Note: The colored arrow shows the starting state.
FSM for connection-oriented transport layer
A connection-open request accepted from application.
Send an open request packet to the other end.
Closed
A close-request packet arrived from the other end.
Send an ACK packet.
Open-Wait-I
Closed-Wait-II
An ACK received from the other end.
Do nothing.
An ACK received from the other end.
Do nothing.
Open-Wait-II
Close-Wait-I
An open-request packet arrived from the other end.
Send an ACK packet.
Established
A connection-close request accepted from application.
Send a close request packet to the other end.
Data transfer occurs when both ends are in the established state.

In a **connectionless transport layer**, the FSM has only one state: the established state. The machines on both the client and server sides remain in the established state, always ready to send and receive transport-layer packets.

A **connection-oriented FSM** requires several states for **establishment** and **termination**:

- **Closed State**: The FSM starts here when there is no connection. It stays in this state until it receives an **open request** from the local process.
- The machine sends an **open request packet** to the remote transport layer and transitions to **open-wait-I**.
- When the acknowledgment is received, it moves to **open-wait-II**. At this point, a **unidirectional connection** is established.

- If a **bidirectional connection** is needed, it waits in this state until the other end also sends a connection request. Upon receiving it, the machine sends an acknowledgment and moves to the **established state**.

In both connectionless and connection-oriented transport layers, the established state represents the stage where data and acknowledgments can be exchanged between the two ends. The established state encompasses a set of data transfer states.
When **tearing down** a connection, the application layer sends a close request to the transport layer, which then sends a close request packet to the other end and transitions to close-wait-I.

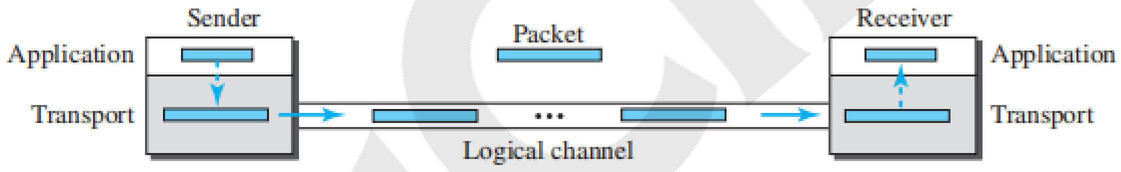
Once an acknowledgment is received, the machine moves to close-wait-II and waits for a close request from the other side. When the request arrives, the machine sends an acknowledgment and moves back to the closed state.

There are various versions of the connection-oriented FSM, where states can be condensed, expanded, or renamed depending on the protocol being used.

TRANSPORT-LAYER PROTOCOLS

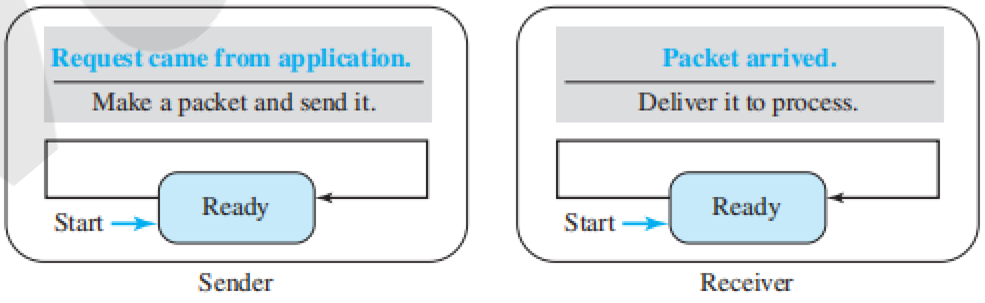
Simple Protocol

Our first protocol is a simple connectionless protocol with neither **flow nor error control**. We assume that the receiver can immediately handle any packet it receives.



The transport layer at the sender gets a message from its application layer, makes a packet out of it, and sends the packet. The transport layer at the receiver receives a packet from its network layer, extracts the message from the packet, and delivers the message to its application layer.

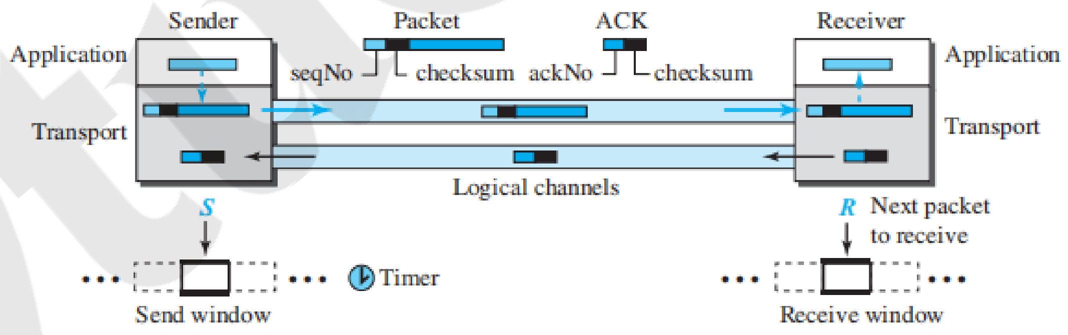
FSMs



- ➔ The sender site should not send a packet until its application layer has a message to send. The receiver site cannot deliver a message to its application layer until a packet arrives.
- ➔ We can show these requirements using two FSMs. Each FSM has only one state, the ***ready state***.
- ➔ The sending machine remains in the ready state until a request comes from the process in the application layer. When this event occurs, the sending machine encapsulates the message in a packet and sends it to the receiving machine.
- ➔ The receiving machine remains in the ready state until a packet arrives from the sending machine. When this event occurs, the receiving machine decapsulates the message out of the packet and delivers it to the process at the application layer.

Stop-and-Wait Protocol

- ➔ Stop-and-Wait is a connection-oriented protocol, which uses both **flow and error control**.
- ➔ Both the sender and the receiver use a **sliding window of size 1**. The sender sends one packet at a time and waits for an acknowledgment before sending the next one.
- ➔ To detect corrupted packets, we need to add a **checksum** to each data packet. When a packet arrives at the receiver site, it is checked. If its checksum is incorrect, the packet is corrupted and silently discarded.
- ➔ The silence of the receiver is a signal for the sender that a packet was either corrupted or lost. Every time the sender sends a packet, it starts a timer.
- ➔ If an acknowledgment arrives before the timer expires, the timer is stopped and the sender sends the next packet (if it has one to send).
- ➔ If the timer expires, the sender resends the previous packet, assuming that the packet was either lost or corrupted. This means that the sender needs to keep a copy of the packet until its acknowledgment arrives.



Sequence Numbers

To prevent duplicate packets, the protocol uses sequence numbers and acknowledgment numbers. A field is added to the packet header to hold the sequence number of that packet.

Acknowledgment Numbers

Since the sequence numbers must be suitable for both data packets and acknowledgments, we use this convention: The acknowledgment numbers always announce the sequence number of the *next packet expected* by the receiver.

FSMs

Since the protocol is a connection-oriented protocol, both ends should be in the *established* state before exchanging data packets. The states are actually nested in the *established* state.

Sender

The sender is initially in the ready state, but it can move between the ready and blocking state. The variable *S* is initialized to 0.

Ready state. When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application layer, the sender creates a packet with the sequence number set to *S*. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The sender then moves to the blocking state.

Blocking state. When the sender is in this state, three events can occur:
a. If an error-free ACK arrives with the ackNo related to the next packet to be sent, which means ackNo = (*S* + 1) modulo 2, then the timer is stopped. The window slides, *S* = (*S* + 1) modulo 2. Finally, the sender moves to the ready state.
b. If a corrupted ACK or an error-free ACK with the ackNo ≠ (S + 1) modulo 2 arrives, the ACK is discarded.
c. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

Receiver

The receiver is always in the *ready* state. Three events may occur:
a. If an error-free packet with seqNo = *R* arrives, the message in the packet is delivered to the application layer. The window then slides, *R* = (*R* + 1) modulo 2. Finally an ACK with ackNo = *R* is sent.
b. If an error-free packet with seqNo ≠ *R* arrives, the packet is discarded, but an ACK with ackNo = *R* is sent.
c. If a corrupted packet arrives, the packet is discarded.

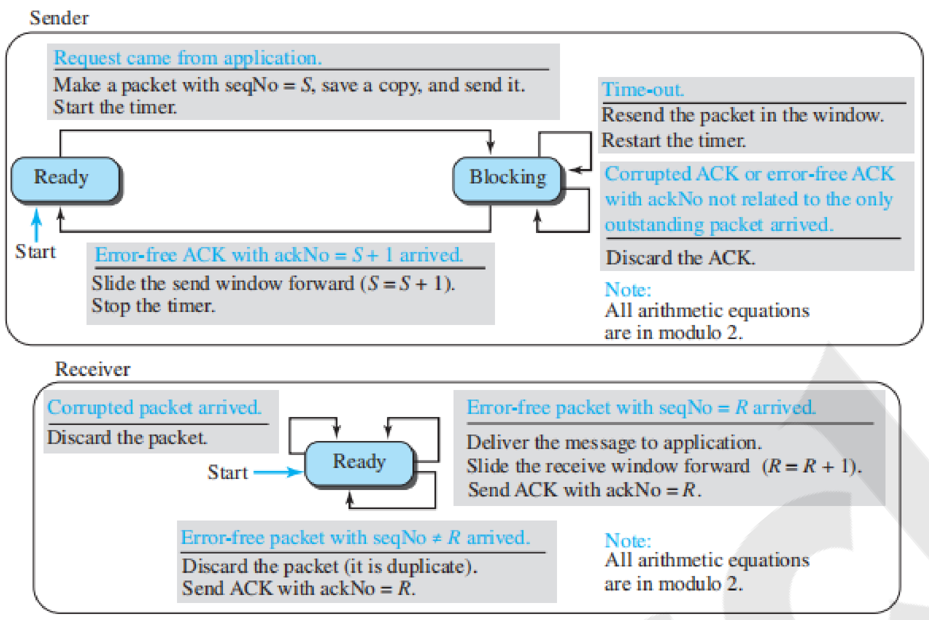
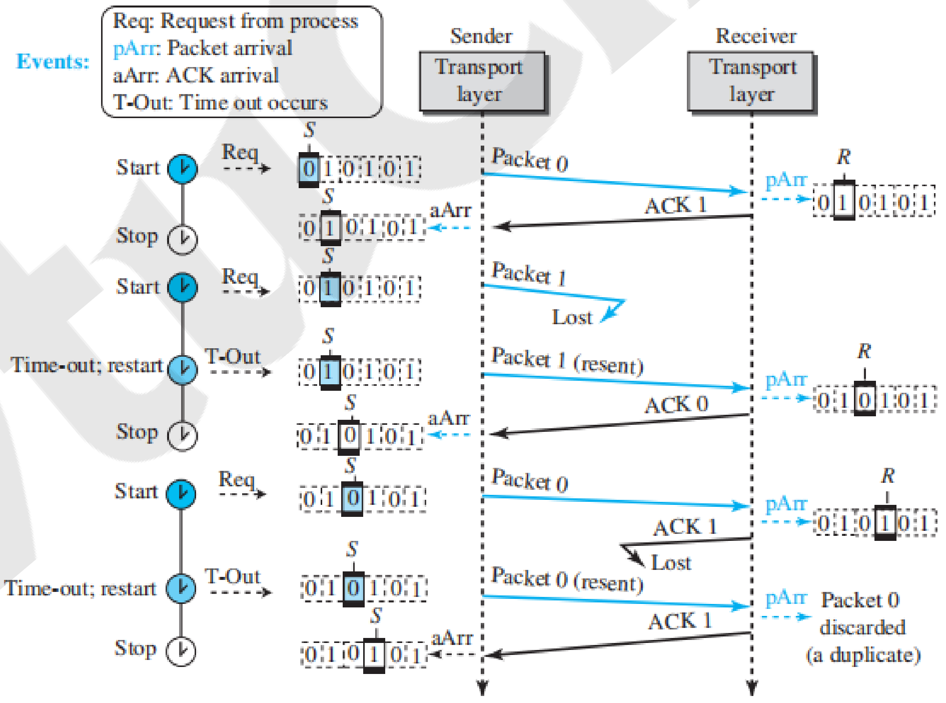
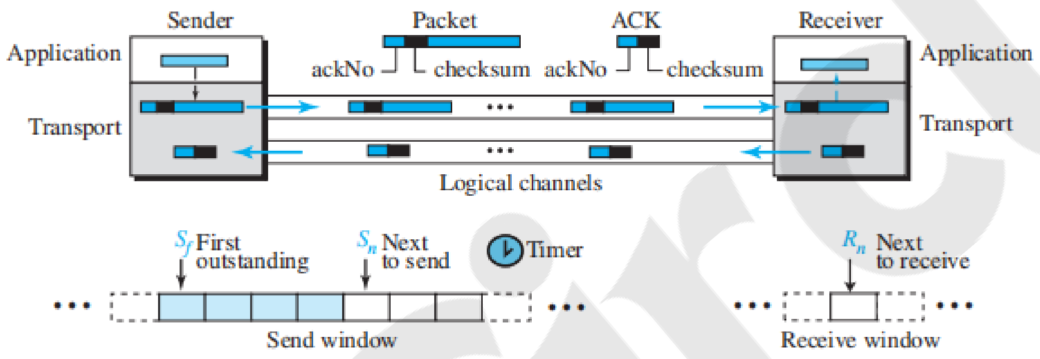

Sender
Request came from application.
Make a packet with seqNo = S, save a copy, and send it.
Start the timer.
Time-out.
Resend the packet in the window.
Restart the timer.
Ready
Blocking
Corrupted ACK or error-free ACK with ackNo not related to the only outstanding packet arrived.
Discard the ACK.
Start
Error-free ACK with ackNo = S + 1 arrived.
Slide the send window forward (S = S + 1).
Stop the timer.
Note:
All arithmetic equations are in modulo 2.
Receiver
Corrupted packet arrived.
Discard the packet.
Error-free packet with seqNo = R arrived.
Deliver the message to application.
Slide the receive window forward (R = R + 1).
Send ACK with ackNo = R.
Start
Ready
Error-free packet with seqNo ≠ R arrived.
Discard the packet (it is duplicate).
Send ACK with ackNo = R.
Note:
All arithmetic equations are in modulo 2.

Figure shows an example of the Stop-and-Wait protocol. Packet 0 is sent and acknowledged. Packet 1 is lost and resent after the time-out. The resent packet 1 is acknowledged and the timer stops. Packet 0 is sent and acknowledged, but the acknowledgment is lost. The sender has no idea if the packet or the acknowledgment is lost, so after the time-out, it resends packet 0, which is acknowledged.



Pipelining
In networking and in other areas, a task is often begun before the previous task has ended. This is known as **pipelining.** There is no pipelining in the Stop-and-Wait protocol because a sender must wait for a packet to reach the destination and be acknowledged before the next packet can be sent.

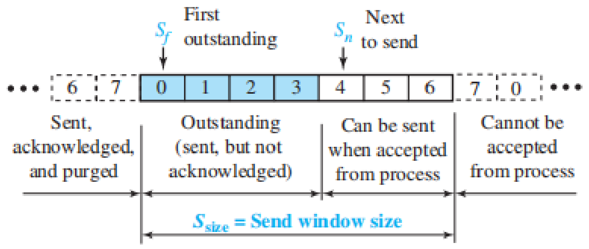
Go-Back-*N* Protocol (GBN)
The key to Go-back-*N* is that we can send several packets before receiving acknowledgments, but the receiver can only buffer one packet. We keep a copy of the sent packets until the acknowledgments arrive. Figure shows the outline of the protocol. Note that several data packets and acknowledgments can be in the channel at the same time.



Sequence Numbers
The sequence numbers are modulo 2*m*, where *m* is the size of the sequence number field in bits.

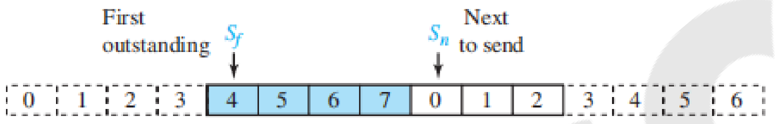
Acknowledgment Numbers
An acknowledgment number in this protocol is cumulative and defines the sequence number of the next packet expected. For example, if the acknowledgment number (ackNo) is 7, it means all packets with sequence number up to 6 have arrived, safe and sound, and the receiver is expecting the packet with sequence number 7.

Send Window
The send window is an imaginary box covering the sequence numbers of the data packets that can be in transit or can be sent. In each window position, some of these sequence numbers define the packets that have been sent; others define those that can be sent. The maximum size of the window is 2*m* – 1.

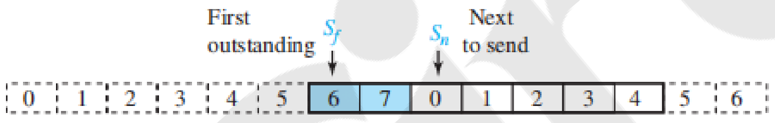

S_f First outstanding
S_n Next to send
6 7 0 1 2 3 4 5 6 7 0
Sent, acknowledged, and purged
Outstanding (sent, but not acknowledged)
Can be sent when accepted from process
Cannot be accepted from process
S_{size} = Send window size

The send window at any time divides sequence numbers into four regions. The first region includes acknowledged packets, which the sender no longer tracks. The second region contains outstanding packets that have been sent but have an unknown status. The third region defines sequence numbers for packets that can be sent but for which data hasn't been received from the application layer. The fourth region consists of sequence numbers that cannot be used until the window slides.

The window itself is an abstraction; three variables define its size and location at any time. We call these variables S_f (send window, the first outstanding packet), *Sn* (send window, the next packet to be sent), and *S*size (send window, size). The variable S_f defines the sequence number of the first (oldest) outstanding packet. The variable *Sn* holds the sequence number that will be assigned to the next packet to be sent. Finally, the variable *S*size defines the size of the window, which is fixed in our protocol.



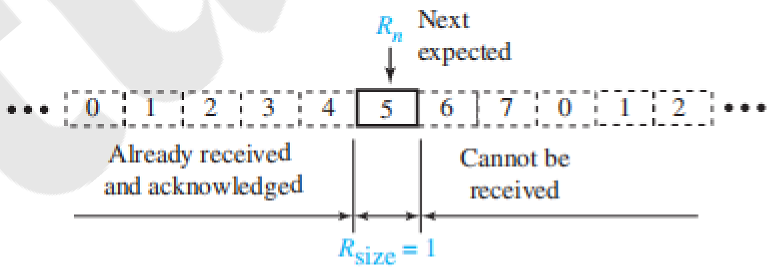
a. Window before sliding



b. Window after sliding (an ACK with ackNo = 6 has arrived)

Receive Window

The receive window ensures correct packet reception and acknowledgment. In Go-Back-N, its size is always 1, as the receiver expects a specific packet. Out-of-order packets are discarded and must be resent. Only packets matching the expected sequence number, Rn, are accepted and acknowledged. The window slides by one slot upon receiving the correct packet, with Rn updated as (Rn + 1) modulo 2m.



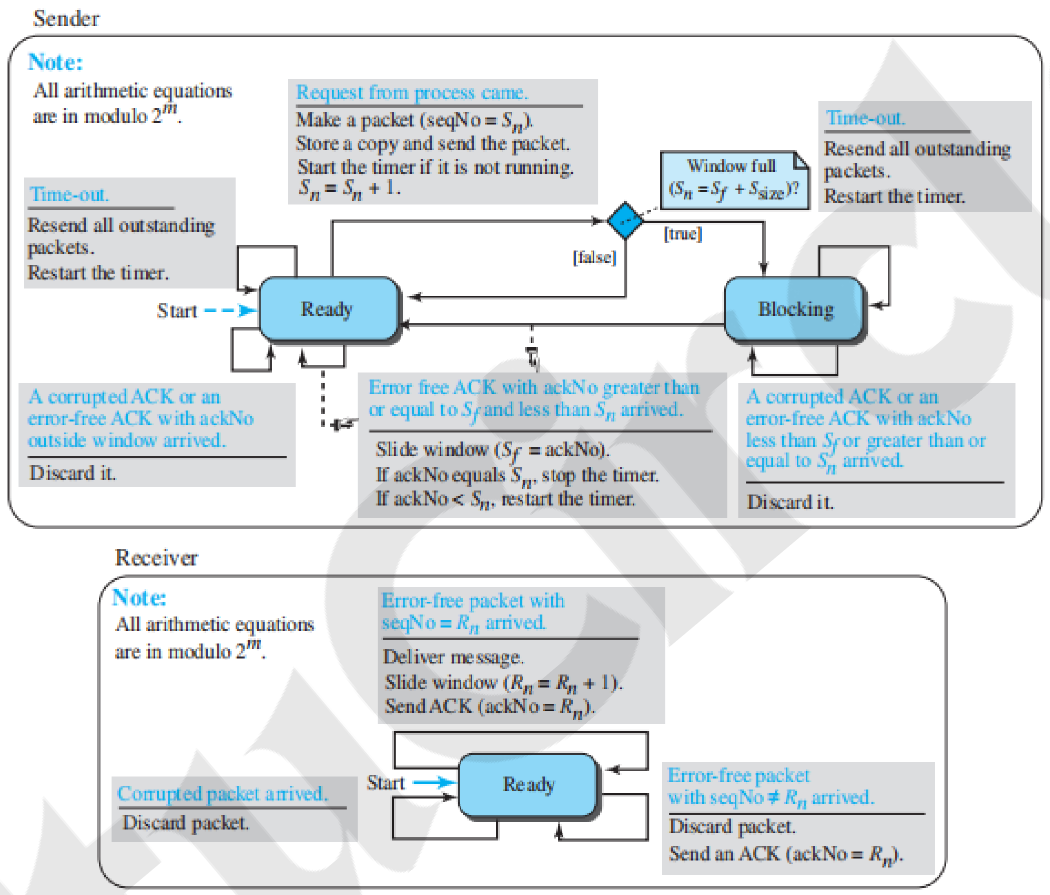
Timers

Although there can be a timer for each packet that is sent, in our protocol we use only one. The reason is that the timer for the first outstanding packet always expires first. We resend all outstanding packets when this timer expires.

Resending packets

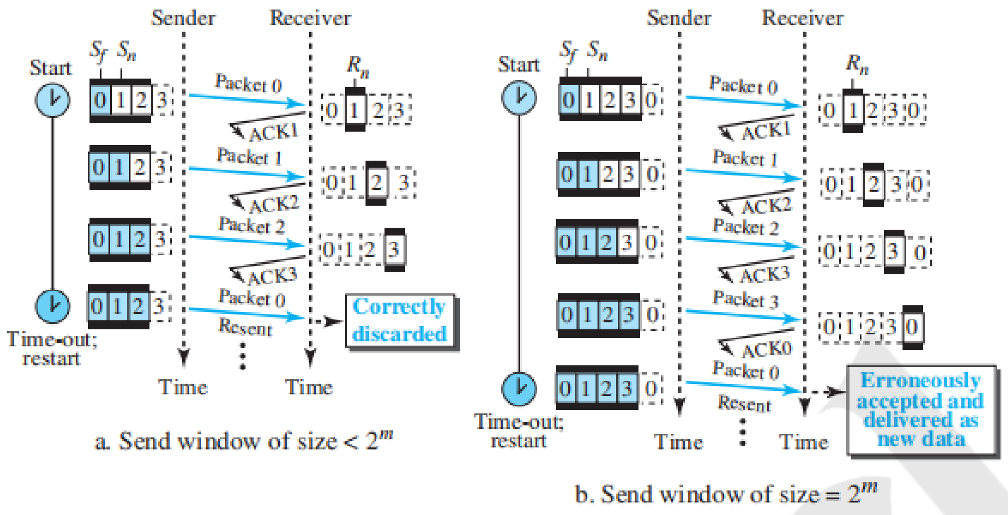
When the timer expires, the sender resends all outstanding packets. For example, suppose the sender has already sent packet 6 (*Sn* = 7), but the only timer expires. If *Sf* = 3, this means that packets 3, 4, 5, and 6 have not been acknowledged; the sender goes back and resends packets 3, 4, 5, and 6. That is why the protocol is called *Go-Back*-N. On a time-out, the machine goes back *N* locations and resends all packets.

FSMs


Sender
Note:
All arithmetic equations are in modulo 2^m.
Request from process came.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer if it is not running.
$S_n = S_n + 1$.
Time-out.
Resend all outstanding packets.
Restart the timer.
Window full ($S_n = S_f + S_{size}$)?
[true]
[false]
Time-out.
Resend all outstanding packets.
Restart the timer.
Start
Ready
Blocking
A corrupted ACK or an error-free ACK with ackNo outside window arrived.
Discard it.
Error free ACK with ackNo greater than or equal to S_f and less than S_n arrived.
Slide window (S_f = ackNo).
If ackNo equals S_n, stop the timer.
If ackNo < S_n, restart the timer.
A corrupted ACK or an error-free ACK with ackNo less than S_f or greater than or equal to S_n arrived.
Discard it.
Receiver
Note:
All arithmetic equations are in modulo 2^m.
Error-free packet with seqNo = R_n arrived.
Deliver message.
Slide window ($R_n = R_n + 1$).
Send ACK (ackNo = R_n).
Corrupted packet arrived.
Discard packet.
Start
Ready
Error-free packet with seqNo ≠ R_n arrived.
Discard packet.
Send an ACK (ackNo = R_n).

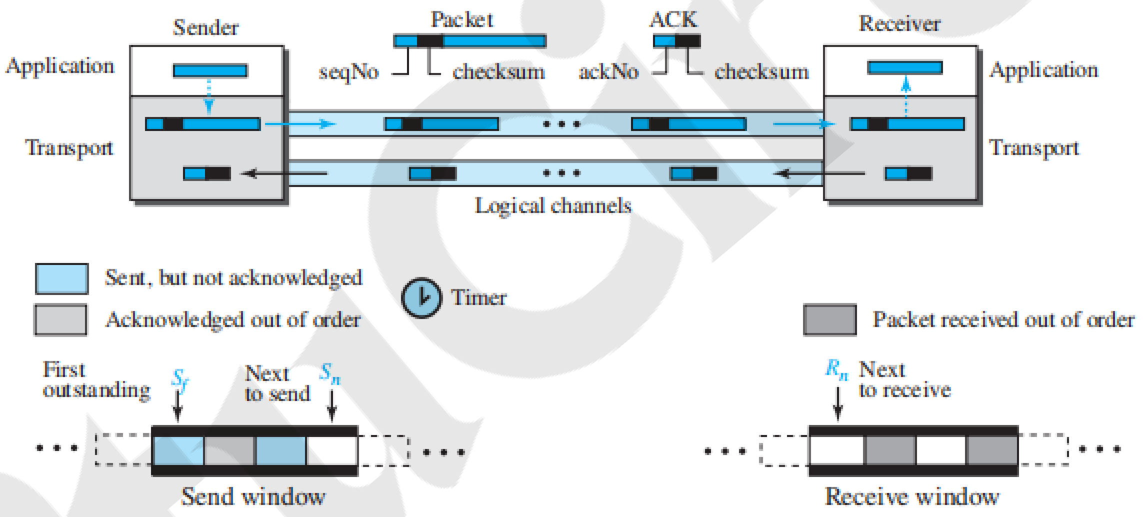
Send Window Size

We can now show why the size of the send window must be less than 2*m*. As an example, we choose *m* = 2, which means the size of the window can be 2*m* − 1, or 3.



Selective-Repeat Protocol

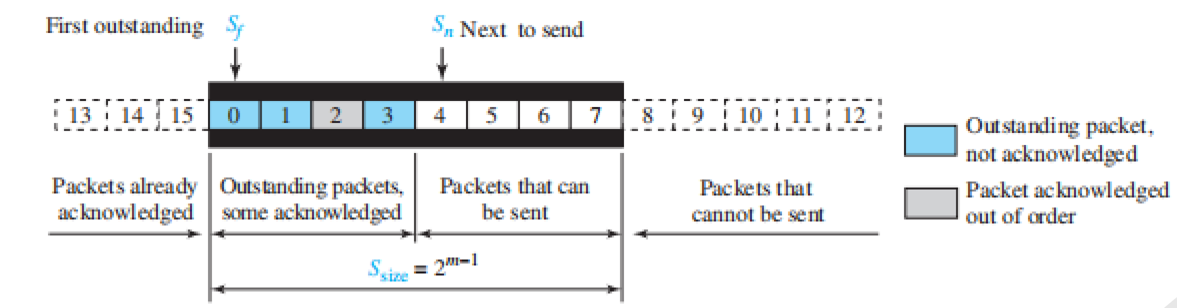
Selective-Repeat (SR) protocol, as the name implies, resends only selective packets, those that are actually lost.

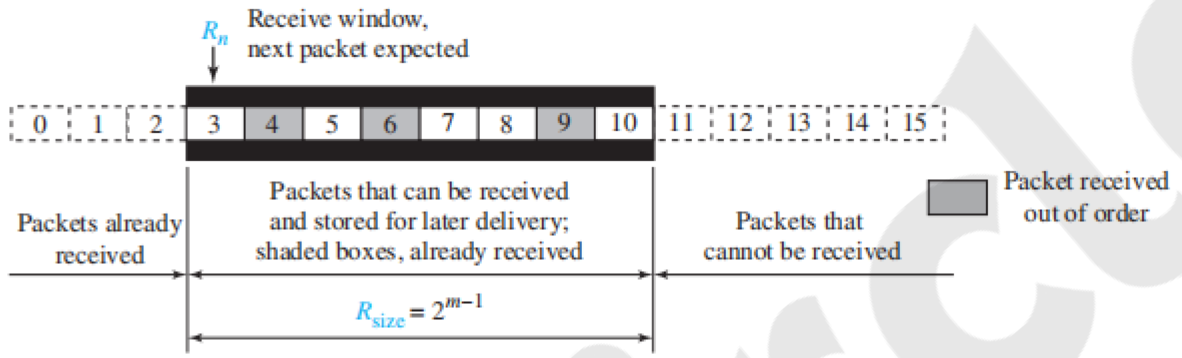


Windows

The Selective-Repeat protocol also uses two windows: a send window and a receive window. However, there are differences between the windows in this protocol and the ones in Go-Back-*N*. First, the maximum size of the send window is much smaller; it is 2^{m-1}. Second, the receive window is the same size as the send window.

The send window maximum size can be 2^{m-1}. For example, if m = 4, the sequence numbers go from 0 to 15, but the maximum size of the window is just 8 (it is 15 in the Go-Back-*N* Protocol).

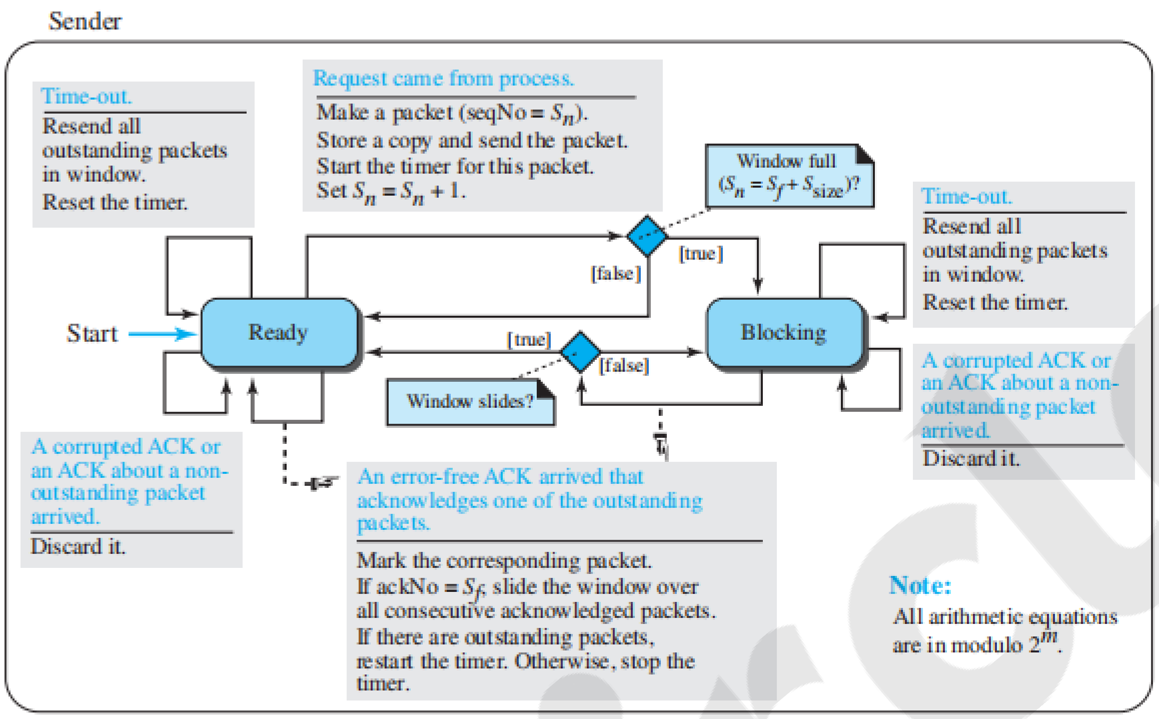

First outstanding S_f
S_n Next to send
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12
Packets already acknowledged
Outstanding packets, some acknowledged
Packets that can be sent
Packets that cannot be sent
$S_{size} = 2^{m-1}$
Outstanding packet, not acknowledged
Packet acknowledged out of order

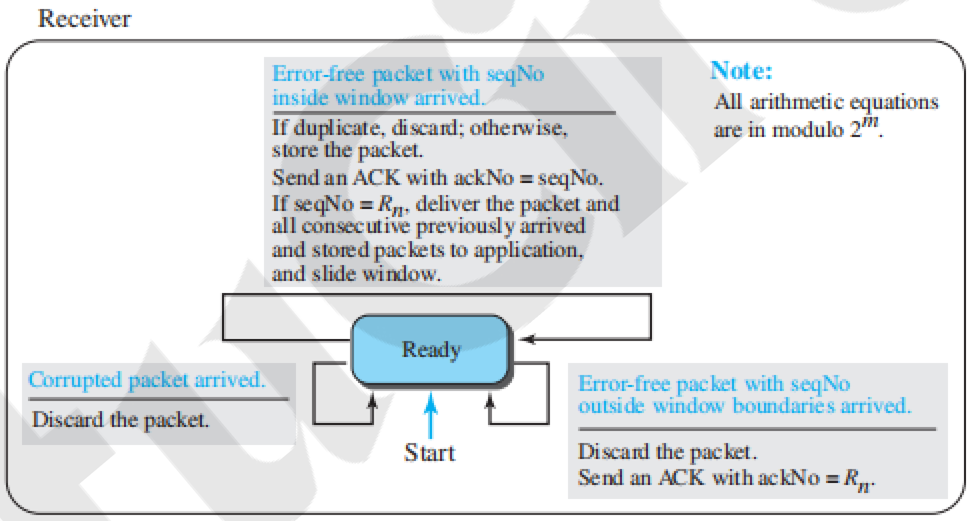

R_n Receive window, next packet expected
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Packets already received
Packets that can be received and stored for later delivery; shaded boxes, already received
Packets that cannot be received
$R_{size} = 2^{m-1}$
Packet received out of order

Timer
Theoretically, Selective-Repeat uses one timer for each outstanding packet. When a timer expires, only the corresponding packet is resent.

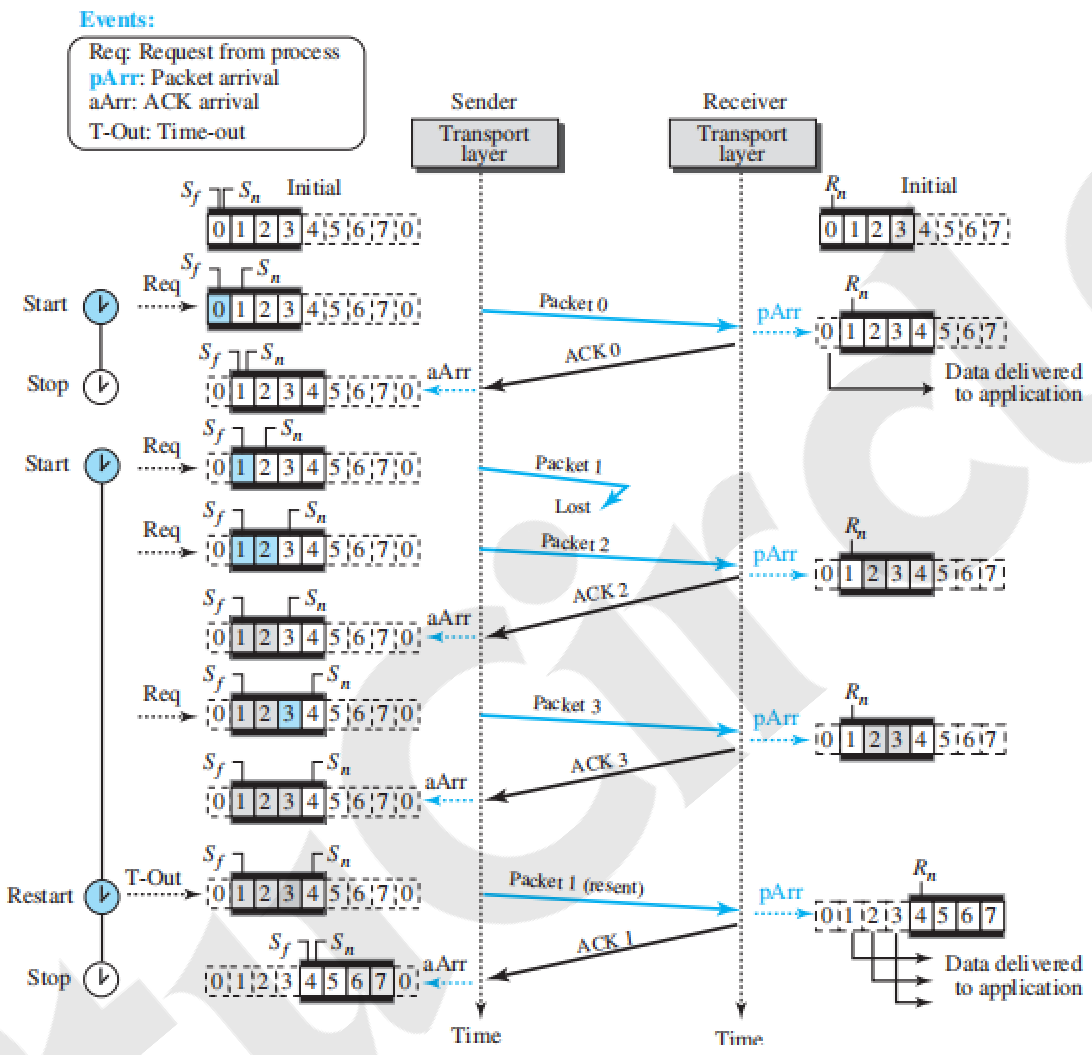
Acknowledgments
There is yet another difference between the two protocols. In GBN an ackNo is cumulative; it defines the sequence number of the next packet expected, confirming that all previous packets have been received safe and sound. The semantics of acknowledgment is different in SR. In SR, an ackNo defines the sequence number of a single packet that is received safe and sound; there is no feedback for any other.

FSMs
Figure shows the FSMs for the Selective-Repeat protocol.


Sender
Time-out.
Resend all outstanding packets in window.
Reset the timer.
Request came from process.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer for this packet.
Set $S_n = S_n + 1$.
Window full ($S_n = S_f + S_{size}$)?
[true]
[false]
Start
Ready
Blocking
[true]
[false]
Window slides?
Time-out.
Resend all outstanding packets in window.
Reset the timer.
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
An error-free ACK arrived that acknowledges one of the outstanding packets.
Mark the corresponding packet.
If ackNo = S_f, slide the window over all consecutive acknowledged packets.
If there are outstanding packets, restart the timer. Otherwise, stop the timer.
Note:
All arithmetic equations are in modulo 2^m.

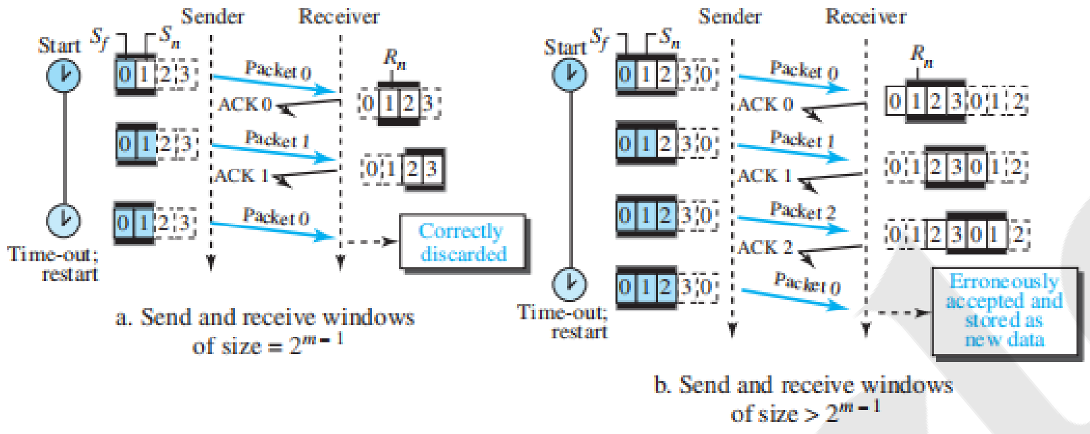

Receiver
Error-free packet with seqNo inside window arrived.
If duplicate, discard; otherwise, store the packet.
Send an ACK with ackNo = seqNo.
If seqNo = R_n, deliver the packet and all consecutive previously arrived and stored packets to application, and slide window.
Note:
All arithmetic equations are in modulo 2^m.
Ready
Corrupted packet arrived.
Discard the packet.
Start
Error-free packet with seqNo outside window boundaries arrived.
Discard the packet.
Send an ACK with ackNo = R_n.

Example
In this example packet 1 is lost. We show how Selective-Repeat behaves in this case. Figure shows the situation.

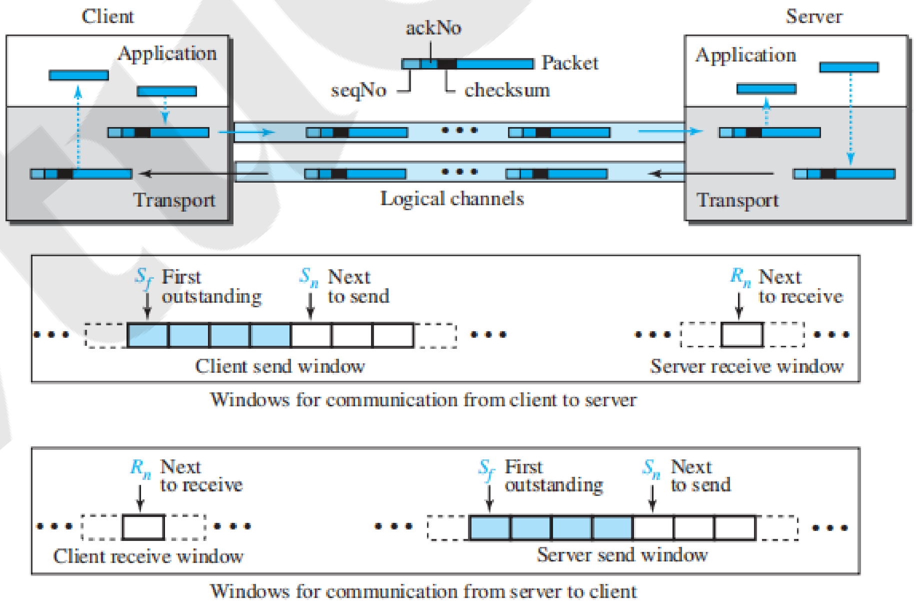


Window Sizes
We can now show why the size of the sender and receiver windows can be at most one-half of 2^m. For an example, we choose $m = 2$, which means the size of the window is $2^{m/2}$ or $2^{(m-1)} = 2$. Figure compares a window size of 2 with a window size of 3.
If the size of the window is 2 and all acknowledgments are lost, the timer for packet 0 expires and packet 0 is resent. However, the window of the receiver is now expecting packet 2, not packet 0, so this duplicate packet is correctly discarded (the sequence number 0 is not in the window). When the size of the window is 3 and all acknowledgments are lost, the sender sends a duplicate of packet 0. However, this time, the window of the receiver expects to receive packet 0

(0 is part of the window), so it accepts packet 0, not as a duplicate, but as a packet in the next cycle. This is clearly an error.


Sender
Receiver
Start
S_f
S_n
R_n
Packet 0
ACK 0
Packet 1
ACK 1
Packet 2
ACK 2
Time-out; restart
Correctly discarded
Erroneously accepted and stored as new data
a. Send and receive windows of size = 2^{m-1}
b. Send and receive windows of size > 2^{m-1}

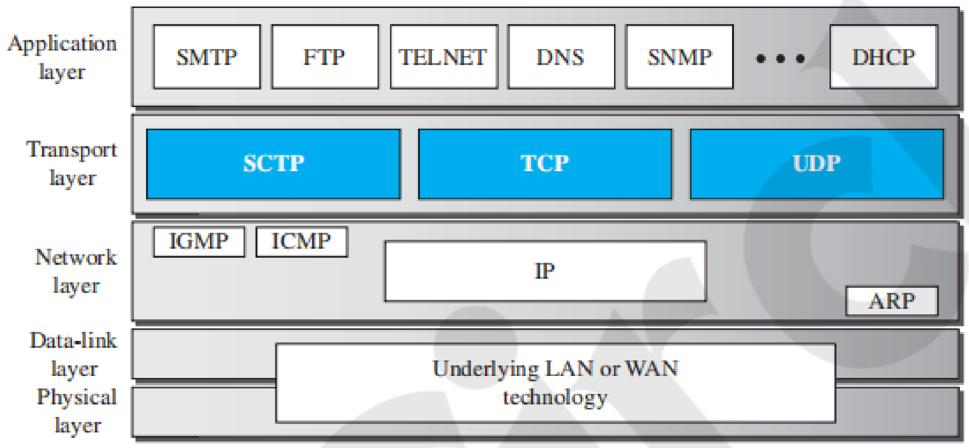
Bidirectional Protocols: Piggybacking
The four protocols discussed are all unidirectional: data packets flow in only one direction and acknowledgments travel in the other direction. In real life, data packets are normally flowing in both directions: from client to server and from server to client. This means that acknowledgments also need to flow in both directions. A technique called **piggybacking** is used to improve the efficiency of the bidirectional protocols. When a packet is carrying data from A to B, it can also carry acknowledgment feedback about arrived packets from B; when a packet is carrying data from B to A, it can also carry acknowledgment feedback about the arrived packets from A.
Figure shows the layout for the GBN protocol implemented bidirectionally using piggybacking. The client and server each use two independent windows: send and receive.



Transport-Layer Protocols

INTRODUCTION

The transport layer in the TCP/IP suite is located between the application layer and the network layer. It provides services to the application layer and receives services from the network layer. The transport layer acts as a liaison between a client program and a server program, a process-to-process connection. The transport layer is the heart of the TCP/IP protocol suite; it is the end-to-end logical vehicle for transferring data from one point to another in the Internet.



Services

Each protocol provides a different type of service and should be used appropriately.

UDP

UDP is an unreliable connectionless transport-layer protocol used for its simplicity and efficiency in applications where error control can be provided by the application-layer process.

TCP

TCP is a reliable connection-oriented protocol that can be used in any application where reliability is important.

SCTP

SCTP is a new transport-layer protocol that combines the features of UDP and TCP.

Port Numbers

One of the responsibilities of transport layer is to create a process-to-process communication; these protocols use port numbers to accomplish this. Port numbers provide end-to-end addresses at the transport layer and allow multiplexing and demultiplexing at this layer, just as IP addresses do at the network layer. Table gives some common port numbers for all three protocols.

Some well-known ports used with UDP and TCP

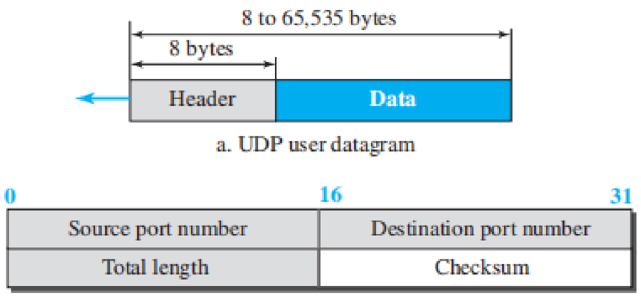
Port	*Protocol*	*UDP*	*TCP*	*SCTP*	*Description*
7	Echo	√	√	√	Echoes back a received datagram
9	Discard	√	√	√	Discards any datagram that is received
11	Users	√	√	√	Active users
13	Daytime	√	√	√	Returns the date and the time
17	Quote	√	√	√	Returns a quote of the day
19	Chargen	√	√	√	Returns a string of characters
20	FTP-data		√	√	File Transfer Protocol
21	FTP-21		√	√	File Transfer Protocol
23	TELNET		√	√	Terminal Network
25	SMTP		√	√	Simple Mail Transfer Protocol
53	DNS	√	√	√	Domain Name Service
67	DHCP	√	√	√	Dynamic Host Configuration Protocol
69	TFTP	√	√	√	Trivial File Transfer Protocol
80	HTTP		√	√	HyperText Transfer Protocol
111	RPC	√	√	√	Remote Procedure Call
123	NTP	√	√	√	Network Time Protocol
161	SNMP-server	√			Simple Network Management Protocol
162	SNMP-client	√			Simple Network Management Protocol

USER DATAGRAM PROTOCOL

The **User Datagram Protocol (UDP)** is a connectionless, unreliable transport protocol. UDP is a very simple protocol using a minimum of overhead. If a process wants to send a small message and does not care much about reliability, it can use UDP. Sending a small message using UDP takes much less interaction between the sender and receiver than using TCP.

User Datagram

UDP packets, called ***user datagrams,*** have a fixed-size header of 8 bytes made of four fields, each of 2 bytes (16 bits). Figure shows the format of a user datagram. The first two fields define the source and destination port numbers. The third field defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The last field can carry the optional checksum.



a. UDP user datagram

b. Header format

UDP Services

Process-to-Process Communication

UDP provides process-to-process communication using **socket addresses,** a combination of IP addresses and port numbers.

Connectionless Services

UDP provides a *connectionless service*. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program. The user datagrams are not numbered. There is no connection establishment and no connection termination. This means that each user datagram can travel on a different path.

Flow Control

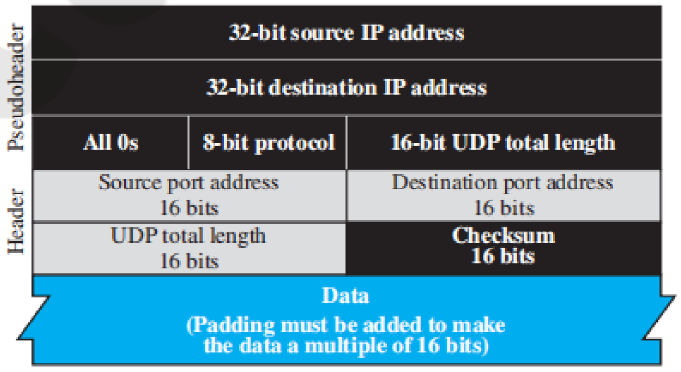
UDP is a very simple protocol. There is no *flow control*, and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service, if needed.

Error Control

There is no *error control* mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded.

Checksum

UDP checksum calculation includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer. The *pseudoheader* is the part of the header of the IP packet in which the user datagram is to be encapsulated with some fields filled with 0s.


Pseudoheader
Header
32-bit source IP address
32-bit destination IP address
All 0s
8-bit protocol
16-bit UDP total length
Source port address 16 bits
Destination port address 16 bits
UDP total length 16 bits
Checksum 16 bits
Data
(Padding must be added to make the data a multiple of 16 bits)

If the checksum does not include the pseudoheader, a user datagram may arrive safe and sound. However, if the IP header is corrupted, it may be delivered to the wrong host. The protocol field is added to ensure that the packet belongs to UDP, and not to TCP. The value of the protocol field for UDP is 17. If this value is changed during transmission, the checksum calculation at the receiver will detect it and UDP drops the packet. It is not delivered to the wrong protocol.

Congestion Control
Since UDP is a connectionless protocol, it does not provide congestion control. UDP assumes that the packets sent are small and sporadic and cannot create congestion in the network.

Encapsulation and Decapsulation
To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages.

Queuing
In UDP, queues are associated with ports. At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process. Other implementations create only an incoming queue associated with each process.

Multiplexing and Demultiplexing
In a host running a TCP/IP protocol suite, there is only one UDP but possibly several processes that may want to use the services of UDP. To handle this situation, UDP multiplexes and demultiplexes.

UDP Applications
Although UDP meets almost none of the criteria we mentioned earlier for a reliable transport-layer protocol, UDP is preferable for some applications. The reason is that some services may have some side effects that are either unacceptable or not preferable.

UDP Features

1. ***Connectionless Service :*** UDP is connectionless, with each packet independent of others. This can be an advantage or disadvantage depending on the application's needs. It's advantageous when a client sends a short request and receives a short response, as both can fit in a single datagram. In such cases, avoiding the overhead of establishing and closing a connection is beneficial. A connection-oriented service requires at least 9 packets exchanged between client and server, while a connectionless service only needs 2. The connectionless service reduces delay, making it preferable when minimizing delay is important.
2. ***Lack of Error Control:*** UDP lacks error control and provides an unreliable service, while most applications expect reliability from a transport-layer protocol. Although reliable service is preferred, it may have drawbacks for certain applications. If a message part is lost or corrupted, it must be resent, causing delays in delivery to the application layer.

This leads to uneven delays between different parts of the message. While some applications may not notice these delays, they can be problematic for others.

3. ***Lack of Congestion Control:*** UDP does not provide congestion control. However, UDP does not create additional traffic in an error-prone network. TCP may resend a packet several times and thus contribute to the creation of congestion or worsen a congested situation. Therefore, in some cases, lack of error control in UDP can be considered an advantage when congestion is a big issue.

Typical Applications

The following shows some typical applications that can benefit more from the services of UDP than from those of TCP.

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FTP that needs to send bulk data.
- UDP is suitable for a process with internal flow- and error-control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.
- UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.
- UDP is used for management processes such as SNMP.
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP).
- UDP is normally used for interactive real-time applications that cannot tolerate uneven delay between sections of a received message.

TRANSMISSION CONTROL PROTOCOL

Transmission Control Protocol (TCP) is a connection-oriented, reliable protocol. TCP explicitly defines connection establishment, data transfer, and connection tear down phases to provide a connection-oriented service. TCP uses a combination of GBN and SR protocols to provide reliability. To achieve this goal, TCP uses checksum (for error detection), retransmission of lost or corrupted packets, cumulative and selective acknowledgments, and timers.

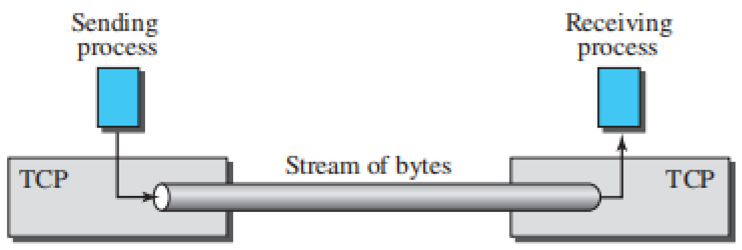
TCP Services

Process-to-Process Communication

As with UDP, TCP provides process-to-process communication using port numbers.

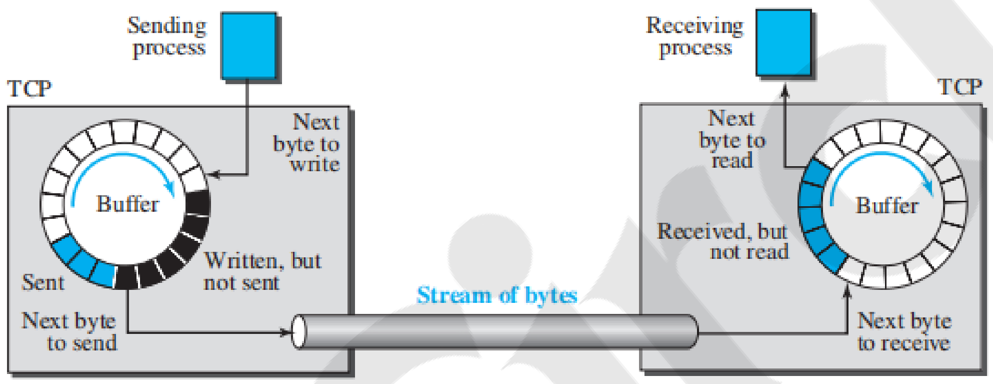
Stream Delivery Service

TCP, unlike UDP, is a stream-oriented protocol. TCP, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary “tube” that carries their bytes across the Internet. This imaginary environment is depicted in Figure. The sending process produces (writes to) the stream and the receiving process consumes (reads from) it.



Sending and Receiving Buffers

Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction.



These buffers are essential for flow control and error control in TCP. A buffer can be implemented as a circular array of 1-byte locations, as shown in the figure, though typically they contain hundreds or thousands of bytes. The figure illustrates data movement in one direction, showing three sections in the sender's buffer:

1. **Empty chambers** (white) that can be filled by the sending process.
2. **Sent but unacknowledged bytes** (colored), which remain in the buffer until acknowledged.
3. **Bytes ready to be sent** (shaded), though TCP may only send part of this due to receiver slowness or network congestion.

Once acknowledged, the sent bytes' chambers are recycled, making them available for reuse, which is why a circular buffer is used.

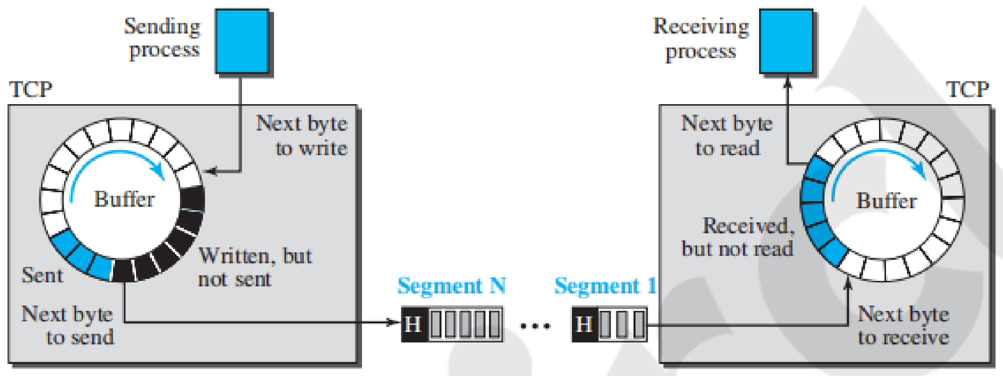
At the receiver, the buffer is simpler, divided into two areas:

1. **Empty chambers** (white), ready to be filled with received bytes.
2. **Filled chambers** (colored), containing received bytes.

Segments shown in the figure vary in size, but real segments typically carry hundreds or thousands of bytes.

Segments

Although buffering handles the speed difference between the producing and consuming processes, one more step is needed before sending data. The network layer transmits data in packets, not as a byte stream. TCP groups bytes into **segments**, adds a control header, and passes them to the network layer for transmission, where they are encapsulated in IP datagrams. This process is transparent to the receiving application, and TCP handles issues like out-of-order, lost, or corrupted segments without the receiving process being aware. Figure shows how segments are created from bytes in the buffers.



Full-Duplex Communication

TCP offers *full-duplex service*, where data can flow in both directions at the same time. Each TCP endpoint then has its own sending and receiving buffer, and segments move in both directions.

Multiplexing and Demultiplexing

Like UDP, TCP performs multiplexing at the sender and demultiplexing at the receiver. However, since TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes.

Connection-Oriented Service

TCP, unlike UDP, is a connection-oriented protocol. When a process at site A wants to send to and receive data from another process at site B, the following three phases occur:

1. The two TCP's establish a logical connection between them.

2. Data are exchanged in both directions.

3. The connection is terminated.

Reliable Service

TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control.

TCP Features

Numbering System Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header. Instead, there are two fields, called the *sequence number* and the *acknowledgment number*. These two fields refer to a byte number and not a segment number.

1. ***Byte Number-*** TCP numbers all data bytes transmitted in a connection, with independent numbering in each direction. The numbering doesn't start from 0 but from an arbitrary number between 0 and 2^{32} - 1. For example, if the starting number is 1057 and 6000 bytes are to be sent, the bytes will be numbered from 1057 to 7056. This numbering is used for flow and error control.

2. ***Sequence Number -***After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent. The sequence number, in each direction, is defined as follows:
   1. The sequence number of the first segment is the ISN (initial sequence number), which is a random number.
   2. The sequence number of any other segment is the sequence number of the previous segment plus the number of bytes (real or imaginary) carried by the previous segment. Later, we show that some control segments are thought of as carrying one imaginary byte.
3. ***Acknowledgment Number -*** In TCP's full duplex communication, both parties can send and receive data simultaneously. Each party numbers the bytes, typically starting with different numbers. The sequence number in each direction indicates the first byte in the segment, while the acknowledgment number confirms receipt of bytes and specifies the next expected byte. The acknowledgment number is cumulative, meaning if it's 5643, all bytes up to 5642 have been received, though the first byte may not be 0.

Segment

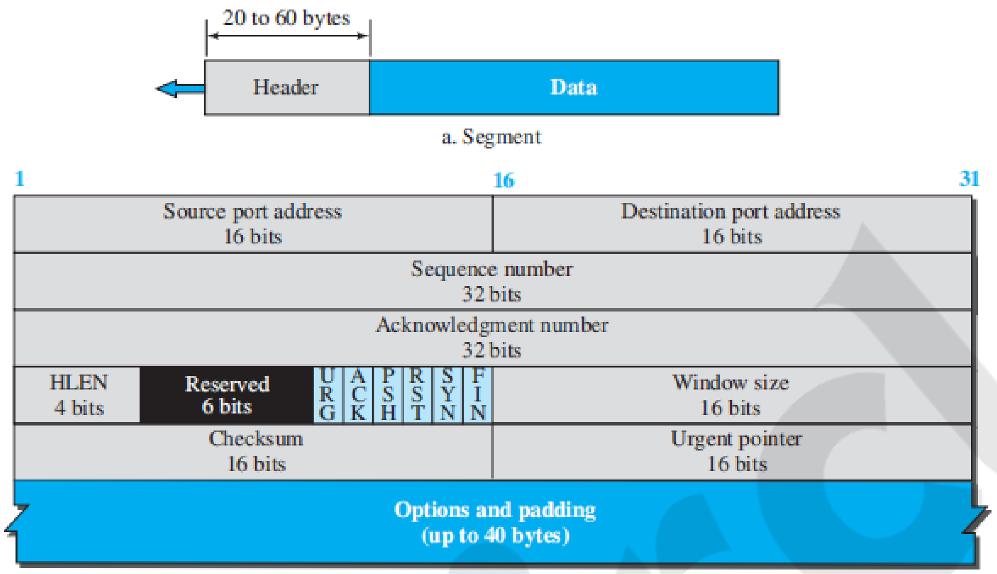
A packet in TCP is called a *segment.*

Format

The format of a segment is shown in Figure . The segment consists of a header of 20 to 60 bytes, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

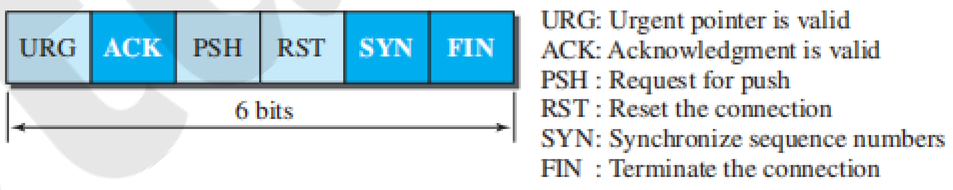
- ***Source port address.*** This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
- ***Destination port address.*** This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.

- ***Sequence number.*** This 32-bit field defines the number assigned to the first byte of data contained in this segment. TCP is a stream transport protocol. To ensure connectivity, each byte to be transmitted is numbered. The sequence number tells the destination which byte in this sequence is the first byte in the segment. During connection establishment each party uses a random number generator to create an **initial sequence number** (ISN), which is usually different in each direction.



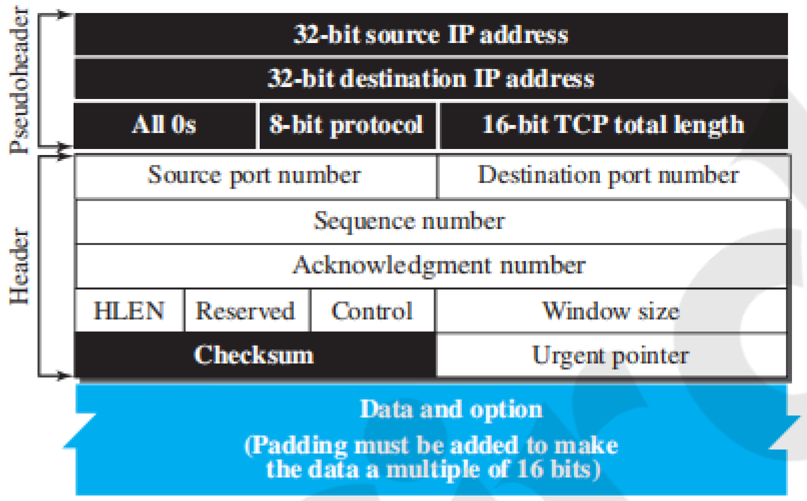
- ***Acknowledgment number.*** This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party. If the receiver of the segment has successfully received byte number *x* from the other party, it returns *x* + 1 as the acknowledgment number. Acknowledgment and data can be piggybacked together.

- ***Header length.*** This 4-bit field indicates the number of 4-byte words in the TCP header. The length of the header can be between 20 and 60 bytes. Therefore, the value of this field is always between 5 (5 × 4 = 20) and 15 (15 × 4 = 60).
- ***Control.*** This field defines 6 different control bits or flags, as shown in Figure. One or more of these bits can be set at a time. These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP.


URG
ACK
PSH
RST
SYN
FIN
6 bits
URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH : Request for push
RST : Reset the connection
SYN: Synchronize sequence numbers
FIN : Terminate the connection

- ***Window size.*** This field defines the window size of the sending TCP in bytes. Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes. This value is normally referred to as the receiving window (*rwnd*) and is determined by the receiver. The sender must obey the dictation of the receiver in this case.

- ***Checksum.*** This 16-bit field contains the checksum. The calculation of the checksum for TCP follows the same procedure as the one described for UDP. However, the use of the checksum in the UDP datagram is optional, whereas the use of the checksum for TCP is mandatory. The same pseudoheader, serving the same purpose, is added to the segment. For the TCP pseudoheader, the value for the protocol field is 6. See Figure



- ***Urgent pointer.*** This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines a value that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.
- ***Options.*** There can be up to 40 bytes of optional information in the TCP header.

Encapsulation

A TCP segment encapsulates the data received from the application layer. The TCP segment is encapsulated in an IP datagram, which in turn is encapsulated in a frame at the data-link layer.

A TCP Connection

TCP is connection-oriented. A connection-oriented transport protocol establishes a logical path between the source and destination. All of the segments belonging to a message are then sent over this logical path. Using a single logical pathway for the entire message facilitates the acknowledgment process as well as retransmission of damaged or lost frames.
TCP is connection-oriented, but its connection is logical, not physical. TCP operates at a higher layer, using IP (a connectionless protocol) to send individual segments. If a segment is lost or corrupted, TCP retransmits it, whereas IP itself doesn't handle retransmissions.
If a segment arrives out of order, TCP holds it until the missing segments arrive; IP is unaware of this reordering.

In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

Connection Establishment

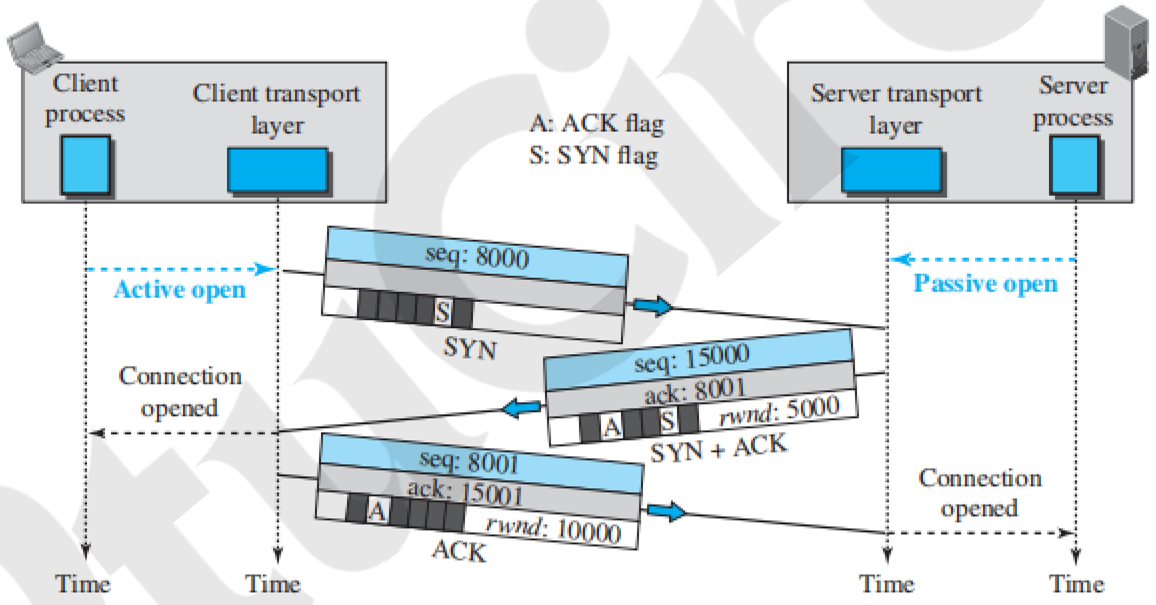
TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.

Three-Way Handshaking

The connection establishment in TCP is called ***three-way handshaking.*** For example, an application program, called the *client*, wants to make a connection with another application program, called the *server*, using TCP as the transport-layer protocol.

The process starts with the server. The server program tells its TCP that it is ready to accept a connection. This request is called a *passive open*. Although the server TCP is ready to accept a connection from any machine in the world, it cannot make the connection itself.

The client program issues a request for an *active open*. A client that wishes to connect to an open server tells its TCP to connect to a particular server. TCP can now start the three-way handshaking process, as shown in Figure.



The three steps in this phase are as follows.

1. The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers. This sequence number is called the *initial sequence number (ISN)*. This segment does not contain an acknowledgment number. The SYN segment is a control segment and carries no data.
2. The server sends the second segment, a SYN + ACK segment with two flag bits set as: SYN and ACK. This segment has a dual purpose. First, it is a SYN segment for communication in the other direction. The server uses this segment to initialize a

sequence number for numbering the bytes sent from the server to the client. The server also acknowledges the receipt of the SYN segment from the client by setting the ACK flag and displaying the next sequence number it expects to receive from the client. Because the segment contains an acknowledgment, it also needs to define the receive.

3. The client sends the third segment. This is just an ACK segment. It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field.

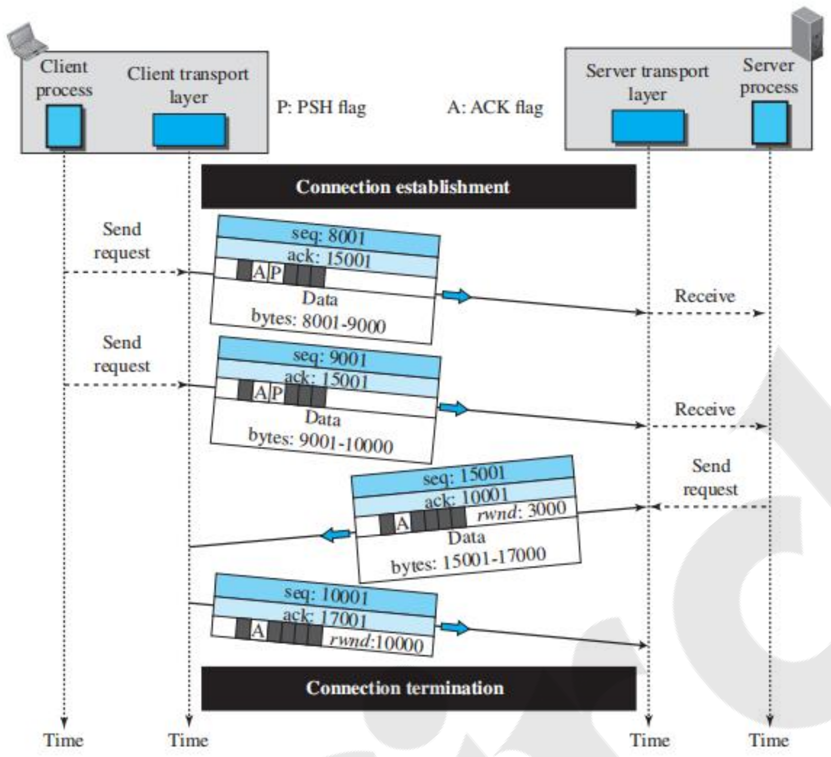
SYN Flooding Attack

The connection establishment procedure in TCP is susceptible to a serious security problem called *SYN flooding attack.* This happens when one or more malicious attackers send a large number of SYN segments to a server pretending that each of them is coming from a different client by faking the source IP addresses in the datagrams. The server, assuming that the clients are issuing an active open, allocates the necessary resources, such as creating transfer control block (TCB) tables and setting timers. The TCP server then sends the SYN + ACK segments to the fake clients, which are lost.

When the server waits for the third leg of the handshaking process, however, resources are allocated without being used. If, during this short period of time, the number of SYN segments is large, the server eventually runs out of resources and may be unable to accept connection requests from valid clients. This SYN flooding attack belongs to a group of security attacks known as a *denial-of-service attack*, in which an attacker monopolizes a system with so many service requests that the system overloads and denies service to valid requests.

Data Transfer

After connection is established, bidirectional data transfer can take place. The client and server can send data and acknowledgments in both directions. The acknowledgment is piggybacked with the data. Figure shows an example.



Pushing Data

TCP uses a buffer at the sending side to store data from the application program, allowing flexibility in segment size selection. The receiving TCP also buffers incoming data, delivering it when convenient. This flexibility enhances TCP efficiency but may not suit applications requiring immediate interaction, where delays in transmission and delivery are unacceptable. In such cases, the sending application can request a push operation. This forces the sending TCP to create and send a segment immediately, setting the push bit (PSH) to ensure the receiving TCP delivers the data without waiting for additional data.

Urgent Data

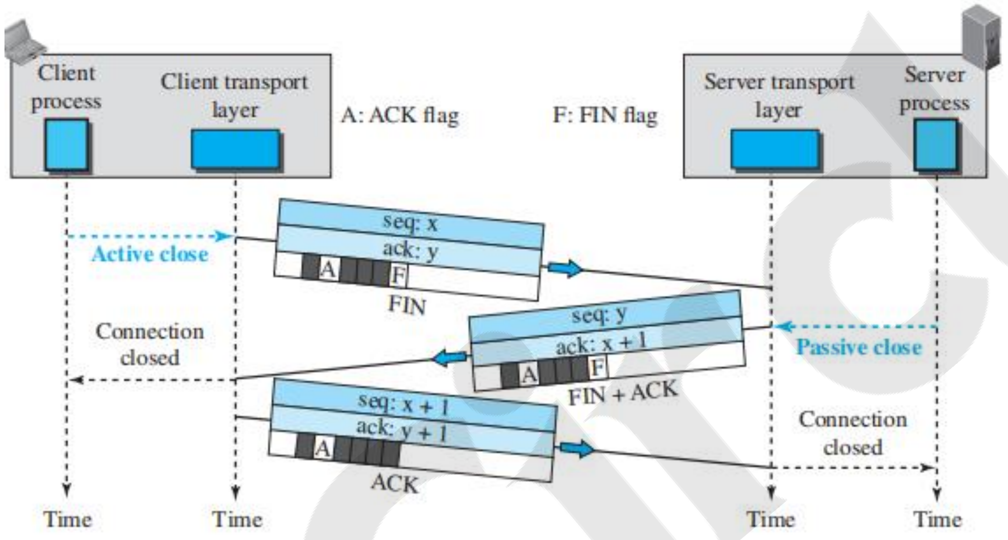
TCP is a stream-oriented protocol, meaning data is presented as a continuous stream of bytes, each with a specific position. However, when an application needs to send urgent data requiring special handling at the receiving end, it can use the URG bit. The sending TCP marks this data as urgent by creating a segment with the urgent data at the beginning and using the urgent pointer to indicate its end. For instance, if the sequence number is 15000 and the urgent pointer is 200, the urgent data covers bytes 15000 to 15200, while the rest of the segment contains non-urgent data. It's important to note that TCP's urgent data is not a priority or out-of-band service; instead, it flags a part of the byte stream for special attention by the receiving application. TCP delivers all bytes in sequence but informs the receiving application of the urgent data's start and end, leaving its handling to the application.

Connection Termination

Either of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client. Most implementations today allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

Three-Way Handshaking

Most implementations today allow *three-way handshaking* for connection termination.



1. In this situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set.
2. The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN + ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction.
3. The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server. This segment contains the acknowledgment number, which is one plus the sequence number received in the FIN segment from the server. This segment cannot carry data and consumes no sequence numbers.

Half-Close

In TCP, one end can stop sending data while still receiving data. This is called a ***half close.*** Either the server or the client can issue a half-close request. It can occur when the server needs all the data before processing can begin. A good example is sorting. When the client sends data to the server to be sorted, the server needs to receive all the data before sorting can start. This means the client, after sending all data, can close the connection in the client-to-server direction. However, the server-to-client direction must remain open to return the sorted data. The server, after receiving the data, still needs time for sorting; its outbound direction must remain open. After half-closing the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server. The client cannot send any more data to the server.

Connection Reset

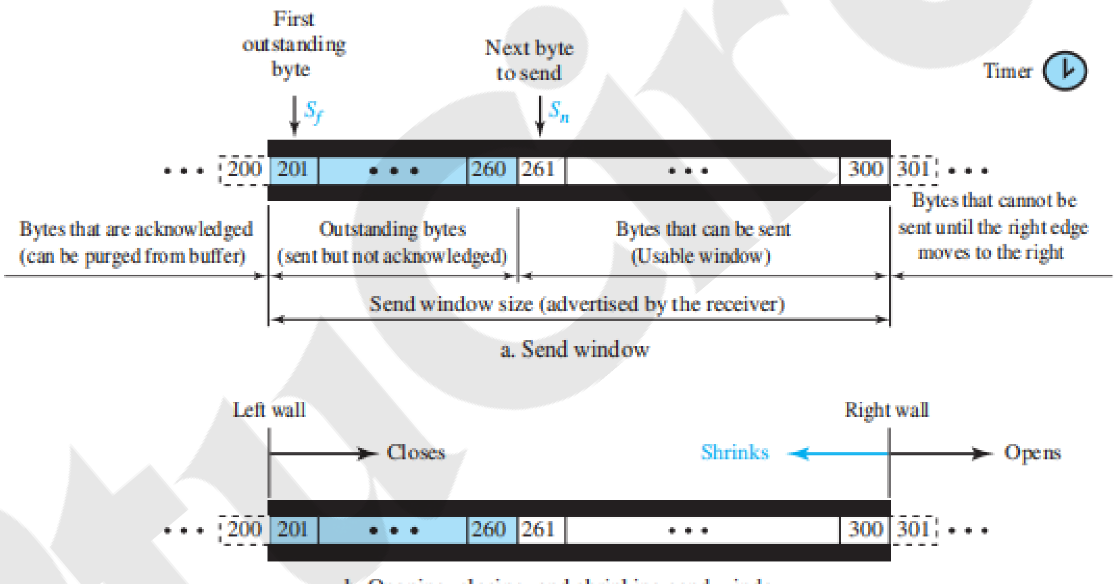
TCP at one end may deny a connection request, may abort an existing connection, or may terminate an idle connection. All of these are done with the RST (reset) flag.

Windows in TCP

TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication.

Send Window

Figure shows an example of a send window. The window size is 100 bytes, but later we see that the send window size is dictated by the receiver (flow control) and the congestion in the underlying network (congestion control). The figure shows how a send window *op*



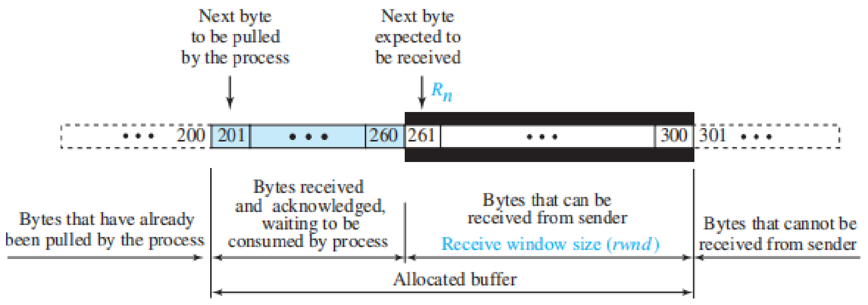
First outstanding byte
S_f
Next byte to send
S_n
Timer
200
201
260
261
300
301
Bytes that are acknowledged (can be purged from buffer)
Outstanding bytes (sent but not acknowledged)
Bytes that can be sent (Usable window)
Bytes that cannot be sent until the right edge moves to the right
Send window size (advertised by the receiver)
Left wall
Closes
Right wall
Shrinks
Opens

a. Send window

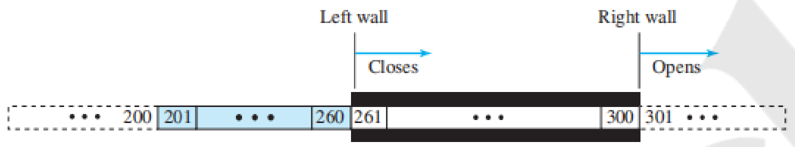
b. Opening, closing, and shrinking send window

Receive Window

Figure shows an example of a receive window. The window size is 100 bytes. The figure also shows how the receive window opens and closes; in practice, the window should never shrink.


Next byte to be pulled by the process
Next byte expected to be received
R_n
200
201
260
261
300
301
Bytes that have already been pulled by the process
Bytes received and acknowledged, waiting to be consumed by process
Bytes that can be received from sender
Receive window size (rwnd)
Bytes that cannot be received from sender
Allocated buffer

a. Receive window and allocated buffer



b. Opening and closing of receive window

Flow Control

The *flow control* balances the rate a producer creates data with the rate a consumer can use the data. TCP separates flow control from error control.

Opening and Closing Windows

To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established. The receive window closes (moves its left wall to the right) when more bytes arrive from the sender; it opens (moves its right wall to the right) when more bytes are pulled by the process.

Shrinking of Windows

The receive window cannot shrink. The send window, on the other hand, can shrink if the receiver defines a value for *rwnd* that results in shrinking the window.

Window Shutdown

Shrinking the send window by moving its right wall to the left is strongly discouraged. However, there is one exception: the receiver can temporarily shut down the window by sending a *rwnd* of 0. This can happen if for some reason the receiver does not want to receive any data from the sender for a while.

Silly Window Syndrome

A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both. Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation. For example, if TCP sends segments containing only 1 byte of data, it means that a 41-byte datagram (20 bytes of TCP header and 20 bytes of IP header) transfers only 1 byte of user data. Here the overhead is 41/1, which indicates that we are using the capacity of the network very inefficiently. The inefficiency is even worse after accounting for the data-link layer and physical-layer overhead. This problem is called the ***silly window syndrome.***

Error Control

TCP is a reliable transport-layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in order, without error, and without any part lost or duplicated.

Checksum

Each segment includes a checksum field, which is used to check for a corrupted segment. If a segment is corrupted, as detected by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost. TCP uses a 16-bit checksum that is mandatory in every segment.

Acknowledgment

TCP uses acknowledgments to confirm the receipt of data segments. Control segments that carry no data, but consume a sequence number, are also acknowledged. ACK segments are never acknowledged.

Acknowledgment Type

In the past, TCP used only one type of acknowledgment: cumulative acknowledgment.

Today, some TCP implementations also use selective acknowledgment.

Cumulative Acknowledgment (ACK)

In TCP, segments are acknowledged cumulatively. The receiver advertises the next byte it expects, ignoring out-of-order segments. This method, called **positive cumulative acknowledgment**, provides no feedback for lost, discarded, or duplicate segments. The 32-bit ACK field in the TCP header is used for this, and it's valid only when the ACK flag is set.

Selective Acknowledgment (SACK)

Many TCP implementations now support **selective acknowledgment (SACK)**, which complements ACK by reporting out-of-order or duplicated byte blocks. Since the TCP header lacks space for this, SACK is implemented as an option at the end of the header.

Generating Acknowledgments

Common rules for generating ACK

1. When end A sends a data segment to end B, it must include (piggyback) an acknowledgment that gives the next sequence number it expects to receive. This rule decreases the number of segments needed and therefore reduces traffic.

2. When the receiver has no data to send and it receives an in-order segment (with expected sequence number) and the previous segment has already been acknowledged, the receiver delays sending an ACK segment until another segment arrives or until a period of time (normally 500 ms) has passed.
3. When a segment arrives with a sequence number that is expected by the receiver, and the previous in-order segment has not been acknowledged, the receiver immediately sends an ACK segment.
4. When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. This leads to the *fast retransmission* of missing segments.
5. When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected. This informs the receiver that segments reported missing have been received.
6. If a duplicate segment arrives, the receiver discards the segment, but immediately sends an acknowledgment indicating the next in-order segment expected. This solves some problems when an ACK segment itself is lost.

Retransmission

The heart of the error control mechanism is the retransmission of segments. When a segment is sent, it is stored in a queue until it is acknowledged. When the retransmission timer expires or when the sender receives three duplicate ACKs for the first segment in the queue, that segment is retransmitted.

Retransmission after RTO

TCP maintains a retransmission time-out (RTO) for each connection. If the timer expires, the segment at the front of the queue (with the smallest sequence number) is retransmitted, and the timer is reset. The RTO is dynamic, adjusting based on the round-trip time (RTT), which is the time taken for a segment to reach its destination and receive an acknowledgment.

Retransmission after Three Duplicate ACK Segments

To avoid waiting for an RTO, many TCP implementations use **fast retransmission**. If three duplicate ACKs (the original and three identical copies) are received for a segment, TCP immediately retransmits the missing segment without waiting for the RTO.

Out-of-Order Segments

TCP implementations today do not discard out-of-order segments. They store them temporarily and flag them as out-of-order segments until the missing segments arrive. However, that out-of-order segments are never delivered to the process. TCP guarantees that data are delivered to the process in order.

Some Scenarios

Normal Operation
In normal bidirectional data transfer, the client sends a segment, and the server sends three. The client delays acknowledgments by 500 ms to check if more segments arrive. If no further segments arrive before the timer expires, the acknowledgment is sent immediately.
Lost Segment
If a segment is lost or corrupted, the receiver stores the out-of-order data in its buffer and sends an acknowledgment for the missing byte. The sender retransmits the lost segment after the RTO timer expires.
Fast Retransmission
If a segment is lost and the receiver sends three duplicate ACKs, the sender immediately retransmits the missing segment without waiting for the RTO timer, following the fast retransmission rule.
Delayed Segment
TCP segments may arrive with delays due to IP's connectionless nature. If a segment is delayed and arrives after its retransmission, it is considered a duplicate and discarded by the receiver.
Duplicate Segment
A duplicate segment occurs when a segment is delayed and the sender resends it, thinking it is lost. The receiver discards any duplicate segment and sends an acknowledgment for the expected sequence number.
Automatically Corrected Lost ACK
In cumulative acknowledgment, if an ACK is lost, the next ACK automatically corrects the loss. The sender continues normally because the lost ACK is not critical as long as the next one arrives.
Lost Acknowledgment Corrected by Resending a Segment
If an acknowledgment is lost and not corrected by the next one, the RTO timer triggers a retransmission of the segment. The receiver discards the duplicate segment and resends the last ACK to inform the sender.
Deadlock Created by Lost Acknowledgment
A deadlock can occur if an acknowledgment with a zero window size is lost. The sender waits for the window to reopen, while the receiver believes the sender is waiting for data. To prevent deadlock, a persistence timer is used.

TCP Congestion Control

TCP uses different policies to handle the congestion in the network.

Congestion Window

To control the number of segments to transmit, TCP uses another variable called a *congestion window, cwnd,* whose size is controlled by the congestion situation in the network. The *cwnd* variable and the *rwnd* variable together define the size of the send window in TCP. The first is related to the congestion in the middle (network); the second is related to the congestion at the end. The actual size of the window is the minimum of these two.

Actual window size = minimum (*rwnd*, *cwnd*)

Congestion Detection

TCP detects network congestion through two main events: time-outs and the receipt of three duplicate ACKs. A time-out occurs when the sender does not receive an ACK for a segment or group of segments before the timer expires, signaling the likelihood of severe congestion and possible segment loss. On the other hand, receiving three duplicate ACKs (four identical ACKs) indicates that one segment is missing, but others have been received, suggesting mild congestion or network recovery. While early TCP versions like **Tahoe** treated both events the same, later versions like **Reno** distinguish between the two, with time-outs indicating stronger congestion than duplicate ACKs. TCP relies on ACKs as the only feedback to detect congestion, where missing or delayed ACKs serve as indicators of network conditions.

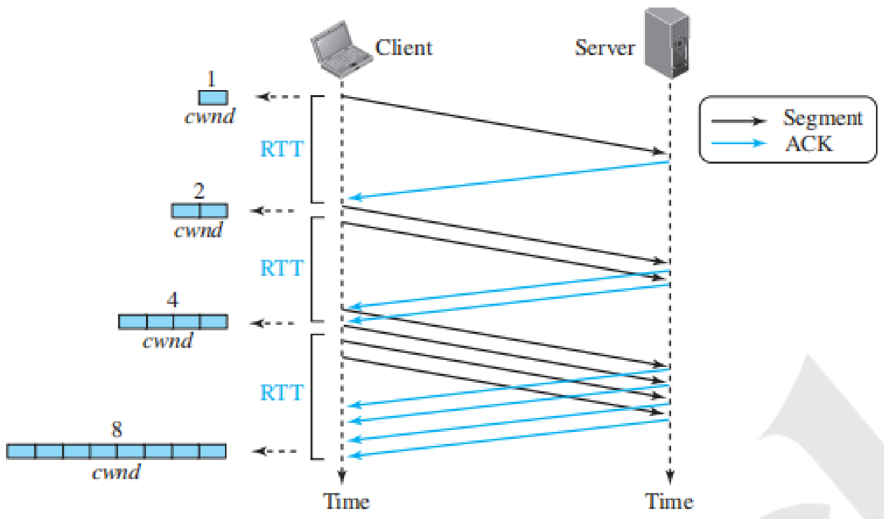
Congestion Policies

TCP's general policy for handling congestion is based on three algorithms: slow start, congestion avoidance, and fast recovery.

Slow Start: Exponential Increase

The slow-start algorithm begins with the congestion window (cwnd) set to one maximum segment size (MSS) and increases by one MSS for each received acknowledgment. The MSS is negotiated during connection establishment. Despite the name, the algorithm grows exponentially. Initially, the sender transmits one segment, and upon receiving the ACK, the cwnd increases by 1, allowing the sender to transmit two segments. Each acknowledgment further increases cwnd, doubling the number of segments the sender can transmit, resulting in rapid growth as long as no congestion is detected. The size of the congestion window in this algorithm is a function of the number of ACKs arrived and can be determined as follows.

If an ACK arrives, *cwnd* = *cwnd* + **1.**



If we look at the size of the *cwnd* in terms of round-trip times (RTTs), we find that the growth rate is exponential in terms of each round trip time, which is a very aggressive approach:

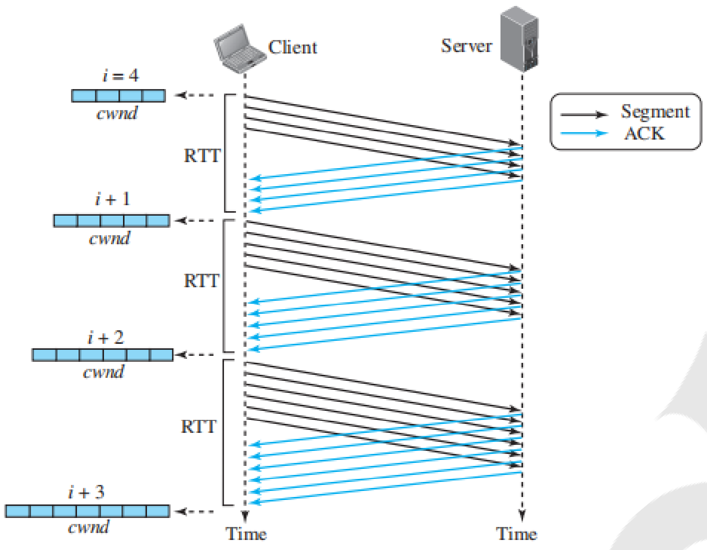
Start	$\rightarrow$	$cwnd = 1 \rightarrow 2^0$
After 1 RTT	$\rightarrow$	$cwnd = cwnd + 1 \;= 1 + 1 = 2 \rightarrow 2^1$
After 2 RTT	$\rightarrow$	$cwnd = cwnd + 2 \;= 2 + 2 = 4 \rightarrow 2^2$
After 3 RTT	$\rightarrow$	$cwnd = cwnd + 4 \;= 4 + 4 = 8 \rightarrow 2^3$

A slow start cannot continue indefinitely. There must be a threshold to stop this phase. The sender keeps track of a variable named *ssthresh* (slow-start threshold). When the size of the window in bytes reaches this threshold, slow start stops and the next phase starts.

Congestion Avoidance: Additive Increase

The slow-start algorithm in TCP increases the size of the congestion window (cwnd) exponentially, which can lead to congestion. To mitigate this, TCP employs the congestion avoidance algorithm, which increases cwnd additively rather than exponentially. When cwnd reaches the slow-start threshold (i), the slow-start phase ends, and the additive phase begins.

In this algorithm, for each set of acknowledged segments (the entire "window"), cwnd is incremented by one. A window represents the number of segments sent during a round-trip time (RTT).



For example, if the sender starts with cwnd = 4, it can send four segments. Upon receiving four ACKs, one segment slot opens up, increasing cwnd to 5. After sending five segments and receiving five acknowledgments, cwnd increases to 6, and so on.
The congestion window can be expressed as:

If an ACK arrives, *cwnd* = *cwnd* + (1/*cwnd*).

The window increases by (1/*cwnd*) portion of the Maximum Segment Size (MSS) in bytes. Thus, all segments in the previous window must be acknowledged to increase cwnd by 1 MSS byte. This results in a linear growth rate of cwnd with each round-trip time (RTT), making it a more conservative approach than slow-start.

Start	$\rightarrow$	$cwnd = \mathbf{i}$
After 1 RTT	$\rightarrow$	$cwnd = \mathbf{i + 1}$
After 2 RTT	$\rightarrow$	$cwnd = \mathbf{i + 2}$
After 3 RTT	$\rightarrow$	$cwnd = \mathbf{i + 3}$

Fast Recovery

The **fast-recovery** algorithm is optional in TCP. The old version of TCP did not use it, but the new versions try to use it. It starts when three duplicate ACKs arrive, which is interpreted as light congestion in the network. Like congestion avoidance, this algorithm is also an additive increase, but it increases the size of the congestion window when a duplicate ACK arrives (after the three duplicate ACKs that trigger the use of this algorithm). We can say

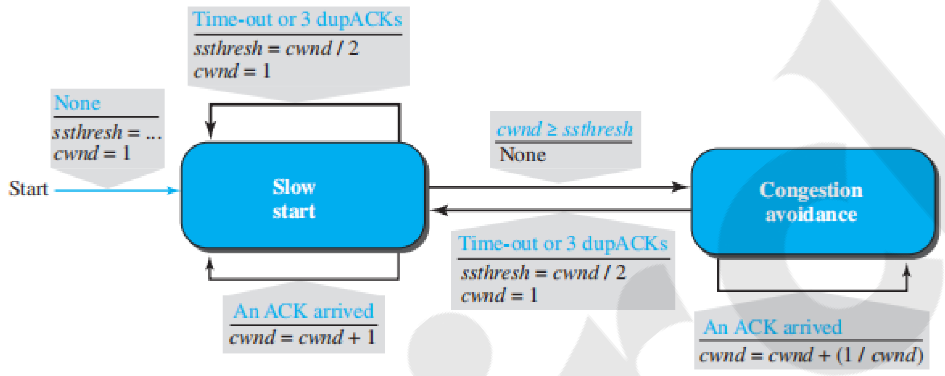
If a duplicate ACK arrives, *cwnd* == *cwnd* + (1 / *cwnd*)

Policy Transition

We discussed three congestion policies in TCP. Now the question is when each of these policies is used and when TCP moves from one policy to another. To answer these questions, we need to refer to three versions of TCP: Taho TCP, Reno TCP, and New Reno TCP.

Taho TCP

The early TCP, known as *Taho TCP*, used only two different algorithms in their congestion policy: *slow start* and *congestion avoidance*.



Taho TCP treats the two signs used for congestion detection, time-out and three duplicate ACKs, in the same way. In this version, when the connection is established, TCP starts the slow-start algorithm and sets the ssthresh variable to a pre-agreed value(normally a multiple of MSS) and the cwnd to 1 MSS.

If congestion is detected (occurrence of time-out or arrival of three duplicate ACKs), TCP immediately interrupts this aggressive growth and restarts a new slow start algorithm by limiting the threshold to half of the current *cwnd* and resetting the congestion window to 1

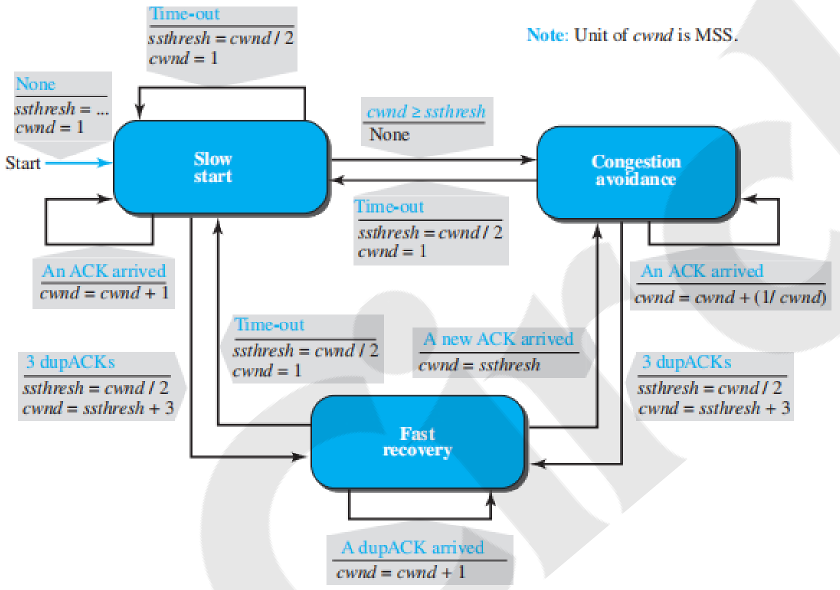
If no congestion is detected while reaching the threshold, TCP learns that the ceiling of its ambition is reached; it should not continue at this speed. It moves to the congestion avoidance state and continues in that state.

In the congestion-avoidance state, the size of the congestion window is increased by 1 each time a number of ACKs equal to the current size of the window has been received.

Reno TCP

A newer version of TCP, called *Reno TCP*, added a new state to the congestion-control FSM, called the fast-recovery state. This version treated the two signals of congestion, time-out and the arrival of three duplicate ACKs, differently. In this version, if a time-out occurs, TCP moves to the slow-start state (or starts a new round if it is already in this state); on the other hand, if three

duplicate ACKs arrive, TCP moves to the fast-recovery state and remains there as long as more duplicate ACKs arrive.
When TCP enters the fast-recovery state, three major events may occur. If duplicate ACKs continue to arrive, TCP stays in this state, but the *cwnd* grows exponentially. If a time-out occurs, TCP assumes that there is real congestion in the network and moves to the slow-start state. If a new (non duplicate) ACK arrives, TCP moves to the congestion-avoidance state, but deflates the size of the *cwnd* to the *ssthresh* value, as though the three duplicate ACKs have not occurred, and transition is from the slow-start state to the congestion-avoidance state.



NewReno TCP

A later version of TCP, called *NewReno TCP*, made an extra optimization on the Reno TCP. In this version, TCP checks to see if more than one segment is lost in the current window when three duplicate ACKs arrive. When TCP receives three duplicate ACKs, it retransmits the lost segment until a new ACK (not duplicate) arrives. If the new ACK defines the end of the window when the congestion was detected, TCP is certain that only one segment was lost. However, if the ACK number defines a position between the retransmitted segment and the end of the window, it is possible that the segment defined by the ACK is also lost. NewReno TCP retransmits this segment to avoid receiving more and more duplicate ACKs for it.

Additive Increase, Multiplicative Decrease

Out of the three versions of TCP, the Reno version is most common today. It has been observed that, in this version, most of the time the congestion is detected and taken care of by observing the three duplicate ACKs. Even if there are some time-out events, TCP recovers from them by aggressive exponential growth.